Introduction to Kubernetes

A very quick and incomplete introduction to Kubernetes, that will however, help you navigate the thing while experimenting on your personal machine(s).

A very quick introduction to Kubernetes

For this tutorial, it is assumed that you have installed the Docker Desktop and have enabled Kubernetes. As far as resources go, 3GB of RAM, 2CPUs and 1G of swap should suffice. It should also be able to follow this tutorial with minor adjustments with any other Kubernetes.

You can also use Rancher Desktop for the purposes of this tutorial.

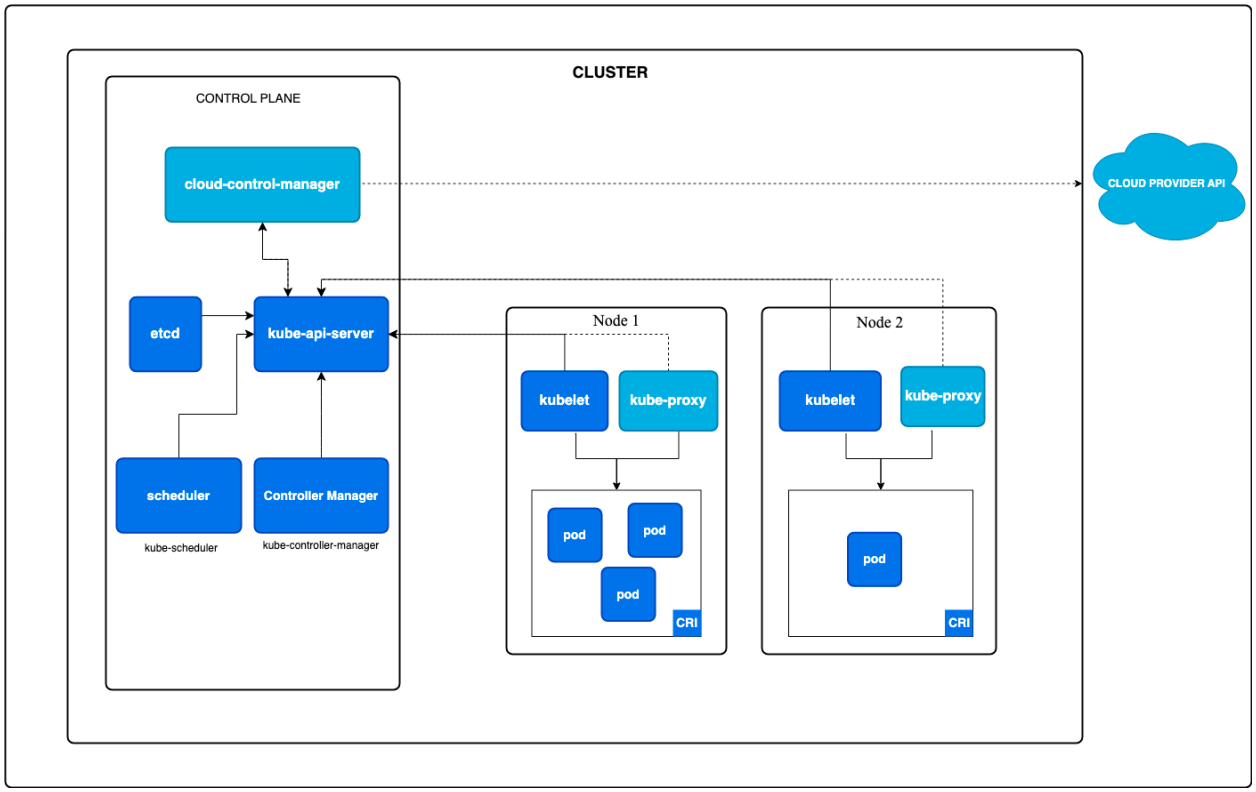
The orchestration wars

Once containers took off the development world, it became apparent that they were also the tool to deploy stuff on the cloud and in any environment, be it Dev, Test, Staging, UAT, Production. A number of choices arose, like Swarm, Nomad, Mesos and Rancher1. Kubernetes won.

What is Kubernetes?

Kubernetes is a system initially developed by Google leveraging their experience from their internal tool to deploy workloads (from VMs, to containers) called Borg. If you are interested to see how Borg works, you can see Bosh (`Bosh == Borg++`, in the same way that `HAL == IBM++`). Kubernetes is written in Go, with only one tiny bit of code in C (initially it was written in Java, but this code was never publicly released).

Kubernetes is a set of programs that is deployed on a number of machines (they can be bare metal, VMs or mixed) which are called a (Kubernetes) cluster. For all practical purposes concerning this tutorial all machines in the cluster run Linux and once Kubernetes is installed on them, they cannot be used for anything else. Kubernetes takes over. Here is what a Kubernetes cluster looks like:



The control plane is where we, the operators of the Kubernetes cluster, send our wishes to be completed. It is usually one to three nodes which colloquially are called masters. Because of Raft's latency, they are never more than 3.

