STRATOSCALE

EVERYTHING KUBERNETES: A PRACTICAL GUIDE

CONTENTS

INTRODUCTION

Kubernetes is an open-source, container management solution originally announced by Google in 2014. After its initial release in July 2015, Google donated Kubernetes to the Cloud Native Computing Foundation. Since then, several stable versions have been released under Apache License.

For a developer, Kubernetes provides a manageable execution environment for deploying, running, managing, and orchestrating containers across clusters or clusters of hosts. For devops and administrators, Kubernetes provides a complete set of building blocks that allow the automation of many operations for managing development, test, and production environments. Container orchestration enables coordinating containers in clusters consisting of multiple nodes when complex containerized applications are deployed. This is relevant not only for the initial deployment, but also for managing multiple containers as a single entity for the purposes of scaling, availability, and so on.

Being infrastructure agnostic, Kubernetes clusters can be installed on a variety of public and private clouds (AWS, Google Cloud, Azure, OpenStack) and on bare metal servers. Additionally, Google Container Engine can provide a deployed Kubernetes cluster. This makes Kubernetes similar to Linux kernel, which provides consistency across different hardware platforms, or Java, which runs on almost any operating system.

KUBERNETES — HIGH LEVEL ARCHITECTURE

NODE

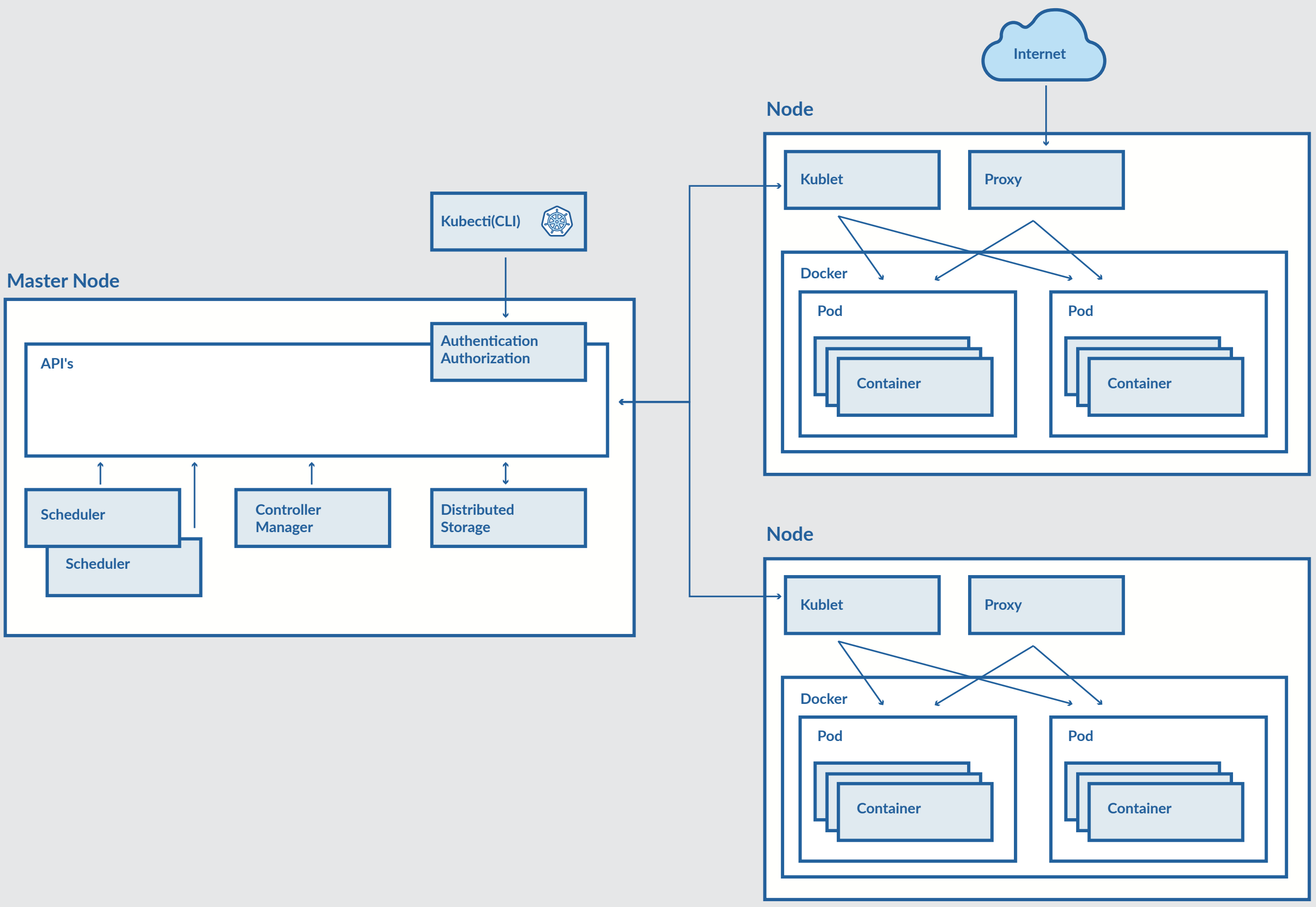
A Kubernetes cluster consists of one or more nodes managed by Kubernetes. The nodes are bare-metal servers, on-premises VMs, or VMs on a cloud provider. Every node contains a container runtime (for example, Docker Engine), kubelet (responsible for starting, stopping, and managing individual containers by requests from the Kubernetes control plane), and kube-proxy (responsible for networking and load balancing).

MASTER NODE

A Kubernetes cluster also contains one or more master nodes that run the Kubernetes control plane. The control plane consists of different processes, such as an API server (provides JSON over HTTP API), scheduler (selects nodes to run containers), controller manager (runs controllers, see below), and etcd (a globally available configuration store).

DASHBOARD AND CLI

A Kubernetes cluster can be managed via the Kubernetes Dashboard, a web UI running on the master node. The cluster can also be managed via the command line tool kubectl, which can be installed on any machine able to access the API server, running on the master node. This tool can be used to manage several Kubernetes clusters by specifying a context defined in a configuration file.


Internet
Node
Kublet
Proxy
Docker
Pod
Container
Pod
Container
Kubecti(CLI)
Master Node
Authentication
Authorization
API's
Scheduler
Scheduler
Controller
Manager
Distributed
Storage
Node
Kublet
Proxy
Docker
Pod
Container
Pod
Container

KUBERNETES BUILDING BLOCKS

Kubernetes provides basic mechanisms for the deployment, maintenance, and scaling of containerized applications. It uses declarative primitives, or building blocks, to maintain the state requested by the user, implementing the transition from the current observable state to the requested state.

THE BASICS

POD

A pod is the smallest deployable unit that can be managed by Kubernetes. A pod is a logical group of one or more containers that share the same IP address and port space. The main purpose of a pod is to support co-located processes, such as an application server and its local cache. Containers within a pod can find each other via localhost, and can also communicate with each other using standard inter-process communications like SystemV semaphores or POSIX shared memory. In other words, a pod represents a "logical host". Pods are not durable; they will not survive scheduling failures or node failures. If a node where the pod is running dies, the pod is deleted. It can then be replaced by an identical pod, with even the same name, but with a new unique identifier (UID).

