Practical Introduction to Kubernetes

Reorganised and adapted by Unai Arronategui for K3s,

from https://kubernetesbyexample.com/

Index

1. Pods

A pod is a collection of containers sharing a network and mount namespace and is the basic unit of deployment in Kubernetes. All containers in a pod are scheduled on the same node.

To launch a pod using the container image :

`quay.io/openshiftlabs/simpleservice:0.5.0`

and exposing a HTTP API on port 9876, execute:

```
$ kubectl run sise --image=quay.io/openshiftlabs/simpleservice:0.5.0 --port=9876
```

We can now see that the pod is running:

```
$ kubectl get pods -o wide
NAME                       READY     STATUS    RESTARTS   AGE
sise-574c4c79d6-rqxqg      1/1       Running   0          1m

$ kubectl top pod
NAME                    CPU(cores)   MEMORY(bytes)
sise-574c4c79d6-rqxqg   0m           12Mi

$ kubectl  describe  pod  sise-574c4c79d6-rqxqg  |  grep  IP:
IP:                     10.42.X.X
```

See section “Interaction with running Pods”, from “Kubectl Cheat Sheet”, to test from within the cluster, from other Pod

`$ kubectl run -i --tty busybox1 --image=busybox -- sh`

this sise pod is accessible via the pod IP *`10.42.X.X`*, which we’ve learned from the *`kubectl describe`* command above:

```
/ # wget -q -O - 10.42.X.X:9876/info
{"host": "10.42.X.X", "version": "0.5.0", "from": "10.42.Y.Y"}
```

If after login out from busybox1 , there is a need to log again in :

kubectl attach busybox1-7c8d57f869-6p5wq -c busybox1 -i -t

Note that basic *`kubectl` run* creates a deployment in *Kubernetes version 1.17.4. To* see them :

$ kubectl get deployment

So in order to get rid of the pod you have to execute *kubectl delete deployment sise.* *busybox1* deployement needs to be deleted in the same way.

Using configuration file

You can also create a pod from a **configuration file**. In this case the pod is running the already known simpleservice image container from above, along with a generic CentOS container, so **2 containers** are started in **1 Pod** :

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/pods/pod.yaml

$ kubectl get pods
NAME                          READY     STATUS    RESTARTS    AGE
twocontainers                 2/2       Running   0           7s
```

Read the configuration file (in a web browser) to study how is composed.

Now we can exec into the CentOS container (named *shell*) and access the simpleservice container, in the same Pod, on ***localhost***, as **they share network** configuration :

```
$ kubectl exec twocontainers -c shell -i -t -- bash
/# curl -s localhost:9876/info
{"host": "localhost:9876", "version": "0.5.0", "from": "127.0.0.1"}
```

Specify the resources field in the pod to influence how much CPU and/or RAM a container in a pod can use (here: 64MB of RAM and 0.5 CPUs):

```
$ kubectl create -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/pods/constraint-pod.yaml

$ kubectl describe pod constraintpod
...
Containers:
  sise:
    ...
    Limits:
      cpu:      500m
      memory:   64Mi
    Requests:
      cpu:      500m
      memory:   64Mi
...
```

Read the configuration file (in a web browser) to see how rsources have been modified, and learn more about resource constraints in Kubernetes via the docs here :

https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/

and here:

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

To remove all the pods created, just run:

```
$ kubectl delete pod twocontainers

$ kubectl delete pod constraintpod
```

To sum up, launching one or more containers (together) in Kubernetes is simple, however doing it directly as shown above comes with a serious limitation: you have to manually take care of keeping them running in case of a failure. A better way to supervise pods is to use deployments, giving you much more control over the life cycle, including rolling out a new version.

2. Labels

Labels are the mechanism you use to organize Kubernetes objects. A label is a key-value pair with certain restrictions concerning length and allowed values but **without any pre-defined meaning**. So you're free to choose labels as you see fit, for example, to express environments such as 'this pod is running in production' or ownership, like 'department X owns that pod'.

Let's create a pod that initially has one label (`env=development`):

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/labels/pod.yaml

$ kubectl get pods --show-labels
NAME        READY     STATUS     RESTARTS    AGE     LABELS
labelex     1/1       Running    0           10m     env=development
```

In above `get pods` command note the `--show-labels` option that output the labels of an object in an additional column.

You can add a label to the pod as:

```
$ kubectl label pods labelex owner=michael

$ kubectl get pods --show-labels
NAME         READY     STATUS     RESTARTS    AGE     LABELS
labelex      1/1       Running    0           16m     env=development,owner=michael
```

To use a label for filtering, for example to list only pods that have an `owner` that equals `michael`, use the `--selector` option:

```
$ kubectl get pods --selector owner=michael
NAME       READY     STATUS     RESTARTS    AGE
labelex    1/1       Running    0           27m
```

The `--selector` option can be abbreviated to `-l`, so to select pods that are labelled with `env=development`, do:

```
$ kubectl get pods -l env=development
NAME       READY     STATUS     RESTARTS    AGE
labelex    1/1       Running    0           27m
```

Oftentimes, Kubernetes objects also support set-based selectors. Let's launch another pod that has two labels (`env=production` and `owner=michael`):

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
```

```
specs/labels/anotherpod.yaml
```

Now, let’s list all pods that are either labelled with `env=development` or with `env=production`:

```
$ kubectl get pods -l 'env in (production, development)'
NAME           READY     STATUS    RESTARTS   AGE
labelex        1/1       Running   0          43m
labelexother   1/1       Running   0          3m
```

Other verbs also support label selection, for example, you could remove both of these pods with:

```
$ kubectl delete pods -l 'env in (production, development)'
```

Beware that this will destroy any pods with those labels.

You can also delete them directly, via their names, with:

```
$ kubectl delete pods labelex

$ kubectl delete pods labelexother
```

Note that **labels** are **not restricted** to **pods**. In fact you can apply them to all sorts of objects, such as nodes or services.

3. Nodes

In Kubernetes, nodes are the (virtual) machines where your workloads in shape of pods run. As a developer you typically don’t deal with nodes directly, however as an admin you might want to familiarize yourself with node operations.

To list available nodes in your cluster (note that the output will depend on the environment you’re using. This example is using the OpenShift Playground):

```
$ kubectl get nodes
NAME   STATUS   ROLES    AGE   VERSION
m      Ready    master   42m   v1.17.4+k3s1
w2     Ready    <none>   41m   v1.17.4+k3s1
w1     Ready    <none>   41m   v1.17.4+k3s1

$ kubectl top nodes
NAME   CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
m      130m         13%    488Mi           50%
w1     48m          4%     339Mi           35%
w2     29m          2%     267Mi           27%
```

One interesting task, from a developer point of view, is to make Kubernetes schedule a pod on a certain node. For this, we first need to label the node we want to target:

```
$ kubectl label nodes crc-rk2fc-master-0 shouldrun=here
node/crc-rk2fc-master-0 labeled
```

Now we can create a pod that gets scheduled on the node with the label `shouldrun=here`:

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/nodes/pod.yaml

$ kubectl get pods -o  wide
NAME            READY   STATUS    RESTARTS   AGE     IP          NODE     NOMINATED NODE     READINESS GATES
onspecificnode  1/1     Running   0          2m31s   10.42.1.5   w1       <none>             <none>
```

To learn more about a specific node, `crc-rk2fc-master-0` in our case, do:

```
$ kubectl describe node m
Name:               m
Roles:              master
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/instance-type=k3s
                    beta.kubernetes.io/os=linux
                    k3s.io/hostname=m
                    k3s.io/internal-ip=192.168.1.90
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=m
                    kubernetes.io/os=linux
                    node-role.kubernetes.io/master=true
                    node.kubernetes.io/instance-type=k3s
Annotations:        alpha.kubernetes.io/provided-node-ip: 192.168.1.90
                    flannel.alpha.coreos.com/backend-data:
{"VtepMAC":"fe:2d:45:28:d4:53"}
                    flannel.alpha.coreos.com/backend-type: vxlan
                    flannel.alpha.coreos.com/kube-subnet-manager: true
                    flannel.alpha.coreos.com/public-ip: 192.168.1.90
                    k3s.io/node-args:
                      ["server","--token","********","--flannel-iface","enp0s8","--bind-
address","192.168.1.90","--node-ip","192.168.1.90","--node-name","m","--...
                    k3s.io/node-config-hash:
NVVADHZNVLE3W2OQXJNHFTDR5Y46UG543H2H6XGKWEVZVIPGWDJQ====
                    k3s.io/node-env:
{"K3S_DATA_DIR":"/var/lib/rancher/k3s/data/6a3098e6644f5f0dbfe14e5efa99bb8fdf60d63cae89f
dffd71b7de11a1f1430"}
                    node.alpha.kubernetes.io/ttl: 0
                    volumes.kubernetes.io/controller-managed-attach-detach: true
CreationTimestamp:  Fri, 24 Apr 2020 12:43:00 +0200
Taints:             k3s-controlplane=true:NoExecute
Unschedulable:      false
Lease:
  HolderIdentity:  m
  AcquireTime:     <unset>
  RenewTime:       Fri, 24 Apr 2020 13:29:55 +0200
Conditions:
  Type                 Status  LastHeartbeatTime                 LastTransitionTime
Reason                       Message
  ----                 ------  -----------------                 ------------------
------                       -------
  NetworkUnavailable   False   Fri, 24 Apr 2020 12:43:15 +0200   Fri, 24 Apr 2020
12:43:15 +0200   FlannelIsUp                  Flannel is running on this node
  MemoryPressure       False   Fri, 24 Apr 2020 13:28:13 +0200   Fri, 24 Apr 2020
12:43:00 +0200   KubeletHasSufficientMemory   kubelet has sufficient memory available
  DiskPressure         False   Fri, 24 Apr 2020 13:28:13 +0200   Fri, 24 Apr 2020
12:43:00 +0200   KubeletHasNoDiskPressure     kubelet has no disk pressure
  PIDPressure          False   Fri, 24 Apr 2020 13:28:13 +0200   Fri, 24 Apr 2020
12:43:00 +0200   KubeletHasSufficientPID      kubelet has sufficient PID available
  Ready                True    Fri, 24 Apr 2020 13:28:13 +0200   Fri, 24 Apr 2020
12:43:10 +0200   KubeletReady                 kubelet is posting ready status. AppArmor
enabled
Addresses:
  InternalIP:  192.168.1.90
  Hostname:    m
```

```
Capacity:
  cpu:                1
  ephemeral-storage:  10098432Ki
  hugepages-2Mi:      0
  memory:             984228Ki
  pods:               110
Allocatable:
  cpu:                1
  ephemeral-storage:  9823754642
  hugepages-2Mi:      0
  memory:             984228Ki
  pods:               110
System Info:
  Machine ID:                 caad551e95584e119f46ac9e133d7187
  System UUID:                1A2BF7FC-C6D9-45D3-803C-8DF306BC8446
  Boot ID:                    38c5d79b-a990-4c74-8d8b-1487f3cff459
  Kernel Version:             4.15.0-96-generic
  OS Image:                   Ubuntu 18.04.4 LTS
  Operating System:           linux
  Architecture:               amd64
  Container Runtime Version:  containerd://1.3.3-k3s2
  Kubelet Version:            v1.17.4+k3s1
  Kube-Proxy Version:         v1.17.4+k3s1
PodCIDR:                      10.42.0.0/24
PodCIDRs:                     10.42.0.0/24
ProviderID:                   k3s://m
Non-terminated Pods:          (0 in total)
  Namespace                   Name    CPU Requests  CPU Limits  Memory Requests  Memory
Limits  AGE
  ---------                   ----    ------------  ----------  ---------------
-------------  ---
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests  Limits
  --------           --------  ------
  cpu                0 (0%)    0 (0%)
  memory             0 (0%)    0 (0%)
  ephemeral-storage  0 (0%)    0 (0%)
Events:
  Type     Reason                   Age                From          Message
  ----     ------                   ----               ----          -------
  Normal   Starting                 47m                kubelet, m    Starting kubelet.
  Warning  InvalidDiskCapacity      47m                kubelet, m    invalid capacity 0
on image filesystem
  Normal   NodeHasSufficientMemory  47m (x2 over 47m)  kubelet, m    Node m status is
now: NodeHasSufficientMemory
  Normal   NodeHasNoDiskPressure    47m (x2 over 47m)  kubelet, m    Node m status is
```

```
now: NodeHasNoDiskPressure
  Normal   NodeHasSufficientPID     47m (x2 over 47m)  kubelet, m     Node m status is
now: NodeHasSufficientPID
  Normal   NodeAllocatableEnforced  47m                kubelet, m     Updated Node
Allocatable limit across pods
  Normal   Starting                 47m                kube-proxy, m  Starting kube-
proxy.
  Normal   NodeReady                46m                kubelet, m     Node m status is
now: NodeReady
```

