Unit 3

Configuration and resource Management

Business Training

1

Outline

- ConfigMaps and Secrets
- Introduction to Kubernetes Volumes
- PersisentVolumes and PersistentVolumeClaims
- Managing resources for Containers and Pods

2

ConfigMaps and Secrets

3

The Configuration pattern

- **The 3rd factor (Configuration) of the Twelve-Factor App principles states:**
  - ***Configuration that varies between deployments should be stored in the environment.***
- **Kubernetes has an integrated pattern for decoupling configuration from application or container**
  - This pattern makes use of two Kubernetes components:
    - **ConfigMaps**
    - **Secrets**

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.**

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: manifest-example
data:
  state: Belgium
  city: Brussels
  content: |
    Brussels hosts many
    EU institutions
```

- **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`

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.**

Configuration and resource management

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.**

```yaml
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`

M.Romdhani, 2022

7

7

Configuration and resource management

Injecting ConfigMaps and Secrets

- **Injecting as environment variable**

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: ["printenv 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: ["printenv USERNAME"]
        env:
        - name: USERNAME
          valueFrom:
            secretKeyRef:
              name: manifest-example
              key: username
      restartPolicy: Never
```

M.Romdhani, 2022

8

8

Injecting ConfigMaps and Secrets

- **Injecting in a command**

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: ["echo Hello ${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: ["echo Hello ${USERNAME}!"]
        env:
        - name: USERNAME
          valueFrom:
            secretKeyRef:
              name: manifest-example
              key: username
      restartPolicy: Never
```


Injecting ConfigMaps and Secrets

- **Injecting as a Volume**

ConfigMap

```
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

```
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
```

Introduction to Kubernetes Volumes

11

Volumes

- **A Volume is a Kubernetes capability that persists data beyond a Pod restart. Essentially, a Volume is a directory that's shareable between multiple containers of a Pod.**
  - 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)

12

Configuration and resource management

Volumes vs. Persistent Volumes

- **Persistent Volumes are a specific category of the wider concept of Volumes. The mechanics for Persistent Volumes are slightly more complex.**
- **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

M.Romdhani, 2022 13

13

Configuration and resource management

Volume Types

- **The Kubernetes documentation offers a long list of Volume types.**
  - Some of the types—for example, azureDisk, awsElasticBlockStore, or gcePersistentDisk—are only available when running the Kubernetes cluster in a specific cloud provid
  - Examples of Volume Types

Type	Description
emptyDir	Empty directory in Pod with read/write access. Only persisted for the lifespan of a Pod. A good choice for cache implementations or data exchange between containers of a Pod.
hostPath	File or directory from the host node’s filesystem.
configMap, secret	Provides a way to inject configuration data. For practical examples, see the previous section of this unit.
nfs	An existing NFS (Network File System) share. Preserves data after Pod restart.
persistentVolumeClaim	Claims a Persistent Volume.

M.Romdhani, 2022 14

14

emptyDir vs. hostPath

- **An emptyDir volume is first created when a Pod is assigned to a Node, and exists as long as that Pod is running on that node.**
  - As the name says, it is initially empty.
  - All Containers in the same Pod can read and write in the same emptyDir volume.
  - When a Pod is restarted or removed, the data in the emptyDir is lost forever.
- **A hostPath volume mounts a file or directory from the host node's filesystem into your pod.**
  - This is not something that most Pods will need, but it offers a powerful escape hatch for some applications.
  - For example, some uses for a hostPath are:
    - running a container that needs access to Docker internals; use a hostPath of /var/lib/docker
    - running cAdvisor in a container; use a hostPath of /dev/cgroups

Sharing a volume between two containers using an emptyDir

- **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
    emptyDir: {}
  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
```

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

17

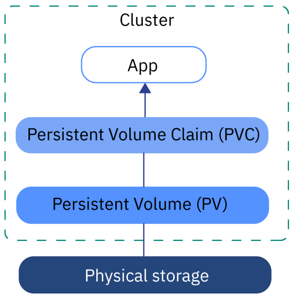
What is a Persistent Volume

- **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.

