Kubernetes Deployment YAML: Learn by Example

What is Kubernetes Deployment YAML?

YAML (which stands for YAML Ain't Markup Language) is a language used to provide configuration for software, and is the main type of input for Kubernetes configurations. It is human-readable and can be authored in any text editor.

A Kubernetes user or administrator specifies data in a YAML file, typically to define a Kubernetes object. The YAML configuration is called a "manifest", and when it is "applied" to a Kubernetes cluster, Kubernetes creates an object based on the configuration.

A Kubernetes Deployment YAML specifies the configuration for a Deployment object—this is a Kubernetes object that can create and update a set of identical pods. Each pod runs specific containers, which are defined in the spec.template field of the YAML configuration.

The Deployment object not only creates the pods but also ensures the correct number of pods is always running in the cluster, handles scalability, and takes care of updates to the pods on an ongoing basis. All these activities can be configured through fields in the Deployment YAML.

Below we'll show several examples that will walk you through the most common options in a Kubernetes Deployment YAML manifest.

Kubernetes Deployment YAML Examples

With Multiple Replicas

The following YAML configuration creates a Deployment object that runs 5 replicas of an NGINX container.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: web
spec:
  selector:
    matchLabels:
      app: web
  replicas: 5
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
       –name: nginx
          image: nginx
          ports:
           –containerPort: 80
```

Important points in this configuration:

- `spec.replicas`—specifies how many pods to run
- `strategy.type`—specifies which deployment strategy should be used. In this case and in the following examples we select RollingUpdate, which means new versions are rolled out gradually to pods to avoid downtime.
- `spec.template.spec.containers`—specifies which container image to run in

each of the pods and ports to expose.

With Resource Limits

The following YAML configuration creates a Deployment object similar to the above, but with resource limits.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: web
spec:
  selector:
    matchLabels:
      app: web
  replicas: 5
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
       –name: nginx
          image: nginx
          resources:
            limits:
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
          ports:
           –containerPort: 80
```

The spec.containers.resources field specifies:

- `limits`—each container should not be allowed to consume more than 200Mi of memory.
- `requests`—each container requires 100m of CPU resources and 200Mi of

memory on the node

With Health Checks

The following YAML configuration creates a Deployment object that performs a health check on containers by checking for an HTTP response on the root directory.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: web
spec:
  selector:
    matchLabels:
      app: web
  replicas: 5
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
       –name: nginx
          image: nginx
          ports:
           –containerPort: 80
          livenessProbe:
            httpGet:
              path: /
              port: 80
            initialDelaySeconds: 5
            periodSeconds: 5
```

The `template.spec.containers.livenessProbe` field defines what the kubelet should check to ensure that the pod is alive:

- `httpGet`specifies that the kubelet should try a HTTP request on the root of the web server on port 80.

- `periodSeconds`specifies how often the kubelet should perform a liveness probe.
- `initialDelaySeconds`specifies how long the kubelet should wait after the pod starts, before performing the first probe.

You can also define readiness probes and startup probes—learn more in the Kubernetes documentation.

With Persistent Volumes

The following YAML configuration creates a Deployment object that creates containers that request a PersistentVolume (PV) using a PersistentVolumeClaim (PVC), and mount it on a path within the container.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: web
spec:
  selector:
    matchLabels:
      app: web
  replicas: 5
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: web
    spec:
      volumes:
       –name: my-pv-storage
          persistentVolumeClaim:
            claimName: my-pv-claim
      containers:
       –name: nginx
          image: nginx
          ports:
           –containerPort: 80
          volumeMounts:
```

```
        –mountPath: "/usr/share/nginx/html"
           name: my-pv-storage
```

Important points in this configuration:

- `template.spec.volumes`—defines a name for the volume, which is referenced below in containers.volumeMounts
- `template.spec.volumes.persistVolumeClaim`—references a PVC. For this to work, you must have some PVs in your cluster and create a PVC object that matches those PVs. You can then reference the existing PVC object here and the pod will attempt to bind to a matching PV.

Learn more about PVs and PVCs in the documentation.

