4 ways to bootstrap a Kubernetes cluster

Today I would like to tell you about bootstrapping your own Kubernetes cluster. The most striking (and perhaps the most challenging) thing about setting up K8s is that there are so many ways to do that. Kubernetes is a powerful tool to manage your microservices, and its architecture is also built with microservices in mind. It’s important to remember that Kubernetes consists of several components, some of which are core components, while others are optional.

What do you need to have a running k8s cluster?

First of all, what are the K8s components that are absolutely necessary to run a K8s cluster? There are seven of them.

Core components:

Control plane components—*these are the components we need to manage our cluster:*

1. kube-apiserver
2. kube-controller-manager
3. kube-scheduler
4. etcd

Runtime components—*these are the components that basically run our containers and make them available via network:*

1. kubelet
2. kube-proxy
3. container runtime

Kube-apiserver. This is the component that exposes very powerful and well-designed Kubernetes API
Kube-controller-manager monitors the state of the cluster (running

and crashed nodes, pods, services etc.) and makes sure that current cluster is in desired state (defined by user/administrator).
Kube-scheduler decides where to run created pods in our cluster. Its 'decisions' are based on resource requirements, affinity rules, etc.
Etcd is a key-value store where Kubernetes stores all cluster data.

All of these components are also called *master components* and should run in several replicas (at least 3) if you want your cluster to be highly available.

Kubelet is an agent that runs pods.
Kube-proxy manages forwarding rules (usually via iptables or ipvs) on the host and provides access to Service resources in Kubernetes.
Container runtime can be any CRI-compatible runtime: Docker, rkt, CRI-O. This component actually runs and manages containers in Linux.

Add-ons:

Aside from the core components, Kubernetes uses lots of powerful add-ons. Though you don't necessarily have to run them all to build a healthy cluster, they might be of some good use. First of all, it is networking:

1. kube-dns
2. CNI plugin

Kube-dns is just a DNS server that handles DNS records for Kubernetes services and provides service discovery.
CNI plugin. If you want to setup a network between your pods, there are (again!) several ways to do that. You can choose and setup any of CNI plugins described in official documentation.

Add-add-ons:

1. ingress-controller
2. cert-manager
3. helm
4. …

After you have all the core components, you can add as much add-ons to your cluster as you need. I won't go into details here, as there're lots of them and they are *quite* optional.

Choices

Basically, to run a Kubernetes cluster, we need 7 core components mentioned above. We can run them on different hosts, but for the sake of simplicity let's assume we have just one server and will run everything there.
So we just need to choose *where* and *how* to run them.

1. *Where* **Self-hosted VS. Systemd**. You can run those binaries either in a traditional Linux way using systemd OR you can run containers each having its own binary component.

2. *How secure* **TLS VS. no TLS (insecure)**. k8s components can talk to each other using either plain insecure connections or secure connections with TLS certificates.

3. *How do I do that* **Manually VS. using bootstrap tools**.

Let's do some math (just for fun ;). Even if we want to run only 7 core components, we can run every one of them (except for kubelet) either self-hosted or in systemd (`2*6` variants), using either secure or insecure etcd and apiserver (`2*2` variants), and all of that we can do either manually or using bootstrap tools (of which I personally know at least five).
Total number: (2 * 6) * (2 * 2) * (1 + 5) = 288 ways to setup a k8s cluster! And it is only core components. But of course we are not going to explore all the variations, just the basic ones. Let's start.

Let's get our hands dirty

Before we start exploring different options we need to meet basic requirements. These are:

1. CRI (in this guide I will use Docker-CE 18.03)

2. kubelet (I will use 1.9.6)
   I will use Ubuntu 16.04 throughout this guide but of course you can use any Linux distro you like, you just need to adjust some commands.

Option 1: good old Linux way

The first way to install k8s cluster is to download all binaries and run them as systemd services. I will not use TLS certificates here as I only want to show how to run binaries for k8s components.

First of all, you need to install etcd on your server. You can easily do it using this guide from OpenStack.

