How to developp with Kubernetes

Developper worstation's tools

Damien Ribeiro

Tinkou consulting

July 15, 2019

Outline

Pre-requirements

Skills

- Be comfortable with UNIX command line
- Docker notions
- Kubernetes notions

Resources

- A computer or a VM with docker installed
- An access to a docker registry
- An access to a kubernetes cluster

Objective

- Our first task is to create a wonderfull worker that will keep telling us that it is alive.
- To keep simple, this worker will be in batch (but any langage will work).
- Then we will run this worker in a container on our local docker installation.

Our worker code

First we place ourself in a working folder WORKER_DIR. And we create the worker core:

worker.sh

```
while true;do
  echo "I'm alive and it is $(date)!"
  sleep 2
done
```

Test our worker

In a shell

```
chmod u+x worker.sh
./worker.sh
```

We can kill our worker with hitting **Ctrl+C**.

Embedded our worker in a container

We need to write a Dockerfile:

Dockerfile

```
FROM alpine

COPY worker.sh .

CMD /worker.sh
```

Create our image

We create the image of our worker:

Command line

```
docker build -t tinkou/worker:v1 .
```

This image can be seen locally in docker:

Command line

```
docker images
```

Run our container

Now that we have our worker image, it is time to run it in docker:

Command line

```
docker run tinkou/worker:v1
```

The output of our worker can be seen.
We can kill our container with **Ctrl+C**.

Time to play a little with our container

We can start our container in background and with giving it a name:

Command line

```
docker run --name worker -d tinkou/worker:v1
```

We can still take a look to the logs:

Command line

```
docker logs -f worker
```

We can quit the follow with **Ctrl+C**.

Time to play a little with our container

We can start, stop and see the logs of our container at will:

Command line

```
docker ps
docker stop worker
docker ps
docker start worker
docker ps
```

This is always the same container running. We can see all existing containers to check it:

Command line

```
docker ps -a
```

Clean after work

We stop and remove the containers and then remove the images:

Command line

```
docker stop $(docker ps -q)
docker rm $(docker ps -aq)
docker rmi $(docker images -q)
```

We can check with:

Command line

```
docker ps
docker ps -a
docker images
```

Objective

Now that we have a worker, we want to run it in a kubernetes cluster.

The first step will be to run a container into kubernetes to make sure that we can do it.
Then, we will run our worker in kubernetes.

How to access to a Kubernetes cluster

First thing first, to deploy something in kubernetes, we need to access to a cluster.

For that, we need to install kubectl on our worstation (installation documentation).

The documentation give instruction to configure the cluster access (using a config file given by your sysadmin).

Very usefull, the documentation explain how to enable the completion.

Working in a dedicated namespace

To avoid impacting other components, we are going to isolate ourselves in a namespace:

Command line

```
kubectl get namespaces
kubectl create namespace my-namespace
kubectl get ns
```

Working in a dedicated namespace

Then we configure kubectl to use his context:

Command line

```
kubectl config get-context

kubectl config set-context training          \
                  --cluster=kubernetes       \
                  --user=kubernetes-admin    \
                  --namespace=my-namespace

kubectl config use-context training
kubectl config get-context
```

