# Chapter 14 - Kubernetes and Ansible

Most real-world applications require a lot more than a couple Docker containers running on a host. You may need five, ten, or dozens of containers running. And when you need to scale, you need them distributed across multiple hosts. And then when you have multiple containers on multiple hosts, you need to aggregate logs, monitor resource usage, etc.

Because of this, many different container scheduling platforms have been developed which aid in deploying containers and their supporting services: Kubernetes, Mesos, Docker Swarm, Rancher, OpenShift, etc. Because of its increasing popularity and support across all major cloud providers, this book will focus on usage of Kubernetes as a container scheduler.

## A bit of Kubernetes history

**Kubernetes logo**

In 2013, some Google engineers began working to create an open source representation of the internal tool Google used to run millions of containers in the Google data centers, named Borg. The first version of Kubernetes was known as Seven of

Nine (another Star Trek reference), but was finally renamed Kubernetes (a mangled translation of the Greek word for 'helmsman') to avoid potential legal issues.

To keep a little of the original geek culture Trek reference, it was decided the logo would have seven sides, as a nod to the working name 'Seven of Nine'.

In a few short years, Kubernetes went from being one of many up-and-coming container scheduler engines to becoming almost a *de facto* standard for large scale container deployment. In 2015, at the same time as Kubernetes' 1.0 release, the Cloud Native Computing Foundation (CNCF) was founded, to promote containers and cloud-based infrastructure.

Kubernetes is one of many projects endorsed by the CNCF for 'cloud-native' applications, and has been endorsed by VMware, Google, Twitter, IBM, Microsoft, Amazon, and many other major tech companies.

By 2018, Kubernetes was available as a service offering from all the major cloud providers, and most other competing software has either begun to rebuild on top of Kubernetes, or become more of a niche player in the container scheduling space.

Kubernetes is often abbreviated 'K8s' (K + eight-letters + s), and the two terms are interchangeable.

## Evaluating the need for Kubernetes

If Kubernetes seems to be taking the world of cloud computing by storm, should you start moving all your applications into Kubernetes clusters? Not necessarily.

Kubernetes is a complex application, and even if you're using a managed Kubernetes offering, you need to learn new terminology and many new paradigms to get applications—especially non-'cloud native' applications—running smoothly.

If you already have automation around existing infrastructure projects, and it's running smoothly, I would not start moving things into Kubernetes unless the following criteria are met:

1. Your application doesn't require much locally-available stateful data (e.g. most databases, many file system-heavy applications).
2. Your application has many parts which can be broken out and run on an ad-hoc basis, like cron jobs or other periodic tasks.

Kubernetes, like Ansible, is best introduced incrementally into an existing organization. You might start by putting temporary workloads (like report-generating jobs) into a Kubernetes cluster. Then you can work on moving larger and persistent applications into a cluster.

If you're working on a green field project, with enough resources to devote some time up front to learning the ins and outs of Kubernetes, it makes sense to at least give Kubernetes a try for running everything.

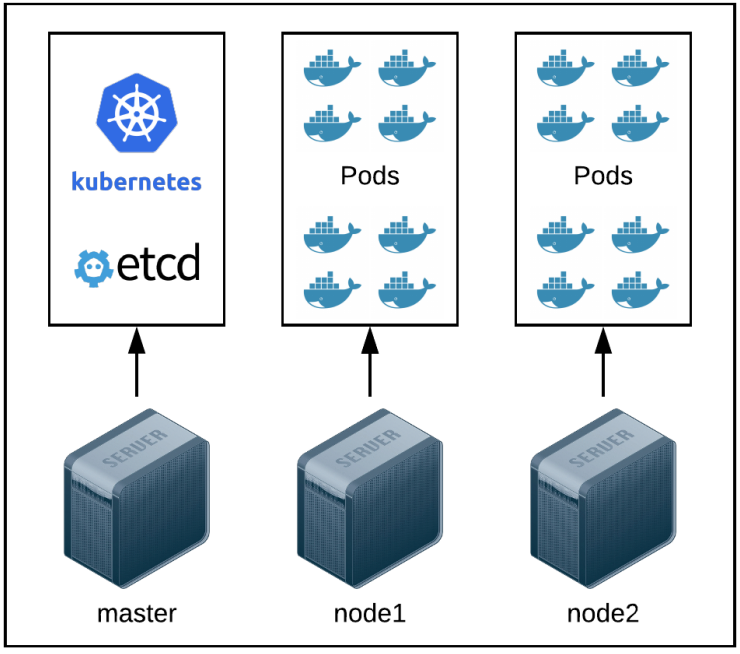
## Building a Kubernetes cluster with Ansible