Then you need to get binaries. You can follow official Kubernetes documentation:

> *Download the latest binary release and unzip it. Server binary tarballs are no longer included in the Kubernetes final tarball, so you will need to locate and run* `./kubernetes/cluster/get-kube-binaries.sh` *to download the client and server binaries. Then locate* `./kubernetes/server/kubernetes-server-linux-amd64.tar.gz` *and unzip* that. *Then, within the second set of unzipped files, locate* `./kubernetes/server/bin`*, which contains all the necessary binaries.*

Basically you need to do this (for k8s 1.9.6):

```
cd ~
wget
https://github.com/kubernetes/kubernetes/releases/download/v
1.9.6/kubernetes.tar.gz
tar xvfz ./kubernetes.tar.gz
./kubernetes/cluster/get-kube-binaries.sh
tar xvfz ./kubernetes/server/kubernetes-server-linux-
amd64.tar.gz
mv ./kubernetes/server/bin/{kube-apiserver,kube-
scheduler,kube-controller-manager,kube-proxy}  /usr/bin/
mv ./kubernetes/client/bin/kubectl /usr/bin/
```

Next we are going to prepare systemd service configs. You can clone Kubernetes contrib repo and find examples in `./contrib/init/systemd/` directory:

```
cd ~
git clone https://github.com/kubernetes/contrib.git
cd ./contrib/init/systemd/
```

First, create user `kube` (we will run components from this user account):

```
useradd kube
```

Then move service files and environ files:

```
mv *.service /etc/systemd/system/
mv ./environ/* /etc/kubernetes/
```

Run kube-apiserver. Adjust options in `/etc/kubernetes/config` and `/etc/kubernetes/apiserver`. Important note: specify etcd address in KUBE_ETCD_SERVERS variable. Enable and run kube-apiserver:

```
systemctl enable kube-apiserver
systemctl start kube-apiserver
```

Repeat this step for kube-scheduler, kube-controller-manager and kube-proxy. Check that all processes are running. Let's check that api is working and we can list namespaces:

```
kubectl get ns
```

If we try to print nodes in our cluster (`kubectl get nodes`) it says `No resources found.` The reason is that we have only launched the control plane so far. A Node in Kubernetes is an instance that has a running `kubelet` binary and is able to run pods. So, let's make our server a Kubernetes node!
First, check that the CRI we are going to use is running:

```
docker info
```

Adjust options in `/etc/kubernetes/kubelet` config file. We need to point kubelet to apiserver. For that we add `--kubeconfig=/etc/kubernetes/kubelet-config.yaml` option and `--cgroup-driver=cgroupfs` option and comment `KUBELET_HOSTNAME`.
Then create this file with the following contents (adjust `<HOSTNAME>` to your environment):

```
apiVersion: v1
clusters:
- cluster:
    server: http://127.0.0.1:8080
  name: kubernetes-systemd
contexts:
- context:
    cluster: kubernetes-systemd
    user: system:node:<HOSTNAME>
  name: system:node:<HOSTNAME>@kubernetes-systemd
current-context: system:node:<HOSTNAME>@kubernetes-systemd
kind: Config
preferences: {}
users:
- name: system:node:<HOSTNAME>
```

Then enable and start kubelet:

```
systemctl enable kubelet
systemctl start kubelet
```

After that you will be able to list nodes in your cluster:

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

Congratulations! You have bootstrapped your k8s cluster using systemd! It does not contain CNI plugin for networking but I'm going to show you how to do it later in Option 4.

Option 2: self-hosted

This option is very similar to the previous one but instead of downloading binaries and running them in systemd, we will use container images with those components. Self-hosted means that we only launch `kubelet` binary in systemd. When it is running, it searches for manifests in `/etc/kubernetes/manifests` directory (where we will put the rest of components) and launches them in containers.
First add the following flag to kubelet config: `--pod-manifest-path=/etc/kubernetes/manifests` . Then create manifests directory:

```
mkdir /etc/kubernetes/manifests
```

Let's prepare our manifests. This is how a manifest for the kube-apiserver looks like:

```
cat << API > /etc/kubernetes/manifests/kube-apiserver.yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    scheduler.alpha.kubernetes.io/critical-pod: ""
  labels:
    component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
spec:
  hostNetwork: true
  containers:
  - command:
    - kube-apiserver
    - --insecure-bind-address=127.0.0.1
    - --etcd-servers=http://127.0.0.1:2379
    - --service-cluster-ip-range=10.96.0.0/12
    - --admission-
control=NamespaceLifecycle,LimitRanger,SecurityContextDeny,S
erviceAccount,ResourceQuota
    image: gcr.io/google_containers/kube-apiserver-
amd64:v1.9.6
    name: kube-apiserver
API
```

You can see that we use image `gcr.io/google_containers/kube-apiserver-amd64:v1.9.6` and just pass some flags inside the container. So, let's create the rest of the manifests.

