March 18, 2017 · KUBERNETES

Self-hosted Kubernetes on bare-metal with Bootkube/Matchbox

10/22/2017: Updated post on this method

I use a cobbler VirtualBox VM on my laptop to PXE boot my three bare-metal servers in my home lab for OpenStack. This enables me to quickly test new OpenStack deployments with setting three "--netboot" cobbler values to true and then rebooting my servers. Cobbler takes care of PXE booting my servers with Ubuntu and with my specific partitioning scheme. I can then use Ansible to prepare my three nodes and then use Ansible to lay down OpenStack.

I was looking for a similar solution for testing Kubernetes when a friend at Rackspace pointed me to bootkube and matchbox. I've used a few methods for deploying a K8s cluster, a manually using Kelsey Hightowers **'Kubernetes the Hard Way'**, **minikube** for local use, and recently **kubeadm**, which was released with K8s 1.5. kubeadm makes setting up a test cluster extremely simple and if you're just wanting to quickly use Kubernetes in a multi-node setup, I would recommend kubeadm. I believe there will be some HA capabilities added to kubeadm in K8s 1.6 that I'm looking forward to trying out.

With bootkube and matchbox I was able to get a similar setup to cobbler where I can PXE boot my three bare-metal servers with **Container Linux** and then bootkube will bootstrap a self-hosted Kubernetes cluster. Matchbox uses groups/profiles and ignition configs. **This documentation** does a good job of describing Matchbox. It took me awhile to understand the general flow of this process.

Matchbox API -> Groups -> Profiles -> Ignition Configs

Instead of even attempting to explain what self-hosted means, I'll defer to **CoreOS** CTO Brandon Philips who wrote a great overview of self-hosted Kubernetes and how bootkube is attempting to solve this workflow.

https://github.com/kubernetes/community/blob/master/contributors/design-proposals/self-hosted-kubernetes.md

As you can see, deploying a self-hosted cluster enables some interesting cluster management abilities like managing the Kubernetes control plane just like any other application managed by Kubernetes.

Setup

Let's take a look at the matchbox setup. I spun up a VirtualBox VM on my laptop with Ubuntu 16.04 to use as my matchbox/bootkube/dnsmasq host. We'll need to create dnsmasq and matchbox containers. I did this setup with rkt instead of docker, but they have **documentation** for both implementations.

Install rkt

```
# git clone https://github.com/coreos/matchbox.git
```

This will download the Container Linux images locally so when it's served, it doesn't have to download those over the Internet.

```
# cd matchbox
# ./scripts/get-coreos stable 1235.9.0 ./examples/assets
```

Take note of the examples directory. It has some preconfigured examples for various things bootkube/matchbox can configure. I'm going to focus on one, **bootkube-install**. This group will install Container Linux to your servers and then bootstrap a self-hosted Kubernetes cluster.

```
root@bootkube:~/matchbox# ls examples/groups/bootkube-install
install.json  node1.json  node2.json  node3.json

root@bootkube:~/matchbox# cat examples/groups/bootkube-install/install.json
{
  "id": "coreos-install",
  "name": "CoreOS Install",
  "profile": "install-reboot",
  "metadata": {
    "coreos_channel": "stable",
    "coreos_version": "1235.9.0",
    "ignition_endpoint": "http://bootkube:8080/ignition",
    "baseurl": "http://bootkube:8080/assets/coreos"
  }
}
```

The bootkube hostname resolves to my VirtualBox VM running matchbox/bootkube/dnsmasq. This group will install Container Linux on the servers local disk from the 'install-reboot' profile. The node1-3 files are where we enter our MAC address for the bare metal servers and any SSH keys we want installed.

```
root@bootkube:~/matchbox# cat examples/groups/bootkube-install/node1.json
{
  "id": "node1",
  "name": "Controller Node",
  "profile": "bootkube-controller",
  "selector": {
    "mac": "44:39:C4:53:4C:39",
    "os": "installed"
  },
  "metadata": {
    "domain_name": "node1.example.com",
    "etcd_initial_cluster": "node1=http://node1:2380",
    "etcd_name": "node1",
    "k8s_dns_service_ip": "10.3.0.10",
    "ssh_authorized_keys": [
      "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDgfTTmM5K1IK9LkMzykO5/9dVe30AWO37fecibe
    ]
  }
}
```

node1 will act as the etcd/controller node. node2 is only a worker node.

```
root@bootkube:~/matchbox# cat examples/groups/bootkube-install/node2.json
{
  "id": "node2",
  "name": "Worker Node",
  "profile": "bootkube-worker",
  "selector": {
    "mac": "D4:AE:52:C8:A1:8D",
    "os": "installed"
  },
  "metadata": {
    "domain_name": "node2.example.com",
    "etcd_endpoints": "node1:2379",
    "k8s_dns_service_ip": "10.3.0.10",
    "ssh_authorized_keys": [
      "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDgfTTmM5K1IK9LkMzykO5/9dVe30AWO37fecibe
    ]
  }
}
```

