# Selfie Shots - Tools 304 Kubernetes

Persistent University

# Additional Kubernetes Objects

# Key learning points :

- Secrets
- ConfigMaps
- DaemonSets
- StatefulSets
- Jobs
- RBAC
- Autoscaling
- Labels

## Secrets

- Pods can access local data using volumes, but there is some data you don't want readable to the naked eye. Passwords may be an example. Someone reading through a YAML file may read a password and remember it. Using the *Secret* API resource, the same password could be encoded. A casual reading would not give away the password. You can create, get, or delete secrets:

```
$ kubectl get secrets
```

- Secrets can be manually encoded with **kubectl create secret**:

```
$ kubectl create secret generic --help
```

```
$ kubectl create secret generic mysql --from-literal=password=root
```

- A secret is not encrypted, only **base64**-encoded. You can see the encoded string inside the secret with **kubectl**. The secret will be decoded and be presented as a string saved to a file. The file can be used as an environmental variable or in a new directory, similar to the presentation of a volume.

## Secrets (Cntd.)

- A secret can be made manually as well, then inserted into a YAML file:

```
$ echo LFTr@1n | base64
TEZUckAxbgo=
```

```
$ vim secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: LF-secret
data:
  password: TEZUckAxbgo=
```

- An alpha feature in v1.7.0 is the type **EncryptionConfig** which allows for encryption at rest of secrets. There are four types, checked in configuration order, each attempting to decrypt the data using pre-populated keys. Encryption uses the first type and first key listed in the configuration file. Currently, **AES-CBC** is considered to be the strongest, but slowest encryption.

# Using Secrets via Environment Variables

- A secret can be used as an environmental variable in a Pod. You can see one being configured in the following example:
- ...

```
spec:
  containers:
  - image: mysql:5.5
    env:
    - name: MYSQL_ROOT_PASSWORD
      valueFrom:
          secretKeyRef:
              name: mysql
              key: password
        name: mysql
```

- There is no limit to the number of Secrets used, but there is a 1MB limit to their size. Each secret occupies memory, along with other API objects, so very large numbers of secrets could deplete memory on a host.
- They are stored in the **tmpfs** storage on the host node, and are only sent to the host running Pod. All volumes requested by a Pod must be mounted before the containers within the Pod are started. So, a secret must exist prior to being requested.

# Mounting Secrets as Volumes

- You can also mount secrets as files using a volume definition in a pod manifest. The mount path will contain a file whose name will be the key of the secret created with the **kubectl create secret** step earlier.

```
...
spec:
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    volumeMounts:
```

## Mounting Secrets as Volumes (Cntd.)

```
- mountPath: /mysqlpassword
    name: mysql
  name: busy
volumes:
- name: mysql
  secret:
    secretName: mysql
```

- Once the pod is running, you can verify that the secret is indeed accessible in the container:

```
$ kubectl exec -ti busybox -- cat /mysqlpassword/password
LFTr@1n
```

# Portable Data with ConfigMaps

- A similar API resource to Secrets is the *ConfigMap*, except the data is not encoded. In keeping with the concept of decoupling in Kubernetes, using a *ConfigMap* decouples a container image from configuration artifacts.
- They store data as sets of key-value pairs or plain configuration files in any format. The data can come from a collection of files or all files in a directory. It can also be populated from a literal value.
- A *ConfigMap* can be used in several different ways. A Pod can use the data as environmental variables from one or more sources. The values contained inside can be passed to commands inside the pod. A Volume or a file in a Volume can be created, including different names and particular access modes. In addition, cluster components like controllers can use the data.
- Let's say you have a file on your local filesystem called **config.js**. You can create a *ConfigMap* that contains this file. The **configmap** object will have a **data** section containing the content of the file:

```
$ kubectl get configmap foobar -o yaml
kind: ConfigMap
apiVersion: v1
metadata:
   name: foobar
data:
   config.js: |
      {
...
```

# ConfigMaps (contd.)

- ConfigMaps can be consumed in various ways:
- Pod environmental variables from single or multiple ConfigMaps
- Use ConfigMap values in Pod commands
- Populate Volume from ConfigMap
- Add ConfigMap data to specific path in Volume
- Set file names and access mode in Volume from ConfigMap data
- Can be used by system components and controllers.

