1. Understand deployments and how to perform rolling update and rollbacks

Create the DaemonSet with nginx:1.10.1 image, update the ds with newer version of the nginx:1.12.1-alpine server, change the updateStrategy to OnDelete

show

Create the yaml file

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: ds-one
spec:
  selector:
    matchLabels:
      name: ds-one
  template:
    metadata:
      labels:
        name: ds-one
    spec:
      containers:
      - name: nginx
        image: nginx:1.10.1
        ports:
        - containerPort: 80
```

Using the yaml file to create the ds

```
kubectl create -f ds.yaml
```

Check the updateStrategy. DaemonSet has two update strategy types:

- OnDelete: after you update a DaemonSet template, new DaemonSet pods will only be created when you manually delete old DaemonSet pods.
- RollingUpdate: after updating a DaemonSet template, old DaemonSet pods will be killed, and new DaemonSet pods will be created automatically. The default is RollingUpdate.

```
kubectl get ds ds-one -o yaml | grep -a3 updateStrategy:
```

Check the Image of the pod. It should be 1.10.1

```
kubectl describe pod ds-one-<tab> | grep Image:
```

Set the new version of nginx server. Check the Image of the pod. It should be 1.12.1-alpine

```
kubectl set image ds ds-one nginx=nginx:1.12.1-alpine
kubectl describe pod ds-one-<tab> | grep Image:
```

Check the rollout history. View the settings for various versions

```
kubectl rollout history ds ds-one
kubectl rollout history ds ds-one --revision=1
kubectl rollout history ds ds-one --revision=2
```

Change DaemonSet back to the earlier version, check the Image:

```
kubectl rollout undo ds ds-one --to-revision=1
kubectl describe pod ds-one-<tab> | grep Image:
```

Change the updateStrategy to OnDelete:

```
kubectl edit ds ds-one

  updateStrategy:
    #rollingUpdate:   <-- Delete these 2 lines
    #  maxUnavailable: 1
    type: OnDelete    <-- Change this line
```

Check the rollout history and undo to revision 2. Check the Image:, still 1.10.1. After delete one pod, the Image of the new Pod will be 1.12.1-alpine

```
kubectl rollout history ds ds-one
kubectl rollout undo ds ds-one --to-revision=2
kubectl describe pod ds-one-<tab> | grep Image:    # <-- still 1.10.1
kubectl delete pod ds-one-<tab>
kubectl describe pod ds-one-<tab> | grep Image:    # <-- new pod should be 1.12.1-alpine
```

2. Know various ways to configure applications

Configuring Command and Arguments on applications

- Create a pod (Pod name: ubuntu-sleeper) with the ubuntu image to run a container to sleep for 5000 seconds.
- Create a pod with the given specifications. By default it displays a 'blue' background. Set the given command line arguments to change it to 'green' (Pod Name: webapp-green, Image: kodekloud/webapp-color, Command line arguments: --color=green)

show

- Using kubectl run with --dry-run option to create yaml file, then add command to the yaml file.

```
kubectl run ubuntu-sleeper --generator=run-pod/v1 --image=ubuntu --dry-run -o yaml > ubuntu-sleeper.yaml
vi ubuntu-sleeper.yaml

spec:
    containers:
    - image: ubuntu
      name: ubuntu-sleeper
      command:
      - sleep
      - "5000"
```

- Using kubectl run with --dry-run option to create yaml file, then add command to the yaml file.

```
kubectl run webapp-green --generator=run-pod/v1 --image=kodekloud/webapp-color --dry-run -o yaml > webapp.yaml
vi webapp.yaml
kubectl create -f webapp.yaml

spec:
    containers:
    - image: kodekloud/webapp-color
      name: webapp-green
      args:
      - --color
      - "green"
```

Configuring Environment Variables

- Create a pod with the given specifications.
  - Pod name: print-greeting, image: ubuntu,
  - 3 env variables GREETING, HONORIFIC, and NAME are set to Warm greetings to, The Most Honorable, and Kubernetes, respectively,
  - run in the bash shell the command: echo $(GREETING) $(HONORIFIC) $(NAME), then sleep 5000.
- Setting env variables using ConfigMap:
  - create a configmap name family with father=lkd, mother=tn, son=bob
  - create a pod name shell-demo, image: nginx, env name (iam) from the key father of configmap family.
  - use envFrom to define all of the ConfigMap's data as Pod environment variables

show

• Using kubectl run with --dry-run option to create yaml file, then add command to the yaml file.