Now let's take a look at the profiles. The first profile it uses is install-reboot.

```
root@bootkube:~/matchbox# ls examples/profiles
bootkube-controller.json  etcd3-gateway.json  grub.json        install-shutdown.json

root@bootkube:~/matchbox# cat examples/profiles/install-reboot.json
{
  "id": "install-reboot",
  "name": "Install CoreOS and Reboot",
  "boot": {
    "kernel": "/assets/coreos/1235.9.0/coreos_production_pxe.vmlinuz",
    "initrd": ["/assets/coreos/1235.9.0/coreos_production_pxe_image.cpio.gz"],
    "args": [
      "coreos.config.url=http://bootkube:8080/ignition?uuid=${uuid}&mac=${mac:hexhyp
      "coreos.first_boot=yes",
      "console=tty0",
      "console=ttyS0",
      "coreos.autologin"
    ]
  },
  "ignition_id": "install-reboot.yaml"
}
```

The last line calls the ignition file that will be used.

```
root@bootkube:~/matchbox# cat examples/ignition/install-reboot.yaml
---
```

```
systemd:
  units:
    - name: installer.service
      enable: true
      contents: |
        [Unit]
        Requires=network-online.target
        After=network-online.target
        [Service]
        Type=simple
        ExecStart=/opt/installer
        [Install]
        WantedBy=multi-user.target
storage:
  files:
    - path: /opt/installer
      filesystem: root
      mode: 0500
      contents:
        inline: |
          #!/bin/bash -ex
          curl "{{.ignition_endpoint}}?{{.request.raw_query}}&os=installed" -o ignit
          coreos-install -d /dev/sda -C {{.coreos_channel}} -V {{.coreos_version}} -
          udevadm settle
          systemctl reboot

{{ if index . "ssh_authorized_keys" }}
passwd:
  users:
    - name: core
      ssh_authorized_keys:
        {{ range $element := .ssh_authorized_keys }}
        - {{$element}}
        {{end}}
{{end}}
```

This is where it lays down the systemd unit files and other system tasks, like installing Container Linux to a specific device, "-d /dev/sda". You can customize or create new ignition files for a specific chassis, etc. Some docs can be found **here**.

node1 then matches the bootkube-controller profile.

```
root@bootkube:~/matchbox# cat examples/profiles/bootkube-controller.json
{
  "id": "bootkube-controller",
  "name": "bootkube Ready Controller",
  "boot": {
    "kernel": "/assets/coreos/1235.9.0/coreos_production_pxe.vmlinuz",
    "initrd": ["/assets/coreos/1235.9.0/coreos_production_pxe_image.cpio.gz"],
    "args": [
      "root=/dev/sda1",
      "coreos.config.url=http://bootkube:8080/ignition?uuid=${uuid}&mac=${mac:hexhyp
      "coreos.first_boot=yes",
      "console=tty0",
      "console=ttyS0",
      "coreos.autologin"
    ]
  },
  "ignition_id": "bootkube-controller.yaml"
}
```

