www.caylent.com

Getting Started with Kubernetes

caylent

Getting Started with Kubernetes

Introduction

Not so long ago, software development involved launching monolith web applications of huge codebases that developed into hulking, hard-to-manage behemoths. Crude container technology had been available since the late 1970s, but the tech wasn't properly adopted until Docker debuted in 2013. From then on, the use of containers dramatically changed traditional IT processes by transforming the way we build, ship, and run distributed applications. However, in just a relatively few short years following Docker's surge in popularity, Kubernetes entered the container orchestration fray and laid waste to any competitors on the field.

Kubernetes has swiftly become the most crucial cloud-native technology in software development. In 2018, container production usage jumped to 38% from 22% just two years before with Kubernetes increasingly becoming the first choice among container users. The rate of Kubernetes adoption is steadily growing at 8% with alternative platform numbers falling or remaining flat. Scale is a key factor behind this growth. When it comes to enterprise statistics, 53% of today's organizations—with more than 1,000 containers—now use Kubernetes in production. (Heptio, 2018[1])

If your organization is about to embrace containers and develop microservices-type applications then this Getting Started with Kubernetes Caylent Guide is for you. We discuss the platform from the ground up to provide an in-depth tour of the core concepts for deploying, scaling, and maintaining reliable containerized applications on Kubernetes.

[1] Heptio. (2018). The State of Kubernetes 2018. Seattle, WA: Heptio. Retrieved from https://go.heptio.com/rs/383-ENX-437/images/Heptio_StateOfK8S_R6.pdf

What Is Kubernetes?

Kubernetes was launched into the open-source GitHub stratosphere by Google in 2014 as an independent solution for managing and orchestrating containers technologies such as Docker and rkt. The platform is really the third iteration of a container system developed by Google called Borg. Borg is Google's internal "cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines." (Verma et al., 2015[2])

After Borg came Omega, which was never publicly launched, but was used as the test bed for a lot of innovations that were folded back into Borg and, concurrently, Kubernetes. Kubernetes is the fruition of lessons learned from all three container management systems over the course of a decade.

Google partnered with Linux in 2015 around the launch time of Kubernetes v1.0 to establish the Cloud Native Computing Foundation (CNCF) as a true landing pad for the project—essentially, the C++ Borg rewritten in Go. The CNCF encourages the open source development and collaboration which surrounds the broad functionality of Kubernetes, making it the extensive and highly popular project that it is today.

Kubernetes, from the Greek term for "helmsman," is intended to make steering container management for multiple nodes as simple as managing containers on a single system. As the platform is based on Docker containers, it also works perfectly with Node apps—so users can run any kind of application on it.

[2] Verma, A., Pedrosa, L., R. Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. (2015). Large-Scale Cluster Management At Google With Borg. Retrieved from https://ai.google/research/pubs/pub43438

Commonly referred to as K8s or Kube, the platform has also started taking over the DevOps scene in the last couple of years by allowing users to implement best practices for speeding up the development process through automated testing, deployments, and updates. Working on Kube allows developers to manage apps and services with almost zero downtime. As well as providing self-healing capabilities, Kubernetes can also detect and restart services if a process fails inside a container.

For creating portable and scalable application deployments that can be scheduled, managed, and maintained easily, it's easy to see why it's becoming the go-to technology of choice. Kubernetes can be used on-premise in a corporate data center, but also integrated with all of the leading public cloud offerings too. Its cross-functionality and heterogeneous cloud support are why the platform has rapidly risen to become the standard for container orchestration.

Kubernetes Features

Kubernetes delivers a comprehensive and constantly upgraded set of features for container orchestration. These include, but aren't limited to:

Self-healing	Containers from failed nodes are automatically replaced and rescheduled. Based on existing rules/policy, Kube will also kill and restart any containers which do not respond to health checks.
Horizontal scaling	Kubernetes can auto scale applications according to resource usage such as CPU and memory. It can also support dynamic scaling defined by customer metrics.
Automated rollouts and rollbacks	Using K8s allows users to roll out and roll back new versions/ configurations of an application, without risking any downtime.