With Affinity Settings

The following YAML configuration creates a Deployment object with affinity criteria that can encourage a pod to schedule on certain types of nodes.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: web
spec:
  selector:
    matchLabels:
      app: web
  replicas: 5
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: web
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
           –matchExpressions:
```

```
            –key: disktype
               operator: In
               values:
              –ssd
      containers:
       –name: nginx
          image: nginx
          ports:
           –containerPort: 80
```

The `spec.affinity`field defines criteria that can affect whether the pod schedules on a certain node or not:

- `spec.affinity.nodeAffinity`—specifies desired criteria of a node which will cause the pod to be scheduled on it
- `spec.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringE›` —specifies that affinity is relevant when scheduling a new pod, but is ignored when the pod is already running.
- `nodeSelectorTerms`—specifies, in this case, that the node needs to have a disk of type SSD for the pod to be scheduled.

There are many other options, including preferred node affinity, and pod affinity, which means the pod is scheduled based on the criteria of other pods running on the same node. Learn more in the documentation.

Alternatives to the Deployment Object

Two common alternatives to the Kubernetes Deployment object are:

- **DaemonSet**—deploys a pod on all cluster nodes or a certain subset of nodes
- **StatefulSet**—used for stateful applications. Similar to a Deployment, but each pod is unique and has a persistent identifier.

Let's see examples of YAML configurations for these two objects. The code is taken from the Kubernetes documentation.

Kubernetes DaemonSet Example YAML

A DaemonSet runs copies of a pod on all cluster nodes, or a selection of nodes within a cluster. Whenever a node is added to the cluster, the DaemonSet

controller checks if it is eligible, and if so, runs the pod on it. When a node is removed from the cluster, the pods are moved to garbage collection. Deleting a DaemonSet also results in removal of the pods it created.

The following YAML file shows how to run a DaemonSet that runs fluentd-elasticsearch for logging purposes.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      # this toleration is to have the daemonset runnable on
      # master nodes
     –key: node-role.kubernetes.io/master
        operator: Exists
        effect: NoSchedule
      containers:
     –name: fluentd-elasticsearch
        image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
       –name: varlog
          mountPath: /var/log
       –name: varlibdockercontainers
```

```
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
     –name: varlog
        hostPath:
          path: /var/log
     –name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

The important fields of this configuration are:

- `selector`—specifies which nodes the pod should run on. In this case we assume that all pods that need the logging component will have the label `fluentd-elasticsearch`
- `spec.tolerations`—tolerations are applied to pods, and allow the pods to schedule on nodes with matching characteristics. In this case we allow the pod to run on a node even if it is a master node.
- `containers, volumes`—specifies what pod and storage volumes the DaemonSet should run on each node.

Kubernetes Statefulset Example YAML

A StatefulSet manages a group of pods while maintaining a sticky identity for each pod, with a persistent identifier that remains even if the pod is shut down and restarted. Pods also have PersistentVolumes that can store data that outlines the lifecycle of each individual pod.

The following example shows a YAML configuration for a headless Service that controls the network domain, and a StatefulSet that runs 3 instances of an NGINX web server.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
 –port: 80
```

```
    name: web
  clusterIP: None
  selector:
    app: nginx
---
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
  minReadySeconds: 10 # by default is 0
  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
  volumeClaimTemplates:
 –metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "my-storage-class"
      resources:
        requests:
          storage: 1Gi
```

The important fields of this configuration are:

- `metadata.name`—must be a valid DNS subdomain name.
- `.spec.selector.matchLabels and .spec.template.metadata.labels`—both of these must match and are referenced by the headless Service to route requests to the application.
- `spec.selector.replicas`—specifies that the StatefulSet should run three replicas of the container, each with a unique persistent identifier.
- `spec.template.spec.containers`—specifies what NGINX image to run and how it should mount the PersistentVolumes.
- `volumeClaimTemplates`—provides persistent storage using the `my-storage-class` storage class. In a real environment, your cluster will have one or more storage classes defined by the cluster administrator, which provide different types of persistent storage.

Kubernetes Deployment with Codefresh

The Codefresh Software Delivery Platform, powered by Argo, lets you answer many important questions within your organization, whether you're a developer or a product manager. For example:

- What features are deployed right now in any of your environments?
- What features are waiting in Staging?
- What features were deployed last Thursday?
- Where is feature #53.6 in our environment chain?

What's great is that you can answer all of these questions by viewing one single dashboard. Our applications dashboard shows:

- Services affected by each deployment
- The current state of Kubernetes components
- Deployment history and log of who deployed what and when and the pull request or Jira ticket associated with each deployment

Related Kubernetes Deployment articles

Kubernetes Deployment: From Basic Strategies to Progressive Delivery

Top 6 Kubernetes Deployment Strategies and How to Choose

Codefresh Steps

Learning Center

Ebooks & Reports

Blog

Connect

Sign Up

Support

Contact Us

Company

About Codefresh

Careers

codefresh

Codefresh is the most trusted GitOps platform for cloud-native apps. It's built on Argo for declarative continuous delivery, making modern software delivery possible at enterprise scale.

Google Cloud Platform
Technology Partner

Microsoft | AZURE PARTNER

docker MEMBER

SOC 2 Type II
COMPLIANT

Microsoft for Startups
CHOICE AWARD