In the control plane node you can see a component named etcd. This is a key-value store that also implements Raft and stores the state of the cluster. It is the source of truth. If the etcd dies, the cluster continues to work, but cannot do anything, like update workloads or even restart an application. Etcd maintenance is so important that it is the sole reason that cloud providers offer managed Kubernetes services.

Worker nodes, run the actual workload. The traffic that is to reach our applications is directed to those nodes and they and only they run our applications.

Of course a node in the cluster can fulfill multiple roles, be it master, worker, etcd, any two of the three, all all three (as is the case with the Docker Desktop VM that runs on your machine).

Based on the diagram above the one fundamental truth must be mentioned:

Kubernetes has been built to service:

- ports 80/tcp and 443/tcp
- protocols HTTP, HTTPS and gRPC
- in a completely firewalled environment

Anything else can be achieved, but it is a show of dexterity, or requires the deployment of specific software like metallb (and / or hardware)

The Kubernetes promise

What Kubernetes promises is that it will make our lives immediately easier. It does not. It hides complexity. It never reduces it, it simply hides it from the eyes of many people. However, it will always do its best, with our assistance, to keep our workloads running and as highly available as possible.

What is kubectl?

kubectl is the command line tool that we use in order to communicate with the `kube-api-server`. It is how we submit our wishes to the Kubernetes cluster and how we query the cluster about the status of our workloads.

Via `kubectl` we submit to Kubernetes files written in YAML. That is why Kubernetes people call themselves YAML Engineers.

YAML is used because Kubernetes can be seen as an Object Oriented System with a pre-existing class hierarchy of objects. We ask Kubernetes to create instances of those objects in order to run our workloads, by describing them in YAML files. The most important objects for this tutorial are Pods, Deployments, Services and Ingress. If you know what these do, you can achieve 90% of your work without hassle. If you want to add your own distinct object classes in your Kubernetes cluster, you need to learn about the operator pattern (outside the scope of this tutorial).

`Kubectl` knows how to communicate with your cluster by reading the file `~/.kube/config` or the one pointed by the `KUBECONFIG` environment variable, or the one pointed by the `--kubeconfig` command line switch. Docker Desktop pre-configures this file when running. Feel free to inspect it and discuss its contents.

Our first interaction

Let's start talking to our Kubernetes cluster then:

```
% kubectl get node
NAME              STATUS   ROLES           AGE   VERSION
docker-desktop   Ready   control-plane   9d v1.29.1
```

We asked the cluster to tell us what nodes are comprising it. We get the response that it is a single node cluster, running version v1.29.1. Let's find something more about this node then

```
% kubectl describe nodes docker-desktop
Name:              docker-desktop
Roles:             control-plane
Labels:            beta.kubernetes.io/arch=arm64
                   beta.kubernetes.io/os=linux
                   kubernetes.io/arch=arm64
                   kubernetes.io/hostname=docker-desktop
                   kubernetes.io/os=linux
                   node-role.kubernetes.io/control-plane=
                   node.kubernetes.io/exclude-from-external-load-balancers=
:
:
Addresses:
  InternalIP:  192.168.65.3
  Hostname:  docker-desktop
Capacity:
  cpu:             2
  ephemeral-storage:  61202244Ki
  hugepages-1Gi:   0
:
:
```

We get a long list of output containing lots of useful information (different parts are of concern to different people). Labels are of note here, as we can use them to label and select between objects in Kubernetes. We can label almost any object. Examples of querying with labels

```
% kubectl get node -l kubernetes.io/os
NAME              STATUS   ROLES           AGE   VERSION
docker-desktop   Ready   control-plane   9d v1.29.1

% kubectl get node -l kubernetes.io/os=linux
NAME              STATUS   ROLES           AGE   VERSION
docker-desktop   Ready   control-plane   9d v1.29.1

% kubectl get node -l kubernetes.io/os=windows
No resources found
```

Running our first Pod

Pods are the smallest kind of workload that we can schedule inside a cluster. A pod can run one or more containers that share a common network space. That is why, confusingly, sometimes they are used interchangeably. A Pod always runs in a single node. Therefore, if a Pod runs two or more containers, those all run in the same node and it cannot be broken in a way such that one container runs on worker A and another on worker B. They are always in the same node. A service from container A in a Pod can communicate with a service from container B in the same Pod by calling `127.0.0.1:port` (the port where B listens to). They can also communicate via IPC or shared volumes.

