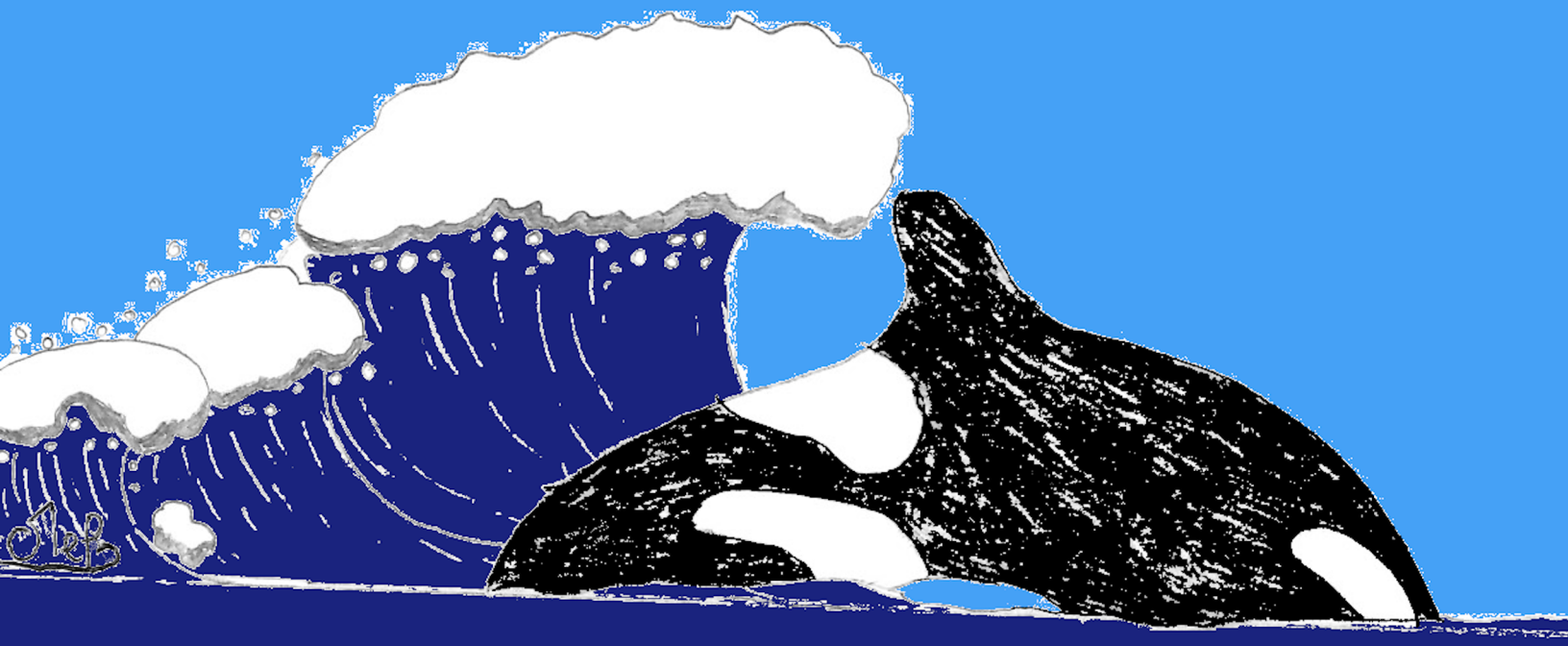

Kubernetes Overview

Prepare CKA & CKAD Certifications

Philippe Martin

Kubernetes Overview

Prepare CKA & CKAD Certifications

Philippe MARTIN

This book is for sale at http://leanpub.com/learning-kubernetes

This version was published on 2020-02-21

Contents

Creating a cluster with kubeadm

In this part, you will deploy a Kubernetes cluster on Virtual machines (VMs) in Google Cloud.

Provisioning compute resources

You will install a single control plane cluster. For this, you will need one virtual machine for the controller, and several (here two) virtual machines for the workers.

A full network connectivity among all machines in the cluster is necessary. For this, you will create a Virtual Private Cloud (VPC) that will host the cluster, and define a subnet to get addresses for the hosts.

From the Google Cloud Console, create a new project `my-project`, then login from the terminal and set the current region, zone and project:

```
$ gcloud auth login
[...]
$ gcloud config set compute/region us-west1
Updated property [compute/region].
$ gcloud config set compute/zone us-west1-c
Updated property [compute/zone].
$ gcloud config set project my-project
Updated property [core/project].
```

Create a dedicated Virtual Private Cloud (VPC):

```
$ gcloud compute networks create kubernetes-cluster --subnet-mode custom
Created [https://www.googleapis.com/compute/v1/projects/my-project/global/networks/k\
ubernetes-cluster].
```

Create a subnet in the `kubernetes-cluster` VPC:

```
$ gcloud compute networks subnets create kubernetes \
  --network kubernetes-cluster \
  --range 10.240.0.0/24
Created [https://www.googleapis.com/compute/v1/projects/my-project/regions/us-west1/\
subnetworks/kubernetes].
```

Create firewall rules for internal communications:

```
$ gcloud compute firewall-rules create \
  kubernetes-cluster-allow-internal \
  --allow tcp,udp,icmp \
  --network kubernetes-cluster \
  --source-ranges 10.240.0.0/24,10.244.0.0/16
Created [https://www.googleapis.com/compute/v1/projects/my-project/global/firewalls/\
kubernetes-cluster-allow-internal].
```

Create firewall rules for external communications:

```
$ gcloud compute firewall-rules create \
  kubernetes-cluster-allow-external \
  --allow tcp:22,tcp:6443,icmp \
  --network kubernetes-cluster \
  --source-ranges 0.0.0.0/0
Created [https://www.googleapis.com/compute/v1/projects/my-project/global/firewalls/\
kubernetes-cluster-allow-external].
```

Reserve a public IP address for the controller:

```
$ gcloud compute addresses create kubernetes-controller \
  --region $(gcloud config get-value compute/region)
Created [https://www.googleapis.com/compute/v1/projects/my-prpject/regions/us-west1/\
addresses/kubernetes-controller].
$ PUBLIC_IP=$(gcloud compute addresses describe kubernetes-controller \
    --region $(gcloud config get-value compute/region) \
    --format 'value(address)')
```

Create a VM for the controller:

```
$ gcloud compute instances create controller \
    --async \
    --boot-disk-size 200GB \
    --can-ip-forward \
    --image-family ubuntu-1804-lts \
    --image-project ubuntu-os-cloud \
    --machine-type n1-standard-1 \
    --private-network-ip 10.240.0.10 \
    --scopes compute-rw,storage-ro,service-management,service-control,logging-write,\
monitoring \
    --subnet kubernetes \
    --address $PUBLIC_IP
Instance creation in progress for [controller]: [...]
```

Create VMs for the workers:

```
$ for i in 0 1; do \
  gcloud compute instances create worker-${i} \
    --async \
    --boot-disk-size 200GB \
    --can-ip-forward \
    --image-family ubuntu-1804-lts \
    --image-project ubuntu-os-cloud \
    --machine-type n1-standard-1 \
    --private-network-ip 10.240.0.2${i} \
    --scopes compute-rw,storage-ro,service-management,service-control,logging-write,\
monitoring \
    --subnet kubernetes; \
done
Instance creation in progress for [worker-0]: [...]
Instance creation in progress for [worker-1]: [...]
```

Install Docker on the hosts

> Repeat these steps for the controller and each worker.

Connect to the host (here the controller):

```
$ gcloud compute ssh controller
```

Install Docker service:

```
# Install packages to allow apt to use a repository over HTTPS
$ sudo apt-get update && sudo apt-get install -y \
  apt-transport-https ca-certificates curl software-properties-common

# Add Docker's official GPG key
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

# Add Docker apt repository
$ sudo add-apt-repository \
  "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) \
  stable"

# Install Docker CE
$ sudo apt-get update && sudo apt-get install -y \
  containerd.io=1.2.10-3 \
  docker-ce=5:19.03.4~3-0~ubuntu-$(lsb_release -cs) \
  docker-ce-cli=5:19.03.4~3-0~ubuntu-$(lsb_release -cs)

$ sudo apt-mark hold containerd.io docker-ce docker-ce-cli

# Setup daemon
$ cat <<EOF | sudo tee /etc/docker/daemon.json
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2"
}
EOF

$ sudo mkdir -p /etc/systemd/system/docker.service.d

# Restart docker
$ sudo systemctl daemon-reload
$ sudo systemctl restart docker
$ sudo systemctl enable docker
```

Install kubeadm, kubelet and kubectl on the hosts

> Repeat these steps for the controller and each worker.

Connect to the host (here the controller):

```
$ gcloud compute ssh controller
```

Install `kubelet`, `kubeadm` and `kubectl`:

```
# Add GPG key
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -

# Add Kubernetes apt repository
$ cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF

# Get Kubernetes apt repository data
$ sudo apt-get update

# List available versions of kubeadm
$ apt-cache madison kubeadm

# Install selected version (here 1.15.7-00)
$ sudo apt-get install -y kubelet=1.15.7-00 kubeadm=1.15.7-00 kubectl=1.15.7-00
$ sudo apt-mark hold kubelet kubeadm kubectl
```

Initialize the control plane node

> Run these steps on the controller only.

Initilize the cluster (that should take several minutes):

```
$ gcloud config set compute/zone us-west1-c # or your selected zone
Updated property [compute/zone].
$ KUBERNETES_PUBLIC_ADDRESS=$(gcloud compute instances describe controller \
  --zone $(gcloud config get-value compute/zone) \
  --format='get(networkInterfaces[0].accessConfigs[0].natIP)')
$ sudo kubeadm init \
  --pod-network-cidr=10.244.0.0/16 \
  --ignore-preflight-errors=NumCPU \
  --apiserver-cert-extra-sans=$KUBERNETES_PUBLIC_ADDRESS
```

At the end of the initilization, a message gives you a command to join the workers to the cluster (a command starting with `kubeadm join`). Please copy this command for a later use.

Save the *kubeconfig* file generated by the installation in your home directory. It will give you an *admin* access to the cluster:

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
$ kubectl get nodes
NAME         STATUS     ROLES    AGE     VERSION
controller   NotReady   master   1m14s   v1.15.7
```

Install the `calico` Pod network add-on:

```
$ kubectl apply -f https://docs.projectcalico.org/v3.8/manifests/canal.yaml
```

Wait the end of the installation. All pods should have a *Running* status:

```
$ kubectl get pods -A
```

When all the pods are Running, the master node should be *Ready*:

```
$ kubectl get nodes
NAME         STATUS   ROLES    AGE     VERSION
controller   Ready    master   2m23s   v1.15.7
```

Join the workers

Repeat these steps for each worker.

Run the command you saved after running `kubeadm init` on the controller:

```
$ sudo kubeadm join 10.240.0.10:6443 --token <token> \
    --discovery-token-ca-cert-hash sha256:<hash>
```

If you didn't save the command, you have to get the token and hash. On the controller, run:

```
$ kubeadm token list
TOKEN                     TTL       EXPIRES
abcdef.ghijklmnopqrstuv   23h       2020-01-19T08:25:27Z
```

Tokens expire after 24 hours. If yours expired, you can create a new one:

```
$ kubeadm token create
bcdefg.hijklmnopqrstuvw
```

To obtain the `hash` value, you can run this command on the controller:

```
$ openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt | \
   openssl rsa -pubin -outform der 2>/dev/null | \
   openssl dgst -sha256 -hex | sed 's/^.* //'
[..hash value..]
```

Control Plane components

The Kubernetes Control Plane is composed of:

- the API Server `kube-apiserver`, the front end for the Kubernetes control plane,
- the Key-value store `etcd`, the backing store of all cluster data,
- the Scheduler `kube-scheduler`, which selects nodes for new pods to run on,
- the Controller Manager `kube-controller-manager`, which embeds all the controllers, including the *Node controller*, *Replication controller*, *Endpoints controller*, *Service account and token controllers*.

On each node, the components running are:

- `kubelet`, which makes sure the pods affected to its node are running correctly,
- `kube-proxy`, which maintains network rules on nodes to satisfy the `Service` demands.

Explore the Control Plane services

The `kubelet` service runs as a Unix service and its status and logs are accessible by using the traditional `systemd` command line tools:

```
$ systemctl status kubelet
[...]
$ journalctl -u kubelet
[...]
```

The other services are running in the Kubernetes cluster and visible in the `kube-system` namespace. You can get the status by using the `kubectl describe` command and the logs by using the `kubectl logs` command:

```
$ kubectl get pods -n kube-system
etcd-controller
kube-apiserver-controller
kube-controller-manager-controller
kube-proxy-123456
kube-proxy-789012
kube-proxy-abcdef
kube-scheduler-controller

$ kubectl describe pods -n kube-system etcd-controller
[...]
$ kubectl logs -n kube-system etcd-controller
[...]
```

You probably wonder what magic makes Kubernetes control plane run into Kubernetes itself? This is thanks to the Static Pods feature, with which it is possible to give definition of Pods directly to a `kubelet` service. You can find the manifests of the pods in the following directory of the controller:

```
$ gcloud compute ssh controller
Welcome to controller
$ ls /etc/kubernetes/manifests/
etcd.yaml
kube-apiserver.yaml
kube-controller-manager.yaml
kube-scheduler.yaml
```

Accessing the cluster

Install kubectl on your dev machine

Linux

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.15.7/bin/lin\
ux/amd64/kubectl
$ chmod +x ./kubectl
$ sudo mv ./kubectl /usr/local/bin/kubectl
# Test it is working correctly
$ kubectl version --client --short
Client Version: v1.15.7
```

macOS

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.15.7/bin/dar\
win/amd64/kubectl
$ chmod +x ./kubectl
$ sudo mv ./kubectl /usr/local/bin/kubectl
# Test it is working correctly
$ kubectl version --client --short
Client Version: v1.15.7
```

Windows

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.17.0/bin/win\
dows/amd64/kubectl.exe
# Move the binary into your PATH,
$ kubectl version --client --short
Client Version: v1.15.7
```

Access the cluster from the dev machine

Get the kubeconfig file for the new cluster:

```
$ gcloud compute scp controller:~/.kube/config kubeconfig
```

Update the IP address to access the cluster in the file:

```
$ KUBERNETES_PUBLIC_ADDRESS=$(gcloud compute instances describe controller \
  --zone $(gcloud config get-value compute/zone) \
  --format='get(networkInterfaces[0].accessConfigs[0].natIP)')
$ sed -i "s/10.240.0.10/$KUBERNETES_PUBLIC_ADDRESS/" kubeconfig
```

If you do not have a kubeconfig file yet, you can copy it in `$HOME/.kube/config`:

```
$ mv -i kubeconfig $HOME/.kube/config
```

If you already have a kubeconfig file, you can merge this new one with the existing one. First examine the `kubeconfig` file:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: <...>
    server: https://<ip>:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
    client-certificate-data: <...>
    client-key-data: <...>
```

You can see that this file defines a cluster named `kubernetes`, a user named `kubernetes-admin` and a context `kubernetes-admin@kubernetes`.

These names are very generic (if you create several clusters with kubeadm, all kubeconfig files will define these elements with the same names), we will first replace them with more specific ones:

```
$ sed -i 's/kubernetes/cka/' kubeconfig
```

Then, we can merge this file with the existing `$HOME/.kube/config` one:

```
$ KUBECONFIG=$HOME/.kube/config:kubeconfig \
  kubectl config view --merge --flatten > config \
  && mv config $HOME/.kube/config
```

Finally, you can switch the current context to `cka-admin@kubernetes`:

```
$ kubectl config use-context cka-admin@kubernetes
Switched to context "cka-admin@kubernetes".
```

Kubernetes Resources

Kubernetes works in a declarative way: you create **resources** with the help of the Kubernetes API, these objects are stored in the etcd store, and controllers work to ensure that what you declared in these objects is correctly deployed in your infrastructure.

Most of the resources are composed of three parts: the **Metadata**, the **Spec** and the **Status**.

The **Spec** is the specification **you** provide to the cluster. This is what the controllers will examine to know what to do.

The **Status** represents the current status of the resource in the infrastructure, as observed by **controllers**. This is what you will examine to know how the resource is deployed in the infrastructure.

The **Metadata** contains other information like the name of the resource, the namespace it belongs to, labels, annotations, etc.

Namespaces

Some resources (called namespaced resources) belong to a `namespace`. A namespace is a logical separation, and names of resources must be unique in a namespace.

RBAC Authorization uses the namespaces to define authorizations based on the namespace of resources.

You can create new namespaces with the command:

```
$ kubectl create namespace my-ns
namespace/my-ns created
```

Then, when running `kubectl` commands, you can specify the namespace on which the command operates with the flag `--namespace`, or `--all-namespaces` (`-A` for short) to operate in all namespaces.

You can also specify the default namespace you want to work on, with:

```
$ kubectl config set-context \
   --current --namespace=my-ns
Context "minikube" modified.
```

Labels and selectors

Any number of key-value pairs, named labels, can be attached to any resource. These labels are mainly used by components of Kubernetes and by tools to select resources by attributes (materialized by labels) instead of by name.

- Lots of `kubectl` commands accept a `--selector` (`-l` for short) flag which permits to select resources by labels:

```
$ kubectl get pods -l app=nginx -A
```

- Services, which route traffic to pods, select the pods with a label selector:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  selector:
    # distribute traffic to all pods with this label
    app: nginx
  ports:
  - port: 80
```

- A Deployment, which is responsible for maintaining alive a given number of pods, uses a label selector to find the pods it is responsible for (see Pod Controllers):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  selector:
    # the pods managed by the Deployment
    # are the ones with this label
    matchLabels:
      app: nginx
  template:
    metadata:
      # pods created by the Deployment
      # will have this label
      labels:
        app: nginx
    spec:
```

```
    containers:
    - image: nginx
      name: nginx
```

- You can select the attributes of the node on which you want to deploy a Pod by using label selectors (see Using label selectors to schedule Pods on specific nodes).

Annotations

Annotations are metadata attached to a resource, generally intended for tools and Kubernetes extensions, but not yet integrated in the Spec part. You can add annotations to a resource in an imperative way:

```
$ kubectl annotate deployments.apps nginx \
   mytool/mem-size=1G
deployment.apps/nginx annotated
```

Or in a declarative way:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    mytool/mem-size: 1G
  name: nginx
[...]
```

The workloads

The Pod is the master piece of the Kubernetes cluster architecture.