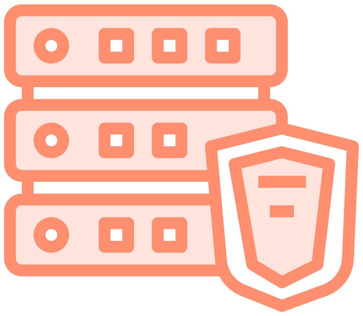
Auto binpacking

The platform auto schedules relevant containers according to resource usage and constraints, without sacrificing availability.

Secrets and configuration management

It's also possible to manage the secrets and configuration details for an application without re-building corresponding images. Through Kubernetes secrets, confidential information to applications can be shared without exposing it to the stack configuration, much like on GitHub.

Installing Kubernetes (For Different OS)



MAC OS X USERS

The two prerequisites for Mac users are that you need to have Homebrew and Homebrew Cask installed. The latter can be installed after Homebrew by running brew tap caskroom/cask in your Terminal. Now, follow these steps:

1. Install Docker for Mac first. Docker is the foundation on which we will create, manage, and run our containers. Installing Docker lets us create containers that will run in Kubernetes Pods.

2. Install VirtualBox for Mac using Homebrew. Next, in your Terminal, run brew cask install virtualbox. VirtualBox allows you to run virtual machines on a Mac (like running Windows inside macOS, except for with a Kubernetes cluster).

3. Now, install kubectl for Mac, the command-line interface tool that allows you to interact with Kubernetes. In your Terminal, run brew install kubectl.

4. Install Minikube according to the latest GitHub release documentation. At the time of writing, this uses the following command in Terminal.

```
curl -Lo minikube https://storage.googleapis.com/minikube/releases/
v1.0.0/minikube-darwin-amd64 && chmod +x minikube && sudo cp minikube
/usr/local/bin/ && rm minikube
```

   Minikube will launch by creating a Kubernetes cluster with a single node. Or you can install via homebrew with brew cask install minikube.

5. Everything should work! Kick off your Minikube cluster with minikube start—though bear in mind this may take a few minutes to work. Then type kubectl api-versions. If you get a list of versions in front of you, everything's working!

WINDOWS

Prerequisites for Kubernetes Windows installation include:

- Hyper-V: Follow Hyper-V installation instructions here
- Chocolatey: Follow Chocolatey Package Manager installation instructions here

To setup local Kubernetes on a Windows machine, follow these steps:

1. Launch Windows PowerShell with Admin Privileges (Right Click -> Run as Administrator).
2. Setup the package Minikube using Chocolatey.

```
##Minikube has a kubernetes-cli dependency which will get auto-installed
along with Minikube ##

choco install minikube
```

As mentioned earlier, kubectl is the command-line interface tool that allows you to interact with Kubernetes.

3. Your PowerShell should now read the following, enter ‘Y’ to continue running the script:

```
The package kubernetes-cli wants to run ‘chocolateyInstall.psl’
Do you want to run the script? ([Y]es/[N]o/[p]rint]): Y
***
The install of kubernetes-cli was successful
***
The install of minikube was successful
```

4. Now to test the minikube installation, just run:

```
minikube version
```

Or update minikube via:

```
minikube update-check
```

6. Now, it’s time to start your new K8s local cluster. Open the PowerShell terminal and run command:

```
minikube start
```

Great! Everything is now installed and it all looks like it’s working. Let’s run through a quick explanation of the components included in these install steps for both Mac and Windows users:

VirtualBox	is a universal tool for running virtual machines on compatible OS including Ubuntu, Windows, and Mac.
Homebrew	is Mac’s go-to package manager for installations and Homebrew Cask extends Homebrew with support for quickly installing Mac applications like Google Chrome, VLC, and, of course, Kubernetes as well as others.
Hyper-V	is Microsoft’s very own virtualization software. Enable this tool to create virtual machines on x86–64 systems on Windows 10. Formerly known as Windows Server Virtualization, Microsoft Hyper-V is a native hypervisor.
Chocolatey	is a package manager like apt-get or yum but solely for Windows. It was designed to act as a decentralized framework for quickly installing applications and necessary tools. Chocolatey is built on the NuGet infrastructure and currently uses PowerShell as its focus for delivering packages.
kubectl	is Kubernetes’ command line application for interacting with your Minikube Kubernetes cluster. It sends HTTP requests from your machine to the Kubernetes API server running on the cluster to manage your Kubernetes environment.

