Kubernetes Administration

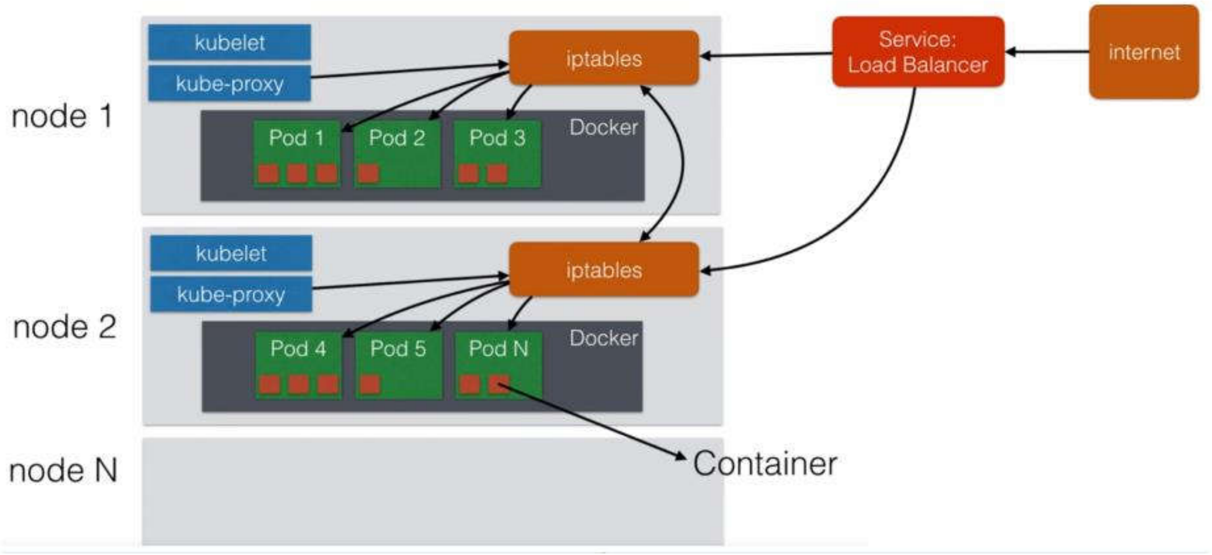
Aniruddh Amonker

aamonker@juniper.net

Revision History

Date(mm/dd/yyyy)	Revision	Description
09/19/2018	1.0	Initial Draft

Kubernetes Architecture



- Kubelet is used to launch the pods and it gets this information from the master node.
- Kube-proxy is used to feed the information about the ports being used by the pods to the iptables. So whenever a pod is launched, the kube-proxy changes the iptables rules so that the new pod is easily routable within the cluster.
- Service in Kubernetes accomplishes the goal of connectivity to the containers/pods from outside or external users

Creating "Hello-World" Pod

- Creating a Kubernetes pod for a simple web server application using pod definition YAML file.

```
root@k8s:~/learning-K8s# cat SimpleWebServer.yml
apiVersion: v1
kind: Pod
metadata:
    name: simplewebserver.cfts.com
    labels:
        app: helloworld
spec:
    containers:
    - name: k8s-demo
      image: anirudh991/k8s-demo
      ports:
      - containerPort: 3000
        name: webserver-port
```

- Once we define a pod definition YAML file, We can use kubectl to create a pod on the k8s cluster.

```
root@k8s:~# kubectl create -f first-app/helloworld.yml
pod "simplewebserver.cfts.com" created

root@k8s:~/learning-K8s# kubectl get pod
NAME                                    READY     STATUS        RESTARTS     AGE
simplewebserver.cfts.com   1/1         Running         0           <invalid>
```

Kubernetes Services

- Pods are very dynamic and mortal in nature. They can be terminated and re-instantiated across different nodes in the kubernetes cluster.
- While each Pod gets its own IP address, even those IP addresses cannot be relied upon to be stable over time as the pods are re-instantiated across different nodes.`
- A Service is an abstraction or a logical bridge between the mortal pods and the end-users or applications.
- Creating a new service for web server pod created above using service definition YAML file.

```
root@k8s:~/learning-K8s# cat SimpleWebServer-service.yml
apiVersion: v1
kind: Service
metadata:
    name: simplewebserver-service
spec:
    ports:
    - port: 31001
      targetPort: webserver-port
      protocol: TCP
    selector:
      app: helloworld
type: NodePort
```

- Above example provides a service of type NodePort. If we don't specify "NodePort: 31001" in the YAML file , a random port will be selected
- Note: by default service can only run between ports 30000-32767, but you could change this behavior by adding the **--service-node-port-range**— argument to the kube-apiserver (in the init scripts)
- Using the service definition YAML file, we can create a service for pod "SimpleWebServer"