The fundamental goal of Kubernetes is to help you manage your containers. The Pod is the minimal piece deployable in a Kubernetes cluster, containing one or several containers.

From the `kubectl` command line, you can run a pod containing a container as simply as running this command:

```
$ kubectl run --generator=run-pod/v1 nginx --image=nginx
pod/nginx created
```

By adding `--dry-run -o yaml` to the command, you can see the YAML template you would have to write to create the same Pod:

```
$ kubectl run --generator=run-pod/v1 nginx --image=nginx --dry-run -o yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: nginx
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

Or, if you greatly simplify the template by keeping only the required fields:

```
-- simple.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
```

You can now start the Pod by using this template:

```
$ kubectl apply -f simple.yaml
pod/nginx created
```

The Pod created is ready … if you are not very fussy. Otherwise, the Pod offers a long list of fields to make it more production-ready. Here are all these fields.

Pod specs

Here is a classification of the Pod specification fields.

- **Containers** fields will define and parameterize more precisely each container of the Pod, whether it is a normal container (`containers`) or an init container (`initContainers`). The `imagePullSecrets` field will help to download containers images from private registries.
- **Volumes** field (`volumes`) will define a list of volumes that containers will be able to mount and share.
- **Scheduling** fields will help you define the most appropriate Node to deploy the Pod, by selecting nodes by labels (`nodeSelector`), directly specifying a node name (`nodeName`), using `affinity` and `tolerations`, by selecting a specific scheduler (`schedulerName`), and by requiring a specific runtime class (`runtimeClassName`). They will also be used to prioritize a Pod over other Pods (`priorityClassName`and `priority`).
- **Lifecycle** fields will help define if a Pod should restart after termination (`restartPolicy`) and fine-tune the periods after which processes running in the containers of a terminating pod are killed (`terminationGracePeriodSeconds`) or after which a running Pod will be stopped if not yet terminated (`activeDeadlineSeconds`). They also help define readiness of a pod (`readinessGates`).
- **Hostname and Name resolution** fields will help define the hostname (`hostname`) and part of the FQDN (`subdomain`) of the Pod, add hosts in the */etc/hosts* files of the containers (`hostAliases`), fine-tune the */etc/resolv.conf* files of the containers (`dnsConfig`) and define a policy for the DNS configuration (`dnsPolicy`).

- **Host namespaces** fields will help indicate if the Pod must use host namespaces for network (`hostNetwork`), PIDs (`hostPID`), IPC (`hostIPC`) and if containers will share the same (non-host) process namespace (`shareProcessNamespace`).
- **Service account** fields will be useful to give specific rights to a Pod, by affecting it a specific service account (`serviceAccountName`) or by disabling the automount of the default service account with `automountServiceAccountToken`.
- **Security Context** field (`securityContext`) helps define various security attributes and common container settings at the pod level.

Container Specs

An important part of the definition of a Pod is the definition of the containers it will contain.

We can separate container fields into two parts. The first part contains fields that are related to the container runtime (image, entrypoint, ports, environment variables and volumes), the second part containing fields that will be handled by the Kubernetes system.

The fields related to container runtime are:

- **Image** fields define the image of the container (`image`) and the policy to pull the image (`imagePullPolicy`).
- **Entrypoint** fields define the command (`command`) and arguments (`args`) of the entrypoint and its working directory (`workingDir`).
- **Ports** field (`ports`) defines the list of ports to expose from the container.
- **Environment variables** fields help define the environment variables that will be exported in the container, either directly (`env`) or by referencing ConfigMap or Secret values (`envFrom`).
- **Volumes** fields define the volumes to mount into the container, whether they are a filesystem volume (`volumeMounts`) or a raw block volume (`volumeDevices`).

The fields related to Kubernetes are:

- **Resources** field (`resources`) helps define the resource requirements and limits for a container.
- **Lifecycle** fields help define handlers on lifecycle events (`lifecycle`), parameterize the termination message (`terminationMessagePath` and `terminationMessagePolicy`), and define probes to check liveness (`livenessProbe`) and readiness (`readinessProbe`) of the container.
- **Security Context** field helps define various security attributes and common container settings at the container level.
- **Debugging** fields are very specialized fields, mostly for debugging purposes (`stdin`, `stdinOnce` and `tty`).

Pod Controllers

The pod, although being the master piece of the Kubernetes architecture, is rarely used alone. You will generally use a Controller to run a pod with some specific policies.

The different controllers handling pods are:

- `ReplicaSet`: ensures that a specified number of pod replicas are running at any given time.
- `Deployment`: enables declarative updates for Pods and ReplicaSets.
- `StatefulSet`: manages updates of Pods and ReplicaSets, taking care of stateful resources.
- `DaemonSet`: ensures that all or some nodes are running a copy of a Pod.
- `Job`: starts pods and ensures they complete.
- `CronJob`: creates a Job on a time-based schedule.

In Kubernetes, all controllers respect the principle of the **Reconcile Loop**: the controller perpetually **watches** for some objects of interest, to be able to detect if the actual state of the cluster (the objects running into the cluster) satisfies the specs of the different objects the controller is responsible for and to adapt the cluster consequently.

Let's examine more precisely how are working the extensively used ReplicaSet and Deployment controllers.

ReplicaSet controller

Fields for a ReplicaSet are:

- `replicas` indicates how many replicas of selected Pods you want.
- `selector` defines the Pods you want the ReplicaSet controller to manage.
- `template` is the template used to create new Pods when insufficient replicas are detected by the Controller.
- `minReadySeconds` indicates the number of seconds the controller should wait after a Pod starts without failing to consider the Pod is ready.

The ReplicaSet controller perpetually **watches** the Pods with the labels specified with `selector`. At any given time, if the number of actual running Pods with these labels:

- is greater than the requested `replicas`, some Pods will be terminated to satisfy the number of replicas. Note that the terminated Pods are not necessarily Pods that were created by the ReplicaSet controller,

- is lower than the requested `replicas`, new Pods will be created with the specified Pod `template` to satisfy the number of replicas. Note that to avoid the ReplicaSet controller to create Pods in a loop, the specified `template` must create a Pod selectable by the specified `selector` (this is the reason why you must set the same labels in the `selector.matchLabels` and `template.metadata.labels` fields).

Note that:

- the `selector` field of a ReplicaSet is immutable,
- changing the `template` of a ReplicaSet will not have an immediate effect. It will affect the Pods that will be created after this change,
- changing the `replicas` field will immediately trigger the creation or termination of Pods.

Deployment controller

Fields for a `Deployment` are:

- `replicas` indicates the number of replicas requested.
- `selector` defines the Pods you want the Deployment controller to manage.
- `template` is the template requested for the Pods.
- `minReadySeconds` indicates the number of seconds the controller should wait after a Pod starts without failing to consider the Pod is ready.
- `strategy` is the strategy to apply when changing the `replicas` of the previously and currently active `ReplicaSets`.
- `revisionHistoryLimit` is the number of `ReplicaSet` to keep for future use.
- `paused` indicates if the Deployment is active or not.
- `progressDeadlineSeconds`

The Deployment controller perpetually **watches** the ReplicaSets with the requested `selector`. Among these:

- if a ReplicaSet with the requested `template` exists, the controller will ensure that the number of replicas for this ReplicaSet equals the number of requested `replicas` (by using the requested `strategy`) and `minReadySeconds` equals the requested one.
- if no ReplicaSet exists with the requested `template`, the controller will create a new ReplicaSet with the requested `replicas`, `selector`, `template` and `minReadySeconds`.
- for ReplicaSets with a non matching `template`, the controller will ensure that the number of `replicas` is set to zero (by using the requested `strategy`).

Note that:

- the `selector` field of a Deployment is immutable,
- changing the `template` field of a Deployment will immediately:
  - either trigger the creation of a new ReplicaSet if no one exists with the requested `selector` and `template`
  - or update an existing one matching the requested `selector` and `template` with the requested `replicas` (using `strategy`),
  - set to zero the number of `replicas` of other ReplicaSets,
- changing the `replicas` or `minReadySeconds` field of a Deployment will immediately update the corresponding value of the corresponding ReplicaSet (the one with the requested `template`).

With this method, the Deployment controller manages a series of ReplicaSets, one for each revision of the Pod template. The active ReplicaSet is the one with a positive number of Replicas, the other revisions having a number of Replicas to zero.

This way, you can switch from one revision to another (for a rollback for example) by switching the Pod template from one revision to another.

Update and Rollback

Let's first deploy an image of the `nginx` server, with the help of a `Deployment`:

```
$ kubectl create deployment nginx --image=nginx:1.10
deployment.apps/nginx created
```

The command `kubectl rollout` provides several subcommands to work with deployments.

The subcommand `status` gives us the status of the deployment:

```
$ kubectl rollout status deployment nginx
deployment "nginx" successfully rolled out
```

The `history` subcommand gives us the history of revisions for the deployment. Here the deployment is at its first revision:

```
$ kubectl rollout history deployment nginx
deployment.apps/nginx
REVISION  CHANGE-CAUSE
1         <none>
```

We will now update the image of `nginx` to use the `1.11` revision. One way is to use the `kubectl set image` command:

```
$ kubectl set image deployment nginx nginx=nginx:1.11
deployment.extensions/nginx image updated
```

We can see with the `history` subcommand that the deployment is at its second revision:

```
$ kubectl rollout history deployment nginx
deployment.apps/nginx
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

The `change-cause` is empty by default. it can either contain the command used to make the rollout by using the `--record` flag:

```
$ kubectl set image deployment nginx nginx=nginx:1.12 --record
deployment.apps/nginx image updated
$ kubectl rollout history deployment nginx
deployment.apps/nginx
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
3         kubectl set image deployment nginx nginx=nginx:1.12 --record=true
```

Or by setting the `kubernetes.io/change-cause` annotation after the rollout:

```
$ kubectl set image deployment nginx nginx=nginx:1.13
deployment.apps/nginx image updated
$ kubectl annotate deployment nginx \
  kubernetes.io/change-cause="update to revision 1.13" \
  --record=false --overwrite=true
$ kubectl rollout history deployment nginx
deployment.apps/nginx
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
3         kubectl set image deployment nginx nginx=nginx:1.12 --record=true
4         update to revision 1.13
```

It is also possible to edit the specifications of the deployment:

```
$ kubectl edit deployment nginx
```

Your preferred editor opens, you can for example add an environment variable `FOO=bar` to the specification of the container:

```
[...]
    spec:
      containers:
      - image: nginx:1.13
        env:
        - name: FOO
          value: bar
[...]
```

After you save the template and quit the editor, the new revision is deployed. Let's verify that the new pod contains this environment variable:

```
$ kubectl describe pod -l app=nginx
[...]
    Environment:
      FOO:  bar
[...]
```

Let's set a change-cause for this release and see the history:

```
$ kubectl annotate deployment nginx \
  kubernetes.io/change-cause="add FOO environment variable" \
  --record=false --overwrite=true
$ kubectl rollout history deployment nginx
deployment.apps/nginx
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
3         kubectl set image deployment nginx nginx=nginx:1.12 --record=true
4         update to revision 1.13
5         add FOO environment variable
```

Now let's rollback the last rollout with the `undo` subcommand:

```
$ kubectl rollout undo deployment nginx
deployment.apps/nginx rolled back
$ kubectl rollout history deployment nginx
deployment.apps/nginx
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
3         kubectl set image deployment nginx nginx=nginx:1.12 --record=true
5         add FOO envvar
6         update to revision 1.13
```

We see that we switched back to the 4th release (which disappeared in the list and has been renamed as the 6th revision).

It is also possible to rollback to a specific revision. For example, to use the `nginx:1.12` image again:

```
$ kubectl rollout undo deployment nginx --to-revision=3
deployment.apps/nginx rolled back
$ kubectl rollout history deployment nginx
deployment.apps/nginx
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
5         add FOO envvar
6         update to revision 1.13
7         kubectl set image deployment nginx nginx=nginx:1.12 --record=true
```

Finally, you can verify that one `ReplicaSet` exists for each revision:

```
$ kubectl get replicaset
NAME               DESIRED   CURRENT   READY   AGE
nginx-65c8969d67   0         0         0       58m
nginx-68b47b4d58   0         0         0       62m
nginx-7856959c59   1         1         1       62m
nginx-84c7fd7848   0         0         0       62m
nginx-857df58577   0         0         0       62m
```

Deployment Strategies

You have seen in the Deployment controller that the deployment controller provides different **strategies** when changing the number of replicas of the old and new replicasets.

The Recreate Strategy

The simplest strategy is the **Recreate** strategy: in this case, the old replicaset will be downsized to zero, and when all pods of this replicaset are stopped, the new replicaset will be upsized to the number of requested replicas.

Some consequences are:

- there will be a small downtime, the time the old pods stop and the new pods start,
- no additional resources are necessary to run previous and new pods in parallel,
- old and new versions will not run simultaneously.

The RollingUpdate Strategy

The **RollingUpdate** strategy is a more advanced strategy, and the one by default when you create a Deployment.

The goal of this strategy is to update from previous to new version without downtime.

This strategy will combine the possibility to downsize and upsize replicasets and the possibility to expose pods through services.

You will see in Discovery and Load-balancing that pods are traditionally accessed through Services. A Service resource declares a list of Endpoints, which are the list of Pods that are exposed through this service. Pods are removed from endpoints of services when there are **not ready** to serve requests and are added when they become **ready** to serve requests.

The readiness of a pod is determined by the state of the **Readiness probes** declared for its containers. If you do not declare readiness probes for your containers, the risk is that the pods are detected ready before they really are, and traffic is sent to them while they are still in their startup phase.

During a rolling update, the deployment controller will, on the one hand:

- upsize the number of replicas of the new versions,
- when a replica is ready, it will be added to the service endpoints by the endpoints controller,

and on the other hand:

- mark replicas of old versions not ready, so they are removed from the service endpoints by the endpoints controller,
- stop these replicas.

Depending on traffic and available resources, you may want to either first augment the number of new versions replicas, then stop old replicas, or conversely first stop old replicas then start new versions replicas.

Fot this, the fields `maxSurge` and `maxUnavailable` of the Deployment `strategy` field indicate how many replicas can be present respectively in addition to or less than the expected number of replicas. Depending on these values, the Deployment controller will either first start new versions or conversely first stop old versions.

Configuring applications

An application can be configured in different ways:

- by passing arguments to the command,
- by defining environment variables,
- using configuration files.

Arguments to the command

We have seen in Container Specs that we can define the arguments of the command with the `Args` field of the container spec.

It is not possible to define the arguments of the command imperatively by using the `kubectl` commands `run` or `create`.

Depending on the definition of the image, it is possible you have to also specify the `Command` value (especially if the Dockerfile does not define an `ENTRYPOINT` but only a `CMD`).

You have to specify the arguments in the template defining the container:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
        command: ["nginx"]
        args: ["-g", "daemon off; worker_priority 10;"]
```

Environment Variables

It is possible to define environment variables for a container either by declaring their values directly, or by referencing their values from `ConfigMaps`, `Secrets` or fields of the object created (deployment, etc).

Declaring values directly

In declarative form

Declaratively, you can add Environment variables to the definition of a container:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
        env:
        - name: VAR1
          value: "value1"
        - name: VAR2
          value: "value2"
```

In imperative form

Imperatively, using the `kubectl run` command, you can define environment variables from the command line:

```
$ kubectl run --generator=run-pod/v1 nginx --image=nginx \
  --env VAR1=value1 \
  --env VAR2=value2
pod/nginx created
```

Note that the variant of the `kubectl run` command to create a deployment is deprecated, in favor of the `kubectl create deployment` command. Unfortunately, the `--env` flag is not accepted by this command. You can instead use `kubectl set env` command to add environment variables after you create the deployment:

```
$ kubectl create deployment nginx --image=nginx
deployment.apps/nginx created
$ kubectl set env deployment nginx \
  --env VAR1=value1 \
  --env VAR2=value2
deployment.apps/nginx env updated
```

Referencing specific values from ConfigMaps and Secrets

In declarative form

Declaratively, when declaring an environment variable, you can indicate that the values should be extracted from a ConfigMap or Secret, one by one:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: vars
data:
  var1: value1
  var2: value2
---
apiVersion: v1
kind: Secret
metadata:
  name: passwords
stringData:
  pass1: foo
---
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  labels:
    app: nginx
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
        env:
        - name: VAR1
          valueFrom:
            configMapKeyRef:
                key: var1
                name: vars
        - name: VAR2
          valueFrom:
            configMapKeyRef:
                key: var2
                name: vars
        - name: PASS1
          valueFrom:
            secretKeyRef:
                key: pass1
                name: passwords
```

Note that if referenced key is not found in the referenced configmaps or secrets, the creation of the deployment will fail. If you want to create the deployment even if a value deos not exist (in this case the corresponding environment variable will not be defined), you can use the `optional` field:

```
    - name: PASS2
      valueFrom:
        secretKeyRef:
            key: pass2
            name: passwords
            optional: true
```

In imperative form

Imperatively, you can also use the `kubectl set env` command with the `--from` and `keys` flags. In this example, you reference only some of the keys defined in the configmap and secret:

```
$ kubectl create configmap vars \
  --from-literal=var1=value1 \
  --from-literal=var2=value2 \
  --from-literal=var3=value3
configmap/vars created
$ kubectl create secret generic passwords \
  --from-literal=pass1=foo \
  --from-literal=pass2=bar
secret/passwords created
$ kubectl create deployment nginx --image=nginx
deployment.apps/nginx created
$ kubectl set env deployment nginx \
  --from=configmap/vars \
  --keys="var1,var2"
deployment.apps/nginx env updated
$ kubectl set env deployment nginx \
  --from=secret/passwords \
  --keys="pass1"
deployment.apps/nginx env updated
```

Referencing all values from ConfigMaps and Secrets

In declarative form

You can also inject all the entries of a ConfigMap or Secret as environment variables (you can also use in this case the `optional` field to indicate that the operation should succeed even if the referenced ConfigMap or Secret does not exist):

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: vars
data:
  var1: value1
  var2: value2
---
apiVersion: v1
kind: Secret
metadata:
  name: passwords
stringData:
  pass1: foo
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
        envFrom:
        - configMapRef:
            name: vars
        - secretRef:
            name: passwords
        - secretRef:
            name: notfound
            optional: true
```