Here is what a very simple Pod looks like in YAML

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
      image: nginx:1.14.2
      ports:
      - containerPort: 80
```

Let's submit it

```
% kubectl apply -f nginx-pod.yaml
pod/nginx created

% kubectl get pod
NAME   READY   STATUS           RESTARTS   AGE
nginx   0/1  ContainerCreating   0          3s

% kubectl get pod
NAME   READY   STATUS     RESTARTS   AGE
nginx   1/1  Running   0        23s
```

Since our YAML file was without errors, it got accepted by the kube-api-server and then Kubernetes took it upon itself to download the `nginx:1.14.2` image from dockerhub and start the Pod. We can describe the pod and see what other information we can get from it

```
% kubectl describe pod nginx
Name:             nginx
Namespace:        default
Priority:         0
Service Account:  default
Node:             docker-desktop/192.168.65.3
Start Time:       Wed, 28 Feb 2024 10:57:28 +0200
Labels:           <none>
```

```
Annotations:      <none>
Status:           Running
IP:               10.1.0.18
IPs:
  IP:  10.1.0.18
Containers:
  nginx:
      Container ID:
docker://8739c63ad20735339e2b8d772081eb55a6de4903fe92d44407351fdfe3d7af91
      Image:      nginx:1.14.2
:
:
```

There are no labels in our Pod, but we could have added some during creation, or we can even label it now

```
% kubectl label pod nginx owner=adamo
pod/nginx labeled

% kubectl describe pod nginx
:
```

We can kill the Pod with

```
% kubectl delete pod nginx
pod "nginx" deleted
```

But wait! You said that Kubernetes promises to keep and scale my application. Now it died and it never got restarted. Yes, this is because a Pod is the building block for high availability and scaling, and by itself it offers nothing of these. It needs to be part of more complex objects.

Enter the ReplicaSet

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx
  labels:
      app: nginx
spec:
  replicas: 3
  selector:
      matchLabels:
      app: nginx
  template:
```

```
      metadata:
      labels:
      app: nginx
      spec:
      containers:
      - name: nginx
      image: nginx:1.14.2
```

Let's see now what we got

```
% kubectl get pod
NAME         READY   STATUS     RESTARTS   AGE
nginx-cflhp   1/1  Running   0        3s
nginx-jgv7z   1/1  Running   0        3s
nginx-nps96   1/1  Running   0        3s

% kubectl get replicasets
NAME   DESIRED   CURRENT   READY   AGE
nginx   3          3     3     41s
```

We see that we have a ReplicaSet defined, with 3 pods running and we can for example try and delete one and see what happens

```
% kubectl delete pod nginx-cflhp; kubectl get pod
pod "nginx-cflhp" deleted
NAME         READY   STATUS           RESTARTS   AGE
nginx-jgv7z   1/1  Running           0           2m1s
nginx-nps96   1/1  Running           0           2m1s
nginx-qq5zm   0/1  ContainerCreating   0          1s
```

ReplicaSets are nice, but we won't use them much in the future. They are briefly mentioned because they form the basis of the most important object when we deploy applications

Deployment has entered the chat

Deployments are the most important Kubernetes objects. Understanding them, means you can understand almost anything else. Roughly they are ReplicaSets with extra metadata. You describe a *desired state* in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

Let's create a deployment of a simple TODO application

```
% kubectl create deployment todo --image yiorgos/docker-nodejs-sample -o yaml
--dry-run=client > deployment-todo.yaml
```

The above command can be used in order to create a boilerplate for any deployment. The file produced. after some editing looks like

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
      app: todo
  name: todo
spec:
  replicas: 1
  selector:
      matchLabels:
      app: todo
  template:
      metadata:
      labels:
      app: todo
      spec:
      containers:
      - image: yiorgos/docker-nodejs-sample
      name: docker-nodejs-sample
```

Observe the YAML file above. It contains four basic sections that are common to *all* Kubernetes YAML files: `apiVersion`, `kind`, `metadata` and `spec`. This is what we get after applying the file

```
% kubectl apply -f deployment-todo.yaml
deployment.apps/todo created

% kubectl get pod
NAME                    READY   STATUS    RESTARTS   AGE
todo-7cfb886964-77zsd   1/1     Running   0          52s
```

We can scale the deployment from the command line for example with

```
% kubectl scale deployment todo --replicas 2
deployment.apps/todo scaled

% kubectl get pod
NAME                    READY   STATUS              RESTARTS   AGE
todo-7cfb886964-77zsd   1/1     Running             0          2m46s
todo-7cfb886964-vz67n   0/1     ContainerCreating   0          2s
```

We can change the image of the application if we want to use something else (a new application version in real life). Let's use `yiorgos/docker-nodejs-sample:1` and see what happens

```
% kubectl apply -f deployment-todo.yaml
deployment.apps/todo configured
```

```
% kubectl get deployments.apps
NAME   READY   UP-TO-DATE   AVAILABLE   AGE
todo   1/1   1            1           7m37s

% kubectl get rs
NAME               DESIRED   CURRENT   READY   AGE
todo-645768dcff    1         1         1       9s
todo-7cfb886964    0         0         0       7m41s
```

What happened here? A new active ReplicaSet was created within the deployment and deployed version `:1` of the image. The previous deployment has been kept "standby" and we can rollback

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

% kubectl rollout undo deployment todo
deployment.apps/todo rolled back

% kubectl rollout history deployment todo
deployment.apps/todo
REVISION  CHANGE-CAUSE
2         <none>
3         <none>
```

More about kubectl rollout undo here.

