Politecnico di Torino

# Introduction to Kubernetes

Alex Palesandro, Marco Iorio


January 21, 2022

## License




## Acknowledgments

The author would like to thank all the people who contributed to this document, particularly Francesco Lucrezia and Alessio Sacco who created on an early version of this lab.

# Contents

# 1 Introduction

This lab aims at practicing with a Kubernetes cluster and getting in touch with its capabilities.

The first part of the lab deals with the creation and setup of the cluster, which consists of three VMs: one playing the role of *control plane* node, and two of *worker* nodes. The VMs will be hosted on CrownLabs (`https://crownlabs.polito.it/`).

Upon cluster installation, the second part of the lab allows you to practice with the basic Kubernetes concepts, such as *pods*, *deployments* and *services*, to understand how to execute simple applications. Afterward, you will create more complex configurations, accounting for high-availability concerns.

**Extra**: In addition to this text, we provide several videos which discuss some parts of the topics of this lab. You can find them at

`https://www.youtube.com/playlist?list=PLTAfidx4guQImT5beuAs4YAhIzuBBoEHk`

# 2 Installing and configuring a Kubernetes cluster with kubeadm

We start creating a new Kubernetes cluster, composed of a single control plane node and two worker nodes (Figure 2.1). The installation process requires the *kubeadm*, *kubectl* and *kubelet* packages to be installed on all servers: the CrownLabs images already include them.

**Note:** the setup proposed for this lab does not lead to the creation of a *highly available cluster* (i.e., which can tolerate the failure of a control plane node), and it is therefore not suggested for production workloads. See the official documentation[1] for additional information about this topic. In the current setup, a failure of the control-plane node may lead to data loss in your cluster and you may need to re-create everything from scratch.

The Kubernetes *control plane*, including as the *api server* and the *controller manager*, governs the cluster behavior. Specifically, the control plane maintains a record of all of the Kubernetes objects in the system, and runs continuous control loops to manage those objects' state. At any given time, the control plane's control loops respond to changes in the cluster and work to make the actual state of all the objects in the system match the desired state that you provided. When you interact with Kubernetes, such as using the `kubectl` command-line interface, you are communicating with your cluster's Kubernetes control plane.

For example, when a *deployment* is created through the Kubernetes APIs, it specifies a new desired state for the system. The Kubernetes control plane reacts to that object creation, and performs the operations required to schedule the desired applications to cluster nodes, thus making the cluster's actual state match the desired state.

The *worker* nodes, instead, are the machines (VMs, physical servers, etc) that run the actual applications and workflows. The Kubernetes control plane controls each node: you will rarely interact with the worker nodes directly.

## 2.1 Configuring the control plane node

First, select a VM that is supposed to act as **control plane node** and change the name of the machine to clarify its function in the Kubernetes cluster:

```
sudo hostnamectl set-hostname control-plane-1
```

**Warning**: you have to disconnect from the VM and open another SSH session to update the prompt.

Then, from the same node, it is possible to leverage *kubeadm* to initialize the control-plane:

```
sudo kubeadm init --pod-network-cidr=10.254.0.0/16 --service-cidr=10.255.0.0/16
```

[1] `http://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/`

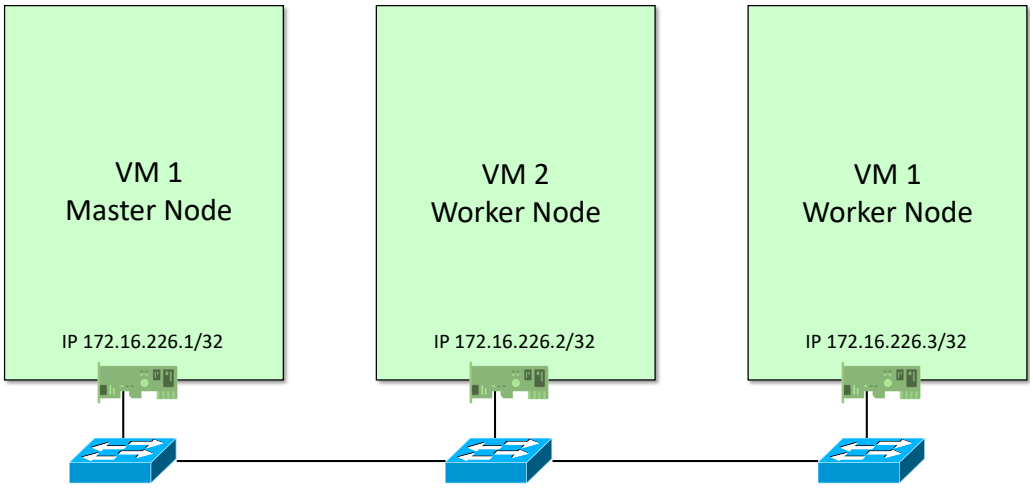


Figure 2.1: Logical setup of the Kubernetes cluster in CrownLabs.

`kubeadm init` first runs a series of *pre-flight* checks to ensure that the machine is ready to run Kubernetes, then downloads and installs the cluster control plane components. This may take several minutes.

**Note**: as part of the installation process, the previous command will print on screen a string that looks like the following:

```
sudo kubeadm join 172.16.183.32:6443 --token lquijh.m7g8o434w21xjaew \
    --discovery-token-ca-cert-hash sha256:2
        a63522f184ed9c7ab3f6e723723e70d1457c133212ee74550f21da769fb5aa1
```

Take note of the above string, because it will be required later in the installation process of the workers. In case you miss this step, do not worry; you can create another token later.