Kubernetes Fundamentals

CONTAINERS, PODS, AND REPLICASETS

Containers are an application-centric packaged approach to launching high-performing, scalable applications on your infrastructure of choice. Using a container image, we confine all the application information along with all its runtime and dependencies together in a predefined format. We leverage that image to create an isolated executable environment known as a container. With container runtimes such as rkt, runC or containerd, we can use those pre-packaged images to generate one or more containers. Sometimes, Docker is also referenced as a container runtime, but technically Docker is a platform which uses containerd as a container runtime.

Containers can be deployed from a given image on multiple platforms of choice, such as from desktops, in the cloud, on VMs, etc. All of these runtimes are good at running containers on a single host. However, in practice, we would prefer a fault-tolerant and scalable solution by connecting multiple nodes together to create a single controller/management unit. This is where Kubernetes comes in as the container orchestrator.

A pod is a collected unit of containers which share a network and mount namespace. They are also the basic unit of deployment in Kubernetes. A pod denotes a single instance for a running process in your Kubernetes cluster. Pods tend to contain one or more containers, such as Docker containers and act as the scheduling unit in Kubernetes. All containers within a pod are logically scheduled on the same node together. When a pod runs multiple containers, the containers are run as a single entity and share the pod's allocated resources.

Pods' shared networking and storage resources rules are as follows:

NETWORK

Pods are automatically assigned unique IP addresses on creation. Pod containers also share the same network namespace, including IP address and network ports. Containers within a pod communicate with each other on localhost.

STORAGE

Pods can identify a set of shared storage volumes to be shared among containers.

ReplicaSets ensure how many versions—or **replicas**—of a pod should be running. It is a **Kubernetes controller** which we use to define a specified number of pod replicas determined by preconfigured values. (A controller in Kubernetes takes care of the tasks that guarantee the desired state of the cluster matches the observed state—one of Kubes' self-healing features.) Without it, we would need to create multiple manifests for the number of pods required—which is a lot of repeat work to deploy replicas of a single application. ReplicaSets will manage all pods according to defined **labels** (**key-value pair** data used to describe attributes of Kube objects that are significant to users).

MASTERS, WORKER NODES, AND CLUSTERS

The **master node** manages the state of a **cluster** and is essentially the entry point for all administrative tasks. There are three ways to communicate with the master node:

- Via the CLI
- Via the GUI (Kubernetes Dashboard)
- Via APIs

To improve **fault tolerance**, it's possible to have more than one master node in the cluster in **High Availability** (HA) mode. In a multiple master node setup though, only one of them will act as a leader performing all the operations; the rest of the master nodes would be followers.

Worker nodes are controlled by the master node and may be a VM or physical machine which runs the applications using pods. Worker nodes are responsible for scheduling pods using the necessary components to run and connect them (see below). We also connect to worker nodes and not to the master node/s if accessing applications from the external world

Working with individual nodes can be very useful for certain tasks, but it's not the best way to optimize Kubernetes. A **cluster** allows you to pool nodes and their resources together to form a single more powerful engine. Thinking about a cluster as a whole becomes more efficient than running individual nodes.

A cluster typically comprises a master node and a set of worker nodes that run in a distributed setup over multiple nodes in a production environment. By using minikube when testing, all the components can run on the same node (physical or virtual). Clusters can be scaled by adding worker nodes to increase the workload capacity of the cluster, thereby providing Kube more room to schedule containers.

Seven main components form a functioning master-slave architectural type cluster:

Master components (which call the shots):

etcd is a distributed key-value store which manages the cluster state. All the master nodes connect to it.

The **API server** performs all the administrative tasks within the master node by validating and processing user/operator REST commands. Cluster changes are stored in the distributed key store (etcd) once executed.

The **scheduler** programs the work between worker nodes and contains the resource usage information for each one. It works according to user/operator configured constraints.

The **controller manager** instigates the control/reconciliation loops that compare the actual cluster state to the desired cluster state in the apiserver and ensure the two match.