There are a few different ways you can build a Kubernetes cluster:

- Using `kubeadm`[190], a tool included with Kubernetes to set up a minimal but fully functional Kubernetes cluster in any environment.
- Using tools like `kops`[191] or `kubespray`[192] to build a production-ready Kubernetes cluster in almost any environment.
- Using tools like Terraform or CloudFormation—or even Ansible modules—to create a managed Kubernetes cluster using a cloud provider like AWS, Google Cloud, or Azure.

There are many excellent guides online for the latter options, so we'll stick to using `kubeadm` in this book's examples. And, lucky for us, there's an Ansible role (`geerlingguy.kubernetes`) which already wraps `kubeadm` in an easy-to-use manner so we can integrate it with our playbooks.

[190] https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm/
[191] https://github.com/kubernetes/kops
[192] https://github.com/kubernetes-incubator/kubespray




**Kuberenetes architecture for a simple cluster**

As with other multi-server examples in this book, we can describe a three server setup to Vagrant so we can build a full 'bare metal' Kubernetes cluster. Create a project directory and add the following in a `Vagrantfile`:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "geerlingguy/debian9"
  config.ssh.insert_key = false
  config.vm.provider "virtualbox"

```

```
  config.vm.provider :virtualbox do |v|
    v.memory = 1024
    v.cpus = 1
    v.linked_clone = true
  end

  # Define three VMs with static private IP addresses.
  boxes = [
    { :name => "master", :ip => "192.168.84.2" },
    { :name => "node1", :ip => "192.168.84.3" },
    { :name => "node2", :ip => "192.168.84.4" },
  ]

  # Provision each of the VMs.
  boxes.each do |opts|
    config.vm.define opts[:name] do |config|
      config.vm.hostname = opts[:name] + ".k8s.test"
      config.vm.network :private_network, ip: opts[:ip]

      # Provision all the VMs using Ansible after last VM is up.
      if opts[:name] == "node2"
        config.vm.provision "ansible" do |ansible|
          ansible.playbook = "main.yml"
          ansible.inventory_path = "inventory"
          ansible.limit = "all"
        end
      end
    end
  end

end
```

The Vagrantfile creates three VMs:

- `master`, which will be configured as the Kubernetes master server, running the scheduling engine.

- `node1`, a Kubernetes node to be joined to the master.
- `node2`, another Kubernetes node to be joined to the master.

You could technically add as many more `nodeX` VMs as you want, but since most people don't have a terabyte of RAM, it's better to be conservative in a local setup!

Once the `Vagrantfile` is ready, you should add an `inventory` file to tell Ansible about the VMs; note our `ansible` configuration in the Vagrantfile points to a playbook in the same directory, `main.yml` and an inventory file, `inventory`. In the inventory file, put the following contents:

```
[k8s-master]
master ansible_host=192.168.84.2 kubernetes_role=master

[k8s-nodes]
node1 ansible_host=192.168.84.3 kubernetes_role=node
node2 ansible_host=192.168.84.4 kubernetes_role=node

[k8s:children]
k8s-master
k8s-nodes

[k8s:vars]
ansible_ssh_user=vagrant
ansible_ssh_private_key_file=~/.vagrant.d/insecure_private_key
```

The inventory is broken up into three groups: `k8s-master` (the Kubernetes master), `k8s-nodes` (all the nodes that will join the master), and `k8s` (a group with all the servers, helpful for initializing the cluster or operating on all the servers at once).

We'll refer to the servers using the `k8s` inventory group in our Kubernetes setup playbook. Let's set up the playbook now:

```
---
- hosts: k8s
  become: yes

  vars_files:
    - vars/main.yml
```

We'll operate on all the `k8s` servers defined in the `inventory`, and we'll need to operate as the root user to set up Kubernetes and its dependencies, so we add `become: yes`. Also, to keep things organized, all the playbook variables will be placed in the included vars file `vars/main.yml` (you can create that file now).

Next, because Vagrant's virtual network interfaces can confuse Kubernetes and Flannel (the Kubernetes networking plugin we're going to use for inter-node communication), we need to copy a custom Flannel manifest file into the VM. Instead of printing the whole file in this book (it's a *lot* of YAML!), you can grab a copy of the file from the URL: https://github.com/geerlingguy/ansible-for-devops/blob/master/kubernetes/files/manifests/kube-system/kube-flannel-vagrant.yml

Save the file in your project folder in the path:

```
files/manifests/kube-system/kube-flannel-vagrant.yml
```

Now add a task to copy the manifest file into place using `pre_tasks` (we need to do this before any Ansible roles are run):

```
  pre_tasks:
    - name: Copy Flannel manifest tailored for Vagrant.
      copy:
        src: files/manifests/kube-system/kube-flannel-vagrant.yml
        dest: "~/kube-flannel-vagrant.yml"
```

Next we need to prepare the server to be able to run `kubelet` (all Kubernetes nodes run this service, which schedules Kubernetes Pods on individual nodes). `kubelet` has a couple special requirements:

- Swap should be disabled on the server (there are a few valid reasons why you might keep swap enabled, but it's not recommended and requires more work to get `kubelet` running well.)
- Docker (or an equivalent container runtime) should be installed on the server.

Lucky for us, there are Ansible Galaxy roles which configure swap and install Docker, so let's add them in the playbook's `roles` section:

```
14   roles:
15     - role: geerlingguy.swap
16       tags: ['swap', 'kubernetes']
17
18     - role: geerlingguy.docker
19       tags: ['docker']
```

We also need to add some configuration to ensure we have swap disabled and Docker installed correctly. Add the following variables in `vars/main.yml`:

```
1  swap_file_state: absent
2  swap_file_path: /dev/mapper/packer--debian--9--amd64--vg-swap_1
3
4  docker_package: docker-ce=5:18.09.0~3-0~debian-stretch
5  docker_install_compose: False
```

The `swap_file_path` is specific to the 64-bit Debian 9 Vagrant box used in the `Vagrantfile`, so if you want to use a different OS or install on a cloud server, the default system swap file may be at a different location.

It's a best practice to specify a Docker version that's been well-tested with a particular version of Kubernetes, and in this case, the latest version of Kubernetes at the time of this writing—1.13—works well with Docker 18.09, so we lock in that package version using the `docker_package` variable.

Back in the `main.yml` playbook, we'll put the last role necessary to get Kubernetes up and running on the cluster:

```
21      - role: geerlingguy.kubernetes
22        tags: ['kubernetes']
```

At this point, our playbook uses three Ansible Galaxy roles. To make installation and maintenance easier, add a `requirements.yml` file with the roles listed inside:

```
1  ---
2  - src: geerlingguy.swap
3  - src: geerlingguy.docker
4  - src: geerlingguy.kubernetes
```

Then run `ansible-galaxy install -r requirements.yml -p ./roles` to install the roles in the project directory.

As a final step, before building the cluster with `vagrant up`, we need to set a few configuration options to ensure Kubernetes starts correctly and the inter-node network functions properly. Add the following variables to tell the Kubernetes role a little more about the cluster:

```
8  kubernetes_version: '1.13'
9  kubernetes_allow_pods_on_master: False
10 kubernetes_pod_network_cidr: '10.244.0.0/16'
11 kubernetes_packages:
12   - name: kubelet=1.13.8-00
13     state: present
14   - name: kubectl=1.13.8-00
15     state: present
16   - name: kubeadm=1.13.8-00
17     state: present
18   - name: kubernetes-cni
19     state: present
20
21 kubernetes_apiserver_advertise_address: "192.168.84.2"
22 kubernetes_flannel_manifest_file: "~/kube-flannel-vagrant.yml"
23 kubernetes_kubelet_extra_args: '--node-ip={{ inventory_hostname }}'
```

Let's go through the variables one-by-one:

- `kubernetes_version`: Kubernetes is a fast-moving target, and it's best practice to specify the version you're targeting—but to update as soon as possible to the latest version!
- `kubernetes_allow_pods_on_master`: It's best to dedicate the Kubernetes master server to managing Kubernetes alone. You can run pods other than the Kubernetes system pods on the master if you want, but it's rarely a good idea.
- `kubernetes_pod_network_cidr`: Because the default network suggested in the Kubernetes documentation conflicts with many home and private network IP ranges, this custom CIDR is a bit of a safer option.
- `kubernetes_packages`: Along with specifying the `kubernetes_version`, if you want to make sure there are no surprises when installing Kubernetes, it's important to also lock in the versions of the packages that make up the Kubernetes cluster.
- `kubernetes_apiserver_advertise_address`: To ensure Kubernetes knows the correct interface to use for inter-node API communication, we explicitly set the IP of the master node (this could also be the DNS name for the master, if desired).
- `kubernetes_flannel_manifest_file`: Because Vagrant's virtual network interfaces confuse the default Flannel configuration, we specify the custom Flannel manifest we copied earlier in the playbook's `pre_tasks`.
- `kubernetes_kubelet_extra_args`: Because Vagrant's virtual network interfaces can also confuse Kubernetes, it's best to explicitly define the `node-ip` to be advertised by `kubelet`.

Whew! We finally have the full project ready to go. It's time to build the cluster! Assuming all the files are in order, you can run `vagrant up`, and after a few minutes, you should have a three-node Kubernetes cluster running locally.

To verify the cluster is operating normally, log into the `master` server and check the node status with `kubectl`:

```
# Log into the master VM.
$ vagrant ssh master

# Switch to the root user.
vagrant@master:~$ sudo su

# Check node status.
root@master# kubectl get nodes
NAME      STATUS    ROLES     AGE       VERSION
master    Ready     master    13m       v1.11.2
node1     Ready     <none>    12m       v1.11.2
node2     Ready     <none>    12m       v1.11.2
```

If any of the nodes aren't reporting `Ready`, then something may be mis-configured. You can check the system logs to see if `kubelet` is having trouble, or read through the Kubernetes documentation to Troubleshoot Clusters[193].

You can also check to ensure all the system pods (which run services like DNS, etcd, Flannel, and the Kubernetes API) are running correctly with the command:

```
root@master# kubectl get pods -n kube-system
```

This should print a list of all the core Kubernetes service pods (some of which are displayed multiple times—one for each node in the cluster), and the status should be `Running` after all the pods start correctly.

The Kubernetes cluster example above can be found in the Ansible for DevOps GitHub repository[194].

## Managing Kubernetes with Ansible

Once you have a Kubernetes cluster—whether bare metal or managed by a cloud provider—you need to deploy applications inside. Ansible has a few modules which make it easy to automate.

[193] https://kubernetes.io/docs/tasks/debug-application-cluster/debug-cluster/
[194] https://github.com/geerlingguy/ansible-for-devops/tree/master/kubernetes

## Ansible's `k8s` module

The `k8s` module (also aliased as `k8s_raw` and `kubernetes`) requires the OpenShift Python client to communicate with the Kubernetes API. So before using the `k8s` role, you need to install the client. Since it's installed with `pip`, we need to install Pip as well.

Create a new `k8s-module.yml` playbook in an `examples` directory in the same project we used to set up the Kubernetes cluster, and put the following inside:

```
---
- hosts: k8s-master
  become: yes

  pre_tasks:
    - name: Ensure Pip is installed.
      package:
        name: python-pip
        state: present

