How to developp with Kubernetes

Developper worstation's tools

Damien Ribeiro

Tinkou consulting

October 10, 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

Pre-requirements

During the workshop, we will need to use several shell at the same time.

We need a tool for that, like for example:

- tmux (in command line)
- screen
- Terminator
- ... or just open several terminals

In local with minikube

We need to install the tools used during the workshops:

Docker	Official release page
Kubectl	Official installation documentation
Minikube	Official installation documentation
Kustomize	Official installation documentation
Skaffold	Official installation documentation
Stern	Official installation documentation

In local with minikube

Minikube consist of a single node kubernetes cluster. Here a few commands useful for the workshop:

Get the cluster IP. This IP will be used as the one of each kubernetes servers (master or node):

```
minikube ip
```

For the registry, we will use the docker local registry of minikube.

In local with minikube

For the following, make sure that minikube is stopped:

In a shell

```
minikube stop
```

Check minikube status:

In a shell

```
minikube status
```

Objective

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

Our worker code

First we place ourselves in a working folder "`worker`". 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 that nothing remain with:

Command line

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

To go farther...

To learn more thing about docker, you can check Docker official documentation or use the docker built-in help:

In a shell

```
docker help
docker help build
...
```

Questions?

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.

In local with minikube

For the following, make sure that minikube is started:

In a shell

```
minikube start
```

Check minikube status:

In a shell

```
minikube status
```

How to access to a Kubernetes cluster

The kubectl client used to access a kubernetes cluster used the configuration file $home/.kube/config
More informations here.

As we are using Minikube, starting it already configure kubectl.

As indicated in this documentation, you can activated the completion for kubectl.

Running our first container in the cluster

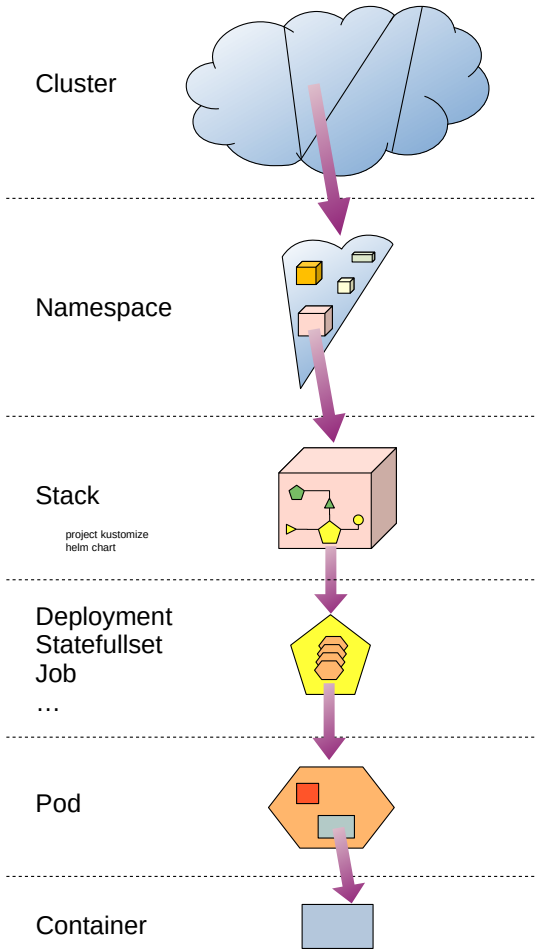
To begin, we are just going to deploy a container sending a ping:

Command line

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

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

Kubernetes application architecture



Prepare the environment

Starting from now, we are going to use 3 shells in 3 differents windows/panes.

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

So, we are going to 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 line in window 1

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

And stop the commands in the other windows.

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
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
```

The "Events" are the interesting part here.

Using a registry

The events indicates that the image can not be found.

The docker daemon in minikube is not the same as the local one.

We need to:

- link the local docker CLI to the minikube docker daemon
- create the image in the minikube docker daemon locale registry

Command line 1