Note that there are more sophisticated methods than shown above, such as using affinity, to assign pods to nodes :

https://kubernetes.io/docs/concepts/configuration/assign-pod-node/

and depending on your use case, you might want to check those out as well.

4. Logging

Logging is one option to understand what is going on inside your applications and the cluster at large. Basic logging in Kubernetes makes the output a container produces available, which is a good use case for debugging. More advanced setups consider logs across nodes and store them in a central place, either within the cluster or via a dedicated (cloud-based) service.

Let’s create a pod called `logme` that runs a container writing to `stdout` and `stderr`:

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/logging/pod.yaml
```

To view the five most recent log lines of the gen container in the `logme` pod, execute:

```
$ kubectl logs --tail=5 logme -c gen
Thu Apr 27 11:34:40 UTC 2017
Thu Apr 27 11:34:41 UTC 2017
Thu Apr 27 11:34:41 UTC 2017
Thu Apr 27 11:34:42 UTC 2017
Thu Apr 27 11:34:42 UTC 2017
```

To stream the log of the gen container in the `logme` pod (like `tail -f`), do:

```
$ kubectl logs -f --since=10s logme -c gen
Thu Apr 27 11:43:11 UTC 2017
Thu Apr 27 11:43:11 UTC 2017
Thu Apr 27 11:43:12 UTC 2017
Thu Apr 27 11:43:12 UTC 2017
Thu Apr 27 11:43:13 UTC 2017
...
```

Note that if you wouldn’t have specified `--since=10s` in the above command, you would have gotten all log lines from the start of the container.

You can also view logs of pods that have already completed their lifecycle. For this we create a pod called `oneshot` that counts down from 9 to 1 and then exits. Using the `-p` option you can print the logs for previous instances of the container in a pod:

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/logging/oneshotpod.yaml
$ kubectl logs -p oneshot -c gen
9
8
7
6
```

```
5
4
3
2
1
```

You can remove the created pods with:

```
$ kubectl delete pod/logme pod/oneshot
```

5. API Server access

Sometimes it’s useful or necessary to directly access the Kubernetes API server, for exploratory or testing purposes.

In order to do this, one option is to proxy the API to your local environment (your host operating system system), using:

```
$ kubectl proxy --port=8080
Starting to serve on 127.0.0.1:8080
```

Now you can query the API (in a separate terminal session) like so:

```
$ curl http://localhost:8080/api/v1
{
  "kind": "APIResourceList",
  "groupVersion": "v1",
  "resources": [
    {
...
    {
      "name": "services/status",
      "singularName": "",
      "namespaced": true,
      "kind": "Service",
      "verbs": [
        "get",
        "patch",
        "update"
      ]
    }
  ]
}
```

Alternatively, without proxying, you can use `kubectl` directly as follows to achieve the same:

```
$ kubectl get --raw /api/v1
```