```
kubectl run print-greeting --generator=run-pod/v1 --image=bash --dry-run -o yaml > envars.yaml
vi envars.yaml
kubectl create -f envars.yaml

spec:
    containers:
    - image: ubuntu
      name: print-greeting
      resources: {}
      env:
      - name: GREETING
        value: Warm greetings to
      - name: HONORIFIC
        value: The Most Honorable
      - name: NAME
        value: Kubernetes
      command: ["/bin/bash"]
      args: ["-c", "echo $(GREETING) $(HONORIFIC) $(NAME); sleep 5000"]
```

Check the output

```
kubectl logs print-greeting
```

• Setting env variables using ConfigMap:

```
kubectl create configmap family --from-literal=father=lkd --from-literal=mother=tn --from-literal=son=bob
kubectl run shell-demo --generator=run-pod/v1 --image=nginx --dry-run -o yaml > config-app.yaml
vi config-app.yaml

spec:
containers:
- image: nginx
    name: shell-demo
    env:
    - name: iam
      valueFrom:
        configMapKeyRef:
          name: family
          key: father
```

Create the pod, and check output:

```
kubectl create -f config-app.yaml
kubectl exec shell-demo -it -- /bin/bash -c 'echo $iam'
```

Modify the env:

```
spec:
containers:
- image: nginx
    name: shell-demo
    envFrom:
    - configMapRef:
       name: family
```

Delete and recreate the pod, check the output:

```
kubectl delete pod shell-demo
kubectl create -f config-app.yaml
kubectl exec shell-demo -it -- /bin/bash -c 'env'
```

Configuring Secrets

A tutorial of how to use and config Secrets is here.

- Create a secret name db-user-password with username=admin, password=kd248asid9sasp, then decode the secret db-user-password
- Using Secrets as Files from a Pod
    - Create a pod name nginx, image=nginx
    - Make the Secret db-user-password available to the Pod as a mounted volume at /etc/db_confidentials
    - Verify the volume exits

show

- Create the secret and decode.

```
kubectl create secret generic db-user-password --from-literal=user=admin --from-literal=password=kd248asid9sasp
kubectl get secret db-user-password -o yaml

apiVersion: v1
data:
    password: a2QyNDhhc2lkOXNhc3A=
    user: YWRtaW4=
kind: Secret
metadata:
    name: db-user-password
type: Opaque
```

Decode the password field:

```
echo 'a2QyNDhhc2lkOXNhc3A=' | base64 --decode
```

- Using Secrets as Files from a Pod
    - Using kubectl run with --dry-run option to create yaml file, then add command to the yaml file.

```
kubectl run nginx --generator=run-pod/v1 --image=nginx --dry-run -o yaml > secret.yaml
vi secret.yaml

spec:
  containers:
  - image: nginx
    name: nginx
    resources: {}
    volumeMounts:
    - name: pass-vol
      mountPath: /etc/db_confidentials
  volumes:
  - name: pass-vol
    secret:
      secretName: db-user-password
```

Verify the volume exits

```
kubectl exec -it nginx -- /bin/bash -c 'df -ha | grep db_confi'
kubectl exec -it nginx -- /bin/bash -c 'ls /etc/db_confidentials'
```

Multi-container Pod

- Create a multi-container pod with 2 containers (Name: yellow, Container 1 Name: lemon, Container 1 Image: busybox, Container 2 Name: gold ,Container 2 Image: redis).

show

```bash kubectl run yellow --generator=run-pod/v1 --image=busybox --dry-run -o yaml > yellow.yaml vi yellow.yaml ``` ```yaml apiVersion: v1 kind: Pod metadata: labels: run: yellow name: yellow spec: containers: - image: busybox name: lemon - image: redis name: gold ```

InitContainers

- Create the pod name red using nginx image,
- Update the pod red to use an initContainer that uses the busybox image and sleeps for 20 seconds

show
```

```
kubectl run red --generator=run-pod/v1 --image=nginx --dry-run -o yaml > init-red.yaml
vi init-red.yaml

spec:
  containers:
  - image: nginx
    name: red
    resources: {}
  initContainers:
  - image: init-container-red
    name: busybox
    command: ["sleep 20"]
```

3. Know how to scale applications

Create the Deployment with nginx image, scale to replicas=3

show

```
kubectl create deployment nginx --image=nginx
kubectl scale deployment nginx --replicas=3
```

Change the deployment to autoscale, with target CPU utilization set to 80% and the number of replicas between 2 and 5.

show

```
kubectl autoscale deployment nginx --min=2 --max=5 --cpu-percent=80
kubectl get hpa
```

See the hpa settings:

```
kubectl describe hpa
```

You can notice that the warning:

```
ScalingActive  False   FailedGetResourceMetric  the HPA was unable to compute the replica count: unable to get metrics for resource cpu: unable to fetch metrics from resource metrics API: the server could not find the requested resource (get pods.metrics.k8s.io)
```

You need to install metrics-server.

```
git clone https://github.com/kubernetes-incubator/metrics-serverhttps://github.com/kubernetes-incubator/metrics-server
kubectl create -f metrics-server/deploy/1.8+/
# edit metric-server deployment to add the flags
# args:
# - --kubelet-insecure-tls
# - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
$ kubectl edit deploy -n kube-system metrics-server
```

metric-server-args

4. Understand the primitives necessary to create a self-healing application

- Kubernetes supports self-healing applications through ReplicaSets and Replication Controllers. The replication controller helps in ensuring that a POD is re-created automatically when the application within the POD crashes. It helps in ensuring enough replicas of the application are running at all times.

- Kubernetes provides additional support to check the health of applications running within PODs and take necessary actions through Liveness and Readiness Probes.