Accessing my deployment from outside of Kubernetes

`Kubectl` allows for a proxy in order for us to be able to access what we launched. We can make it accessible to our browser at `http://127.0.0.1:3000` by running

```
% kubectl port-forward deployments/todo 3000
Forwarding from 127.0.0.1:3000 -> 3000
Forwarding from [::1]:3000 -> 3000
Handling connection for 3000
Handling connection for 3000
Handling connection for 3000
Handling connection for 3000
Handling connection for 3000
Handling connection for 3000
Handling connection for 3000
```

We will see later how to properly access something, but this allows us to connect to a deployment for quick debugging and tests.

Service

Let's run also this deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
      app: docker-names
  name: docker-names
spec:
  replicas: 1
  selector:
      matchLabels:
      app: docker-names
  template:
      metadata:
      labels:
      app: docker-names
      spec:
      containers:
      - image: yiorgos/docker_names
      name: docker-names
      ports:
      - containerPort: 8000
```

After applying, query this pod to figure out the IP address

```
% kubectl describe pod docker-names-5c7c6786c8-5cc5w | grep IP
IP:               10.1.0.31
IPs:
  IP:       10.1.0.31
```

This means that anyone *inside the cluster* can access it:

```
% kubectl run -it curl --image curlimages/curl -- sh
If you don't see a command prompt, try pressing enter.
~ $ curl 10.1.0.31:8000/name
{"n":3,"host":"docker-names-5c7c6786c8-5cc5w","name":"musing_keldysh"}
```

This means that we can access this service from within the cluster by directly accessing the the Pod's IP address, but this will change if the pod gets deleted due to malfunction, upgrade or other systemic reasons. It is impossible to demand that the IP address be known to people and applications all the time. And what about the case where we have scaled an application to hundreds of Pods?

Inside the cluster means that you are logged in where the kubernetes components run. In the case of your laptop this means within the VM that runs Kubernetes. Something like `rdctl shell` or `wsl -d rancher-desktop` for example.

This is where services come into play. Service objects come in many flavors, but we are only interested in the `clusterIP` (the default if a type is not stated) ones for now. A Service is a method for exposing a network application that is running as one or more Pods in your cluster. So let's expose the service like that

```
apiVersion: v1
kind: Service
metadata:
  name: docker-names
spec:
  selector:
      app: docker-names
  ports:
      - protocol: TCP
      port: 8000
      targetPort: 8000
```

We can now apply the file and see what else happens

```
% kubectl apply -f service-docker-names.yaml
service/docker-names created

% kubectl get svc
NAME         TYPE         CLUSTER-IP         EXTERNAL-IP   PORT(S)    AGE
docker-names   ClusterIP   10.100.191.181   <none>      8000/TCP   98s
kubernetes   ClusterIP   10.96.0.1          <none>      443/TCP      9d

% kubectl attach -it curl
If you don't see a command prompt, try pressing enter.
~ $ curl docker-names:8000/name
{"n":4,"host":"docker-names-5c7c6786c8-5cc5w","name":"gallant_feynman"}
```

What we learned here? First, that we can attach to an already running interactive container we had created previously and second, that we can access via DNS, the service name that we created.

Keep in mind that a Service object connects us to Pods, not Deployments. This is important to remember. The selector in the service is about a label we attach to a Pod.

Intermission: Does all this seem nerve-racking and frustrating? Nothing that a little comic book cannot help with: [The illustrated children's guide to Kubernetes](#).

This also means that we can access the service via port forwarding in the same way we did before with the deployment

```
% kubectl port-forward services/docker-names 8000
Forwarding from 127.0.0.1:8000 -> 8000
Forwarding from [::1]:8000 -> 8000
Handling connection for 8000
Handling connection for 8000
Handling connection for 8000
Handling connection for 8000
```

Accessing Pod / container logs

We can see the container logs for a running Pod with `kubectl logs`:

```
% kubectl logs docker-names-5c7c6786c8-5cc5w
INFO: Will watch for changes in these directories: ['/usr/app']
INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO: Started reloader process [1] using WatchFiles
INFO: Started server process [8]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: 10.1.0.1:50274 - "GET / HTTP/1.1" 404 Not Found
INFO: 10.1.0.1:40024 - "GET /names HTTP/1.1" 404 Not Found
INFO: 10.1.0.1:40034 - "GET /random HTTP/1.1" 404 Not Found
INFO: 10.1.0.1:41698 - "GET /docker/random HTTP/1.1" 404 Not Found
INFO: 10.1.0.1:39806 - "GET /docker HTTP/1.1" 404 Not Found
INFO: 10.1.0.1:45036 - "GET /name HTTP/1.1" 200 OK
INFO: 10.1.0.1:45038 - "GET /name HTTP/1.1" 200 OK
INFO: 10.1.0.1:39898 - "GET /doc HTTP/1.1" 404 Not Found
INFO: 10.1.0.34:34976 - "GET /name HTTP/1.1" 200 OK
INFO: 10.1.0.34:60390 - "GET /name HTTP/1.1" 200 OK
INFO: 127.0.0.1:34800 - "GET / HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:34800 - "GET /favicon.ico HTTP/1.1" 200 OK
INFO: 127.0.0.1:56550 - "GET /name HTTP/1.1" 200 OK
INFO: 127.0.0.1:56550 - "GET /favicon.ico HTTP/1.1" 200 OK
```

Of course more proper log management can be done with the help of tools like [fluentd](#) and [graylog](#).

Getting "inside" the Pod / container

Sometimes we feel the need to get “inside” the container, just like we would ssh into a virtual machine where an application is running. Example

```
% kubectl exec -it docker-names-5c7c6786c8-5cc5w -- ba
sh
root@docker-names-5c7c6786c8-5cc5w:/usr/app# ls
__pycache__  app.py  names_generator.py  requirements.txt
root@docker-names-5c7c6786c8-5cc5w:/usr/app#
```

As you can see, without interrupting the container, we gained access inside it and we could look around and figure out stuff

NodePorts

So far we saw that we can use `kubectl port-forward` in order to access a Pod, a Deployment or a Service that exists inside the Kubernetes. While this is handy, it is not the most convenient of things when working on our machines, and here the `NodePort` type of Service object can help. Furthermore, using nodeports, one can expose outside Kubernetes services that are not HTTP, HTTPS and gRPC. To create a nodeport we apply the following YAML

```
apiVersion: v1
kind: Service
metadata:
  name: docker-names
spec:
  type: NodePort
  selector:
      app: docker-names
  ports:
      - protocol: TCP
      port: 8000
      targetPort: 8000
      nodePort: 30123
```

The observant will note that the two differences with the previous service definition are the `type: NodePort` and the `nodePort: 30123` lines. Let’s apply this and see what happens

```
% kubectl apply -f nodeport-docker-names.yaml
service/docker-names configured

% kubectl get svc
NAME         TYPE         CLUSTER-IP         EXTERNAL-IP   PORT(S)          AGE
docker-names   NodePort  10.100.191.181   <none>        8000:30123/TCP   32m
kubernetes   ClusterIP   10.96.0.1          <none>       443/TCP           9d
```

If now, from our terminal we `curl http://127.0.0.1:30123/name` we will get a response (do it now). Also, the service is still available “inside” Kubernetes as `docker-names:30123`. Had we

omitted the `nodePort: 30123` line in the YAML, Kubernetes would have chosen a random port between `30000` and `32767` to expose the internal service “to the outside”.

We can now do a small cleanup and continue with our TODO application

```
% kubectl delete deployments.apps docker-names
deployment.apps "docker-names" deleted

% kubectl delete service docker-names
service "docker-names" deleted
```

Environment variables

If we check the logs for the TODO application we will see that it stores data in an SQLite file

```
% kubectl logs todo-6db458f487-bpqwm
Using sqlite database at /tmp/todo.db
Listening on port 3000
```

And by reviewing the source code we see that this location is governed by the `SQLITE_DB_LOCATION` environment value

```
// src/persistence/sqlite.js
:
const location = process.env.SQLITE_DB_LOCATION || '/tmp/todo.db';
:
```

Should we like to change the location of this file, we need to set the environment variable then. This is doable fairly easily as can be seen here

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
      app: todo
  name: todo
spec:
  replicas: 1
  selector:
      matchLabels:
      app: todo
  template:
      metadata:
      labels:
      app: todo
      spec:
      containers:
      - image: yiorgos/docker-nodejs-sample
```

```
      name: docker-nodejs-sample
      ports:
      - containerPort: 3000
      env:
      - name: SQLITE_DB_LOCATION
      value: "/usr/app/todo.db"
```

Always put the value in double-quotes (or single quotes if it is a JSON value) unless you know exactly what you are doing. Without double quotes YAML interprets a value of `1025e36` as a number (`1025 * 10^36`) and not as the beginning of a commit hash for example. What can possibly go wrong? So double, or single quotes all the time.