Worker/slave node components (which execute the application workloads):

kubelet interacts with the underlying container runtime engine to bring up containers as needed and monitors the health of pods.

kube-proxy is a network proxy/load balancer which manages network connectivity to the containers through services.

Container runtime, as mentioned previously such as Docker or rkt, executes the containers.

You can run each of these components as standard Linux processes, or as Docker ones.

SERVICES, INGRESSES, AND NETWORKING

As pods are ephemeral in nature, any resources like IP addresses allocated to them are not reliable—as pods can be rescheduled or just die abruptly. To overcome this challenge, Kubernetes offers a higher-level abstraction known as a service, which groups pods in a logical collection with a policy to access them via labels and selectors. On configuration, pods will launch with pre-configured labels (key/value pairs to organize by release, deployment or tier, etc.). For example:

```
labels:
    app: nginx
    tier: backend
```

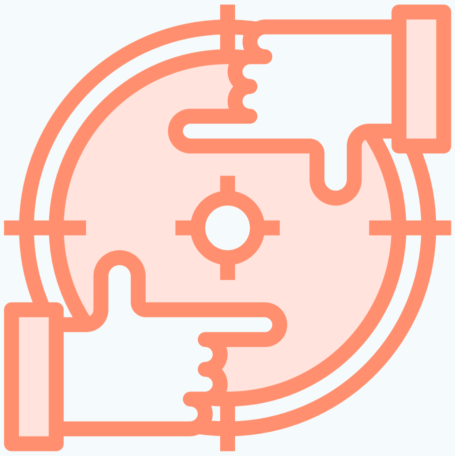
Users can then use selectors to tell the resource, whatever it may be (e.g., service, deployment, etc.) to identify pods according to that label. By default, each named service is assigned an IP address, which is routable only inside the cluster. Services then act as a common access point to pods from the external world through a connection endpoint to communicate with the appropriate pod and forwards traffic. The network proxy deamon kube-proxy listens to the API server for every service endpoint creation/deletion from each worker node, then it sets up route channel accordingly.

By default, Kubernetes isolates pods and the outside world. Connecting with a service in a pod means opening up a route channel for communication. The collection of routing rules which govern how external users access services is referred to as ingress. There are three general strategies in Kubernetes for exposing your application through ingress:

- Through **nodeport** which exposes the application on a port across each of your nodes
- Through **loadbalancer** which creates an external load balancer that navigates to a Kubernetes service in your cluster
- Through a Kubernetes **ingress resource**

Ingress is an integral concept in K8s as it allows simple host or URL based HTTP routing, but it is always instigated by a third-party proxy. These third-party proxies are known as ingress controllers, and they're responsible for reading the ingress resource data and processing that info accordingly. Different ingress controllers have extended the routing rules in different ways to support alternative use cases. For an in-depth guide to managing Kubernetes ingresses, read our article here.

Since a Kubernetes cluster consists of various components in the form of nodes and pods, understanding how they communicate is essential. As mentioned, Kubernetes assigns an IP address to each pod. So, unlike the Docker networking model, there is no need to map host ports to container ports. Admittedly, the Kube networking implementation is a bit more complex than Docker. But this is in order to simplify the process of optimizing even complicated legacy applications to run in a container environment.



Kubernetes networking—as there is no default model, all implementations must work through a third-party network plug-in (e.g., Project Calico, Weave Net, Flannel, etc.)—is responsible for routing all internal requests between hosts to the right pod. service, load balancer, or ingress controllers organize external access (see above). Pods act much the same as VMs or physical hosts with regards to naming, load balancing, port allocation, and application configuration.

When it comes to networking communication of pods, Kubernetes sets certain conditions and requirements:

1. All pods can communicate with each other without the need to use network address translation (NAT)
2. Nodes are also able to communicate with all pods, without the need for NAT
3. Each pod will see itself with the same IP address that other pods see it with

This leaves us with three networking challenges to overcome in order to take advantage of Kubernetes:

Container-to-container networking:

All the containers within a given service will have the same IP address and port space—as assigned by the pod's assigned network namespace. So, because all the containers all reside within the same namespace, they are able to communicate with one another via localhost.