```
eval $(minikube docker-env)
docker images
docker build -t tinkou/worker:v1 .
```

Using a registry

After a moment, the pod start and stern start to display logs.
What happen?

Using a registry

After a moment, the pod start and stern start to display logs. What happen?

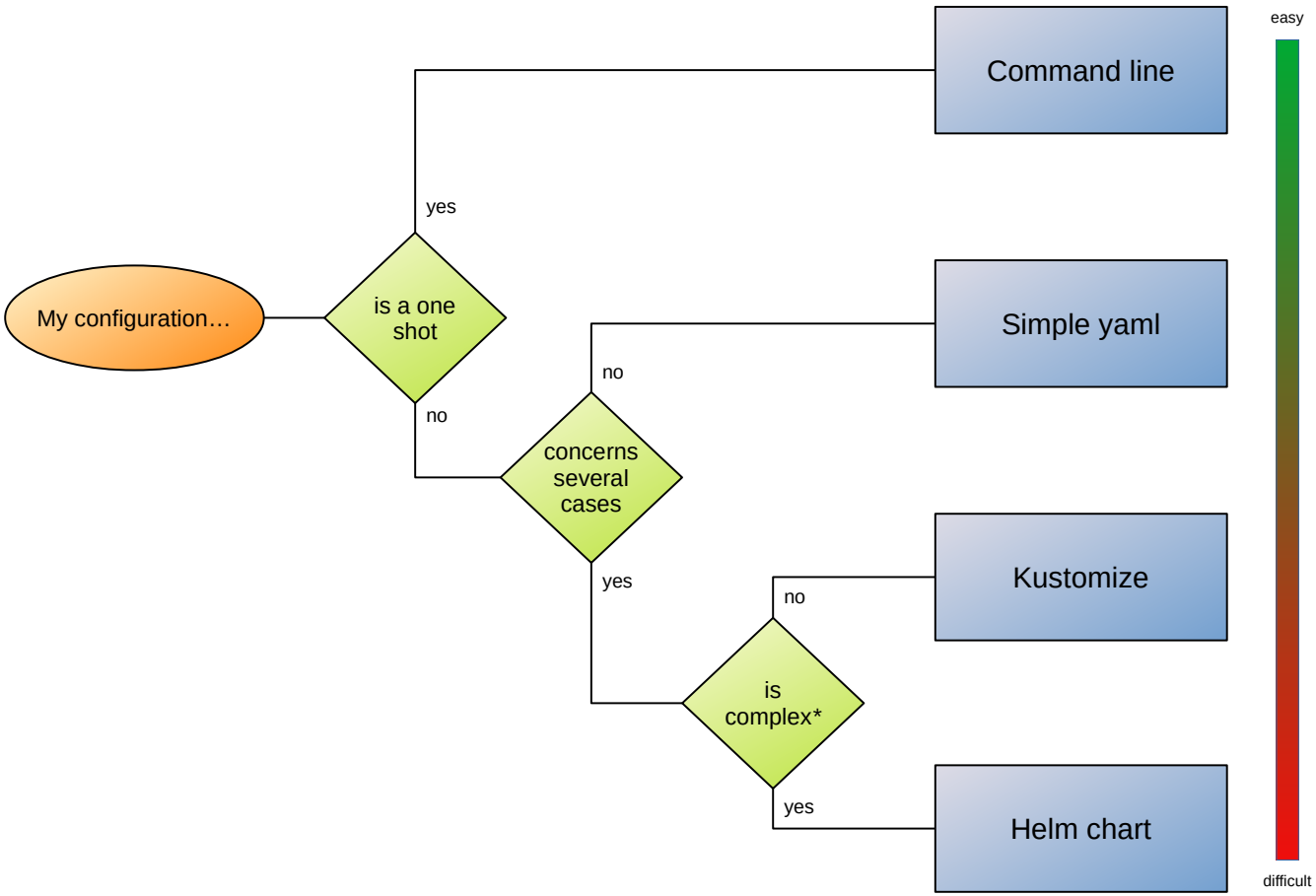
The different elements of kubernetes retry periodically. And as the image is now available, the pods can successfully start.

Questions?

Objective

Now that we have a worker in kubernetes, we wants to be able to configure it.
Also, we want to be able to differentiate between development and production configuration.

Kustomize among other solutions



* 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 which layers are applied.
Each 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 in folder worker

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

Command line 2 still had "`stern worker`" running

Command line 3 still had "`kubectl get pods -w`" running

Initialize the base

We are starting with a source.yaml file describing our deployment

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: tinkou/worker:v1
```

Initialize the base

Command line 1 in folder base

```
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 in folder base

```
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 in folder worker

```
docker build -t tinkou/worker:v2 .
```

Deploy the new version in our cluster

Adapt in source.yaml the field `.spec.template.spec`:

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: tinkou/worker:v2
        env:
        - name: NAME
          value: Bink
```

Deploy the new version in our cluster

And finally apply the modification:

Command line 1 in folder worker

```
kubectl apply -k kube/base
```

Deploy the new version in our cluster

And finally apply the modification:

Command line 1 in folder worker

```
kubectl apply -k kube/base
```

To clean what have been deployed:

Command line 1 in folder worker

```
kubectl delete -k kube/base
```

Deploy the new version in our cluster

There are too many operations.

Is there a way to simplify this?

Skaffold

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

The Skaffold documentation can be found her.

Initialize skaffold

Let's initialize skaffold:

Command line 1 in folder worker

```
skaffold init
```

Follow the application command line interface.

Initialize skaffold

Skaffold detect the yaml and skaffold.yaml needs to be configured to use kustomize:

skaffold.yaml