The ignition file for the bootkube-controller.

```
root@bootkube:~/matchbox# cat examples/ignition/bootkube-controller.yaml
---
systemd:
  units:
    - name: etcd-member.service
      enable: true
      dropins:
        - name: 40-etcd-cluster.conf
          contents: |
            [Service]
            Environment="ETCD_IMAGE_TAG=v3.1.0"
            Environment="ETCD_NAME={{.etcd_name}}"
            Environment="ETCD_ADVERTISE_CLIENT_URLS=http://{{.domain_name}}:2379"
            Environment="ETCD_INITIAL_ADVERTISE_PEER_URLS=http://{{.domain_name}}:23
            Environment="ETCD_LISTEN_CLIENT_URLS=http://0.0.0.0:2379"
            Environment="ETCD_LISTEN_PEER_URLS=http://0.0.0.0:2380"
            Environment="ETCD_INITIAL_CLUSTER={{.etcd_initial_cluster}}"
            Environment="ETCD_STRICT_RECONFIG_CHECK=true"
    - name: docker.service
      enable: true
    - name: locksmithd.service
      dropins:
        - name: 40-etcd-lock.conf
          contents: |
            [Service]
            Environment="REBOOT_STRATEGY=etcd-lock"
    - name: kubelet.path
      enable: true
      contents: |
        [Unit]
        Description=Watch for kubeconfig
        [Path]
        PathExists=/etc/kubernetes/kubeconfig
        [Install]
        WantedBy=multi-user.target
    - name: wait-for-dns.service
      enable: true
      contents: |
        [Unit]
        Description=Wait for DNS entries
        Wants=systemd-resolved.service
        Before=kubelet.service
        [Service]
        Type=oneshot
        RemainAfterExit=true
        ExecStart=/bin/sh -c 'while ! /usr/bin/grep '^[^#[:space:]]' /etc/resolv.con
        [Install]
        RequiredBy=kubelet.service
    - name: kubelet.service
      contents: |
        [Unit]
        Description=Kubelet via Hyperkube ACI
        [Service]
        Environment="RKT_OPTS=--uuid-file-save=/var/run/kubelet-pod.uuid \
          --volume=resolv,kind=host,source=/etc/resolv.conf \
          --mount volume=resolv,target=/etc/resolv.conf \
          --volume var-lib-cni,kind=host,source=/var/lib/cni \
          --mount volume=var-lib-cni,target=/var/lib/cni \
          --volume var-log,kind=host,source=/var/log \
          --mount volume=var-log,target=/var/log"
        EnvironmentFile=/etc/kubernetes/kubelet.env
        ExecStartPre=/bin/mkdir -p /etc/kubernetes/manifests
        ExecStartPre=/bin/mkdir -p /srv/kubernetes/manifests
        ExecStartPre=/bin/mkdir -p /etc/kubernetes/checkpoint-secrets
        ExecStartPre=/bin/mkdir -p /etc/kubernetes/cni/net.d
        ExecStartPre=/bin/mkdir -p /var/lib/cni
        ExecStartPre=-/usr/bin/rkt rm --uuid-file=/var/run/kubelet-pod.uuid
        ExecStart=/usr/lib/coreos/kubelet-wrapper \
          --kubeconfig=/etc/kubernetes/kubeconfig \
          --require-kubeconfig \
          --cni-conf-dir=/etc/kubernetes/cni/net.d \
          --network-plugin=cni \
          --lock-file=/var/run/lock/kubelet.lock \
          --exit-on-lock-contention \
          --pod-manifest-path=/etc/kubernetes/manifests \
          --allow-privileged \
          --hostname-override={{.domain_name}} \
          --node-labels=master=true \
          --cluster_dns={{.k8s_dns_service_ip}} \
          --cluster_domain=cluster.local
        ExecStop=-/usr/bin/rkt stop --uuid-file=/var/run/kubelet-pod.uuid
        Restart=always
        RestartSec=10
        [Install]
        WantedBy=multi-user.target
    - name: bootkube.service
      contents: |
        [Unit]
        Description=Bootstrap a Kubernetes control plane with a temp api-server
        [Service]
        Type=simple
        WorkingDirectory=/opt/bootkube
        ExecStart=/opt/bootkube/bootkube-start
storage:
  {{ if index . "pxe" }}
  disks:
    - device: /dev/sda
      wipe_table: true
      partitions:
        - label: ROOT
  filesystems:
    - name: root
      mount:
        device: "/dev/sda1"
        format: "ext4"
        create:
          force: true
          options:
            - "-LROOT"
  {{end}}
  files:
    - path: /etc/kubernetes/kubelet.env
      filesystem: root
      mode: 0644
      contents:
        inline: |
          KUBELET_ACI=quay.io/coreos/hyperkube
          KUBELET_VERSION=v1.5.2_coreos.2
    - path: /etc/hostname
      filesystem: root
      mode: 0644
      contents:
        inline:
          {{.domain_name}}
    - path: /etc/sysctl.d/max-user-watches.conf
      filesystem: root
      contents:
        inline: |
          fs.inotify.max_user_watches=16184
    - path: /opt/bootkube/bootkube-start
      filesystem: root
      mode: 0544
      user:
        id: 500
      group:
        id: 500
      contents:
        inline: |
          #!/bin/bash
          # Wrapper for bootkube start
          set -e
          BOOTKUBE_ACI="${BOOTKUBE_ACI:-quay.io/coreos/bootkube}"
          BOOTKUBE_VERSION="${BOOTKUBE_VERSION:-v0.3.7}"
          BOOTKUBE_ASSETS="${BOOTKUBE_ASSETS:-/opt/bootkube/assets}"
          exec /usr/bin/rkt run \
            --trust-keys-from-https \
            --volume assets,kind=host,source=$BOOTKUBE_ASSETS \
            --mount volume=assets,target=/assets \
            $RKT_OPTS \
            ${BOOTKUBE_ACI}:${BOOTKUBE_VERSION} --net=host --exec=/bootkube -- start

{{ if index . "ssh_authorized_keys" }}
passwd:
  users:
    - name: core
      ssh_authorized_keys:
        {{ range $element := .ssh_authorized_keys }}
        - {{$element}}
        {{end}}
{{end}}
```