Data Persistence

Pods and their containers are ephemeral. This means that once stopped or restarted all locally generated data will be wiped out. This is not what we want for our TODO application. We need the list to remain in place. For this reason we need to have some volume persistence and work with it. Storage in Kubernetes is a black art that requires one to have intimate knowledge of the solutions involved, even when using solutions like Longhorn. But for the developer's machine, there is thankfully the solution of hostPath. We can now try to deploy our TODO application with a volume to make it restart resistant

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
      app: todo
  name: todo
spec:
  replicas: 1
  selector:
      matchLabels:
      app: todo
  template:
      metadata:
      labels:
      app: todo
      spec:
      volumes:
      - name: todo
      hostPath:
      path: /Users/adamo/tmp/todo
      type: Directory
      containers:
      - image: yiorgos/docker-nodejs-sample:1
      name: docker-nodejs-sample
```

```
      ports:
      - containerPort: 3000
      volumeMounts:
      - name: todo
      mountPath: /db
      env:
      - name: SQLITE_DB_LOCATION
      value: "/db/todo.db"
---
apiVersion: v1
kind: Service
metadata:
  name: todo
spec:
  type: NodePort
  selector:
      app: todo
  ports:
      - protocol: TCP
      port: 3000
      targetPort: 3000
```

What do we learn here? First, we can declare multiple Kubernetes objects in the same YAML file, provided that we separate them with `---`. Second, we take advantage of how Docker Desktop allows us to mount an external directory as a volume in our Deployment to achieve persistence. Similar methods are used by other distributions also.

We also exposed the TODO application via NodePort. Access the application and add some tasks. To delete the pod or just restart the deployment and verify that the data persists

```
% kubectl rollout restart deployment todo
deployment.apps/todo restarted
```

Secrets

If you check the code of the sample application you will see that it is also possible to use a Postgres database. That is a very helpful circumstance, because we would not be able to horizontally scale the application and also point to a single file. There is also the peculiarity that even though we can scale the Pods to multiple replicas because they share the same template, this template cannot propagate to volumes. We need different techniques to do the same with volumes as we will see later.

So back to Postgres configuration. We need to pass an obvious secret to the application. The database password. All other information can be passed as environment variables the way we know them. A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in

a container image. Using a Secret means that you don't need to include confidential data in your application code. So let's create our secret

```
% kubectl create secret generic postgres-password
--from-literal="password=nothakmem"
secret/postgres-password created

% kubectl get secrets
NAME               TYPE   DATA   AGE
postgres-password   Opaque   1       62s

% kubectl get secrets postgres-password -o yaml
apiVersion: v1
data:
  password: bm90c3RvcmZ1bmQ=
kind: Secret
metadata:
  creationTimestamp: "2024-02-28T13:43:15Z"
  name: postgres-password
  namespace: default
  resourceVersion: "2098"
  uid: 59015e85-2282-44fe-9164-f7c5fc456650
type: Opaque
```

We created a secret (of type `generic`; the other two types are `docker` for registry credentials and `tls` for SSL certificates). Generic and Opaque mean the same, but you create Generic and it gets reported back as Opaque.

We can ask for a more detailed report for the secret with `-o yaml`. Thanks to this we learn that Kubernetes secrets are not really secrets, they are base64 encodings. In fact we can read the plaintext with

```
% kubectl get secrets postgres-password -o jsonpath='{
.data.password}' | base64 -d ; echo
nothakmem
```

You can read more about JSONPath here.

So now our deployment becomes

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
      app: todo
  name: todo
spec:
```

```
  replicas: 1
  selector:
      matchLabels:
      app: todo
  template:
      metadata:
      labels:
      app: todo
      spec:
      containers:
      - image: yiorgos/docker-nodejs-sample:1
      name: docker-nodejs-sample
      ports:
      - containerPort: 3000
      env:
      - name: POSTGRES_HOST
      value: "test-psql.hakmem.net"
      - name: POSTGRES_DB
      value: "docker-nodejs-sample"
      - name: POSTGRES_USER
      value: "docker"
      - name: POSTGRES_PASSWORD
      valueFrom:
            secretKeyRef:
            name: postgres-password
            key: password
```

and we can see that it connects to Postgres

```
% kubectl logs todo-5866ffdd-g6mtk
Waiting for test-psql.hakmem.net:5432.
Connected!
Connected to postgres db at host test-psql.hakmem.net
Connected to db and created table todo_items if it did not exist
Listening on port 3000
```

We can now even scale the application as much as we like.