18

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.**



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
```

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

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

Configure a Pod to Use a PersistentVolume for Storage

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

```
kubectl get pod task-pv-pod
```

  - Initialize the index.html page from with /tmp/data

```
echo "Bonjour, Bonjour ..."
>/usr/share/nginx/html/index.html
```

    - Check that /usr/share/nginx/html contains `index.html` having the ri content.
  - Get a shell to the container running in your Pod:

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

```yaml
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
```


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
```

Managing resources for Containers and Pods

23

Why managing resources is important ?

- **Within Kubernetes, containers are scheduled as pods. By default, a pod in Kubernetes will run with no limits on CPU and memory in a default namespace.**
  - This can create several problems related to contention for resources.
  - Setting Kubernetes requests and limits effectively has a major impact on application performance, stability, and cost. And yet working with many teams over the past year has shown us that determining the right values for these parameters is hard.
- **Limits**
  - **Resource limits help the Kubernetes scheduler better handle resource contention.**
    - When a Pod uses more memory than its limit, its processes will be killed by the kernel to protect other applications in the cluster. Pods will be CPU throttled when they exceed their CPU limit.
    - If no limit is set, then the pods can use excess memory and CPU when available.

24

Configuration and resource management

Why managing resources is important ?

- **When setting CPU and memory limits, the tradeoffs are similar but not quite the same.**
  - For example, setting the aggregated amount of CPU limits higher than the allocated number of CPUs exposes applications to potential throttling risk. Provisioning additional CPUs (i.e. increase spend) is one potential answer while reducing CPU limits for certain applications (i.e. increase throttling risk) is another.

Limit	Too low	Too high
CPU	CPU throttling	Starve other Pods or resource inefficiency
Memory	Killed by kernel	Starve other Pods or resource inefficiency

M.Romdhani, 2022 25

25

Configuration and resource management

Determining the right values

- **When setting requests, start by determining the acceptable probability of a container's usage exceeding its request in a specific time window, e.g. 24 hours. To predict this in future periods, we can analyze historical resource usage.**
- **You can classify applications into different availability tiers and apply these rules of thumb for targeting the appropriate level of availability**

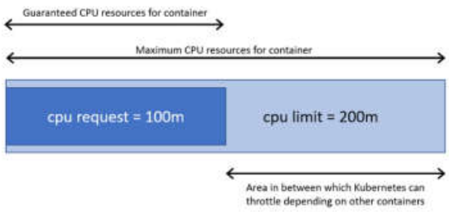
M.Romdhani, 2022 26

26

Managing Resources for Containers

- **When you specify a Pod, you can optionally specify how much of each resource a Container needs.**
  - The most common resources to specify are CPU and memory (RAM)
- **Requests and Limits**
  - The **requests** is the amount guaranteed by the control plane.
    - Requests affect scheduling decisions !
  - The **limits** are "hard limits". The container is not allowed to use more of that resource than the limit.

```
resources:
  requests:
    cpu: 100m
    memory: 300Mi
  limits:
    cpu: 1
    memory: 300Mi
```



Pod quality of service

- **Each pod is assigned a QoS class (visible in status.qosClass).**
  - **If limits = requests**:
    - as long as the container uses less than the limit, it won't be affected
    - If all containers in a pod have (limits=requests), QoS is considered "***Guaranteed***"
  - **If requests < limits**:
    - as long as the container uses less than the request, it won't be affected
    - otherwise, it might be killed/evicted if the node gets overloaded
    - if at least one container has (requests<limits), QoS is considered "***Burstable***"
  - **If a pod doesn't have specified any request nor limit**, QoS is considered "*BestEffort*"
- **When a node is overloaded, BestEffort pods are killed first. Then, Burstable pods that exceed their limits.**
- **If we only use Guaranteed pods, no pod should ever be killed (as long as they stay within their limits)**

Configuration and resource management

Managing Resources for Containers

- **Meaning of CPU units**
  - One cpu, in Kubernetes, is equivalent to 1 vCPU/Core for cloud providers and 1 hyperthread on bare-metal Intel processors.
  - Fractional requests are allowed. The expression 0.1 is equivalent to the expression 100m, which can be read as "one hundred millicpu"
- **Meaning of Memory units**
  - You can express memory as a plain integer or as a fixed-point integer using one of these suffixes: E, P, T, G, M, K. You can also use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki.
- **The following Pod has two Containers.**
  - Each Container has a request of **0.25 cpu and 64MiB** and a limit of **0.5 cpu and 128MiB** of memory.

