Openshift Controllers

Saif Ur Rehman

IBM Cloud

Topics

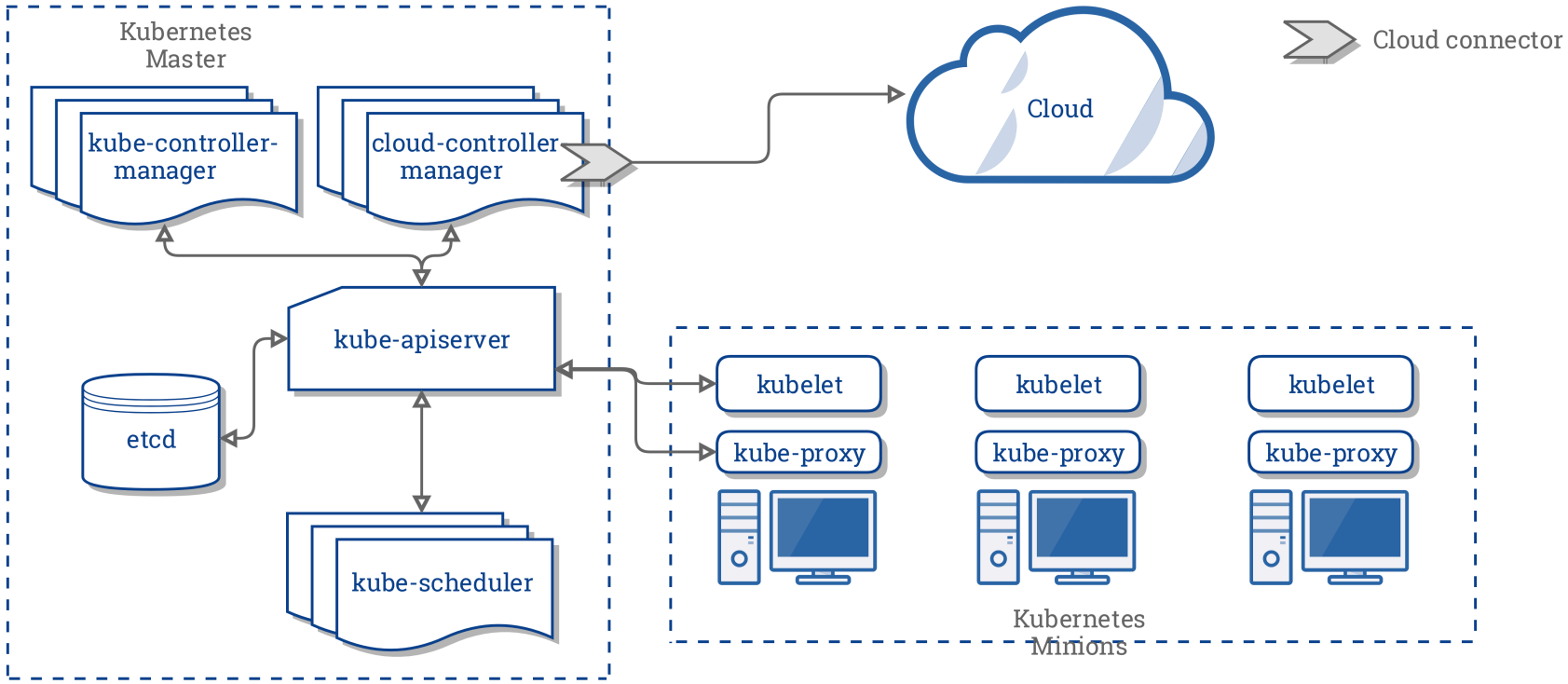
- Kubernetes Core Workload API
- Deployments
- ReplicaSet and Replication Controller
- StatefulSet
- DeamonSet
- Jobs and Cronjobs
- Blue-Green Deployments
- Health Checks and Probe Configuration
- Sidecar Containers

What is a Kubernetes Controller

Kubernetes Controllers:

A group of controllers take care of routine tasks to **ensure the state of the cluster matches the observed state**.

Controllers manage resources like pods, services, etc



Examples:

Replica Sets maintains a correct number of pods running in the cluster

Job runs a task to completion

DaemonSet

Node Controller looks up the state of servers and responds when servers go down

Basically, **each controller is responsible for a particular resource in the Kubernetes world**

Kubernetes Core Workload API

Core Objects Evolution

- As of Kubernetes 1.9/OpenShift 3.9, these objects are known as **Core Workloads API** and are fully supported:
  - `Pod`
  - `ReplicationController`
  - `DeploymentConfig` (OpenShift only)
  - `ReplicaSet`
  - `Deployment`
  - `DaemonSet`
  - `StatefulSet`