Further, if you want to explore the supported API versions and/or resources, you can use the following commands:

```
$ kubectl api-versions
admissionregistration.k8s.io/v1beta1
...
```

```
v1

$ kubectl api-resources
NAME                                  SHORTNAMES       APIGROUP         NAMESPACED
KIND
bindings          true          Binding
componentstatuses                     cs          false        ComponentStatus
configmaps                            cm          true         ConfigMap
...
```

hljs.initHighlightingOnLoad(); renderMathInElement(document.body); var doNotTrack = false;
if (!doNotTrack) {
    (function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
    (i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new Date();a=s.createElement(o),
    m=s.getElementsByTagName(o)
[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)
    })(window,document,'script','https://www.google-analytics.com/analytics.js','ga');
    ga('create', 'UA-34425776-2', 'auto');

    ga('send', 'pageview');
}
hljs.initHighlightingOnLoad(); renderMathInElement(document.body); var doNotTrack = false;
if (!doNotTrack) {
    (function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
    (i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new Date();a=s.createElement(o),
    m=s.getElementsByTagName(o)
[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)
    })(window,document,'script','https://www.google-analytics.com/analytics.js','ga');
    ga('create', 'UA-34425776-2', 'auto');

```
        ga('send', 'pageview');
}
```

6. Deployments

A **deployment** is a **supervisor** for pods, giving you fine-grained control over how and when a new pod version is rolled out as well as rolled back to a previous state.

Let’s create a deployment called `sise-deploy` that supervises two replicas of a pod as well as a replica set:

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/deployments/d09.yaml
```

You can have a look at the deployment, as well as the the replica set and the pods the deployment looks after like so:

```
$ kubectl get deploy
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
sise-deploy   2         2         2            2           10s

$ kubectl get rs
NAME                     DESIRED   CURRENT   READY     AGE
sise-deploy-3513442901   2         2         2         19s

$ kubectl get pods
NAME                           READY     STATUS    RESTARTS   AGE
sise-deploy-3513442901-cndsx   1/1       Running   0          25s
sise-deploy-3513442901-sn74v   1/1       Running   0          25s
```

Note the **naming** of the **pods** and **replica set**, **derived** from the deployment name.

At this point in time the `sise` containers running in the pods are configured to return the version `0.9`. Let’s verify that from within the cluster (using *`kubectl describe`* first to get the IP of one of the pods and, next, using a *busybox* pod as in the beginning):

```
/ # wget -q -O - 10.42.X.X:9876/info
{"host": "10.42.X.X:9876", "version": "0.9", "from": "10.42.Y.Y"}
```

Let’s now see what happens if we change that version to `1.0` in an updated deployment:

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/deployments/d10.yaml
deployment "sise-deploy" configured
```

Note that you could have used `kubectl edit deploy/sise-deploy` alternatively to achieve the same by manually editing the deployment.

What we now see is the rollout of two new pods with the updated version `1.0` as well as the two old pods with version `0.9` being terminated:

```
$ kubectl get pods
NAME                            READY     STATUS         RESTARTS    AGE
sise-deploy-2958877261-nfv28    1/1       Running        0           25s
sise-deploy-2958877261-w024b    1/1       Running        0           25s
sise-deploy-3513442901-cndsx    1/1       Terminating    0           16m
sise-deploy-3513442901-sn74v    1/1       Terminating    0           16m
```

Also, a new replica set has been created by the deployment:

```
$ kubectl get rs
NAME                      DESIRED    CURRENT    READY     AGE
sise-deploy-2958877261    2          2          2         4s
sise-deploy-3513442901    0          0          0         24m
```

Note that during the deployment you can check the progress using *`kubectl rollout status deploy/sise-deploy`*.

To verify that if the new `1.0` version is really available, you need to be **aware** that **pods** has **changed** the **name** and **IP**, so **again using** *`kubectl describe`* get the IP of one of the new pods,and access to them with *busybox* Pod:

```
/ # wget -q -O - 10.42.X.X:9876/info
{"host": "10.42.X.X:9876", "version": "1.0", "from": "10.42.Y.Y"}
```

A history of all deployments is available via:

```
$ kubectl rollout history deploy/sise-deploy
deployments "sise-deploy"
REVISION        CHANGE-CAUSE
1               <none>
2               <none>
```

If there are problems in the deployment Kubernetes will automatically roll back to the previous version, however you can also explicitly roll back to a specific revision, as in our case to revision 1 (the original pod version):

```
$ kubectl rollout undo deploy/sise-deploy --to-revision=1
deployment "sise-deploy" rolled back

$ kubectl rollout history deploy/sise-deploy
deployments "sise-deploy"
REVISION        CHANGE-CAUSE
2               <none>
3               <none>

$ kubectl get pods
```

```
NAME                           READY     STATUS    RESTARTS   AGE
sise-deploy-3513442901-ng8fz   1/1       Running   0          1m
sise-deploy-3513442901-s8q4s   1/1       Running   0          1m
```

At this point in time we’re back at where we started, with two new pods serving again version `0.9`.

Finally, to clean up, we remove the deployment and with it the replica sets and pods it supervises:

```
$ kubectl delete deploy sise-deploy
deployment "sise-deploy" deleted
```

See also the docs:

https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

for more options on deployments and when they are triggered.

7. Services

A service is an abstraction for pods, providing a stable, so called virtual IP (VIP) address. While pods may come and go and with it their IP addresses, a service **allows clients** to **reliably connect** to the containers running in the pod using the VIP. The `virtual` in VIP means it is not an actual IP address connected to a network interface, but its purpose is purely to forward traffic to one or more pods. Keeping the mapping between the VIP and the pods up-to-date is the job of kube-proxy, a process that runs on every node, which queries the API server to learn about new services in the cluster.

