Learning Objectives

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

- Explore API versions.
- Discuss rapid change and development.
- Deploy and configure and application using a Deployment.
- Examine primitives for a self-healing application.
- Scale an application.

API Objects

Overview

This chapter is about additional API resources or objects. We will learn about resources in the **v1** API group, among others. Stability increases and code becomes more stable as objects move from **alpha** versions, to **beta**, and then **v1**, indicating stability.

DaemonSets, which ensure a Pod on every node, and and StatefulSets, which stick a container to a node and otherwise act like a deployment, have progressed to **apps/v1** stability. Jobs and CronJobs are now in **batch/v1**.

Role-Based Access Control (RBAC), essential to security, has made the leap from **v1alpha1** to the stable **v1** status in one release.

As a fast moving project keeping track of changes, any possible changes can be an important part of the ongoing system administration. Release notes, as well as discussions to release notes, can be found in version-dependent subdirectories in the Features tracking repository for Kubernetes releases on GitHub. For example, the v1.17 release feature status can be found online, on the *Kubernetes v1.17.0 Release Notes* page.

v1 API Group

The **v1** API group is no longer a single group, but rather a collection of groups for each main object category. For example, there is a **v1** group, a **storage.k8s.io/v1** group, and an **rbac.authorization.k8s.io/v1**, etc. Currently, there are eight v1 groups.

Objects

- Node

Represents a machine - physical or virtual - that is part of your Kubernetes cluster. You can get more information about nodes with the **kubectl get nodes** command. You can turn on and off the scheduling to a node with the **kubectl cordon/uncordon** commands.

- Service Account

Provides an identifier for processes running in a pod to access the API server and performs actions that it is authorized to do.

- Resource Quota

It is an extremely useful tool, allowing you to define quotas per namespace. For example, if you want to limit a specific namespace to only run a given number of pods, you can write a **resourcequota** manifest, create it with **kubectl** and the quota will be enforced.

- Endpoint

Generally, you do not manage endpoints. They represent the set of IPs for pods that match a particular service. They are handy when you want to check that a service actually matches some running pods. If an endpoint is empty, then it means that there are no matching pods and something is most likely wrong with your service definition.

Discovering API Groups

We can take a closer look at the output of the request for current APIs. Each of the name values can be appended to the URL to see details of that group. For example, you could drill down to find included objects at this URL: https://localhost:6443/apis/apiregistrationk8s.io/v1beta1.

If you follow this URL, you will find only one resource, with a name of apiservices. If it seems to be listed twice, the lower output is for status. You'll notice that there are different verbs or actions for each. Another entry is if this object is namespaced, or restricted to only one namespace. In this case, it is not.

```
$ curl https://localhost:6443/apis --header "Authorization: Bearer $token" -k

{
  "kind": "APIGroupList",
  "apiVersion": "v1",
  "groups": [
    {
      "name": "apiregistration.k8s.io",
      "versions": [
        {
          "groupVersion": "apiregistration.k8s.io/v1",
          "version": "v1"
        }
      ],
      "preferredVersion": {
        "groupVersion": "apiregistration.k8s.io/v1",
        "version": "v1"
        }
```

You can then curl each of these URIs and discover additional API objects, their characteristics and associated verbs.

Deploying an Application

Using the **kubectl create** command, we can quickly deploy an application. We have looked at the Pods created running the application, like nginx. Looking closer, you will find that a Deployment was created, which manages a ReplicaSet, which then deploys the Pod.

Objects

- Deployment

It is a controller which manages the state of ReplicaSets and the pods within. The higher level control allows for more flexibility with upgrades and administration. Unless you have a good reason, use a deployment.

- ReplicaSet

Orchestrates individual pod lifecycle and updates. These are newer versions of Replication Controllers, which differ only in selector support.

- Pod

As we've already mentioned, it is the lowest unit we can manage; it runs the application container, and possibly support containers.

DaemonSets

Should you want to have a logging application on every node, a DaemonSet may be a good choice. The controller ensures that a single pod, of the same type, runs on every node in the cluster. When a new node is added to the cluster, a Pod, same as deployed on the other nodes, is started. When the node is removed, the DaemonSet makes sure the local Pod is deleted. DaemonSets are often used for logging, metrics and security pods, and can be configured to avoid nodes.