In imperative form

Imperatively, you can also use the `kubectl set env` command with the `--from` flags:

```
$ kubectl create configmap vars \
  --from-literal=var1=value1 \
  --from-literal=var2=value2
configmap/vars created
$ kubectl create secret generic passwords \
  --from-literal=pass1=foo
secret/passwords created
$ kubectl create deployment nginx --image=nginx
deployment.apps/nginx created
$ kubectl set env deployment nginx \
  --from=configmap/vars
deployment.apps/nginx env updated
$ kubectl set env deployment nginx \
  --from=secret/passwords
deployment.apps/nginx env updated
```

Referencing values from Pod fields

Declaratively, it is possible to reference the value of some fields of the Pod:

- metadata.name,
- metadata.namespace,
- metadata.uid,
- spec.nodeName,
- spec.serviceAccountName,
- status.hostIP,
- status.podIP.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  replicas: 1
  selector:
```

```
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
        env:
        - name: POD_NAME
          valueFrom:
            fieldRef:
                fieldPath: metadata.name
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
                fieldPath: metadata.namespace
        - name: POD_UID
          valueFrom:
            fieldRef:
                fieldPath: metadata.uid
        - name: POD_NODENAME
          valueFrom:
            fieldRef:
                fieldPath: spec.nodeName
        - name: POD_SERVICEACCOUNTNAME
          valueFrom:
            fieldRef:
                fieldPath: spec.serviceAccountName
        - name: POD_HOSTIP
          valueFrom:
            fieldRef:
                fieldPath: status.hostIP
        - name: POD_PODIP
          valueFrom:
            fieldRef:
                fieldPath: status.podIP
```

After applying this template, you can examine the values of the environment variables into the container:

```
$ kubectl exec -it nginx-xxxxxxxxxx-yyyyy bash -- -c "env | grep POD_"
```

Referencing values from Container resources fields

Declaratively, it is possible to reference the values of resources requests and limits for a container. You can use the `divisor` field to divide the value by the given divisor:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
        resources:
          requests:
            cpu: "500m"
            memory: "500Mi"
          limits:
            cpu: "1"
            memory: "1Gi"
        env:
        - name: M_CPU_REQUEST
          valueFrom:
            resourceFieldRef:
                resource: requests.cpu
                divisor: "0.001"
        - name: MEMORY_REQUEST
          valueFrom:
            resourceFieldRef:
```

```
            resource: requests.memory
      - name: M_CPU_LIMIT
        valueFrom:
          resourceFieldRef:
            resource: limits.cpu
            divisor: "0.001"
      - name: MEMORY_LIMIT
        valueFrom:
          resourceFieldRef:
            resource: limits.memory
```

The variables will get the following values:

```
M_CPU_REQUEST=500
MEMORY_REQUEST=524288000
M_CPU_LIMIT=1000
MEMORY_LIMIT=1073741824
```

Configuration file from ConfigMap

It is possible to mount ConfigMap contents in the Container filesystem. Each key/value of the mounted ConfigMap will be a filename and its content in the mount directory.

For example, you can create this ConfigMap in declarative form:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config
data:
  nginx.conf: |
    server {
      location / {
        root /data/www;
      }

      location /images/ {
        root /data;
      }
    }
```

Or in imperative form:

```
$ cat > nginx.conf <<EOF
server {
    location / {
        root /data/www;
    }

    location /images/ {
        root /data;
    }
}
EOF
$ kubectl create configmap config --from-file=nginx.conf
configmap/config created
```

Then you can mount the ConfigMap in the Container:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      volumes:
      - name: config-volume
        configMap: config
      containers:
      - image: nginx
        name: nginx
        volumeMounts:
        - name: config-volume
          mountPath: /etc/nginx/conf.d/
```

Finally, the file `/etc/nginx/conf.d/nginx.conf` in the container will contain the value of the

`nginx.conf` key of the ConfigMap.

Configuration file from Secret

Similarly, it is possible to mount the contents of Secrets. first create a secret, in declarative form:

```
apiVersion: v1
kind: Secret
metadata:
  name: passwords
stringData:
  password: foobar
```

of imperative form:

```
$ kubectl create secret generic passwords \
  --from-literal=password=foobar
secret/passwords created
```

Then mount the Secret in the Container:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      volumes:
      - name: passwords-volume
        secret:
          secretName: passwords
```

```
    containers:
    - image: nginx
      name: nginx
      volumeMounts:
      - name: passwords-volume
        mountPath: /etc/passwords
```

Finally, the file `/etc/passwords/password` in the container will contain `foobar`.

Configuration file from Pod fields

Declaratively, it is possible to mount a volume with files containing Pod values:

- metadata.name,
- metadata.namespace,
- metadata.uid,
- metadata.labels,
- metadata.annotations.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
        volumeMounts:
        - name: pod-info
```

```
          mountPath: /pod
          readOnly: true
      volumes:
        - name: pod-info
          downwardAPI:
            items:
            - path: metadata/name
              fieldRef:
                fieldPath: metadata.name
            - path: metadata/namespace
              fieldRef:
                fieldPath: metadata.namespace
            - path: metadata/uid
              fieldRef:
                fieldPath: metadata.uid
            - path: metadata/labels
              fieldRef:
                fieldPath: metadata.labels
            - path: metadata/annotations
              fieldRef:
                fieldPath: metadata.annotations
```

As a result, in the container, you can find files in `/pod/metadata`:

```
$ kubectl exec nginx-xxxxxxxxxx-yyyyy bash -- -c \
  'for i in /pod/metadata/*; do echo $i; cat -n $i; echo ; done'
/pod/metadata/annotations
     1  kubernetes.io/config.seen="2020-01-11T17:21:40.497901295Z"
     2  kubernetes.io/config.source="api"
/pod/metadata/labels
     1  app="nginx"
     2  pod-template-hash="789ccf5b7b"
/pod/metadata/name
     1  nginx-xxxxxxxxxx-yyyyy
/pod/metadata/namespace
     1  default
/pod/metadata/uid
     1  631d01b2-eb1c-49dc-8c06-06d244f74ed4
```

Configuration file from container resources fields

Declaratively, it is possible to mount a volume with files containing the values of resources requests and limits for a container. You can use the `divisor` field to divide the value by the given divisor:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
        resources:
          requests:
            cpu: "500m"
            memory: "500Mi"
          limits:
            cpu: "1"
            memory: "1Gi"
        volumeMounts:
        - name: resources-info
          mountPath: /resources
          readOnly: true
      volumes:
        - name: resources-info
          downwardAPI:
            items:
            - path: limits/cpu
              resourceFieldRef:
                resource: limits.cpu
                divisor: "0.001"
                containerName: nginx
            - path: limits/memory
              resourceFieldRef:
                resource: limits.memory
                containerName: nginx
```

```
      - path: requests/cpu
        resourceFieldRef:
          resource: requests.cpu
          divisor: "0.001"
          containerName: nginx
      - path: requests/memory
        resourceFieldRef:
          resource: requests.memory
          containerName: nginx
```

As a result, in the container, you can find files in `/resources`:

```
$ kubectl exec nginx-85d7c97f64-9knh9 bash -- -c \
  'for i in /resources/*/*; do echo $i; cat -n $i; echo ; done'
/resources/limits/cpu
     1  1000
/resources/limits/memory
     1  1073741824
/resources/requests/cpu
     1  500
/resources/requests/memory
     1  524288000
```

Configuration file from different sources

It is possible to mount volumes with files containing information from a mix of ConfigMaps, Secrets, Pod fields and containers resources fields.

The main difference with previous examples is that it is here possible to mix values from these different sources inside the same directory.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: values
data:
  cpu: "4000"
  memory: "17179869184"
---
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  labels:
    app: nginx
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
        resources:
          requests:
            cpu: "500m"
            memory: "500Mi"
          limits:
            cpu: "1"
            memory: "1Gi"
        volumeMounts:
        - name: config
          mountPath: /config
          readOnly: true
      volumes:
      - name: config
        projected:
          sources:
          - configMap:
              name: values
              items:
              - key: cpu
                path: cpu/value
              - key: memory
                path: memory/value
          - downwardAPI:
              items:
              - path: cpu/limits
                resourceFieldRef:
                  resource: limits.cpu
```

```
        divisor: "0.001"
        containerName: nginx
    - path: memory/limits
      resourceFieldRef:
        resource: limits.memory
        containerName: nginx
    - path: cpu/requests
      resourceFieldRef:
        resource: requests.cpu
        divisor: "0.001"
        containerName: nginx
    - path: memory/requests
      resourceFieldRef:
        resource: requests.memory
        containerName: nginx
```

As a result, in the container, you can find files in `/config`:

```
$ kubectl exec nginx-7d797b5788-xzw79 bash -- -c \
  'for i in /config/*/*; do echo $i; cat -n $i; echo ; done'
/config/cpu/limits
     1  1000
/config/cpu/requests
     1  500
/config/cpu/value
     1  4000
/config/memory/limits
     1  1073741824
/config/memory/requests
     1  524288000
/config/memory/value
     1  17179869184
```

Scaling an application

We have seen in the Spec of the `ReplicaSet` and `Deployment` a `replicas` field. This field indicates how many replicas of a pod should be running.

Manual scaling

In declarative form, it is possible to edit the Spec of the Deployment to change the value of this field:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 4
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
```

In imperative form, the command `kubectl scale` is used to change this value:

```
$ kubectl create deployment nginx --image=nginx
deployment.apps/nginx created
$ kubectl scale deployment nginx --replicas=4
deployment.apps/nginx scaled
```

Auto-scaling

The `HorizontalPodAutoscaler` resource (commonly called **HPA**) can be used to scale deployments automatically depending on the CPU usage of the current replicas.

`HPA` depends on the installation of the Kubernetes Metrics Server[1] on the cluster.

To install it, run:

```
$ git clone https://github.com/kubernetes-sigs/metrics-server.git
$ cd metrics-server
$ kubectl create -f deploy/kubernetes/
```

You have to edit the `metrics-server` deployment in order to add the `--kubelet-insecure-tls` flag to the command started in the container:

```
$ kubectl edit deployment metrics-server -n kube-system
```

```
    spec:
      containers:
      - args:
        - --cert-dir=/tmp
        - --secure-port=4443
        - --kubelet-insecure-tls # add this line
```

You can verify that the installation succeeded when you get results from the following command:

```
$ kubectl top nodes
NAME         CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
controller   95m          9%     1256Mi          34%
worker-0     34m          3%     902Mi           25%
worker-1     30m          3%     964Mi           26%
```

You can now start a deployment with a single replica. Note that a CPU resource request is necessary for HPA to work:

[1] https://github.com/kubernetes-sigs/metrics-server

```
$ kubectl create deployment nginx --image=nginx
deployment.apps/nginx created
$ kubectl set resources --requests=cpu=0.05 deployment/nginx
deployment.extensions/nginx resource requirements updated
```

Create now an `HorizontalPodAutoscaler` resource for this deployment, that will be able to autoscale the deployment from 1 to 4 replicas, with a CPU utilization of 5%. In imperative form:

```
$ kubectl autoscale deployment nginx \
  --min=1 --max=4 --cpu-percent=5
horizontalpodautoscaler.autoscaling/nginx autoscaled
```

Or in declarative form:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
spec:
  minReplicas: 1
  maxReplicas: 4
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
  targetCPUUtilizationPercentage: 5
```

To augment the CPU utilization of the pod currently running, you can use the `curl` command to make a lot of requests on it:

```
$ kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
nginx-xxxxxxxxxx-yyyyy   1/1     Running   0          31s
$ kubectl port-forward pod/nginx-xxxxxxxxxx-yyyyy 8084:80
Forwarding from 127.0.0.1:8084 -> 80
```

And in another terminal:

```
$ while : ; do curl http://localhost:8084; done
```

In parallel, you can follow the utilization of CPU by the pod with the command:

```
$ kubectl top pods
NAME                     CPU(cores)   MEMORY(bytes)
nginx-xxxxxxxxxx-yyyyy   3m           2Mi
```

Once the CPU utilization becomes more than 5%, a second pod will be automatically deployed:

```
$ kubectl get hpa nginx
NAME    REFERENCE          TARGETS   MINPODS   MAXPODS   REPLICAS
nginx   Deployment/nginx   7%/5%     1         4         2

$ kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
nginx-5c55b4d6c8-fgnlz   1/1     Running   0          12m
nginx-5c55b4d6c8-hzgfl   1/1     Running   0          81s
```

If you stop the `curl` requests and watch the created HPA, you can see that the number of replicas will be set to 1 again 5 minutes after the CPU utilization is low again:

```
$ kubectl get hpa nginx -w
NAME    REFERENCE          TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
nginx   Deployment/nginx   4%/5%     1         4         2          10m
nginx   Deployment/nginx   3%/5%     1         4         2          10m
nginx   Deployment/nginx   0%/5%     1         4         2          11m
nginx   Deployment/nginx   0%/5%     1         4         1          16m
```

And you can examine the events created by the HPA:

```
$ kubectl describe hpa nginx
[...]
Events:
  Type    Reason             Age    From                       Message
  ----    ------             ----   ----                       -------
  Normal  SuccessfulRescale  10m    horizontal-pod-autoscaler  New size: 2; reason: \
cpu resource utilization (percentage of request) above target
  Normal  SuccessfulRescale  2m47s  horizontal-pod-autoscaler  New size: 1; reason: \
All metrics below target
```

Application Self-Healing

When you start a Pod on a cluster, it is scheduled on a specific node of the cluster. If the node, at a given moment, is not able to continue to host this Pod, the Pod will not be restarted on a new node: the application is not self-healing.

Let's have a try, on a cluster with more than one worker (for example, on the cluster installed in Creating a cluster with kubeadm).

First run a Pod, then examine on which node it has been scheduled:

```
$ kubectl run --generator=run-pod/v1 nginx --image=nginx
pod/nginx created
$ kubectl get pods -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP           NODE
nginx   1/1     Running   0          12s   10.244.1.8   worker-0
```

Here, the pod has been scheduled on the node `worker-0`.

Let's put this node in maintenance mode, to see what happens to the Pod:

```
$ kubectl drain worker-0 --force
node/worker-0 cordoned
WARNING: deleting Pods not managed by ReplicationController, ReplicaSet, Job, Daemon\
Set or StatefulSet: default/nginx
evicting pod "nginx"
pod/nginx evicted
node/worker-0 evicted
$ kubectl get pods
No resources found in default namespace.
```

You can see that the Pod you created has disappeared and has not been recreated in another node.

We finished our experiment, you can make your node schedulable again:

```
$ kubectl uncordon worker-0
node/worker-0 uncordoned
```

Controller to the rescue

We have seen in Pod Controllers that using Pod Controller ensures your pod is scheduled in another node if one node stops to work.

Let's make the experience again, with a `Deployment`:

```
$ kubectl create deployment nginx --image=nginx
deployment.apps/nginx created
$ kubectl get pods -o wide
NAME                     READY   STATUS    RESTARTS   AGE   IP           NODE
nginx-554b9c67f9-ndtsz   1/1     Running   0          11s   10.244.1.9   worker-0
$ kubectl drain worker-0
node/worker-0 cordoned
evicting pod "nginx-554b9c67f9-ndtsz"
pod/nginx-554b9c67f9-ndtsz evicted
node/worker-0 evicted
$ kubectl get pods -o wide
NAME                     READY   STATUS    RESTARTS   AGE   IP           NODE
nginx-554b9c67f9-5kz5v   1/1     Running   0          4s    10.244.2.9   worker-1
```

This time, we can see that a Pod has been recreated in another node of the cluster: our app now survives a node eviction.

Liveness probes

It is possible to define a liveness probe for each container of a Pod. If the kubelet is not able to execute the probe successfully a given number of times, the container is considered not healhty and is restarted into the same Pod.

This probe should be used to detect that the container is not responsive.

There are three possibilities for the liveness probe:

- make an HTTP request,

  if your container is an HTTP server, you can add an endpoint that always replies with a success response, and define the probe with this endpoint. If you backend is not healthy anymore, it is probable that this endpoint will not respond either.
- execute a command.

  most server applications have an associate CLI application. You can use this CLI to execute a very simple operation on the server. If the server is not healhty, it is probable it will not respond to this simple request either.
- make a TCP connection,

  when a server running in a container communicates via a non-HTTP protocol (on top of TCP), you can try to open a socket to the application. If the server is not healthy, it is probable that it will not respond to this connection request.

You have to use the declarative form to declare liveness probes.

A note about Readiness probes

Note that it is also possible to define a **Readiness probe** for a container. The main role of the readiness probe is to indicate if a Pod is ready to serve network requests. The pod will be added to the list of backends of matching `Services` when the readiness probe succeeds.

Later during the container execution, if a readiness probe fails, the Pod will be removed from the list of backends of `Services`. This can be useful to detect that a container is not able to handle more connections (if it is already treating a lot of connections for example) and stop sending new ones.

