**Learn Kubernetes using interactive Browser based scenarios**

*Launch Single Node Kubernetes Cluster*

Minikube version: Its basically cheek the version what is running I your pc . Minikube started a virtual machine for you, and a Kubernetes cluster is now running in that VM.

kubectl cluster-info:

It’s a Details of the cluster and its health status can be discovered.

kubectl get nodes:

To view the nodes in the cluster using  this command.

With a running Kubernetes cluster, containers can now be deployed.

minikube addons enable dashboard:

Enable the dashboard using Minikube with the command

kubectl apply -f /opt/kubernetes-dashboard.yaml:

Make the Kubernetes Dashboard available by deploying the following YAML definition. This should only be used on Katacoda.v

*Launch Single Node multi-node cluture Kubeadm.*

kubeadm init --token=102952.1a7dd4cc8d1f4cc5 --kubernetes-version $(kubeadm version -o short):

In production, it's recommend to exclude the token causing kubeadm to generate one on our behalf.

In Deploy container networking interface we use an wave works so that deployment definition can be viewed at cat/opt/wave-kube.yaml . Also this can be deployed using kubect1 apply

kubectl get pod -n kube-system:

Weave will now deploy as a series of Pods on the cluster. The status of this can be viewed using the command

Once the Master and CNI has initialised, additional nodes can join the cluster as long as they have the correct token. The tokens can be managed via kubeadm token.

kubectl get nodes: The Kubernetes CLI, This is the command which is used to configuration to acces the cluster.

Docker Image : e cluster. A Docker image is a file used to execute code in Docker container which is used to create , run and Deploy application container.

The command below create a Pod based on the Docker Image katacoda/docker-http-server.

kubectl create deployment http --image=katacoda/docker-http-server:latest

kubectl apply -f dashboard.yaml:

Deploy the dashboard yaml with the command

*Deploy Container Using Kubectl*

minikube start --wait=false:

this cluster components and download the Kubectl CLI.

kubectl expose deployment http --external-ip="172.17.0.39" --port=8000 --target-port=80:

This command used to expose the container port 80 on the host 8000 building to the external-ip of the host.

kubectl scale --replicas=3 deployment http

The command kubectl scale allows us to adjust the number of Pods running for a particular deployment or replication controller.

.*Deploy Container Using YAML*

YAML is a data serialization language that is often used for writing configuration file. Depending on whom you ask, YAML stands for yet another markup language.

As before, details of all the Service objects deployed with kubectl get svc. By describing the object it's possible to discover more details about the configuration kubectl describe svc webapp1-svc.

***Deploy Guestbook Web App Example***

Check everything is up using the following health Check:

 kubectl cluster-info kubectl get nodes

***Deploy Guest Web App Example***

Start a single-node cluster using the helper script. The helper script will launch the API, Master, a Proxy and DNS discovery. The Web App uses DNS Discovery to find the Redis slave to store data.

launch.sh

he YAML defines a redis server called *redis-master* using the official *redis* running port *6379*.

kubectl create -f redis-master-controller.yaml

The *kubectl create* command takes a YAML definition and instructs the master to start the controller.

kubectl create -f redis-master-service.yaml

The YAML defines the name of the replication controller, *redis-master*, and the ports which should be proxied.

***Networking Introduction***

This will deploy a web app with two replicas to showcase load balancing along with a service. The Pods can be viewed at kubectl get pods

Target ports allows us to separate the port the service is available on from the port the application is listening on. TargetPort is the Port which the application is configured to listen on. Port is how the application will be accessed from the outside.

Similar to previously, the service and extra pods are deployed via kubectl apply -f clusterip-target.yaml

While TargetPort and ClusterIP make it available to inside the cluster, the NodePort exposes the service on each Node’s IP via the defined static port. No matter which Node within the cluster is accessed, the service will be reachable based on the port number defined.

kubectl apply -f nodeport.yaml

When running in the cloud, such as EC2 or Azure, it's possible to configure and assign a Public IP address issued via the cloud provider. This will be issued via a Load Balancer such as ELB. This allows additional public IP addresses to be allocated to a Kubernetes cluster without interacting directly with the cloud provider.

As Katacoda is not a cloud provider, it's still possible to dynamically allocate IP addresses to LoadBalancer type services. This is done by deploying the Cloud Provider using kubectl apply -f cloudprovider.yaml. When running in a service provided by a Cloud Provider this is not required.

While the IP address is being defined, the service will show Pending. When allocated, it will appear in the service list.

kubectl get svc

kubectl describe svc/webapp1-loadbalancer-svc

***Create Ingress Routing***

To start, deploy an example HTTP server that will be the target of our requests. The deployment contains three deployments, one called *webapp1* and a second called *webapp2*, and a third called *webapp3* with a service for each.

cat deployment.yaml

Ingress rules are an object type with Kubernetes. The rules can be based on a request host (domain), or the path of the request, or a combination of both.