Once the configuration completed, it is possible to make *kubectl* work for the current user, through the following:

```
mkdir -p $HOME/.kube
# The 'kubectl' command expect the KUBECONFIG to be in folder '~/.kube'
# Hence, copying the KUBECONFIG created during the installation process in the above
    folder
sudo cp /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

**Note #1**: the *kubeconfig* file contains all the parameters required to connect to a Kubernetes cluster (e.g., IP address of the API server, etc). This data is used by `kubectl` to connect to the above cluster.

**Note #2**: you can connect to multiple clusters from the same workstation by simply replacing/updating the *kubeconfig* file.

Kubernetes does not embed the logic to interconnect different pods. Instead, it leverages a modular architecture based on the *Container Network interface (CNI)* specifications. Hence, it is possible to choose among many alternative network plugins. In this lab, we select Cilium[2], an open source solution which provides highly scalable networking and network policy capabilities. **Warning:** Neither Calico nor Flannel (two very popular alternatives) do not work properly on CrownLabs, due to the underlying technology.

[2] `https://cilium.io/`

Cilium can be installed through the provided CLI:

```
# Download the Cilium CLI
curl -L --remote-name-all https://github.com/cilium/cilium-cli/releases/latest/
    download/cilium-linux-amd64.tar.gz
# Uncompress the TAR in the /usr/local/bin folder
sudo tar xzvfC cilium-linux-amd64.tar.gz /usr/local/bin
rm cilium-linux-amd64.tar.gz

# Install Cilium
cilium install --ipam=kubernetes
```

Once the Cilium configuration completed, the control plane node should turn into the *Ready* status. You can check the nodes status through `kubectl get nodes` command.

## 2.2 Configuring worker nodes

When the control plane node is ready, access the other two VMs to join them to the cluster as worker nodes. First, change their hostname for easier identification through the following command (remember to close and reopen the SSH session to update the prompt):

```
sudo hostnamectl set-hostname worker-<num>
```

replacing `<num>` with 1 and 2 for the two VMs.

Then, run the `kubeadm join` command echoed during the control plane setup process (if lost, a new one can be generated through `kubeadm token create --print-join-command` from the control plane node) to join the worker node to the control plane. The command should be **similar** to the following (remember to prepend `sudo`):

```
sudo kubeadm join 172.16.183.32:6443 --token lquijh.m7g8o434w21xjaew \
    --discovery-token-ca-cert-hash sha256:2
        a63522f184ed9c7ab3f6e723723e70d1457c133212ee74550f21da769fb5aa1
```

The token is used for mutual authentication between the control-plane node and the joining nodes. The token included here is secret. Keep it safe, because anyone with this token can add authenticated nodes to your cluster.

Repeat the join process on both worker nodes. Then, move to the control plane and check all nodes are correctly ready through:

```
kubectl get nodes
```

**Question** (left to the student): why the command `kubectl get nodes` does not work if issued on a worker node?

## 2.3 Accessing the cluster from a management workstation

Instead of issuing the `kubectl` commands directly from the control plane node, it is possible to use a separate management workstation (e.g., a client VM in CrownLabs). To this end, it is necessary to copy the *kubeconfig* to the workstation by typing, on the client workstation, the following commands (remember to replace `__MASTER_IP__` with the IP address of the control node):

```
mkdir -p $HOME/.kube
scp crownlabs@__MASTER_IP__:~/.kube/config $HOME/.kube/
```

**Warning:** the control plane IP (TCP port 6443) needs to be reachable from the management workstation for the `kubectl` commands to work.

Then you can verify `kubectl` correctly works:

```
kubectl get nodes
```

**Hint**: To simplify the interaction with the server, is is strongly suggested to enable bash auto-completion (and the `k` alias) for `kubectl` commands:

```
echo 'source <(kubectl completion bash)' >> $HOME/.bashrc
echo 'alias k=kubectl' >> $HOME/.bashrc
echo 'complete -F __start_kubectl k' >> $HOME/.bashrc
source $HOME/.bashrc
```

## 2.4 Deploying the Kubernetes Dashboard

To simplify the interaction with the cluster, it is possible to use the official Kubernetes dashboard. The Dashboard UI is not deployed by default. To deploy it, run the following command:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.4.0/aio/
    deploy/recommended.yaml
```

To protect your cluster data, the dashboard deploys with a minimal RBAC configuration by default. Currently, it only supports Bearer Token authentication. To create a token with administrative privileges for this lab, you can proceed as follows:

```
kubectl create serviceaccount -n kubernetes-dashboard dashboard-admin
kubectl create clusterrolebinding dashboard-admin --clusterrole=cluster-admin \
    --serviceaccount=kubernetes-dashboard:dashboard-admin
```

Then you can retrieve the authentication token:

```
NAME=$(kubectl get serviceaccount -n kubernetes-dashboard dashboard-admin \
    --no-headers -o custom-columns=':.secrets[0].name')
TOKEN=$(kubectl get secret -n kubernetes-dashboard $NAME \
    -o custom-columns=":.data.token" --no-headers | base64 -d)
echo Token: $TOKEN
```

By default, the Dashboard is exposed through a ClusterIP service, which can only be accessed from within the cluster. One of the possibilities to access it from the management workstation is to leverage the `kubectl port-forward` command, which forwards all traffic sent to a local port (e.g., 8443) to the port of a remote service:

```
kubectl port-forward -n kubernetes-dashboard svc/kubernetes-dashboard 8443:443
```

Then, the Dashboard can be accessed through a browser at `https://localhost:8443`.

# 3 Deploying simple applications through Kubernetes

Since you have now a fully working Kubernetes cluster, we can use it to host some simple applications.

## 3.1 Listing the running pods

First, let take a look at the running pods. You can see that there are no pods running in the *default* namespace:

```
kubectl get pods -o wide
```

At the beginning, this command returns no resources because you have not deployed any applications yet. However, you will use this command several times in the next sections to monitor the progress of your applications.

Instead, you can find some infrastructural pods within the *kube-system* namespace:

```
kubectl get pods -n kube-system -o wide
```

The *-o wide* option gives information about the pod IPs and host location. The `-n kube-system` option, instead, specifies the namespace for which we ask to see the resources. `kube-system` is the namespace for objects created by the Kubernetes system. `default`, on the other hand, is the one for objects with no namespace explicitly configured.

## 3.2 Running a stateless application & service

You can run an application by creating a Kubernetes Deployment object, and you can describe a Deployment through a YAML file. For example, this YAML file describes a Deployment that runs the `nginxdemos/hello:0.2-plain-text` Docker image (an NGINX server that serves a simple page containing its hostname, IP address and port as wells as the request URI and the local time):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # run two replicas of the following template
  template:
    metadata:
      labels:
        app: nginx
```

```
    spec:
      containers:
      - name: nginx
        image: nginxdemos/hello:0.2-plain-text
        ports:
        - name: http
          containerPort: 80
```

*This code snipped is attached to the PDF file as "snippets/deployment-nginx.yaml"*

**Warning**: YAML files (e.g. `deployment-nginx.yaml`) strongly rely on indentation for their structure. Hence, do *not* copy the previous snippet directly from the PDF file, since it will totally scramble the indentation. Moreover, before moving on, verify (and fix, if necessary) the file layout using an editor of your choice (e.g., `nano deployment-nginx.yaml`).

Create a Deployment based on the YAML file:

```
kubectl apply -f deployment-nginx.yaml
```

Display information about the Deployment:

```
kubectl describe deployment nginx
```

The output displays the information about the label, replicas, and the template of the pod.

List the pods created by the deployment:

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

The command displays all the pods in the cluster with the specified label, as specified by the `-l` parameter. In our case there are two running pods, make sure the status of each is `RUNNNING`.

Try out the following command to further scale the deployment:

```
kubectl scale deployment nginx --replicas=5
```

This command can take a few seconds before all the 5 replicas are ready. Monitor the status of the pods with the `kubectl get pods -l app=nginx` command.

Try to delete one nginx pod with:

```
kubectl delete pod <pod_name>
```

and see what happens, by using (to be executed first in another terminal):

```
kubectl get deployment nginx --watch
```

to monitor the changes. The `--watch` flag to start watching updates to a particular object, press `CTRL + C` to stop the watch process. Scale down the deployment to two instances as before.