It lays down some more systemd unit files and other system things. It sets up the service that monitors for "/etc/kubernetes/kubeconfig", which will then kick off kubelet via hyperkube. It also creates /opt/bootkube/bootkube-start, which is a wrapper for the temporary bootkube rkt container.

Install bootkube on the VirtualBox VM so we can generate all the asset files our hosts require.

```
root@bootkube:~/matchbox# wget https://github.com/kubernetes-incubator/bootkube/relea
root@bootkube:~/matchbox# tar xzf bootkube.tar.gz
root@bootkube:~/matchbox# ./bin/linux/bootkube version
Version: v0.3.9

root@bootkube:~/matchbox# ./bin/linux/bootkube render --asset-dir=assets --api-server
```

That should generate files in assets/

```
root@bootkube:~/matchbox# ls assets/
auth  manifests  tls
```

The manifests files are what are used to build the self-hosted Kubernetes control plane.

```
root@bootkube:~/matchbox# ls assets/manifests/
kube-apiserver-secret.yaml  kube-controller-manager-disruption.yaml  kube-controller
```

Now we're ready to start the matchbox/dnsmasq containers. These are values that worked for my setup, you might have to change these up a bit. If you wanted to create a new group/profile/ignition directory structure, you can do that and mount your own files in these commands.

```
root@bootkube:~/matchbox# rkt run coreos.com/dnsmasq:v0.3.0 --net=host -- -d -q --dh

root@bootkube:~/matchbox# rkt run --net=host --mount volume=data,target=/var/lib/mat
```

When starting the matchbox container, we specify the "source=$PWD/examples/groups/bootkube-install" location for the files we edited.

At this point you should be able to reboot your servers, they will install CoreOS, then reboot and then run the bootkube-controller and bootkube-worker profiles, which run the ignition files for those profiles.

Kubernetes cluster setup

The first action your controller node will attempt is to start up an etcd cluster.

```
node1 ~ # systemctl status etcd-member
● etcd-member.service - etcd (System Application Container)
   Loaded: loaded (/usr/lib/systemd/system/etcd-member.service; enabled; vendor pres
  Drop-In: /etc/systemd/system/etcd-member.service.d
           └─40-etcd-cluster.conf
   Active: active (running) since Thu 2017-03-16 04:35:13 UTC; 5min ago
     Docs: https://github.com/coreos/etcd
  Process: 7051 ExecStartPre=/usr/bin/rkt rm --uuid-file=/var/lib/coreos/etcd-member
  Process: 7024 ExecStartPre=/usr/bin/mkdir --parents /var/lib/coreos (code=exited, 
 Main PID: 7090 (etcd)
    Tasks: 10
   Memory: 27.5M
      CPU: 1.298s
   CGroup: /system.slice/etcd-member.service
           └─7090 /usr/local/bin/etcd

Mar 16 04:35:13 node1 etcd-wrapper[7090]: 2017-03-16 04:35:13.554742 I | raft: a9aee
Mar 16 04:35:13 node1 etcd-wrapper[7090]: 2017-03-16 04:35:13.554752 I | raft: a9aee
Mar 16 04:35:13 node1 etcd-wrapper[7090]: 2017-03-16 04:35:13.554757 I | raft: raft.
Mar 16 04:35:13 node1 etcd-wrapper[7090]: 2017-03-16 04:35:13.554919 I | etcdserver:
Mar 16 04:35:13 node1 etcd-wrapper[7090]: 2017-03-16 04:35:13.554945 I | etcdserver:
Mar 16 04:35:13 node1 etcd-wrapper[7090]: 2017-03-16 04:35:13.554953 I | embed: read
Mar 16 04:35:13 node1 etcd-wrapper[7090]: 2017-03-16 04:35:13.555170 N | embed: serv
Mar 16 04:35:13 node1 systemd[1]: Started etcd (System Application Container).
Mar 16 04:35:13 node1 etcd-wrapper[7090]: 2017-03-16 04:35:13.556941 N | etcdserver/
Mar 16 04:35:13 node1 etcd-wrapper[7090]: 2017-03-16 04:35:13.556969 I | etcdserver/
```