Running our first container in the cluster

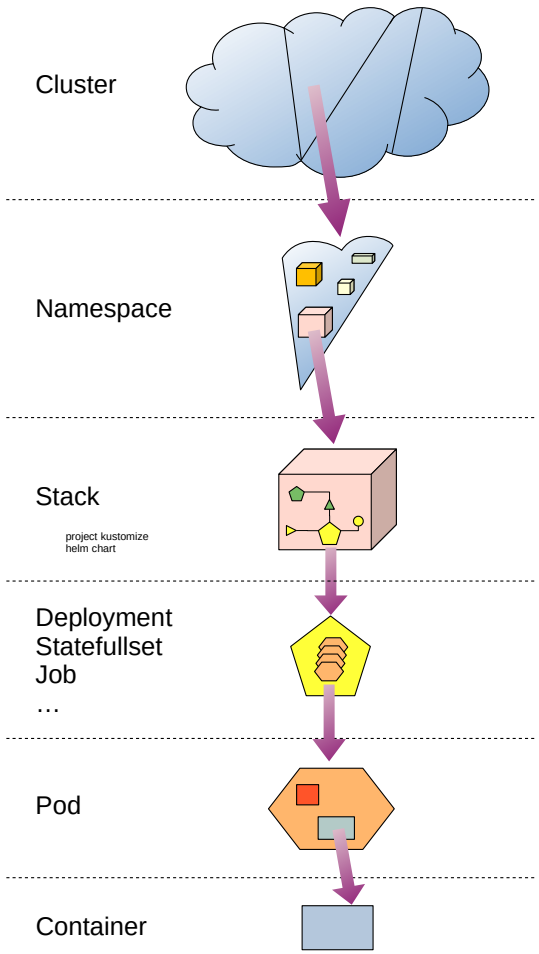
To begin, we are just going to deploy a container sending a ping:

Command line

```
kubectl run pingpong --image=alpine ping 1.1.1.1

kubectl get pods
kubectl get deployments -o wide
kubectl get replicaset -o wide
```

Kubernetes application architecture



Looking at logs in a kubernetes cluster

We want to display the logs of this container:

Command line in window 1

```
kubectl logs -f pingpong-XXXX-XXXX
```

Something happens and we need to recreate the pod:

Command line in window 2

```
kubectl get pods -w
```

Command line in window 3

```
kubectl delete pod pingpong-XXXX-XXXX
```

Looking at logs in a kubernetes cluster

There are several problems:

- we do not see which logs comes from which pod
- in fact we do not see the new containers logs
- the follow is broken by the operation

How to solve this and be able to follow logs even if pods are moving?

Using stern to look at logs

Stern installation instructions can be found here.

So, we are going the redo the operation but this time using stern:

Command line in window 1

```
stern pingpong
```

Command line in window 2

```
watch kubectl get all
```

Command line in window 3

```
kubectl delete pod pingpong-XXXX-XXXX
```

Clean the namespace

Before returning to our worker, a little cleaning can be wise:

Command line1

```
kubectl delete deployment pinpong
kubectl get all
```

Running our worker in kubernetes

Now that we know how to run something in kubernetes, we are going to do it with our worker:

Command line in window 2

```
stern worker
```

Command line in window 3

```
kubectl get pods -w
```

Command line in window 1

```
docker images
docker build -t tinkou/worker:v1 .
kubectl run worker --image=tinkou/worker:v1
```

Troubleshooting

Kubectl get pods indicate that something went wrong.
Check kubernetes object to find the problem root cause:

Command line in window 1

```
kubectl describe deployment worker
kubectl describe replicaset worker-XXXX
kubectl describe pod worker-XXXX-XXXX
```

Using a registry

As our cluster can't access our local image, let's push it in a repository instead:

Command line 1

```
docker tag tinkou/worker:v1 \
           <registry>/training/worker-<id>:v1
docker login <registry>
docker push <registry>/training/worker-<id>:v1
```

`<registry>` is the registry given by your sysadmin.
`<id>` is used to differenciate the images between the different trainees.

Using a registry

And we try again by forcing a pod reconstruction::

Command line 1

```
kubectl edit deployment worker
```

And change the image field value.

Adding docker registry credentials

The logs on the pod indicate that docker registry credentials aren't valid.
We need to store in a kubernetes secret our registry credentials:

Command line 1

```
kubectl create secret docker-registry regcred \
        --docker-server=<registry>             \
        --docker-username=<user>               \
        --docker-password=<password>           \
        --docker-email=<yourEmail>
```

Adding docker registry credentials

Get our worker deployment yaml and add to it these credentials:

Command line 1

```
kubectl get deployment worker -o yaml >deployment.yaml
```

deployment.yaml

```
spec:
  template:
    spec:
      imagePullSecrets:
      - name: regcred
```

And apply it to update the configuration:

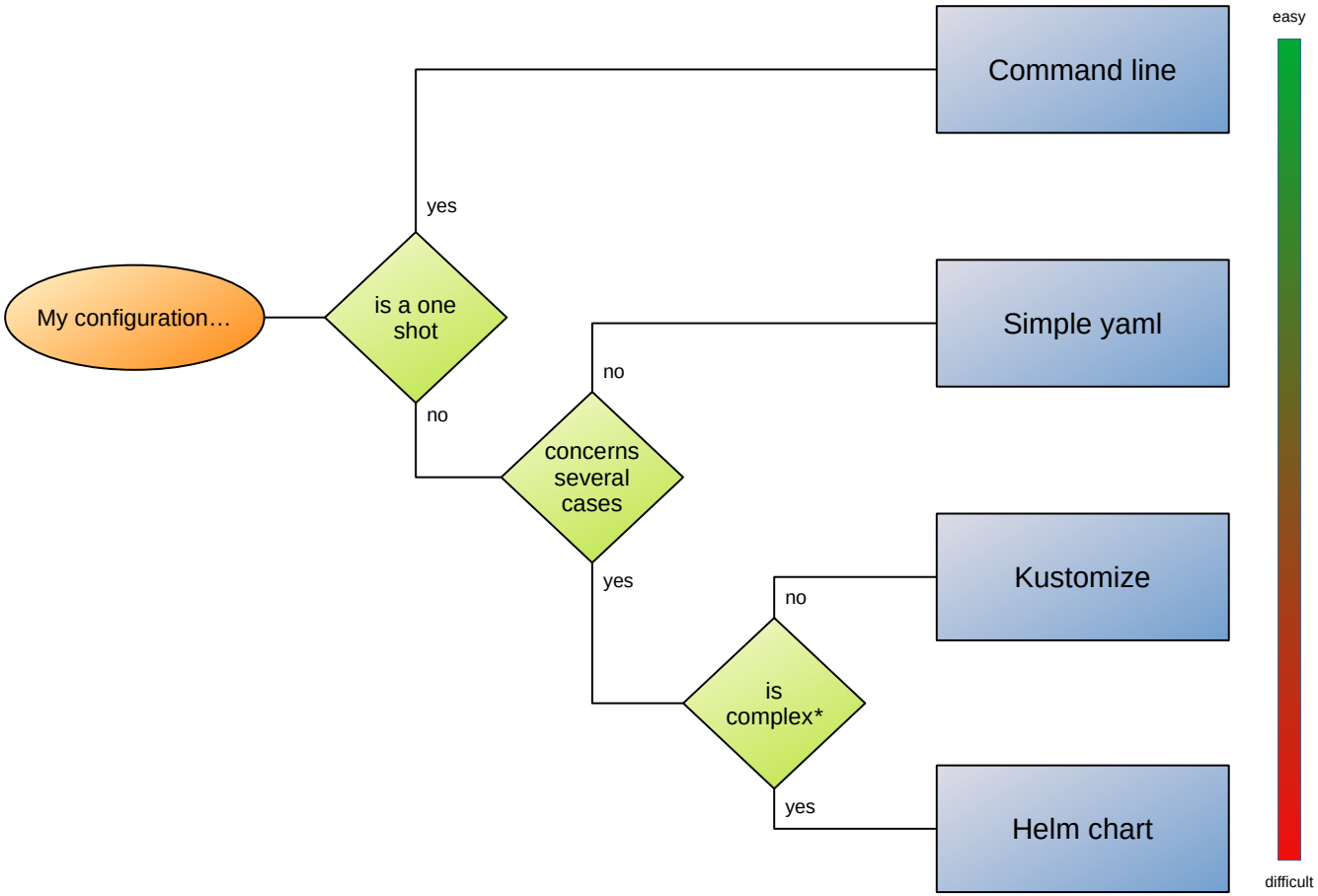
Command line 1

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

Objective

Now that we have a worker in kubernetes, we wants to be able to configure it.

Kustomize



* complex as needing conditionals to valorize or depending on contextual values

Layer based configuration

Kustomize is a tool now integrated in kubectl.