Pod-to-service networking:

Services act as an abstraction layer on top of pods, assigning a single virtual IP address to a specified group. Once these pods are associated with that virtual IP address, any traffic which is addressed to it will be routed to the corresponding group. The set of pods linked to a service can be changed at any time, but the service IP address will remain static.

Pod-to-pod networking:

Each pod exists in its own Ethernet namespace. This namespace then needs to communicate with other network namespaces that are located on the same node. Linux provides a mechanism for connecting namespaces using a virtual Ethernet device (VED or 'veth pair'). The VED comprises of a pair of virtual interfaces.

In order to connect two Pod namespaces, one side of the VED is assigned to the root network namespace. The other member of the veth pair is then assigned to the Pod's network namespace. The VED then acts like a virtual cable that connects the root network namespace to that of the Pod's network namespace and allowing them to exchange data.

Tools for Working with Kubernetes

SUGGESTED TOOLS

Due to its rising popularity and open-source nature, the list of in-built and external tools for enhancing Kubernetes usage is extensive and far too widespread to cover here. For a glimpse into the top 50 + tools that Caylent suggest to begin with for improving your work with the platform, check out our curated list here.

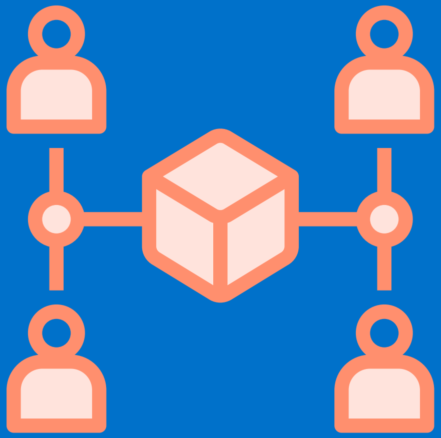
TO GET STARTED

As pods are ephemeral in nature, any resources like IP addresses allocated to them are not reliable—as pods can be rescheduled or just die abruptly. To overcome this challenge, Kubernetes offers a higher-level abstraction known as a service, which groups pods in a logical collection with a policy to access them via labels and selectors. On configuration, pods will launch with pre-configured labels (key/value pairs to organize by release, deployment or tier, etc.). For example:

- MiniKube, as mentioned previously, is the easiest and most recommended way to initiate an all-in-one Kubernetes cluster locally.
- Bootstrap a minimum viable Kubernetes cluster that conforms to best practices with kubeadm—a first-class citizen of the Kubernetes ecosystem. As well as a set of building blocks to setup clusters, it is easily extendable to provide more functionality.

- Using KubeSpray means we can install Highly Available Kubernetes clusters on GCE, Azure, AWS, OpenStack or bare metal machines. The Kubernetes Incubator project tool is based on Ansible, and it's available on most Linux distributions.
- Kops allows us to create, destroy, upgrade, and maintain highly available, production-grade Kubernetes clusters from the CLI. It can also provision the machines too.

Clusters



CLUSTER CONFIGURATION OPTIONS

With Kubernetes, users can leverage different configurations through four major installation types as presented below:

All-in-One Single-Node Configuration

With an all-in-one configuration, both the master and worker components are run on a single node. This setup is beneficial for development, testing, and training and can be run easily on Minikube. Do not use this setup in production.

A Single-Node etcd, Single-Master, and Multi-Worker Configuration

For this installation, there is a single master node that runs a single-node etcd instance and is connected to multiple worker nodes.

Single-Node etcd, Multi-Master, and Multi-Worker Configuration

In this high-availability setup, there are multiple master nodes but only a single-node etcd instance. Multiple worker nodes are connected to the multiple master nodes. One master will be the leader.

Multi-Node etcd, Multi-Master, and Multi-Worker Configuration

In this installation, etcd is configured outside the Kubernetes cluster in a clustered mode with many master and worker nodes connected. This is considered the most sophisticated and recommended production setup.

ONE CLUSTER OR MANY?

One of Kubernetes' major strengths is just how much flexibility can be gained for deploying and operating containerized workloads on the platform. Every variable from the number of pods, containers, and nodes per cluster as well as a host of other parameters can be customized to your singular configuration.