HTTP request liveness probe

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    livenessProbe:
      httpGet:
        scheme: HTTP
        port: 80
        path: /healthz
```

Here, we define a probe that queries the `/healthz` endpoint. As **nginx** is not configured by default to reply to this path, it will reply with a 404 response code and the probe will fail. This is not a real case, but that simulates an nginx server that would reply in error to a simple request.

You can see in the Pod events that after three failed probes, the container is restarted:

```
$ kubectl describe pod nginx
[...]
Events:
  Type     Reason     Age                From               Message
  ----     ------     ----               ----               -------
  Normal   Started    31s                kubelet, minikube  Started container nginx
  Normal   Pulling    0s (x2 over 33s)   kubelet, minikube  Pulling image "nginx"
  Warning  Unhealthy  0s (x3 over 20s)   kubelet, minikube  Liveness probe failed: HT\
TP probe failed with statuscode: 404
  Normal   Killing    0s                 kubelet, minikube  Container nginx failed li\
veness probe, will be restarted
```

Command liveness probe

```
apiVersion: v1
kind: Pod
metadata:
  name: postgres
spec:
  containers:
  - image: postgres
    name: postgres
    livenessProbe:
      initialDelaySeconds: 10
      exec:
        command:
        - "psql"
        - "-h"
        - "localhost"
        - "-U"
        - "unknownUser"
        - "-c"
        - "select 1"
```

Here, the liveness probe tries to connect to the server using the `psql` command and execute a very simple SQL query (`SELECT 1`) as user `unknownUser`. As this user does not exist, the query will fail.

You can see in the Pod events that after three failed probes, the container is restarted:

```
$ kubectl describe pod postgres
[...]
Events:
  Type     Reason     Age                From               Message
  ----     ------     ----               ----               -------
  Normal   Scheduled  <unknown>                default-scheduler  Successfully assig\
ned default/postgres to minikube
  Warning  Unhealthy  0s (x3 over 20s)         kubelet, minikube  Liveness probe fai\
led: psql: error: could not connect to server: FATAL:  role "unknownUser" does not e\
xist
  Normal   Killing    0s                       kubelet, minikube  Container postgres\
 failed liveness probe, will be restarted
```

TCP connection liveness probe

```
apiVersion: v1
kind: Pod
metadata:
  name: postgres
spec:
  containers:
  - image: postgres
    name: postgres
    livenessProbe:
      initialDelaySeconds: 10
      tcpSocket:
        port: 5433
```

Here, the liveness probe tries to connect to the container on the `5433` port. As `postgres` listens on the port `5432`, the connection will fail.

You can see in the Pod events that after three failed probes, the container is restarted:

```
$ kubectl describe pod postgres
[...]
Events:
  Type     Reason     Age                From               Message
  ----     ------     ----               ----               -------
  Normal   Started    25s                kubelet, minikube  Started container postgres
  Warning  Unhealthy  0s (x3 over 15s)   kubelet, minikube  Liveness probe failed: di\
al tcp 172.17.0.3:5433: connect: connection refused
  Normal   Killing    0s                 kubelet, minikube  Container postgres failed\
 liveness probe, will be restarted
```

Resource limits and Quality of Service classes

You can define for each container of Pods resources (CPU and memory) requests and limits.

The resources requests values are used to schedule a Pod in a node having at least the requested resources available (see Resource requests).

If you do not declare limits, each container will still have access to all the resources of the node: in this case, if some pods are not using all their requested resources at a given time, some other containers will be able to use them, and vice versa.

In contrast, if a limit is declared for a container, the container will be constrained to that resources. If it tries to allocate more memory than its limit, it will get a memory allocation error and will probably crash or work on a degraded mode; and it will have access to the CPU in its limit only.

Depending on whether the requests and limits values are declared or not, a different Quality of Service is assured for a Pod:

- if all containers of a Pod have declared requests and limits for all resources (CPU and memory), and the limits equal the requests, the Pod will be running with a *Guaranteed* QoS class,
- or, if at least one container of a Pod has a resource request or limit, the Pod will be running with a *Burstable* QoS class,
- otherwise, if no request nor limit are declared for its containers, the Pod will be running with a *Best Effort* QoS class.

If a node runs out of an incompressible resource (memory), the associated kubelet can decide to eject one or more Pods, to prevent total starvation of the resource.

The Pods evicted are decided depending on their Quality of Service class: first *Best Effort* ones, then *Burstable* ones, and finally the *Guaranteed* ones.

Scheduling Pods

When you want to run a Pod into a Kubernetes cluster, you generally do not specify on which node you want the Pod to run. This is the job of the Kubernetes scheduler to determine on which node it will be running.

The Pod Specs contain a `nodeName` field indicating on which node the Pod is scheduled.

The scheduler perpetually watches for Pods; when it finds a Pod with an empty `nodeName` field, the scheduler determines the best node on which to schedule this pod, then modifies the Pod Spec to write the `nodeName` field with the selected node.

In parallel, the kubelet components, affected to specific nodes, watch the Pods; when a Pod marked with a `nodeName` matches the node of a kubelet, the Pod is affected to the kubelet, which deploys it to its node.

Using label selectors to schedule Pods on specific nodes

The Pod Spec contains a `nodeSelector` field, as a map of key-value pairs. When set, the Pod is deployable only on nodes having each value-pair as label.

The typical usage is that nodes have some labels indicating some specificities, and when you want to deploy a Pod on a node having a specificity, you add the corresponding label to the `nodeSelector` field of the Pod.

Adding labels to nodes

The first step is to add labels to nodes. Consider you have four nodes, two with SSD disks and two with HDD disks. You can label the nodes with the commands:

```
$ kubectl label node worker-0 disk=ssd
node/worker-0 labeled
$ kubectl label node worker-1 disk=ssd
node/worker-1 labeled
$ kubectl label node worker-2 disk=hdd
node/worker-2 labeled
$ kubectl label node worker-3 disk=hdd
node/worker-3 labeled
```

Among these nodes, two provide GPU units. Let's label them:

```
$ kubectl label node worker-0 compute=gpu
node/worker-0 labeled
$ kubectl label node worker-2 compute=gpu
node/worker-1 labeled
```

Adding node selectors to pods

Imagine you want to deploy a Pod that needs an SSD disk, you can create this deployment. The Pod will be schedulable on `worker-0` and `worker-1`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      nodeSelector:
        disk: ssd
      containers:
      - image: nginx
        name: nginx
```

Now, if the Pod needs an SSD disk **and** a GPU unit, you can create this deployment. The Pod will be schedulable on `worker-0` only:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      nodeSelector:
        disk: ssd
        compute: gpu
      containers:
      - image: nginx
        name: nginx
```

Manual scheduling

Remember, the Kubernetes scheduler looks for Pods with empty `nodeName` and kubelet components look for Pods with their associated node name.

If you create a Pod and specify yourself the `nodeName` in its Spec, the scheduler will never see it, and the associated kubelet will adopt it immediately. The effect is that the Pod will be scheduled to the specified node, without the help of the scheduler.

DaemonSets

The DaemonSet Kubernetes resource guarantees that all (or a given subset of) nodes run a copy of a given Pod.

The typical use of a DaemonSet is to deploy daemons (storage daemons, logs daemons, monitoring daemons) on every node of a cluster.

Because of this particularity, the Pods created by DaemonSets are not scheduled by the Kubernetes scheduler, but by the DaemonSet controller itself.

The Spec of a DaemonSet is similar to a Deployment Spec, with these differences:

- The DaemonSet Spec does not contain a `replicas` field, as this quantity is given by the number of selected nodes,
- The `strategy` field of the Deployment is replaced by an `updateStrategy` field in the DaemonSet,
- the `progressDeadlineSeconds` and `paused` fields are absent from the DaemonSet spec.

By default, the DaemonSet will deploy its Pod on every node of the cluster. If you want to select a subset of nodes only, you can use the `nodeSelector` field of the Pod Spec to select the nodes by label, as you would do for a Deployment (see Using label selectors to schedule Pods on specific nodes).

As an example, here is a DaemonSet that would deploy an hypothetical GPU daemon on nodes labelled with `compute=gpu`:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: gpu-daemon
  labels:
    app: gpu-daemon
spec:
  selector:
    matchLabels:
      app: gpu-daemon
  template:
    metadata:
      labels:
        app: gpu-daemon
    spec:
      nodeSelector:
        compute: gpu
      containers:
      - image: gpu-daemon
        name: gpu-daemon
```

Static Pods

Static Pods are directly attached to a kubelet daemon. They are declared in files located on the host of the node running the kubelet daemon, in a specific directory.

You can find the directory in the kubelet configuration file, under the `staticPodPath` field.

The configuration of a kubelet is accessible via the Kubernetes API, at the following path: `/api/v1/nodes/<node-name>/proxy/configz`

To access the API easily, you can run the `kubectl proxy` command:

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

You now have access to the API on `http://127.0.0.1:8001`, with the same rights you have with `kubectl`.

Now in another terminal, you can execute the `curl` command to get the configuration of the kubelet daemon on the `worker-0` node:

```
$ curl "http://localhost:8001/api/v1/nodes/worker-0/proxy/configz" \
  | python -m json.tool
{
    "kubeletconfig": {
        "staticPodPath": "/etc/kubernetes/manifests",
[...]
    }
}
```

You can now create a manifest to declare a Pod on this directory, on the `worker-0` host:

```
$ gcloud compute ssh worker-0
Welcome to Ubuntu 18.04.3 LTS
$ cat <<EOF | sudo tee /etc/kubernetes/manifests/nginx.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
EOF
```

Back to your developer machine, you can see that the Pod appears in the list of the running Pods, and that kubelet suffixed the name of the pod with the name of the node:

```
$ kubectl get pods
NAME             READY   STATUS    RESTARTS   AGE
nginx-worker-0   1/1     Running   0          11s
```

If you delete the Pod, it will be immediately recreated by kubelet:

```
$ kubectl delete pod nginx-worker-0
pod "nginx-worker-0" deleted
$ kubectl get pods
NAME             READY   STATUS    RESTARTS   AGE
nginx-worker-0   0/1     Pending   0          1s
```

When you remove the manifest from the `worker-0` host, the Pod will be immediately deleted by kubelet.

```
$ gcloud compute ssh worker-0
Welcome to Ubuntu 18.04.3 LTS
$ sudo rm /etc/kubernetes/manifests/nginx.yaml
```

Resource requests

Each node has a maximum capacity of CPUs and memory. Each time a Pod is scheduled to a node, the CPUs and memory amounts requested by this Pod are removed from the CPUs and memory available on this node.

A Pod cannot be scheduled to a node if the available resources on this node are less than the amounts requested by this Pod.

For this raeson, it is very important to declare resources requests for **all** the Pods you deploy. Otherwise, the computation of the available resources would be inaccurate.

In imperative form

The `kubectl set resources` command is used to set resources requests on an object (here a Deployment):

```
$ kubectl create deployment nginx --image=nginx
deployment.apps/nginx created
$ kubectl set resources deployment nginx \
  --requests=cpu=0.1,memory=1Gi
deployment.extensions/nginx resource requirements updated
```

In declarative form

You can declare resource requests for each container of a Pod, using the `resources` field of the Container Spec.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
        resources:
          requests:
            cpu: 100m
            memory: 1Gi
```

Running multiple schedulers

Kubernetes is shipped with a default scheduler and gives you the possibility to run your own schedulers in parallel. In that case, when you create a Pod, you will be able to select the scheduler you want to be used for this Pod.

You can get an example scheduler on the feloy/scheduler-round-robin GitHub repository[2].

[2] https://github.com/feloy/scheduler-round-robin

The code of this scheduler is simplistic and must not be used in production, but demonstrates the lifecycle of a scheduler:

- listening for Pods without a `nodeName` value,
- selecting a node,
- binding the selected node to the Pod,
- sending an event.

Once this new scheduler is deployed in your cluster (please follow the instructions in the repository), you can verify that the scheduler has found the workers of the cluster:

```
$ kubectl logs scheduler-round-robin-xxxxxxxxxx-yyyyy -f
found 2 nodes: [worker-0 worker-1]
```

You can now create a deployment specifying this specific scheduler:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  replicas: 4
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      schedulerName: scheduler-round-robin
      containers:
      - image: nginx
        name: nginx
```

Using the command `kubectl get pods -o wide`, you can see that two pods have been deployed in `worker-0` and two in `worker-1`.

Examine schedulers events

If you look at the events attached to a deployed Pod, you can see that is has been scheduled by the `scheduler-round-robin` scheduler:

```
$ kubectl describe pod nginx-xxxxxxxxxx-yyyyy
[...]
Events:
  Type    Reason     Age   From                   Message
  ----    ------     ----  ----                   -------
  Normal  Scheduled   30s  scheduler-round-robin  pod nginx-6dcb7cd47-9c9w5 schedule\
d to node worker-0
```

You can go through all the events to find the events created by schedulers:

```
$ kubectl get events | grep Scheduled
0s          Normal    Scheduled                pod/nginx-554b9c67f9-snpkb          \
          Successfully assigned default/nginx-554b9c67f9-snpkb to worker-0
```

Discovery and Load-balancing

When you deploy a Pod, it is not easily accessible. If you define a Pod with several containers, these containers will be available to communicate via the localhost interface, but containers of a Pod won't be able to communicate with containers of another pod without knowing the IP address of the other Pod.

But the Pod is volatile. We have seen that a Pod by itself is not usable because it can be evicted from a node at any time and won't be recreated automatically. Controllers like `ReplicaSet` are necessary to guarantee that a given number of replicas of a Pod are running. In this case, even if you get the IP address of a Pod, it is not guaranteed that this Pod survives, and the next one won't have the same IP address of the first one.

Services

The `Service` Kubernetes resource is used to make a Pod accessible via the network in a reproducible way.

In imperative form, you can use the `kubectl expose` command:

```
$ kubectl create deployment nginx --image=nginx
deployment.apps/nginx created
$ kubectl expose deployment nginx --port 80
service/webapp exposed
$ kubectl get services
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
webapp       ClusterIP   10.97.68.130   <none>        80/TCP    5s
```

In declarative form, you can create the service with this template:

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  ports:
  - port: 80
  selector:
    app: nginx
```

Now, we can try to deploy another pod, connect to it and try to communicate with this `nginx` pod:

```
$ kubectl run --generator=run-pod/v1 \
  --image=busybox box \
   sh -- -c 'sleep $((10**10))'
pod/box created
$ kubectl exec -it box sh
/ # wget http://webapp -q -O -
<!DOCTYPE html>
<html>
[...]
</html>
/ # exit
```

Selectors

You have seen with the previous examples that a Service makes a Pod accessible. More generally, a Service is a frontend for a list of **Backend** Pods. This list of Backend pods is determined by the `selector` field of the `Service` resource: all the pods with labels keys and values matching this selector are eligible to be part of the list of Backends.

Readiness Probes

Once the list of Pods eligible to be a backend of a Service is determined by the `selector` field, the readiness of the pods is also taken into account. The pods are effectively inserted in the list of backends when the pod is in the **Ready** state.

The Pod is considered ready when all its containers are ready, and a container is ready either when it does not define a `readinessProbe` or when its `readinessProbe` succeeds.

Note that this readiness is considered when the pod starts, but also during all the life of the pod. At any time, a container can declare itself not ready (because it thinks it is not able to handle more requests for example), and the pod will be immediately removed from the list of backends of matching services.

Endpoints

The effective backends of a `Service` are materialized by the `Endpoints` resource. The Endpoints controller is in charge of creating and deleting endpoints, depending on the readiness of pods and selectors of Services.

You can see the list of endpoints for a service (here `nginx`) with the command:

```
$ kubectl get endpoints nginx
NAME    ENDPOINTS                       AGE
nginx   172.17.0.10:80,172.17.0.9:80    7m48s
```

In this case, the service has two endpoints and the traffic for the service will be routed to these two endpoints.

You can get the details of the two endpoints. Here, you see that the two endpoints are the two pods `nginx-86c57db685-g4fqr` and `nginx-86c57db685-9hp58`:

```
$ kubectl get endpoints nginx -o yaml
apiVersion: v1
kind: Endpoints
metadata:
  labels:
    app: nginx
  name: nginx
  [...]
subsets:
- addresses:
  - ip: 172.17.0.10
    nodeName: minikube
    targetRef:
      kind: Pod
      name: nginx-86c57db685-g4fqr
      namespace: default
      resourceVersion: "621228"
      uid: adb2d120-14ed-49ad-b9a5-4389412b73b1
  - ip: 172.17.0.9
    nodeName: minikube
    targetRef:
      kind: Pod
      name: nginx-86c57db685-9hp58
      namespace: default
      resourceVersion: "621169"
      uid: 5b051d79-9951-4ca9-a436-5dbe3f46169b
```

```
  ports:
  - port: 80
    protocol: TCP
```

Services Types

ClusterIP

By default, the service is created with the `ClusterIP` type. With this type, the service is accessible from inside the cluster only.

An IP address local to the cluster will be reserved for this service, and a DNS entry will be created that points to this address, under the form `<name>.<namespace>.svc.cluster.local`, in our example `webapp.default.svc.cluster.local`.

If you examine the `resolv.conf` file inside a container, you see that the `search` entry indicates:

```
$ kubectl exec -it box cat /etc/resolv.conf
nameserver 10.96.0.10
search default.svc.cluster.local svc.cluster.local cluster.local
```

Thanks to this, from inside a Pod, you can access a service defined in the namespace with its name only (here `webapp`) or its name.namespace (here `webapp.default`) or name.namespace.svc (here `webapp.default.svc`) or its complete name (`webapp.default.svc.cluster.local`).

To access a service defined in another namespace, you will have to specify at least the name and namespace (for example `another-app.other-namespace`).

NodePort

If you want to access a Service from outside the cluster, you can use the `NodePort` type. In addition to creating a ClusterIP, this will allocate a port (in the range 30000-32767 by default) on every node of the cluster, which will route to the ClusterIP.

LoadBalancer

If you want to access a Service from outisde the Cloud environment, you can use the `LoadBalancer` type. In addition to creating a `NodePort`, it will create an external load-balancer (if you use a managed Kubernetes cluster like Google GKE, Azure AKS, Amazon EKS, etc), which routes to the ClusterIP via the `NodePort`.

ExternalName

This is a specific type of Service, where the `selector` field is not used and the service redirects instead to an external DNS name, using a DNS `CNAME` record.

Ingress

With a LoadBalancer service, you can access a micro-service of your application. If you have several applications, each with several access points (at least frontend and api), you will need to reserve lots of load balancers, which can be very costly.

An `Ingress` is equivalent to an Apache or nginx **Virtual host**: it permits to multiplex access points to several micro-services into a single load-balancer. The selection is done on the hostname and path of the request.

You have to install an **Ingress controller** in your cluster in order to use `Ingress` resources.

Install nginx Ingress controller

You can follow the installation instructions[3]. In summary, you have to execute:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-\
0.27.1/deploy/static/mandatory.yaml
[...]
$ kubectl create namespace ingress-nginx
namespace/ingress-nginx created
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-\
0.27.1/deploy/static/provider/baremetal/service-nodeport.yaml
service/ingress-nginx created
```

Get the ports on which the ingress controller listens for external connections, in this case the ports *32351* (mapped to port 80) and *31296* (mapped to 443), and store them on environment variables for later use:

[3] https://kubernetes.github.io/ingress-nginx/deploy/

```
$ kubectl get services -n ingress-nginx
NAME            TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)                      \
   AGE
ingress-nginx   NodePort   10.100.169.243   <none>        80:32351/TCP,443:31296/TCP\
   9h
$ HTTP_PORT=32351
$ HTTPS_PORT=31296
```

Note that the `ingress-nginx` service is of type `NodePort`: the service will be accessible on each worker of the cluster, on the ports 32351 (for HTTP connections) and 31296 (for HTTPS connections).