LABEL

A label is a key/value pair that is attached to Kubernetes resource, for example, a pod. Labels can be attached to resources at creation time, as well as added and modified at any later time.

SELECTOR

A label selector can be used to organize Kubernetes resources that have labels. An equality-based selector defines a condition for selecting resources that have the specified label value. A set-based selector defines a condition for selecting resources that have a label value within the specified set of values.

CONTROLLER

A controller manages a set of pods and ensures that the cluster is in the specified state. Unlike manually created pods, the pods maintained by a replication controller are automatically replaced if they fail, get deleted, or are terminated. There are several controller types, such as replication controllers or deployment controllers.

REPLICATION CONTROLLER

A replication controller is responsible for running the specified number of pod copies (replicas) across the cluster.

DEPLOYMENT CONTROLLER

A deployment defines a desired state for logical group of pods and replica sets. It creates new resources or replaces the existing resources, if necessary. A deployment can be updated, rolled out, or rolled back. A practical use case for a deployment is to bring up a replica set and pods, then update the deployment to re-create the pods (for example, to use a new image). Later, the deployment can be rolled back to an earlier revision if the current deployment is not stable.

REPLICA SET

A replica set is the next-generation replication controller. A replication controller supports only equality-based selectors, while a replica set supports set-based selectors.

SERVICE

A service uses a selector to define a logical group of pods and defines a policy to access such logical groups. Because pods are not durable, the actual pods that are running may change. A client that uses one or more containers within a pod should not need to be aware of which specific pod it works with, especially if there are several pods (replicas).

There are several types of services in Kubernetes, including ClusterIP, NodePort, LoadBalancer. A ClusterIP service exposes pods to connections from inside the cluster. A NodePort service exposes pods to external traffic by forwarding traffic from a port on each node of the cluster to the container port. A LoadBalancer service also exposes pods to external traffic, as NodePort service does, however it also provides a load balancer.

USING LABELS AND SELECTORS FOR FINE-GRAINED CONTROL

A Kubernetes controller, for example, uses a selector to define a set of managed pods so that pods in that set have the corresponding label. A label is just a key/value pair that is attached to Kubernetes resources such as pods. Labels can be attached to resources when they are created, or added and modified at any time. Each resource can have multiple labels. For example:

```
release: stable
environment: dev
```

A label selector defines a set of resources by specifying a requirements for their labels. For example:

```
environment = dev
environment != live
environment in (dev, test)
environment notin (live)
release = stable, environment = dev
```

The first two selectors have an equality-based requirement, the third and fourth selectors have a set-based requirement. The last selector contains the comma separator, which acts as a logical "AND" operator, so the selector defines a set of resources where the label "release" equals "stable" and the label "environment" equals "dev."

SERVICE DISCOVERY

Kubernetes supports finding a service in two ways: through environment variables and using DNS.

ENVIRONMENT VARIABLES

Kubernetes injects a set of environment variables into pods for each active service. Such environment variables contain the service host and port, for example:

```
MYSQL_SERVICE_HOST=10.0.150.150
MYSQL_SERVICE_PORT=3306
```

An application in the pod can use these variables to establish a connection to the service.

The service should be created before the replication controller or replica set creates a pod's replicas. Changes made to an active service are not reflected in a previously created replica.

DNS

Kubernetes automatically assigns DNS names to services. A special DNS record can be used to specify port numbers as well. To use DNS for service discovery, a Kubernetes cluster should be properly configured to support it.

3 STORAGE BUILDING BLOCKS

VOLUME

A container file system is ephemeral: if a container crashes, the changes to its file system are lost. A volume is defined at the pod level, and is used to preserve data across container crashes. A volume can be also used to share data between containers in a pod. A volume has the same lifecycle as the the pod that encloses it—when a pod is deleted, the volume is deleted as well. Kubernetes supports different volume types, which are implemented as plugins.

PERSISTENT VOLUME

A persistent volume represents a real networked storage unit in a cluster that has been provisioned by an administrator. Persistent storage has a lifecycle independent of any individual pod. It supports different access modes, such as mounting as read-write by a single node, mounting as read-only by many nodes, and mounting as read-write by many nodes. Kubernetes supports different persistent volume types, which are implemented as plugins. Examples of persistent volume types include AWS EBS, vSphere volume, Azure File, GCE Persistent Disk, CephFS, Ceph RBD, GlusterFS, iSCSI, NFS, and Host Path.

PERSISTENT VOLUME CLAIM

A persistent volume claim defines a specific amount of storage requested and specific access modes. Kubernetes finds a matching persistent volume and binds it with the persistent volume claim. If a matching volume does not exist, a persistent volume claim will remain unbound indefinitely. It will be bound as soon as a matching volume become available.

CHOOSING THE RIGHT BLOCK FOR THE JOB

Designed as a simple building block; a replication controller's only responsibility is to maintain the specified number of replicas. A replication controller counts only live pods;, terminated pods are excluded. Other Kubernetes building blocks should be used together with replication controllers for more advanced tasks. For example, an autoscaler can monitor application-specific metrics and dynamically change the number of replicas in the existing replication controller. In addition, a replication controller does not support scheduling policies, meaning you cannot provide rules for choosing cluster nodes to run pods from the managed set.

A replica set is another Kubernetes building block. The major difference between it and a replication controller is that replication controllers do not support selectors with set-based requirements, while replica sets support such selectors. From this perspective, a replica set is just a more advanced version of a replication controller.

Using only pods and replication controllers to deploy an application is, at least in part, an imperative form of managing software, because it usually requires manual steps. A Kubernetes deployment is an alternative that enables completely declarative application deployment.

SECRET

A Kubernetes secret allows users to pass sensitive information, such as passwords, authentication tokens, SSH keys, and database credentials, to containers. A secret can then be referenced when declaring a container definition, and read from within containers as environment variables or from a local disk.

CONFIG MAP

A Kubernetes config map allows users to externalize application configuration parameters from a container image and define application configuration details, such as key/value pairs, directory content, or file content. Config map values can be consumed by applications through environment variables, local disks, or command line arguments.

JOB

A job is used to create one or more pods and ensure that a specified number of them successfully terminate. It tracks the successful completions, and when a specified number of successful completions is reached, the job itself is complete. There are several types of jobs, including non-parallel jobs, parallel jobs with a fixed completion count, and parallel jobs with a work queue. A job should be used instead of a replication controller if you need to spread pods across cluster nodes and ensure, for example, so that each node has only one running pod of the specified type.

DAEMON SET

A daemon set ensures that all or some nodes run a copy of a pod. A daemon set tracks the additional and removal of cluster nodes and adds pods for nodes that are added to the cluster, terminates pods on nodes that are being removed from a cluster. Deleting a daemon set will clean up the pods it created. A typical use case for a daemon set is running a log collection daemon or a monitoring daemon on each node of a cluster.