So now we have a working etcd cluster for Kubernetes.

```
node1 ~ # etcdctl cluster-health
member a9aee06e6a14d468 is healthy: got healthy result from http://node1:2379
cluster is healthy
```

Secure copy the kubeconfig to /etc/kubernetes/kubeconfig on every node which will activate kubelet.service. The /etc/systemd/system/kubelet.service will start kubelet via hyperkube ACI.

```
root@bootkube:~/matchbox# scp assets/auth/kubeconfig core@node1:/home/core/kubeconfi
root@bootkube:~/matchbox# ssh core@node1 'sudo mv kubeconfig /etc/kubernetes/kubecon
```

Now we move the asset files over and start up bootkube, which will setup our temporary API and bootstrap a self-hosted Kubernetes cluster.

```
root@bootkube:~/matchbox# scp -r assets core@node1:/home/core
root@bootkube:~/matchbox# ssh core@node1 'sudo mv assets /opt/bootkube/assets && sud
```

You can view this by tailing the bootkube services log. Once it's done, you should see the following.

```
node1 ~ # journalctl -f -u bootkube

...

Mar 17 21:59:36 node1 bootkube-start[8093]: [149795.171672] bootkube[5]: All self-ho
```

Here's rkt and docker output. As you can see, the temporary bootkube container has already stopped.

```
node1 ~ # rkt list
UUID        APP     IMAGE NAME                      STATE   CREATED     STARTED     NETW
615573b4    etcd        quay.io/coreos/etcd:v3.1.0          running 1 day ago   1 da
75cb7ba5    hyperkube   quay.io/coreos/hyperkube:v1.5.2_coreos.2    running 44 minut
a6848aa8    bootkube    quay.io/coreos/bootkube:v0.3.7          exited  8 minutes ag

node1 ~ # docker ps
CONTAINER ID        IMAGE
8aec69372861        quay.io/coreos/hyperkube:v1.5.2_coreos.2
285099a245ec        quay.io/coreos/hyperkube:v1.5.2_coreos.2
1027ddd38138        quay.io/coreos/hyperkube:v1.5.2_coreos.2
76b515b653ed        quay.io/coreos/hyperkube:v1.5.2_coreos.2
54cb191700fc        quay.io/coreos/hyperkube:v1.5.2_coreos.2
8f1ab95709c5        gcr.io/google_containers/pause-amd64:3.0
```

```
37c55a435736        gcr.io/google_containers/pause-amd64:3.0
6351298ade64        gcr.io/google_containers/pause-amd64:3.0
fa1098928dda        gcr.io/google_containers/pause-amd64:3.0
506ef2593ecb        busybox
e589d8122961        quay.io/coreos/pod-checkpointer:5b585a2d731173713fa6871c436f6c53
c30794abbc0a        quay.io/coreos/flannel:v0.7.0-amd64
5ca78d87f35c        gcr.io/google_containers/pause-amd64:3.0
b129fa7d747d        quay.io/coreos/pod-checkpointer:5b585a2d731173713fa6871c436f6c53
4b341ce04e34        quay.io/coreos/hyperkube:v1.5.2_coreos.2
04a038ef35fe        gcr.io/google_containers/pause-amd64:3.0
7473cfa44f71        gcr.io/google_containers/pause-amd64:3.0
febfefa18eee        gcr.io/google_containers/pause-amd64:3.0
0d7d3db10141        gcr.io/google_containers/pause-amd64:3.0
```

For your worker nodes, you only need to copy over the kubeconfig file and it will activate the services needed and will automatically join the Kubernetes cluster, assuming your kubeconfig is correct.