We have to add a firewall rule that enables the traffic on these ports on the worker VMs:

```
$ gcloud compute firewall-rules create \
  kubernetes-cluster-allow-external-ingress \
  --allow tcp:$HTTP_PORT,tcp:$HTTPS_PORT \
  --network kubernetes-cluster \
  --source-ranges 0.0.0.0/0
```

Get the public IP address of the first worker:

```
$ WORKER_IP=$(gcloud compute instances describe worker-0 \
  --zone $(gcloud config get-value compute/zone) \
  --format='get(networkInterfaces[0].accessConfigs[0].natIP)')
```

From your local computer, you can try to connect to these ports:

```
$ curl http://$WORKER_IP:$HTTP_PORT
<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx/1.17.7</center>
</body>
</html>
$ curl -k https://$WORKER_IP:$HTTPS_PORT
<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx/1.17.7</center>
</body>
</html>
```

If you can see these responses, the ingress controller is running correctly and you are redirected to the default backend, that returns 404 errors.

Accessing applications

Now, let's create a simple application (an Apache server) and expose it via an Ingress resource:

```
$ kubectl create deployment webapp --image=httpd
deployment.apps/webapp created
$ kubectl expose deployment webapp --port 80
service/webapp exposed
$ kubectl apply -f - <<EOF
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: webapp-ingress
spec:
  backend:
    serviceName: webapp
    servicePort: 80
EOF
```

With this Ingress configuration, all the requests to the ingress controller will be routed to the `webapp` service:

```
$ curl http://$WORKER_IP:$HTTP_PORT/
<html><body><h1>It works!</h1></body></html>
```

Now, let's multiplex several applications on the same Ingress, by deploying a second web application, this time the `kennship/http-echo` image:

```
$ kubectl create deployment echo --image=kennship/http-echo
deployment.apps/echo created
$ kubectl expose deployment echo --port 3000
service/echo exposed
$ kubectl delete ingress webapp-ingress
ingress.extensions "webapp-ingress" deleted
$ kubectl apply -f - <<EOF
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: plex-ingress
```

```
spec:
  rules:
  - host: webapp.com
    http:
      paths:
      - path: /
        backend:
          serviceName: webapp
          servicePort: 80
  - host: echo.com
    http:
      paths:
      - path: /
        backend:
          serviceName: echo
          servicePort: 3000
EOF
```

We can now access the different applications by using their host names:

```
$ curl -H 'Host: echo.com' http://$WORKER_IP:$HTTP_PORT/
{"path":"/","headers":{"host":"echo.com","x-request-id":"3502371d3a479598bc393aa61a8\
b6896","x-real-ip":"10.240.0.20","x-forwarded-for":"10.240.0.20","x-forwarded-host":\
"echo.com","x-forwarded-port":"80","x-forwarded-proto":"http","x-scheme":"http","use\
r-agent":"curl/7.58.0","accept":"*/*"},"method":"GET","body":{},"fresh":false,"hostn\
ame":"echo.com","ip":"::ffff:10.244.1.49","ips":[],"protocol":"http","query":{},"sub\
domains":[],"xhr":false}

$ curl -H 'Host: webapp.com' http://$WORKER_IP:$HTTP_PORT/
<html><body><h1>It works!</h1></body></html>
```

HTTPS and Ingress

If you try to connect using HTTPS protocol, on the HTTPS port of the Ingress controller (here 31296), you can see that the Ingress controller is using a fake certificate:

```
$ curl -k -v -H 'Host: webapp.com' https://$WORKER_IP:$HTTPS_PORT/
[...]
* Server certificate:
*  subject: O=Acme Co; CN=Kubernetes Ingress Controller Fake Certificate
*  start date: Jan 17 16:59:01 2020 GMT
*  expire date: Jan 16 16:59:01 2021 GMT
*  issuer: O=Acme Co; CN=Kubernetes Ingress Controller Fake Certificate
[...]
```

Let's use our own certificate, in this example an auto-generated one, but the procedure would be the same with a signed one. First generate the certificate, then create a `tls` secret containing the certificate:

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
    -out echo-ingress-tls.crt \
    -keyout echo-ingress-tls.key \
    -subj "/CN=echo.com/O=echo-ingress-tls"
[...]
$ kubectl create secret tls echo-ingress-tls \
    --key echo-ingress-tls.key \
    --cert echo-ingress-tls.crt
secret/echo-ingress-tls created
```

Then add a section to the `Ingress` resource:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: plex-ingress
spec:
  tls:
  - hosts:
    - echo.com
    secretName: echo-ingress-tls
  rules:
  - host: webapp.com
    http:
      paths:
      - path: /
        backend:
          serviceName: webapp
          servicePort: 80
```

```
  - host: echo.com
    http:
      paths:
      - path: /
        backend:
          serviceName: echo
          servicePort: 3000
```

With these changes, the `echo.com` requests will now use this new certificate:

```
$ curl -k -v --resolve echo.com:$HTTPS_PORT:$WORKER_IP  https://echo.com:$HTTPS_PORT/
[...]
* Server certificate:
*  subject: CN=echo.com; O=echo-ingress-tls
*  start date: Jan 17 18:10:33 2020 GMT
*  expire date: Jan 16 18:10:33 2021 GMT
*  issuer: CN=echo.com; O=echo-ingress-tls
*  SSL certificate verify result: self signed certificate (18), continuing anyway.
[...]
```

And `webapp.com` will still use the default one:

```
$ curl -k -v --resolve webapp.com:$HTTPS_PORT:$WORKER_IP  https://webapp.com:$HTTPS_\
PORT/
[...]
* Server certificate:
*  subject: O=Acme Co; CN=Kubernetes Ingress Controller Fake Certificate
*  start date: Jan 17 16:59:01 2020 GMT
*  expire date: Jan 16 16:59:01 2021 GMT
*  issuer: O=Acme Co; CN=Kubernetes Ingress Controller Fake Certificate
[...]
```

Security

Authentication

Kubernetes defines two kind of users: **normal users** and **service accounts**.

Normal user authentication

Normal users are not managed by the Kubernetes API. You must have an external system managing users and their credentials. Authentication for normal users can be handled by different methods:

- Client Certificate,
- HTTP Basic Auth,
- Bearer Token,
- Authentication proxy.

Client Certificate authentication

When using `kubeadm` to install a cluster, the API server is configured with the option:

```
--client-ca-file=/etc/kubernetes/pki/ca.crt
```

This option is necessary for the cluster to enable Client Certificates authentication. The `ca.crt` contains the certificate authority.

For a new user, the first step for the user is to create a certificate signing request (CSR):

```
# Create a private key
$ openssl genrsa -out user.key 4096
[...]
```

Create the `csr.cnf` configuration file to generate the CSR:

```
[ req ]
default_bits = 2048
prompt = no
default_md = sha256
distinguished_name = dn

[ dn ]
CN = user
O = company

[ v3_ext ]
authorityKeyIdentifier=keyid,issuer:always
basicConstraints=CA:FALSE
keyUsage=keyEncipherment,dataEncipherment
extendedKeyUsage=serverAuth,clientAuth
```

Create the CSR:

```
$ openssl req -config ./csr.cnf -new -key user.key -nodes -out user.csr
```

Second, an admin of the cluster have to sign the CSR using the `CertificateSigningRequest` resource provided by the Kubernetes API:

```
# Write user CSR as base64 data
$ export CSR=$(cat user.csr | base64 | tr -d '\n')

# Create a template for the CertificateSigningRequest
$ cat > user-csr.yaml <<EOF
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: user-csr
spec:
  groups:
  - system:authenticated
  request: ${CSR}
  usages:
  - digital signature
  - key encipherment
  - server auth
  - client auth
EOF
```

```
# Insert CSR data into the template and apply it
$ cat user-csr.yaml | envsubst | kubectl apply -f -
certificatesigningrequest.certificates.k8s.io/user-csr created

# Verify CSR resource is created
$ kubectl get certificatesigningrequests.certificates.k8s.io user-csr
NAME       AGE   REQUESTOR          CONDITION
user-csr   52s   kubernetes-admin   Pending

# Approve the certificate
$ kubectl certificate approve user-csr
certificatesigningrequest.certificates.k8s.io/user-csr approved

# Verify CSR is approved ans issued
$ kubectl get certificatesigningrequests.certificates.k8s.io user-csr
NAME       AGE     REQUESTOR          CONDITION
user-csr   2m17s   kubernetes-admin   Approved,Issued

# Extract the issued certificate
$ kubectl get csr user-csr -o jsonpath='{.status.certificate}' \
  | base64 --decode > user.crt
```

Next, the admin have to create a kubeconfig `file` for the user:

There is three parts in the kubeconfig files: server information, user information and context information.

The command `kubectl config set-cluster` is used to write server information:

> Note in the following commands the presence of the flag `--kubeconfig=userconfig`. When ths flag is set, the `kubectl` tool will work on this file, instead of on the default one `~/.kube/config`.

```
$ CLUSTER_ENDPOINT=$(kubectl config view --minify --raw -o jsonpath='{.clusters[0].c\
luster.server}')
$ CLUSTER_CA=$(kubectl config view --minify --raw -o jsonpath='{.clusters[0].cluster\
.certificate-authority-data}')
$ echo -n $CLUSTER_CA | base64 -d > cluster.crt
$ CLUSTER_NAME=$(kubectl config view --minify --raw -o jsonpath='{.clusters[0].name}\
')
$ kubectl --kubeconfig=userconfig config set-cluster $CLUSTER_NAME \
   --server=$CLUSTER_ENDPOINT \
   --certificate-authority=cluster.crt --embed-certs
Cluster "cka" set.
```

The command `kubectl config set-credentials` is used to write user information:

```
$ USER_NAME=user
$ kubectl --kubeconfig=userconfig config set-credentials $USER_NAME \
   --client-certificate=user.crt --embed-certs
User "user" set.
```

The command `kubectl config set-context` is used to write context information and `kubectl config use-context` to select the current context:

```
$ kubectl --kubeconfig=userconfig config set-context default \
   --cluster=$CLUSTER_NAME \
   --user=$USER_NAME \
   --namespace=default
Context "default" created.
$ kubectl --kubeconfig=userconfig config use-context default
Switched to context "default".
```

The admin can authorize the user to read information about nodes with these commands (more on the next section about Authorization):

```
$ kubectl create clusterrole nodes-read \
   --verb=get,list,watch --resource=nodes
clusterrole.rbac.authorization.k8s.io/nodes-read created
$ kubectl create clusterrolebinding user-nodes-read \
   --clusterrole=nodes-read --user=user
clusterrolebinding.rbac.authorization.k8s.io/user-nodes-read created
```

The admin can now send this `userconfig` file to the user.

The user will need to add information about its private key in the file, with the `kubectl config set-credentials` command:

```
$ USER_NAME=user
$ kubectl --kubeconfig=userconfig config set-credentials $USER_NAME \
   --client-key=user.key --embed-certs
```

As `user`, let's try to list the nodes and pods with this `userconfig` file:

```
$ kubectl --kubeconfig=userconfig get nodes
NAME         STATUS   ROLES    AGE   VERSION
controller   Ready    master   8h    v1.15.7
worker-0     Ready    <none>   8h    v1.15.7
worker-1     Ready    <none>   8h    v1.15.7

$ kubectl --kubeconfig=userconfig get pods
Error from server (Forbidden): pods is forbidden: User "user" cannot list resource "\
pods" in API group "" in the namespace "default"
```

The user can see nodes as expected and, hopefully, does not have the right to get pods, as only explicit given accesses are authorized.

HTTP Basic Auth

For Kubernetes API server to support HTTP Basic Authentication, you must specify the following option:

```
--basic-auth-file=somefile
```

Let's create the `/etc/kubernetes/pki/basic-auth` file with this content:

```
$ echo mypassword,pmartin,pmartin | \
   sudo tee /etc/kubernetes/pki/basic-auth
$ sudo chmod 600 /etc/kubernetes/pki/basic-auth
```

And add the option to the `/etc/kubernetes/manifests/kube-apiserver.yaml` file:

```
[...]
spec:
  containers:
  - command:
    - kube-apiserver
    - --advertise-address=10.240.0.10
    - --allow-privileged=true
    - --authorization-mode=Node,RBAC
    - --basic-auth-file=/etc/kubernetes/pki/basic-auth
[...]
```

Verify that the API server restarted to get the changes into account, by looking at ths `AGE` information:

```
$ kubectl get pods -n kube-system kube-apiserver-controller
NAME                        READY   STATUS    RESTARTS   AGE
kube-apiserver-controller   1/1     Running   3          15s
```

Now the user `pmartin` is registered in the API server, let's create a kubeconfig file for this user.

There is three parts in the kubeconfig files: server information, user information and context information.

The command `kubectl config set-cluster` is used to write server information:

> Note in the following commands the presence of the flag `--kubeconfig=userconfig`. When ths flag is set, the `kubectl` tool will work on this file, instead of on the default one `~/.kube/config`.

```
$ CLUSTER_ENDPOINT=$(kubectl config view --minify --raw -o jsonpath='{.clusters[0].c\
luster.server}')
$ CLUSTER_CA=$(kubectl config view --minify --raw -o jsonpath='{.clusters[0].cluster\
.certificate-authority-data}')
$ echo -n $CLUSTER_CA | base64 -d > cluster.crt
$ CLUSTER_NAME=$(kubectl config view --minify --raw -o jsonpath='{.clusters[0].name}\
')
$ kubectl --kubeconfig=userconfig config set-cluster $CLUSTER_NAME \
   --server=$CLUSTER_ENDPOINT \
   --certificate-authority=cluster.crt --embed-certs
Cluster "cka" set.
```

The command `kubectl config set-credentials` is used to write user information:

```
$ USER_NAME=pmartin
$ kubectl --kubeconfig=userconfig config set-credentials $USER_NAME \
   --username=pmartin --password=mypassword
User "pmartin" set.
```

The command `kubectl config set-context` is used to write context information and `kubectl config use-context` to select the current context:

```
$ kubectl --kubeconfig=userconfig config set-context default \
   --cluster=$CLUSTER_NAME \
   --user=$USER_NAME \
   --namespace=default
Context "default" created.
$ kubectl --kubeconfig=userconfig config use-context default
Switched to context "default".
```

The admin can authorize the user to read information about nodes with these commands (more on the next section about Authorization):

```
$ kubectl create clusterrole nodes-read \
   --verb=get,list,watch --resource=nodes
clusterrole.rbac.authorization.k8s.io/nodes-read created
$ kubectl create clusterrolebinding pmartin-nodes-read \
   --clusterrole=nodes-read --user=pmartin
clusterrolebinding.rbac.authorization.k8s.io/pmartin-nodes-read created
```

The admin can now send this `userconfig` file to the user.

As `pmartin`, let's try to list the nodes and pods with this `userconfig` file:

```
$ kubectl --kubeconfig=userconfig get nodes
NAME         STATUS   ROLES    AGE   VERSION
controller   Ready    master   8h    v1.15.7
worker-0     Ready    <none>   8h    v1.15.7
worker-1     Ready    <none>   8h    v1.15.7

$ kubectl --kubeconfig=userconfig get pods
Error from server (Forbidden): pods is forbidden: User "pmartin" cannot list resourc\
e "pods" in API group "" in the namespace "default"
```

The user can see nodes as expected and, hopefully, does not have the right to get pods, as only explicit given accesses are authorized.

Bearer Token Authentication

For Kubernetes API server to support Bearer Token Authentication, you must specify the following option:

```
--token-auth-file=somefile
```

Let's create the `/etc/kubernetes/pki/tokens` file with this content:

```
$ echo 22C1192A24CE822DDB2CB578BBBD8,foobar,foobar | \
   sudo tee /etc/kubernetes/pki/tokens
$ sudo chmod 600 /etc/kubernetes/pki/tokens
```

And add the option to the `/etc/kubernetes/manifests/kube-apiserver.yaml` file:

```
[...]
spec:
  containers:
  - command:
    - kube-apiserver
    - --advertise-address=10.240.0.10
    - --allow-privileged=true
    - --authorization-mode=Node,RBAC
    - --token-auth-file=/etc/kubernetes/pki/tokens
[...]
```

Verify that the API server restarted to get the changes into account, by looking at ths `AGE` information:

```
$ kubectl get pods -n kube-system kube-apiserver-controller
NAME                        READY   STATUS    RESTARTS   AGE
kube-apiserver-controller   1/1     Running   3          15s
```

Now the user `foobar` is registered in the API server, let's create a kubeconfig file for this user.

There is three parts in the kubeconfig files: server information, user information and context information.

The command `kubectl config set-cluster` is used to write server information:

> Note in the following commands the presence of the flag `--kubeconfig=userconfig`. When ths flag is set, the `kubectl` tool will work on this file, instead of on the default one `~/.kube/config`.

```
$ CLUSTER_ENDPOINT=$(kubectl config view --minify --raw -o jsonpath='{.clusters[0].c\
luster.server}')
$ CLUSTER_CA=$(kubectl config view --minify --raw -o jsonpath='{.clusters[0].cluster\
.certificate-authority-data}')
$ echo -n $CLUSTER_CA | base64 -d > cluster.crt
$ CLUSTER_NAME=$(kubectl config view --minify --raw -o jsonpath='{.clusters[0].name}\
')
$ kubectl --kubeconfig=userconfig config set-cluster $CLUSTER_NAME \
   --server=$CLUSTER_ENDPOINT \
   --certificate-authority=cluster.crt --embed-certs
Cluster "cka" set.
```

The command `kubectl config set-credentials` is used to write user information:

```
$ USER_NAME=foobar
$ kubectl --kubeconfig=userconfig config set-credentials $USER_NAME \
   --token=22C1192A24CE822DDB2CB578BBBD8
User "foobar" set.
```

The command `kubectl config set-context` is used to write context information and `kubectl config use-context` to select the current context:

```
$ kubectl --kubeconfig=userconfig config set-context default \
   --cluster=$CLUSTER_NAME \
   --user=$USER_NAME \
   --namespace=default
Context "default" created.
$ kubectl --kubeconfig=userconfig config use-context default
Switched to context "default".
```

The admin can authorize the user to read information about nodes with these commands (more on the next section about Authorization):

```
$ kubectl create clusterrole nodes-read \
   --verb=get,list,watch --resource=nodes
clusterrole.rbac.authorization.k8s.io/nodes-read created
$ kubectl create clusterrolebinding foobar-nodes-read \
   --clusterrole=nodes-read --user=foobar
clusterrolebinding.rbac.authorization.k8s.io/foobar-nodes-read created
```

The admin can now send this `userconfig` file to the user.