NAMESPACE

A namespace provides a logical partition of the cluster's resources. Kubernetes resources can use the same name when found in different namespaces. Different namespaces can be assigned different quotas for resource limitations.

QUOTA

A quota sets resource limitations, such as CPU, memory, number of pods or services, for a given namespace. It also forces users to explicitly request resource allotment for their pods.

IMPERATIVE VS. DECLARATIVE ORCHESTRATION

Before getting to the practical steps of the Kubernetes deployment, it's important to understand the key approaches to orchestration.

The classic **imperative approach** for managing software involves several steps or tasks, some of which are manual. When working in a team, it is usually required that these steps be documented, and, in an ideal case, automated. Preparing good documentation for a classic imperative administrative procedure and automating these steps can be non-trivial tasks, even if each of the steps is simple.

A **declarative** approach for administrative tasks is intended to solve such challenges. With a declarative approach, an administrator defines a target state for a system (application, server, or cluster). Typically, a domain-specific language (DSL) is used to describe the target state. An administrative tool, such as Kubernetes, takes this definition as an input and takes care of how to achieve the target state from the current observable state.

HANDS-ON: GETTING STARTED

Minikube is an ideal tool for getting started with Kubernetes on a single computer. It enables running of a single-node Kubernetes cluster in a virtual machine. It can be used on GNU/Linux or OS X and requires VirtualBox, KVM (for Linux), xhyve (OS X), or VMware Fusion (OS X) to be installed on your computer. Minikube creates a new virtual machine with GNU/Linux, installs and configures Docker and Kubernetes, and finally runs a Kubernetes cluster.

In the following instructions, Minikube is used to install a single-node Kubernetes cluster on a machine with 64 bit GNU/Linux (Debian or Ubuntu) and KVM. Refer to the Minikube documentation if you want to use an alternative configuration.

```
$ minikube start --vm-driver=kvm

Starting local Kubernetes cluster...

Kubernetes is available at https://192.16

Kubectl is now configured to use the cl
```

INSTALLATION

1. Install the kubectl command line tool locally:

```
$ curl -Lo kubectl \
http://storage.googleapis.com/kubernetes-release/\
release/v1.3.0/bin/linux/amd64/kubectl \
&& chmod +x kubectl \
&& sudo mv kubectl /usr/local/bin/
```

2. Next, install the KVM driver:

```
$ sudo curl -L \
 https://github.com/dhiltgen/docker-machine-kvm/\
 releases/download/v0.7.0/docker-machine-driver-kvm \
 -o /usr/local/bin/docker-machine-driver-kvm
 $ sudo chmod +x /usr/local/bin/docker-machine-driver-kvm
```

3. Install Minikube:

```
$ curl -Lo minikube \
https://storage.googleapis.com/minikube/\
releases/v0.6.0/minikube-linux-amd64 \
&& chmod +x minikube \
&& sudo mv minikube /usr/local/bin/
```

4. Start the Minikube cluster:

```
$ minikube start --vm-driver=kvm
Starting local Kubernetes cluster...
Kubernetes is available at https://192.168.42.213:8443.
Kubectl is now configured to use the cluster.
```

The Kubernetes cluster is up and running.
Let’s start with a simple deployment using an existing image:

```
$ kubectl run hello-minikube \
--image=gcr.io/google_containers/echoserver:1.4 \
--port=8080
deployment “hello-minikube” created

$ kubectl expose deployment hello-minikube --type=NodePort
service “hello-minikube” exposed
```

5. Check that the pod is up and running:

```
$ kubectl get pod
NAME            READY     STATUS    RESTARTS   AGE
hello-minikube-2433534028-ouxw8   1/1       Running    0       4m
```

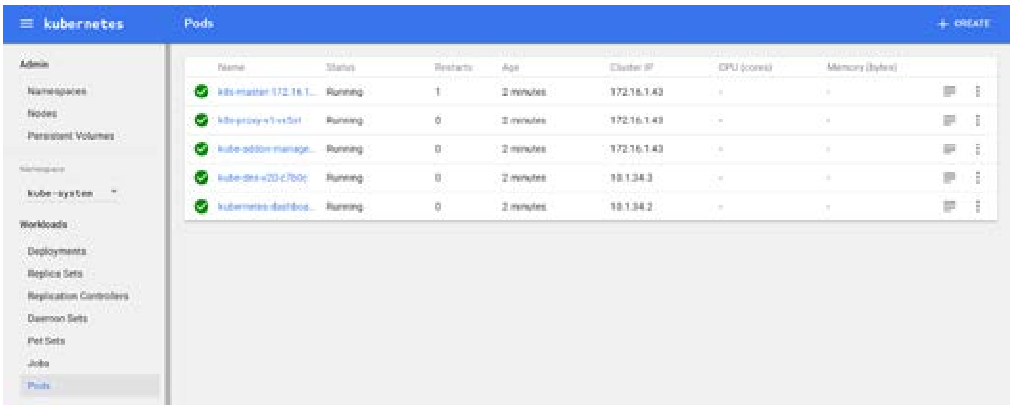
“Running” should appear in the STATUS field. If “ContainerCreating” appears instead, wait a few moments, then repeat the last command.

6. Check that the service works:

```
$ curl $(minikube service hello-minikube --url)
CLIENT VALUES:
client_address=172.17.0.1
command=GET
real path=/
query=nil
request_version=1.1
request_uri=http://192.168.42.213:8080/
SERVER VALUES:
server_version=nginx: 1.10.0 - lua: 10001
HEADERS RECEIVED:
accept=*/*
host=192.168.42.213:31759
user-agent=curl/7.35.0
BODY:
-no body in request-
```

7. Execute the following command to open the Kubernetes Dashboard in your web browser:

```
$ minikube dashboard
```



	Name	Status	Restarts	Age	Cluster IP	CPU (cores)	Memory (bytes)
✓	k8s-master-172.16.1...	Running	1	2 minutes	172.16.1.43	-	-
✓	k8s-proxy-v1-vx5rl	Running	0	2 minutes	172.16.1.43	-	-
✓	kube-addon-manage...	Running	0	2 minutes	172.16.1.43	-	-
✓	kube-dns-v20-c7b0c	Running	0	2 minutes	10.1.34.3	-	-
✓	kubernetes-dashboa...	Running	0	2 minutes	10.1.34.2	-	-

8. To stop the cluster (shut down the virtual machine and preserve its state), execute the following command:

```
$ minikube stop
Stopping local Kubernetes cluster...
Stopping “minikubeVM”...
```

9. To start the cluster again and restore it to the previous state, execute the following command:

```
$ minikube start --vm-driver=kvm
```

10. To delete the cluster (delete the virtual machine and its state), execute the following command:

```
$ minikube delete
```

Note: There are other open-source tools, such as kubeadm, that simplify installation of Kubernetes cluster in public clouds, on-premises virtual machines, and bare-metal servers. However, there are still many things that are out of scope. For example, you still need a reliable distributed block or file storage, you still need to think about HA, scalability, networking and security. Sometimes, it is simpler to use Kubernetes as a Service.