    - name: Ensure OpenShift client is installed.
      pip:
        name: openshift
        state: present
```

We'll soon add a task to create a Kubernetes deployment that runs three Nginx replicas based on the official Nginx Docker image. Before adding the task, we need to create a Kubernetes manifest, or definition file. Create a file in the path `examples/files/nginx.yml`, and put in the following contents:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: a4d-nginx
  namespace: default
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

We won't get into the details of how Kubernetes manifests work, or why it's structured the way it is. If you want more details about this example, please read through the Kubernetes documentation, specifically Creating a Deployment[195].

Going back to the `k8s-module.yml` playbook, add a `tasks` section which uses the `k8s` module to apply the `nginx.yml` manifest to the Kubernetes cluster:

[195] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-a-deployment

```
16    tasks:
17      - name: Apply definition file from Ansible controller file system.
18        k8s:
19          state: present
20          definition: "{{ lookup('file', 'files/nginx.yml') | from_yaml }\
21  }"
```

We now have a complete playbook! Run it with the command:

```
ansible-playbook -i ../inventory k8s-module.yml
```

If you log back into the master VM (`vagrant ssh master`), change to the root user (`sudo su`), and list all the deployments (`kubectl get deployments`), you should see the new deployment that was just applied:

```
root@master:/home/vagrant# kubectl get deployments
NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
a4d-nginx   3         3         3            3           3m
```

People can't access the deployment from the outside, though. For that, we need to expose Nginx to the world. And to do that, we could add more to the `nginx.yml` manifest file, *or* we can also apply it directly with the `k8s` module. Add another task:

```
22      - name: Expose the Nginx service with an inline Service definition.
23        k8s:
24          state: present
25          definition:
26            apiVersion: v1
27            kind: Service
28            metadata:
29              labels:
30                app: nginx
31              name: a4d-nginx
32              namespace: default
33            spec:
```

```
34              type: NodePort
35              ports:
36              - port: 80
37                protocol: TCP
38                targetPort: 80
39              selector:
40                app: nginx
```

This definition is defined inline with the Ansible playbook. I generally prefer to keep the Kubernetes manifest definitions in separate files, just to keep my playbooks more concise, but either way works great!

If you run the playbook again, then log back into the master to use `kubectl` like earlier, you should be able to see the new `Service` using `kubectl get services`:

```
root@master:/home/vagrant# kubectl get services
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        A\
GE
a4d-nginx    NodePort    10.101.211.71   <none>        80:30681/TCP   3m
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP        5d
```

The Service exposes a `NodePort` on each of the Kubernetes nodes—in this case, port `30681`, so you can send a request to any node IP or DNS name and the request will be routed by Kubernetes to an Nginx service Pod, no matter what node it's running on.

So in the example above, I visited `http://192.168.84.3:30681/`, and got the default Nginx welcome message:


**Welcome to nginx message in browser**

For a final example, it might be convenient for the playbook to output a debug message with the NodePort the Service is using. In addition to applying or deleting Kubernetes manifests, the `k8s` module can get cluster and resource information that can be used elsewhere in your playbooks.

Add two final tasks to retrieve the NodePort for the `a4d-nginx` service using `k8s_info`, then display it using `debug`:

```
42      - name: Get the details of the a4d-nginx Service.
43        k8s_info:
44          api_version: v1
45          kind: Service
46          name: a4d-nginx
47          namespace: default
48        register: a4d_nginx_service
49
50      - name: Print the NodePort of the a4d-nginx Service.
51        debug:
52          var: a4d_nginx_service.resources[0].spec.ports[0].nodePort
```

When you run the playbook, you should now see the NodePort in the debug output:

```
TASK [Print the NodePort of the a4d-nginx Service.] ******************\
**********************************************
ok: [master] => {
    "a4d_nginx_service.result.spec.ports[0].nodePort": "30681"
}
```

For bonus points, you can build a separate cleanup playbook to delete the Service and Deployment objects using `state: absent`:

```
---
- hosts: k8s-master
  become: yes

  tasks:
    - name: Remove resources in Nginx Deployment definition.
      k8s:
        state: absent
        definition: "{{ lookup('file', 'files/nginx.yml') | from_yaml }\
}"