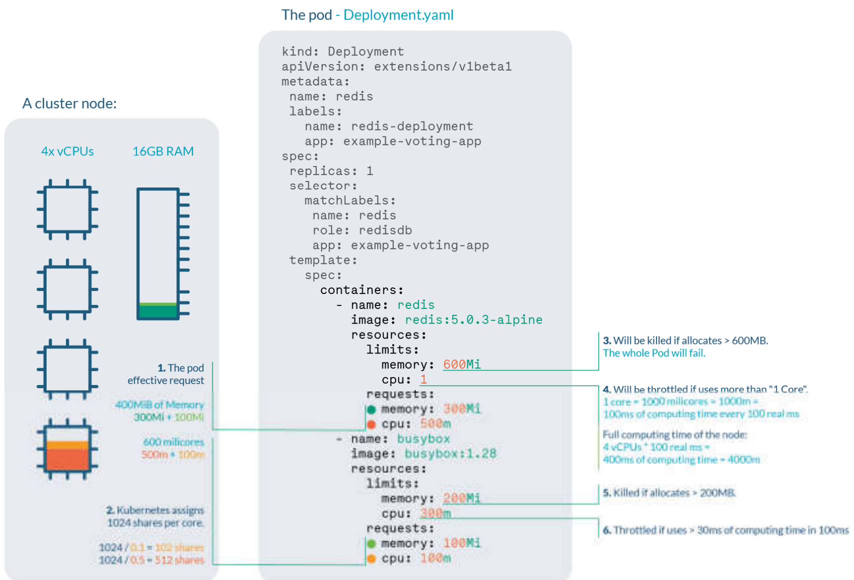
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: "password"
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

M.Romdhani, 2022

29

Configuration and resource management

Managing Resources for Containers




```yaml
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
 name: redis
 labels:
   name: redis-deployment
   app: example-voting-app
spec:
 replicas: 1
 selector:
   matchLabels:
    name: redis
    role: redisdb
    app: example-voting-app
 template:
   spec:
     containers:
       - name: redis
         image: redis:5.0.3-alpine
         resources:
           limits:
             memory: 600Mi
             cpu: 1
           requests:
             memory: 300Mi
             cpu: 500m
       - name: busybox
         image: busybox:1.28
         resources:
           limits:
             memory: 200Mi
             cpu: 300m
           requests:
             memory: 100Mi
             cpu: 100m
```

Source: https://sysdig.com/blog/kubernetes-limits-requests/

30

30

Requests and Limits default values

- **If we specify a limit without a request, the request is set to the limit.**
- **If we specify a request without a limit, there will be no limit (which means that the limit will be the size of the node)**
  - Unless there are default values defined for our namespace!
- **If we don't specify anything, the request is zero and the limit is the size of the node.**
  - This is generally not what we want. A container without a limit can use up all the resources of a node
  - if the request is zero, the scheduler can't make a smart placement decision.

Defining min, max, and default resources using LimitRange

- **We can create LimitRange objects to indicate any combination of:**
  - min and/or max resources allowed per pod
  - default resource limits
  - default resource requests
  - maximal burst ratio (limit/request)
- **LimitRange objects are namespaced**
- **They apply to their namespace only**

```
apiVersion: v1
kind: LimitRange
metadata:
  name: my-very-detailed-limitrange
spec:
  limits:
  - type: Container
    min:
      cpu: "100m"
    max:
      cpu: "2000m"
      memory: "1Gi"
    default:
      cpu: "500m"
      memory: "250Mi"
    defaultRequest:
      cpu: "500m"
```

Namespace Quotas

- Quotas are enforced by creating a ResourceQuota object
- ResourceQuota is for limiting the total resource consumption of a namespace
- If we can have multiple ResourceQuota objects in the same namespace, the most restrictive values are used

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: a-little-bit-of-compute
spec:
  hard:
    configmaps: "10"
    secrets: "10"
    services: "10"
    requests.cpu: "10"
    requests.memory: 10Gi
    limits.cpu: "900"
    limits.memory: 20Gi
```

- When a ResourceQuota is created, we can see how much of it is used:

```
kubectl describe resourcequota my-resource-quota
```