```
kubectl scale deployment nginx --replicas=2
```

You can now see that the application consists of two pods only.

With your pods running, it is now time to put them behind a service, an abstraction which exposes a set of pods through a single network endpoint. Use the `kubectl create` command to create the service.

```
kubectl create -f service-nginx.yaml
```

The YAML file describing the service should be like the following:

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

*This code snipped is attached to the PDF file as "snippets/service-nginx.yaml"*

The IP address of the service can be accessed via:

```
kubectl get services nginx
```

The output reports service information, and the IP can be used to access the service. Use first *ClusterIP* as *ServiceType* and try to access the application from the master node. Use `curl` of `wget` to request web pages of the service. You can use the output templating feature of kubectl to run curl as a one-line command.

```
curl -ks http://$(kubectl get svc nginx -o=jsonpath="{.spec.clusterIP}")
```

**Question:** This command has to be executed from a node of the cluster to work. Why?

Then, try to use the *NodePort* service type and try to access the app again. When using a *NodePort* service, you can directly contact one of the node IP addresses to access the app. Modify the type of the service by applying the patch to the existing service.

```
kubectl apply -f service-nodeport.yaml
```

The patch specifies which port to expose and the new ServiceType, by means of `nodePort`.

```
kind: Service
apiVersion: v1
metadata:
  name: nginx
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30100
  type: NodePort
```

*This code snipped is attached to the PDF file as "snippets/service-nginx-nodeport.yaml"*

The two services have different behaviors, indeed:

- `ClusterIP`: Exposes the Service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster. This is the default ServiceType.
- `NodePort`: Exposes the Service on each Node's IP at a static port (the NodePort). A ClusterIP Service, to which the NodePort Service routes, is automatically created. You are able to contact the NodePort Service, from outside the cluster, by requesting `<NodeIP>:<NodePort>`.

Other two service types are available in Kubernets, `LoadBalancer` and `ExternalName`, but those are out of scope in this lab.

When using the `NodePort` service type, you can also connect to the application from the web browser of your management workstation at `http://<IP_MASTER_NODE>:30100`.

**Warning:** The cluster nodes can only be accessed from your other VMs hosted by CrownLabs.

## 3.3 Stateful applications & storage provisioner

On-disk files in a container are ephemeral, which is problematic in case of stateful application. First, when a container crashes, the *kubelet* restarts it, but the files are lost – the container starts with a clean state. Second, when running containers together in a pod, it is often necessary to share files among them.

To execute stateful applications in Kubernetes, it is possible to leverage the *PersistentVolume (PV)* and *PersistentVolumeClaim (PVC)* abstractions. A PersistentVolume is a piece of storage in the cluster that has been provisioned either by an administrator or dynamically using Storage Classes. A PersistentVolumeClaim (PVC) is an abstract request for storage by a user, which eventually triggers the creation of the associated PersistentVolume (when dynamic creation is supported) and binds a pod to it.

### 3.3.1 Creating the Storage Class

In this lab we leverage `Local Persistent Volumes`, which allow to expose local folders as PersistentVolumes, and use them in PersistentVolumeClaim objects.

**Warning**: Local volumes are not appropriate for most applications. Using local storage ties your application to that specific node, making your application harder to schedule. If that node or local volume encounters a failure and becomes inaccessible, then that pod also becomes inaccessible. For these reasons, we recommended to use dynamic allocation provided by tools like NFS, Ceph, and OpenStack Cinder.

Some use cases that are suitable for local storage include: *(i)* caching of datasets that can leverage data gravity for fast processing, *(ii)* distributed storage systems that shard or replicate data across multiple nodes. Examples include distributed datastores like Cassandra, or distributed file systems like Gluster or Ceph. Suitable workloads are tolerant of node failures, data unavailability, and data loss. They provide critical, latency-sensitive infrastructure services to the rest of the cluster, and should run with high priority compared to other workloads.

Before any PersistentVolumeClaims for your local PersistentVolumes can be setup, a `StorageClass` must be created with the volume binding mode set to *WaitForFirstConsumer*:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
```

```
  name: local-storage
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

*This code snipped is attached to the PDF file as "snippets/storage-class.yaml"*

This volume binding mode configuration tells the PersistentVolume controller to not immediately bind a PersistentVolumeClaim. Instead, the system waits until a pod that needs to use a volume is scheduled. The scheduler then chooses an appropriate local PersistentVolume to bind to, taking into account the pod's other scheduling constraints and policies. This ensures that the initial volume binding is compatible with any pod resource requirements, selectors, affinity and anti-affinity policies, and more.

**Warning:** dynamic provisioning is currently not supported by local volumes. All local PersistentVolumes must be statically created. In this case, local disks must first be pre-partitioned, formatted, and mounted on the local node by an administrator. Directories on a shared file system are also supported, but must also be created before use.

Once the storage class is created:

```
kubectl apply -f storage-class.yaml
```

it is possible to deploy a PersistentVolume. In this example, the local volume is mounted at "/data/" on node "worker-2":

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
  local:
    path: /data/
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - worker-2
```

*This code snipped is attached to the PDF file as "snippets/pv.yaml"*

**Warning:** It is necessary to manually create the directory on the node we decided to use for this PV. For example, on `worker-2`:

```
sudo mkdir --parents /data
```

Note that there is a `nodeAffinity` field in the PersistentVolume object: this is how the Kubernetes scheduler understands that this PersistentVolume is tied to a specific node. nodeAffinity is a required

field for local PersistentVolumes. **Warning:** Check that the node listed in the affinity section is part of your cluster.