Etcd (we use volume to make data consistent):

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    scheduler.alpha.kubernetes.io/critical-pod: ""
  labels:
    component: etcd
    tier: control-plane
  name: etcd
  namespace: kube-system
spec:
  containers:
  - command:
    - etcd
    - --data-dir=/var/lib/etcd
    - --listen-client-urls=http://127.0.0.1:2379
    - --advertise-client-urls=http://127.0.0.1:2379
    image: gcr.io/google_containers/etcd-amd64:3.1.11
    name: etcd
    volumeMounts:
    - mountPath: /var/lib/etcd
      name: etcd
  hostNetwork: true
  volumes:
  - hostPath:
      path: /var/lib/etcd
      type: DirectoryOrCreate
    name: etcd
```

Kube-controller-manager:

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    scheduler.alpha.kubernetes.io/critical-pod: ""
  labels:
    component: kube-controller-manager
    tier: control-plane
  name: kube-controller-manager
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-controller-manager
    - --master=http://127.0.0.1:8080
    image: gcr.io/google_containers/kube-controller-manager-
amd64:v1.9.6
    name: kube-controller-manager
  hostNetwork: true
```

And kube-scheduler:

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    scheduler.alpha.kubernetes.io/critical-pod: ""
  labels:
    component: kube-scheduler
    tier: control-plane
  name: kube-scheduler
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-scheduler
    - --master=http://127.0.0.1:8080
    image: gcr.io/google_containers/kube-scheduler-
```

```
amd64:v1.9.6
    name: kube-scheduler
  hostNetwork: true
```

Restart kubelet and then you will see (using `docker ps`) that it launched all the manifests. Let’s check them using kubectl:

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

You should see that all four pods are running. And if we list nodes we should see our server again:

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

That’s it! We have launched a self-hosted k8s cluster!

Option 3: the hard way

If you want to setup all the components and provision all the certificates manually, you can follow the tutorial Bootstrap Kubernetes the hard way.

Option 4: easy & securely

My favorite way to setup a Kubernetes cluster is to use kubeadm. It is a simple and powerful tool: you can specify just several options you need and run it.

Step 1/2. Kubeadm init

The simplest way is to run:

`kubeadm init`

and it will do all the magic. Namely, it creates certificates and places them in the `/etc/kubernetes/pki` directory; it creates manifests and configuration files for `kube-apiserver`, `kube-controller` and `kube-scheduler` (and for `etcd` if you did not specify an external etcd cluster). Then it runs kubelet and when the control plane components are ready, it runs `kube-proxy` Daemonset and `kube-dns` Deployment.

But if you want to specify some options (and I recommend you to do that) you should use kubeadm config file. A simple version of it looks as follows (see all options here):

```
apiVersion: kubeadm.k8s.io/v1alpha1
kind: MasterConfiguration
networking:
  serviceSubnet: 10.96.0.0/12
  podSubnet: 10.224.0.0/16
etcd:
  endpoints: # Use this to specify etcd servers
  - https://10.0.0.1:2379
  - https://10.0.0.2:2379
  - https://10.0.0.3:2379
  caFile: /etc/kubernetes/etcd/ca.pem
  certFile: /etc/kubernetes/etcd/etcd.pem
  keyFile: /etc/kubernetes/etcd/etcd-key.pem
apiServerCertSANs: # Use this to specify IP SANs for
certificates
  - 172.16.0.1
  - 172.16.0.2
  - 172.16.0.3
  - 127.0.0.1
api:
  advertiseAddress: 172.16.0.1
```

And then simply run:

```
kubeadm init --config config.yaml
```

and wait until it does all the work. After that it creates file `/etc/kubernetes/admin.conf` which you can use for authentication with the API. To make your life easier add the following line at the end of your `.bashrc` file:

```
export KUBECONFIG=/etc/kubernetes/admin.conf
source <(kubectl completion bash)
```

And you will be able to use kubectl. That’s it!

Step 2/2. Add overlay network

Now, you might want to launch web-applications inside your cluster, but there is no use in it if there is no network. The simplest yet powerful way is to use overlay network. There are many options for that. For example, you can install calico plugin. Here is the guide how to do it for our k8s installation.

So, it was a brief ;) overview of k8s bootstrapping, I hope you find it useful. I would love to receive your comments and questions. Also follow us on Twitter and join our Telegram chat to stay tuned! Cheers!