ConfigMaps

Our use of the secrets makes one think whether it is possible to describe all the above environment variables as secrets and thus use them by accessing their values directly from the Kubernetes API and not having to place them inside a YAML or other configuration file. This would also make it easier to maintain one version of the YAML and control the values of the variables depending on the environment the Pods run on. Enter configMaps, API objects used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment

variables, command-line arguments, or as configuration files in a volume. To create and then examine a configMap:

```
% kubectl create configmap postgres-config
--from-literal="host=test-psql.hakmem.net"
--from-literal="db=docker-nodejs-sample" --from-literal="user=docker"
configmap/postgres-config created

adamo@donnager:~/tmp
Wed Feb 28, 16:11 [10243] % kubectl get cm postgres-config -o yaml
apiVersion: v1
data:
  db: docker-nodejs-sample
  host: test-psql.hakmem.net
  user: docker
kind: ConfigMap
metadata:
  creationTimestamp: "2024-02-28T14:11:21Z"
  name: postgres-config
  namespace: default
  resourceVersion: "4368"
  uid: 7eb0a347-f1b2-4118-b67d-9e93459897d7
```

Similarly then to the case of Secrets, using the configMap, the deployment YAML becomes

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
      app: todo
  name: todo
spec:
  replicas: 1
  selector:
      matchLabels:
      app: todo
  template:
      metadata:
      labels:
      app: todo
      spec:
      containers:
      - image: yiorgos/docker-nodejs-sample:1
      name: docker-nodejs-sample
      ports:
      - containerPort: 3000
      env:
      - name: POSTGRES_HOST
```

```
      valueFrom:
            configMapKeyRef:
            name: postgres-config
            key: host
      - name: POSTGRES_DB
      valueFrom:
            configMapKeyRef:
            name: postgres-config
            key: db
      - name: POSTGRES_USER
      valueFrom:
            configMapKeyRef:
            name: postgres-config
            key: user
      - name: POSTGRES_PASSWORD
      valueFrom:
            secretKeyRef:
            name: postgres-password
            key: password
```

Further reading for Secrets and ConfigMaps

It should be noted that anything exposed via a Secret or a ConfigMap can be also accessed as a volume and mounted from the containers of a Pod to be used as a (read-only) file. Thus if your application expects a configuration file, it can be a ConfigMap, mounted as such in the end. Or it can be a Secret, if it is a "secret file" passed to the Pod as a volume to be used afterwards.

You are also encouraged to read about the Downward API which also allows for information to be communicated to Pod containers either via volumes or environment variables.

Namespaces

So far in our discussion we have overlooked mentioning namespaces. In Kubernetes, *namespaces* provide a mechanism for isolating groups of resources within a single cluster. Names of resources need to be unique within a namespace, but not across namespaces. Namespace-based scoping is applicable only for namespaced objects *(e.g. Deployments, Services, etc)* and not for cluster-wide objects *(e.g. StorageClass, Nodes, PersistentVolumes, etc)*.

```
% kubectl get ns
NAME              STATUS   AGE
default           Active   59m
kube-node-lease   Active   59m
kube-public       Active   59m
kube-system       Active   59m
```

Any object that is not specifically placed in a namespace, is placed in the `default` namespace. You can create a namespace with `kubectl create namespace adamo` and you can delete the

namespace with `kubectl delete ns adamo`. Deleting a namespace deletes all the namespaced objects it contains!

```
% kubectl -n adamo create deployment docker-names --image yiorgos/docker_names
deployment.apps/docker-names created

% kubectl -n adamo expose deployment docker-names --port 8000
service/docker-names exposed

% kubectl -n adamo get svc
NAME         TYPE         CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
docker-names   ClusterIP   10.96.238.214   <none>       8000/TCP   7s
```

Call this service from the `default` namespace

```
% kubectl run -it curl --image curlimages/curl -- sh
If you don't see a command prompt, try pressing enter.
~ $ curl http://docker-names.adamo.svc.cluster.local:8000
{"n":1,"host":"docker-names-8556cd4c77-srrnf","name":"peaceful_dirac"}
```

Which shows you how you can access services that run in another namespace.

> A note about contexts. In Kubernetes lingo sometimes the term context is used. What is a context? A context is a triple stated as (cluster, user, namespace) as these are stated inside the ~/.kube/config file. But Kubernetes does not have actual users you say. True, but a user in Kubernetes is defined by a key/cert pair your administrator issues and is combined with other RBAC directives that allow certain actions to take place or not, by the one who has the key/cert pair.

Ingress

As we have said, Kubernetes is built to service HTTP, HTTPS and gRPC via ports 80 and 443. So far we have only exposed services via NodePorts. 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.

The simplest Ingress implementation is nginx-ingress. There are a plethora of other implementations, based for example on Traefik, HA Proxy, Istio, Linkerd. Managed Kubernetes services come with their own Ingress implementations too. Rancher Desktop comes with Traefik pre-installed.

We need to install an Ingress controller in our machine, together with the cert-manager in order to support automatic issuance of self-signed certificates.

```
% kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.8.2/de
ploy/static/provider/cloud/deploy.yaml
:
% kubectl apply -f
https://github.com/cert-manager/cert-manager/releases/download/v1.14.3/cert-man
ager.yaml
:
```

You need to wait a few minutes for the required components to be installed and then we are ready to run our own ingress for the TODO application

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: todo
  annotations:
      nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - host: kubernetes.docker.internal
      http:
      paths:
      - path: /
      pathType: Prefix
      backend:
      service:
           name: todo
            port:
            number: 3000
```

After applying the above configuration, accessing http://kubernetes.docker.internal should take you to the TODO application.

