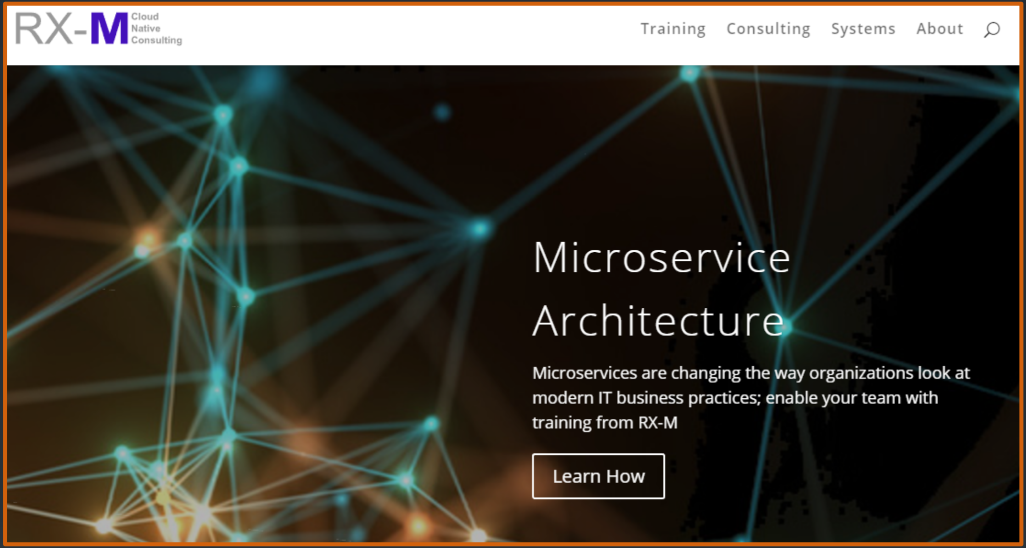
RX-M Cloud Native Consulting

Kubernetes

Foundation

RX-M Cloud Native Advisory, Consulting and Training

- Microservice Oriented
  - Microservices Foundation [3 Day]
  - Building Microservices on AWS [3 Day]
  - Building Microservices with Go [3 Day]
  - Building Microservices with Thrift [2 Day]
  - Building Microservices with gRPC [2 Day]
- Container Packaged
  - Docker Foundation [3 Day]
  - Docker Advanced [2 Day]
  - OCI [2 Day]
  - CNI [2 Day]
  - Containerd [2 Day]
  - Rocket [2 Day]
- Dynamically Managed
  - Cloud Native Container Networking and Cisco ACI [3 Day]
  - Docker Orchestration (Compose/Swarm) [2 Day]
  - Kubernetes Foundation [2 Day]
  - Kubernetes Advanced [3 Day]
  - Mesos Foundation [2 Day]
  - MANTL [2 Day]
  - Nomad [2 Day]



Microservice Architecture

Microservices are changing the way organizations look at modern IT business practices; enable your team with training from RX-M

5: Controllers

Objectives

- Understanding labels and selectors and how they are used to identify resources in the cluster
- Explore the types, roles and workings of Controllers
  - Deployments, Replica Sets, Replication Controllers, DaemonSets, Jobs, CronJobs, StatefulSets, PetSets, HorizontalPodAutoscaler
    - Understand the differences between RSs & RCs
- Explore additional spec config options
  - Policies
  - Health and readiness
  - Pre and post actions

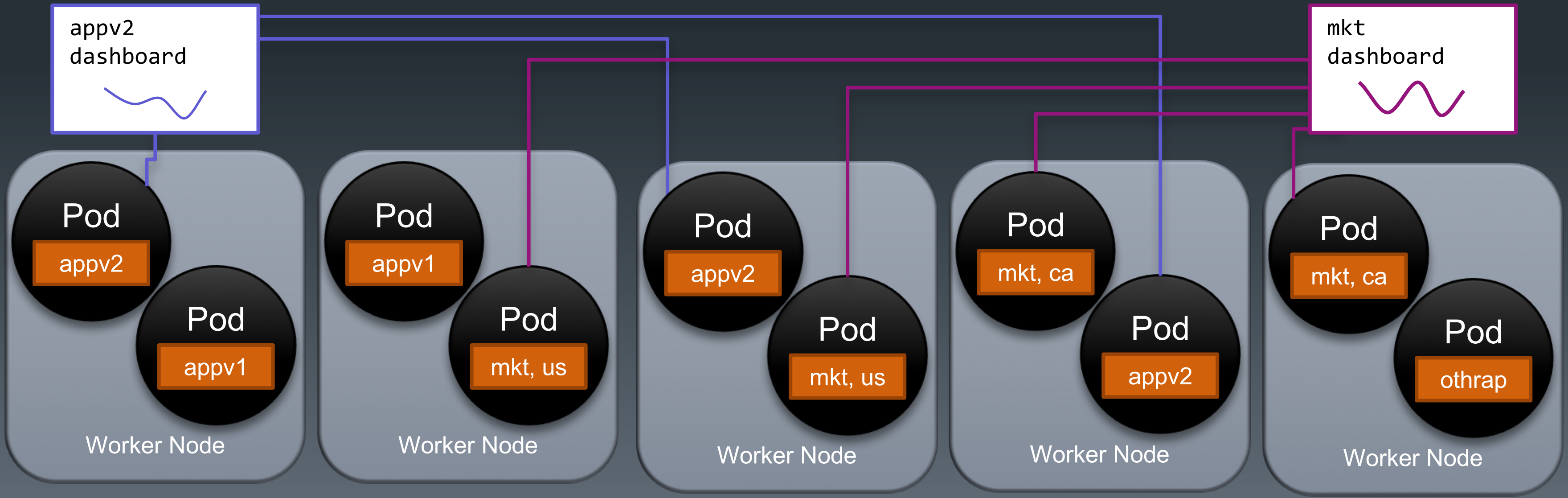
Controller Types

- While pods can be created directly, in a cluster they are almost always created through a Controller
- Deployments – for pods that back long running services with support for upgrades and scaling
  - Deployment object describes a desired state
    - Deployment controller changes the actual state to the desired state at a controlled rate
- Replication Controller (RC) – ensures that a specified number of Pod replicas are running at any one time
- Replica Set (RS) – the next-generation Replication Controller
  - Deployments recommended instead of directly using ReplicaSets, unless you require custom update orchestration or don't require updates
    - You may never need to manipulate ReplicaSet objects
  - Only difference between a RS and a RC right now is the selector support
- Daemon Set – ensures that all (or some) Nodes run a copy of a Pod
- Job – for one shot pods used to complete a start to finish task
- CronJob – manages time-based Jobs
- StatefulSets – for pods with an identity tied to state storage in volumes or other backends
  - PetSets – renamed StatefulSet starting in Kubernetes version 1.5
- HorizontalPodAutoscaler – automatically scales the number of pods in a RC, Deployment or RS based on observed metrics