Install the kubectl binary on your machine and copy over the kubeconfig file from assets/auth/kubeconfig. Then on your local machine, set the the KUBECONFIG variable to your kubeconfig file and check out your Kubernetes cluster!

```
export KUBECONFIG=kubeconfig

[17:09]shane@work~$ kubectl cluster-info
Kubernetes master is running at https://node1:443
KubeDNS is running at https://node1:443/api/v1/proxy/namespaces/kube-system/services

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

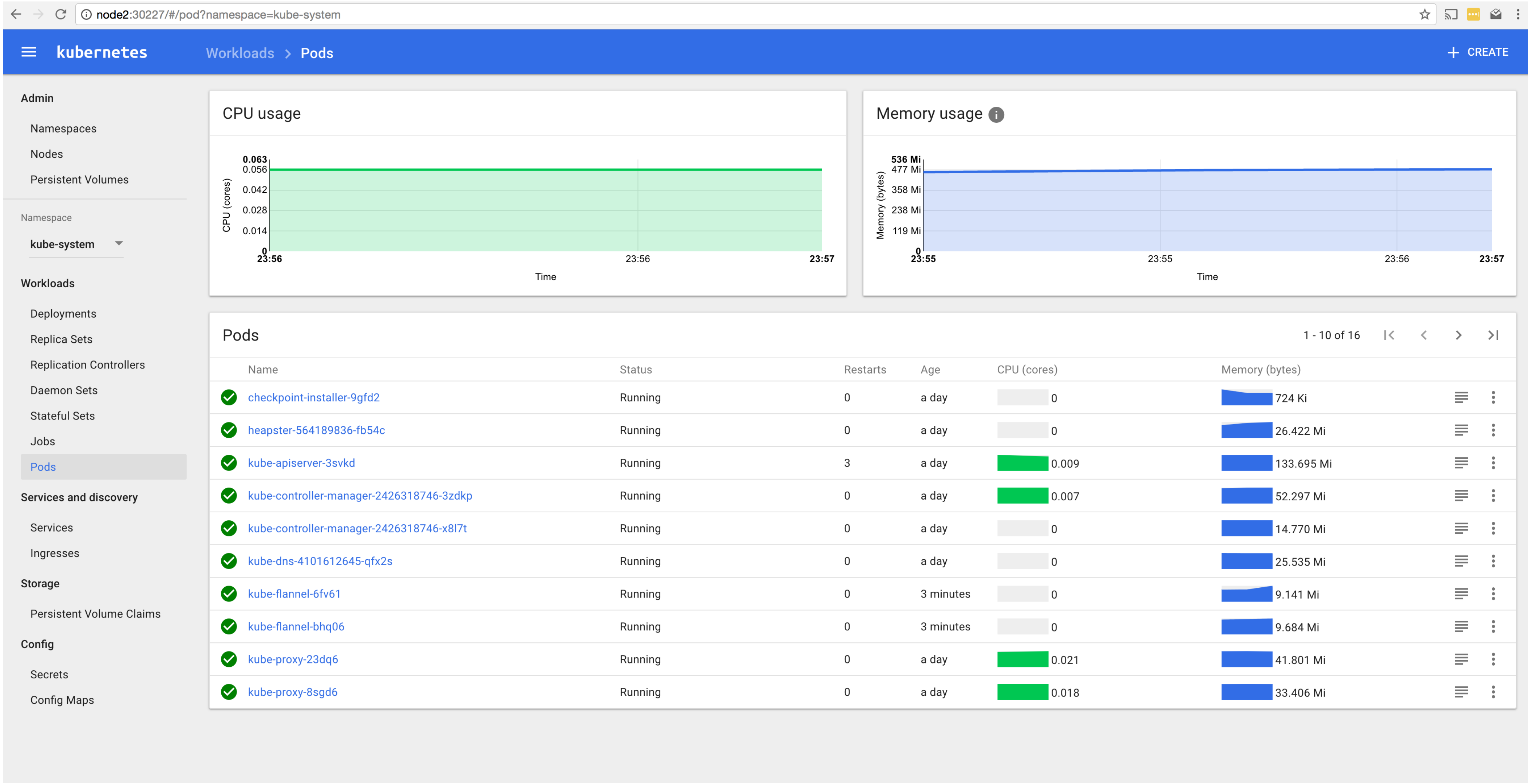
Woot!