Deployments

Use Cases for `Deployment`

- Create `Deployment` to roll out `ReplicaSet`
- Declare new state of pods by updating `PodTemplateSpec` of `Deployment`
- Roll back to earlier `Deployment` revision if current state of `Deployment` not stable
- Scale up `Deployment` to facilitate more load
- Pause `Deployment` to apply multiple fixes to `PodTemplateSpec`, then resume it to start new rollout
- Use status of `Deployment` as indicator that rollout is stuck
- Clean up older, unneeded `ReplicaSet` objects

Deployments

Manipulating Deployments

```
oc rollout latest <dcname>
oc rollout history dc/<name> [--revision=1]
oc deploy --cancel dc/<name>
oc deploy --retry dc/<name>
oc rollout undo dc/<name> [--to-revision=1]
oc rollout pause dc/<name>
oc rollout resume dc/<name>
oc scale dc/<name> --replicas=10
oc autoscale dc/<name> --min=10 --max=15 --cpu-percent=80
oc set resources dc/<name> -c=<container> --limits=cpu=200m,memory=512Mi
```

Deployments

Deployment Triggers

- Deployment configuration can contain triggers
  - Drive creation of new deployment processes in response to cluster events
- Available triggers:
  - Configuration change
  - Image change

Deployments

Deployment Triggers - `ConfigChange`

- Trigger: change detected in deployment configuration pod template
- Result: new replication controller created

```
triggers:
  - type: "ConfigChange"
```

If a `ConfigChange` trigger is defined in a deployment configuration, the first replication controller is automatically created soon after the deployment configuration itself is created. The deployment configuration is not paused when the replication controller is created.

Deployments

Deployment Triggers - ImageChange

- Trigger: change in image stream tag content (new version of image pushed)
- Result: new replication controller created

```
triggers:
  - type: "ImageChange"
    imageChangeParams:
      automatic: true
      from:
        kind: "ImageStreamTag"
        name: "origin-ruby-sample:latest"
        namespace: "myproject"
      containerNames:
        - "helloworld"
```

New version of app gets deployed

Deployments

Deployment Triggers Using the Command Line

- Set `ImageChange` and `ConfigChange` triggers:

```
oc set triggers dc/<name> --from-image=myproject/origin-ruby-sample:latest -c helloworld

oc set triggers dc/<name> --from-config=true
```

- Set all triggers `manual`, `auto`, `remove-all`:

```
oc set triggers dc/<name> --manual
oc set triggers dc/<name> --auto
oc set triggers dc/<name> --remove-all
```

Deployment Strategies

- Deployment strategy determines deployment process
- Defined by deployment configuration
- Uses readiness checks to determine if new pod is ready for use
  - If check fails, deployment configuration continues trying to run pod until timeout
  - Timeout value set in `TimeoutSeconds` in `dc.spec.strategy.*params`
  - Default: `10m`
- Available deployment strategies:
  - Rolling (default)
  - Recreate

StatefulSet

Overview

- Manages deployment and scaling of set of pods (For Stateful Applications)
- Guarantees ordering by maintaining unique, sticky identity for each pod
- Like `Deployment`, `StatefulSet` manages `Pod` objects based on identical container specification
  - Although specifications are same, `Pod` objects in `StatefulSet` are not interchangeable
  - Each `Pod` object has persistent identifier that it maintains across any rescheduling
- `StatefulSet` also operates according to controller pattern
- Define desired state in `StatefulSet` object, `StatefulSet` controller makes updates to get there from current state

StatefulSet

Like Deployments:

- StatefulSet manages Pods that are based on an identical container spec

Unlike Deployments:

- StatefulSet maintains a sticky identity for each of their Pods.
- These pods are created from the same spec, but are **not interchangeable**
- each pod has a persistent identifier that it maintains across any rescheduling

StatefulSet are useful for apps that require one or more of the following

Using `StatefulSet`

- `StatefulSet` valuable for applications that require one or more of following:
  - Stable, unique network identifiers
  - Stable, persistent storage
  - Ordered, graceful deployment and scaling
  - Ordered, graceful deletion and termination
  - Ordered, automated rolling updates
- *Stable* synonymous with persistence across pod (re)scheduling

The state information and other resilient data for any given **StatefulSet** Pod is maintained in persistent disk storage associated with the **StatefulSet**

StatefulSet

Limitations

- Storage for pod must be either of following:
  - Provisioned by `PersistentVolume` provisioner based on requested storage class
  - Pre-provisioned by administrator
- Deleting and scaling down `StatefulSet` does **not** delete volumes associated with `StatefulSet`
  - Done to ensure data safety, which is generally more valuable than automatic purge of all related `StatefulSet` resources
- `StatefulSet` currently requires headless service to be responsible for network identity of pods
  - You are responsible for creating service

StatefulSet

Headless Service Example

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

StatefulSet

StatefulSet Example

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx # has to match .spec.template.metadata.labels
  serviceName: "nginx"
  replicas: 3 # by default is 1
  template:
    metadata:
      labels:
        app: nginx # has to match .spec.selector.matchLabels
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: k8s.gcr.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
```

IBM

StatefulSet

volumeClaimTemplates Example