Controllers use Labels & Selectors to manage Pods

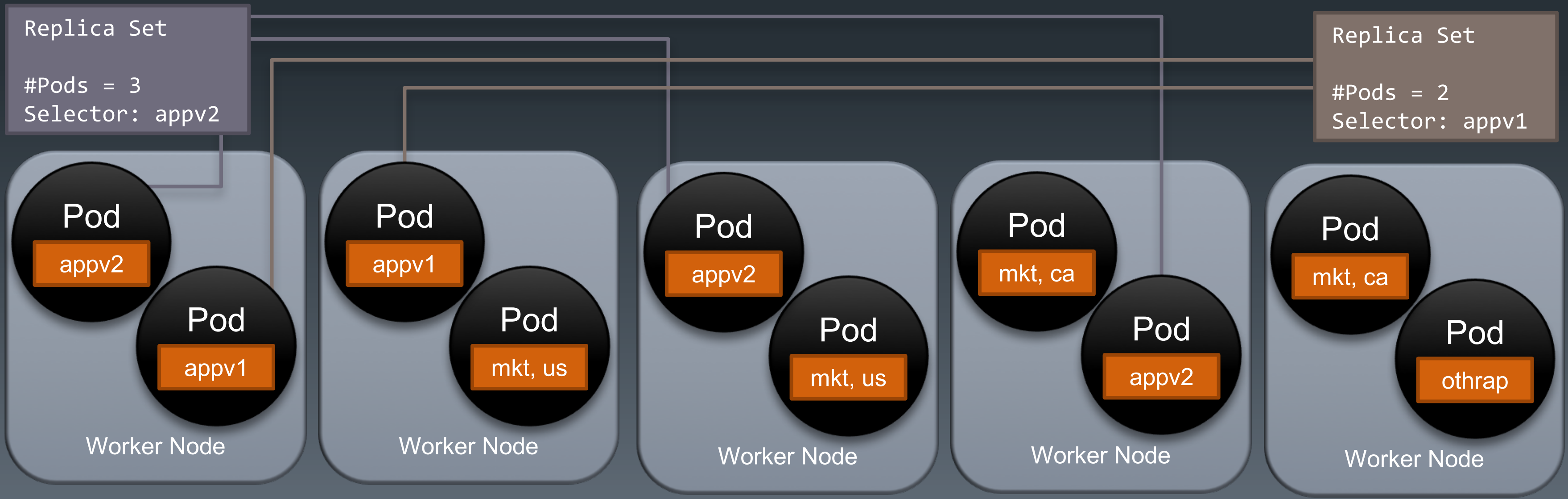
Labels

- From one to many
  - When working on a single physical node, tools generally don't operate on containers in bulk
  - When moving to a cluster you want to easily scale out across nodes, requiring you to work in terms of sets of things instead of singletons
  - You want to keep those sets similarly configured
- In Kubernetes sets of pods are managed using two concepts:
  - labels
  - Replica Sets
- Every pod in Kubernetes has a set of key/value pairs associated with it called labels
- You can select a set of pods by constructing a query based on labels
- Kubernetes has no opinion on the "correct" way to organize pods
  - It is up to you to organize your pods in a way that makes sense to you
  - You can organize by application tiers, geographic location, development environment etc.
  - Labels are non-hierarchical, you can organize your pods in multiple ways simultaneously



Replica Sets & Selectors

- A Replica Set maintains a pool of pods based on a desired replication count, a pod template defined in a Deployment and a label selector/query
  - e.g. if you wanted to run a frontend tier with 3 pods, you would create a Deployment with an appropriate pod template (pointing at your container image) and a `replicas` count of 3
  - You would identify the set of pods that the Replica Set is managing with a label query, for example `env=prod,tier=fe`
- The Replica Set takes an easy to understand desired state and tirelessly works to make it true
- If you want to scale in or out all you have to do is change the desired replication count



```
Replica Set

#Pods = 3
Selector: appv2
```

```
Replica Set

#Pods = 2
Selector: appv1
```

Selectors

- There are several options you can use to select on labels:
  - `=` or `==`
    - You can use either style to select keys with values equal to the string on the right
    - `name = apache`
  - `!=`
    - Select keys with values that do not equal the string on the right
    - `environment != test`
  - `in`
    - Select resources whose labels have keys with values in this set
    - `tier in (web,app)`
  - `notin`
    - Select resources whose labels have keys with values not in this set
    - `tier not in (lb,app)`
  - <Key name>
    - Use a key name only to select resources whose labels contain this key
    - `tier`
- Multiple selectors can be separated by commas and are logically "AND"ed
- Many kubectl subcommands offer the `-l` switch to specify label selectors (get, delete, etc.)

```
$ kubectl describe deployment logger
...
Selector:   deployment=demo,app=logger
...
$ kubectl get pod -l "deployment"
NAME          READY     STATUS     RESTARTS    AGE
logger-84ocd  1/1       Running    0           4m
logger-tc068  1/1       Running    0           4m
logger-zgtj9  1/1       Running    0           4m
$ kubectl get pod -l "deployment=demo"
NAME          READY     STATUS     RESTARTS    AGE
logger-84ocd  1/1       Running    0           4m
logger-tc068  1/1       Running    0           4m
logger-zgtj9  1/1       Running    0           4m
$ kubectl get pod -l "deployment=prod"
No resources found.
$ kubectl get pod -l "deployment!=prod"
NAME          READY     STATUS     RESTARTS    AGE
logger-84ocd  1/1       Running    0           5m
logger-tc068  1/1       Running    0           5m
logger-zgtj9  1/1       Running    0           5m
$ kubectl get pod -l "deployment in (prod,demo)"
NAME          READY     STATUS     RESTARTS    AGE
logger-84ocd  1/1       Running    0           4m
logger-tc068  1/1       Running    0           4m
logger-zgtj9  1/1       Running    0           4m
$ kubectl get pod -l "deployment in (prod,demo), app=logger"
NAME          READY     STATUS     RESTARTS    AGE
logger-84ocd  1/1       Running    0           9m
logger-tc068  1/1       Running    0           9m
logger-zgtj9  1/1       Running    0           9m
```

