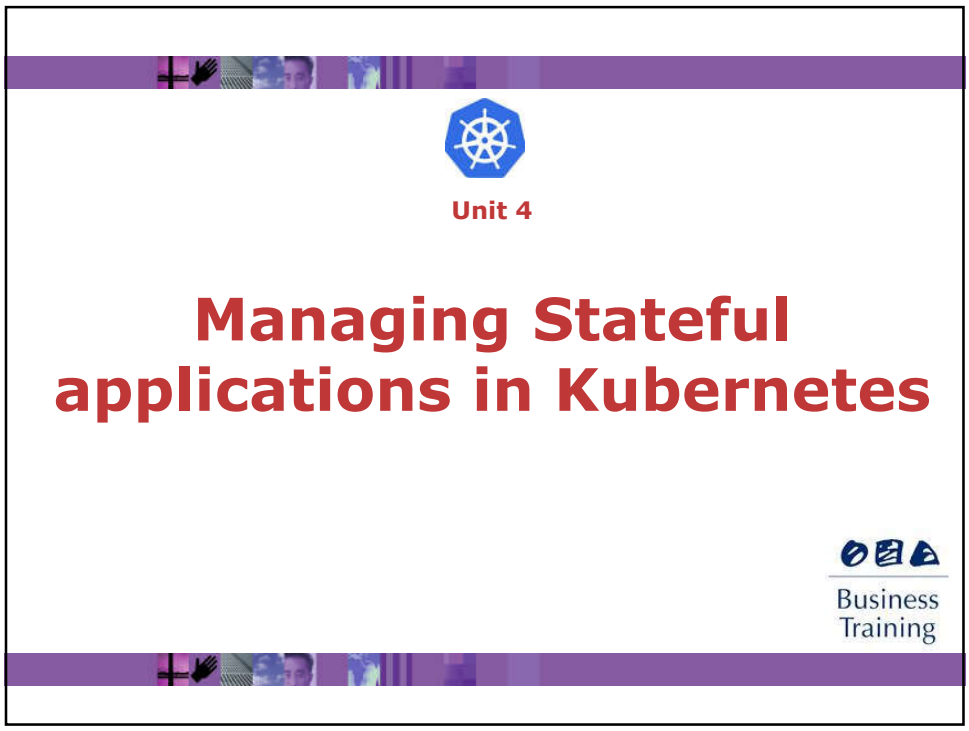


1

Outline

- Kubernetes Volumes
- ConfigMaps, Secrets, and StatefulStets
- Storage classes
- Persistent Volumes (PVs) and Persistent Volume Claims (PVCs)

2

Kubernetes Volumes

3

Volumes

- **Volumes are special directories that are mounted in containers**
- **Volumes can have many different purposes:**
  - share files and directories between containers running on the same machine
  - share files and directories between containers and their host
  - centralize configuration information in Kubernetes and expose it to containers
  - manage credentials and secrets and expose them securely to containers
  - store persistent data for stateful services
  - access storage systems (like EBS, NFS, Portworx, and many others)
- **Kubernetes volumes vs. Docker volumes**
  - Kubernetes and Docker volumes are very similar
  - Docker volumes allow us to share data between containers running on the same host
  - Kubernetes volumes allow us to share data between containers in the same pod

4

Managing stateful applications

Volumes vs. Persistent Volumes

- **Volumes and Persistent Volumes are related, but very different!**
- **Volumes:**
  - appear in Pod specifications (we'll see that in a few slides)
  - do not exist as API resources (cannot do kubectl get volumes)
- **Persistent Volumes:**
  - are API resources (can do kubectl get persistentvolumes)
  - correspond to concrete volumes (e.g. on a SAN, EBS, etc.)
  - cannot be associated with a Pod directly; but through a Persistent Volume Claim