By default, when you create a cluster, the master and its nodes are launched in a single compute or availability zone that you pre-configure. It's possible to improve the availability and resilience of your clusters by establishing regional clusters. A regional cluster supplies a single static endpoint for the whole cluster and distributes your cluster's pods across multiple zones of a given region. It's your choice whether a cluster is zonal or regional when you create it. It's important to note too that existing zonal cluster can't be converted to regional or vice versa.

When it comes to the question of how many clusters though, there are a number of considerations to take into account. The **Kubernetes documentation** offers the following advice:

“The selection of the number of Kubernetes clusters may be a relatively static choice, only revisited occasionally. By contrast, the number of nodes in a cluster and the number of pods in a service may change frequently according to load and growth.”

One of the biggest considerations needs to be the impact on internal and external customers when Kubernetes is running in production. For example, an enterprise environment may require a multi-tenant cluster for distinct teams within the organization to run effectively. A multi-tenant cluster can be divided across multiple users and/or workloads—these are known as tenants. Operating multiple clusters can assist to:

- Separate tenants and workloads
- Improve high availability
- Establish maintenance lifecycles to match particular workloads

While it is feasible to run multiple clusters per availability zones, it is advisable to run fewer clusters with more VMs per availability zone. Choosing fewer clusters per availability zone can help with the following:

- Improved pod binpacking thanks to more nodes in one cluster (lower resource fragmentation).
- Lower operational overheads
- Reduced per-cluster costs for ongoing resources

Regional clusters, however, replicate cluster masters and nodes across more than one zone within in a single region. Choosing this option can help with:

Improved resilience from single zone failure so your control plane and resources aren't impacted

Reduced downtime from master failures as well as zero downtime for master upgrades, and resizing

Deployments

CREATE AND EXPOSE A DEPLOYMENT

When you have a cluster up and running with Minikube, it's then possible to deploy your containerized app on top of it. In K8s, a deployment is the recommended way to deploy pod or ReplicaSet. Simply define a deployment configuration to tell K8s how to create and update your app instances. Once the deployment configuration is defined, the Kube master will schedule the appropriate app instances onto your cluster nodes.

If you want to revise a deployment, describes the state that you want in a ReplicaSet. Then during a rollout, the deployment controller will adapt the current state to match the described state that you want at a controlled rate. All deployment revisions can also be rolled back and scaled too.

Use the CLI Kubectl terminal to create and manage a deployment. To create a deployment, specify the application's container image and the number of replicas that you want to run. Run your first app with the `kubectl run` command to create a new deployment. Here's an example of the output which creates a ReplicaSet to bring up 3 pods.

```
apiVersion: v1
kind: Deployment
metadata:
 name: example-deployment # Name of our deployment



spec:
 replicas: 3
 selector:
   app: nginx
 template:
   metadata:
     labels:
       app: nginx
       status: serving
   spec:
     containers:
     - image: nginx:1.9.7
       name: nginx
       ports:
         - containerPort: 80
```

SCALE AND UPDATE A DEPLOYMENT

Before anything else, first check to see if the deployment was created successfully by running the `kubectl rollout status` and `kubectl get deployment` command. The first will show if it failed or not and the second will indicate how many replicas are available, have been updated, and how many are open to end users.

```
$ kubectl rollout status deployment example-deployment
deployment “example-deployment” successfully rolled out
```

```
$ kubectl get deployment
NAME                               DESIRED    CURRENT    UP-TO-DATE    AVAILABLE
example-deployment                 3          3          3             3
```

To scale a deployment, use the following command:

```
kubectl scale deployment.v1.apps/nginx-deployment --replicas=10

deployment.apps/nginx-deployment scaled
```

If you have changed your mind about the image number and want to update the nginx Pods to use the nginx:1.9.1 image instead of the nginx:1.7.9 image. use the kubectl command --record as follows:

```
kubectl --record deployment.apps/nginx-deployment set image deployment.v1.apps/
nginx-deployment nginx=nginx:1.9.1
```

EXECUTING ZERO DOWNTIME DEPLOYMENTS

Remove any downtime from your production environment so that users don't feel let down when they need your app the most. To do that, simply run a readiness probe. This is essentially a check that Kubernetes implements to ensure that your pod is ready to send traffic to it. If it's not ready, then Kubernetes won't use that pod. Easy!

```
readinessProbe:
  httpGet:
    path: /
    port: 80
  initialDelaySeconds: 5
  periodSeconds: 5
  successThreshold: 1
```

This tells Kubernetes to send an http get request down the path every five seconds. If the readinessprobe is successful, then Kube will mark the pod as ready and start sending traffic to it.