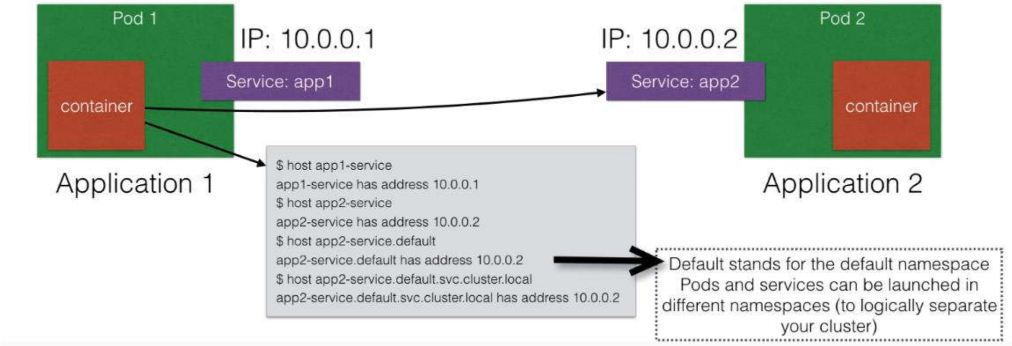
```
root@k8s:~/learning-K8s# kubectl create -f SimpleWebServer-service.yml
service "simplewebserver-service" created

root@k8s:~/learning-K8s# kubectl describe svc "simplewebserver-service"
Name:                     simplewebserver-service
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=helloworld
Type:                     NodePort
IP:                       10.102.118.222
Port:                     <unset>  31001/TCP
TargetPort:               webserver-port/TCP
NodePort:                 <unset>  31001/TCP
Endpoints:                172.17.0.4:3000
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

- A service can also be created using "Kubectl expose" command

```
kubectl expose pod simplewebserver.cfts.com --type=NodePort --name=simplewebserver-service
```

DNS Service Discovery

- The DNS service is used within pods to find other services running on the same cluster.
- Multiple containers within one pod do not need this service as they can contact each other directly.
- With service discovery, apps/containers can use FQDN of a service to connect across pods. However for service discovery to work, the container app must have a service defined and exposed.



```
$ host app1-service
app1-service has address 10.0.0.1
$ host app2-service
app2-service has address 10.0.0.2
$ host app2-service.default
app2-service.default has address 10.0.0.2
$ host app2-service.default.svc.cluster.local
app2-service.default.svc.cluster.local has address 10.0.0.2
```

Default stands for the default namespace
Pods and services can be launched in
different namespaces (to logically separate
your cluster)

- Kubernetes cluster by default launches a pod named "Kube-dns" inside "Kube-system" namespace.
- When any application pod/container is deployed, kubernetes sets the IP address of kube-dns service pod as the DNS server IP address inside the application pod/container
- A container/app running inside one pod can query the DNS service to receive ip address and port number of another pod hosting another service

```
root@node51:/opt/openstack-helm# kubectl get svc -n kube-system
NAME                   TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)              AGE
etcd                   ClusterIP   10.96.232.136   <none>        6666/TCP             5d
ingress                ClusterIP   10.111.19.212   <none>        80/TCP,443/TCP       1d
ingress-error-pages    ClusterIP   None            <none>        80/TCP               1d
ingress-exporter       ClusterIP   10.96.81.253    <none>        10254/TCP            1d
kube-dns               ClusterIP   10.96.0.10      <none>        53/UDP,53/TCP        5d
tiller-deploy          ClusterIP   10.99.6.152     <none>        44134/TCP,44135/TCP  5d

root@node51:/opt/openstack-helm# kubectl exec -i -t contrail-config-zbp5t -c contrail-config-api -n
contrail – bash

(config-api)[root@node52 /]$ cat /etc/resolv.conf
nameserver 10.96.0.10
search contrail.svc.cluster.local svc.cluster.local cluster.local
options ndots:5

(config-api)[root@node52 /]$ ping analytics-api-server
PING analytics-api-server.contrail.svc.cluster.local (10.105.208.169) 56(84) bytes of data.

^C
```

Deployments

- A deployment in Kubernetes is a declaration that allows you to do app deployments and updates with ease.

- We define the state of our application when using the deployment object. The Kubernetes than makes sure the clusters matches your desired state.

- For e.g an app state can be to run this application and have replicated 5 times.

- With Deployment object we can:

  - Create a deployment (deploying an app)

  - Update a deployment (deploying a new version)

  - Do rolling updates (zero downtime deployments)

  - Rollback to previous deployment versions

  - Scale up the Deployment to facilitate more load

Creating Deployments

- Deployment Definition YAML:

```
root@k8s:~/learning-k8s# cat deployment/helloworld.yml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: helloworld-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: helloworld
    spec:
      containers:
      - name: k8s-demo
        image: anirudh991/k8s-demo
        ports:
        - name: webserver-port
          containerPort: 3000


root@k8s:~/learning-k8s# kubectl create -f deployment/helloworld.yml
deployment.extensions "helloworld-deployment" created


root@k8s:~/learning-k8s# kubectl get deployments
NAME                    DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
helloworld-deployment   3         3         3            2           16s

root@k8s:~/learning-k8s# kubectl get pod --show-labels
NAME                                     READY     STATUS    RESTARTS   AGE       LABELS
helloworld-deployment-67bd889c49-6b6rg   1/1       Running   0          1m        app=helloworld,pod-
template-hash=2368445705
helloworld-deployment-67bd889c49-h47vg   1/1       Running   0          1m        app=helloworld,pod-
template-hash=2368445705
helloworld-deployment-67bd889c49-lw4wn   1/1       Running   0          1m        app=helloworld,pod-
template-hash=2368445705

root@k8s:~/learning-k8s# kubectl rollout status deployment/helloworld-deployment
deployment "helloworld-deployment" successfully rolled out

root@k8s:~/learning-k8s# kubectl describe deployment helloworld-deployment
Name:                   helloworld-deployment
Namespace:              default
CreationTimestamp:      Mon, 06 Aug 2018 21:22:55 -0700
Labels:                 app=helloworld
```

```
Annotations:            deployment.kubernetes.io/revision=1
Selector:               app=helloworld
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
Pod Template:
  Labels:  app=helloworld
  Containers:
   k8s-demo:
    Image:        anirudh991/k8s-demo
    Port:         3000/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   helloworld-deployment-67bd889c49 (3/3 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  4m    deployment-controller  Scaled up replica set helloworld-deployment-
67bd889c49 to 3
```

- Similar to how a service needs to be exposed after creation in order for outside access, we can similarly expose a deployment which will automatically create a service for our deployment.

```
root@k8s:~/learning-k8s# kubectl expose deployment helloworld-deployment --type=NodePort
service "helloworld-deployment" exposed


root@k8s:~/learning-k8s# kubectl get services
NAME                    TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
helloworld-deployment   NodePort    10.102.52.67     <none>        3000:30148/TCP   1m
root@k8s:~/learning-k8s#
root@k8s:~/learning-k8s#

root@k8s:~/learning-k8s# kubectl describe service helloworld-deployment
Name:                     helloworld-deployment
Namespace:                default
Labels:                   app=helloworld
Annotations:              <none>
Selector:                 app=helloworld
Type:                     NodePort
IP:                       10.102.52.67
Port:                     <unset> 3000/TCP
TargetPort:               3000/TCP
NodePort:                 <unset> 30148/TCP
Endpoints:                172.17.0.2:3000,172.17.0.5:3000,172.17.0.7:3000
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

Some Useful Deployment Commands

Command	Description
kubectl get deployments	Get information on current deployments
kubectl get rs	Get information about the replica sets
kubectl get pods —show-labels	get pods, and also show labels attached to those nods
kubectl rollout status deployment/helloworld-deployment	Get deployment status
kubectl set image deployment/helloworld-deployment k8s-demo=k8s-demo:2	Run k8s-demo with the image label version 2
kubectl edit deployment/helloworld-deployment	Edit the deployment object
kubectl rollout status deployment/helloworld-deployment	Get the status of the rollout
kubectl rollout history deployment/helloworld-deployment	Get the rollout history
kubectl rollout undo deployment/helloworld-deployment	Rollback to previous version
kubectl rollout undo deployment/helloworld-deployment —to-revision—n	rollback to any version version

Healthchecks

Liveness Probes

- Under certain scenarios it is possible that application may malfunction even though the pod/container may still be running.
- Healthchecks in Kubernetes help to detect and resolve problems with the application
- There are two types of health checks
  - Running a command inside of a container periodically
  - Periodic checks on a URL (HTTP). This method of health checks is more widely used.
- With checks using URL, the app normally exposes a URL which we can use to confirm if the app is running healthy.
- Health checks are configured in our POD definition YAML file in the following way:

```
root@k8s:~# cat learning-k8s/deployment/helloworld-healthcheck.yml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: helloworld-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: helloworld
    spec:
      containers:
      - name: k8s-demo
        image: anirudh991/k8s-demo
        ports:
        - name: webserver-port
          containerPort: 3000
        livenessProbe:
          httpGet:
            path: /
            port: webserver-port
          initialDelaySeconds: 15
          timeoutSeconds: 30
```

- Kubernetes will use the Liveness URL to check if the app is responding on the URL (http-get **Error! Hyperlink reference not valid.**) or not. If for some reason, kubernetes do not get 200 OK response on the health check URL, it will terminate the pod and launch a new pod.
- If we want to edit any Liveness parameters of the pod, we can use command "**kubectl edit deployment/<name_of_deployment>"**

```
root@k8s:~# kubectl edit deployment/helloworld-deployment
   -----skipped-----
      spec:
      containers:
      - image: anirudh991/k8s-demo
        imagePullPolicy: Always
        livenessProbe:
          failureThreshold: 3
          httpGet:
            path: /
            port: webserver-port
            scheme: HTTP
          initialDelaySeconds: 15
          periodSeconds: 10
          successThreshold: 1
          timeoutSeconds: 30
      ------skipped----------
```