LOGGING

Basic logging in Kubernetes behaves much like logging in Docker. There is a kubectl logs command that will show you all the information written to stdout and stderr for a given container in a pod. If a pod has only one container, there is no need to specify it explicitly, however when a pod has several containers we need to add `-c container` to the end of the command. As with Docker, we can opt to follow logs, to reduce the number of recent lines with `--tail` and we can filter them by date. Unlike Docker, Kubernetes enables us to check the logs of a container that crashed using the `--previous` option.

The ability to keep the logs of a previous container is available as long as the pod it was run in remains available. When a pod is removed, so are its logs.

Log rotation is performed by Kubernetes. The default values are daily rotation or 10 MB to avoid log files taking up all the available disk space. Up to five rotations are kept for historical evidence. Remember that only the last rotation is displayed with kubectl logs; if you want to access an earlier one, you must do so manually.

Per our example with the Hello Minikube service, we can use `kubectl logs hello-minikube-2433534028-ouxw8` to view the access log containing our curl request.

All information so far concerns per-node log files. There are no cluster-level logging capabilities built into Kubernetes, but there are some common methods that can be implemented for this purpose.

DEDICATED AGENT RUNNING ON EVERY NODE

In this approach, a logging agent is run on every node, preferably through a DeamonSet replica. A popular choice in this space is fluentd. It can be configured with various backends among which are Google Cloud Platform and Elasticsearch.

Fluentd even goes as far as to provide a ready to use DaemonSet YAML on GitHub. All that needs to be done is edit its configuration to point at our logging backend.

DEDICATED CONTAINER INSIDE A POD

There are several use cases for this approach. The general idea is that a dedicated logging component in each pod either writes logs to its stdout and stderr or delivers the logs directly to a logging backend. With such a sidecar approach, we can aggregate all logs from different containers in a pod to a single stream that can be accessed with kubectl logs or we can split logs of one application into different logical streams. For example, Webservers' access.log and error.log can each be streamed by a dedicated container to its stdout, so we can check them separately with `kubectl logs $podname -c access-log and kubectl logs $podname -c error-log.`

DIRECT LOGGING FROM AN APPLICATION

If the application running in a pod can already communicate with a logging backend, it is possible to skip the Kubernetes logging options altogether. While direct logging may offer some performance advantages and provide slightly better security (all data stored in just one place), it also prevents us from using `kubectl logs`. In most cases, this approach is discouraged.

MONITORING

As Kubernetes containers are actually Linux processes, we can use our favourite tools to monitor cluster performance. Basic tools, such as top or kubectl top, will behave as expected. It's also possible to use solutions that are dedicated to Kubernetes. One such solution is Heapster. Heapster aggregates events and data from across the cluster. It runs as a pod in system namespace. Discovery and querying of resources is done automatically with data coming from a kubelet managing node.

Storage is configurable with InfluxDB and Google Cloud Monitoring being the most popular choices. When building a DIY cluster InfluxDB with Grafana for visualization is the preferred choice. Using it with Minikube requires two easy steps. First, enable the Heapster addon (`minikube addons enable heapster`) by opening the dashboard with Minikube addons. Then open Heapster.

By default, dashboards are available for Cluster and for Pods, one for monitoring running nodes and overall cluster utilization, the other for running pods. The information presented for pods includes CPU usage, memory usage, network usage, and filesystem usage. It is possible to edit existing graphs or add custom ones based on data exported by Heapster.

WORKING WITH MULTIPLE CLUSTERS

So far, we have used kubectl to connect to only one cluster created by Minikube. But kubectl can be configured to use multiple clusters and multiple contexts to connect to them. To check available contexts, we use kubectl config get-contexts.

We can confirm that only one context and only one cluster is defined by kubectl config view. It should look like this:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: $HOME/.minikube/ca.crt
    server: https://192.168.99.100:8443
  name: minikube
contexts:
- context:
    cluster: minikube
    user: minikube
  name: minikube
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate: $HOME/.minikube/apiserver.crt
    client-key: $HOME/.minikube/apiserver.key
```

The config file used by kubectl is stored at `~/.kube/config`. We can edit it with a text editor and add another cluster, context and user. When ready, `kubectl config get-contexts` should show our newly added context without marking it as current. This is the desired state:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: $HOME/.minikube/ca.crt
    server: https://192.168.99.100:8443
  name: minikube
- cluster:
    certificate-authority: $HOME/.minikube/ca.crt
    server: https://192.168.99.100:8443
  name: secondkube
contexts:
- context:
    cluster: minikube
    user: minikube
  name: minikube
- context:
    cluster: secondkube
    user: secondkube
  name: secondkube
current-context: secondkube
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate: $HOME/.minikube/apiserver.crt
    client-key: $HOME/.minikube/apiserver.key
- name: secondkube
  user:
    client-certificate: $HOME/.minikube/apiserver.crt
    client-key: $HOME/.minikube/apiserver.key
```

To switch context, we use `kubectl config use-context secondkube`. We can verify the switch was successful again with `kubectl config get-contexts`. The marker for current should have moved to the new context. All `kubectl` commands from now on will be executed in a selected context (which in our example is exactly the same as the first one).

HANDS-ON: DEPLOYING AN APPLICATION

In this example, we deploy the WordPress content management system with a MySQL backend. It is a classic two-tier application, where the first tier is the application server (WordPress) that uses the second tier for data persistence (MySQL).

STEP 1. CREATE A KUBERNETES SECRET

As discussed above, Kubernetes secrets allow users to pass sensitive information, such as passwords, database credentials, to containers. In the first step, we need to define a Kubernetes secret that contains a database name, user, and password. It should also contain the root password for MySQL.

Before creating a secret, we need to encode such information in Base64 format. Let's assume we want to use the following values in our application:

- "app-db" as a database name
- "app-user" as a database user name
- "app-pass" as a database password
- "app-rootpass" as a database root password

Note that we need to provide a database root password to allow WordPress to create the required database. To encode these values to Base64 format, we can use the standard base64 utility that is available in almost all Linux distributions:

```
$ echo -n "app-db" | base64
YXBwLWRi

$ echo -n "app-user" | base64
YXBwLXVzZXI=

$ echo -n "app-pass" | base64
YXBwLXBhc3M=

$ echo -n "app-rootpass" | base64
YXBwLXJvb3RwYXNz
```

We use the "-n" option to make sure that the new line symbol ("\n") is not included in the encoded value.

To define a new Kubernetes secret, create a new file, app-secret.yaml, with the following content:

```
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
type: Opaque
data:
  dbname: YXBwLWRi
  dbuser: YXBwLXVzZXI=
  dbpass: YXBwLXBhc3M=
  dbrootpass: YXBwLXJvb3RwYXNz
```

Note:Kubernetes allows its building blocks to be defined using JSON and YAML formats. These formats are quite popular now as a more lightweight alternative to XML. The idea behind XML, JSON, and YAML is to provide a universal text notation to serialize data in both machine- and human-readable form. In this example, we will use YAML.