Deployment Config

- The four first-level elements for a Deployment config are common among all top-level Kubernetes resource definitions
  - `kind` - in this case, set to Deployment
  - `apiVersion` - set to apps/v1beta1 for the Deployment kind
  - `metadata` - labels
  - `spec` - the Deployment spec (see spec reference URL below)
- Keys related to the Deployment kind
  - `replicas:`
    - Defines the desired number of pods
  - `selector:`
    - Optional field that specifies a label selector for the Pods targeted by this deployment
      - Tells the Replica Set which pods to watch
      - If unspecified, `.spec.selector.matchLabels` defaults to `.spec.template.metadata.labels`
    - Assigning selectors that cause more than one RS to attempt to control a Pod is a ***user error***
      - K8s tries to avoid oscillation but can not always
  - `template:`
    - Defines a template to launch a new pod
    - Contains the same elements defined for pod configs
- Note that, if used, selector values need to match the labels values specified in the pod template

```yaml
apiVersion: apps/v1beta1
# before 1.6 use extensions/v1beta1
kind: Deployment
metadata:
  name: logger
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: logger
        deployment: demo
    spec:
      containers:
      - name: logger
        image: logagg:latest
        ports:
        - containerPort: 80
```

https://kubernetes.io/docs/resources-reference/v1.7/#deployment-v1beta1

Deployment/Replication Resources

API v1 [2015/7/10]

- Available since Kubernetes v0.x
- kubectl rolling-update
  - Imperative command used to migrate from one set of pod images to another
  - No support for rollback
- ReplicationController
  - Abbreviation: rc
  - Created by kubectl run in Kubernetes v1.0
  - Maintains a defined number of pod replicas
  - Spec includes:
    - replicas
    - selector
    - template (PodSpec) [required]
  - Will be deprecated in the future and ultimately removed

A Deployment that configures a ReplicaSet is now the recommended way to set up replication

Extension API v1Beta1 [2015/11/09]

- Available since Kubernetes v1.1
- Deployment
  - Abbreviation: deploy
  - Created by kubectl run in Kubernetes v1.1+
  - Declarative way to define the deployment of a set of pod images
  - Spec includes:
    - replicas
    - selector
    - template (PodSpec) [required]
    - Also:
      - strategy – how old pods will be replaced
      - minReadySeconds – how long after containers are up to wait before making Pod ready
      - revisionHistoryLimit – number of old ReplicaSets to retain (for rollback)
- ReplicaSet
  - Abbreviation: rs
  - Maintains a defined number of pod replicas
  - Used to perform a specific deployment

Rollouts

- ReplicationController
- Rollouts
  - `kubectl create -f abc.yaml`
  - `kubectl rolling-update abc-v1 -f abc-v2.yaml`

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: website
  labels:
    bu: sales
spec:
  replicas: 3
template:
    metadata:
      labels:
        appname: webserver
        deployment: test
      spec:
        containers:
        - name: podweb
          image: nginx
          ports:
          - containerPort: 80
```

- Deployment
- Rollouts
  - `kubectl create -f abc.yaml`
  - `kubectl rollout pause deploy/abc`
  - `kubectl rollout resume deploy/abc`
  - `kubectl rollout status deploy/abc`
  - `kubectl rollout history deploy/abc`
  - `kubectl rollout undo deploy/abc`

```yaml
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: website
  labels:
    bu: sales
spec:
  replicas: 3
  template:
    metadata:
      labels:
        appname: webserver
        deployment: test
      spec:
        containers:
        - name: podweb
          image: nginx
          ports:
          - containerPort: 80
```

Simple Deployment / RS Walk Through

```
$ kubectl create -f mydep.yaml
deployment "logger" created
$ kubectl get deployment
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
logger    3         3         3            3           1m
$ kubectl get rs
NAME                DESIRED   CURRENT   READY     AGE
logger-1072152842   3         3         3         1m
$ kubectl get pod
NAME                      READY     STATUS    RESTARTS   AGE
logger-1072152842-bj5bk   1/1       Running   0          2m
logger-1072152842-fh29f   1/1       Running   0          2m
logger-1072152842-q8cjc   1/1       Running   0          2m
$ kubectl describe deployment logger
Name:                      logger
Namespace:                 default
CreationTimestamp:         Wed, 01 Aug 2017 12:48:13 -0800
Labels:                    app=logger
                           deployment=demo
Annotations:               deployment.kubernetes.io/revision=1
Selector:                  app=logger,deployment=demo
Replicas:                  3 updated | 3 total | 3 available | 0 unavailable
StrategyType:              RollingUpdate
MinReadySeconds:           0
RollingUpdateStrategy:     25% max unavailable, 25% max surge
Pod Template:
  Labels:                  app=logger
                           deployment=demo
  Containers:
   logger:
    Image:                 logagg
    Port:                  80/TCP
    Environment:           <none>
    Mounts:                <none>
  Volumes:                 <none>
Conditions:
  Type                     Status        Reason
  ----                     ------        ------
  Available                True          MinimumReplicasAvailable
 Progressing               True          NewReplicaSetAvailable
OldReplicaSets:            <none>
NewReplicaSet: logger-1072152842 (3/3 replicas created)
Events:
  FirstSeen     LastSeen    Count   From                     SubObjectPath   Type       Reason              Message
  ---------     --------    -----   ----                     -------------   --------   ------              -------
  3m            3m          1       {deployment-controller}                  Normal     ScalingReplicaSet   Scaled up replica set logger-1072152842 to 3
```

```
$ cat mydep.yaml

apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: logger
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: logger
        deployment: demo
    spec:
      containers:
      - name: logger
        image: logagg:latest
        ports:
        - containerPort: 80
```

DaemonSets

- Ensures that all nodes run a copy of a pod
  - As nodes are added to the cluster, pods are added to them
- Good for running:
  - Storage daemons (e.g. glusterd, ceph, …)
  - Log collectors (e.g. fluentd, logstash, …)
  - Node monitoring (Prometheus Node Exporter, cAdvisor, collectd, New Relic agent, Ganglia gmond, …)
- Marking a node unschedulable does not affect DaemonSet controller
- Must have a RestartPolicy equal to Always, or be unspecified, (which defaults to Always)
- Can make pods even when the scheduler has not been started
  - Also great for cluster bootstrap

```yaml
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: monitor
  labels:
    app: monitor
spec:
  template:
    metadata:
      labels:
        app: monitor
    spec:
      containers:
      - name: monitor-container
        image: google/cadvisor
        ports:
          - containerPort: 8080
```

Jobs

- Job
  - Creates one or more pods and ensures that a specified number of them successfully terminate
    - As pods successfully complete, the job tracks the successful completions; when a specified number of successful completions is reached, the job itself is complete
    - Will start a new Pod if the first pod fails or is deleted
  - 3 main types:
    - Non-parallel – one pod is started, job is complete as soon as Pod terminates successfully
    - Parallel with fixed completion count
      - Specify a non-zero positive value for .spec.completions
      - Job is complete when there is one successful pod for each value in the range: 1 to .spec.completions
    - Parallel Jobs with a *work queue* – support parallel processing of a set of independent but related work items
      - Examples: emails to be sent, frames to be rendered, files to be transcoded, ranges of keys in a NoSQL database to scan
      - Not designed to support closely-communicating parallel processes, as commonly found in scientific computing
  - In 1.5 moved to API version `batch/v1.Job` from `extensions/v1beta1.Job` (deprecated)

- CronJob
  - Time-based Job resource: it runs a job periodically on a given schedule, written in Cron format
    - Point in time
    - Repeatedly at point in time
  - Required `schedule` key - takes a Cron format string