When local volumes are manually created, the only supported persistentVolumeReclaimPolicy is *Retain*. When the PersistentVolume is released from the PersistentVolumeClaim, an administrator must manually clean up and set up the local volume again for reuse.

Create the PersistentVolume through:

```
kubectl apply -f pv.yaml
```

### 3.3.2 Deploying MySQL

After all the preparatory work, a local volume can be requested in exactly the same way as any other PersistentVolume type: through a PersistentVolumeClaim. Just specify the appropriate StorageClassName for local volumes in the PersistentVolumeClaim object, and the system takes care of the rest.

Hence, to run a stateful application, create a Kubernetes Deployment and connect it to an existing PersistentVolume using a PersistentVolumeClaim. For example, the following describes a deployment that runs MySQL and references the PersistentVolumeClaim. The file defines a volume mount for /var/lib/mysql, and then creates a PersistentVolumeClaim that looks for a 1G volume. This claim is satisfied by any existing volume that meets the requirements, or by a dynamic provisioner.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        env:
        # Use a secret in real usage
        - name: MYSQL_ROOT_PASSWORD
          value: password
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
```

*This code snipped is attached to the PDF file as "snippets/mysql-deployment.yaml"*

**Note**: The mysql password is defined in the configuration, and this is insecure. See Kubernetes Secrets for a secure solution.

The field `strategy.type: Recreate` in the Deployment configuration YAML file instructs Kubernetes to not use rolling updates. Rolling updates will not work, as you cannot have more than one Pod running at a time bound to the PersistentVolume. The `Recreate` strategy will stop the first pod before creating a new one with the updated configuration.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
spec:
  storageClassName: local-storage
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

*This code snipped is attached to the PDF file as "snippets/mysql-pvc.yaml"*

First, deploy the PVC configuration:

```
kubectl apply -f mysql-pvc.yaml
```

Then, deploy the deployment configuration:

```
kubectl apply -f mysql-deployment.yaml
```

You can display information about the deployment with:

```
kubectl describe deployment mysql
```

The output highlights the number of replicas, the mount, and the used volume. Make sure the status of the pod is running, by running:

```
kubectl get pods -l app=mysql
```

The PersistentVolumeClaim can be inspected with the following command:

```
kubectl describe pvc mysql-pv-claim
```

and check the status is *Bound* and the capacity is as described.

### 3.3.3 Accessing the DB

The preceding YAML file creates a service that allows other Pods in the cluster to access the database.

Connect to the MySQL server:

```
kubectl exec -it <MYSQL_POD_NAME> -- bash
mysql -u root --password
```

You have entered the pod, and opened the mysql console. If the database is up and running, you should see an output like this:

```
mysql>
```

Once you are connected to the MySQL server, a welcome message is displayed and the `mysql>` prompt appears, at which SQL statements are to be sent to the server for execution. You can create a new database by using a `CREATE DATABASE` statement:

```
mysql> CREATE DATABASE pets;
Query OK, 1 row affected (0.01 sec)
```

and check if the database has been created:

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| pets               |
| sys                |
+--------------------+
5 rows in set (0.00 sec)
```

Exite the pod typing `CTRL + D`.

Test the presistence of data and delete the pod of mysql,

```
kubectl delete pod <MYSQL_POD_NAME>
```

After a minute, try to access the database again with the same client, and get the available databases.

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| pets               |
| sys                |
+--------------------+
5 rows in set (0.00 sec)
```

You can see the `pets` database is still available.

**Note.** This solution is suitable for single-instance applications only. Don't scale the app. The underlying PersistentVolume can only be mounted to one pod. For clustered stateful apps, use the *StatefulSet* abstraction.

Before proceeding, delete the deployed objects:

```
kubectl delete -f service-nginx.yaml
kubectl delete -f service-nginx-nodeport.yaml
kubectl delete -f deployment-nginx.yaml
kubectl delete -f mysql-deployment.yaml
```

# 4 Self-healing applications with Kubernetes

One of the most in-demand container orchestration features is monitoring of containers' health and ability of the orchestrator to deal with unhealthy containers according to the specified configuration. For example, the orchestrator can launch a new container as a replacement for a container that is not healthy. In Kubernetes, the same technique applies to a pod, the smallest deployable unit, consisting of one or more containers that share the same IP address and port space. When something goes wrong, meaning the desired state and the actual state doesn't match, Kubernetes will try to close the gap, forever.

The main point is, how can we determine which container is healthy and which is not? As we know, there is a single main process that is running in a container. Such a process can start other child processes within a container, if necessary. Every such process, including the main process, can have its own lifecycle - but if the main process stops, the container stops as well.

A container is healthy, by the most general definition, if its main process is running. If the container's main process is terminated unexpectedly, then the container is considered unhealthy.

**Note:** You should monitor health only for containers that are running permanently, such as web servers or databases. If you are running a container that is expected to stop sometime, there is no reason to monitor its health. Instead, you should analyze its result, such as its exit code. You can use a Kubernetes *Job* for pods that are expected to terminate on their own.

In addition, you should take into consideration that the container keeps running and is considered healthy even if one or more child processes are terminated unexpectedly. Furthermore, the main process might be running but not working as expected, because, for example, it was not configured properly. For such cases, we need an application-specific way to determine the container's health.

Each Kubernetes pod has a `phase` field, which provides a simple, high-level summary of where the pod is in its lifecycle. The pod can be in one of the following phases:

- `Pending` - If the pod is created, but one or more of containers are not running yet. For example, it can take some time for the container image to be downloaded over the network.
- `Running` – All of the containers in the pod are running.
- `Succeeded` - All of the containers in the pod are terminated with zero exit code.
- `Failed` - All of the containers in the pod are terminated and at least one container failed (either exited with non-zero exit code or was terminated by the system).
- `Unknown` - The state of the pod could not be obtained for some reason.

For each pod, Kubernetes can periodically execute *liveness* and *readiness* probes, if they are defined for the pod. A probe is an executable action, that can check if the specified condition is met. There are three types of actions:

- `ExecAction` - Executes a specified command in the container. The condition is considered successful if the command exits with zero as its exit code.

- `TCPSocketAction` - Performs a TCP check against the container's IP address on a specified port. The condition is considered successful if the port is open.
- `HTTPGetAction` - Performs an HTTP GET request against the container's IP address on a specified port and path. The condition is considered successful if the response has an HTTP status code greater than or equal to 200 and less than 400.

A *liveness probe* determines whether the container is running or not. If the liveness probe fails, then Kubernetes kills the container. A new container can be started instead, if a *restart policy* says so. Although there is no default liveness probe for a container, you do not necessarily need one, because Kubernetes will automatically perform the correct action in accordance with the pod's restart policy.

A *readiness probe* determines whether the container is ready to service requests. If the readiness probe fails, the endpoints controller removes the pod's IP address from the endpoints of all services for that pod. There is no default readiness probe for a container.

A pod has a *restartPolicy* field with possible values *Always, OnFailure,* and *Never*. The default value is Always. The restart policy applies to all of the containers in the pod. Failed containers are restarted on the same node with a delay that grows exponentially up to 5 minutes.

## 4.1 Define and create a liveness probe for a pod

To simulate a failure in a reproducible manner, we force failures in a Kubernetes deployment, to verify the self-heling properties of Kubernetes. First, define a new correct pod in the `echoserver-pod.yaml` file. In this case, we use an existing image echoserver, which is a simple HTTP server that responds with the HTTP headers it receives:

```
apiVersion: v1
kind: Pod
metadata:
  name: echoserver
spec:
  containers:
    - image: gcr.io/google_containers/echoserver:1.4
      name: echoserver
      ports:
        - containerPort: 8080
      livenessProbe:
        httpGet:
          path: /
          port: 8080
        initialDelaySeconds: 15
        timeoutSeconds: 1
```

*This code snipped is attached to the PDF file as "`snippets/echoserver-pod.yaml`"*

In this example, the liveness probe is defined for the port 8080 and root path. It specifies also other fields:

- `initialDelaySeconds` - The number of seconds after the container has started before liveness probes are initiated.
- `timeoutSeconds` - The number of seconds after which the probe times out. The default value is 1 second; the minimum value is 1.

- `periodSeconds` - How often (in seconds) to perform the probe. The default value is 10 seconds; the minimum value is 1.

Create the echoserver pod:

```
kubectl create -f echoserver-pod.yaml
```

Check that the pod is running and there are no restarts:

```
kubectl get pod echoserver
```

where the output is supposed to be like:

```
NAME         READY     STATUS    RESTARTS   AGE
echoserver   1/1       Running   0          15s
```

## 4.2 Define and create a failing liveness probe for a pod

Edit the file echoserver-pod.yaml. Change the pod's name to echoserver-failing, and change the port number (8080) in the liveness probe to another value, for example, 8081:

```
apiVersion: v1
kind: Pod
metadata:
  name: echoserver-failing
spec:
  containers:
    - image: gcr.io/google_containers/echoserver:1.4
      name: echoserver
      ports:
        - containerPort: 8080
      livenessProbe:
        httpGet:
          path: /
          port: 8081
        initialDelaySeconds: 15
        timeoutSeconds: 1
```

*This code snipped is attached to the PDF file as "snippets/echoserver-pod-failing.yaml"*

Since the echoserver pod will not respond on port 8081, the liveness probe will fail. To check this behavior, create a new echoserver-failing pod:

```
kubectl create -f echoserver-pod-failing.yaml
```

Then, monitor the status of the pods:

```
kubectl get pod --watch
```

You should notice that the restart count of the pod with the failing liveness probe keeps increasing.

```
NAME                READY     STATUS    RESTARTS   AGE
echoserver-failing  1/1       Running   2          1m
```

As you can see, Kubernetes has restarted our pod several times. You can run the following command to investigate what it is happening in detail:

```
kubectl describe pod echoserver-failing
```

In the output of `kubectl describe`, in the Events section, you can see the following information:

```
Killing    76s  Container echoserver failed liveness probe, will be restarted
Unhealthy  56s  Liveness probe failed: ...
```

As the failure is auto-generated, it will never reach a stable state, that satisfies the desired property. On the contrary, generally faults are accidental, and Kubernetes guarantees the state will be eventually accomplished, unless configuration errors are present.

Before proceeding with the next task, clean the Kubernetes environment with:

```
kubectl delete pod echoserver-failing
kubectl delete pod echoserver
```

**To sum up.** Kubernetes provides key mechanisms that allow users to monitor containers' health and restart them in case of failures: probes and the restart policy. Probes are executable actions, which check if the specified conditions are met. The pod's restart policy specifies actions for failed containers. It is worth noting that files in a container are ephemeral, so when a container gets restarted, the changes to the data will be lost. If a container is not stateless, it is necessary to use volumes for persistent storage.

# 5 Resource allocation and scaling

This section is inspired by the article "*Kubernetes best practices: Resource requests and limits*" from the Google GCP blog[1].

So far, we analyzed how we can simply schedule pods on Kubernetes, making the assumption that the cluster will always be able to host our application. However, when Kubernetes schedules a Pod, it's important that the containers have enough resources to actually run. If you schedule a large application on a node with limited resources, it is possible for the node to run out of resources, resulting in unexpected behavior.

They are many reason for an application to require more resources. This could be caused by horizontal scaling policies, where more replicas are spinned to decrease latency, to unexpected behavior that increases the CPU/Memory consumption.

In this section, we first understand how to take control of the cluster avoiding bad resource allocation. Then, we focus on scaling an application dynamically by using its resource consumption as target.

## 5.1 Pre-requirements

However, before digging in resource consumption management, we install the *metrics-server* to rely on a straightforward mechanism to scrape information about real consumption.

### 5.1.1 Deploying the metrics server

In order to collect information about pods and nodes resource consumption, you need a metrics server:

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/
    download/components.yaml
```

To make it work, you have to set the `--kubelet-insecure-tls` flag in the metrics-server deployment. In particular, it has to be added to the args in `spec.containers`.

This can be done by doing a simple kubectl command:

```
kubectl edit deploy metrics-server -n kube-system
```

An alternative may be to download the components.yaml and add the `--kubelet-insecure-tls` option before applying it.

```
hostNetwork: true
containers:
- name: metrics-server
....
```

[1] `https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-resource-requests-and-limits`.

```
  args:
    - --cert-dir=/tmp
    - --secure-port=4443
    - --kubelet-preferred-address-types=InternalIP
    - --kubelet-insecure-tls=true
```

**Warning.** Make sure to use spaces rather than <TAB> for editing the file.

The metric service is now available in Kubernetes:

```
kubectl get pods -n kube-system -o wide
```

You can see a new entry for the pod, with the name `metrics-server-XXXXXXXXX`.

Describe the metrics server deployment and checking that one replica is available:

```
kubectl -n kube-system describe deployment metrics-server
```

Verify there is one replica available.