```
apiVersion: skaffold/v1beta15
kind: Config
metadata:
  name: worker
build:
  artifacts:
  - image: tinkou/worker
deploy:
  kustomize:
    path: kube/base
```

The `apiVersion` can change with skaffold version.

Using Skaffold

Let's use Skaffold to build our image:

Command line 1 in folder worker

```
skaffold run
```

Using Skaffold

Let's use Skaffold to build our image:

Command line 1 in folder worker

```
skaffold run
```

A particularity of Skaffold is that to work with minikube, the current context of kubectl must be "minikube".
The current context can be with:

Command line 1

```
kubectl config current-context
```

Using skaffold

We are going to test a little skaffold commands:

Command line 1 in folder worker

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

build only build the image and push it in the registry (except in minikube)

run build, push and run the image in kubectl current cluster

delete delete run element from the kubectl current cluster

Using skaffold

Lets try the dev function:

Command line 1 in folder worker

```
skaffold dev
```

It works like run, but displaying logs and keeping the hand on the shell.

Using skaffold

Open a command line in a new window 4.
Try to modify the file worker.sh in this new command line.

And look at how skaffold dynamically build and deploy each time the file is saved.

Isolate the configuration in a configmap

Add a file conf.env in the folder base:

conf.env

```
NAME=Trent
```

Command line 4 in the folder kube/base

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

Isolate the configuration in a configmap

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: tinkou/worker:v2
        envFrom:
        - configMapRef:
            name: worker
```

Define a local configuration

Create a folder `local` in `worker/kube`, and add in it the file `patch.yaml`:

patch.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: worker
data:
  NAME: Iris
```

Define a local configuration

Create a folder `local` in `worker/kube`, and add in it the file `patch.yaml`:

patch.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: worker
data:
  NAME: Iris
```

Nothing happen in skaffold dev.

Define a local configuration: kustomize

First create a new kustomize project with a base resource and the patch:

Command line 4 in folder worker/kube/local

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

Still nothing in skaffold dev.

Define a local configuration: skaffold

Indicate to skaffold to use this local values when using minikube:

skaffold.yaml

```
apiVersion: skaffold/v1beta15
kind: Config
metadata:
  name: worker
build:
  artifacts:
  - image: tinkou/worker
deploy:
  kustomize:
    path: kube/base
profiles:
- name: local
  activation:
  - kubeContext: minikube
  deploy:
    kustomize:
      path: /kube/local
```

Skaffold dev modifications detection

Now the modification is deployed.
Lets modify the base configuration:

kube/base/conf.env

```
NAME=Dor
```

Skaffold dev modifications detection

Now the modification is deployed.
Lets modify the base configuration:

kube/base/conf.env

```
NAME=Dor
```

Skaffold do not react.

Try to modify several files to see how skaffold detection works.

Cleaning

If you kill the command `skaffold dev` with **Ctrl+C**, skaffold will clean what it has deployed.

Questions?