Readiness Probes

- Besides Liveness probes, Kubernetes Healthcheck mechanisms also provide Readiness probes on a container within the pod.
- While liveness probes indicate whether a container is running, readiness probes indicates whether container is ready to serve the requests.
- If liveness probes fail to receive response, the container is restarted. However if readiness probes fail, the container will not be restarted but the failing pod's ip address will be removed from the service endpoint; so that the failing pod will not serve any requests.
- Readiness probes make sure that at the startup of a pod, the pod will only start receiving traffic when the probes succeed.
- Readiness and liveness probes are both identical in their configuration.
- Readiness probes define the READY status in the output of "kubectl get pod"

```
root@minikube-k8s:~/learning-k8s# cat deployment/helloworld-liveness-readiness.yml
-------------Skipped------------
    spec:
      containers:
      - name: k8s-demo
        image: anirudh991/k8s-demo
        ports:
        - name: webserver-port
          containerPort: 3000
        livenessProbe:
          httpGet:
            path: /
            port: webserver-port
          initialDelaySeconds: 15
          timeoutSeconds: 30
        readinessProbe:
          httpGet:
            path: /
            port: webserver-port
          initialDelaySeconds: 15
          timeoutSeconds: 30
```

- With the readinessProbe active, kubernetes will first check if the app is READY to serve requests.

```
root@k8s:~/learning-k8s/deployment# kubectl get pod
NAME                                     READY     STATUS    RESTARTS   AGE
hello-minikube-c8b6b4fdc-9f5vp           1/1       Running   3          3d
helloworld-deployment-7f8bfbdd85-8bw72   0/1       Running   0          0s
helloworld-deployment-7f8bfbdd85-fskj6   0/1       Running   0          0s
helloworld-deployment-7f8bfbdd85-hxts4   0/1       Running   0          0s
```

- We can see that even though the pods are running, the READY status is not active yet.
- The "initialDelaySeconds" is configured to be 15, to induce a delay of 15 secs before ReadinessProbe starts checking.

Pod States

- The status of a Pod can be seen in the output of "kubectl get pods"

  ```
  root@k8s:~/learning-k8s/deployment# kubectl get pod
  NAME                                     READY     STATUS    RESTARTS   AGE
  hello-minikube-c8b6b4fdc-9f5vp           1/1       Running   3          3d
  helloworld-deployment-7f8bfbdd85-8bw72   1/1       Running   0          29m
  helloworld-deployment-7f8bfbdd85-fskj6   1/1       Running   0          29m
  helloworld-deployment-7f8bfbdd85-hxts4   1/1       Running   0          29m
  ```

- A pod in a “Running” state indicates:
  - Pod has been bound to a node
  - All the containers inside the pod have been created
  - At least one container is still running, or is starting/restarting

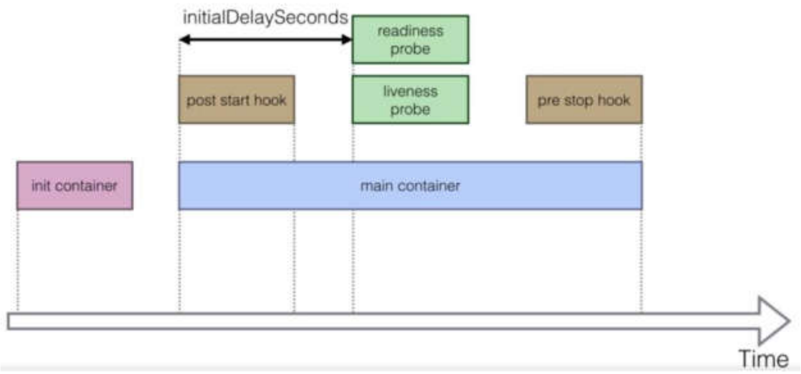
- Other valid statuses are:

  - **Pending**: Pod has been accepted but is not running. This happens when the container image is still downloading or If the pod cannot be scheduled because of resource constraints
  - **Succeeded**: All containers within this pod have been terminated successfully and will not be restarted
  - **Failed**: All the containers within this pod have been terminated and at least one container returned a failure code. The failure code is the exit code of the process when a container terminates.
  - **Unknown**: State of the pod couldn't be determined. This could be due to issues such as network errors where the node hosting this pod itself is down.

- Besides Pod state, there is also container state which is reported from the Docker. So if we need to check the status of container as reported by Docker, we can use the following command.

```
root@k8s:~# kubectl get pod helloworld-deployment-7f8bfbdd85-8bw72 -o yaml
--------------Skipped--------------------------------------------------------------
  containerStatuses:
  - containerID: docker://160e1631a461c9e5135a9785aedd79081f574d4f783c6809c885d23531a6cd79
    image: anirudh991/k8s-demo:latest
    imageID: docker-pullable://anirudh991/k8s-
    lastState:
      terminated:
        containerID: docker://dfb2a17b2bb16d29afb256ef96ce28cb3c994245b06e01a02cfb4dbdbdaba388
        exitCode: 255
        finishedAt: 2018-08-09T04:01:29Z
        reason: Error
        startedAt: 2018-08-09T00:25:30Z
    name: k8s-demo
    ready: true
    restartCount: 1
    state:
      running:
        startedAt: 2018-08-09T04:02:27Z
  hostIP: 10.0.2.15
  phase: Running
  podIP: 172.17.0.6
  qosClass: BestEffort
  startTime: 2018-08-09T00:25:22Z
```

Pod Lifecycle



Init Container

- In the lifecycle of a pod, kubernetes can create an "init container" if specified under Pod specification YAML file.
- This "init container" will be a separate container that kubernetes creates and is different from the main container that this Pod is hosting.
- The init container can be used if we need to execute some commands before the main container starts.
- This is particularly important when we use volumes where we set permissions and create directories before the main container starts.