In the `app-secret.yaml file,` we specified the required Kubernetes API version and the data type to let Kubernetes know that we are defining a secret. In addition, the file defines four keys (`dbname, dbuser, dbpass, dbrootpass`) with the corresponding values we encoded above. Now we can create our Kubernetes secret using its definition in the `app-secret.yaml` file:

```
$ kubectl create -f app-secret.yaml
secret "app-secret" created
```

Let's verify the secret creation:

```
$ kubectl get secrets
NAME           TYPE          DATA    AGE
app-secrets    Opaque         4             1m
```

STEP 2. CREATE A PERSISTENT VOLUME

Next, we will create a Kubernetes persistent volume to provide the underlying storage for our MySQL database. To define a new persistent volume, create a new file, app-pv.yam, with the following content:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: app-pv
  labels:
    vol: mysql
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /data/app
```

In this file, we use a HostPath volume, which will just pass a directory from the host into a container for consumption. To specify that the persistent volume will be used for our application, we added a label (`vol: mysql`) that can be used later in a selector.

Before creating the actual persistent volume, verify that the directory /data/app exists:

```
$ sudo mkdir -p /data/app
```

Now we can create our persistent volume:

```
$ kubectl create -f app-pv.yaml
persistentvolume "app-pv" created
```

Let's verify that the persistent volume is available:

```
$ kubectl describe pv/app-pv
Name:            app-pv
Labels:          vol=mysql
Status:          Available
Claim:
Reclaim Policy: Retain
Access Modes:    RWO
Capacity:        1Gi
Message:
Source:
  Type:          HostPath (bare host directory volume)
  Path:          /data/app
No events.
```

STEP 3. CLAIM A PERSISTENT VOLUME

For MySQL, we need to claim our previously created persistent volume. Create a new file, app-pvc.yaml, with the following content:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: app-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  selector:
    matchLabels:
      vol: mysql
```

The label selector "matchLabels" is used to make the association to the persistent volume that we created early. To create the persistent volume claim using its definition execute the following:

```
$ kubectl create -f app-pvc.yaml
persistentvolumeclaim "app-pvc" created
```

STEP 4. DEPLOY MYSQL

Now we will create a new deployment for MySQL using the existing Docker image. Create a new file, `mysql-deployment.yaml`, with the following content:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: mysql-deployment
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: "mysql:5.6"
        ports:
        - containerPort: 3306
        volumeMounts:
        - mountPath: "/var/lib/mysql"
          name: mysql-pd
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: app-secret
              key: dbrootpass
        - name: MYSQL_USER
          valueFrom:
            secretKeyRef:
              name: app-secret
              key: dbuser
        - name: MYSQL_PASSWORD
          valueFrom:
            secretKeyRef:
              name: app-secret
              key: dbpass
        - name: MYSQL_DATABASE
          valueFrom:
            secretKeyRef:
              name: app-secret
              key: dbname
      volumes:
        - name: mysql-pd
          persistentVolumeClaim:
            claimName: app-pvc
```

There is quite a lot of stuff happening in the above YAML. Let's break it down:

- We are specifying a standard pod's information, such as container name (mysql), image to use (mysql:5.6), and exposed port to access (3306).
- We are specifying the number of replicas (1) and attaching a label (`app: mysql`).
- We are then mounting a volume to the `"/var/lib/mysql"` directory in the container, where the volume to mount is named `"mysql-pd"` and is declared at the bottom of this document.
- We are also declaring environment variables to initialize. The MySQL image we are using that is available on Docker Hub supports environment variable injection. The four environment variables we are initializing are defined and used within the Docker image itself. The environment variables set all reference different keys we defined in our secret earlier. When this container starts up, we will automatically have MySQL configured with the desired root user password. We will also have the database for WordPress created with appropriate access granted for our WordPress user.

To create the deployment using its definition, execute the following:

```
$ kubectl create -f mysql-deployment.yaml
deployment "mysql-deployment" created
```

Let's verify that everything was created successfully:

```
$ kubectl get pv
NAME    CAPACITY  ACCESSMODES   STATUS  CLAIM
REASON   AGE
app-pv  1Gi       RWO           Bound   default/app-pvc
10m

$ kubectl get pvc
NAME     STATUS   VOLUME  CAPACITY  ACCESSMODES  AGE
app-pvc  Bound    app-pv  0                      5m

$ kubectl get deployments
NAME              DESIRED   CURRENT   UP-TO-DATE  AVAILABLE
AGE
mysql-deployment  1         1         1           1
2m
```

STEP 5. CREATE A SERVICE FOR MYSQL

As we know, pods are ephemeral. They come and go, with each newly created pod receiving a new and different IP address. To connect to a database, the WordPress application should know its IP address. If the database container is ephemeral, then how should our application keep track of the database server's IP addresses? We need an IP address that is decoupled from that pod and that never changes, and this is exactly what Kubernetes Services offer. To define a service for MySQL, create a new file, `mysql-service.yaml`, with the following content:

```
apiVersion: v1
kind: Service
metadata:
  name: mysql-service

spec:  ports:
    - port: 3306
      protocol: TCP
      targetPort: 3306
  selector:
    app: mysql
```

To create the actual service execute, the following command:

```
$ kubectl create -f mysql-service.yaml
service "mysql-service" created
```

Let's verify that the service is created and correctly mapped:

```
$ kubectl describe svc/mysql-service
Name:                mysql-service
Namespace:           default
Labels:              <none>
Selector:            app=mysql
Type:                ClusterIP
IP:                  ...
Port:                <unset> 3306/TCP
Endpoints:           172.17.0.3:3306
Session Affinity:   None
No events.

$ kubectl get pods -o wide
NAME                 READY    STATUS     RESTARTS   AGE    IP
mysql-deployment-... 1/1      RUNNING    0          30m
172.17.0.3
```

From the output above, we can verify the service was correctly mapped to the pod for our MySQL deployment in that the Endpoints IP address for the service aligns with the IP address for the MySQL Pod.

STEP 6. DEPLOY WORDPRESS

To define a deployment for WordPress, create a new file, wordpress-deployment.yaml, with the following content:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: wordpress-deployment
spec:
  replicas: 2
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: wordpress
    spec:
      containers:
      - name: "wordpress"
        image: "wordpress:4.5-apache"
        ports:
        - containerPort: 80
        env:
        - name: WORDPRESS_DB_HOST
          value: mysql-service
        - name: WORDPRESS_DB_USER
          valueFrom:
            secretKeyRef:
              name: app-secrets
              key: dbuser
        - name: WORDPRESS_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: app-secret
              key: dbpass
        - name: WORDPRESS_DB_NAME
          valueFrom:
            secretKeyRef:
              name: app-secret
              key: dbname
```

In this file, we are specifying standard pod information such as container name (wordpress), image to use (wordpress: 4.5-apache), exposed port to access (80), and number of replicas (2). We are also attaching a label (app: wordpress) to the pod's replicas. One of the key things we accomplish in the `YAML` above is the initialization of the environment variable `"WORDPRESS_DB_HOST"` to a value of "mysql-service". This is how we are tell the WordPress application to access its database through the Kubernetes service we created in the previous step.