## Using ConfigMaps

- Like secrets, you can use *ConfigMaps* as environment variables or using a volume mount. They must exist prior to being used by a Pod, unless marked as **optional**. They also reside in a specific namespace.
- In the case of environment variables, your pod manifest will use the **valueFrom** key and the **configMapKeyRef** value to read the values. For instance:

```
env:
- name: SPECIAL_LEVEL_KEY
  valueFrom:
    configMapKeyRef:
      name: special-config
      key: special.how
```

- With volumes, you define a volume with the **configMap** type in your pod and mount it where it needs to be used.

```
volumes:
    - name: config-volume
      configMap:
        name: special-config
```

# DaemonSets

- A newer object to work with is the DaemonSet. This controller ensures that a single pod exists on each node in the cluster. Every Pod uses the same image. Should a new node be added, the DaemonSet controller will deploy a new Pod on your behalf. Should a node be removed, the controller will delete the Pod also.
- The use of a DaemonSet allows for ensuring a particular container is always running. In a large and dynamic environment, it can be helpful to have a logging or metric generation application on every node without an administrator remembering to deploy that application.
- Use **kind: DaemonSet**.
- As usual, you get all the CRUD operations via **kubectl**:

**$ kubectl get daemonsets**

**$ kubectl get ds**

# StatefulSets

- According to Kubernetes documentation, StatefulSet is the workload API object used to manage stateful applications. Pods deployed using a **StatefulSet** use the same Pod specification. How this is different than a Deployment is that a **StatefulSet** considers each Pod as unique and provides ordering to Pod deployment.
- In order to track each Pod as a unique object, they get an identity composed of stable storage, stable network identity, and an ordinal. This identity remains with the node, regardless to which node the Pod is running on at any one time.
- The default deployment scheme is sequential, starting with 0, such as **app-0**, **app-1**, **app-2**, etc. A following Pod will not launch until the current Pod reaches a running and ready state. They are not deployed in parallel.
- **StatefulSets** are stable as of Kubernetes v1.9.

# Jobs

- Jobs are part of the **batch** API group. They are used to run a set number of pods to completion. If a pod fails, it will be restarted until the number of completion is reached.
- While they can be seen as a way to do batch processing in Kubernetes, they can also be used to run one-off pods. A *Job* specification will have a parallelism and a completion key. If omitted, they will be set to one. If they are present, the parallelism number will set the number of pods that can run concurrently, and the completion number will set how many pods need to run successfully for the *Job* itself to be considered done. Several *Job* patterns can be implemented, like a traditional work queue.
- *Cronjobs* work in a similar manner to Linux jobs, with the same time syntax. There are some cases where a job would not be run during a time period or could run twice; as a result, the requested Pod should be idempotent.
- An option **spec** field is **.spec.concurrencyPolicy** which determines how to handle existing jobs, should the time segment expire. If set to **Allow**, the default, another concurrent job will be run. If set to **Forbid**, the current job continues and the new job is skipped. A value of **Replace** cancels the current job and starts a new job in its place.

# Role Based Access Control (RBAC)

- The last API resources that we will look at are in the **rbac.authorization.k8s.io** group. We actually have four resources: *ClusterRole*, *Role*, *ClusterRoleBinding*, and *RoleBinding*. They are used for **Role Based Access Control** (RBAC) to **Kubernetes**.

```
$ curl localhost:8080/apis/rbac.authorization.k8s.io/v1beta1
...
    "groupVersion": "rbac.authorization.k8s.io/v1beta1",
    "resources": [
...
        "kind": "ClusterRoleBinding"
...
        "kind": "ClusterRole"
...
        "kind": "RoleBinding"
...
        "kind": "Role"
...
```

- These resources allow us to define Roles within a cluster and associate users to these Roles. For example, we can define a Role for someone who can only read pods in a specific namespace, or a Role that can create deployments, but no services. We will talk more about RBAC later in the course.

