Understanding Deployments and DeploymentConfigs

Deployments and *DeploymentConfigs* in OKD are API objects that provide two similar but different methods for fine-grained management over common user applications. They are composed of the following separate API objects:

- A DeploymentConfig or a Deployment, either of which describes the desired state of a particular component of the application as a Pod template.
- DeploymentConfigs involve one or more *ReplicationControllers*, which contain a point-in-time record of the state of a DeploymentConfig as a Pod template. Similarly, Deployments involve one or more *ReplicaSets*, a successor of ReplicationControllers.
- One or more Pods, which represent an instance of a particular version of an application.

Building blocks of a deployment

Deployments and DeploymentConfigs are enabled by the use of native Kubernetes API objects ReplicationControllers and ReplicaSets, respectively, as their building blocks.

Users do not have to manipulate ReplicationControllers, ReplicaSets, or Pods owned by DeploymentConfigs or Deployments. The deployment systems ensures changes are propagated appropriately.

If the existing deployment strategies are not suited for your use case and you must run manual steps during the lifecycle of your deployment, then you should consider creating a Custom deployment strategy.

The following sections provide further details on these objects.

ReplicationControllers

A ReplicationController ensures that a specified number of replicas of a Pod are running at all times. If Pods exit or are deleted, the ReplicationController acts to instantiate more up to the defined number. Likewise, if there are more running than desired, it deletes as many as necessary to match the defined amount.

A ReplicationController configuration consists of:

- The number of replicas desired (which can be adjusted at runtime).
- A Pod definition to use when creating a replicated Pod.
- A selector for identifying managed Pods.

A selector is a set of labels assigned to the Pods that are managed by the ReplicationController. These labels are included in the Pod definition that the ReplicationController instantiates. The ReplicationController uses the selector to determine how many instances of the Pod are already running in order to adjust as needed.

The ReplicationController does not perform auto-scaling based on load or traffic, as it does not track either. Rather, this requires its replica count to be adjusted by an external auto-scaler.

The following is an example definition of a ReplicationController:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend-1
spec:
  replicas: 1   1
  selector:     2
    name: frontend
  template:     3
    metadata:
      labels:   4
        name: frontend  5
    spec:
      containers:
      - image: openshift/hello-openshift
        name: helloworld
        ports:
        - containerPort: 8080
          protocol: TCP
      restartPolicy: Always
```

1 The number of copies of the Pod to run.

2 The label selector of the Pod to run.

3 A template for the Pod the controller creates.

4 Labels on the Pod should include those from the label selector.

5 The maximum name length after expanding any parameters is 63 characters.

ReplicaSets

Similar to a ReplicationController, a ReplicaSet is a native Kubernetes

API object that ensures a specified number of pod replicas are running at any given time. The difference between a ReplicaSet and a ReplicationController is that a ReplicaSet supports set-based selector requirements whereas a replication controller only supports equality-based selector requirements.

Only use ReplicaSets if you require custom update orchestration or do not require updates at all. Otherwise, use Deployments. ReplicaSets can be used independently, but are used by deployments to orchestrate pod creation, deletion, and updates. Deployments manage their ReplicaSets automatically, provide declarative updates to pods, and do not have to manually manage the ReplicaSets that they create.

The following is an example `ReplicaSet` definition:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend-1
  labels:
    tier: frontend
spec:
  replicas: 3
  selector:  1
    matchLabels:  2
      tier: frontend
    matchExpressions:  3
      - {key: tier, operator: In, values: [fronten
d]}
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - image: openshift/hello-openshift
        name: helloworld
        ports:
        - containerPort: 8080
          protocol: TCP
      restartPolicy: Always
```

1 A label query over a set of resources. The result of `matchLabels` and `matchExpressions` are logically conjoined.

2 Equality-based selector to specify resources with labels that match the selector.

3 Set-based selector to filter keys. This selects all resources with key equal to `tier` and value equal to `frontend`.

DeploymentConfigs

Building on ReplicationControllers, OKD adds expanded support for the software development and deployment lifecycle with the concept of *DeploymentConfigs*. In the simplest case, a DeploymentConfig creates a new ReplicationController and lets it start up Pods.

However, OKD deployments from DeploymentConfigs also provide the ability to transition from an existing deployment of an image to a new one and also define hooks to be run before or after creating the ReplicationController.

The DeploymentConfig deployment system provides the following capabilities:

- A DeploymentConfig, which is a template for running applications.
- Triggers that drive automated deployments in response to events.
- User-customizable deployment strategies to transition from the previous version to the new version. A strategy runs inside a Pod commonly referred as the deployment process.
- A set of hooks (lifecycle hooks) for executing custom behavior in different points during the lifecycle of a deployment.
- Versioning of your application in order to support rollbacks either manually or automatically in case of deployment failure.

- Manual replication scaling and autoscaling.

When you create a DeploymentConfig, a ReplicationController is created representing the DeploymentConfig's Pod template. If the DeploymentConfig changes, a new ReplicationController is created with the latest Pod template, and a deployment process runs to scale down the old ReplicationController and scale up the new one.