```yaml
apiVersion: batch/v2alpha1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

CronJob requires a cluster at v1.5+ with batch/v2alpha1 API turned on by passing **`--runtime-config=batch/v2alpha1`** to the API server

Horizontal Pod Autoscaler

- Allows the number of pods in a replication controller or deployment to scale automatically based on metrics
- Metrics include
  - Observed CPU utilization – scales the number of pods in a Deployment, Replica Set or Replication Controller based on observed CPU utilization
    - Autoscaler periodically queries CPU utilization for the pods it targets
      - Default 30 seconds
        - Period controlled by `--horizontal-pod-autoscaler-sync-period`
    - Requires Heapster monitoring in your cluster for autoscaling to work
  - Custom metrics (alpha)
    - Makes use of the `autoscaling/v2alpha1` API
      - `targetCPUUtilizationPercentage` field replaced with an array called `metrics`
    - For example, queries per second or average request latency
    - Pods must have cAdvisor-specific custom metrics endpoint configured
- Launched via `kubectl autoscale` or YAML spec

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: apache
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
    kind: Deployment
    name: apache
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

Spec Policies

- It is a Kubernetes best practice to always use a Deployment with long running Pods
  - Even single pods with one container benefit from automatic restart by a Replica Set
- The config to the right demonstrates a JSON-based Deployment config
  - imagePullPolicy is a container setting that tells Kubernetes when to pull the image for the container
    - Always (default if :latest tag is specified)
    - Never
    - IfNotPresent (default if :latest tag is not specified)
  - restartPolicy is a Pod setting that applies to all container in the pod
    - Possible values:
      - Always (default)
      - OnFailure
      - Never
    - Failed containers are restarted by the Kubelet
    - Restarts use an exponential back-off delay in multiples of sync-frequency 0, 1x, 2x, 4x, 8x ... capped at 5 minutes and reset after 10 minutes of successful execution
    - A ReplicationController is only appropriate for pods with RestartPolicy = Always
  - dnsPolicy sets the DNS search order with the Kubernetes cluster DNS server first followed by normal container DNS
    - More on cluster DNS in the services section

JSON Deployment example

```
{
  "apiVersion": "apps/v1beta1",
  "kind": "Deployment",
  "metadata": {
    "name": "frontend-controller",
    "labels": {
      "state": "serving"
    }
  },
  "spec": {
    "replicas": 2,
    "template": {
      "metadata": {
        "labels": {
          "app": "frontend"
        }
      },
      "spec": {
        "volumes": null,
        "containers": [
          {
            "name": "my-redis",
            "image": "redis",
            "ports": [
              {
                "containerPort": 80,
                "protocol": "TCP"
              }
            ],
            "imagePullPolicy": "IfNotPresent"
          }
        ],
        "restartPolicy": "Always",
        "dnsPolicy": "ClusterFirst"
      }
    }
  }
}
```

Health Checks

- Kubernetes provides two layers of health checking
  - HTTP or TCP checks
    - K8s can attempt to connect to a particular endpoint and give a status of healthy on a successful connection
  - Application-specific health checks
    - K8s can be configured to use a command line scripts
- An RS/RC will restart a container failing a health check
- livenessProbe
  - Specifies a health check
  - Available options (typically only one of):
    - httpGet
      - Takes a path and port
      - Status codes between 200 and 399 are all considered healthy by the probe
    - tcpSocket
      - Takes a port
      - Any successful connect passes
    - exec
      - Takes a command
      - Exit code of 0 is success
- readinessProbe
  - Uses the same settings but uses test to assess container readiness

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
        livenessProbe: # don't use all at once!
          httpGet:
            path: /status/
            port: 80
          tcpSocket:
            port: 80
          exec:
            command: /usr/bin/health/ck.sh
          initialDelaySeconds: 30
          timeoutSeconds: 1
          periodSeconds: 10
```

Pre/Post Actions

- Containers can be assigned lifecycle actions to run after the container starts and before the container stops
  - Convenient for initializing the started container or cleaning up before the container shutdown
- Management of the container blocks until lifecycle actions are complete
  - Unless the container process fails, in which case the action is aborted
- These hooks are called at least once (be prepared for more than one call!)
- postStart
  - Called immediately after a container is created
  - If the handler fails, the container is terminated and restarted according to its restart policy
  - Other management of the container blocks until the action completes
- preStop
  - Called immediately before a container is terminated
  - The container is terminated after the handler completes
  - The reason for termination is passed to the handler
    - “reason” values:
      - Delete - Delete command was issued via kubectl or the API
      - Health - Health check fails
      - Dependency - Dependency failure (e.g. disk mount failure)
  - Regardless of the outcome of the handler, the container is terminated
  - Other management of the container blocks until the hook completes