```
[17:12]shane@work~$ kubectl get nodes
NAME      STATUS    AGE
node1     Ready     14m
node2     Ready     14m

[17:12]shane@work~$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                       READY     STATUS    RESTARTS
kube-system   checkpoint-installer-fpdvj                 1/1       Running   0
kube-system   kube-apiserver-gtjmv                       1/1       Running   2
kube-system   kube-controller-manager-2426318746-bbn1k   1/1       Running   0
kube-system   kube-controller-manager-2426318746-v84jr   1/1       Running   0
kube-system   kube-dns-4101612645-j745c                  4/4       Running   0
kube-system   kube-flannel-8vhwn                         2/2       Running   0
kube-system   kube-flannel-pzv0l                         2/2       Running   1
kube-system   kube-proxy-w3qqh                           1/1       Running   0
kube-system   kube-proxy-zjxq1                           1/1       Running   0
kube-system   kube-scheduler-2947727816-3z5mq            1/1       Running   0
```

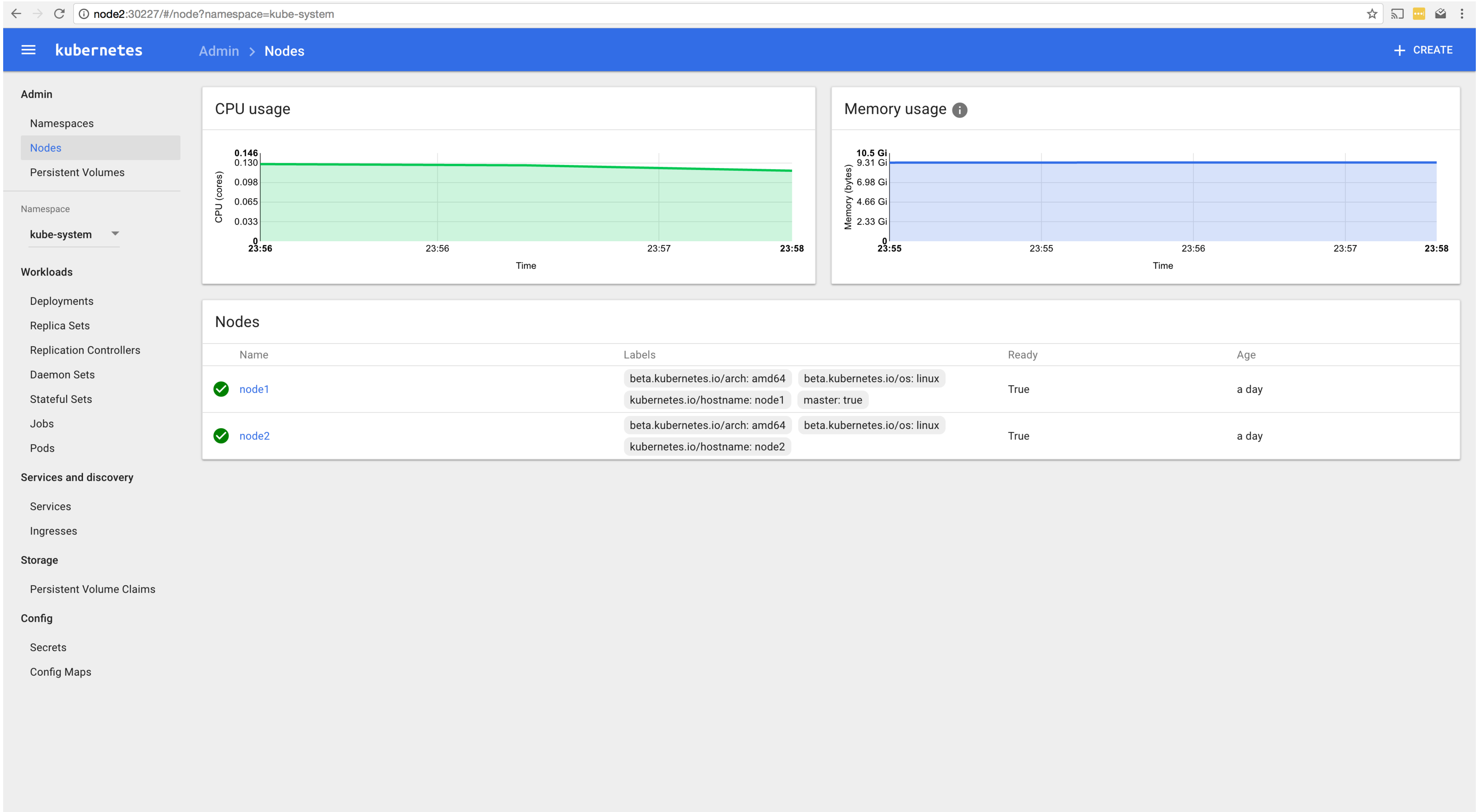
```
kube-system   kube-scheduler-2947727816-j2lzl           1/1       Running   0
kube-system   pod-checkpointer-node1                    1/1       Running   0
```

Here are some screenshots from the Kubernetes dashboard from a two node setup.



Pods

Name	Status	Restarts	Age	CPU (cores)	Memory (bytes)
✓ checkpoint-installer-9gfd2	Running	0	a day	0	724 Ki
✓ heapster-564189836-fb54c	Running	0	a day	0	26.422 Mi
✓ kube-apiserver-3svkd	Running	3	a day	0.009	133.695 Mi
✓ kube-controller-manager-2426318746-3zdkp	Running	0	a day	0.007	52.297 Mi
✓ kube-controller-manager-2426318746-x8l7t	Running	0	a day	0	14.770 Mi
✓ kube-dns-4101612645-qfx2s	Running	0	a day	0	25.535 Mi
✓ kube-flannel-6fv61	Running	0	3 minutes	0	9.141 Mi
✓ kube-flannel-bhq06	Running	0	3 minutes	0	9.684 Mi
✓ kube-proxy-23dq6	Running	0	a day	0.021	41.801 Mi
✓ kube-proxy-8sgd6	Running	0	a day	0.018	33.406 Mi



Nodes

Name	Labels	Ready	Age
✓ node1	beta.kubernetes.io/arch: amd64 beta.kubernetes.io/os: linux kubernetes.io/hostname: node1 master: true	True	a day
✓ node2	beta.kubernetes.io/arch: amd64 beta.kubernetes.io/os: linux kubernetes.io/hostname: node2	True	a day

Not knowing about bootkube and matchbox two weeks ago I'm really impressed with the work that's gone into the projects. It makes the full process of describing your environment in YAML/JSON files (something I like from openstack-ansible) to PXE booting and bootstrapping bare metal servers into a Kubernetes cluster impressively simple. All of these manual steps can easily be converted to an Ansible playbook or any other configuration management software. The projects are moving quickly and some of this workflow will probably change. I'm looking forward to where these projects go, making larger scale, bare-metal/self-hosted clusters as simple as possible. Some additional information can be found with the following links.