    - name: Remove the Nginx Service.
      k8s:
        state: absent
        api_version: v1
        kind: Service
        namespace: default
        name: a4d-nginx
```

You could build an entire ecosystem of applications using nothing but Ansible's `k8s` module and custom manifests. But there are many times when you might not have the time to tweak a bunch of Deployments, Services, etc. to get a complex application running, especially if it's an application with many components that you're not familiar with.

Luckily, the Kubernetes community has put together a number of 'charts' describing common Kubernetes applications, and you can install them using Helm[196].

[196] https://www.helm.sh

## Managing Kubernetes Applications with Helm

Helm consists of two components: `helm`, a binary which you install on a control machine to control applications in a Kubernetes cluster, and Tiller, the component which runs inside the Kubernetes cluster and coordinates chart operations triggered by Helm.

To automate Helm and Tiller setup, we'll create a playbook that installs the `helm` binary, configures Kubernetes to allow Tiller to manage resources, and then runs `helm init` to initialize Tiller in the cluster.

Create a `helm.yml` playbook in the `examples` directory, and put in the following:

```
---
- hosts: k8s-master
  become: yes

  tasks:
    - name: Create Tiller ServiceAccount.
      k8s:
        state: present
        definition:
          apiVersion: v1
          kind: ServiceAccount
          metadata:
            name: tiller
            namespace: kube-system

    - name: Apply Tiller RBAC definition.
      k8s:
        state: present
        definition: "{{ lookup('file', 'files/tiller-rbac.yml') | from_\
yaml }}"

    - name: Retrieve helm binary archive.
      unarchive:
        src: https://storage.googleapis.com/kubernetes-helm/helm-v2.10.\
```

```
0-linux-amd64.tar.gz
        dest: /tmp
        creates: /usr/local/bin/helm
        remote_src: yes

    - name: Move helm binary into place.
      command: >
        cp /tmp/linux-amd64/helm /usr/local/bin/helm
        creates=/usr/local/bin/helm

    - name: Set up Helm and Tiller.
      command: helm init --service-account tiller
      register: helm_init_result
      changed_when: "'already installed' not in helm_init_result.stdout"
```

Tiller needs to be allowed to manage certain resources in the cluster, so the first two tasks in this playbook define a `tiller` namespace which Tiller will operate within, and then apply an RBAC definition which defines a `ClusterRoleBinding` that allows Tiller to operate as a `cluster-admin`. Create the `tiller-rbac.yml` file inside the `examples/files` directory, with the contents:

```
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name: tiller
    namespace: kube-system
```

The next tasks in the Helm playbook download `helm`, place it in `/usr/local/bin`, and then initializes `helm` in the local user account, and `tiller` in the Kubernetes

cluster (utilizing the `tiller` service account we configured in the `tiller-rbac.yml` manifest).

At this point, if you run the playbook, then check running services, you should see Tiller running, ready to deploy new Helm charts:

```
root@master:/home/vagrant# kubectl get services -n kube-system
NAME            TYPE        CLUSTER-IP      PORT(S)         AGE
kube-dns        ClusterIP   10.96.0.10      53/UDP,53/TCP   6d
tiller-deploy   ClusterIP   10.99.161.154   44134/TCP       5m
```

Let's take it a little further, though, and automate the deployment of a chart maintained in Helm's `stable` chart collection. Add more tasks to the playbook:

```
38      - name: Get Tiller's ClusterIP.
39        k8s:
40          api_version: v1
41          kind: Service
42          name: tiller-deploy
43          namespace: kube-system
44        register: tiller_service
45
46      - name: Set the Helm host and port.
47        set_fact:
48          helm_host: "{{ tiller_service.result.spec.clusterIP }}"
49          helm_port: "{{ tiller_service.result.spec.ports[0].port }}"
50
51      - name: Wait for Tiller to become responsive.
52        wait_for:
53          host: '{{ helm_host }}'
54          port: '{{ helm_port }}'
55          state: started
56
57      - name: List installed Helm charts.
58        command: helm list
59        environment:
```