```yaml
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: apache-hook
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: apache-hook
    spec:
      containers:
      - name: apache-hook
        image: bitnami/apache:latest
        ports:
        - containerPort: 80
        lifecycle:
          postStart:
            httpGet:
              path: http://my.regsvr.com/register/
              port: 80
          preStop:
            exec:
              command: ["/usr/bin/apachectl","-k", "graceful- stop"]
```

Summary

- Controllers are designed to:
  - Run with a replication factor: Deployments & RSs or RCs
  - Run everywhere: DaemonSets
  - Run once: Jobs
  - Run on a schedule: CronJobs
  - Run stateful apps: StatefulSets (covered later)
- Deployments/ReplicaSets are key components in the orchestration of horizontally-scaled microservices
- RSs/RCs use kubelets to perform on-node monitoring and restart tasks
- Pods can be monitored for health and readiness
- Pods can be configured to perform post-start and pre-shutdown tasks

Lab 5

- Deployments and ReplicaSets

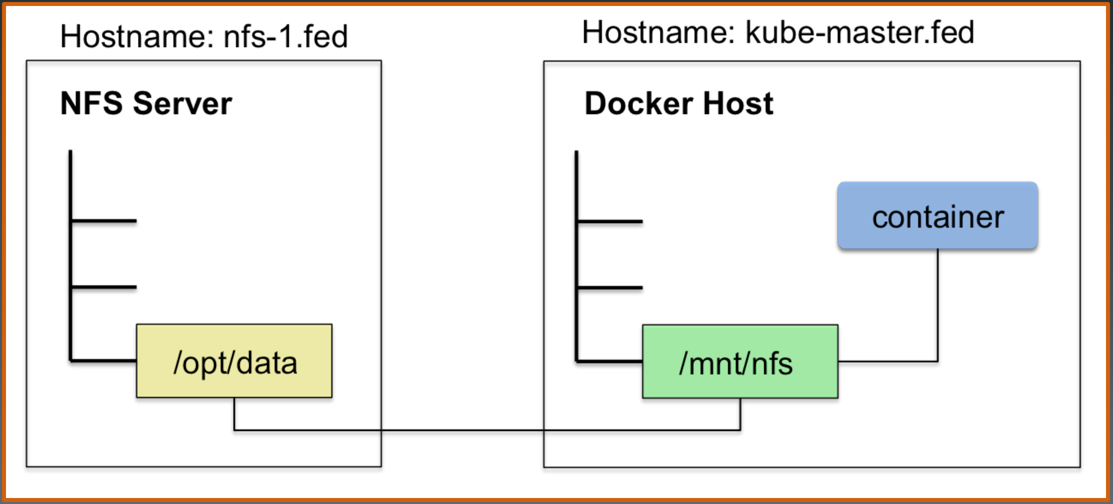
7: Managing State

Objectives

- Explore the lifecycle of volumes in Kubernetes
- Discover the different types of volumes available to Pods and containers in a Kubernetes cluster
- Examine ways to abstract storage providers
- Discuss creating and leveraging secrets in configs
  - Learn about encrypting secrets at rest

Kubernetes Volumes

- A Volume is a container mountable filesystem feature with storage and persistence rules outside of the scope of the container
- Cases for volumes:
  - On-disk files in a container are ephemeral
    - When a container crashes, the kubelet will replace it by rerunning the original image, the files from the dead container will be lost
    - It may be necessary to persist data across container failures
  - When running containers together in a Pod it is often necessary to share files between those containers
    - Storage outside of the container mount namespace can be shared across containers
- A standard Kubernetes volume has a lifetime the same as the pod that encloses it
  - This is different from the Docker volume model, wherein volumes remain until explicitly deleted, regardless of whether there are any running containers using it
- It is important to remember that pods are anchored by the infrastructure container which cannot crash
  - Thus a pod volume outlives any containers that run within the Pod except the pause container preserving volume data across container restarts
  - Only when a Pod is deleted or the node the pod runs on crashes does the volume cease to exist



Volume Types

- Kubernetes supports many type of volumes, and a Pod can use any number of them simultaneously. Volume types supported include:
  - emptyDir – provides the pod with an empty directory (data is deleted when the Pod terminates)
  - hostPath – mounts a given host path into containers using the volume
    - Example: containerized kubelet that needs /var/lib/docker mounted to talk to docker.sock
  - gcePersistentDisk – mounts a Google Compute Engine block storage volume
  - awsElasticBlockStore – mounts an Amazon Web Service EBS volume
  - nfs – mounts an NFS share
  - iscsi – allows an existing iSCSI (SCSI over IP) volume to be mounted
  - fc (fibre channel) – allows an existing fibre channel volume to be mounted in a pod
  - flocker – mounts a flocker volume
  - glusterfs – mounts a glusterfs volume (Glusterfs is an open source networked filesystem)
  - rbd – mounts a Rados Block Device volume
  - cephfs – mounts an existing CephFS volume
  - gitRepo – mounts an empty directory and clones a git repository into it for your pod to use
  - secret – mounts a volume as an in-memory tmpfs (never touches the host disk)
    - Used to pass sensitive information, such as passwords, to pods
  - downwardAPI – mounts pod metadata
  - projected – maps several existing volume sources into the same directory
    - Following types of volume sources can be projected: secret, downwardAPI, configMap
  - azureFileVolume – mounts an Azure File Volume
  - azureDisk – mounts a Microsoft Azure Data Disk
  - vsphereVolume – mounts a vSphere VMDK Volume
  - Quobyte – allows an existing Quobyte volume to be mounted
  - PortworxVolume – elastic block storage layer that runs hyperconverged with Kubernetes
  - ScaleIO – software-based storage platform of scalable shared block networked storage
  - StorageOS – allows an existing StorageOS volume to be mounted into your pod
  - PersistentVolume (PV) – storage provisioned by an administrator aimed at abstracting storage drivers
    - local (alpha in 1.7) – local storage device such as a disk, partition or directory
      - Can only be used as a statically created PersistentVolume

emptyDir

- One of the easiest ways to achieve improved persistence amid container crashes and data sharing within a pod is to use the emptydir volume
- Can be mounted at the same or different paths in each container
- Created when a Pod is assigned to a Node, erased when a Pod is removed
  - Persists beyond container crashes
- Stored on the medium backing the Node (SSD, network storage, etc.)
  - Specifying Memory creates a RAM-backed filesystem
    - Cleared on machine reboot
    - Counts against container memory limits

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-pd-pod
spec:
  containers:
  - image: ngnix:1.13
    ports:
    - containerPort: 80
    name: memory-pd-nginx
    volumeMounts:
    - mountPath: /memory-pd
      name: memory-volume
  volumes:
  - name: memory-volume
    emptyDir:
      medium: Memory
```

hostPath

- Mounts a file or directory from the Node's filesystem into a Pod
- Pods with identical configs may behave differently on different Nodes due to differences in files on the Nodes
- Resource-aware scheduling will not account for resources used by hostPath
- Created directories only writeable by root
  - Config options:
    - Run process as root in privileged container
    - Modify file perms on the host to write to the volume
- Unlike emptyDir, which is erased when a Pod is removed, the contents of hostPath volumes are preserved and the volume is merely unmounted

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  labels:
    app: cadvisor
spec:
  template:
    metadata:
      labels:
        app: cadvisor
    spec:
      containers:
      - name: cadvisor
        image: google/cadvisor
        securityContext:
          privileged: true
        volumeMounts:
          - name: var-run
            mountPath: /var/run
            readOnly: false
          - name: sys
            mountPath: /sys
            readOnly: true
          - name: docker
            mountPath: /var/lib/docker
            readOnly: true
      volumes:
      - name: var-run
        hostPath:
          path: /var/run
      - name: sys
        hostPath:
          path: /sys
      - name: docker
        hostPath:
          path: /var/lib/docker
```

Persistent Disks

- Types:
  - nfs
    - NFS can be mounted by multiple writers simultaneously
  - iscsi
    - Can be mounted as read-only by multiple consumers simultaneously
    - No simultaneous writers allowed
  - fc (fibre channel)
  - flocker
    - Flocker is an open-source clustered container data volume manager that provides orchestration of data volumes backed by a variety of storage backends
  - glusterfs
    - GlusterFS can be mounted by multiple writers simultaneously
  - rbd
    - RBD is can be mounted as read-only by multiple consumers simultaneously but can only be mounted by a single consumer in read-write mode - no simultaneous writers allowed
  - cephfs
    - CephFS can be mounted by multiple writers simultaneously
- A persistent volume that exists outside the container allows one to save important data across containers outages
- Volumes at the pod level can be shared between containers in the same application stack and within the same pod
- Unlike emptyDir, which is erased when a Pod is removed, the contents of nfs, iscsi, fc, flocker, glusterfs, rbd and cephfs volumes are preserved and the volume is merely unmounted

