Kubernetes

Raúl Estrada

Octubre 2020

1. Overview

In this tutorial, we'll have a brief theoretical introduction to Kubernetes. In particular, we'll discuss the following topics:

- Need for a container orchestration tool
- Features of Kubernetes
- Kubernetes architecture
- Kubernetes API

For a more in-depth understanding, we can also have a look at the official documentation.

2. Container Orchestration

In a nutshell, Docker is a container runtime: it provides features for packaging, shipping and running single instances of an app in a standardized way, also known as a container.

However, as complexity increases, new needs appear; automated deployment, the orchestration of containers, scheduling apps, granting high-availability, managing a cluster of several app instances, and so on.

There are quite a few tools available on the market. However, Kubernetes is establishing itself more and more like a substantial competitor.

3. Kubernetes Features

Kubernetes, in short, is a system **for orchestration of containerized applications across a cluster of nodes, including networking and storage infrastructure**. Some of the most important features are:

- Resource scheduling: it ensures, that *Pods* are distributed optimally over all available nodes
- Auto-scaling: with increasing load, the cluster can dynamically allocate additional nodes, and deploy new *Pods* on them
- Self-healing: the cluster supervises containers and restarts them, if required, based on defined policies
- Service-discovery: *Pods* and *Services* are registered and published via DNS
- Rolling updates/rollbacks: supports rolling updates based on sequential redeployment of Pods and containers
- Secret/configuration management: supports secure handling of sensitive data like passwords or API keys
- Storage orchestration: several 3rd party storage solutions are supported, which can be used as external volumes to persist data

4. Understanding Kubernetes

The ***Master*** maintains the desired state of a cluster. When we interact with our cluster, e. g. by using the *kubectl* command-line interface, we're always communicating with our cluster's master.

Nodes in a cluster are the machines (VMs, physical servers, etc.) that run our applications. The Master controls each node.

A node needs a ***container runtime***. Docker is the most common runtime used with Kubernetes.

Minikube is a Kubernetes distribution, which enables us to run a single-node cluster inside a VM on a workstation for development and testing.

The ***Kubernetes API*** provides an abstraction of the Kubernetes concepts by wrapping them into objects (we'll have a look in the following section).

kubectl is a command-line tool, we can use it to create, update, delete, and inspect these API objects.

5. Kubernetes API Objects

An API object is a "record of intent" – once we create the object, the cluster system will continuously work to ensure that object exists.

Every object consists of two parts: the object spec and the object status. The spec describes the desired state for the object. The status describes the actual state of the object and is supplied and updated by the cluster.

In the following section, we'll discuss the most important objects. After that, we'll have a look at an example, how spec and status look like in reality.

5.1. Basic Objects

A ***Pod*** is a basic unit that Kubernetes deals with. It encapsulates one or more closely related containers, storage resources, a unique network IP, and configurations on how the container(s) should run, and thereby represents a single instance of an application.

Service is an abstraction which groups together logical collections of Pods and defines how to access them. Services are an interface to a group of containers so that consumers do not have to worry about anything beyond a single access location.

Using ***Volumes***, containers can access external storage resources (as their file system is ephemeral), and they can read files or store them permanently. Volumes also support the sharing of files between containers. A long list of Volume types is supported.

With ***Namespaces***, Kubernetes provides the possibility to run multiple virtual clusters on one physical cluster. Namespaces provide scope for names of resources, which have to be unique within a namespace.

5.2. Controllers

Additionally, there are some higher-level abstractions, called controllers. Controllers build on the basic objects and provide additional functionality:

A ***Deployment*** controller provides declarative updates for Pods and ReplicaSets. We describe the desired state in a Deployment object, and the Deployment controller changes the actual state to the desired.

A ***ReplicaSet*** ensures that a specified number of Pod replicas are running at any given time.

With ***StatefulSet***, we can run stateful applications: different from a Deployment, Pods will have a unique and persistent identity. Using StatefulSet, we can implement applications with unique network identifiers, or persistent storage, and can guarantee ordered, graceful deployment, scaling, deletion, and termination, as well as ordered and automated rolling updates.