## Autoscaling

- In the autoscaling group we find the **Horizontal Pod Autoscalers** (**HPA**). This is a stable resource. HPAs automatically scale *Replication Controllers*, *ReplicaSets*, or *Deployments* based on a target of 50% CPU usage by default. The usage is checked by the kubelet every 30 seconds, and retrieved by Heapster every minute. HPA checks with Heapster every 30 seconds. Should a Pod be added or removed, HPA waits 180 seconds before further action.

- Other metrics can be used and queried via REST. The autoscaler does not collect the metrics, it only makes a request for the aggregated information and increases or decreases the number of replicas to match the configuration.

- The **Cluster Autoscaler** (**CA**) adds or removes nodes to the cluster, based on the inability to deploy a Pod or having nodes with low utilization for at least 10 minutes. This allows dynamic requests of resources from the cloud provider and minimizes expenses for unused nodes. If you are using CA, nodes should be added and removed through c**luster-autoscaler-** commands. Scale-up and down of nodes is checked every 10 seconds, but decisions are made on a node every 10 minutes. Should a scale-down fail, the group will be rechecked in 3 minutes, with the failing node being eligible in five minutes. The total time to allocate a new node is largely dependent on the cloud provider.

- Another project still under development is the **Vertical Pod Autoscaler**. This component will adjust the amount of CPU and memory requested by Pods.

## Scaling and Rolling Updates

- The API server allows for the configurations settings to be updated for most values. There are some immutable values, which may be different depending on the version of Kubernetes you have deployed.
- A common update is to change the number of replicas running. If this number is set to zero, there would be no containers, but there would still be a *ReplicaSet* and *Deployment*. This is the backend process when a *Deployment* is deleted.

```
$ kubectl scale deploy/dev-web --replicas=4
deployment "dev-web" scaled
```

```
$ kubectl get deployments
NAME     DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
dev-web  4        4        4           1          12m
```

- Non-immutable values can be edited via a text editor, as well. Use **edit** to trigger an update. For example, to change the deployed version of the **nginx** web server to an older version:

```
$ kubectl edit deployment nginx
....
      containers:
      - image: nginx:1.8 #<<---Set to an older version
        imagePullPolicy: IfNotPresent
        name: dev-web
....
```

- This would trigger a rolling update of the deployment. While the deployment would show an older age, a review of the Pods would show a recent update and older version of the web server application deployed.

## Labels

- Part of the metadata of an object is a *label*. Though labels are not API objects, they are an important tool for cluster administration. They can be used to select an object based on an arbitrary string, regardless of the object type. Labels are immutable as of API version **apps/v1**.
- Every resource can contain labels in its metadata. By default, creating a *Deployment* with **kubectl run** adds a label, as we saw in:

```
....
    labels:
        pod-template-hash: "3378155678"
        run: ghost
....
```

- You could then view labels in new columns:

```
$ kubectl get pods -l run=ghost
NAME                    READY STATUS  RESTARTS AGE
ghost-3378155678-eq5i6 1/1   Running 0        10m

$ kubectl get pods -Lrun
NAME                    READY STATUS  RESTARTS AGE RUN
ghost-3378155678-eq5i6 1/1   Running 0        10m ghost
nginx-3771699605-4v27e 1/1   Running 1         1h nginx
```

# Labels

- While you typically define labels in pod templates and in the specifications of Deployments, you can also add labels on the fly:

```
$ kubectl label pods ghost-3378155678-eq5i6 foo=bar

$ kubectl get pods --show-labels
NAME                    READY STATUS  RESTARTS AGE LABELS
ghost-3378155678-eq5i6 1/1   Running 0        11m foo=bar, pod-template-hash=3378155678,run=ghost
```

- For example, if you want to force the scheduling of a pod on a specific node, you can use a *nodeSelector* in a pod definition, add specific labels to certain nodes in your cluster and use those labels in the pod.

```
....
spec:
  containers:
  - image: nginx
  nodeSelector:
    disktype: ssd
```

# Summary

At the end of this session, we see that you are now able to

- Configure secrets and ConfigMaps.
- Understand DaemonSets, StatefulSets, Jobs, RBAC
- Understand autoscaling
- Understand Labels

# Lab Exercise

- Create following and use with existing deployment
  - Secrets
  - ConfigMaps
  - Horizontal Pod Autoscaler
  - Label