It manage the configuration using a layer based system.
That means that a configuration is the result of a base configuration, on wich layers are applyed.
Eache layer can contains new resources or new patches.

As a base layer is considered as a resource, that enable to define a configuration as the concatenation of several other configurations and their patches.

Initialize the base

We are creating a folder tree to sort our files

Command line 1

```
mkdir kube
cd kube
mkdir base
cd base
```

Initialize the base

We are starting with a fresh new deployment.yaml

source.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: worker
  labels:
    tinkou: worker
spec:
  selector:
    matchLabels:
      tinkou: worker
  template:
    metadata:
      labels:
        tinkou: worker
    spec:
      containers:
      - name: worker
        image: <registry>/training/worker-<id>:v1
      imagePullSecrets:
      - name: regcred
```

Initialize the base

Command line 1

```
touch kustomization.yaml
kustomize edit fix
kustomize edit add resource source.yaml
kustomize build
kubectl apply -k .
```

Initialize the base

As we modify an immutable field, we need to delete the previous deployment before:

Command line 1

```
kubectl delete deployment worker
kubectl apply -k .
```

Add a parameter to our worker

Modify the worker to use a parameter:

worker.sh

```
while true; do
  echo "I'm $NAME and it is $(date)"
  sleep 2
done
```

Deploy the new version in our cluster

First we need to create a new version of the image:

Command line 1

```
docker build -t <registry>/training/worker-<id>:v2 \
                ../../
docker push <registry>/training/worker-<id>:v2
```

Deploy the new version in our cluster

Change the deployment configuration:

source.yaml

Replace v1 by v2
Add to the containers spec part:

```
spec:
  containers:
    env:
    - name: NAME
      value: <myName>
```

And finaly apply the modification:

Command line 1

```
kubectl apply -k .
```

Deploy the new version in our cluster

There are too many operations.

Is there a way to simplify this?

Skaffold

Skaffold is a developper oriented tool create to simplify the packaging and deployment on kubernetes.

The Skaffold documentation can be found her.

Installation documentation can be found here.

Initialize skaffold

Let's return in WORKER_DIR and initialize skaffold:

Command line 1

```
skaffold init
```

Follow the application command line interface.

Initialize skaffold

By default skaffold detect the yaml, so it need to be configured to use kustomize:

skaffold.yaml

Remove the tag from the image

Replace the block .deploy.kubectl by:

```
deploy:
  kustomize:
    path: kube/base
```

Using skaffold

We are going to test a little skaffold commands:

Command line 1

```
skaffold build
skaffold run
skaffold delete
skaffold dev
```

Try to modify the file worker.sh.

Create a kustomize layer for skaffold

Skaffold do not upgrade the image. . .

Command line 4

```
kubectl describe deployment worker
```

Create a kustomize layer for skaffold

We need to let skaffold manage the image tag version, by created a dedicated configuration:

Command line 4

```
cd kube
mkdir skaffold
vi patch.yaml
```

patch.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: worker
spec:
  template:
    spec:
      containers:
      - name: worker
        image: <registry>/training/worker-<id>
```

Create a kustomize layer for skaffold

Command line 4

```
touch kustomization.yaml
kustomize edit fix
kustomize edit add resource ../base
kustomize edit add patch patch.yaml
```

Indicate the new kustomize source to skaffold:

skaffold.yaml

```
deploy:
  kustomize:
    path: kube/skaffold
```

And the worker should be automatically updated.

Isolate the configuration in a configmap

Add a file conf.env in the folder base:

conf.env

```
NAME=Georges
```

Command line 4

```
kustomize edit add configmap worker \
                  --from-env-file conf.env
```

deployment.yaml

Replace the block .spec.template.spec.containers.env by:

```
envFrom:
  - configMapRef:
      name: worker
```

Define a dev configuration

Create a configuration specific to the current development:

patch.yaml

Add a the file end:

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: worker
data:
  NAME: Marion
```

Skaffold dev modification detection

Try to modify (or just touch) several files to test which one trigger a skaffold build or run.
And finally clean everything with Ctrl+C to interrupt the skaffold dev command.