With ***DaemonSet***, we can ensure that all or a specific set of nodes in our cluster run one copy of a specific Pod. This might be helpful if we need a daemon running on each node, e. g. for application monitoring, or for collecting logs.

A ***GarbageCollection*** makes sure that certain objects are deleted, which once had an owner, but no longer have one. This helps to save resources by deleting objects, which are not needed anymore.

A ***Job*** creates one or more Pods, makes sure that a specific number of them terminates successfully, and tracks the successful completions. Jobs are helpful for parallel processing of a set of independent but related work items, like sending emails, rendering frames, transcoding files, and so on.

5.3. Object Metadata

Metadata are attributes, which provide additional information on objects.

Mandatory attributes are:

- Each object must have a ***Namespace*** (we already discussed that before). If not specified explicitly, an object belongs to the *default* Namespace.
- A ***Name*** is a unique identifier for an object in its Namespace.
- A ***Uid*** is a value unique in time and space. It helps to distinguish between objects, which have been deleted and recreated.

There are also optional metadata attributes. Some of the most important are:

- **Labels** are a key/value pairs, which can be attached to objects to categorize them. It helps us to identify a collection of objects, which satisfy a specific condition. They help us to map our organizational structures on objects in a loosely coupled way.
- **Label selectors** help us to identify a set of objects by their labels.
- **Annotations** are key/value pairs, too. In contrast to labels, they are not used to identify objects. Instead, they can hold information about their respective object, like build, release, or image information.

5.4. Example

After having discussed the Kubernetes API in theory, we'll now have a look at an example.

API objects can be specified as JSON or YAML files. **However, the documentation recommends YAML for manual configuration.**

In the following, we'll define the spec part for the Deployment of a stateless application. After that, we will have a look how a status returned from the cluster might look like.

The specification for an application called *demo-backend* could look like this:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-backend
spec:
  selector:
      matchLabels:
        app: demo-backend
        tier: backend
  replicas: 3
  template:
    metadata:
      labels:
        app: demo-backend
        tier: backend
    spec:
      containers:
        - name: demo-backend
          image: demo-backend:latest
          ports:
            - containerPort: 8080
```

As we can see, we specify a *Deployment* object, called *demo-backend*. The *spec:* part below actually is a nested structure, and contains the following API objects discussed in the previous sections:

- *replicas: 3* specifies a *ReplicationSet* with replication factor 3 (i. e. we'll have three instances of the *Deployment*)
- *template:* specifies one *Pod*
- Within this *Pod,* we can use *spec: containers:* to assign one or more containers to our *Pod*. In this case, we have one container called *demo-backend*, which is instantiated from an image also called *demo-backend*, version *latest*, and it listens to port 8080
- We also attach *labels* to our pod: *app: demo-backend* and *tier: backend*
- With *selector: matchLabels:*, we link our *Pod* to the *Deployment* controller (mapping to labels *app: demo-backend* and *tier: backend*)

If we query the state of our *Deployment* from the cluster, the response will look something like this:

```
Name:                   demo-backend
Namespace:              default
CreationTimestamp:      Thu, 22 Mar 2018 18:58:32 +0100
Labels:                 app=demo-backend
Annotations:            deployment.kubernetes.io/revision=1
Selector:               app=demo-backend
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=demo-backend
  Containers:
   demo-backend:
    Image:        demo-backend:latest
    Port:         8080/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Progressing    True    NewReplicaSetAvailable
  Available      True    MinimumReplicasAvailable
OldReplicaSets:  <none>
NewReplicaSet:   demo-backend-54d955ccf (3/3 replicas created)
Events:          <none>
```

As we can see, the deployment seems to be up and running, and we can recognize most of the elements from our specification.

We have a Deployment with the replication factor of 3, with one pod containing one container, instantiated from image *demo-backend:latest*.

All attributes, which are present in the response but weren't defined in our specification, are default values.

6. Getting Started With Kubernetes

We can run Kubernetes on various platforms: from our laptop to VMs on a cloud provider, or a rack of bare metal servers.

To get started, Minikube might be the easiest choice: it enables us to run a single-node cluster on a local workstation for development and testing.

Have a look at the official documentation for further local-machine solutions, hosted solutions, distributions to be run on IaaS clouds, and some more.