Unit 3

# Configuration

Business Training

1



## Outline

- ConfigMaps and Secrets
- Managing Resources for Pods and Containers



2

# ConfigMaps and Secrets

3



## Configuration Pattern

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



4

# 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: |
    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`







# 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.**

# 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

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

- **data: Contains key-value pairs of base64 encoded content.**

- **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`





# 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
```

# 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
```

# Managing resources for Containers and Pods

11



## Why managing resources is important ?

- **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.



12

## Why managing resources is important ?

- **The Tradeoffs**
  - Determining the right level for requests and limits is about managing trade-offs, as shown in the following tables.
  - **When setting requests**, there is inherently a tradeoff between the cost of running an application and the performance/outage risk for this application. Balancing these risks depends on the relative cost of extra CPU/RAM compared to the expected cost of an application throttle or outage event.
    - For example, if allocating another 1 Gb of RAM ($5 cost) reduces the risk of an application outage event ($10,000 cost) by 1% then it would be worth the additional cost of these compute resources.

| Request | Too low | Too high |
|---|---|---|
| CPU | Starvation – may not get CPU cycles needed | Inefficiency – requires extra CPUs to schedule other Pods |
| Memory | Kill risk – may be terminated if other pods need memory | Inefficiency – requires extra RAM to schedule other Pods |







## Why managing resources is important ?

- **When setting limits, the tradeoffs are similar but not quite the same.**
  - The tradeoff here is the relative performance of individual applications on your shared infrastructure vs the total cost of running these applications.
  - 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 |

## Why managing resources is important ?

- **Determining the right valuesPermalink**
  - 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:







## Managing Resources for Containers

- **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.
- **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 amout 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)**







## Don’t use Memory Swap !

- **The semantics of memory and swap limits on Linux cgroups are complex**
  - In particular, it's not possible to disable swap for a cgroup (the closest option is to reduce "swappiness")
- **The architects of Kubernetes wanted to ensure that Guaranteed pods never swap**
  - The only solution was to disable swap entirely !
- **If you don't care that pods are swapping, you can enable swap**

Configuration

# 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.

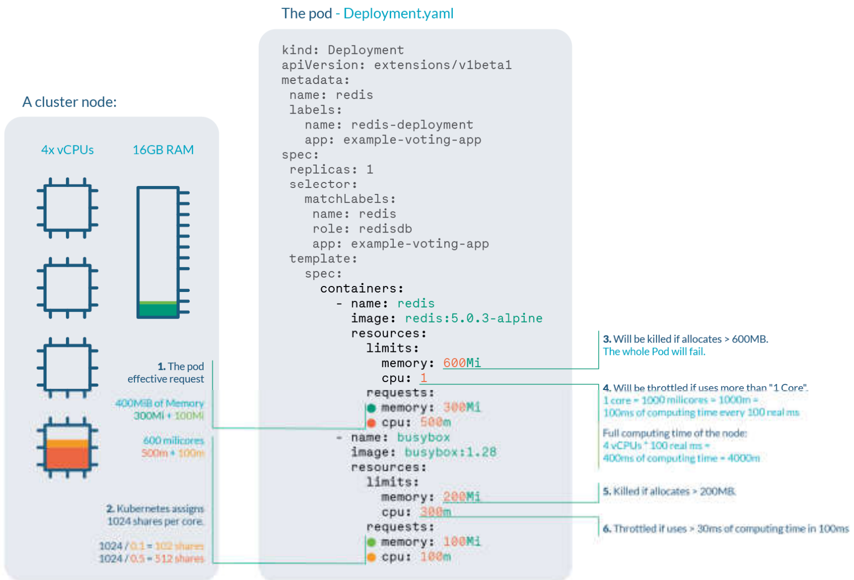
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: "password"
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"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

*M.Romdhani, 2021*

19

Configuration

# Managing Resources for Containers




20

20

# 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**

```yaml
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

```
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
```