Instances of your application are automatically added and removed from both service load balancers and routers as they are created. As long as your application supports graceful shutdown when it receives the `TERM` signal, you can ensure that running user connections are given a chance to complete normally.

The OKD `DeploymentConfig` object defines the following details:

1. The elements of a `ReplicationController` definition.
2. Triggers for creating a new deployment automatically.
3. The strategy for transitioning between deployments.
4. Lifecycle hooks.

Each time a deployment is triggered, whether manually or automatically, a deployer Pod manages the deployment (including scaling down the old ReplicationController, scaling up the new one, and running hooks). The deployment pod remains for an indefinite amount of time after it completes the Deployment in order to retain its logs of the Deployment. When a deployment is superseded by another, the previous ReplicationController is retained to enable easy rollback if needed.

Example DeploymentConfig definition

```
apiVersion: v1
kind: DeploymentConfig
metadata:
  name: frontend
spec:
  replicas: 5
  selector:
    name: frontend
  template: { ... }
  triggers:
  - type: ConfigChange  1
  - imageChangeParams:
      automatic: true
      containerNames:
      - helloworld
      from:
        kind: ImageStreamTag
        name: hello-openshift:latest
    type: ImageChange   2
  strategy:
    type: Rolling       3
```

1 A `ConfigChange` trigger causes a new Deployment to be created any time the ReplicationController template changes.

2 An `ImageChange` trigger causes a new Deployment to be created each time a new version of the backing image is available in the named imagestream.

3 The default `Rolling` strategy makes a downtime-free transition between Deployments.

Deployments

Kubernetes provides a first-class, native API object type in OKD called *Deployments*. Deployments serve as a descendant of the OKD-specific DeploymentConfig.

Like DeploymentConfigs, Deployments describe the desired state of a particular component of an application as a Pod template. Deployments create ReplicaSets, which orchestrate Pod lifecycles.

For example, the following Deployment definition creates a ReplicaSet to bring up one **`hello-openshift`** Pod:

Deployment definition

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-openshift
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello-openshift
  template:
    metadata:
      labels:
        app: hello-openshift
    spec:
      containers:
      - name: hello-openshift
        image: openshift/hello-openshift:latest
        ports:
        - containerPort: 80
```

Comparing Deployments and DeploymentConfigs

Both Kubernetes Deployments and OKD-provided DeploymentConfigs are supported in OKD; however, it is recommended to use Deployments unless you need a specific feature or behavior provided by DeploymentConfigs.

The following sections go into more detail on the differences between the two object types to further help you decide which type to use.

Design

One important difference between Deployments and DeploymentConfigs is the properties of the CAP theorem (https://en.wikipedia.org/wiki/CAP_theorem) that each design has chosen for the rollout process. DeploymentConfigs prefer consistency, whereas Deployments take availability over consistency.

For DeploymentConfigs, if a node running a deployer Pod goes down, it will not get replaced. The process waits until the node comes back online or is manually deleted. Manually deleting the node also deletes the corresponding Pod. This means that you can not delete the Pod to unstick the rollout, as the kubelet is responsible for deleting the associated Pod.

However, Deployments rollouts are driven from a controller manager. The controller manager runs in high availability mode on masters and uses leader election algorithms to value availability over consistency. During a failure it is possible for other masters to act on the same Deployment at the same time, but this issue will be reconciled shortly after the failure occurs.

DeploymentConfigs-specific features

Automatic rollbacks

Currently, Deployments do not support automatically rolling back to the last successfully deployed ReplicaSet in case of a failure.

Triggers

Deployments have an implicit `ConfigChange` trigger in that every change in the pod template of a deployment automatically triggers a new rollout. If you do not want new rollouts on pod template changes, pause the deployment:

```
$ oc rollout pause deployments/<name>
```

Lifecycle hooks

Deployments do not yet support any lifecycle hooks.

Custom strategies

Deployments do not support user-specified Custom deployment strategies yet.

Deployments-specific features

Rollover

The deployment process for Deployments is driven by a controller loop, in contrast to DeploymentConfigs which use deployer pods for every new rollout. This means that a Deployment can have as many active ReplicaSets as possible, and eventually the deployment controller will scale down all old ReplicaSets and scale up the newest one.

DeploymentConfigs can have at most one deployer pod running, otherwise multiple deployers end up conflicting while trying to scale up what they think should be the newest ReplicationController. Because of this, only two ReplicationControllers can be active at any point in time. Ultimately, this translates to faster rapid rollouts for Deployments.

Proportional scaling

Because the Deployment controller is the sole source of truth for the sizes of new and old ReplicaSets owned by a Deployment, it is able to scale ongoing rollouts. Additional replicas are distributed proportionally based on the size of each ReplicaSet.

DeploymentConfigs cannot be scaled when a rollout is ongoing because the DeploymentConfig controller will end up having issues with the deployer process about the size of the new ReplicationController.

Pausing mid-rollout

Deployments can be paused at any point in time, meaning you can also pause ongoing rollouts. On the other hand, you cannot pause deployer pods currently, so if you try to pause a DeploymentConfig in the middle of a rollout, the deployer process will not be affected and will continue until it finishes.