Pre-Post Start/Stop Hook

- If defined under the Pod YAML definition, a "post start hook" starts at the same time as the main container.
- Post start hook runs within the main container and can be very useful to run any commands inside the main container when the container starts.
- Pre stop hook on the other hand is executed when the container stops. Thus if we need to do some work right before the container stops, then we can use pre-stop hook.

Probes

- Probes refer to Readiness and Liveness probes.
- Probes are defined in the main container and will only be active after the specified "initailDelaySeconds" have elapsed.
- "InitialDelaySeconds" are necessary to give the main container app to successfully complete its start process.

Secrets

- Secrets provide a way in Kubernetes to distribute credentials, keys, passwords, or secret data to the pods.
- We can also use the same mechanism to provide secrets to our application running inside pod/container.
- Secrets can be created using multiple ways:
  - Kubectl create secret command
    - Using credentials from username & passwd files
    - Using ssh keys or SSL certificates
  - A secret YAML definition file

```
root@k8s:~# kubectl create secret generic helloworld-app-creds --from-file=./username.txt --from-file=./password.txt
Secret 'first-app-user-pass' created
```

- A secret can also be a SSH key or an SSL certificate

```
root@k8s:~# kubectl create secret generic ssh-key-secret --from-file=ssh-privatekey=/root/.ssh/id_rsa
secret "ssh-key-secret" created
```

- Another way to define Secrets is by using a YAML file

```
root@k8s:~/learning-k8s/deployment# cat helloworld-secrets.yml
apiVersion: v1
kind: Secret
metadata:
  name: db-secrets
type: Opaque
data:
  username: cm9vdA==
  password: cGFzc3dvcmQ=
```

- type: Opaque means that from kubernetes's point of view the contents of this Secret is unstructured, it can contain arbitrary key-value pairs.
- Username and password are specified in base64 strings

```
root@k8s:~# cat ./username.txt | base64
cm9vdA==
root@k8s:~# cat ./password.txt | base64

cGFzc3dvcmQ=
```

- Once the YAML file is defined we can use Kubectl command to create a secret

```
root@k8s:~# kubectl create -f learning-k8s/deployment/helloworld-secrets.yml
secret "db-secrets" created
```

Using Secrets

- Once the secrets are created, we can use them in the following ways
  - Use secrets as environment variables
  - Use secrets as a file in a pod. This requires using volumes inside a container
  - We can also use an external image to pull the secrets from a private image registry. In this case our container will pull a second image and read the data from that image.

Using secrets as volume mounts

- Create a new secret using any of the methods explained above.

```
root@k8s:~# kubectl create -f kubernetes-course/deployment/helloworld-secrets.yml
secret "db-secrets" created
```

- Use “db-secrets” inside of a Deployment YAML file

```
root@k8s:~/kubernetes-course/deployment# cat helloworld-secrets-volumes.yml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: helloworld-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: helloworld
    spec:
      containers:
      - name: k8s-demo
        image: anirudh991/k8s-demo
        ports:
        - name: webserver-port
          containerPort: 3000
        volumeMounts:
        - name: cred-volume
          mountPath: /etc/creds
          readOnly: true
      volumes:
      - name: cred-volume
        secret:
          secretName: db-secrets
```

```
root@minikube-k8s:~# kubectl create -f learning-k8s/deployment/helloworld-secrets-volumes.yml
deployment.extensions/helloworld-deployment created

root@minikube-k8s:~# kubectl get pod
NAME                                    READY   STATUS        RESTARTS   AGE
helloworld-deployment-d47cfffbf-8xmdb   1/1     Running       0          2m

root@minikube-k8s:~# kubectl describe pod helloworld-deployment-d47cfffbf-8xmdb
----------------skipped--------------------------------
Containers:
  k8s-demo:Mounts:
      /etc/creds from cred-volume (ro)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-zhxrf (ro)
Volumes:
  cred-volume:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  db-secrets
    Optional:    false
  default-token-zhxrf:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-zhxrf
    Optional:    false
----------------skipped--------------------------------
```