```
apiVersion: v1
kind: Pod
metadata:
  name: flocker-pod
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
      volumeMounts:
          - name: www-root
            mountPath: /usr/share/nginx/html
  volumes:
    - name: www-root
      flocker:
        datasetName: my-flocker-vol

---
apiVersion: v1
kind: Pod
metadata:
  name: fc-pod
spec:
 containers:
   - image: mysql
      name: fc-pod
     volumeMounts:
       - name: fc-vol
         mountPath: /var/lib/mysql
 volumes:
   - name: fc-vol
     fc:
       targetWWNs: ['500a0982991b8dc5', '500a0982891b8dc5']
       lun: 2
       fsType: ext4
       readOnly: true
```

Cloud Volumes

- Cloud vendors (particularly AWS and Google) offering block devices can support named volume mounting
- gcePersistentDisk – volume mounts a Google Compute Engine (GCE) Persistent Disk into your pod
  - Restrictions:
    - Nodes using GCE-PDs must be GCE VMs
    - GCE-PDs need to be in the same project and zone as the cluster
    - You can attach up to 16 persistent disks to most instances and up to 64 TB of total persistent disk space
      - Instances with shared-core machine types or custom machine types with less than 3.75 GB of memory are limited to a maximum of 4 persistent disks and 3 TB of total persistent disk space
      - You can attach up to 128 persistent disks to instances with predefined machine types depending on the number of vCPUs in that instance (Beta)
- awsElasticBlockStore – volume mounts an AWS EBS Volume into your pod
  - Restrictions
    - Nodes using EBS volumes must be AWS EC2 instances
    - Instances must be in the same region and availability-zone as the EBS volume
    - EBS only supports a single EC2 instance mounting a volume
    - Amazon recommends a maximum of 40 EBS volumes per Node with one of those 40 reserved for the root volume
      - Attaching more than 40 volumes can cause boot failures
- You must create a GCE-PD or EBS volume *before* you can use it
- Unlike emptyDir, which is erased when a Pod is removed, the contents of GCE-PD and EBS volumes are preserved and the volume is merely unmounted

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    cloud: test-gce
spec:
  containers:
  - image: ngnix:1.9.1
    name: gce-nginx
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: gce-pd
  volumes:
  - name: gce-pd
    gcePersistentDisk:
      pdName: mysite-volume-1
      fsType: ext4
---
apiVersion: v1
kind: Pod
metadata:
  labels:
    cloud: test-aws
spec:
  containers:
  - image: ngnix:1.13
    name: aws-nginx
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: awsvol
  - name: awsvol
    awsElasticBlockStore:
      volumeID: vol-066c9038349ed7fdd
      fsType: ext4
```

Data Provider Abstraction

- Abstracts details of how storage is provided from how it is consumed
  - PersistentVolume (PV) – piece of storage in the cluster that has been provisioned dynamically or by a cluster admin and is available for consumption
    - Are volume plugins like Volumes but have a lifecycle independent of any individual pod that uses the PV
    - Capture the details of the implementation of the storage: NFS, iSCSI, or cloud-provider-specific storage, etc
  - persistentVolumeClaim (PVC) – PVCs consume PV resources the way Pods consume node resources
    - Claims can request storage sizes and access modes
      - Like Pods, which can request specific levels of resources (CPU and Memory)
    - Pods access storage by using a PVC as a volume
  - StorageClass provides a way for administrators to describe the "classes" of storage they offer
    - Enables a variety of PVs that differ in more ways than just size and access modes, without exposing users to the details of how those volumes are implemented
      - Map to quality-of-service levels, backup policies, or to arbitrary policies determined by cluster admins
      - Kubernetes is unopinionated about what classes represent

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontendpod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: fivegigpvc
  volumes:
    - name: fivegigpvc
      persistentVolumeClaim:
        claimName: fivegigslowpvc
```

Persistent Volumes

- Provisioning
  - Static – cluster admin creates a number of PVs
    - Include the details of the real storage
    - Exist as resources in the API
  - Dynamic – PersistentVolume Provisioner may try to provision a PV when none of the static PVs matches a user's PVC
    - Based on Storage Classes and volume plugin feature availability
- Capacity – PVs will have a storage capacity set using the PV's `capacity` attribute
- Access Modes – can be set to modes supported by a given provider
  - ReadWriteOnce (RWO) – can be mounted as read-write by a single node
  - ReadOnlyMany (ROX) – can be mounted read-only by many nodes
  - ReadWriteMany (RWX) – can be mounted as read-write by many nodes
  - PV can only be mounted using one access mode at a time, even if it supports many
    - ex: GCE-PD can be mounted as ReadWriteOnce by a single node or ReadOnlyMany by many nodes but not at the same time
- Reclaim policy for a PV tells the cluster what to do with the volume after it has been released of its claim, can be:
  - Retained – requires manual reclamation
    - Previous claimant's data remains on the volume until cleaned or the PV is deleted
  - Recycled - performs a scrub (`rm -rf /thevolume/*`) on the volume and makes it available again for a new claim
    - Must use a volume plugin that supports the recycled feature
  - Deleted – removes both the PV object as well as deleting the associated external storage, such as an AWS EBS, GCE PD, Azure Disk, or Cinder volume
    - Dynamically provisioned PVs are always deleted
- Class (optional) – ensures that the PV will only be bound to PVCs requesting the given Storage Class

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: fivegigslowpv
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  nfs:
    path: /tmp
    server: 172.17.0.2
```

Persistent Volume Claims

- Created with a specific amount of storage and an access mode
  - Resources – claims request quantities of storage
  - Access modes – PVCs use the same conventions as PVs
    - Mounting claims with "Many" modes (ROX, RWX) only possible within one namespace
- Class (optional) – only PVs of the requested class can be bound to the PVC
- Selector (optional) – enables granular filtering of available PVs
  - Only volumes with labels that match the selector can be bound to the claim
  - `matchLabels` – volume must have a label with this value
  - `matchExpressions` – list of requirements made by specifying key, a list of values, and an operator that relates the key and values
    - Valid operators: `In`, `NotIn`, `Exists`, `DoesNotExist`
  - Requirements are ANDed together—all must be satisfied to match
- Control loop matches new PVCs with PVs
  - A dynamically-provisioned PV will always bind to the PVC that requested it
- Will remain unbound indefinitely if a matching PV does not exist
  - Claims are bound as matching volumes become available
  - Cluster with many 50Gi PVs would not match any to a 100Gi PVC

