Learning Objectives

By the end of this chapter, you should be able to:

- Discuss Deployment configuration details.
- Scale a Deployment up and down.
- Implement rolling updates and rollback.
- Use Labels to select various objects.

MANAGING STATE WITH DEPLOYMENTS

Overview

The default controller for a container deployed via **kubectl run** is a Deployment. While we have been working with them already, we will take a closer look at configuration options.

As with other objects, a deployment can be made from a YAML or JSON spec file. When added to the cluster, the controller will create a ReplicaSet and a Pod automatically. The containers, their settings and applications can be modified via an update, which generates a new ReplicaSet, which, in turn, generates new Pods.

The updated objects can be staged to replace previous objects as a block or as a rolling update, which is determined as part of the deployment specification. Most updates can be configured by editing a YAML file and running **kubectl apply**. You can also use **kubectl edit** to modify the in-use configuration. Previous versions of the ReplicaSets are kept, allowing a rollback to return to a previous configuration.

We will also talk more about labels. Labels are essential to administration in Kubernetes, but are not an API resource. They are user-defined key-value pairs which can be attached to any resource, and are stored in the metadata. Labels are used to query or select resources in your cluster, allowing for flexible and complex management of the cluster.

As a label is arbitrary, you could select all resources used by developers, or belonging to a user, or any attached string, without having to figure out what kind or how many of such resources exist.

Deployments

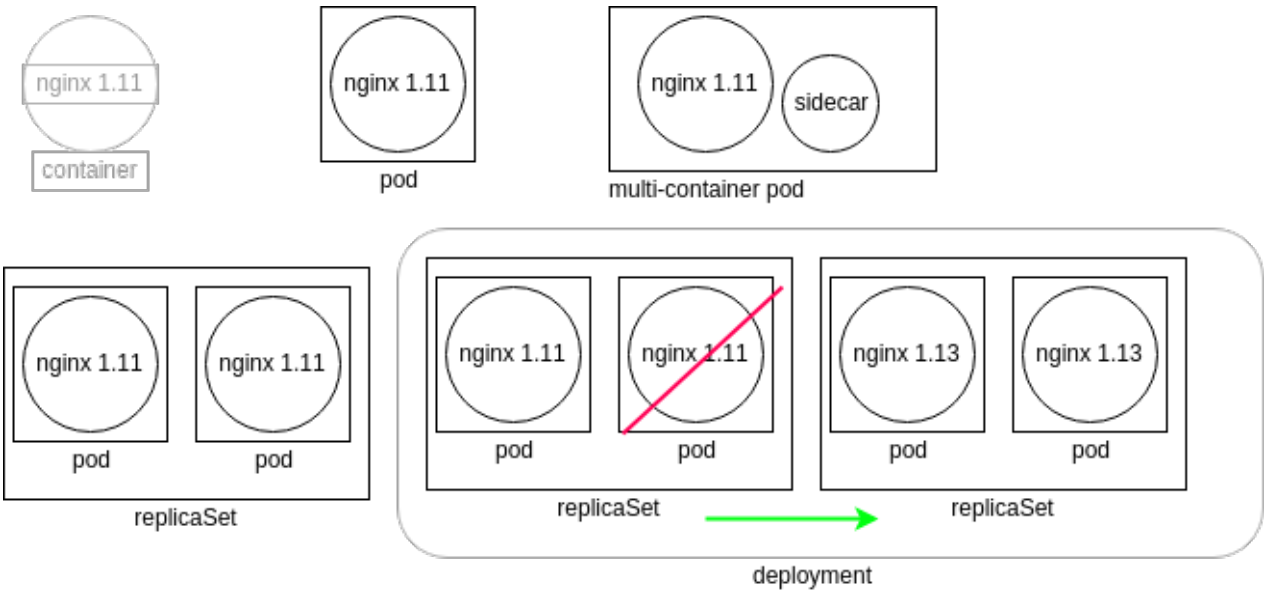
ReplicationControllers (RC) ensure that a specified number of pod replicas is running at any one time. ReplicationControllers also give you the ability to perform rolling updates. However, those updates are managed on the client side. This is problematic if the client loses connectivity, and can leave the cluster in an unplanned state. To avoid problems when scaling the ReplicationControllers on the client side, a new resource was introduced in the **apps/v1** API group: Deployments.