To create the actual deployment, execute the following command:

```
$ kubectl create -f wordpress-deployment.yaml
deployment "wordpress-deployment" created
```

Verify the deployment:

```
$ kubectl get deployments
NAME                 DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
mysql-deployment     1       1       1          1         45m
wordpress-deployment 2       2       2          2         5m
```

Get a list of created pods:

```
$ kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
mysql-deployment-...     1/1     Running   0          45m
wordpress-deployment-... 1/1     Running   0          6m
wordpress-deployment-... 1/1     Running   0          6m
```

Make note of the name of one of the WordPress pods from the output above. Execute an interactive shell within that pod:

```
$ kubectl exec -it wordpress-deployment-... bash
```

Let's check that the MySQL service can be resolved within the pod using the service's name:

```
root@wordpress# getent hosts mysql-service
10.0.0.248      mysql-service.default.svc.cluster.local
```

The above output verifies that mysql-service can be resolved through DNS to the ClusterIP address that was assigned to the MySQL service (your IP address may be different).

Now let's verify that WordPress is properly configured:

```
root@wordpress# grep -i db /var/www/html/wp-config.php
define('DB_NAME', 'app-db');
define('DB_USER', 'app-user');
define('DB_PASSWORD', 'app-pass');
define('DB_HOST', 'mysql-service');
...
```

The WordPress pod has configured itself using the environment environments "injected" into container using the values from the Kubernetes secret we defined earlier.

STEP 7. CREATE A SERVICE FOR WORDPRESS

The final step is to expose the WordPress application to external users. For this, we again need a service. In this step, we expose a port on the node running our application, and forward it to port 80 of our container. This allows us to access the application, but it probably is not the approach one would take in production, especially if Kubernetes is hosted by a service provider. Kubernetes can integrate with load balancing services offered by platforms such as GCE and AWS. If you are using either of those, then that would be an approach to take for using the load balancing functionality offered by those platforms.

To define a service for the WordPress application, create a new file, wordpress-service.yaml, with the following content:

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress-service
  labels:
    app: wordpress
spec:
  type: NodePort
  ports:
    - port: 80
      nodePort: 30080
  selector:
    app: wordpress
```

To create the actual service using the definition from the `wordpress-service.yaml` file, execute the following command:

```
$ kubectl create -f wordpress-service.yaml
service "wordpress-service" created
```

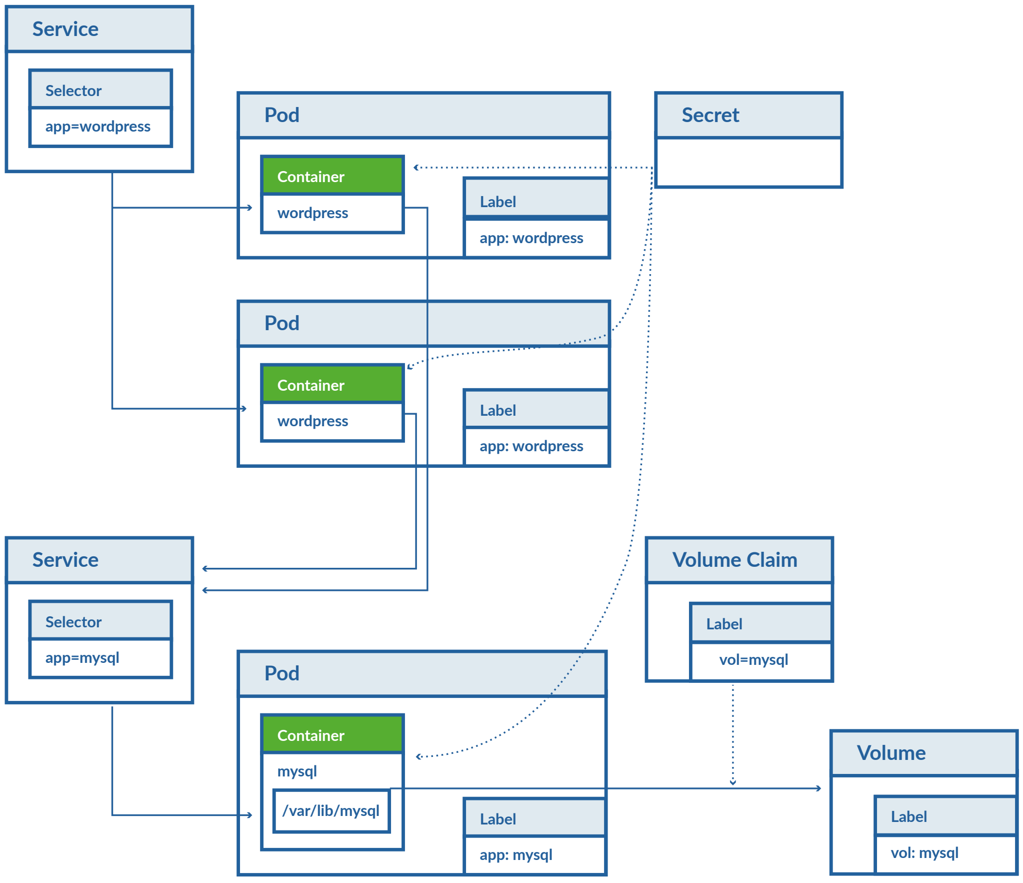
Verify its status:

```
$ kubectl describe svc/wordpress-service
Name:                 wordpress-service
Namespace:            default
Labels:               app=wordpress
Selector:             app=wordpress
Type:                 NodePort
IP:                   ...
Port:                 <unset> 80/TCP
NodePort:             <unset> 30080/TCP
Endpoints:            ...:80,...:80,...:80
Session Affinity:   None
No events.
```

STEP 8. TEST THE WORDPRESS APPLICATION

Open your browser and navigate to http://<nodeIP>:30080, where <nodeIP> is the address of your Kubernetes cluster node. You can follow the installation wizard to get WordPress up and running through the browser. Congratulations!

The following diagram shows all of the Kubernetes building blocks we defined and created for our application:



In production, we may want to update the WordPress deployment increasing the number of replicas to handle a high load for the application. We can easily do it manually, but the preferred way in Kubernetes is to use auto-scaling feature. For the MySQL deployment in our example, there is no simple way to increase the number of MySQL pods, because we cannot share the same persistent volume between several MySQL processes: MySQL does not support that. To scale the database and make it highly available we can use, for example, a Galera cluster in the multi-master node. In this case, each Galera pod will use its own persistent volume and Galera cluster will replicate its data between pods using its own protocol.

DIY CLUSTER CONSIDERATIONS

What if you need a production ready Kubernetes cluster, but for some reason you cannot use the existing cloud offerings, such as Google Container Engine? Kubernetes can be installed on a variety platforms, including on-premises VMs, VMs on a cloud provider, and bare-metal servers. There are several tools that allow installing production ready Kubernetes cluster on a variety of targets:

- kargo
- kube-deploy
- kube-admin

When deploying a Kubernetes cluster in production, one should take care of several things to ensure the best possible outcome. High Availability of master nodes helps minimize downtime and data loss as well as eliminates single point of failure. Networking should be both robust and scalable to handle growing needs (e.g., The number of nodes in a cluster to handle more replicas). Finally, some users may want to take advantage of multi-site support to uniformly handle geographically dispersed data centers.

HIGH AVAILABILITY

Possible cases of failure in Kubernetes clusters usually point to pods, nodes, and master nodes. Pod failures can be handled by built-in Kubernetes features, so the main concern here is to provide persistent storage if needed. Node failures can be handled by master nodes and require use of services outside of Kubernetes. For example, kubelet talks to an external load-balancer rather than directly to clients; if the entire node fails, traffic can be load balanced to the node with corresponding pods. Finally, the master controller can fail, or one of its services can die. We need to replicate the master controller and its components for a Highly Available environment. Fortunately, multiple master nodes are also accounted for in Kubernetes.

Furthermore, when it comes to monitoring the deployment, It is advisable that process watchers be implemented to "watch" the services that exist on the master node. For example, the API service can be monitored by a kubelet. It can be configured with less aggressive security settings to monitor non-Kubernetes components such as privileged containers. On a different level is the issue of what happens if a kubelet dies. Monitoring processes can be deployed to ensure that the kubelet can be restarted. Finally, redundant storage service can be achieved with clustered etcd.

SECURITY

First of all, direct access to cluster nodes (either physical or through SSH) should be restricted. `kubectl exec` allows access to containers—this should be enough. Use Security Contexts to segregate privileges. Defining quotas for resources helps prevent DoS attacks. Selectively grant users permissions according to their business needs. Finally, consider separating network traffic that is not related (e.g., The load balancer only needs to see a front-end service, while the back-end service has no need to contact the load balancer).

SCALE

Kubernetes allows for adding and removing nodes dynamically. Each new node has to be configured appropriately and pointed at the master node. The main processes of interest are `kubelet` and `kube-proxy`. For larger scale clusters, a means of automation is preferred, such as Ansible or Salt. If the cluster is running on one of supported cloud providers, there is also an option to try the Cluster Autoscaler.

```
$ minikube start --vm-driver=kvm
Starting local Kubernetes cluster...
Kubernetes is available at https://192.168.42.213:8443.
Kubectl is now configured to use the cluster.
```

SUMMARY

Kubernetes is an open-source project that is well supported by community. It allows application development to be completely infrastructure-agnostic and avoids vendor lock-in.

Installing, maintaining, and manually monitoring a production-ready Kubernetes cluster on premises is a difficult task. For the installation, high availability and networking models should be chosen and properly implemented. A tool to manage nodes in the cluster or to monitor the cluster's and nodes' health is also handy.

Keeping previous suggestions in mind, you should be able to roll out your own HA cluster for Kubernetes. It takes some work, both in terms of planning and actual execution. Planning carefully should save you much time later on when you start to scale your services. However if you want to take advantage of Kubernetes features but are not keen on maintaining your own cluster, Stratoscale's Kubernetes-as-a-Service may suit your needs perfectly.

ABOUT STRATOSCALE SYMPHONY

Stratoscale is the cloud infrastructure company, providing comprehensive cloud infrastructure software solutions for service providers, enterprise IT and development teams. The company's comprehensive cloud data center software, Stratoscale Symphony, can be deployed in minutes on commodity x86 servers, providing an Amazon Web Services (AWS) experience with the ability to augment aging VMware infrastructure. Stratoscale was named a "Cool Vendor in Servers and Virtualization" by Gartner and is backed by over $70M from leading investors including: Battery Ventures, Bessemer Venture Partners, Cisco, Intel, Qualcomm Ventures, SanDisk and Leslie Ventures.

SIMPLE CREATION

Admins can create private clusters via intuitive GUI or API.

ONGOING MONITORING

Easily monitor cluster health and usage.

MINIMAL LEARNING CURVE

Leverage Kubernetes GUI and maintain existing practices.

MULTI-TENANCY

Kubernetes is transformed to offer a true multi-tenant service.

USING KUBECTL CLI

FOR SINGLE-CLICK MANAGEMENT OF KUBERNETES CLUSTERS

Symphony's fully managed Kubernetes-as-a-Service removes the operational barriers of adopting a container-based strategy. Admins can leverage KubeCtl CLI or Symphony's intuitive GUI to create Kubernetes clusters and easily monitor cluster health and usage.

Symphony keeps the clusters up and running according to defined sizes and provides network access to cluster endpoints and node maintenance.

SYMPHONY: YOUR ON-PREM AWS-REGION

This simple example demonstrates how you can leverage the agility and simplicity of cloud methodologies within your on-prem environment. Symphony's service offers easy and single-click creation, monitoring and management of Kubernetes clusters to ensure a smooth transition towards a containers-driven application strategy.

STEP 1.

ASSIGN STORAGE AND NETWORK FOR THE CLUSTER

Use KubeCtl CLI commands to allocate a storage pool, network (VPC) and floating IP to the new Kubernetes cluster.

STEP 2.

CREATE A NEW KUBERNETES CLUSTER

Use KubeCtl CLI commands to create the new cluster based on the required number of nodes and the parameters for each node, including number of CPUs and memory. The assigned floating IP will be used as the cluster's end-point.

STEP 3.

MONITOR KUBERNETES CLUSTERS

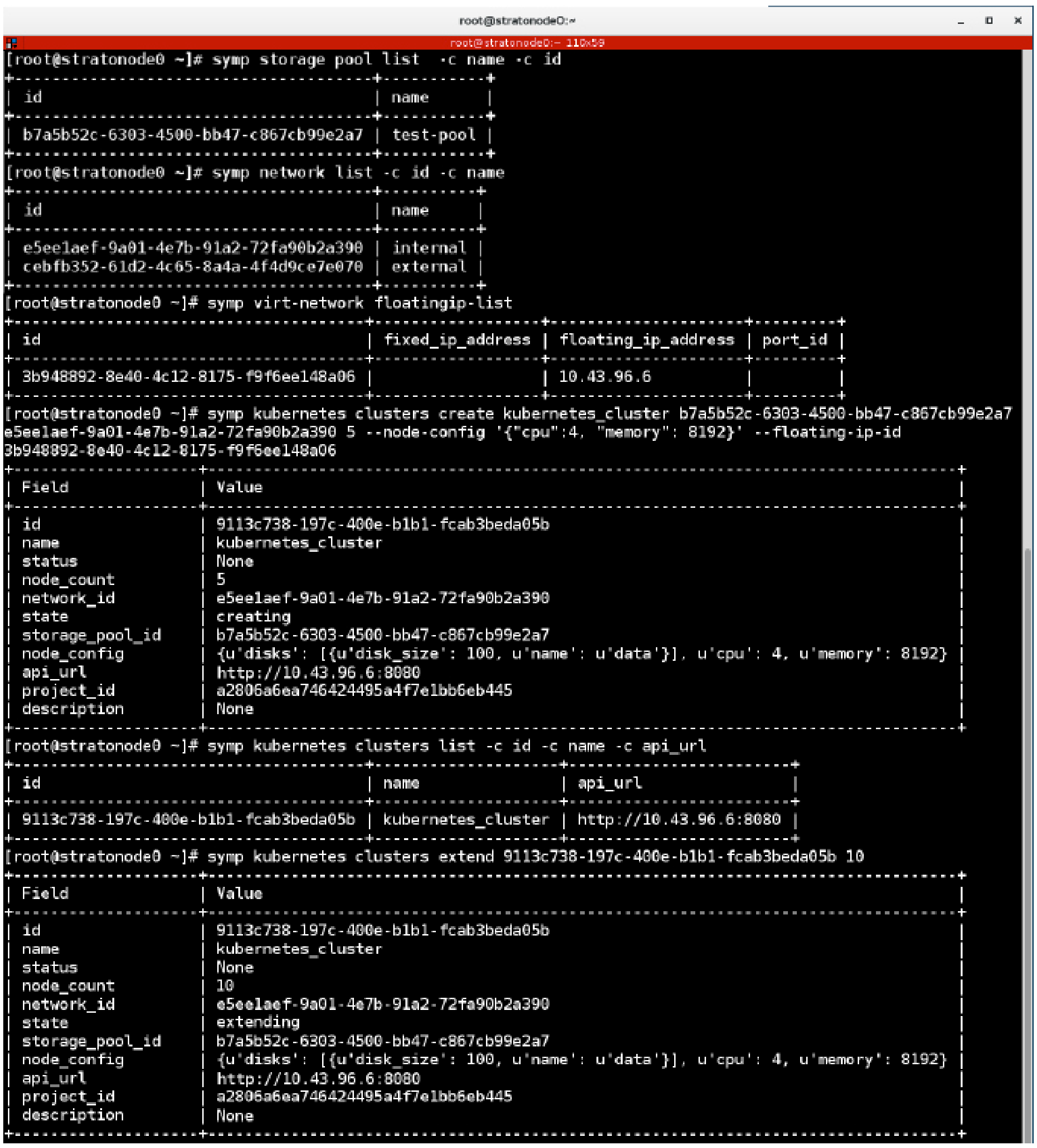
Use KubeCtl CLI commands to continuously monitor the clusters and check their status.

STEP 4.

MANAGE AND EXTEND KUBERNETES CLUSTERS

As needs and requirements evolve, use KubeCtl CLI commands to easily expand the cluster and increase the number of Kubernetes nodes.

FOR SINGLE-CLICK MANAGEMENT OF KUBERNETES CLUSTERS