Let's create a pod supervised by an ReplicationController (RC) and a service along with it:

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/services/rc.yaml

$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/services/svc.yaml
```

Now we have the supervised pod running:

```
$ kubectl get pods -l app=sise
NAME           READY     STATUS    RESTARTS   AGE
rcsise-6nq3k   1/1       Running   0          57s

$ kubectl describe pod rcsise-6nq3k | less
Name:                   rcsise-6nq3k
Namespace:              default
Security Policy:        restricted
Node:                   localhost/192.168.99.100
Start Time:             Tue, 25 Apr 2017 14:47:45 +0100
Labels:                 app=sise
Status:                 Running
IP:                     10.42.2.11
Controllers:            ReplicationController/rcsise
Containers:
...
```

You can, from within the cluster, access the pod directly via its assigned IP `10.42.2.11`:

```
/ # wget -q -O - 10.42.2.11:9876/info
{"host": "10.42.2.11:9876", "version": "0.5.0", "from": "10.42.Y.Y"}
```

This is however, as mentioned above, **not advisable** since the **IPs** assigned to **pods may change**. **Hence**, enter the `simpleservice` we've created:

```
$ kubectl get svc
NAME              CLUSTER-IP       EXTERNAL-IP   PORT(S)                   AGE
simpleservice     172.30.228.255   <none>        80/TCP                    5m

$ kubectl describe svc simpleservice | less
Name:                   simpleservice
Namespace:              default
Labels:                 <none>
Selector:               app=sise
Type:                   ClusterIP
IP:                      10.43.170.9
Port:                   <unset> 80/TCP
Endpoints:              10.42.3.2:9876
Session Affinity:       None
No events.
```

The **service keeps track of** the **pods** it forwards traffic to **through** the **label**, in our case `app=sise`.

To observe their behavior from within the cluster, we start, again, the *busybox* Pod. And from there we can now access `simpleservice` with the **IP** of the **service**, not the Pod IP:

```
/ # wget -q -O - 10.43.48.66:80/info
{"host": "10.43.48.66:80", "version": "0.5.0", "from": "10.42.3.8"}
```

What makes the Virtual IP (VIP) `10.43.48.66` forward the traffic to the pod? The answer is: IPtables, which is essentially a long list of rules that tells the Linux kernel what to do with a certain IP package.

Looking at the rules that concern our service (executed on a cluster node - *vagrant ssh w1* -) yields:

```
vagrant@w1:~$ sudo iptables-save | egrep "simpleservice|KUBE-SVC-EZC6WLOVQADP4IAW|
KUBE-:KUBE-SEP-BKRJGF27PSUUPB5L - [0:0]
:KUBE-SVC-EZC6WLOVQADP4IAW - [0:0]
-A KUBE-SEP-BKRJGF27PSUUPB5L -s 10.42.3.2/32 -j KUBE-MARK-MASQ
-A KUBE-SEP-BKRJGF27PSUUPB5L -p tcp -m tcp -j DNAT --to-destination 10.42.3.2:9876
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.170.9/32 -p tcp -m comment --comment
"default/simpleservice: cluster IP" -m tcp --dport 80 -j KUBE-MARK-MASQ
-A KUBE-SERVICES -d 10.43.170.9/32 -p tcp -m comment --comment "default/simpleservice:
cluster IP" -m tcp --dport 80 -j KUBE-SVC-EZC6WLOVQADP4IAW
-A KUBE-SVC-EZC6WLOVQADP4IAW -j KUBE-SEP-BKRJGF27PSUUPB5L
```

Above you can see the four rules that `kube-proxy` has thankfully added to the routing table, essentially stating that TCP traffic to 10.43.170.9`:80` should be forwarded to

10.42.3.2:9876, which is our pod.

Let’s now add a second pod by scaling up the RC supervising it:

```
$ kubectl scale --replicas=2 rc/rcsise
replicationcontroller "rcsise" scaled

$ kubectl get pods -l app=sise
NAME           READY     STATUS    RESTARTS   AGE
rcsise-6nq3k   1/1       Running   0          15m
rcsise-nv8zm   1/1       Running   0          5s
```

When we now check the relevant parts of the routing table again, now in machine vagrant@m, we notice the addition of a bunch of IPtables rules:

```
vagrant@m:~$ sudo iptables-save | egrep "simpleservice|KUBE-SVC-EZC6WLOVQADP4IAW|KUBE-
UBE-SEP-VU2JGOHHBAZT2PPO"
:KUBE-SEP-BKRJGF27PSUUPB5L - [0:0]
:KUBE-SEP-VU2JGOHHBAZT2PPO - [0:0]
:KUBE-SVC-EZC6WLOVQADP4IAW - [0:0]
-A KUBE-SEP-BKRJGF27PSUUPB5L -s 10.42.3.2/32 -j KUBE-MARK-MASQ
-A KUBE-SEP-BKRJGF27PSUUPB5L -p tcp -m tcp -j DNAT --to-destination 10.42.3.2:9876
-A KUBE-SEP-VU2JGOHHBAZT2PPO -s 10.42.2.3/32 -j KUBE-MARK-MASQ
-A KUBE-SEP-VU2JGOHHBAZT2PPO -p tcp -m tcp -j DNAT --to-destination 10.42.2.3:9876
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.170.9/32 -p tcp -m comment --comment
"default/simpleservice: cluster IP" -m tcp --dport 80 -j KUBE-MARK-MASQ
-A KUBE-SERVICES -d 10.43.170.9/32 -p tcp -m comment --comment "default/simpleservice:
cluster IP" -m tcp --dport 80 -j KUBE-SVC-EZC6WLOVQADP4IAW
-A KUBE-SVC-EZC6WLOVQADP4IAW -m statistic --mode random --probability 0.50000000000 -j
KUBE-SEP-VU2JGOHHBAZT2PPO
-A KUBE-SVC-EZC6WLOVQADP4IAW -j KUBE-SEP-BKRJGF27PSUUPB5L
```

In above routing table listing we see rules for the newly created pod serving at 10.42.2.3:9876 as well as an additional rule:

```
-A KUBE-SVC-EZC6WLOVQADP4IAW -m statistic --mode random --probability 0.50000000000 -j
KUBE-SEP-VU2JGOHHBAZT2PPO
```

This causes the traffic to the service being equally split between our two pods by invoking the `statistics` module of IPtables.

You can remove all the resources created by doing:

```
$ kubectl delete svc simpleservice

$ kubectl delete rc rcsise
```

8. Service Discovery

Service discovery is the process of figuring out how to connect to a service. While there is a service discovery option based on environment variables available, the DNS-based service discovery is preferable. Note that DNS is a cluster add-on so make sure your Kubernetes distribution provides for one or install it yourself.

Let's create a service named `thesvc` and an RC supervising some pods along with it:

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/sd/rc.yaml

$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/sd/svc.yaml
```

Now we want to connect to the `thesvc` service from within the cluster, say, from another service. To simulate this, we create a jump pod in the same namespace (`default`, since we didn't specify anything else):

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/sd/jumpod.yaml
```

The DNS add-on will make sure that our service *`thesvc`* is available via the Fully Qualified Domain Name (DNS) *`thesvc.default.svc.cluster.local`* from other pods in the cluster.

We can directly connect to and consume the service (in the same namespace) like so:

```
$ kubectl exec -it jumpod -c shell -- curl http://thesvc/info
{"host": "thesvc", "version": "0.5.0", "from": "10.42.1.5"}
```

Note that the IP address `172.17.0.5` above is the cluster-internal IP address of the jump pod.

To access a service that is deployed in a different namespace than the one you're accessing it from, use a FQDN in the form `$SVC.$NAMESPACE.svc.cluster.local`.

Let's see how that works by creating:

1. a namespace `other`
2. a service `thesvc` in namespace `other`
3. an RC supervising the pods, also in namespace `other`

If you're not familiar with namespaces, check out the namespace examples first.

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/sd/other-ns.yaml

$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/sd/other-rc.yaml

$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/sd/other-svc.yaml
```

We’re now in the position to consume the service `thesvc` in namespace `other` from the `default` namespace (again via the jump pod):

```
$ kubectl exec -it jumpod -c shell -- curl http://thesvc.other/info
{"host": "thesvc.other", "version": "0.5.0", "from": "10.42.1.5"}
```

Summing up, DNS-based service discovery provides a flexible and generic way to connect to services across the cluster.

You can destroy all the resources created with:

```
$ kubectl delete pods jumpod

$ kubectl delete svc thesvc

$ kubectl delete rc rcsise

$ kubectl delete ns other
```

Keep in mind that **removing** a **namespace** will **destroy every resource inside**.

9. Port Forward

Kubernetes port forward by example

In the context of developing apps on Kubernetes it is often useful to quickly access a service from your local environment without exposing it using, for example, a load balancer or an ingress resource. In this case you can use port forwarding.

Let’s create an app consisting of a deployment and a service called `simpleservice`, serving on port `80`:

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/pf/app.yaml
```

Let’s say we want to access the `simpleservice` service from the local environment, say, your laptop, on port `8080`. So we forward the traffic as follows:

```
$ kubectl port-forward service/simpleservice 8080:80
Forwarding from 127.0.0.1:8080 -> 9876
Forwarding from [::1]:8080 -> 9876
```

We can see from above that the traffic gets eventually routed through the service to the pod serving on port 9876.

Now we can invoke the service locally like so (using a separate terminal session):

```
$ curl localhost:8080/info
{"host": "localhost:8080", "version": "0.5.0", "from": "127.0.0.1"}
```

Remember that port forwarding is not meant for production traffic but for development and experimentation.

You can remove the `simpleservice` with

```
$ kubectl delete service/simpleservice
$ kubectl delete deployment sise-deploy
```

10. Namespaces

Namespaces provide a scope for Kubernetes resources, carving up your cluster in smaller units. You can think of it as a workspace you're sharing with other users. Many resources such as pods and services are namespaced, while some, for example, nodes are not namespaced (but cluster-wide). As a developer you'd usually use an assigned namespace, however admins may wish to manage them, for example to set up access control or resource quotas.

Let's list all namespaces (note that the output will depend on the environment you're using):

```
$ kubectl get ns
NAME              STATUS   AGE
default           Active   20m
kube-system       Active   20m
kube-public       Active   20m
kube-node-lease   Active   20m
...
```

You can learn more about a namespace using the `describe` verb, for example:

```
$ kubectl describe ns default
Name:         default
Labels:       <none>
Annotations:  <none>
Status:       Active

No resource quota.

No LimitRange resource.
```

Let's now create a new namespace called `test` now:

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/ns/ns.yaml
namespace "test" created

$ kubectl get ns
NAME              STATUS   AGE
default           Active   24m
kube-system       Active   24m
kube-public       Active   24m
kube-node-lease   Active   24m
test              Active   29s
```

Alternatively, we could have created the namespace using the *`kubectl create namespace test` command*.