Deployments allow server-side updates to pods at a specified rate. They are used for canary and other deployment patterns. Deployments generate ReplicaSets, which offer more selection features than ReplicationControllers, such as **matchExpressions**.

```
$ kubectl create deployment dev-web --image=nginx:1.13.7-alpine
deployment "dev-web" created
```

Object Relationship

Here you can see the relationship between objects from the container, which Kubernetes does not directly manage, up to the deployment.



Nested Objects

The boxes and shapes are logical, in that they represent the controllers, or watch loops, running as a thread of the kube-controller-manager. Each controller queries the kube-apiserver for the current state of the object they track. The state of each object on a worker node is sent back from the local kubelet.

The graphic in the upper left represents a container running nginx 1.11. Kubernetes does not directly manage the container. Instead, the kubelet daemon checks the pod specifications by asking the container engine, which could be Docker or cri-o, for the current status. The graphic to the right of the container shows a pod which represents a watch loop checking the container status. kubelet compares the current pod spec against what the container engine replies and will terminate and restart the pod if necessary.

A multi-container pod is shown next. While there are several names used, such as *sidecar* or *ambassador*, these are all multi-container pods. The names are used to indicate the particular reason to have a second container in the pod, instead of denoting a new kind of pod.

On the lower left we see a **replicaSet**. This controller will ensure you have a certain number of pods running. The pods are all deployed with the same **podSpec**, which is why they are called replicas. Should a pod terminate or a new pod be found, the replicaSet will create or terminate pods until the current number of running pods matches the specifications. Any of the current pods could be terminated should the spec demand fewer pods running.

The graphic in the lower right shows a deployment. This controller allows us to manage the versions of images deployed in the pods. Should an edit be made to the deployment, a new **replicaSet** is created, which will deploy pods using the new **podSpec**. The deployment will then direct the old **replicaSet** to shut down pods as the new **replicaSet** pods become available. Once the old pods are all terminated, the deployment terminates the old **replicaSet** and the deployment returns to having only one **replicaSet** running.

Deployment Details

In the previous page, we created a new deployment running a particular version of the nginx web server.

To generate the YAML file of the newly created objects, do:

```
$ kubectl get deployments,rs,pods -o yaml
```

Sometimes, a JSON output can make it more clear:

```
$ kubectl get deployments,rs,pods -o json
```

Now we will look at the YAML output, which also shows default values not passed to the object when created:

```
apiVersion: v1
items:
- apiVersion: apps/v1
  kind: Deployment
```

Explanation of Objects

- apiVersion

A value of **v1** shows that this object is considered to be a stable resource. In this case, it is not the deployment. It is a reference to the **List** type.

- items

As the previous line is a **List**, this declares the list of items the command is showing.

- - apiVersion

The dash is a YAML indication of the first item, which declares the **apiVersion** of the object as **apps/v1**. This indicates the object is considered stable. Deployments are controller used in most cases.

- kind

This is where the type of object to create is declared, in this case, a deployment.

Deployment Configuration Metadata

Continuing with the YAML output, we see the next general block of output concerns the metadata of the deployment. This is where we would find labels, annotations, and other non-configuration information. Note that this output will not show all possible configuration. Many settings which are set to false by default are not shown, like **podAffinity** or **nodeAffinity**.

```
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  creationTimestamp: 2017-12-21T13:57:07Z
  generation: 1
  labels:
    app: dev-web
  name: dev-web
  namespace: default
  resourceVersion: "774003"
  selfLink: /apis/apps/v1/namespaces/default/deployments/dev-web
  uid: d52d3a63-e656-11e7-9319-42010a800003
```

- annotations

These values do not configure the object, but provide further information that could be helpful to third-party applications or administrative tracking. Unlike labels, they cannot be used to select an object with **kubectl**.