```
volumeClaimTemplates:
- metadata:
    name: www
  spec:
    accessModes: [ "ReadWriteOnce" ]
    storageClassName: my-storage-class
    resources:
      requests:
        storage: 1Gi
```

StatefulSet

Deployment and Scaling Guarantees

- For `StatefulSet` with N replicas:
  - Deployed pods are created sequentially in order from {0..N-1}
  - Pods are terminated in reverse order from {N-1..0}
- Before scaling operation is applied to pod, all predecessors must be running and ready
- Before pod is terminated, all successors must be completely shut down

DaemonSet

Overview

- All (or some) nodes run copy of pod specified in `DaemonSet` object
- As nodes are added to cluster, pods are added to them
- As nodes are removed from cluster, those pods are garbage-collected
- Deleting `DaemonSet` cleans up pods it created
- Typical uses of `DaemonSet`:
  - Running cluster storage daemon, such as `glusterd` or `ceph`, on each node
  - Running logs collection daemon, such as Fluentd or `logstash`, on every node
  - Running node monitoring daemon on every node
    - Examples: Fluentd, Prometheus Node Exporter, `collectd`, Datadog agent, New Relic agent, Ganglia gmond

IBM

Jobs and Cronjobs

- Job runs any number of pods to completion
- Tracks overall progress of task
- Updates status with information about active, succeeded, failed pods
- Deleting job cleans up any created pods
- Cronjobs use cron notation to schedule repeating jobs
  - `schedule: "*/1 * * * ?"` to run every minute
- Use `oc run` to create Jobs and Cronjobs

Blue – Green Deployments

- Use blue-green deployment to test new application version in production environment before moving traffic to it
- These deployments:
  - Run two versions of application at same time
  - Move production traffic from old to new version

Blue – Green Deployments

Using a Route and Two Services

- Route points to service
- To point to different service, change at any time
- To test new version of code, connect to new service before production traffic is routed to it
- To roll back, change route back

Blue – Green Deployments

Example of Using a Route and Two Services

1. Create two copies of application:

```
oc new-app openshift/deployment-example:v1 --name=example-green
oc new-app openshift/deployment-example:v2 --name=example-blue
```

2. Create route that points to old service:

```
oc expose svc/example-green --name=bluegreen-example
```

3. Edit route and change service name to `example-blue`:

```
oc patch route/bluegreen-example -p '{"spec":{"to":{"name":"example-blue"}}}'
```

Blue – Green Deployments

Example of Using Two Services with an A/B Route

1. Create two copies of application:

```
oc new-app openshift/deployment-example:v1 --name=example-green
oc new-app openshift/deployment-example:v2 --name=example-blue
```

2. Create route that points to first service and set second service as additional back end with weight 0:

```
oc expose svc/example-green --name=bluegreen-example
oc set route-backends bluegreen-example example-green=100 example-blue=0
```

3. Change weights to roll over to blue:

```
oc set route-backends bluegreen-example example-green=0 example-blue=100
```

Health Checks

- Reasons software system components become unhealthy:
  - Transient issues such as temporary connectivity loss
  - Configuration errors
  - Problems with external dependencies
- Features in OpenShift applications that detect and handle unhealthy containers:
  - Liveness probe: checks if container in which it is configured is still running
  - Readiness probe: determines if container is ready to service requests

Health Checks

Probe Timeouts

- Timing of probe controlled by two fields
- Both expressed in units of seconds

Field	Description
`initialDelaySeconds`	How long to wait after container starts to begin probe
`timeoutSeconds`	How long to wait for probe to finish • If time exceeded, OpenShift considers probe to have failed • Default: `1`

Health Checks – Probe Configuration

HTTP Checks

- Ideal for complex applications that can return `200` status when completely initialized
- Kubelet uses webhook to determine health of container
- Check deemed successful if hook returns HTTP response code between `200` and `399`

```
...
readinessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 15
  timeoutSeconds: 1
...
```

Health Checks – Probe Configuration

Container Execution Checks

- Kubelet executes command inside container
- Check successful if exits with status 0

```
...
livenessProbe:
  exec:
    command:
    - cat
    - /tmp/health
  initialDelaySeconds: 15
  timeoutSeconds: 1
...
```

Health Checks – Probe Configuration

TCP Socket Checks

- Ideal for applications that do not start listening until initialization complete
- Kubelet attempts to open socket to container
- Container considered healthy if check establishes connection

```
...
livenessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 15
  timeoutSeconds: 1
...
```

Sidecar Containers

Overview

- Sidecar containers run alongside primary container in pod
- Have two specific functions:
  - Take log files from file and redirect to `stdout`
    - Mostly custom containers
  - Provide authentication to pods that do not have authentication built in
    - OpenShift provides `oauth-proxy` container
- Add to pod template
  - Share pod resources such as volume mounts, shared memory, etc.