- We can connect to the container to see the volume mount for secret created inside of the container.
  The credentials are created as files inside of container. The container app can then be configured to read the credentials from these files

```
root@minikube-k8s:~# kubectl exec -i -t helloworld-deployment-d47cfffbf-8xmdb -c k8s-demo -- ls -l
/etc/creds
total 0
lrwxrwxrwx 1 root root 15 Oct  2 06:29 password -> ..data/password
lrwxrwxrwx 1 root root 15 Oct  2 06:29 username -> ..data/username

root@helloworld-deployment-6d4c6b79d9-cldcm:/app# cat /etc/creds/username
rootroot@helloworld-deployment-6d4c6b79d9-cldcm:/app#

root@helloworld-deployment-6d4c6b79d9-cldcm:/app# cat /etc/creds/password
passwordroot@helloworld-deployment-6d4c6b79d9-cldcm:/app#
```

Configmaps

- Any configuration parameters required by an app/container which are not secrets, can be provided using Configmaps.
- All the config parameters in a configmap are provided using Key, Value pairs. An app can then read these key,value pairs using:
  - Environment variables
  - CLI arguments in the Pod configuration
  - As volumes (in a similar fashion as secrets)
- When using volumes, we can specify an entire configuration file (for e.g contrail-api.conf) as a configmap
- Kubernetes then creates new files within the container from the key value pairs with keys as filenames and values as contents of the files.
- The config file in this case will be mounted as a volume where the application expects its config file

Creating a configmap

- Create a configmap using “create configmap” command

```
root@minikube-k8s:~/learning-k8s/configmap# cat contrail-api.conf
listen_ip_addr=172.17.0.1
listen_port=8082
http_server_port=8084
log_file=/var/log/contrail/contrail-api.log
log_level=SYS_NOTICE
log_local=1
list_optimization_enabled=True
auth=keystone
aaa_mode=rbac
cloud_admin_role=admin
global_read_only_role=
cassandra_server_list=172.17.0.1:9161
zk_server_ip=172.17.0.1:2181

root@minikube-k8s:~/learning-k8s/configmap# kubectl create configmap contrail-configmap --from-file
contrail-api.conf
configmap/contrail-configmap created

root@minikube-k8s:~/learning-k8s/configmap# kubectl get configmap
NAME                 DATA   AGE
contrail-configmap   1      38s

root@minikube-k8s:~/learning-k8s/configmap# kubectl describe configmap contrail-configmap
Name:         contrail-configmap
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
contrail-api.conf:
----
listen_ip_addr=172.17.0.1
listen_port=8082
http_server_port=8084
log_file=/var/log/contrail/contrail-api.log
log_level=SYS_NOTICE
log_local=1
list_optimization_enabled=True
auth=keystone
aaa_mode=rbac
cloud_admin_role=admin
global_read_only_role=
cassandra_server_list=172.17.0.1:9161
zk_server_ip=172.17.0.1:2181

Events:  <none>
```

- We can now create a pod/deployment that utilizes the above configmap using volumes.

```
root@minikube-k8s:~/learning-k8s/configmap# cat helloworld-configmap.yml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: configmap-deployment
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: helloworld
    spec:
      containers:
      - name: k8s-demo
        image: anirudh991/k8s-demo
        ports:
        - name: webserver-port
          containerPort: 3000
        volumeMounts:
        - name: config-volume
          mountPath: /etc/contrail
      volumes:
      - name: config-volume
        configMap:
          name: contrail-configmap
          items:
          - key: contrail-api.conf
            path: contrail-api.conf


root@minikube-k8s:~/learning-k8s/configmap# kubectl create -f helloworld-configmap.yml
deployment.extensions/helloworld-deployment created

root@minikube-k8s:~/learning-k8s/configmap# kubectl get pod
NAME                                    READY   STATUS    RESTARTS   AGE
helloworld-deployment-7886c9c5b-xmmf9   1/1     Running   0          10s

root@minikube-k8s:~/learning-k8s/configmap# kubectl exec -i -t helloworld-deployment-7886c9c5b-xmmf9 -c
k8s-demo -- cat /etc/contrail/contrail-api.conf
listen_ip_addr=172.17.0.1
listen_port=8082
http_server_port=8084
log_file=/var/log/contrail/contrail-api.log
log_level=SYS_NOTICE
log_local=1
list_optimization_enabled=True
auth=keystone
aaa_mode=rbac
cloud_admin_role=admin
global_read_only_role=
cassandra_server_list=172.17.0.1:9161
zk_server_ip=172.17.0.1:2181
```

- Here is another example of a pod that exposes configmap as an environment variable