- creationTimestamp

Shows when the object was originally created. Does not update if the object is edited.

- generation

How many times this object has been edited, such as changing the number of replicas, for example.

- labels

Arbitrary strings used to select or exclude objects for use with **kubectl**, or other API calls. Helpful for administrators to select objects outside of typical object boundaries.

- name

This is a *required* string, which we passed from the command line. The name must be unique to the namespace.

- resourceVersion

A value tied to the etcd database to help with concurrency of objects. Any changes to the database will cause this number to change.

- selfLink

References how the kube-apiserver will ingest this information into the API.

- uid

Remains a unique ID for the life of the object.

Deployment Configuration Spec

There are two **spec** declarations for the deployment. The first will modify the ReplicaSet created, while the second will pass along the Pod configuration.

```
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: dev-web
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
```

- spec

A declaration that the following items will configure the object being created.

- progressDeadlineSeconds

Time in seconds until a progress error is reported during a change. Reasons could be quotas, image issues, or limit ranges.

- replicas

As the object being created is a ReplicaSet, this parameter determines how many Pods should be created. If you were to use **kubectl edit** and change this value to two, a second Pod would be generated.

- revisionHistoryLimit

How many old ReplicaSet specifications to retain for rollback.

- selector

A collection of values ANDed together. All must be satisfied for the replica to match. Do not create Pods which match these selectors, as the deployment controller may try to control the resource, leading to issues.

- matchLabels

Set-based requirements of the Pod selector. Often found with the **matchExpressions** statement, to further designate where the resource should be scheduled.

- strategy

A header for values having to do with updating Pods. Works with the later listed **type**. Could also be set to **Recreate**, which would delete all existing pods before new pods are created. With **RollingUpdate**, you can control how many Pods are deleted at a time with the following parameters.

- maxSurge

Maximum number of Pods over desired number of Pods to create. Can be a percentage, default of 25%, or an absolute number. This creates a certain number of new Pods before deleting old ones, for continued access.

- maxUnavailable

A number or percentage of Pods which can be in a state other than **Ready** during the update process.

- type

Even though listed last in the section, due to the level of white space indentation, it is read as the type of object being configured. (e.g. **RollingUpdate**).

Deployment Configuration Pod Template

Next, we will take a look at a configuration template for the pods to be deployed. We will see some similar values.

```
template:
  metadata:
  creationTimestamp: null
    labels:
      app: dev-web
  spec:
    containers:
    - image: nginx:1.13.7-alpine
      imagePullPolicy: IfNotPresent
      name: dev-web
      resources: {}
      terminationMessagePath: /dev/termination-log
```

```
      terminationMessagePolicy: File
    dnsPolicy: ClusterFirst
    restartPolicy: Always
    schedulerName: default-scheduler
    securityContext: {}
    terminationGracePeriodSeconds: 30
```

- template

Data being passed to the ReplicaSet to determine how to deploy an object (in this case, containers).

- containers

Key word indicating that the following items of this indentation are for a container.

- image

This is the image name passed to the container engine, typically Docker. The engine will pull the image and create the Pod.

- imagePullPolicy

Policy settings passed along to the container engine, about when and if an image should be downloaded or used from a local cache.

- name

The leading stub of the Pod names. A unique string will be appended.

- resources

By default, empty. This is where you would set resource restrictions and settings, such as a limit on CPU or memory for the containers.

- terminationMessagePath

A customizable location of where to output success or failure information of a container.

- terminationMessagePolicy

The default value is **File**, which holds the termination method. It could also be set to **FallbackToLogsOnError**, which will use the last chunk of container log if the message file is empty and the container shows an error.