As `foobar`, let's try to list the nodes and pods with this `userconfig` file:

```
$ kubectl --kubeconfig=userconfig get nodes
NAME         STATUS   ROLES    AGE   VERSION
controller   Ready    master   8h    v1.15.7
worker-0     Ready    <none>   8h    v1.15.7
worker-1     Ready    <none>   8h    v1.15.7

$ kubectl --kubeconfig=userconfig get pods
Error from server (Forbidden): pods is forbidden: User "foobar" cannot list resource\
 "pods" in API group "" in the namespace "default"
```

The user can see nodes as expected and, hopefully, does not have the right to get pods, as only explicit given accesses are authorized.

Service account authentication

In contrast to normal users, service accounts are managed by the Kubernetes API. The authentication is handled by JWT Tokens.

When a `ServiceAccount` is created in a namespace, the *Token Controller* creates a `Secret` in the same namespace and with a name prefixed by the service account name, and populates it with the public CA of the API server, a signed token and the name of the current namespace.

When a namespace is created, the *Service Account controller* creates a `default` service account in this namespace. This creation in turn leads to the creation of the associated `Secret`.

A `serviceAccountName` field of the Pod Spec indicates which service account is attached to the Pod. By default, if you do not specify any service account name, its value is `default`. You can set the `automountServiceAccountToken` field of the Pod Spec to `false` to indicate that no service account should be used.

The `Secret` associated with the Pod's service account is automatically mounted into the Pod filesystem, in a well-known directory. The Kubernetes clients inside the Pod are aware of this path and use these credentials to connect to the API server.

As an example, let's create a service account and use it for a container containing the `kubectl` command, to test the accesses from inside the container:

```
# Create a service account for a kubectl pod
$ kubectl create serviceaccount kubectl
serviceaccount/kubectl created

# Get the name of the associated secret
$ SECRET_NAME=$(kubectl get sa kubectl -o jsonpath='{.secrets[0].name}')

# Show the secret contents
$ kubectl get secret $SECRET_NAME -o yaml
[...]

# Create the nodes-read cluster role
$ kubectl create clusterrole nodes-read \
   --verb=get,list,watch --resource=nodes
clusterrole.rbac.authorization.k8s.io/nodes-read created

# Bind the nodes-read role to the service account kubectl in default namespace
$ kubectl create clusterrolebinding default-kubectl-nodes-read \
   --clusterrole=nodes-read --serviceaccount=default:kubectl
clusterrolebinding.rbac.authorization.k8s.io/default-kubectl-nodes-read created

# Execute kubectl container with kubectl service account
$ kubectl run --generator=run-pod/v1 kubectl \
   --image=bitnami/kubectl:latest \
   --serviceaccount=kubectl \
   --command sh -- -c "sleep $((10**10))"
pod/kubectl created

# Connect into the kubectl container
$ kubectl exec -it kubectl bash

# Get nodes
/$ kubectl get nodes
NAME         STATUS   ROLES    AGE   VERSION
controller   Ready    master   10h   v1.15.7
worker-0     Ready    <none>   10h   v1.15.7
worker-1     Ready    <none>   10h   v1.15.7

# Try to get pods
/$ kubectl get pods
Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:defaul\
t:kubectl" cannot list resource "pods" in API group "" in the namespace "default"
```

```
# Show the mounted files from service account secret
/$ ls /var/run/secrets/kubernetes.io/serviceaccount/
ca.crt  namespace  token
```

Service account outside the cluster

Note that the token associated with a service account is also usable from outside the cluster. You can for example create a kubeconfig file containing this token and use it from your dev machine:

```
$ CLUSTER_ENDPOINT=$(kubectl config view --minify --raw -o jsonpath='{.clusters[0].c\
luster.server}')
$ CLUSTER_CA=$(kubectl config view --minify --raw -o jsonpath='{.clusters[0].cluster\
.certificate-authority-data}')
$ echo -n $CLUSTER_CA | base64 -d > cluster.crt
$ CLUSTER_NAME=$(kubectl config view --minify --raw -o jsonpath='{.clusters[0].name}\
')
$ kubectl --kubeconfig=saconfig config set-cluster $CLUSTER_NAME \
   --server=$CLUSTER_ENDPOINT \
   --certificate-authority=cluster.crt --embed-certs
Cluster "cka" set.

$ USER_NAME=kubectl
$ SECRET_NAME=$(kubectl get sa kubectl -o jsonpath='{.secrets[0].name}')
$ TOKEN=$(kubectl get secrets $SECRET_NAME -o jsonpath='{.data.token}' | base64 -d)
$ kubectl --kubeconfig=saconfig config set-credentials $USER_NAME \
   --token=$TOKEN
User "kubectl" set.

$ kubectl --kubeconfig=saconfig config set-context default \
   --cluster=$CLUSTER_NAME \
   --user=$USER_NAME \
   --namespace=default
Context "default" created.
$ kubectl --kubeconfig=saconfig config use-context default
Switched to context "default".

# List the nodes
$ kubectl --kubeconfig=saconfig get nodes
NAME         STATUS   ROLES    AGE   VERSION
controller   Ready    master   10h   v1.15.7
worker-0     Ready    <none>   10h   v1.15.7
worker-1     Ready    <none>   10h   v1.15.7
```

```
# Try to list the pods
$ kubectl --kubeconfig=saconfig get pods
Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:defaul\
t:kubectl" cannot list resource "pods" in API group "" in the namespace "default"
```

Authorization

The Kubernetes API is a REST API. The authorization for an operation is evaluated at the request level. The user have to be granted access to all parts of the request against all policies to be authorized for this request. Admission controllers can be used to fine-tune the parts of a request authorized for a user.

Authorization is managed by modules and several modules can be installed at the same time. Each module is checked in sequence and the first allowing or denying the request stops the process of authorization. If no module have an opinion on the request, then the request is denied.

The following modules are available:

- ABAC: Attribute-Based Access Control,
- RBAC: Role-based access control,
- Webhook: HTTP callback mode,
- Node: special-purpose module for kubelets,
- AlwaysDeny: blocks all requests, for testing purpose,
- AlwaysAllow: to completely disable authorization.

To select the modules to use, you have to specify their names in the `--authorization-mode` flag of the `apiserver` service. For a cluster installed with kubeadm, the default value of the flag is `--authorization-mode=Node,RBAC`.

Anatomy of an API server request

Resource Requests

Resource requests are used to operate on Kubernetes resources.

The endpoints of Resources requests are of the form:

- `/api/v1/...` for the resources of the core group and
- `/apis/<group>/<version>/...` for resources of other apis.

Endpoints for **Namespaced Resources** are of the form:

- `/api/v1/namespaces/<namespace>/<resource>` and

- `/apis/<group>/<version>/namespaces/<namespace>/<resource>`.

Endpoints for **Non-namespaced Resources** are of the form:

- `/api/v1/<resource>` and
- `/apis/<group>/<version>/<resource>`.

For Resource Requests, an **API request Verb** is used. Common verbs are:

- `create` to create a new resource object,
- `update` to replace a resource object with a new one,
- `patch` to change specific fields of a resource object,
- `get` to retrieve a resource object,
- `list` to retrieve all resource objects within one namespace or across all namespaces,
- `watch` to stream events (create, update, delete) on object(s),
- `delete` to delete a resource object,
- `deletecollection` to delete all resource objects within one namespace.

You can obtain the list of all the resources with their api group, their supported verbs and whether they are namespaced or not with the command `kubectl api-resources -o wide`.

Non-resource Requests

Non-resource requests are all the other requests and used to access information about the cluster.

For non-resource requests, an **HTTP request Verb** is used, corresponding to lowercased HTTP methods, like `get`, `post`, `put`, and `delete`.

Request Attributes for Authorization

The attributes taken into account for authorization are:

- attributes from authentication
  - `user`: the authenticated user,
  - `group`: list of group names to which the user belongs,
  - `extra`: key-value pairs provided by the authentication layer,
- attributes from request:
  - for a resource request:
    - api group (empty for *core* group),
    - namespace (for namespaced-resource requests only),
    - resource,
    - resource name (mandatory for `get`, `update`, `patch`, and `delete` verbs),
    - subresource name (optional),
    - API request verb,
  - for a non-resource request:
    - request path,
    - HTTP request verb.

RBAC Mode

The RBAC mode introduces two concepts: the **Role**, which defines a list of permissions, and the **Role Binding**, which binds a Role to a user or a group of users (normal users, groups or service accounts).

Role and ClusterRole

Two resources are defined, depending on whether the role is defined within a namespace with `Role` or cluster-wide with `ClusterRole`.

The Role and ClusterRole Spec contain a `rules` field, an array of `PolicyRule` structure containing these subfields:

- `verbs`: list of allowed verbs, or `VerbAll` (or `"*"` in yaml) to bypass verification on this field,
- `apiGroups`: list of allowed api groups, or `APIGroupAll` (or `"*"` in yaml) to bypass verification on this field,
- `resources`: list of allowed resources, or `ResourceAll` (or `"*"` in yaml) to bypass verification on this field,
- `resourceNames`: list of allowed objects, or empty to bypass verification on this field,
- `nonResourceURLs`: list of allowed non-resource URLs (for ClusterRole only), or empty to bypass verification on this field.

Each PolicyRule represents a set of permissions to grant for this role.

For a request to be allowed for a role, the Verb, API group, Resource, Resource name and non-resource URL attributes (if applicable) of the request must be present in the corresponding field in any policy rule of the role when the verification is active for this field.

A role is used to grant access to namespaced resources in the namespace the role is defined.

A cluster role is used to grant access to:

- namespaced resources (like pods) in any namespace,
- namespaced resources (like pods) across all namespaces (using –all-namespaces flag),
- non-namespaced resources (like nodes),
- non-resource endpoints (like `/healthz`).

RoleBinding and ClusterRoleBinding

`RoleBinding` and `ClusterRoleBinding` Specs both contain a `roleRef` field (a `RoleRef` structure) and a `subjects` field (an array of `Subject` structure).

These role bindings grant the referenced role to the specified subjects.

The `roleRef` field references:

- for a ClusterRoleBinding: a ClusterRole.
- for a RoleBinding: a ClusterRole or a Role in the same namespace

The `RoleRef` structure is composed of the fields:

- apiGroup: must be `rbac.authorization.k8s.io`
- kind: must be `Role` or `ClusterRole`
- name: the name of Role or ClusterRole

The `Subject` structure is composed of the fields:

- apiGroup: `""` for ServiceAccount, `rbac.authorization.k8s.io` for `User` and `Group`
- kind: must be `User`, `Group` or `ServiceAccount`
- name: name of the `User`/`Group`/`ServiceAccount` subject
- namespace: namespace of subject, for `ServiceAccount`

The different possible bindings are:

- a `RoleBinding` referencing a `Role`: gives access to namespaced resources in the namespace of the role and role binding
- a `RoleBinding` referencing a `ClusterRole`: gives access to namespaced resources in the namespace of the role binding (used to reuse a ClusterRole in different namespaces with different subjects)
- a `ClusterRoleBinding` referencing a `ClusterRole`: gives access to namespaced resources in all namespaces and across all namespaces, to non-namespaced resources and to non-resource endpoints.

Examples

Here are some examples of Role definitions and their signifitcations:

The `role-read` role allows the user to get and list all namespaced resources in the `default` namespace:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: role-read
  namespace: default
rules:
- apiGroups:
  - "*"
  resources:
  - "*"
  verbs:
  - "get"
  - "list"

---

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: role-read
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: role-read
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: user
```

You can also create these resources in imperative mode:

```
$ kubectl create role role-read --verb=get,list --resource="*.*"
role.rbac.authorization.k8s.io/role-read created
$ kubectl create rolebinding role-read --role=role-read --user=user
rolebinding.rbac.authorization.k8s.io/role-read created
```

These requests will be authorized:

```
$ export KUBECONFIG=userconfig
$ kubectl get pods kubectl # get core/pods
$ kubectl get pods # list core/pods
$ kubectl get deployments # list extensions/deployments
```

These requests will not be authorized:

```
$ export KUBECONFIG=userconfig
$ kubectl get namespaces # namespaces are cluster-scope
$ kubectl delete pods kubectl # delete not in verbs
$ kubectl get pods -n kube-system # not in default namespace
```

You can now delete the role and role binding:

```
$ unset KUBECONFIG
$ kubectl delete rolebinding role-read
rolebinding.rbac.authorization.k8s.io "role-read" deleted
$ kubectl delete role role-read
role.rbac.authorization.k8s.io "role-read" deleted
```

The `role-create-pod` role allows the user to create pods in the `default` namespace:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: role-create-pod
rules:
- apiGroups:
  - ""
  resources:
  - "pods"
  verbs:
  - "create"

---

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: role-create-pod
roleRef:
  apiGroup: rbac.authorization.k8s.io
```

```
  kind: Role
  name: role-create-pod
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: user
```

You can also create these resources in imperative mode:

```
$ kubectl create role role-create-pod --resource=pods --verb=create
role.rbac.authorization.k8s.io/role-create-pod created
$ kubectl create rolebinding role-create-pod --role=role-create-pod --user=user
rolebinding.rbac.authorization.k8s.io/role-create-pod created
```

This request will be authorized:

```
$ export KUBECONFIG=userconfig
$ kubectl run --generator=run-pod/v1 nginx --image=nginx # create core/pods
```

These requests will not be authorized:

```
$ export KUBECONFIG=userconfig
$ kubectl get pods # list verb
$ kubectl get pods nginx # get verb
$ kubectl create deployment nginx --image=nginx # extensions/deployments
$ kubectl run --generator=run-pod/v1 nginx --image=nginx -n other # other namespace
```

You can now delete the role and role binding:

```
$ unset KUBECONFIG
$ kubectl delete rolebinding role-create-pod
rolebinding.rbac.authorization.k8s.io "role-create-pod" deleted
$ kubectl delete role role-create-pod
role.rbac.authorization.k8s.io "role-create-pod" deleted
```

The `cluster-role-read` role allows the user to get and list all namespaced resources in and across all namespaces, as well as all non-namespaced resources:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-role-read
rules:
- apiGroups:
  - "*"
  resources:
  - "*"
  verbs:
  - "get"
  - "list"

---

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-role-read
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-role-read
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: user
```

You can also create these resources in imperative mode:

```
$ kubectl create clusterrole cluster-role-read --resource="*.*" --verb=get,list
clusterrole.rbac.authorization.k8s.io/cluster-role-read created
$ kubectl create clusterrolebinding cluster-role-read --clusterrole=cluster-role-rea\
d --user=user
clusterrolebinding.rbac.authorization.k8s.io/cluster-role-read created
```

These requests will be authorized:

```
$ export KUBECONFIG=userconfig
$ kubectl get pods kubectl # get core/pods
$ kubectl get pods # list core/pods
$ kubectl get pods -n kube-system # list core/pods in other namespace
$ kubectl get pods -A # list core/pods across all namespaces
$ kubectl get deployments # list extensions/deployments
$ kubectl get nodes # list (non-namespaced) nodes
```

These requests will not be authorized:

```
$ export KUBECONFIG=userconfig
$ kubectl delete pods kubectl # delete not in verbs
```

You can now delete the role and role binding:

```
$ unset KUBECONFIG
$ kubectl delete rolebinding cluster-role-read
rolebinding.rbac.authorization.k8s.io "cluster-role-read" deleted
$ kubectl delete role cluster-role-read
role.rbac.authorization.k8s.io "cluster-role-read" deleted
```

Security Contexts

You can configure security contexts at Pod and Container levels.

At Pod level

The Pod Spec defines several fields in its `PodSecurityContext` structure, accessible in the `securityContext` field.

User and Groups

By default, the processes inside the containers of a Pod run with the `root` rights. Thanks to the container isolation, the `root` rights inside the container are limited.

But in some circumstances, for example when mounting external filesystems inside the container, you would like that the processes run with specific user and group rights.

The `runAsNonRoot` field helps to ensure that the processes in containers of the Pod are running as non-root user. If no user is defined in the container image definition and if no `runAsUSer` is defined here or in the `SecurityContext` at container level, kubelet will refuse to start the Pod.

With the `runAsUser`, `runAsGroup` and `supplementalGroups` fields, you can affect the first process of the containers of the Pod to a specific user and a specific group, and add these processes to supplemental groups.

As an example, when running a Pod with the following specs:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: box
  name: box
spec:
  selector:
    matchLabels:
      app: box
  template:
    metadata:
      labels:
        app: box
    spec:
      securityContext:
        runAsNonRoot: true
        runAsUser: 1000
        runAsGroup: 1001
        supplementalGroups:
        - 1002
        - 1003
      containers:
      - image: busybox
        name: box
        command:
         - sh
         - -c
         - "touch /tmp/ready && sleep $((10**10))"
```

You can examine the users and groups for processes and created files:

```
$ ps -o pid,user,group,comm
PID   USER     GROUP    COMMAND
    1 1000     1001     sleep
    7 1000     1001     sh
   14 1000     1001     ps
$  ls -l /tmp/ready
-rw-r--r--  1 1000  1001  0 Jan 23 09:52 /tmp/ready
$ id
uid=1000 gid=1001 groups=1002,1003
```

SELinux options

`seLinuxOptions` applies a SELinux context to all containers of the Pod.