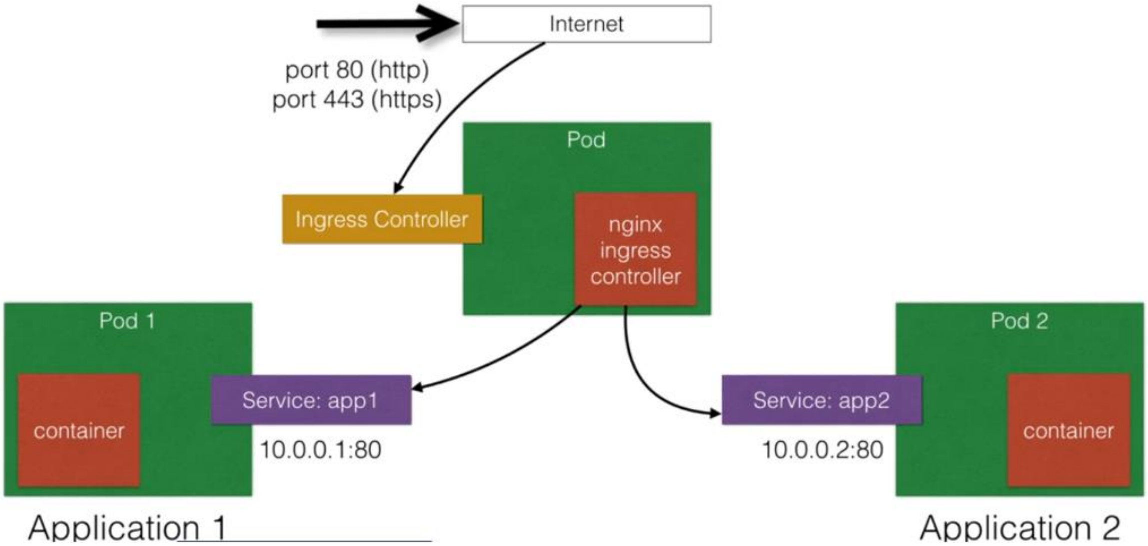
```
root@minikube-k8s:~/learning-k8s/configmap# kubectl create configmap log-config --from-
literal=log_info=INFO
configmap/log-config created

root@minikube-k8s:~/learning-k8s/configmap# cat log-configmap.yml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: configmap-deployment
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: helloworld
    spec:
      containers:
      - name: k8s-demo
```

```
        image: anirudh991/k8s-demo
        ports:
        - name: webserver-port
          containerPort: 3000
        env:
          - name: LOG_LEVEL
            valueFrom:
              configMapKeyRef:
                name: log-config
                key: log_info
```

```
root@minikube-k8s:~/learning-k8s/configmap# kubectl create -f log-configmap.yml
deployment.extensions/configmap-deployment created

root@minikube-k8s:~/learning-k8s/configmap# kubectl describe pod configmap-deployment-86779d468c-4nrdz
Name:           configmap-deployment-86779d468c-4nrdz
Namespace:      default
Node:           minikube/10.0.2.15
Start Time:     Mon, 08 Oct 2018 15:47:10 -0700
Labels:         app=helloworld
                pod-template-hash=4233580247
Annotations:    <none>
Status:         Running
IP:             172.17.0.6
Controlled By:  ReplicaSet/configmap-deployment-86779d468c
Containers:
  k8s-demo:
    Container ID:   docker://25afe55a40e5391feb306be6eb25a9887b3a7551cb4e1407b2a6ec841f97456f
    Image:          anirudh991/k8s-demo
    Image ID:       docker-pullable://anirudh991/k8s-demo@sha256:
    Port:           3000/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Mon, 08 Oct 2018 15:47:15 -0700
    Ready:          True
    Restart Count:  0
    Environment:
      LOG_LEVEL:  <set to the key 'log_info' of config map 'log-config'>  Optional: false
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-zhxrf (ro)

    root@minikube-k8s:~/learning-k8s/configmap# kubectl exec -i -t configmap-deployment-86779d468c-4nrdz -
    c k8s-demo -- bash
    root@configmap-deployment-86779d468c-4nrdz:/app# echo $LOG_LEVEL
    INFO
```

Ingress Controller

- Ingress controller is similar to kubernetes services in a manner that it allows an application to easily expose its services for external users
- Kubernetes provides a default ingress controller, or users can write their own ingress controllers
- Default ingress controller that comes with Kubernetes is a “NGINX” ingress controller.



- In the above example, when user try to access the server over internet on port 80 or port 443, the request is going to hit the ingress controller.
- We define what ports need to be handled by the ingress controller. In the example above we are handling requests on port 80 and 443.
- In this case we are using the Nginx ingress controller that comes with Kubernetes.
- This ingress controller will then distribute the traffic to our pods running inside the kubernetes cluster.
- Which requests go to which pods/services depends on the ingress rules we define for our ingress controller.
- Ingress rules are specified using an ingress object YAML file.
- An Ingress with no rules, sends all traffic to a single default backend.

Types of Ingress

- Single service ingress:
  - Similar to NodePort service where only a single service is exposed.