- dnsPolicy

Determines if DNS queries should go to **coredns** or, if set to **Default**, use the node's DNS resolution configuration.

- restartPolicy

Should the container be restarted if killed? Automatic restarts are part of the typical strength of Kubernetes.

- scheduleName

Allows for the use of a custom scheduler, instead of the Kubernetes default.

- securityContext

Flexible setting to pass one or more security settings, such as SELinux context, AppArmor values, users and UIDs for the containers to use.

- terminationGracePeriodSeconds

The amount of time to wait for a **SIGTERM** to run until a **SIGKILL** is used to terminate the container.

Deployment Configuration Status

The **status** output is generated when the information is requested:

```
status:
  availableReplicas: 2
  conditions:
  - lastTransitionTime: 2017-12-21T13:57:07Z
    lastUpdateTime: 2017-12-21T13:57:07Z
    message: Deployment has minimum availability.
    reason: MinimumReplicasAvailable
    status: "True"
    type: Available
  observedGeneration: 2
  readyReplicas: 2
  replicas: 2
  updatedReplicas: 2
```

The output above shows what the same deployment were to look like if the number of replicas were increased to two. The times are different than when the deployment was first generated.

Explanation of Additional Elements

- availableReplicas

Indicates how many were configured by the ReplicaSet. This would be compared to the later value of **readyReplicas**, which would be used to determine if all replicas have been fully generated and without error.

- observedGeneration

Shows how often the deployment has been updated. This information can be used to understand the rollout and rollback situation of the deployment.

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      READY   UP-TO-DATE   AVAILABLE   AGE
dev-web   4/4     4            4           20s
```

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.

Deployment Rollbacks

With some of the previous ReplicaSets of a Deployment being kept, you can also roll back to a previous revision by scaling up and down. The number of previous configurations kept is configurable, and has changed from version to version. Next, we will have a closer look at rollbacks, using the **--record** option of the **kubectl create** command, which allows annotation in the resource definition.

```
$ kubectl create deploy ghost --image=ghost --record

$ kubectl get deployments ghost -o yaml
deployment.kubernetes.io/revision: "1"
kubernetes.io/change-cause: kubectl create deploy ghost --image=ghost --record
```

Should an update fail, due to an improper image version, for example, you can roll back the change to a working version with **kubectl rollout undo**:

```
$ kubectl set image deployment/ghost ghost=ghost:09 --all

$ kubectl rollout history deployment/ghost deployments "ghost":
REVISION   CHANGE-CAUSE
1          kubectl create deploy ghost --image=ghost --record
2          kubectl set image deployment/ghost ghost=ghost:09 --all

$ kubectl get pods
NAME                     READY   STATUS             RESTARTS   AGE
ghost-2141819201-tcths   0/1     ImagePullBackOff   0          1m

$ kubectl rollout undo deployment/ghost ; kubectl get pods

NAME                     READY   STATUS    RESTARTS   AGE
ghost-3378155678-eq5i6   1/1     Running   0          7s
```

You can roll back to a specific revision with the --to-revision=2 option.

You can also edit a Deployment using the kubectl edit command.

You can also pause a Deployment, and then resume.

```
$ kubectl rollout pause deployment/ghost

$ kubectl rollout resume deployment/ghost
```

Please note that you can still do a rolling update on ReplicationControllers with the **kubectl rolling-update** command, but this is done on the client side. Hence, if you close your client, the rolling update will stop.

Using 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**.

There are ways of effecting the kube-apischeduler such that some nodes will not run a DaemonSet.

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 create** 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 -L run
NAME                     READY   STATUS    RESTARTS   AGE   RUN
ghost-3378155678-eq5i6   1/1     Running   0          10m   ghost
nginx-3771699605-4v27e   1/1     Running   1          1h    nginx
```

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
```