To launch a pod in the newly created namespace `test`, do:

```
$ kubectl apply --namespace=test -f https://raw.githubusercontent.com/openshift-
evangelists/kbe/master/specs/ns/pod.yaml
```

Note that using above method the namespace becomes a runtime property, that is, you can deploy the same pod or service, etc. into multiple namespaces (for example: dev and `prod`). Hard-coding the namespace directly in the `metadata` section like shown in the following is possible but causes less flexibility when deploying your apps:

```
apiVersion: v1
kind: Pod
metadata:
  name: podintest
  namespace: test
```

To list namespaced objects such as our pod `podintest`, run the following command:

```
$ kubectl get pods --namespace=test
NAME        READY     STATUS    RESTARTS   AGE
podintest   1/1       Running   0          16s
```

You can remove the namespace (**and everything inside**) with:

```
$ kubectl delete ns test
```

If you’re an admin, you might want to check out the docs :

https://kubernetes.io/docs/tasks/administer-cluster/namespaces/

for more info how to handle namespaces.

11. Volumes

A Kubernetes volume is essentially a directory accessible to all containers running in a pod. In contrast to the container-local filesystem, the data in volumes is preserved across container restarts. The medium backing a volume and its contents are determined by the volume type:

- node-local types such as *emptyDir* or *hostPath*
- file-sharing types such as *nfs*
- cloud provider-specific types like *awsElasticBlockStore*, *azureDisk*, or *gcePersistentDisk*
- distributed file system types, for example *glusterfs* or *cephfs*
- special-purpose types like *secret*, *gitRepo*

A special type of volume is *PersistentVolume*, which we will cover elsewhere.

Let's create a pod with two containers that use an emptyDir volume to exchange data:

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/volumes/pod.yaml

$ kubectl describe pod sharevol
Name:                   sharevol
Namespace:              default
...
Volumes:
  xchange:
    Type:       EmptyDir (a temporary directory that shares a pod's lifetime)
    Medium:
```

We first exec into one of the containers in the pod, c1, check the volume mount and generate some data:

```
$ kubectl exec -it sharevol -c c1 -- bash
[root@sharevol /]# mount | grep xchange
/dev/vda3 on /tmp/xchange type xfs (rw,relatime,seclabel,attr2,inode64,rjquota)
[root@sharevol /]# echo 'some data' > /tmp/xchange/data
```

When we now exec into c2, the second container running in the pod, we can see the volume mounted at `/tmp/data` and are able to read the data created in the previous step:

```
$ kubectl exec -it sharevol -c c2 -- bash
[root@sharevol /]# mount | grep /tmp/data
/dev/vda3 on /tmp/data type xfs (rw,relatime,seclabel,attr2,inode64,prjquota)
[root@sharevol /]# cat /tmp/data/data/
cat: /tmp/data/data/: Not a directory
[root@sharevol /]# cat /tmp/data/data
some data
```

Note that in each container you need to decide where to mount the volume and that for `emptyDir` you currently can not specify resource consumption limits.

You can remove the pod with:

```
$ kubectl delete pod/sharevol
```

As already described, this will destroy the shared volume and all its contents.

12. Persistent Volumes

Kubernetes persistent volumes by example

A persistent volume (PV) is a cluster-wide resource that you can use to store data in a way that it persists beyond the lifetime of a pod. The PV is not backed by locally-attached storage on a worker node but by networked storage system such as *EBS* or *NFS* or a distributed filesystem like *Ceph*.

Create one persistent volume first using:

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/pv/pv.yaml
```

In order to use a PV you need to claim it first, using a persistent volume claim (PVC). The PVC requests a PV with your desired specification (size, speed, etc.) from Kubernetes and binds it then to a pod where you can mount it as a volume. So let’s create such a PVC, asking Kubernetes for 1 GB of storage using the default storage class:

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/pv/pvc.yaml

$ kubectl get pvc
NAME      STATUS   VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS    AGE
myclaim   Bound    pvc-27fed6b6-3047-11e9-84bb-12b5519f9b58   1Gi        RWO
gp2-encrypted   18m
```

To understand how the persistency plays out, let’s create a deployment that uses above PVC to mount it as a volume into `/tmp/persistent`:

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/pv/deploy.yaml
```

Now we want to test if data in the volume actually persists. For this we find the pod managed by above deployment, exec into its main container and create a file called `data` in the `/tmp/persistent` directory (where we decided to mount the PV):

```
$ kubectl get po
NAME                         READY   STATUS    RESTARTS   AGE
pv-deploy-69959dccb5-jhxx    1/1     Running   0          16m

$ kubectl exec -it pv-deploy-69959dccb5-jhxxw -- bash
bash-4.2$ touch /tmp/persistent/data
bash-4.2$ ls /tmp/persistent/
data  lost+found
```

It’s time to destroy the pod and let the deployment launch a new pod. The expectation is that the PV is available again in the new pod and the data in `/tmp/persistent` is still present. Let’s check that:

```
$ kubectl delete po pv-deploy-69959dccb5-jhxxw
pod pv-deploy-69959dccb5-jhxxw deleted

$ kubectl get po
NAME                         READY   STATUS    RESTARTS   AGE
pv-deploy-69959dccb5-kwrrv   1/1     Running   0          16m

$ kubectl exec -it pv-deploy-69959dccb5-kwrrv -- bash
bash-4.2$ ls /tmp/persistent/
data  lost+found
```

And indeed, the `data` file and its content is still where it is expected to be.