Sysctls

If you want to set kernel parameters from inside a container, you will get the following error:

```
$ sysctl -w kernel.shm_rmid_forced=1
sysctl: error setting key 'kernel.shm_rmid_forced': Read-only file system
```

You can pass these values from the Pod Spec:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: box
  name: box
spec:
  selector:
    matchLabels:
      app: box
  template:
    metadata:
      labels:
        app: box
    spec:
      securityContext:
        sysctls:
        - name: kernel.shm_rmid_forced
          value: "1"
      containers:
      - image: busybox
        name: box
```

Note that only *safe* sysctls are allowed to be changed by default. If you want to change other sysctls, you will need to allow them via the *kubelet* configuration (using `kubelet --allowed-unsafe-sysctls 'comma-separated list of sysctls'`).

At Container level

The Container Spec defines several fields in its `SecurityContext` structure, accessible in the `securityContext` field.

User and Groups

As within the Pod spec, you can assert that the image is running as non root user with the `runAsNonRoot` field, and you can specify specific user and group with `runAsGroup` and `runAsUser` for the first process of the container.

If specified at both Pod and Container level, the information in the Container spec is used.

SELinux options

`seLinuxOptions` applies a SELinux context to this container. If specified at both Pod and Container level, the information in the Container spec is used.

Capabilities

- `capabilities`:
  modifies initial capabilities for the container initial process,
- `allowPrivilegeEscalation`:
  indicates if the processes can get more capabilities at runtime.

Others

- `privileged`:
  this is equivalent as running as root in the host. **Don't do this unless you exactly know what you are doing**,
- `readOnlyRootFilesystem`: the processes won't be able to change or create files into the container filesystem. This prevents attackers from installing new programs in the container for example.

Network policies

By default, traffic between Pods of a cluster is unrestricted. You can fine-tune the traffic authorization between the Pods by declaring network policies, using the `NetworkPolicy` resource.

The spec of the `NetworkPolicy` resource contains the fields:

- `podSelector`: selects Pods to which this policy applies. An empty value matches all pods in the namespace,
- `policyTypes`: indicates if you want to apply `Ingress` rules, `Egress` rules or both,
- `ingress`: the allow ingress rules for the selected pods,
- `egress`: the allow egress rules for the selected pods.

First create three pods (an Apache server, an nginx server and a busybox Pod) and two services to expose the two web servers:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: box-deployment
spec:
  selector:
    matchLabels:
      app: box
  template:
    metadata:
      labels:
        app: box
    spec:
      containers:
      - name: box
        image: busybox
        command:
        - sh
        - -c
        - "sleep $((10**10))"

---

apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpd-deployment
spec:
  selector:
    matchLabels:
      app: httpd
  template:
    metadata:
      labels:
        app: httpd
    spec:
      containers:
      - name: httpd
        image: httpd

---

apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx

---

apiVersion: v1
kind: Service
metadata:
  name: httpd-service
spec:
  type: ClusterIP
  selector:
    app: httpd
  ports:
  - port: 80

---

apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: ClusterIP
  selector:
    app: nginx
  ports:
  - port: 80
```

You can see that from the `box` container, you can access both `apache` and `nginx`:

```
$ kubectl exec -it box-deployment-xxxxxxxxxx-yyyyy sh
# wget -q http://httpd-service -O -
[... apache response ...]
# wget -q http://nginx-service -O -
[... nginx response ...]
```

With a first `NetworkPolicy`, you can disallow all ingress traffic to the Pods:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: netpol
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

You can see you cannot connect anymore to `apache` and `nginx` pods from the `box` pod:

```
$ kubectl exec -it box-deployment-xxxxxxxxxx-yyyyy sh
# wget -q http://httpd-service -O -
<no reply>
# wget -q http://nginx-service -O -
<no reply>
```

You can now allow traffic to port `80` of `nginx` from the `box` pods:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: netpol2
spec:
  podSelector:
    matchLabels:
      app: nginx
  policyTypes:
    - Ingress
  ingress:
  - ports:
    - port: 80
    from:
    - podSelector:
        matchLabels:
          app: box
```

Working with private Docker registries

Up to now, you have referenced public images to deploy containers. But when you will want to deploy your own applications, it is possible that you do not want to make your container images public, but store them in a private registry.

In this case, kubelet will need to get the necessary credentials to be able to download the images stored in the private registry.

Using imagePullSecrets

This is the recommended way to give access to the registry when you do not have access to the nodes or when nodes are created automatically.

The first step is to create a `Secret` containing the registry credentials.

If you already have logged in or can log in to the registry from your computer with the command `docker login`, you should have a file `~/.docker/config.json`. You can then create a `Secret` from this file with the command:

```
$ kubectl create secret generic regcred \
    --from-file=.dockerconfigjson=<path/to/.docker/config.json> \
    --type=kubernetes.io/dockerconfigjson
```

Or, you can create a `Secret` from the login information:

```
$ kubectl create secret docker-registry regcred \
   --docker-server=$DOCKER_REGISTRY_SERVER \
   --docker-username=$DOCKER_USER \
   --docker-password=$DOCKER_PASSWORD \
   --docker-email=$DOCKER_EMAIL
```

Once the secret is created in the namespace, you can reference it from Pod Specs with the `imagePullSecrets` field (note that you could reference several Secrets, if your Pod contains several containers, getting images from different registries):

```
# box.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: box
  name: box
spec:
  selector:
    matchLabels:
      app: box
  template:
    metadata:
      labels:
        app: box
    spec:
      imagePullSecrets:
      - name: regcred
      containers:
      - image: yourlogin/test
        name: box
```

An important thing to know is that once the image is downloaded by a node, all Pods executing in this same node will be allowed to use this image, even if they do not specify an `imagePullSecrets`.

To test it, first deploy the pod below on a multi-worker cluster, and see on which node it is deployed:

```
$ kubectl apply -f box.yaml
deployment.apps/box created
$ kubectl get pods -o wide
NAME                   READY   STATUS    [...]  NODE
box-865486655c-c76sj   1/1     Running   [...]  worker-0
```

In this case, the image has been downloaded by `worker-0`. Now update the deployment to remove the `imagePullSecrets` and deploy several replicas, so they will be deployed on other nodes too. Also set the `imagePullPolicy` to `IfNotPresent`, so kubelet will use an already present image if available:

```
$ kubectl delete -f box.yaml
deployment.apps "box" deleted
$ kubectl apply -f - <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: box
  name: box
spec:
  replicas: 2
  selector:
    matchLabels:
      app: box
  template:
    metadata:
      labels:
        app: box
    spec:
      containers:
      - image: yourlogin/test
        name: box
        imagePullPolicy: IfNotPresent
EOF
deployment.apps/box created
$ kubectl get pods -o wide
NAME                   READY   STATUS         AGE [...] NODE
box-865486655c-n4pg6   0/1     ErrImagePull   5s [...] worker-1
box-865486655c-w55kp   1/1     Running        5s [...] worker-0
```

You can see that the Pod in `worker-0` started succesfully, but the one in the other worker failed to get the image.

Pre-pulling images on nodes

You have seen it is only necessary that an image is pulled on the node to be available for Pods. So, let's connect to the `worker-1` node, connect to the Docker registry, then manually pull the image:

```
$ gcloud compute ssh worker-1
Welcome to worker-1
$ sudo docker login
Username: yourlogin
Password:
$ docker pull yourlogin/test
docker.io/yourlogin/test:latest
```

If you go back to your computer, you can see that the Pod in `worker-1` can now be started:

```
$ kubectl get pods -o wide
NAME                   READY   STATUS    [...]  NODE
box-865486655c-2t6fw   1/1     Running   [...]  worker-1
box-865486655c-nnpr2   1/1     Running   [...]  worker-0
```

Giving credentials to kubelet

In the previous step, you logged in the private registry to manually download the image. During login, docker created a `~/.docker/config.json` to store the credentials used during login.

You can copy this file in a directory recognized by kubelet, so kubelet will try to download images using these credentials:

```
$ gcloud compute ssh worker-1
Welcome to worker-1
$ cp $HOME/.docker/config.json /var/lib/kubelet/
$ sudo systemctl kubelet restart
```

Storage

Persistent Volumes

A persistent volume (**PV**) is a storage resource provisioned by the cluster administrators. The provisioning can be manual or automatic.

A PV declaration is made of two parts:

- its capabilities (access modes, capacity, storage class, volume mode - filesystem or raw),
- its implementation (local, NFS, cloud storage resource, etc).

These storage resources are intended to be used by Pods, through the use of `PersistentVolumeClaim`'s: a Pod will claim a persistent volume with specific capabilities, and Kubernetes will try to find a Persistent volume marching these capabilities (independently of the implementation details).

If available, the Persistent Volume will be mounted into the filesystem of the node deploying the Pod, and finally exposed to the Pod. The implentation part indicates to kubelet how to mount the storage into a filesystem.

Creating an NFS Persistent Volume

As an example, we will manually create a PV implemented by an NFS volume.

First create an NFS volume in Google Cloud:

```
$ gcloud filestore instances create nfs-server \
   --project=$(gcloud config get-value project) \
   --zone=$(gcloud config get-value compute/zone) \
   --tier=STANDARD \
   --file-share=name="vol1",capacity=1TB \
   --network=name="kubernetes-cluster"
Waiting for [operation-...] to finish...done.
$ gcloud filestore instances describe nfs-server \
   --project=$(gcloud config get-value project) \
   --zone=$(gcloud config get-value compute/zone)
createTime: '2020-01-24T07:43:58.279881289Z'
fileShares:
- capacityGb: '1024'
```

```
  name: vol1
name: projects/yourproject/locations/us-west1-c/instances/nfs-server
networks:
- ipAddresses:
  - 172.25.52.106    # Note this IP address
  network: kubernetes-cluster
  reservedIpRange: 172.25.52.104/29
state: READY
tier: STANDARD
```

On the workers, install the NFS drivers to be able to mount NFS filesystems:

Repeat these steps for each worker:

```
$ gcloud compute ssh worker-0
Welcome to worker-0
$ sudo apt-get -y update
$ sudo apt-get -y install nfs-common
```

You can test that the worker can mount the filesystem:

```
$ gcloud compute ssh worker-0
Welcome to worker-0
$ sudo mkdir /mnt/nfs
$ sudo mount 172.25.52.106:/vol1 /mnt/nfs
$ ls /mnt/nfs
lost+found
$ sudo umount /mnt/nfs
```

You can now define the NFS persistent volume:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-nfs
spec:
  # capabilities
  accessModes:
  - ReadWriteOnce
  - ReadOnlyMany
  - ReadWriteMany
  capacity:
    storage: 1Ti
```

```
  volumeMode: Filesystem
  # implementation
  nfs:
    path: /vol1
    server: 172.25.52.106
```

Access modes

In a general way, a storage system can be accessed for read and write operations, and can or cannot be accessed by several clients for each of these operations simultaneously, depending on the technology used by the storage system.

The `PersistentVolume accessModes` field indicates which are the capabilities of the underlying storage system in term of simultaneous access on read only or read/write modes. Three values are defined:

ReadWriteOnce (RWO)
: the storage is accessible for read and write operations by a single client,

ReadOnlyMany (ROX)
: the storage is accessible for read only operations, by several clients,

ReadWriteMany (RWX)
: the storage is accessible for read and write operations, by several clients.

If the PV has a **Many** capability, several pods will be able to use this PV at the same time. Note that for a PV to be used by several pods, the access mode claimed must be the same by all the pods: it is not possible that one Pod uses a `ReadOnlyMany` mode and another pod uses `ReadWriteMany` on the same PV.

Claiming a Persistent volume

When a Pod needs a Persistent volume, it must **claim** one. It does not claim a particular persistent volume, but rather claims a list of capabilities. The persistent volume controller will affect the best possible persistent volume depending on the capabilities matching.

The `PersistentVolumeClaim` resource is used, and its Spec structure defines these fields:

`accessModes`
: the access mode requested (see access modes)

`selector`
: a label selector to match specific persistent volumes based on labels,

`resources`
: the persistent volume must provide at least `resources.requests.storage` and, if defined, at most `resources.limits.storage`,

storageClassName
: the storage provider can define different storage class names, you can specify which class the persistent volume should belong to,

volumeMode
: the mode of storage: **Filesystem** or **Block**.

To continue the preceding example, you can create a claim that will match the provisioned NFS persistent volume. Precisely, a volume accessible by several clients in read-write mode, with at least 500Gi and at most 1.5Ti of storage:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pv-rwx-500g
spec:
  storageClassName: ""
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 500Gi
    limits:
      storage: 1500Gi
```

Now, you can deploy two pods, using the same storage: one database system, and one box:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  labels:
    app: pg
  name: pg
spec:
  serviceName: pg
  selector:
    matchLabels:
      app: pg
  template:
    metadata:
      labels:
        app: pg
    spec:
```

```
      containers:
      - image: postgres
        name: pg
        env:
        - name: PGDATA
          value: /var/lib/postgresql/data/db-files
        volumeMounts:
        - name: data
          mountPath: /var/lib/postgresql/data
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: pv-rwx-500g
          readOnly: false

---

apiVersion: apps/v1
kind: StatefulSet
metadata:
  labels:
    app: box
  name: box
spec:
  serviceName: box
  selector:
    matchLabels:
      app: box
  template:
    metadata:
      labels:
        app: box
    spec:
      containers:
      - image: busybox
        name: box
        volumeMounts:
        - name: data
          mountPath: /mnt
        command:
          - sh
          - -c
          - "sleep $((10**10))"
```

```
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: pv-rwx-500g
          readOnly: false
```

If you examine the logs of the database pod, using the command `kubectl logs db-0`, you can see that the system created a new database and stores its files on the directory:

```
/var/lib/postgresql/data/db-files
```

From the `box` pod, you can access the same filesystem:

```
$ kubectl exec -it box-0 sh
# cd /mnt/db-files/
/mnt/db-files # ls
PG_VERSION            pg_multixact          pg_tblspc
base                  pg_notify             pg_twophase
global                pg_replslot           pg_wal
pg_commit_ts          pg_serial             pg_xact
pg_dynshmem           pg_snapshots          postgresql.auto.conf
pg_hba.conf           pg_stat               postgresql.conf
pg_ident.conf         pg_stat_tmp           postmaster.opts
pg_logical            pg_subtrans           postmaster.pid
```

Clean-up

Persistent volumes are expensive. Remember to delete the NFS volume when you do not need it anymore:

```
$ gcloud filestore instances delete nfs-server \
   --project=$(gcloud config get-value project) \
   --zone=$(gcloud config get-value compute/zone)
```

Using auto-provisioned persistent volumes

Generally, Kubernetes engines into the cloud provide auto-provisioned persistent volumes. For example, in Google Cloud GKE, GCE persistent disks are used for auto-provisioned persistent volumes.

First deploy a Kubernetes cluster:

```
$ gcloud beta container clusters create "kluster" \
   --project $(gcloud config get-value project) \
   --zone $(gcloud config get-value compute/zone) \
   --cluster-version "1.15.7-gke.23" \
   --machine-type "g1-small" \
   --image-type "COS" \
   --disk-type "pd-standard" \
   --disk-size "30" \
   --num-nodes "1"
[...]
$ gcloud container clusters get-credentials kluster \
   --zone $(gcloud config get-value compute/zone) \
   --project $(gcloud config get-value project)
```

Now create a PVC:

```
# auto-pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: disk-rwo-10g
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi


$ kubectl apply -f auto-pvc.yaml
persistentvolumeclaim/disk-rwo-10g created
$ kubectl get pvc
NAME           STATUS   VOLUME      CAPACITY   ACCESS MODES   STORAGECLASS   AGE
disk-rwo-10g   Bound    pvc-[...]   10Gi       RWO            standard       3s
$ kubectl get pv
NAME        CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM               \
   STORAGECLASS   REASON   AGE
pvc-[...]   10Gi       RWO            Delete           Bound    default/disk-rwo-10g\
   standard               3s
```

You can see that as soon as you created a claim, a PV resource has been created.

You can now deploy a Pod using this claim:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  labels:
    app: box
  name: box
spec:
  serviceName: box
  selector:
    matchLabels:
      app: box
  template:
    metadata:
      labels:
        app: box
    spec:
      containers:
      - image: busybox
        name: box
        volumeMounts:
        - name: data
          mountPath: /mnt
        command:
          - sh
          - -c
          - "sleep $((10**10))"
      volumes:
      - name: data
        persistentVolumeClaim:
          claimName: disk-rwo-10g
          readOnly: false
```

When you do not need the persistent volume anymore, you can release it by deleting the claim:

```
$ kubectl delete pvc disk-rwo-10g
persistentvolumeclaim "disk-rwo-10g" deleted
$ kubectl get pv
No resources found in default namespace.
```

Clean up

Remove the previously deployed Kubernetes cluster:

```
$ gcloud beta container clusters delete "kluster" \
   --project $(gcloud config get-value project) \
   --zone $(gcloud config get-value compute/zone)
```

Monitoring and Logging

Basic logging

Containers have to output logs to standard output (stdout) or standard error (stderr) streams for the logs to be available within the Kubernetes infrastructure.

You can use the `kubectl logs` command to display the logs of a **particular pod**:

```
$ kubectl logs nginx
127.0.0.1 - - [01/Feb/2020:17:05:58 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.58.0\
" "-"
```

Or for a **set of pods**, selected by their labels (use `--prefix` to differentiate the pods):

```
$ kubectl logs -l app=nginx --prefix
[pod/nginx-86c57db685-cqm6c/nginx] 127.0.0.1 - - [01/Feb/2020:17:45:14 +0000] "GET /\
 HTTP/1.1" 200 612 "-" "curl/7.58.0" "-"
[pod/nginx-86c57db685-5f73p/nginx] 127.0.0.1 - - [01/Feb/2020:17:45:17 +0000] "GET /\
 HTTP/1.1" 200 612 "-" "curl/7.58.0" "-"
```

- `--follow` (`-f` for short) allows to **follow** the stream, and use Ctrl-C to stop.
- `--previous` (`-p` for short) allows to view the logs of the **previous** containers, which is useful when a container crashed and you want to see the error that made it crash.
- `--container=name` (`-c name` for short) shows the logs of a specific container, and `--all-containers` shows the logs of all the containers.
- `--timestamps` displays timestamps of logs at the beginning of lines.

Upgrading the cluster

Upgrade the controller

First install the desired version of kubeadm:

```
$ gcloud compute ssh controller
Welcome to controller
$ sudo apt update && apt-cache policy kubeadm
$ sudo apt update && \
   sudo apt-get install \
   -y --allow-change-held-packages \
   kubeadm=1.16.6-00
$ sudo apt-mark hold kubeadm
$ kubeadm version -o short
v1.16.6
```

Drain the controller node:

```
$ kubectl drain controller --ignore-daemonsets \
   --delete-local-data
node/controller evicted
```

Check the possible upgrade plans:

```
$ sudo kubeadm upgrade plan

[...]