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: fivegigslowpvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: slow
  selector:
    matchLabels:
      release: "stable"
    matchExpressions:
      - {key: environment, operator: In, values: [dev]}
```

Storage Classes

- Admins set the name and other parameters of a class when first creating StorageClass objects
  - Cannot be updated once they are created
- Provisioner – determines what volume plugin is used for provisioning PVs
  - ex: GCEPersistentDisk, AWSElasticBlockStore, VsphereVolume, Flocker, etc.
  - Internal provisioners prefixed with `kubernetes.io` and shipped with Kubernetes
  - External provisioners are independent programs that follow a specification defined by Kubernetes
    - ex: NFS doesn't provide an internal provisioner, but an external provisioner can be used
- Parameters – describe attributes of volumes belonging to the storage class
  - Are provisioner specific though some may be similar
    - Best to examine docs of a given provisioner for details on parameters
- Default StorageClass can be specified for PVCs that don't request any particular class to bind to

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  zones: us-central1-a, us-central1-b

---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
```

StatefulSets

- Beta as of 1.5, replacement for PetSets
- Pods in a StatefulSet have a unique ordinal index, a stable network identity, stable storage
  - Ordered deployment, scaling, deletion and termination
    - For a StatefulSet with N replicas, Pods are given a unique index, in order from {0…N-1}
      - K8s v1.7+ uses `podManagementPolicy` to determine ordering
      - `OrderedReady` – the first pod must be "running and ready" prior to the following pods start (Also terminated in reverse order)
      - `Parallel` – launch or terminate all Pods in parallel
    - Each Pod in the StatefulSet will be assigned an integer ordinal that is unique over the Set
  - Unique network identifiers – each Pod in a StatefulSet derives its hostname from the name of the StatefulSet and the ordinal of the Pod
    - Requires the use of a headless service
      - The CNAME of the headless service points to SRV records (one for each Pod)
        - The IP can change so clients should use the SRV
      - kube-proxy never acts on headless services
  - Stable persistent storage – one PV for each VolumeClaimTemplate
    - PVs associated with the Pods' PVCs are not deleted when the Pods, or StatefulSet are deleted & must be done manually

- Update strategies (K8s v1.7+) – configure automated rolling updates
  - `OnDelete` (default) – users must manually delete Pods to cause the controller to create new Pods that reflect modifications made to a StatefulSet's .spec.template
  - `RollingUpdate` – deletes and recreates each Pod in the same order as Pod termination updating each Pod one at a time
    - It will wait until an updated Pod is Running and Ready prior to updating its predecessor
    - Can be partitioned, by specifying a .spec.updateStrategy.rollingUpdate.partition
      - If a partition is specified, all Pods with an ordinal that is greater than or equal to the partition will be updated
      - All Pods with an ordinal that is less than the partition will not be updated
        - If deleted, will be recreated at the previous version
      - Useful if you want to stage an update, roll out a canary, or perform a phased roll out

```yaml
# Headless service
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
  clusterIP: None
  selector:
    app: nginx

---
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: nginx
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: slow
      resources:
        requests:
          storage: 5Gi
```

Secrets

- Kubernetes objects used to hold sensitive information
  - Passwords
  - OAuth tokens
  - ssh keys
- Safer and more flexible than putting it verbatim in a pod definition or in a Docker image
- Users can create secrets
  - The system also creates some secrets
- Can only be referenced by pods in the same namespace where created
- To use a secret, a pod needs to reference the secret
- Each item must be base64 encoded
- Limited to 1MB in size
  - Discourages large secrets which would exhaust apiserver and kubelet memory
- Stored as plaintext in etcd
  - Secret data is at rest on the disk that etcd uses

```
$ echo -n "admin" | base64
YWRtaW4=

$ echo -n "1f2d1e2e67df" | base64
MWYyZDFlMmU2N2Rm
```

```
apiVersion: v1
kind: Secret
metadata:
  name: prod-db-secret
type: Opaque
data:
  password: MWYyZDFlMmU2N2Rm
  username: YWRtaW4=
```