```
[root@stratonode0 ~]# symp storage pool list  -c name -c id
+--------------------------------------+-----------+
| id                                   | name      |
+--------------------------------------+-----------+
| b7a5b52c-6303-4500-bb47-c867cb99e2a7 | test-pool |
+--------------------------------------+-----------+
[root@stratonode0 ~]# symp network list -c id -c name
+--------------------------------------+----------+
| id                                   | name     |
+--------------------------------------+----------+
| e5ee1aef-9a01-4e7b-91a2-72fa90b2a390 | internal |
| cebfb352-61d2-4c65-8a4a-4f4d9ce7e070 | external |
+--------------------------------------+----------+
[root@stratonode0 ~]# symp virt-network floatingip-list
+--------------------------------------+------------------+---------------------+---------+
| id                                   | fixed_ip_address | floating_ip_address | port_id |
+--------------------------------------+------------------+---------------------+---------+
| 3b948892-8e40-4c12-8175-f9f6ee148a06 |                  | 10.43.96.6          |         |
+--------------------------------------+------------------+---------------------+---------+
[root@stratonode0 ~]# symp kubernetes clusters create kubernetes_cluster b7a5b52c-6303-4500-bb47-c867cb99e2a7
e5ee1aef-9a01-4e7b-91a2-72fa90b2a390 5 --node-config '{"cpu":4, "memory": 8192}' --floating-ip-id
3b948892-8e40-4c12-8175-f9f6ee148a06
+---------------------+---------------------------------------------------------------------------------+
| Field               | Value                                                                           |
+---------------------+---------------------------------------------------------------------------------+
| id                  | 9113c738-197c-400e-b1b1-fcab3beda05b                                            |
| name                | kubernetes_cluster                                                              |
| status              | None                                                                            |
| node_count          | 5                                                                               |
| network_id          | e5ee1aef-9a01-4e7b-91a2-72fa90b2a390                                            |
| state               | creating                                                                        |
| storage_pool_id     | b7a5b52c-6303-4500-bb47-c867cb99e2a7                                            |
| node_config         | {u'disks': [{u'disk_size': 100, u'name': u'data'}], u'cpu': 4, u'memory': 8192} |
| api_url             | http://10.43.96.6:8080                                                          |
| project_id          | a2806a6ea746424495a4f7e1bb6eb445                                                |
| description         | None                                                                            |
+---------------------+---------------------------------------------------------------------------------+
[root@stratonode0 ~]# symp kubernetes clusters list -c id -c name -c api_url
+--------------------------------------+--------------------+------------------------+
| id                                   | name               | api_url                |
+--------------------------------------+--------------------+------------------------+
| 9113c738-197c-400e-b1b1-fcab3beda05b | kubernetes_cluster | http://10.43.96.6:8080 |
+--------------------------------------+--------------------+------------------------+
[root@stratonode0 ~]# symp kubernetes clusters extend 9113c738-197c-400e-b1b1-fcab3beda05b 10
+---------------------+---------------------------------------------------------------------------------+
| Field               | Value                                                                           |
+---------------------+---------------------------------------------------------------------------------+
| id                  | 9113c738-197c-400e-b1b1-fcab3beda05b                                            |
| name                | kubernetes_cluster                                                              |
| status              | None                                                                            |
| node_count          | 10                                                                              |
| network_id          | e5ee1aef-9a01-4e7b-91a2-72fa90b2a390                                            |
| state               | extending                                                                       |
| storage_pool_id     | b7a5b52c-6303-4500-bb47-c867cb99e2a7                                            |
| node_config         | {u'disks': [{u'disk_size': 100, u'name': u'data'}], u'cpu': 4, u'memory': 8192} |
| api_url             | http://10.43.96.6:8080                                                          |
| project_id          | a2806a6ea746424495a4f7e1bb6eb445                                                |
| description         | None                                                                            |
+---------------------+---------------------------------------------------------------------------------+
```

For more information, visit:

http://www.stratoscale.com

US Phone: +1 877 420-3244 | Email: sales@stratoscale.com