> If you find that your browser cannot access `kubernetes.docker.internal` and complains about DNS resolution, then it might be the case that Docker Desktop did not properly update your `/etc/hosts`file. You need to add the following line to it
>
> `127.0.0.1 kubernetes.docker.internal`

We can make this Ingress SSL aware using the cert-manager

```
---
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
```

```
metadata:
  name: self-signed-issuer
spec:
  selfSigned: {}
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: todo
  annotations:
      nginx.ingress.kubernetes.io/rewrite-target: /
      cert-manager.io/cluster-issuer: "self-signed-issuer"
spec:
  ingressClassName: nginx
  tls:
  - hosts:
      - kubernetes.docker.internal
      secretName: todo-docker-internal
  rules:
  - host: kubernetes.docker.internal
      http:
      paths:
      - path: /
      pathType: Prefix
      backend:
      service:
            name: todo
            port:
            number: 3000
```

You can now access https://kubernetes.docker.internal - ignore the warning for the self signed certificate.

In production installations `cert-manager` can be used to auto-issue valid certificates with Let's Encrypt.

Sidecar containers

As we have already stated a Pod runs one or more containers. Of them, the container that is of most interest to us is called the main container of the Pod, whereas the others that coexist to facilitate the work of the main container are called sidecar containers. They extend the functionality of the main application container by providing additional services, or functionality such as logging, monitoring, security, or data synchronization, without directly altering the primary application code.

In our case one such sidecar usage could be the deployment of a pgbouncer container alongside the TODO application, thus deferring the proper database connection management to the pgbouncer

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
      app: todo
  name: todo
spec:
  replicas: 1
  selector:
      matchLabels:
      app: todo
  template:
      metadata:
      labels:
      app: todo
      spec:
      containers:
      - name: pgbouncer
      image: bitnami/pgbouncer
      ports:
      - containerPort: 5432
      env:
      - name: PGBOUNCER_PORT
      value: "5432"
      - name: POSTGRESQL_USERNAME
      valueFrom:
            configMapKeyRef:
            name: postgres-config
            key: user
      - name: POSTGRESQL_PASSWORD
      valueFrom:
            secretKeyRef:
            name: postgres-password
            key: password
      - name: POSTGRESQL_DATABASE
      valueFrom:
            configMapKeyRef:
            name: postgres-config
            key: db
      - name: POSTGRESQL_HOST
      valueFrom:
            configMapKeyRef:
            name: postgres-config
            key: host
```

```
      - name: PGBOUNCER_DATABASE
      valueFrom:
            configMapKeyRef:
            name: postgres-config
            key: db
      - image: yiorgos/docker-nodejs-sample:1
      name: docker-nodejs-sample
      ports:
      - containerPort: 3000
      env:
      - name: POSTGRES_HOST
      value: "127.0.0.1"
      - name: POSTGRES_DB
      valueFrom:
            configMapKeyRef:
            name: postgres-config
            key: db
      - name: POSTGRES_USER
      valueFrom:
            configMapKeyRef:
            name: postgres-config
            key: user
      - name: POSTGRES_PASSWORD
      valueFrom:
            secretKeyRef:
            name: postgres-password
            key: password
```

As you can see both containers can access the same secrets and configmaps at the same time.

Init Containers

Sometimes, prior to starting an application, it is necessary to perform some initial preparatory work, using tooling not available in the application image, or just preparing some configuration or other job before starting. Init containers exist to facilitate this need. Standard containers of a Pod won't start unless all init containers complete successfully first. Here is an example of a web server deployment where each Pod is configured to report back its hostname.

```
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  type: NodePort
  selector:
      app: nginx
  ports:
```

```
      - protocol: TCP
      port: 80
      targetPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
      app: nginx
  name: nginx
spec:
  replicas: 2
  selector:
      matchLabels:
      app: nginx
  template:
      metadata:
      labels:
      app: nginx
      spec:
      volumes:
      - name: html
      emptyDir: {}
      initContainers:
      - name: busybox
      image: busybox
      volumeMounts:
      - name: html
      mountPath: /html
      command:
      - sh
      - -c
      - echo $HOSTNAME > /html/index.html
      containers:
      - image: nginx
      name: nginx
      volumeMounts:
      - name: html
      mountPath: /usr/share/nginx/html
```

Here we start a deployment for nginx with two replicas. For each Pod the init container (`busybox`) runs a shell command that writes the Pod's hostname to `index.html` inside an ephemeral directory. Then this directory gets mounted in the proper position for the web server to be able to serve `index.html`. emptyDir is the only kind of volume that can be used with multiple replicas in the same deployment object.

We can now access the nodeport multiple times and we will see that both pods respond to our calls at a roughly equal distribution.