5

Managing stateful applications

Adding a volume to a Pod

- **We add a volume to Pod manifest**
  - This will mount that volume in a container in the Pod
- **By default, this volume will be an emptyDir(an empty directory)**
  - It will "shadow" the directory where it's mounted
- **We can add the volume in two places:**
  - at the Pod level (to declare the volume)
  - at the container level (to mount the volume)

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-volume
spec:
  volumes:
  - name: www
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: www
      mountPath: /usr/share/nginx/html/
```


6

Sharing a volume between two containers

- **The volume `www` is added at the pod level**
- **We have 2 containers**
  - Ngnix mounts www to /usr/share/nginx/html
  - Git mounts www to /www/
- **As a result, Nginx now serves this website**

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-git
spec:
  volumes:
  - name: www
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: www
      mountPath: /usr/share/nginx/html/
  - name: git
    image: alpine
    command: [ "sh", "-
c", "apk add git && git clone https://g
ithub.com/octocat/Spoon-Knife /www" ]
    volumeMounts:
    - name: www
      mountPath: /www/
  restartPolicy: OnFailure
```


7

Init Containers

- **In the previous example, there is a short period of time during which the website is not available**
  - because the git container hasn't done its job yet !
- **We can define containers that should execute before the main ones**
  - They will be executed in order(instead of in parallel)
  - They must all succeed before the main containers are started
- **Other uses of init containers**
  - Load content
  - Generate configuration (or certificates)
  - Database migrations
  - Waiting for other services to be up

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-init
spec:
  volumes:
  - name: www
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: www
      mountPath: /usr/share/nginx/html/
  initContainers:
  - name: git
    image: alpine
    command: [ "sh", "-
c", "apk add git && git clone https://g
ithub.com/octocat/Spoon-Knife /www" ]
    volumeMounts:
    - name: www
      mountPath: /www/
```


8

ConfigMaps, Secrets, and StatefulStets

9

Configuration Pattern

- **Kubernetes has an integrated pattern for decoupling configuration from application or container**
  - This pattern makes use of two Kubernetes components:
    - **ConfigMaps**
    - **Secrets**

10

Managing stateful applications

ConfigMaps

- **Externalized data stored within kubernetes.**
- **Can be referenced through several different means:**
  - environment variable
  - a command line argument (via env var)
  - injected as a file into a volume mount
- **Can be created from a manifest, literals, directories, or files directly.**

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: manifest-example
data:
  state: Belgium
  city: Brussels
  content: |
    Look at this,
    its multiline!
```

- **Imperative style:**
  - `$ kubectl create configmap literal-example --from-literal="city=Brussels" --from-literal=state=Belgium`
  - `$ kubectl create configmap file-example --from-file=cm/city --from-file=cm/state`

M.Romdhani, 2020 11

11

Managing stateful applications

Secrets

- **Functionally identical to a ConfigMap.**
- **Stored as base64 encoded content.**
- **Encrypted at rest within etcd (if configured!).**
- **Stored on each worker node in tmpfs directory.**
- **Ideal for username/passwords, certificates or other sensitive information that should not be stored in a container.**

M.Romdhani, 2020 12

12

Managing stateful applications

Secrets

- **type: There are three different types of secrets within Kubernetes:**
  - **docker-registry** - credentials used to authenticate to a container registry
  - **generic/Opaque** - literal values from different sources
  - **tls** - a certificate based secret
- **data: Contains key-value pairs of base64 encoded content.**

```
apiVersion: v1
kind: Secret
metadata:
  name: manifest-secret
type: Opaque
data:
  username: S3ViZXJuZXRlcw==
  password: cGFzc3dvcmQ=
```

- **Imperative style:**
  - `$ kubectl create secret generic literal-secret --from-literal=username=administrator --from-literal=password=password`
  - `kubectl create secret generic file-secret --from-file=secret/username --from-file=secret/password`