Another strategy for avoiding downtime is the rolling update technique that looks like this:

```
 strategy:
 type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 1
```

If you combine these two strategies, your deployment.yaml should look something like this in the end:

```
strategy:
apiVersion: v1
kind: ReplicationController
metadata:
 name: webserver-rc
spec:
 strategy:
 type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 1
 template:
   metadata:
     labels:
       app: webserver
       status: serving
   spec:
     containers:
     - image: nginx:1.9.7
       name: nginx
       ports:
         - containerPort: 80
       readinessProbe:
         httpGet:
           path: /
           port: 80
         initialDelaySeconds: 5
         periodSeconds: 5
  successThreshold: 1
```

Summary

Congratulations! Over the course of this guide you've:

- Covered the history and main features of the Kubernetes platform
- Gotten to grips with the fundamental architecture and components of Kubernetes, including:
    - Containers, Pods, and ReplicaSets
    - Masters, Worker Nodes, and Clusters
    - Services, Ingresses, and Networking
- Installed Kubernetes on your machine of choice
- Discovered the best tools to use to optimise your workload with the platform
- Launched, scaled, updated, and minimized the downtime on your first Kubernetes deployment

Caylent & Kubernetes

If you would like to know more about optimizing Kubernetes for your work loads check out our extensive resource archive on various aspects of the platform in depth here. For anything else related to Kubernetes (services, consulting, and more), don't hesitate to contact us!

Advanced Material

STATELESS APPLICATIONS VERSUS STATEFUL APPLICATIONS

An application's underlying architecture is one of the most crucial factors to consider before running a new application in production. The terms often discussed in this context is working out if the app will be 'stateless' or 'stateful.' Both types have their own advantages and disadvantages.

One of the major factors for choosing Kubernetes in container orchestration is the capability to harmonize stateful and stateless apps together. It's possible to take advantage of exactly the same compute, storage, and network resources across web and API servers, database, message queue, cache, and file stores at the same time. The second reason is that Kubernetes provides consistent and declarative provisioning across all environments (testing, staging, production, etc.).

DEFINITION

A stateless application is one which requires no persistent storage. Your cluster is only responsible for hosting the code and other static content. That's it. No need to change databases and there are no writes or leftover files when pods are deleted.

Alternatively, a stateful application looks after several other parameters in a cluster. Dynamic databases, for example, which even when the app is deleted or goes offline must persist on the disk. When deployed in a cloud, stateful applications pose a challenge as they can be hard to scale in a distributed, virtualized environment. Also, limiting data stores to specific locations is no simple task in an environment where VMs themselves are stateless. Which means that stateful apps can become problematic in the cloud. On Kubernetes this can raise several issues.

RUNNING STATEFUL APPLICATIONS

At the heart of a stateful application are the following aspects:

- Ordered operations which are covered in the sequencing needed to deploy an app. First, run the set of operations that brings up the metadata tier, then run the second set of operations that brings up the actual data tier.
- Stable network IDs instead of IP addresses (which aren't reliable as mentioned earlier).
- Persistent volumes so that pods can access data at all times even if the data moves from one node to another or if a pod restarts.

STATEFULSET BASICS

Bypass these challenges by using StatefulSets in Kubernetes. These are essentially workload API objects for managing stateful applications. StatefulSets acts in the same manner as a controller. The controller initiates the required updates to move from the current state to reach the defined desired state in a StatefulSet object.

There are three crucial components behind a StatefulSet:

- Headless services that control the network domain.
- The StatefulSet component that indicates how many container replicas need to be launched in unique pods.
- The VolumeClaimTemplates that provide stable storage through PersistentVolumes delivered by a PersistentVolume Provisioner.