Components that must be upgraded manually after you have upgraded the control plane \
with 'kubeadm upgrade apply':
COMPONENT   CURRENT       AVAILABLE
Kubelet     3 x v1.15.7   v1.16.6

Upgrade to the latest stable version:

COMPONENT            CURRENT          AVAILABLE
API Server           v1.15.9-beta.0   v1.16.6
```

```
Controller Manager   v1.15.9-beta.0   v1.16.6
Scheduler            v1.15.9-beta.0   v1.16.6
Kube Proxy           v1.15.9-beta.0   v1.16.6
CoreDNS              1.3.1            1.6.2
Etcd                 3.3.10           3.3.15-0

You can now apply the upgrade by executing the following command:

        kubeadm upgrade apply v1.16.6
```

Several possibilities are displayed: one to upgrade to the latest version of the series and another one to upgrade to the latest stable version available with the version of `kubeadm` used.

In the screen above, is displayed only the plan to upgrade to the latest stable version.

Let's start the upgrade:

```
$ sudo kubeadm upgrade apply v1.16.6
[...]
[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.16.6". Enjoy!
```

Make the node schedulable again:

```
$ kubectl uncordon controller
node/controller uncordoned
```

Upgrade `kubelet` and `kubectl` on the controller:

```
$ sudo apt-get update && \
   sudo apt-get install \
   -y --allow-change-held-packages \
   kubelet=1.16.6-00 kubectl=1.16.6-00
$ sudo apt-mark hold kubelet kubectl
```

At this point, the controller node should show the latest version:

```
$ kubectl get nodes
NAME         STATUS   ROLES    AGE   VERSION
controller   Ready    master   13d   v1.16.6
worker-0     Ready    <none>   13d   v1.15.7
worker-1     Ready    <none>   13d   v1.15.7
```

Upgrade the workers

> Repeat these steps for each worker.

First drain the node, from your machine:

```
$ kubectl drain worker-0 --ignore-daemonsets
node/worker-0 cordoned
node/worker-0 drained
```

Then, connect to the node and install the desired version of kubeadm:

```
$ gcloud compute ssh worker-0
Welcome to worker-0
$ sudo apt update && \
   sudo apt-get install \
   -y --allow-change-held-packages \
   kubeadm=1.16.6-00
$ sudo apt-mark hold kubeadm
$ kubeadm version -o short
v1.16.6
```

Upgrade the node:

```
$ sudo kubeadm upgrade node
[...]
[upgrade] The configuration for this node was successfully updated!
```

Upgrade `kubelet` and `kubectl` on the node:

```
$ sudo apt-get update && \
   sudo apt-get install \
   -y --allow-change-held-packages \
   kubelet=1.16.6-00 kubectl=1.16.6-00
$ sudo apt-mark hold kubelet kubectl
```

From your machine, make the node schedulable again:

```
$ kubectl uncordon worker-0
node/worker-0 uncordoned
```

After upgrading all worker nodes, you should obtain the latest release on each node of the cluster:

```
$ kubectl get nodes
NAME         STATUS   ROLES    AGE   VERSION
controller   Ready    master   13d   v1.16.6
worker-0     Ready    <none>   13d   v1.16.6
worker-1     Ready    <none>   13d   v1.16.6
```

Upgrading the operating system

If you need to reboot the host of a cluster node for maintenance (for example for a kernel or hardware upgrade), you first need to **drain** the node.

```
$ kubectl drain $NODENAME --ignore-daemonsets
node/nodename drained
```

Draining the node will have two effects:

- evict all pods from this node, all pods controlled by a replicaset being re-scheduled on another node,
- make this node unschedulable, so that no new pod is scheduled on this node during the maintenance.

You can now safely make maintenance operations on the operating system or hardware.

Once the maintenance is over, you can **uncordon** the node, to make the node schedulable again:

```
$ kubectl uncordon $NODENAME
node/nodename uncordoned
```

Backup a cluster

Backup the files `/etc/kubernetes/pki/ca.crt` and `/etc/kubernetes/pki/ca.key` after cluster installation.

Periodically create a snapshot of the `etcd` database with the command:

```
$ kubectl exec -it -n kube-system etcd-controller \
   sh -- -c "ETCDCTL_API=3 etcdctl snapshot save snapshot.db \
   --cacert /etc/kubernetes/pki/etcd/server.crt \
   --cert /etc/kubernetes/pki/etcd/ca.crt \
   --key /etc/kubernetes/pki/etcd/ca.key"
Snapshot saved at snapshot.db
$ kubectl cp -n kube-system etcd-controller:snapshot.db snapshot.db
```

Restore a cluster

Install the controller again by following the steps in Creating a cluster with kubeadm and stop before running `kubeadm init`.

Copy ca.crt and ca.key in /etc/kubernetes/pki, restoring the good permissions:

```
# ls -l /etc/kubernetes/pki/
total 8
-rw-r--r-- 1 root root 1025 Feb  1 10:43 ca.crt
-rw------- 1 root root 1675 Feb  1 10:43 ca.key
```

Place snapshot.db in /mnt, then run:

```
$ docker run --rm \
   -v '/mnt:/backup' \
   -v '/var/lib/etcd:/var/lib/etcd' \
   --env ETCDCTL_API=3 \
   'k8s.gcr.io/etcd-amd64:3.1.12' \
   /bin/sh -c \
   "etcdctl snapshot restore /backup/snapshot.db ; mv /default.etcd/member/ /var/lib\
/etcd/"
```

Install the control plane again:

```
$ gcloud config set compute/zone us-west1-c # or your selected zone
Updated property [compute/zone].
$ KUBERNETES_PUBLIC_ADDRESS=$(gcloud compute instances describe controller \
  --zone $(gcloud config get-value compute/zone) \
  --format='get(networkInterfaces[0].accessConfigs[0].natIP)')
$ sudo kubeadm init \
  --pod-network-cidr=10.244.0.0/16 \
  --ignore-preflight-errors=NumCPU \
  --apiserver-cert-extra-sans=$KUBERNETES_PUBLIC_ADDRESS \
  --ignore-preflight-errors=DirAvailable--var-lib-etcd
```

kubectl

`kubectl` is the command-line tool used to work on Kubernetes clusters. You can use it to create applications resources, cluster resources, interact with running containers, manage the cluster.

`kubectl completion`
: this command outputs the shell code to execute to make the auto-completion work with the `kubectl` command. Its simplest usage is to *"source"* its output, that will make auto-completion available for the current shell session:

```
$ source <(kubectl completion bash)
```

or, if using the `zsh` shell:

```
$ source <(kubectl completion zsh)
```

Run `kubectl completion --help` to get the instructions how to install the completion in a permanent way.

Managing kubeconfig file

`kubectl`, and other Kubernetes programs, get the credentials necessary to connect to Kubernetes clusters in a *kubeconfig* file. This file is by default searched at `$HOME/.kube/config`. It is possible to use another file either by using the `--kubeconfig` flag or by defining the `KUBECONFIG` environment variable.

The value of `KUBECONFIG` is a colon-delimited list of paths to config files, for example:

```
/etc/kubernetes/config:$HOME/.kube/config
```

When defining several paths, kubectl will **merge** these different files (in memory) and use the results as a single config file.

A kubeconfig file is composed of:

- a **list of clusters informations** (certificate, API URL),
- a **list of users credentials**,
- a **list of contexts**, each context referencing existing *cluster, user and default namespace.*
- a **current context**, poiting to a specific context in the list of contexts.

`kubectl config`
: this command provides subcommands to edit the list of clusters, users and contexts, and to switch the current context.

- `get-clusters`, `set-cluster`, `delete-cluster` to edit clusters informations,
- `set-credentials` to edit users credentials,
- `get-contexts`, `set-context`, `delete-context`, `rename-context` to edit contexts,
- `set`, `unset` are generic subcommands to edit any field of the kubeconfig file,
- `current-context` to get the name of the current context,
- `use-context` to set the name of the current context.
- `view` to view the kubeconfig information visible by the command (depending on `--kubeconfig` flag and `KUBECONFIG` environment variable), especially useful to see the result of merging several config files, or getting the information necessary for the current context with `--minify`.

Generic commands

`kubectl apply`
: most kubectl commands are used in *imperative* mode. The `apply` command is on the contrary used in *declarative* mode, for *applying* YAML/JSON templates, that is, for declaring resources to the cluster API based on their definitions in YAML or JSON files.

`kubectl get`
: get a tabular list of resources, or their complete definitions with the `-o yaml` flag,

`kubectl delete`
: delete a resource or a set of resources,

`kubectl edit`
: interactively edit a resource definition in your preferred editor (you can change the `EDITOR` environment variable to change it).

`kubectl create namespace`
: create a new namespace.

Creating applications resources

Creating and parameterizing Workloads

`kubectl run`
: deploy a new workload. Most forms of these commands are deprecated.

  The one not deprecated is to deploy a Pod with a single container:

```
kubectl run --generator=run-pod/v1 podname --image=imagename
```

Flags are available to set specific fields of the container: image pull policy, environment variables, ports, resources requests and limits.

The `dry-run -o yaml` is useful to output the declarative form of the pod, that can be edited and applied later with `kubectl apply`.

`kubectl create deployment`

create a deployment in its simplest form: `kubectl create deployment nginx --image=nginx`. No flags are available to parameterize the deployment.

The `dry-run -o yaml` is useful to output the declarative form of the deployment, that can be edited and applied later with `kubectl apply`.

The `set`, `scale` commands are useful to parameterize the deployemnt.

`kubectl create cronjob`

create a cronjob, given an image and a schedule. You can also specify the command and args to pass to the container, useful when using a generic image like `busybox`:

```
kubectl create cronjob pinghost --image=busybox \
--schedule="10 * * * *" \
-- sh -c 'ping -c 1 myhost'
```

Use `man 5 crontab` to get the specifications of the `schedule` flag.

The `dry-run -o yaml` is useful to output the declarative form of the cronjob, that can be edited and applied later with `kubectl apply`.

`kubectl create job`

execute a job. Two forms are available:

- Create a job given an image, and optionally a command (very similar to the `create cronjob` command but for a single job):

```
kubectl create job pinghost --image=busybox \
-- sh -c 'ping -c 1 myhost'
```

- Create a job from an existing cronjob, bypassing the cronjob's schedule:

```
kubectl create job pinghost --from=cronjob/pinghost
```

`kubectl scale`

scale applicable resources (Deployment, ReplicaSet, StatefulSet) to the given number of replicas. The selection of the resources to scale is done by type/name, label selector or file. Some preconditions can be specified: number of current replicas, and version of the resource.

`kubectl autoscale`
: create an auto-scaler for applicable resources (Deployment, ReplicaSet, StatefulSet). The selection of the resources to auto-scale is done by type/name or file. The important flags are `min`, `max` and `cpu-percent` to indicate the limits in number of replicas and the **average** CPU percent at which the resource will be scaled.

`kubectl rollout`
: set of commands for rolling update and rollback of applicable resources (Deployment, DaemonSet, StatefulSet). See Rolling update and Rollback for details.

Configuring Workloads

`kubectl create configmap`
: create a `ConfigMap` resource, getting key/value pairs:

- from env-style files (`--from-env-file`), the keys and values will be extracted from the file:

```
# .env
key1=value1
key2=value2
```

- from files (`--from-file`): the key will be the filename or the specified key and the value the contents of the file, for the specified file or all files of the specified directory:

```
--from-file=dir --from-file=file1.txt --from-file=key=file2.txt
```

- from literal values (`--from-literal`):

```
--from-literal=key1=value1 --from-literal=key2=value2
```

`kubectl create secret`
: create a `Secret` resource. Three forms are available:

- `create secret generic`: very similar to `create configmap`, to add key/value pairs from env-style files, files and literal values.
- `create secret docker-registry`: create a Secret to be used as an `imagePullSecrets` (see Working with private Docker registries). Two forms are available, from a Docker config file (`--from-file`), or by specifying registry information (`--docker-server`, `--docker-username`, `--docker-password` and `--docker-email`).
- `create secret tls`: create a TLS secret given a public/private key pair (`--cert` and `--key`) stored in files.

Exposing pods

`kubectl create service`
: create a `Service` resource. Four forms are available, one for each service type:

- `create service clusterip` creates a ClusterIP service, specifying the ports with the `--tcp` flag:

```
kubectl create service clusterip my-svc \
 --tcp=8084:80
```

A selector `app=my-svc` will be added, to match pods with this label. You will probably need to edit it to match the labels of the desired pods.

It is possible to specify your own IP for the service with `--clusterip=x.y.z.a`, or create a Headless service with `--clusterip=None`.

- `create service nodeport` creates a NodePort service. In addition to the `clusterip` variant, it is possible to specify a node port with the `--node-port` flag:

```
kubectl create service nodeport my-svc \
 --tcp=8084:80 --node-port=30080
```

- `create service loadbalancer` creates a LoadBalancer service, specifying the clusterIP ports with the `--tcp` flag:

```
kubectl create service loadbalancer my-svc \
 --tcp=8084:80
```

Note that it is not possible to specify a port for the underlying node port.

- `create service externalname` creates an ExternalName service:

```
kubectl create service externalname my-svc \
 --external-name=an-svc.domain.com
```

`kubectl expose`
: expose an applicable resource (service, pod, replicaset, deployment) creating a new `Service` resource.

Authorization

`kubectl create role`

`kubectl create clusterrole`

`kubectl create rolebinding`

`kubectl create clusterrolebinding`

`kubectl create serviceaccount`

`kubectl auth`
: All these commands are used to create and verify authorizations. See Authorization for details.

Annotate and label

`kubectl annotate`
: Attach metadata to any kind of resource, not used by Kubernetes but intended to be used by tools or system extensions.

`kubectl label`
: Edit labels on any kind of resource, used as selectors.

Interacting with application

```
kubectl attach
kubectl exec
kubectl cp
kubectl describe
kubectl logs
kubectl port-forward
kubectl proxy
kubectl top
```

Managing clusters

`kubectl taint`
: Edit taints of nodes.

```
kubectl uncordon
kubectl cordon
kubectl drain
kubectl certificate
kubectl cluster-info
kubectl version
```

Getting Documentation

```
kubectl api-versions
kubectl api-resources
kubectl explain
kubectl options
kubectl help
```

Curriculum CKA 1.15

Scheduling (5%)

Use label selectors to schedule Pods
Using label selectors to schedule Pods on specific nodes
Understand the role of DaemonSets
DaemonSets
Understand how resource limits can affect Pod scheduling
Resource requests
Understand how to run multiple schedulers and how to configure Pods to use them
Running multiple schedulers
Manually schedule a pod without a scheduler
Manual scheduling, Static Pods
Display scheduler events
Examine schedulers events
Know how to configure the Kubernetes scheduler
Scheduling Pods

Logging/Monitoring (5%)

Understand how to monitor all cluster components
Control Plane components
Understand how to monitor applications
Auto-scaling
Manage cluster component logs
Explore control-plane services
Manage application logs
Basic logging

Application Lifecycle Management (8%)

Understand Deployments and how to perform rolling updates and rollbacks
ReplicaSet controller, Deployment Controller and Update and Rollback, Deployment Strategies
Know various ways to configure applications
Configuring applications

Know how to scale applications
Scaling an application
Understand the primitives necessary to create a self-healing application
Application Self-Healing

Cluster Maintenance (11%)

Understand Kubernetes cluster upgrade process
Upgrading the cluster
Facilitate operating system upgrades
Upgrading the operating system
Implement backup and restore methodologies
Backup a cluster, Restore a cluster

Security (12%)

Know how to configure authentication and authorization
Authentication, Authorization
Understand Kubernetes security primitives
Security
Know to configure network policies
Networkpolicies
Create and manage TLS certificates for cluster components
Client Certificate authentication
Work with images securely
Working with private Docker registries
Define security contexts
Security Contexts
Secure persistent key value store
TODO

Storage (7%)

Understand persistent volumes and know how to create them
Persistent Volumes
Understand access modes for volumes
Access modes
Understand persistent volume claims primitive
Claiming a Persistent volume

Understand Kubernetes storage objects
Storage
Know how to configure applications with persistent storage
Claiming a Persistent volume

Troubleshooting (10%)

Troubleshoot application failure
Interacting with application
Troubleshoot control plane failure
Control Plane components
Troubleshoot worker node failure
Control Plane components
Troubleshoot networking
Discovery and Load-balancing

Core Concepts (19%)

Understand the Kubernetes API primitives
Kubernetes Resources, The workloads
Understand the Kubernetes cluster architecture
Creating a cluster with kubeadm
Understand Services and other network primitives
Discovery and Load-balancing

Curriculum CKAD 1.15

Core Concepts (13%)

Understand Kubernetes API primitives
Kubernetes Resources, The workloads
Create and configure basic Pods
The workloads

Configuration (18%)

Understand ConfigMaps
Configuring applications
Understand SecurityContexts
Security Contexts
Define an application's resource requirements
Resource requests, Resource limits and Quality of Service classes
Create & consume Secrets
Configuring applications
Understand ServiceAccounts
Service account authentication

Multi-Container Pods (10%)

Understand Multi-Container Pod design patterns(e.g. ambassador, adapter, sidecar)
TODO

Observability (18%)

Understand LivenessProbes and ReadinessProbes
Liveness-probes, A note about Readiness probes, Readiness Probes
Understand container logging
Basic logging
Understand how to monitor applications in Kubernetes
Auto-scaling
Understand debugging in Kubernetes
Interacting with application

Pod Design (20%)

Understand how to use Labels, Selectors, and Annotations
Labels and selectors, Annotations, Service selectors, Using label selectors to schedule Pods on specific nodes
Understand Deployments and how to perform rolling updates
Update and Rollback, Deployment Strategies
Understand Deployments and how to perform rollbacks
Update and Rollback
Understand Jobs and CronJobs
TODO

Services and Networking (13%)

Understand Services
Discovery and Load balancing
Demonstrate basic understanding of NetworkPolicies
Network policies

State Persistence (8%)

Understand PersistentVolumeClaims for storage
Storage