The Horizontal Pod Autoscaling controller makes use of the metrics provided by the `metrics.k8s.io` API, which is provided by the metrics server. After the metrics server is running on your cluster and has had a chance to collect metrics from the cluster (give it a minute or so), you should be able to use the `kubectl top` command to see the resource usage of the pods and nodes in your cluster.

By exploiting the metric server, you can see the resource usage of the pods and nodes in your cluster. Run:

```
kubectl top pod -A
```

This command prints for each pod the quantity of CPU and Memory currently used. After a minute or so, you are able to see the usage of nodes as well:

```
kubectl top nodes
```

## 5.2 Requests and Limits

When you define a pod template, it is possible to specify how many resources each container needs. There are two main types of resources: **CPU** and **Memory**. The Kubernetes scheduler uses them to figure out **where** to run your pods and each Kubelet to instrument the kernel to limit resource consumption of different pods.

As clearly stated in the Kubernetes[2], you can specify two kind of resource quantities: **requests** and **limits**.

When you specify the **request** for containers in a Pod:

- the **Scheduler** uses this information to decide which node to place the Pod on.
- the **Kubelet** also reserves at least the requested amount of that system resource specifically for that container to use.

When you specify a resource **limit** for a Container:

- the **Kubelet** enforces those limits so that the running container is not allowed to use more of that resource than the limit you set.

[2]https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

### 5.2.1 CPU

Limits and requests for CPU resources are measured in CPU units. In fact, one cpu, in Kubernetes, is equivalent to 1 vCPU/Core for cloud providers and 1 hyperthread on bare-metal Intel/Amd processors.

It is important to understand that the CPU is always requested as an absolute quantity, never as a relative quantity; 0.1 is the same amount of CPU on a single-core, dual-core, or 48-core machine. Fractional requests for CPU resources are allowed. For example, if your container needs 3 full cores to run, you would put the value "3000m". If your container only needs $\frac{1}{2}$ of a core, you would put a value of "500m".

**Hint:** Unless your app is specifically designed to take advantage of multiple cores (scientific computing and some databases come to mind), it is usually a best practice to keep the CPU request at '1' or below, and run more replicas to scale it out. This gives the system more flexibility and reliability.

**Notice:** If your app starts hitting your CPU limits, the container will be artificially restricted or **throttled**. However, it will not be terminated or evicted. Analyzed in Section 4.1, liveness check can be used to be sure that your application is fully functional.

**Notice:** If you put in a value larger than the core count of your biggest node, your pod will never be scheduled.

### 5.2.2 Memory

Limits and requests for memory are measured in bytes. You can express memory as a plain integer or as a fixed-point number using one of these suffixes: E, P, T, G, M, K. You can also use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki. For example, the following represent roughly the same value: 128974848, 129e6, 129M, 123Mi

**Notice:** Like CPU, if you put in a memory request that is larger than the amount of memory on your nodes, the pod will never be scheduled.

**Warning: Unlike CPU resources, memory cannot be compressed**. In other words, if a container **goes past its memory limit it will be terminated** (i.e. OOMKilled).

### 5.2.3 Inspect available resources

To inspect the occupation of node, you can rely on:

```
kubectl top nodes
```

To observe the globally available resources of a certain node, we can use:

```
kubectl get nodes worker-1 -o yaml
```

Questions:

- Which are the fields in the node resource that state the available resources? (They are two)
- Explain the meaning of those fields by leveraging the "kubectl explain" tool.

### 5.2.4 Example

The following presents an example pod configuration, composed of two containers, each with the amount of resources specified in the appropriate section:

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  [...]
  containers:
  - name: app
    image: nginx
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: log-aggregator
    image: my-wonderful-sidecar
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

*This code snipped is attached to the PDF file as "./snippets/requests.yaml"*

### 5.2.5 Exercise

First, let run a pod which attempts to consume an entire CPU:

```
kubectl create deploy stress --image=alexeiled/stress-ng:0.12.05 \
    -- /stress-ng --cpu=1
```

- How many CPUs is the pod consuming? (It takes some time before the metrics are shown by the `kubectl top po` command)

Let us edit the deployment to change the desired amount of CPU consumed (100m), adding the resources section as showed in the pod example at 5.2.4:

```
kubectl edit deploy stress
```

Alternatively, you can dump the manifest to a file, modify it and then re-apply it:

```
kubectl get deploy stress -o yaml > stress-deploy.yaml
kubectl apply -f stress-deploy.yaml
```

Check again the resource usage. How many CPUs is the pod consuming now?

Before moving to the next section, let delete the `stress` deployment:

```
kubectl delete deploy stress
```

## 5.3 Horizontal pod autoscaling

The Horizontal Pod Autoscaler (HPA) automatically scales the number of pods in a replication controller, deployment, replica set or stateful set based on observed CPU utilization. In this example, we demonstrate Horizontal Pod Autoscaler using a custom docker image based on the `php-apache` image.

## 5.4 Autoscaling pods based on CPU usage

To test the scaling of the application, we use a php server based on the apache web server. It defines an `index.php` page which performs some CPU intensive computations:

```
<?php
    $x = 0.0001;
    for ($i = 0; $i <= 1000000; $i++) {
    $x += sqrt($x);
    }
    echo "OK!";
?>
```

First, start a deployment running the image containing the server, and expose it as a service:

```
kubectl create deploy php-apache --image=k8s.gcr.io/hpa-example --port=80
kubectl expose deploy php-apache --port=80
```

Let's edit the deployment, specifying the requests and limits in the resources section:

```
...
    resources:
      requests:
        memory: "128Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "250m"
...
```

Now that the server is running, we create the autoscaler using `kubectl autoscale` command.

```
kubectl autoscale deployment php-apache --cpu-percent=25 --min=1 --max=10
```

This command creates a Horizontal Pod Autoscaler (HPA) that maintains between 1 and 10 replicas of the Pods controlled by the php-apache deployment we created in the first step of these instructions. Roughly speaking, the HPA will increase and decrease the number of replicas (via the deployment) to maintain an average CPU utilization across all Pods of 25%.

**Note.** We set the value of 25% CPU utilization for testing purposes and to not overload the cluster. In production-grade solutions, a reasonable value is 50%, but this may change according to the business

logic. For more details on the autoscaling algorithm see the official documentation[3].