```
60          HELM_HOST: '{{ helm_host }}:{{ helm_port }}'
61        register: helm_list_results
62        changed_when: False
63
64      - name: Install phpMyAdmin with Helm.
65        command: >
66          helm install --name phpmyadmin stable/phpmyadmin
67          --set service.type=NodePort
68        environment:
69          HELM_HOST: '{{ helm_host }}:{{ helm_port }}'
70        when: "'phpmyadmin' not in helm_list_results.stdout"
```

After `helm init` is run, the `tiller-deploy` service takes a little time to start up. The first three tasks get the tiller IP address and port, then wait for tiller to be responsive (using `wait_for`) before the rest of the playbook runs.

Then we check all deployed charts using the command `helm list`, and if the chart we are deploying (`phpmyadmin`) is not in the list results, we install it with the command `helm install stable/phpmyadmin`. Note that we explicitly define the `HELM_HOST` environment variable. Without this, the `helm` command may have a hard time finding the right host for Tiller.

Also, because the default for most Helm charts is to use a service type of `LoadBalancer`, and it's a little difficult to set up Load Balancer services in a bare metal Kubernetes cluster, we are overriding the `service.type` for the `stable/phpmyadmin` chart and forcing it to use `NodePort`.

Many charts (e.g. `stable/wordpress`, `stable/drupal`, `stable/jenkins`) will install but won't fully run on this Kubernetes cluster, because they require Persistent Volumes (PVs), which require some kind of shared filesystem (e.g. NFS, Ceph, Gluster, or something similar) among all the nodes. If you want to use charts which require PVs, check out the NFS configuration used in the Raspberry Pi Dramble[197] project, which allows applications to use Kubernetes PVs and PVCs.

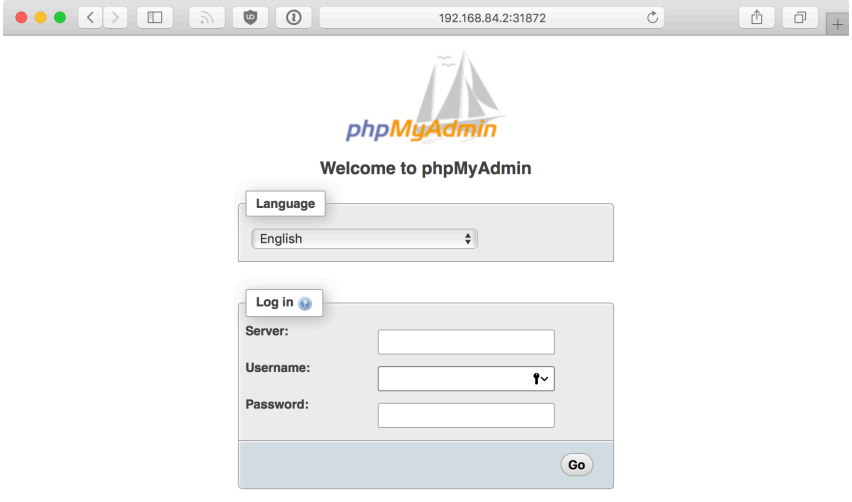
At this point, you could log into the master, change to the root user (`sudo su`), and

[197] https://github.com/geerlingguy/raspberry-pi-dramble

run `kubectl get services` to see the `phpmyadmin` service's `NodePort`, but it's better to automate that step at the end of the `helm.yml` playbook:

```
72      - name: Get the details of the phpmyadmin Service.
73        k8s:
74          api_version: v1
75          kind: Service
76          name: phpmyadmin
77          namespace: default
78        register: phpmyadmin_service
79
80      - name: Print the NodePort of the phpmyadmin Service.
81        debug:
82          var: phpmyadmin_service.result.spec.ports[0].nodePort
```

Run the playbook, grab the debug value, and append the port to the IP address of any of the cluster members. Once the `phpmyadmin` deployment is running and healthy (this takes about 30 seconds), you can access phpMyAdmin at http://192.168.84.3:31872/ (substituting the `NodePort` from your own cluster):




**phpMyAdmin running in the browser on a NodePort**

## Interacting with Pods using the `kubectl` connection plugin

Ansible ships with a number of Connection Plugins. Last chapter, we used the `docker` connection plugin to interact with Docker containers natively, to avoid having to use SSH with a container or installing Ansible inside the container.

This chapter, we'll use the `kubectl` connection plugin, which allows Ansible to natively interact with running Kubernetes pods.

One of the main tenets of 'immutable infrastructure' (which is truly realized when you start using Kubernetes correctly) is *not logging into individual containers and running commands*, so this example may seem contrary to the core purpose of Kubernetes. However, it is sometimes necessary to do so. In cases where your applications are not built in a way that works completely via external APIs and Pod-to-Pod communication, you might need to run a command directly inside a running Pod.

Before using the `kubectl` connection plugin, you should already have the `kubectl` binary installed and available in your `$PATH`. You should also have a running Kubernetes cluster; for this example, I'll assume you're still using the same cluster from the previous examples, with the `phpmyadmin` service running.

Create a new playbook in the `examples` directory, named `kubectl-connection.yml`. The first thing we'll do in the playbook is retrieve the `kubectl` config file from the master server so we can run commands delegated directly to a Pod of our choosing:

```
---
# This playbook assumes you already have the kubectl binary installed
# and available in the $PATH.
- hosts: k8s-master
  become: yes

  tasks:
    - name: Retrieve kubectl config file from the master server.
      fetch:
        src: /root/.kube/config
        dest: files/kubectl-config
        flat: yes
```

After using `fetch` to grab the config file, we need to find the name of the `phpmyadmin` Pod. This is necessary so we can add the Pod directly to our inventory:

```
14      - name: Get the phpmyadmin Pod name.
15        command: >
16          kubectl --no-headers=true get pod -l app=phpmyadmin
17          -o custom-columns=:metadata.name
18        register: phpmyadmin_pod
```

I've used the `kubectl` command directly here, because there's no simple way using the `k8s` module and Kubernetes' API to directly get the name of a Pod for a given set of conditions—in this case, with the label `app=phpmyadmin`.

We can now add the pod by name name (using `phpmyadmin_pod.stdout`) to the current play's inventory:

```
20      - name: Add the phpmyadmin Pod to the inventory.
21        add_host:
22          name: '{{ phpmyadmin_pod.stdout }}'
23          ansible_kubectl_namespace: default
24          ansible_kubectl_config: files/kubectl-config
25          ansible_connection: kubectl
```

The `ansible_connection: kubectl` is key here; it tells Ansible to use the `kubectl` connection plugin when connecting to this host.

There are a number of options you can pass to the `kubectl` connection plugin to tell it how to connect to your Kubernetes cluster and pod. In this case, the location of the downloaded `kubectl` config file is passed to `ansible_kubectl_config` so Ansible knows where the cluster configuration exists. It's also a good practice to always pass the `namespace` of an object, so we've set that as well.

Now that we have a new host (in this case, the phpmyadmin service's Pod) added to the inventory, let's run a task directly against it:

```
    # Note: Python is required to use other modules.
    - name: Run a command inside the container.
      raw: date
      register: date_output
      delegate_to: '{{ phpmyadmin_pod.stdout }}'

    - debug: var=date_output.stdout
```

The `raw` task passes through the given command directly using `kubectl exec`, and returns the output. The `debug` task should then print the output of the `date` command, run inside the container.

You can do a lot more with the `kubectl` connection plugin, and you could even have a Dynamic inventory which populates a whole set of Pods for you to work with. It's generally not ideal to directly interact with pods, but when it's necessary, it's nice to be able to automate it with Ansible!

The `raw` module was used to run the `date` command in this example because all other Ansible modules require Python to be present on the container running in the Pod. For many use cases, running a `raw` command should be adequate. But if you want to be able to use any other modules, you'll need to make sure Python is present in the container *before* you try using the `kubectl` connection plugin with it.

# Summary

There are many ways you can build a Kubernetes cluster, whether on a managed cloud platform or bare metal. There are also many ways to deploy and manage applications within a Kubernetes cluster.

Ansible's robust variable management, Jinja templating, and YAML support makes it a strong contender for managing Kubernetes resources. At the time of this writing, Ansible has a stable `k8s` module, an experimental `helm` module, and a `kubectl` connection plugin, and the interaction between Ansible and Kubernetes is still being refined every release.

```
 ______________________________________
/ Never try to teach a pig to sing. It \
| wastes your time and annoys the pig. |
\ (Proverb)                            /
 --------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```