Note that the default behavior is that even when the deployment is deleted, the PVC (and the PV) continues to exist. This storage protection feature helps avoiding data loss. Once you’re sure you don’t need the data anymore, you can go ahead and delete the PVC and with it eventually destroy the PV:

```
$ kubectl delete pvc myclaim
persistentvolumeclaim "myclaim" deleted
```

The types of PV available in your Kubernetes cluster depend on the environment (on-premise or public cloud). Check out the Stateful Kubernetes reference site if you want to learn more about this topic :

https://stateful.kubernetes.sh/#storage

13. StatefulSet

If you have a stateless app you want to use a deployment. However, for a stateful app you might want to use a StatefulSet. Unlike a deployment, the `StatefulSet` provides certain guarantees about the identity of the pods it is managing (that is, predictable names) and about the startup order. Two more things that are different compared to a deployment: for network communication you need to create a headless services and for persistency the `StatefulSet` manages a persistent volume per pod.

In order to see how this all plays together, we will be using an educational Kubernetes-native NoSQL datastore.

Let’s start with creating the stateful app, that is, the `StatefulSet` along with the persistent volumes and the headless service:

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/mehdb/master/
app.yaml
```

After a minute or so, you can have a look at all the resources that have been created:

```
$ kubectl get sts,po,pvc,svc
NAME                     DESIRED   CURRENT   AGE
statefulset.apps/mehdb   2         2         1m

NAME            READY    STATUS    RESTARTS   AGE
pod/mehdb-0     1/1      Running   0          1m
pod/mehdb-1     1/1      Running   0          56s

NAME                                 STATUS    VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS   AGE
persistentvolumeclaim/data-mehdb-0   Bound     pvc-bc2d9b3b-310d-11e9-aeff-123713f594ec
1Gi        RWO            ebs            1m
persistentvolumeclaim/data-mehdb-1   Bound     pvc-d4b7620f-310d-11e9-aeff-123713f594ec
1Gi        RWO            ebs            56s

NAME            TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
service/mehdb   ClusterIP   None         <none>        9876/TCP   1m
```

Now we can check if the stateful app is working properly. To do this, we use the `/status` endpoint of the headless service `mehdb:9876` and since we haven’t put any data yet into the datastore, we’d expect that `0` keys are reported:

```
$ kubectl run -it --rm jumpod --restart=Never --image=quay.io/openshiftlabs/jump:0.2 --
curl mehdb:9876/status?level=full
If you don't see a command prompt, try pressing enter.
```

```
0
pod "jumpod" deleted
```

And indeed we see `0` keys being available, reported above.

Note that sometimes a `StatefulSet` is not the best fit for your stateful app. You might be better off defining a custom resource along with writing a custom controller to have finer-grained control over your workload.

hljs.initHighlightingOnLoad(); renderMathInElement(document.body); var doNotTrack = false;
if (!doNotTrack) {
    (function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){
    (i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new Date();a=s.createElement(o),
    m=s.getElementsByTagName(o)
[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)

})(window,document,'script','https://www.google-analytics.com/analytics.js','ga');
    ga('create', 'UA-34425776-2', 'auto');

    ga('send', 'pageview');
}

14. Jobs

A job in Kubernetes is a supervisor for pods carrying out batch processes, that is, a process that runs for a certain time to completion, for example a calculation or a backup operation.

Let’s create a job called `countdown` that supervises a pod counting from 9 down to 1:

```
$ kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/
specs/jobs/job.yaml
```

You can see the job and the pod it looks after like so:

```
$ kubectl get jobs
NAME                      DESIRED   SUCCESSFUL   AGE
countdown                 1         1            5s

$ kubectl get pods
NAME                READY     STATUS      RESTARTS   AGE
countdown-qkjx8     0/1       Completed   0          2m17s
```

To learn more about the status of the job, do:

```
$ kubectl describe jobs/countdown
Name:             countdown
Namespace:        default
Selector:         controller-uid=4960c8be-6a3f-11ea-84fd-0242ac11002a
Labels:           controller-uid=4960c8be-6a3f-11ea-84fd-0242ac11002a
                  job-name=countdown
Annotations:      kubectl.kubernetes.io/last-applied-configuration:
                    {"apiVersion":"batch/v1","kind":"Job","metadata":{"annotations":
{},"name":"countdown","namespace":"default"},"spec":{"template":{"metadata...
Parallelism:      1
Completions:      1
Start Time:       Fri, 20 Mar 2020 00:11:03 +0000
Completed At:     Fri, 20 Mar 2020 00:11:12 +0000
Duration:         9s
Pods Statuses:    0 Running / 1 Succeeded / 0 Failed
Pod Template:
  Labels:  controller-uid=4960c8be-6a3f-11ea-84fd-0242ac11002a
           job-name=countdown
  Containers:
   counter:
    Image:        centos:7
    Port:         <none>
    Host Port:    <none>
    Command:
      bin/bash
      -c
      for i in 9 8 7 6 5 4 3 2 1 ; do echo $i ; done
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age    From            Message
  ----    ------            ----   ----            -------
  Normal  SuccessfulCreate  3m18s  job-controller  Created pod: countdown-qkjx8
```

And to see the output of the job via the pod it supervised, execute:

```
kubectl logs countdown-qkjx8
9
8
7
6
5
4
3
2
1
```

To clean up, use the `delete` verb on the job object which will remove all the supervised pods:

```
$ kubectl delete job countdown
job "countdown" deleted
```

Note that there are also more advanced ways to use jobs, for example, by utilizing a work queue or scheduling the execution at a certain time via cron jobs.

```
        ga('create', 'UA-34425776-2', 'auto');

        ga('send', 'pageview');
}
```