Check the current status of autoscaler by running:

```
kubectl get hpa
```

The output shows that the current CPU consumption is 0% as we are not sending any requests to the server (the `CURRENT` column shows the average across all the pods controlled by the corresponding deployment).

```
NAME       REFERENCE                TARGET    MINPODS MAXPODS   REPLICAS   AGE
php-apache Deployment/php-apache    0% / 25%  1       10        1          18s
```

### 5.4.1 Increase load

To see how the autoscaler reacts to increased load, start a container, and send an infinite loop of queries to the php-apache service (please run it in a different terminal):

```
kubectl run -it --rm load-generator --image=busybox -- /bin/sh

If you don't see a command prompt, try pressing enter.

while true; do wget -q -O- http://php-apache.default.svc.cluster.local; done
```

Within a minute or so, stop the pod that generated the requests with `CTRL + C, CTRL + D`. Then monitor the higher CPU load by executing:

```
kubectl get hpa
```

The output shows that the current CPU consumption has increased.

```
NAME       REFERENCE               TARGET      CURRENT  MINPODS  MAXPODS  REPLICAS
php-apache Deployment/php-apache  305% / 25%  305%     1        10       1
```

In this snapshot, CPU consumption has increased to 305% of the request. As a result, the deployment was resized to 7 replicas, as confirmed by the command:

```
kubectl get deployment php-apache
```

with output:

```
NAME            READY       UP-TO-DATE      AVAILABLE       AGE
php-apache      7/7         7               7               19m
```

It may take a few minutes to stabilize the number of replicas. Since the amount of load is not controlled in any way it may happen that the final number of replicas will differ from this example.

### 5.4.2 Stop load

Before finishing the experiment, stop the user load. In the terminal where we created the container with the `busybox` image, terminate the load generation by typing `<Ctrl> + C`.

After a minute or so, verify the resulting state:

[3] `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale`

```
kubectl get hpa
```

the output confirms the CPU utilization is 0%.

```
NAME       REFERENCE                TARGET    MINPODS MAXPODS   REPLICAS   AGE
php-apache Deployment/php-apache    0% / 25%  1       10        1          11m
```

To verify the number of replicas, run the following command:

```
kubectl get deployment php-apache
```

since CPU utilization is dropped to 0, HPA autoscaled the number of replicas back down to 1.

```
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
php-apache   1/1     1            1           27m
```

Additional metrics can be used when autoscaling the `php-apache` Deployment by making use of the `autoscaling/v2beta2` API version. Examples of other metrics are requests per second, number of HTTP GET requests, and so on.

**To sum up.** Kubernetes comes with many services already deployed, yet it may need to be extended for production purposes. To do so, custom services such as *metrics-server* can be installed and communicates with the API service. Hence, other apps can exploit new services as well. In this section, we installed the *metrics-server* to make the services scale up and down according to the incoming traffic.

# 6 Permission management

As specified in documentation, every call to the Kubernetes API has to be authenticated and authorized. The last part is normally driven by RBAC (Role-based access control) policies.

The key elements for the RBAC module in Kubernetes are the following:

- **Roles/ClusterRoles**: contain rules that represent a set of permissions. Permissions are purely additive (there are no "deny" rules). A Role always sets permissions within a particular namespace; when you create a Role, you have to specify the namespace it belongs to. ClusterRole, by contrast, is a non-namespaced resource. The resources have different names (Role and ClusterRole) because a Kubernetes object always has to be either namespaced or not namespaced; it can't be both. In addition, ClusterRoles can define permissions on cluster-scoped resources. In general, to define a role within a namespace, we can use a Role; if we want to define a role cluster-wide, use a ClusterRole.
- **RoleBinding/ClusterRoleBindings**: a role binding grants the permissions defined in a role to a user or set of users. It holds a list of subjects (users, groups, or service accounts), and a reference to the role being granted. A RoleBinding grants permissions within a specific namespace whereas a ClusterRoleBinding grants that access cluster-wide. A RoleBinding may reference any Role in the same namespace. Alternatively, a RoleBinding can reference a ClusterRole and bind that ClusterRole to the namespace of the RoleBinding. If you want to bind a ClusterRole to all the namespaces in your cluster, you use a ClusterRoleBinding.
- **Subjects** (i.e. ServiceAccounts/User/Groups): represent the identities of accounts which want to access the API, and can be associated to particular ClusterRoles/Roles via ClusterRoleBinding/RoleBinding. ServiceAccount are used for technical accounts (i.e. applications inside pods that want to access the API).

In the following snippet, you can observe how to define a role:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

*This code snipped is attached to the PDF file as "./snippets/role.yaml"*

Each role has a set of associated rules. Each rule is characterized by an APIGroup and resources to identify the objects and a list of actions.

The rolebinding on the contrary is composed by a roleRef, which identifies the ClusterRole/Role object to assign, and a set of subjects to be bound to the permissions.

```
apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "jane" to read pods in the "default" namespace.
# You need to already have a Role named "pod-reader" in that namespace.
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
# You can specify more than one "subject"
- kind: User
  name: jane # "name" is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  # "roleRef" specifies the binding to a Role / ClusterRole
  kind: Role #this must be Role or ClusterRole
  name: pod-reader # this must match the name of the Role or ClusterRole you wish to
      bind to
  apiGroup: rbac.authorization.k8s.io
```

*This code snipped is attached to the PDF file as "./snippets/rolebinding.yaml"*

**Exercise:** Let's create a pod which is capable to interact with the API server and get the pod list as we do from our client with kubectl.

- Detect which identity is mounted in a specific pod we want to run.

  **Hint:** Have a look to the `serviceaccount` field when getting a pod resource (e.g. `kubectl get po -o yaml`)

- Assign the correct permission to the pod service account. You can define a new role for the occasion or rely on existing clusterroles.

**Hint**: You can create a pod with `kubectl` inside through:

```
kubectl run -it --rm kubectl --command --image=bitnami/kubectl:1.22.4 -- /bin/bash
```

# 7 Expose an application using Kubernetes

There are several ways to expose an application outside of a Kubernetes cluster.

Services are an abstract way of exposing an application running on a set of pods as a network service. A service provides a single point of access from outside the Kubernetes cluster and allows you to dynamically access a group of replica pods.

For internal application access within a Kubernetes cluster, a `ClusterIP` service is the preferred method. It is a default setting in Kubernetes and uses an internal IP address to access the service.

To expose a service to external network requests, `NodePort` and `LoadBalancer` services, and `Ingress` resources are possible options. The key advantage of using an ingress resource is that it makes possible to expose multiple services using a single abstract endpoint, a load balancer, or both at once. Taking this approach, different applications can enact host, prefix, and other rules to route traffic to defined service resources however they prefer.

