Setting up a Kubernetes Cluster

# Folder

First thing’s first! We will need to pull the latest version of the GitLab project. One this is done everything can be found in the “**session3**” folder.

# SSH

Once this is done you can SSH into your Kubernetes cluster by running the following command on Linux/OSX:

$ ssh -i id\_meetup vagrant@<ip\_address\_of\_kubernetes\_master>

$ ssh -i id\_meetup vagrant@<ip\_address\_of\_kubernetes\_worker>

**Note:** In order to do the same on windows, you will have to refer to the documentation of the specific client you are using in regards to working with users and private keys.

The IP addresses of each node will be provided to your team.

We are now in a remote computer somewhere in the ether!

# Sudo

In order to setup and manage Kubernetes, we will need to run all commands as a root user. To become root, run the following:

$ sudo su

# Cluster Setup

To set up the master node, we run:

$ kubeadm init --token 90f324.dfa440add2b1bd93 --pod-network-cidr 10.244.0.0/16

This will take a little while to run, as it pulls a number of docker images and starts setting up some basic pods to get the cluster up and running.

A few notes on the options:

* token
  + This would generally be a randomly generated token used to securely verify worker nodes and other software when connecting to the cluster. We are using a fixed value in order to make things more consistent for the meetup
* pod-network-cidr
  + In order to secure communications within the Kubernetes cluster, Kubernetes creates its own internal `pod` network and here we are specifying the range in which it will allocate IPs
  + You can ignore this for the most part, but understand that Kubernetes networking can be a fairly complex concept, and that in production deployments understanding this can be very important

Once this is complete, you can run the following on the worker node:

$ kubeadm join --token 90f324.dfa440add2b1bd93 ipaddressofmaster:6443

Since we are running a very small cluster here, we also want to be able to take advantage of the resources on the master node. This is generally something we want to avoid for production, but is acceptable during development. To do this, we run:

$ kubectl taint nodes --all node-role.kubernetes.io/master-

After this a number of things will be deployed on both nodes, and now would be a good time to watch and see the progress. In order to do this, we will need to set up access to Kubernetes control software, `kubectl`. First we run:

$ export KUBECONFIG=/etc/kubernetes/admin.conf

This will tell kubectl where to find the configuration file for the cluster (as well as credentials mentioned earlier). Note that if you exit your terminal, you will have to run this command again (unless you add it to your bash profile, though we will not cover this). In order to see the status of pods in the system, you can run:

$ kubectl get pods --all-namespaces -o wide

This will show you a list of the current pods, the namespace they are in, the status, and what node they are running on.

Eventually the status for every pod will have “**RUNNING**”… We now have an entire cluster ready to go!

This process is a bit tedious. However, we can make it better by deploying a Kubernetes `dashboard` application before proceeding. We can do this with the following:

$ kubectl create -f /images/kubernetes-dashboard.yaml

If we run the previous command again we will see a new pod! Once it’s status is “**RUNNING**” we can access this new application in our cluster. So let’s run that command again!

$ kubectl get pods --all-namespaces -o wide

The IP that is shown in the result will NOT work. This is an IP address created by the network plugin (one of the many in the Kubernetes network used in proxying network traffic within the cluster)

You can take the node name (such as meetup0) and use this to find the actual address. In order to find out what the IP address is, you can run:

$ nslookup <name\_of\_node>

**For example:** $ nslookup meetup0

This will return an IP address, that you can directly enter into your browser with port 30000.

Typically, in a Kubernetes deployment, you would also have to specify a (randomly generated) port designated by the service associated with the pod. The reason this is done is to prevent port conflicts on a node/machine when multiple services are being run on the same port (such as multiple web servers on port 80).

Under more production-like circumstances, this process would be more streamlined. For example, in AWS, there would be a load balancer set up and “cnames” would be used. With this you would be able to enter a domain name directly into your browser, regardless of where the pod was actually located.

# Load Docker Images

To finish setting up Kubernetes for our application, you will need to run:

$ docker load -i /images/meetup-session3-spark-master.img

$ docker load -i /images/meetup-session3-spark-slave.img

$ docker load -i /images/meetup-session3-zeppelin.img

This will download three docker images that we can use within our cluster.

# Deploy Zeppelin Application

Once this is complete, we can start deploying our Zeppelin application, run:

$ kubectl create -f /images/zeppelin.yaml

And you should be able to view details from the dashboard.

For reference, a list of application ports is as follows:

|  |  |
| --- | --- |
| Application Name | Port |
| kubernetes-dashboard | 30000 |
| zeppelin | 30001 |
| tensorboard | 30002 |

You can now access Zeppelin using the same method used to find the dashboard, or by using the dashboard itself to find the location.

# Deploy Spark Cluster

You will notice that while we have our notebooks set up, the application does not work correctly. The reason for this is that Zeppelin is now set up for a production environment, rather than development like it was before!

In production, Spark and Zeppelin do NOT run on the same pod, as that is inefficient and not loss tolerant. In order to get things working correctly, we will need to deploy a Spark cluster:

$ kubectl create -f /images/spark-master.yaml

$ kubectl create -f /images/spark-slave.yaml

Don’t worry about setting up any IP addresses for connecting to this cluster, as the Kubernetes network will be able to find it for you. After the pods have completed setting up correctly, you should now be able to run Zeppelin as expected.