With StatefulSets, developers can achieve:

- Ordered pod creation with ordinal index: For a StatefulSet with n replicas, each pod will be appointed an integer ordinal, from 0 up through n-1, which is individual over the Set.
- A stable and unique hostname and network identity for pods across restarts so when re-spawning a pod will be treated by the cluster as a new member.
- Stable storage bound to the ordinal index/name through dynamically provisioned PersistentVolumes. Persistent disks will then stay attached to hostname pods even if the pods are rescheduled.

Templating Kubernetes Resources

HELM AND HELM CHARTS INTRO

As mentioned previously, we rely on YAML files to configure Kubernetes clusters. The YAML files are there to describe everything from the way Pods need to be configured to how load balancing is done by the cluster. Unfortunately, setting up a new Kubernetes cluster means creating a .yaml file for that cluster every time. For many, this means copying and pasting configurations from another cluster and making manual adjustments to it.

This iterative process is exactly what Helm can help you with. Rather than having to configure a YAML file manually, you can simply call a Helm Chart—a Chart is the term for Helm's package configuration—and let Helm do the rest for you. Helm is a custom built Kubernetes package manager similar to NPM in NodeJS and Maven in Java.

CREATING REUSABLE TEMPLATES

Helm Charts act as the template for Kubernetes clusters configurations. The template itself can be customized to mimic the setup parameters of the particular cluster. There is no need to make manual adjustments to variables such as host name or provisions since Helm will take the correct variables and make the necessary adjustments.

Helm lends itself perfectly to the deployment of cloud-native apps. It completely reduces deployment complexity and lets developers focus more on other more important things. On top of that, Helm Charts also introduce standardized and reusable templates to the Kubernetes.

The great news is that there are plenty of Helm Charts that you can use straight out of the box on GitHub. Do you need to add a MongoDB database to your setup? There's a Chart for that. Want to deploy a testing environment for your app? You can create a Chart to simplify that process too. Even the most complex Kubernetes setup can be packaged into its own Helm Chart for iteration.

Pre-configured Helm Charts are handy for setting up Kubernetes clusters quickly, but you can also create your own Charts. Use Helm to help with deploying test environments and distributing pre-release versions of your application.

But why stop at pre-release versions? Helm Charts can further be used as a delivery method for production apps. You can, for instance, install WordPress on an empty cluster by running the correct Helm Chart for it; everything will be automated, and you will swiftly have a WordPress site up and running in seconds.

Since you can use values to define variables, Helm Charts can also be used for deploying the app onto different clusters or setups. Each variable can be adjusted on the fly as the Helm Chart is processed, allowing for greater flexibility in general.

UPGRADING CHARTS AND REVERTING CHANGES WITH ROLLBACKS

In order to upgrade Charts, simply make the necessary adaptations to your values.yaml

then update your Chart version number. Now push the changes to the repository, package the Helm Chart, and double check the Chart Repository. Run sudo helm upgrade [new chart name] and supply the release name and the chart name you want to upgrade.

To rollback a Chart to a previous revision, you only need to provide the release name and the revision number that you want to rollback to. Execute the command sudo helm rollback [new chart name] 1.

It's that simple.

STORING REUSABLE TEMPLATES

Helm Charts maintains an official chart repository here and welcomes participation through chart submissions. It's also simple to create and manage your own chart repository too in the form of an HTTP server that will store your index.yaml files and packaged charts too. As it can be any HTTP server that works with YAML and tar files and answer GET requests, feel free to use an Amazon S3 bucket, a Google Cloud Storage (GCS) bucket or your own GitHub pages as you wish. Follow the Helm best practice chart repository structure as defined here to ensure easy location and sharing capabilities.

For a comprehensive list of tools to make working with Helm that much simpler, don't forget to check out our article on 15+ Useful Helm Charts Tools here.

References

Heptio. (2018). The State of Kubernetes 2018. Seattle, WA: Heptio. Retrieved from https://go.heptio.com/rs/383-ENX-437/images/Heptio_StateOfK8S_R6.pdf

Verma, A., Pedrosa, L., R. Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. (2015). Large-Scale Cluster Management At Google With Borg. Retrieved from https://ai.google/research/pubs/pub43438