Let's deploy an application:

```
kubectl apply -f https://raw.githubusercontent.com/liqotech/microservices-demo/master
    /release/kubernetes-manifests.yaml
```

## 7.1 Services

We may observe that this a micro-service based website, with multiple deployments and service. The "frontend" service exposes a web server to serve the end-user.

**Exercise** Let's have a look to the services we just created:

- How can we access to the front-end service?
- Why does the "frontend-external" service remain with a "Pending" external IP?

## 7.2 Ingress

An Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

**Exercise**: With the commands you used in the previous exercise create a deployment and a service.

An Ingress may be configured to give services externally-reachable URLs, load balance traffic, terminate SSL/TLS, and offer name-based virtual hosting. An Ingress does not expose arbitrary ports or protocols. Exposing services other than HTTP and HTTPS to the Internet typically uses a service of type NodePort or LoadBalancer.

Despite the ingress resources are native in Kubernetes, similar as for the CNI, there is **no default ingress controller provided at Kubernetes installation**.

Therefore, we need to install a specific ingress controller to effectively expose application without having to rely on services. In this section, we will install an ingress-controller based on the NGINX web server. The goal of this Ingress controller is the assembly the NGINX configuration file (nginx.conf) in the ingress-controller pod.

NGINX Ingress Controller can be easily installed via:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/
    controller-v1.1.1/deploy/static/provider/baremetal/deploy.yaml
```

Additionally, let configure the NGINX Ingress Controller as the default one, so that each new ingress resource is automatically bound to it:

```
kubectl annotate ingressclass nginx ingressclass.kubernetes.io/is-default-class=true
```

If we observe the services exposed (`kubectl get services -n ingress-nginx`), we may notice that we can access the ingress-controller using a NodePort service since we do not have any external integration for a LoadBalancer service.

This apply will create a custom namespace. Let's investigate which pod are deployed:

```
kubectl get po -n ingress-nginx
```

**Exercise:** Let's inspect the frontend service of the application we just deployed:

- Can we access it directly? If yes, how?
- Let's create an Ingress with the following values. What parts of the infrastructure are missing to properly use it?

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
spec:
  rules:
   - host: www.example.local
     http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend
            port:
              number: 80
```

*This code snipped is attached to the PDF file as "./snippets/ingress.yaml"*

Let's try if the ingress configuration is effective:

```
kubectl get ingress

NAMESPACE   NAME              CLASS   HOSTS               ADDRESS         PORTS
default     minimal-ingress   nginx   www.example.local   172.16.134.52   80, 443
```

Despite the cluster has not external DNS integration, we can test the functionality of the ingress using curl:

```
curl -H "Host: www.example.local" ADDRESS:NODE_PORT
```

**Warning:** since the ingress controller is exposed through a NodePort service, it cannot be accessed directly on port 80. Instead, it is necessary to use the corresponding NodePort, which can be retrieved from the ingress controller service (`kubectl get services -n ingress-nginx`).

The `-H` option adds a custom headers to our request. We specified the '`www.example.local`' host, which enables to the ingress controller to redirect our request to the specific webserver in charge of that website using the *domain name* instead of the server *IP address*. This allows re-using the same IP address for many different web sites, served by different (and independent) web servers, as shown in Figure 7.1.

### 7.2.1 Ingress resources vs. virtual hosts

While sometimes it seems that the ingress controller looks similar to have a webserver such as `nginx` or `apache`) hosting multiple websites (i.e., *virtual hosts*), actually is this rather different.

In fact, the ingress controller will redirect the requests to multiple independent web servers, while the virtual host primitive will create multiple domains within the same web server. As a consequence, the Ingress controller enables each website to be updated (e.g., restarted, updated content, etc.) independently from the others. In addition, this allows the two different webservers to be controlled by different users.

Instead, with the virtual host primitive any modification required by a single webserver (e.g., a configuration update requiring a restart of the process) may impact also on the second webserver; in addition, we cannot allow two independent users to control the unique webserver.

On the other side, the ingress controller requires three different processes (i.e., pods) running and the necessity to split the TCP connection (the first connection terminates on the ingress controller, while a second TCP connection is established between the ingress and the actual web server). Instead, the virtual hosts requires a single process (i.e., pod) and a single TCP connection.

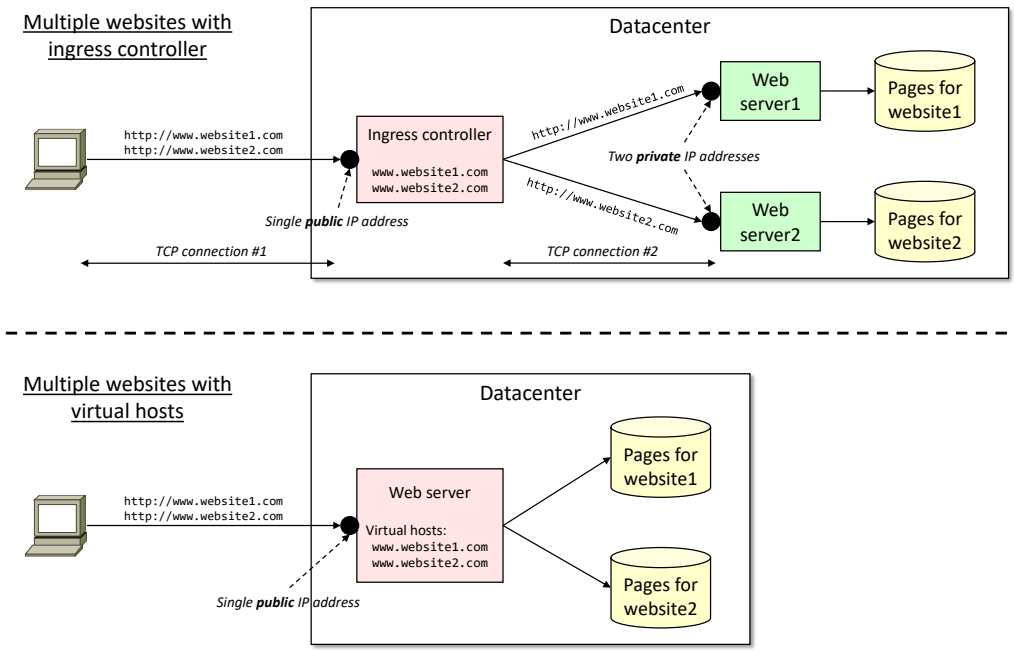


Figure 7.1: Different setup of two web servers sharing a single IP public address: with Kubernetes ingress controller (top), and with the virtual host primitive (bottom).