- **Bootkube**

- **Matchbox**

- **Self-Hosted Kubernetes Docs**

EMAIL FACEBOOK TWITTER LINKEDIN TUMBLR REDDIT GOOGLE+

5 Comments **Shane Cunningham** 1 **Login**

♡ **Recommend** 1 Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Matt Johnson • 2 years ago

Hey. Great article, did you have to manually place certs into /etc/ssl/certs for ETCD to function? etcd is having issues starting (both via your instructions, the coreos terraform instructions and others). None of the instructions seem to work as stated. Thanks.

• Reply • Share ›

dghubble ➦ Matt Johnson • a year ago

Would you mind filing an issue?

• Reply • Share ›

cunninghamshane Mod ➦ Matt Johnson • 2 years ago

Hey Matt, I did not have to do anything with TLS certs. There might have been some changes since I posted that it now requires the certs. I plan on re-writing this with the latest matchbox/bootkube and maybe throwing tectonic in there also.

• Reply • Share ›

dghubble ➦ cunninghamshane • a year ago

Yeah, the newer cluster examples now always use TLS to secure etcd. The project docs let you copy these to nodes on your own (similar to scp of the kubeconfig) or use the Terraform module that declares and automates this cluster creation process.
https://github.com/coreos/m...

Cheers, nice article!

• Reply • Share ›

yazpik • 2 years ago

Great post !

• Reply • Share ›

ALSO ON **SHANE CUNNINGHAM**

pfSense, Squid and logging full URLs

1 comment • 5 years ago

Андрей Михайлович — sorry for my english :)Hello! Thanx for solution, But in reports in LightSquid i see short url anyway :(How to see full URLs in LightSquid reports ?

XenServer 6.2 and PCI passthrough for LSI SAS1068E

21 comments • 5 years ago

cunninghamshane — Hi emil,I know this response is late but it seems the solution I posted has had a few issues with Windows/Windows Server VMs. I have not tested this with a Windows VMs. If you get this working in

Monitor OpenIndiana/Solaris 11 with check_mk_agent and Nagios

6 comments • 5 years ago

shane — Nice! Good to hear!

Deploying OpenStack Kilo with openstack-ansible

4 comments • 3 years ago

cunninghamshane — Glad it helped, Purshottam! You are correct, p4p1 will have all tenant traffic and will need a way to reach the Internet.