13

Managing stateful applications

Injecting ConfigMaps and Secrets

- **Injecting as environment variable**

ConfigMap

```
apiVersion: batch/v1
kind: Job
metadata:
  name: cm-env-example
spec:
  template:
    spec:
      containers:
        - name: mypod
          image: alpine:latest
          command: ["/bin/sh", "-c"]
          args: ["printenv CITY"]
          env:
            - name: CITY
              valueFrom:
                configMapKeyRef:
                  name: manifest-example
                  key: city
      restartPolicy: Never
```

Secret

```
apiVersion: batch/v1
kind: Job
metadata:
  name: secret-env-example
spec:
  template:
    spec:
      containers:
      - name: mypod
        image: alpine:latest
        command: ["/bin/sh", "-c"]
        args: ["printenv USERNAME"]
        env:
        - name: USERNAME
          valueFrom:
            secretKeyRef:
              name: manifest-example
              key: username
      restartPolicy: Never
```


14

Managing stateful applications

Injecting ConfigMaps and Secrets

- Injecting in a command

ConfigMap

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: cm-env-example
spec:
  template:
    spec:
      containers:
        - name: mypod
          image: alpine:latest
          command: ["/bin/sh", "-c"]
          args: ["echo Hello ${CITY}!"]
          env:
            - name: CITY
              valueFrom:
                configMapKeyRef:
                  name: manifest-example
                  key: city
      restartPolicy: Never
```

Secret

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: secret-env-example
spec:
  template:
    spec:
      containers:
      - name: mypod
        image: alpine:latest
        command: ["/bin/sh", "-c"]
        args: ["echo Hello ${USERNAME}!"]
        env:
        - name: USERNAME
          valueFrom:
            secretKeyRef:
              name: manifest-example
              key: username
      restartPolicy: Never
```


15

Managing stateful applications

Injecting ConfigMaps and Secrets

- Injecting as a Volume

ConfigMap

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: cm-vol-example
spec:
  template:
    spec:
      containers:
      - name: mypod
        image: alpine:latest
        command: ["/bin/sh", "-c"]
        args: ["cat /myconfig/city"]
        volumeMounts:
        - name: config-volume
          mountPath: /myconfig
      restartPolicy: Never
      volumes:
      - name: config-volume
        configMap:
          name: manifest-example
```

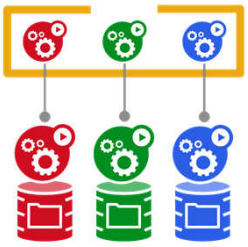
Secret

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: secret-vol-example
spec:
  template:
    spec:
      containers:
      - name: mypod
        image: alpine:latest
        command: ["/bin/sh", "-c"]
        args: ["cat /mysecret/username"]
        volumeMounts:
        - name: secret-volume
          mountPath: /mysecret
      restartPolicy: Never
      volumes:
      - name: secret-volume
        secret:
          secretName: manifest-example
```


16

Statefulsets

- **Tailored to managing Pods that must persist or maintain state.**
- **Pod lifecycle will be ordered and follow consistent patterns.**
  - Assigned a unique ordinal name following the convention of '<statefulset name>-<ordinal index>'.
- **At a first glance, they look like deployments, but they have some significant differences**
  - a stateful set defines a pod spec and a number of replicas R
  - it will make sure that R copies of the pod are running
  - updating the pod spec will cause a rolling update to happen



17

Statefulsets

- **StatefulSet Deployments provide:**
  - Stable, unique network identifiers
  - Stable, persistent storage
  - Ordered, graceful deployment and scaling
  - Ordered, automated rolling updates

18

StatefulSet features

- Pods in a stateful set are numbered (from 0 to R-1) and ordered
- They are started and updated in order (from 0 to R-1)
- A pod is started (or updated) only when the previous one is ready
- They are stopped in reverse order (from R-1 to 0)
- Each pod know its identity (i.e. which number it is in the set)
- Each pod can discover the IP address of the others easily
- The pods can persist data on attached volumes

19

Why not using just Volumes ?