An example set of rules are defined within cat ingress-rules.yaml

The important parts of the rules are defined below.

the traffic will be routed to the defined place.

The first request will be processed by the *webapp1* deployment.

curl -H "Host: my.kubernetes.example" 172.17.0.32/webapp1

The second request will be processed by the *webapp2* deployment.

curl -H "Host: my.kubernetes.example" 172.17.0.32/webapp2

***Liveness and Readiness Healtchecks***

To start Launch Cluster we need to launch a Kubernetes cluster.

Launch.sh is the command to start component and download the kubectl

When deploying the Replication Controller, each Pod has a Readiness and Liveness check. Each check has the following format for performing a healthcheck over HTTP.

livenessProbe:

httpGet:

path: /

port: 80

initialDelaySeconds: 1

timeoutSeconds: 1

The settings can be changed to call different endpoints, for example, /ping, based on our application.

The first Pod, bad-frontend is an HTTP service which always returns a 500 error indicating it hasn't started correctly. We can view the status of the Pod with kubectl get pods --selector="name=bad-frontend"

***Getting Started With CRI-O and Kubeadm***

The first stage of initialising the cluster is to launch the master node. The master is responsible for running the control plane components, etcd and the API server. Clients will communicate to the API to schedule workloads and manage the state of the cluster.

Task: The following will use CRI-O, a lightweight container runtime for Kubernetes. There is currently a bug meaning that CRI-O needs to be restarted before beginning. Execute the workaround with systemctl restart crio

The command below will initialise the cluster with a known token to simplify the following steps. The command points to an alternative Container Runtime Interface (CRI), in this case, CRI-O.

kubeadm init --cri-socket=/var/run/crio/crio.sock --kubernetes-version $(kubeadm version -o short)

In production, it's recommend to exclude the token causing kubeadm to generate one on your behalf.

Instead, everything is managed via CRI-O. The status of which can be explored via crictl

crictl images

crictl ps

In this scenario a single node has been provisioned. Remove the taint to deploy applications to the Master node.

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

Kubernetes has abstract away the underlying Container runtime meaning the commands behave as expected.

kubectl get pods --all-namespaces

***Running Stateful Services on Kubernetes***

NFS is a protocol that allows nodes to read/write data over a network. The protocol works by having a master node running the NFS daemon and stores the data. This master node makes certain directories available over the network. Clients access the masters shared via drive mounts. From the viewpoint of applications, they are writing to the local disk. Under the covers, the NFS protocol writes it to the master.

Create two new PersistentVolume definitions to point at the two available NFS shares.

kubectl create -f nfs-0001.yaml

kubectl create -f nfs-0002.yaml

View the contents of the files using cat nfs-0001.yaml nfs-0002.yaml

Once created, view all *PersistentVolumes* in the cluster using kubectl get pv

Our Pods can now read/write. MySQL will store all database changes to the NFS Server while the HTTP Server will serve static from the NFS drive. When upgrading, restarting or moving containers to a different machine the data will still be accessible.

To test the HTTP server, write a 'Hello World' *index.html* homepage. In this scenario, we know the HTTP directory will be based on *data-0001* as the volume definition hasn't driven enough space to satisfy the MySQL size requirement.

docker exec -it nfs-server bash -c "echo 'Hello World' > /exports/data-0001/index.html"

Based on the IP of the Pod, when accessing the Pod, it should return the expected response.

ip=$(kubectl get pod www -o yaml |grep podIP | awk '{split($0,a,":"); print a[2]}'); echo $ip

curl $ip

Deleting a Pod will cause it to remove claims to any persistent volumes. New Pods can pick up and re-use the NFS share.

kubectl delete pod www

kubectl create -f pod-www2.yaml

***Use Kubernets To Manage Secrets And Password***

Using the command line tool we can create the Base64 strings and store them as variables to use in a file. username=$(echo -n "admin" | base64) password=$(echo -n "a62fjbd37942dcs" | base64)

The secret is defined using *yaml*. Below we'd using the variables defined above and providing them with friendly labels which our application can use. This will create a collection of key/value secrets that can be accessed via the name, in this case *test-secret*

echo "apiVersion: v1 kind: Secret metadata: name: test-secret type: Opaque data: username: $username password: $password" >> secret.yaml

This *yaml* file can be used to with Kubectl to create our secret. When launching pods that require access to the secret we'll refer to the collection via the friendly-name.

Use kubectl to create our secret.

kubectl create -f secret.yaml

The following command allows you to view all the secret collections defined.

kubectl get secrets

In the next step we'll use these secrets via a Pod.

***Deploy Doker Compose Files with Kompose***

Kompose takes existing Docker Compose files and enables them to be deployed onto Kubernetes. Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a Compose file to configure your application’s services.

Kompose also has the ability to take existing Compose files and generate the related Kubernetes Manifest files.

The command kompose convert will generate the files, viewable via ls.

Kompose also supports different Kubernetes distributions, for example OpenShift.

kompose --provider openshift convert