- Simple Fanout
  - Exposes multiple service endpoints for a single host

```
foo.bar.com -> / foo     s1:80
               / bar     s2:80
```

- Name based virtual hosting
  - Name-based virtual hosts use multiple host names for the same IP address.

```
helloworld-v1.example.com --|                    |-> helloworld-v1.example.com s1:80
                            |---178.91.123.132---|
helloworld-v2.example.com-- |                    |-> helloworld-v2.example.com s2:80
```

Deploying Ingress Controller

- In this example we are going to use default Nginx Ingress controller that Kubernetes provides in a name based virtual hosting fashion.
- We can either write our own deployment for the ingress controller or download the default definition available from GitHub:

  https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/deploy/mandatory.yaml

- Kubernetes runs "nginx-ingress-controller" pod listening on ports 80 and 443 by default.

```
-------------------skipped-----------------------------
          ports:
            - name: http
              containerPort: 80
            - name: https
              containerPort: 443
-------------------skipped------------------------------
```

```
root@minikube-k8s:~/learning-k8s/ingress# cat helloworld-v1.yml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: helloworld-v1-deployment
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: helloworld-v1
    spec:
      containers:
      - name: k8s-demo
        image: anirudh991/k8s-demo
        ports:
        - name: webserver-port
          containerPort: 3000
---
apiVersion: v1
kind: Service
metadata:
  name: helloworld-v1
spec:
  type: NodePort
  ports:
  - port: 80
    nodePort: 30303
    targetPort: 3000
    protocol: TCP
    name: http
  selector:
    app: helloworld-v1
```

- Creating deployments for the “helloworld” pods and their respective services.

```
root@minikube-k8s:~/learning-k8s/ingress# cat helloworld-v2.yml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: helloworld-v2-deployment
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: helloworld-v2
    spec:
      containers:
      - name: k8s-demo
        image: anirudh991/k8s-demo:2
        ports:
        - name: webserver-port
          containerPort: 3000
---
apiVersion: v1
kind: Service
metadata:
  name: helloworld-v2
spec:
  type: NodePort
  ports:
  - port: 80
    nodePort: 30304
    targetPort: 3000
    protocol: TCP
    name: http
  selector:
    app: helloworld-v2
```

- Defining rules in an “ingress” object that our ingress controller will use to direct the traffic to appropriate pods.

```
root@minikube-k8s:~/learning-k8s/ingress# cat ingress.yml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: helloworld-rules
spec:
  rules:
  - host: helloworld-v1.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: helloworld-v1
          servicePort: 80
  - host: helloworld-v2.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: helloworld-v2
          servicePort: 80
```

- Deploying the Kubernetes ingress controller, ingress rules and the helloworld pod deployments.

```
root@minikube-k8s:~/learning-k8s/ingress# kubectl create -f nginx-ingress-controller.yml
daemonset.extensions/nginx-ingress-controller created
configmap/nginx-configuration created
configmap/tcp-services created
configmap/udp-services created
serviceaccount/nginx-ingress-serviceaccount created
clusterrole.rbac.authorization.k8s.io/nginx-ingress-clusterrole created
role.rbac.authorization.k8s.io/nginx-ingress-role created
rolebinding.rbac.authorization.k8s.io/nginx-ingress-role-nisa-binding created
clusterrolebinding.rbac.authorization.k8s.io/nginx-ingress-clusterrole-nisa-binding created
```

```
root@minikube-k8s:~/learning-k8s/ingress# kubectl get pod
NAME                             READY     STATUS    RESTARTS   AGE
echoheaders-2pnz4                1/1       Running   0          6m
nginx-ingress-controller-k7szk   1/1       Running   5          9m

root@minikube-k8s:~/learning-k8s/ingress# kubectl create -f helloworld-v1.yml
service/helloworld-v1 created

root@minikube-k8s:~/learning-k8s/ingress# kubectl create -f helloworld-v2.yml
deployment.extensions/helloworld-v2-deployment created

root@minikube-k8s:~/learning-k8s/ingress# kubectl create -f ingress.yml
ingress.extensions/helloworld-rules created

root@minikube-k8s:~/learning-k8s/ingress# kubectl describe ingress
Name:             helloworld-rules
Namespace:        default
Address:
Default backend:  default-http-backend:80 (<none>)
Rules:
  Host                       Path  Backends
  ----                       ----  --------
  helloworld-v1.example.com
                             /      helloworld-v1:80 (<none>)
  helloworld-v2.example.com
                             /      helloworld-v2:80 (<none>)
Annotations:
Events:  <none>

root@minikube-k8s:~/learning-k8s/ingress# kubectl get pod
NAME                                        READY     STATUS    RESTARTS   AGE
echoheaders-2pnz4                           1/1       Running   0          15m
helloworld-v1-deployment-5968dcf867-ckzwt   1/1       Running   0          4m
helloworld-v2-deployment-5f6bcd87b6-nrl9h   1/1       Running   0          2m
nginx-ingress-controller-k7szk              1/1       Running   5          18m

root@minikube-k8s:~/learning-k8s/ingress# curl http://192.168.99.100/
CLIENT VALUES:
client_address=('172.17.0.5', 54016) (172.17.0.5)
command=GET
path=/
real path=/
query=
request_version=HTTP/1.1

root@minikube-k8s:~/learning-k8s/ingress# curl http://192.168.99.100/ -H 'host:helloworld-v1.example.com'
Hello World!

root@minikube-k8s:~/learning-k8s/ingress# curl http://192.168.99.100/ -H 'host:helloworld-v2.example.com'
Hello World-v2!
```

References:

- https://kubernetes.io/docs/home
- https://stackoverflow.com/questions/tagged/kubernetes
- https://www.udemy.com/share/1002XmA0QddVhVQHQ=/