- We can attach volumes to pods and deployments, but there are some shortcomings
  - If a Deployment uses a volume, all replicas end up using the same volume
  - That volume must then support concurrent access
  - Their lifecycle (creation, deletion...) is managed outside of the Kubernetes API
- What we really need is a way for each replica to have its own volume
  - The Pods of a Stateful set can have individual volumes
    - in a Stateful set with 3 replicas, there will be 3 volumes
    - This introduces a bunch of new Kubernetes resource types: Persistent Volumes, Persistent Volume Claims, Storage Classes

20

Managing stateful applications

StatefulSet manifest

- **serviceName**:
  - The name of the associated headless service; or a service without a **ClusterIP**.
  - serviceName is first unique aspect of statefulsets
    - maps to the name of a service you create along with the StatefulSet that helps provide pod network identity
    - Headless service, or a service without a ClusterIP. No NAT'ing, No load balancing.
    - It will contain a list of the pod endpoints and provides a mechanism for unique DNS names
- **revisionHistoryLimit**:
  - The number of previous iterations of the StatefulSet to retain (as in Deployments and DaemonSets).

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sts-example
spec:
  replicas: 2
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: stateful
  serviceName: app
  updateStrategy:
   type: RollingUpdate
   rollingUpdate:
     partition: 0
  template:
    <pod template>
```


21

Managing stateful applications

Statefulset, a recap

- **A Stateful sets manages a number of identical pods (like a Deployment)**
- **These pods are numbered, and started/upgraded/stopped in a specific order**
- **These pods are aware of their number (e.g., #0 can decide to be the primary, and #1 can be secondary)**
- **These pods can find the IP addresses of the other pods in the set (through a headless service)**
- **These pods can each have their own persistent storage (Deployments cannot do that)**

22

Managing stateful applications

Headless service

- A headless service is a service with a service IP but instead of load-balancing it will return the IPs of our associated Pods.
- This allows us to interact directly with the Pods instead of a proxy. It's as simple as specifying None for .spec.clusterIP and can be utilized with or without selectors - you'll see an example with selectors in a moment.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-headless-service
spec:
  clusterIP: None # <--
  selector:
    app: test-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
```


23

Managing stateful applications

Headless service Example

Deployment of 5 pods

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-deployment
  labels:
    app: api
spec:
  replicas: 5
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
      - name: api
        image: eddiehale/hellonodeapi
        ports:
        - containerPort: 3000
```

A Regular Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: normal-service
spec:
  selector:
    app: api
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
```

A Headless Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: headless-service
spec:
  clusterIP: None # <-- Don't forget!!
  selector:
    app: api
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
```

- Hop in the cluster and run nslookup
  - nslookup normal-service, will show one entry
  - nslookup headless-service will show five entries

24

Storage classes

25

What are Storage Classes ?

- **Storage classes are an abstraction on top of a external storage resource that has dynamic provisioning capability**
  - Work hand-in-hand with the external storage system to enable dynamic provisioning of storage by eliminating the need for the cluster admin to pre-provision a volume
- **A StorageClass provides a way for administrators to describe the "classes" of storage they offer.**
  - Each StorageClass contains the fields provisioner, parameters, and reclaimPolicy, which are used when a PersistentVolume belonging to the class needs to be dynamically provisioned.

26

Managing stateful applications

StorageClass manifest

- **provisioner: Defines the 'driver' to be used for provisioning of the external storage.**
- **parameters: A hash of the various configuration parameters for the provisioner.**
- **reclaimPolicy: The behaviour for the backing storage when the PVC is deleted.**
  - **Retain** - manual clean-up
  - **Delete** - storage asset deleted by provider

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 1Mi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/C/tmp/data"
```

M.Romdhani, 2020 27

27

Managing stateful applications

Available Storage Options

- **Kubernet storage options :**
  - Node Local storage : **emptyDir** (initially empty, the same lifecycle as the pod), **hostPath** (file or directory from the host node's filesystem into your pod)
  - File-sharing types such as nfs
  - Cloud provider-specific types like awsElasticBlockStore, azureDisk, VsphereVolume, PortworxVolume, gcePersistentDisk
  - Distributed file system types, for example glusterfs or cephfs
- **More are supported through plugins**