As usual, you get all the CRUD operations via **kubectl**:

```
$ kubectl get daemonsets

$ kubectl get ds
```

StatefulSets

According to Kubernetes documentation, a StatefulSet is the workload API object used to manage stateful applications. Pods deployed using a StatefulSet use the same Pod specification. How this is different than a Deployment is that a StatefulSet considers each Pod as unique and provides ordering to Pod deployment.

In order to track each Pod as a unique object, the controllers uses an identity composed of stable storage, stable network identity, and an ordinal. This identity remains with the node, regardless of which node the Pod is running on at any one time.

The default deployment scheme is sequential, starting with 0, such as **app-0**, **app-1**, **app-2**, etc. A following Pod will not launch until the current Pod reaches a running and ready state. They are not deployed in parallel.

StatefulSets are stable as of Kubernetes v1.9.

Autoscaling

In the autoscaling group we find the **Horizontal Pod Autoscalers** (**HPA**). This is a stable resource. HPAs automatically scale Replication Controllers, ReplicaSets, or Deployments based on a target of 50% CPU usage by default. The usage is checked by the kubelet every 30 seconds, and retrieved by the Metrics Server API call every minute. HPA checks with the Metrics Server every 30 seconds. Should a Pod be added or removed, HPA waits 180 seconds before further action.

Other metrics can be used and queried via REST. The autoscaler does not collect the metrics, it only makes a request for the aggregated information and increases or decreases the number of replicas to match the configuration.

The **Cluster Autoscaler** (**CA**) adds or removes nodes to the cluster, based on the inability to deploy a Pod or having nodes with low utilization for at least 10 minutes. This allows dynamic requests of resources from the cloud provider and minimizes expenses for unused nodes. If you are using CA, nodes should be added and removed through **cluster-autoscaler-** commands. Scale-up and down of nodes is checked every 10 seconds, but decisions are made on a node every 10 minutes. Should a scale-down fail, the group will be rechecked in 3 minutes, with the failing node being eligible in five minutes. The total time to allocate a new node is largely dependent on the cloud provider.

Another project still under development is the Vertical Pod Autoscaler. This component will adjust the amount of CPU and memory requested by Pods.

Jobs

Jobs are part of the **batch** API group. They are used to run a set number of pods to completion. If a pod fails, it will be restarted until the number of completion is reached.

While they can be seen as a way to do batch processing in Kubernetes, they can also be used to run one-off pods. A Job specification will have a parallelism and a completion key. If omitted, they will be set to one. If they are present, the parallelism number will set the number of pods that can run concurrently, and the completion number will set how many pods need to run successfully for the Job itself to be considered done. Several Job patterns can be implemented, like a traditional work queue.

Cronjobs work in a similar manner to Linux jobs, with the same time syntax. There are some cases where a job would not be run during a time period or could run twice; as a result, the requested Pod should be idempotent.

An option spec field is **.spec.concurrencyPolicy**, which determines how to handle existing jobs, should the time segment expire. If set to **Allow**, the default, another concurrent job will be run. If set to **Forbid**, the current job continues and the new job is skipped. A value of **Replace** cancels the current job and starts a new job in its place.

RBAC

The last API resources that we will look at are in the **rbac.authorization.k8s.io** group. We actually have four resources: ClusterRole, Role, ClusterRoleBinding, and RoleBinding. They are used for Role Based Access Control (RBAC) to Kubernetes.

```
$ curl localhost:8080/apis/rbac.authorization.k8s.io/v1

...
    "groupVersion": "rbac.authorization.k8s.io/v1",
    "resources": [
...
    "kind": "ClusterRoleBinding"
...
    "kind": "ClusterRole"
...
    "kind": "RoleBinding"
...
    "kind": "Role"
...
```

These resources allow us to define Roles within a cluster and associate users to these Roles. For example, we can define a Role for someone who can only read pods in a specific namespace, or a Role that can create deployments, but no services. We will talk more about RBAC later in the course.