Mounting Secrets

- A secret can be used with a Pod in three ways:
  - As files in a volume
  - Environment variables
  - Used by kubelet when pulling images for the pod
- When a pod is created secret existence is not checked
- If the kubelet can not fetch the secret:
  - It will report a pod event
  - The kubelet will periodically retry fetching the secret
  - Once fetched the kubelet will create and mount the secret volume
  - None of the pod's containers will start until all the pod's volumes are mounted
- Files
  - You can specify the permission mode bits a secret or files of part of a secret will have
    - If you don't specify any, 0644 is used by default
  - When a secret being consumed in a volume is updated, projected keys are eventually updated as well
    - Depends on kubelet sync period
- Environment variables
  - Appear as normal environment variables containing the base-64 decoded values of the secret data
- An imagePullSecret is a way to pass a secret that contains a Docker (or other) image registry password to Kubelet so it can pull a private image on behalf of your Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secret-file-pod
  labels:
    secretpod: db-client
spec:
  volumes:
  - name: secret-volume
    secret:
      secretName: prod-db-secret
  containers:
  - name: secret-file-container
    image: redis
    volumeMounts:
    - name: secret-volume
      readOnly: true
      mountPath: "/etc/secret-volume"
---
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
    - name: secret-env-container
      image: redis
      env:
        - name: SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: prod-db-secret
              key: username
        - name: SECRET_PASSWORD
          valueFrom:
            secretKeyRef:
              name: prod-db-secret
              key: password
```

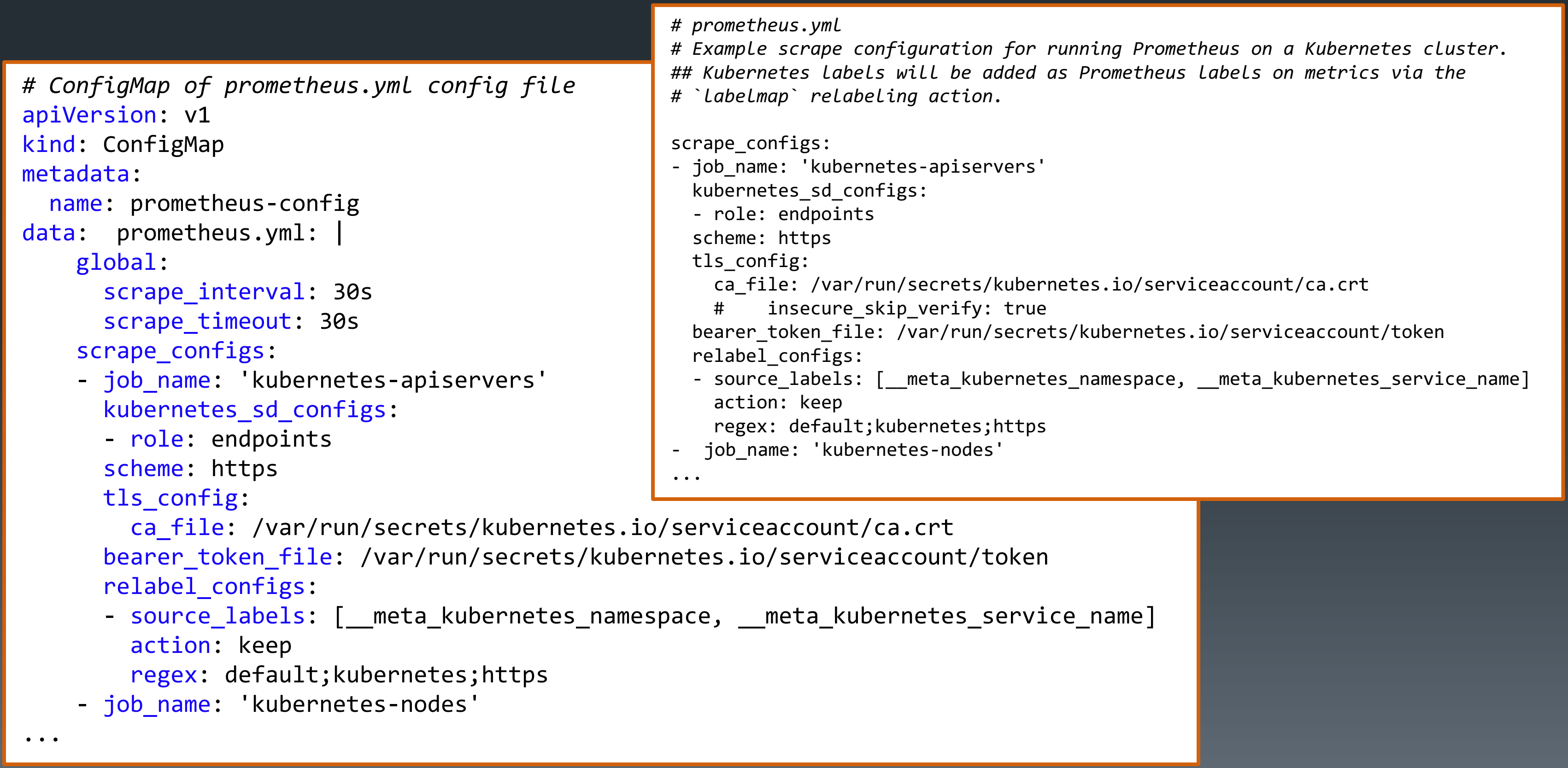
Encrypted Secret Data

- Encryption at rest is alpha as of version 1.7.0
  - Set `--experimental-encryption-provider-config` flag on the apiserver to point to the location of the EncryptionConfig file
  - After restarting apiserver, newly created or updated secrets should be encrypted when stored
- EncryptionConfig file contains keys that can decrypt content in etcd
  - Restrict permissions on masters so only the user who runs apiserver can read it
- The `providers` array is an ordered list of the possible encryption providers:
  - `aescbc` – recommended, strongest but slowest
    - AES-CBC PKCS#7 padding
  - `secretbox` – faster than, but not as strong as aescbc
    - XSalsa20 and Poly 1305
  - `aesgcm` – fastest, not recommended if not using automated key rotation as must be rotated every 200k writes
    - AES-GCM with random nonce
  - `identity` – no encryption
    - When set as first provider, will disable encryption at rest
- Provider key secrets generated with a 32 byte random key and base64 encoding it
  - ex: `head -c 32 /dev/urandom | base64`

```yaml
kind: EncryptionConfig
apiVersion: v1
resources:
  - resources:
    - secrets
    providers:
    - aesgcm:
        keys:
        - name: key1
          secret: c2VjcmV0IGlzIHNlY3VyZQ==
        - name: key2
          secret: dGhpcyBpcyBwYXNzd29yZA==
    - aescbc:
        keys:
        - name: key1
          secret: c2VjcmV0IGlzIHNlY3VyZQ==
        - name: key2
          secret: dGhpcyBpcyBwYXNzd29yZA==
    - secretbox:
        keys:
        - name: key1
          secret: YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd4eXoxMjM0NTY=
    - identity: {}
```

ConfigMaps

- Many applications require configuration via some combination of config files, command line arguments, and environment variables
- These configuration artifacts should be decoupled from image content in order to keep containerized applications portable
- The ConfigMap resource provides mechanisms to inject containers with configuration data while keeping containers agnostic of Kubernetes
- ConfigMap can be used to store fine-grained information like individual properties or coarse-grained information like entire config files or JSON blobs



```
# ConfigMap of prometheus.yml config file
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
data:  prometheus.yml: |
    global:
      scrape_interval: 30s
      scrape_timeout: 30s
    scrape_configs:
    - job_name: 'kubernetes-apiservers'
      kubernetes_sd_configs:
      - role: endpoints
      scheme: https
      tls_config:
        ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
      relabel_configs:
      - source_labels: [__meta_kubernetes_namespace, __meta_kubernetes_service_name]
        action: keep
        regex: default;kubernetes;https
    - job_name: 'kubernetes-nodes'
...
```

```
# prometheus.yml
# Example scrape configuration for running Prometheus on a Kubernetes cluster.
## Kubernetes labels will be added as Prometheus labels on metrics via the
# `labelmap` relabeling action.

scrape_configs:
- job_name: 'kubernetes-apiservers'
  kubernetes_sd_configs:
  - role: endpoints
  scheme: https
  tls_config:
    ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
    #    insecure_skip_verify: true
  bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  relabel_configs:
  - source_labels: [__meta_kubernetes_namespace, __meta_kubernetes_service_name]
    action: keep
    regex: default;kubernetes;https
-  job_name: 'kubernetes-nodes'
...
```

ConfigMaps vs Secrets

Paul Morie (author of both resources):

“

I'm the author of both of these features. The idea is that you should:

1. Use secrets for things which are actually secret like API keys, credentials, etc
2. Use config map for not-secret configuration data

In the future there will likely be some differentiators for secrets like rotation or support for backing the secret API w/ HSMs, etc. In general we like intent-based APIs, and the intent is definitely different for secret data vs. plain old configs.

Hope that helps.
“

http://stackoverflow.com/questions/36912372/kubernetes-secrets-vs-configmaps

Summary

- Kubernetes volumes have the same lifetime as the pods that declare them
  - Kubernetes volumes are not exactly the same as Docker volumes
- Kubernetes supports many volume types & providers
- Volumes are enabled via plugins
- Kubernetes provides powerful storage abstraction using PVs, PVCs & StorageClasses
- StatefulSets leverage abstracted storage for stateful applications
- Secrets can be used to pass sensitive data to Pods
  - kubelet can use secrets for logging into registries
  - Secrets can be encrypted by using EncryptionConfigs

Lab 7

- Volumes