M.Romdhani, 2020 28

28

Persistent Volumes (PVs) and Persistent Volume Claims (PVCs)

29

What is a Persistent Volume

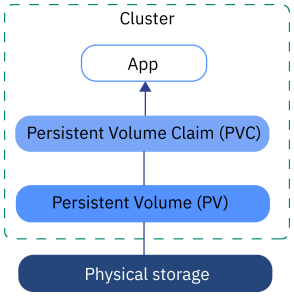
- **A PersistentVolume (PV) represents a storage resource.**
  - Containers are ephemeral constructs. Any changes to the running container is **lost** when the container stops running. PV are there to persist data for containers and Pods.
- **PVs are a cluster wide resource linked to a backing storage provider: NFS, GCEPersistentDisk, RBD etc.**
  - Generally provisioned by an administrator.
- **Their lifecycle is handled independently from a pod**
- **CANNOT be attached to a Pod directly. Relies on a PersistentVolumeClaim (PVC)**
  - The way a user consumes a PV is by creating a PVC.

30

Managing stateful applications

PVs vs. PVCs

- **Persistent Volume (PV) –**
  - It is a piece of network storage that has been provisioned by the administrator. It's a resource in the cluster which is independent of any individual pod that uses the PV.
- **Persistent Volume Claim (PVC)**
  - It is a request for storage by a user that can be fulfilled by a PV.
- **PVs and PVCs are independent from Pod lifecycles and preserve data through restarting, rescheduling, and even deleting Pods.**



31

Managing stateful applications

Configure a Pod to Use a PersistentVolume for Storage

- **Step 1 - Create a PersistentVolume**
  - The configuration file specifies that the volume is at `/mnt/data` on the cluster's Node
  - View information about the PersistentVolume:

    `kubectl get pv task-pv-volume`

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 1Mi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/C/tmp/data"
```

- **Step 2 - Create a PersistentVolumeClaim**
  - This **PersistentVolumeClaim** requests a volume of at 1 megabytes that can provide read-write access for at least one Node
  - Look again at the PersistentVolume:

    `kubectl get pv task-pv-volume`

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Mi
```


32

Configure a Pod to Use a PersistentVolume for Storage

- **Step 3 - Create a Pod**
  - Notice that the Pod's configuration file specifies a PersistentVolumeClaim, but it does not specify a PersistentVolume. From the Pod's point of view, the claim is a volume.
  - Verify that the container in the Pod is running:

    ```
    kubectl get pod task-pv-pod
    ```
  - Initialize the index.html page from within C:\tmp\data

    ```
    echo "Bonjour, Bonjour ..."
    >/usr/share/nginx/html/index.html
    ```
    - Check that /usr/share/nginx/html contains `index.html` having the right content.
  - Get a shell to the container running in your Pod:

    ```
    kubectl exec -it task-pv-pod --
    /bin/bash
    ```

```
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
        claimName: task-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        -
 mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```


33

Configure a Pod to Use a PersistentVolume for Storage

- **Step 3 - Create a Pod (Continued)**
  - From within the pod shell, run the following commands

    ```
    # Be sure to run these 3 commands inside the root shell that comes from
    # running "kubectl exec" in the previous step
    apt update
    apt install -y curl
    curl http://localhost/
    ```
    - The curl output shows the text that you wrote to the **index.html** file on the hostPath volume (the string **`"Bonjour, Bonjour ..."`**)
- **Step 4 – Clean Up**
  - Delete the Pod, the PersistentVolumeClaim and the PersistentVolume:

    ```
    kubectl delete pod task-pv-pod
    kubectl delete pvc task-pv-claim
    kubectl delete pv task-pv-volume
    ```
  - In the shell on your Node, remove the file and directory that you created:

    ```
    rm /mnt/data/index.html
    rmdir /mnt/data
    ```


34