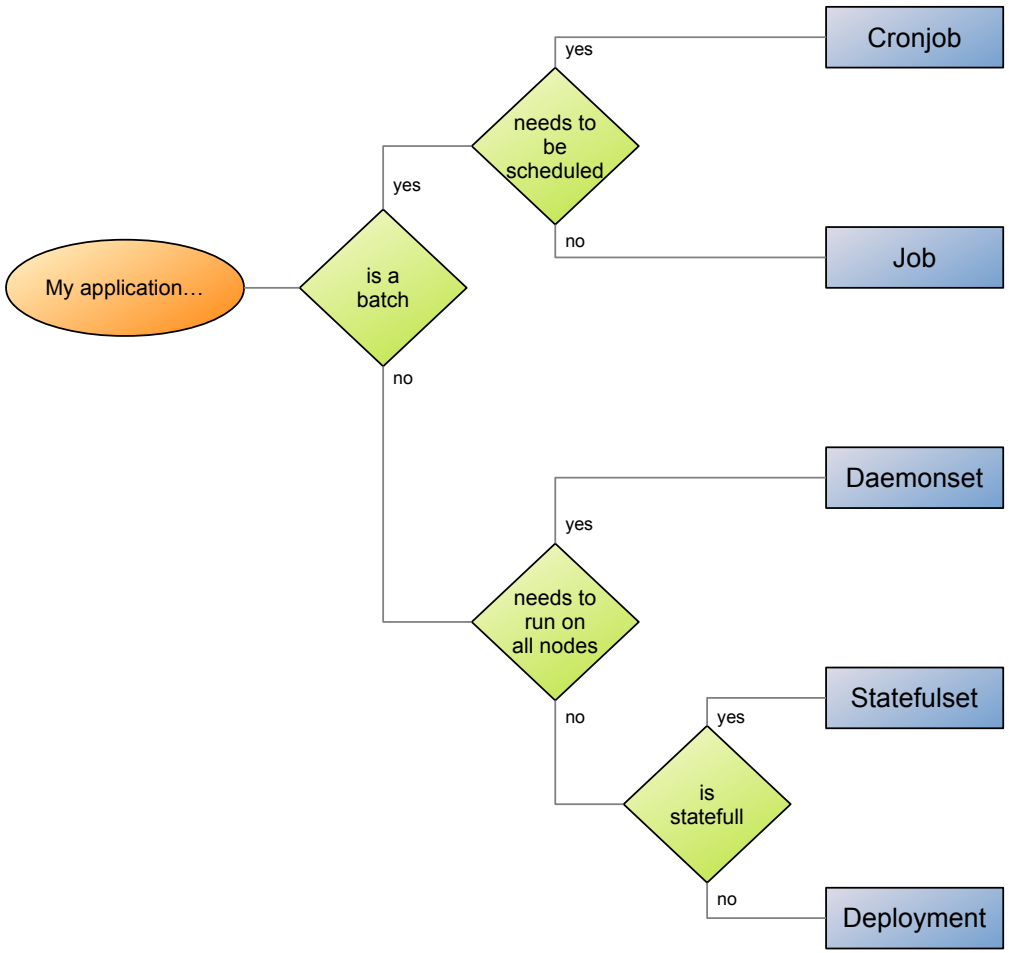
Objective

Our worker is working fine, but induce too many stress on the system.We want now to deploy it as a batch.

As such, we want to be able to schedule it.

We are now going to create a batch performing the same task as our worker.

What kind use to deploy our application



Create the batch

Beside the folder `worker`, create a folder `batch`.

Create the batch itself:

batch.sh

```
echo "I'm Humphrey and it is $(date)"
```

Now, each execution is unitary.

Exercise

Build and run in docker an image `tinkou/batch:v1` containing this batch.

solution

Dockerfile

```
FROM alpine

COPY batch.sh .
RUN chmod u+x batch.sh

CMD /batch.sh
```

Command line 1

```
docker build -t tinkou/batch:v1 .
docker run tinkou/batch:v1
```

Create our first CronJob

cronjob.yaml

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: my-batch
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          labels:
            tinkou: batch
        spec:
          containers:
          - name: runner
            image: tinkou/batch:v1
            env:
            - name: NAME
              value: Chem
          restartPolicy: Never
```

Create our first CronJob

Command line 2

```
stern -l tinkou=batch
```

Command line 3

```
watch kubectl get all
```

Command line 1

```
kubectl apply -f cronjob.yaml
```

Wait for a few batches to run...

```
kubectl logs -l tinkou=batch
```

Conclusions

Conclusions

- CronJob are easy to defined
- stern is less adapt than a standard kubectl logs for jobs

CronJobs useful options

- A field .spec.suspend enable to suspend the scheduling of a CronJob
- Jobs in error aren't removed
- The history limit of successful jobs can be set

Exercise

Modify the configuration to use kustomize and skaffold with:

- a base configuration similar as the current one
- a local configuration for minikube with NAME=Dolph

A solution

In a new folder batch/kube

```
kube
|- base
|   |- conf.env
|   |- cronjob.yaml
|   |- kustomization.yaml
|- local
    |- conf.yaml
    |- kustomization.yaml
```

kube/base/conf.env

```
NAME=Humphrey
```

A solution

kube/base/cronjob.yaml

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: my-batch
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          labels:
            tinkou: batch
        spec:
          containers:
          - name: runner
            image: tinkou/batch:v1
            envFrom:
            - configMapRef:
                name: batch
          restartPolicy: Never
```

A solution

Command line 1 in folder kube/base

```
touch kustomization.yaml
kustomize edit add resource cronjob.yaml
kustomize edit add configmap batch \
                              --from-env-file conf.env
```

A solution

kube/local/conf.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: batch
data:
  NAME: Dolph
```

A solution

Command line 1 in folder kube/local

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

A solution

batch/skaffold.yaml

```
apiVersion: skaffold/v1beta12
kind: Config
build:
  artifacts:
  - image: tinkou/batch
deploy:
  kustomize:
    path: kube/base
profiles:
- name: local
  activation:
  - kubeContext: minikube
  deploy:
    kustomize:
      path: kube/local
```

A solution

Command line 1 in folder batch

```
skaffold run
```

Take a quick look at the scheduling options and logs:

Command line 1

```
kubectl describe cronjob my-batch
```

Clean after working:

Command line 1 in folder batch

```
skaffold delete
```

Questions?

Objective

Our project now need a web service.

This web service needed to be monitored and scalable.

Create the web service itself

Create a new working folder `web`.

web/index.html

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Web service training</title>
  </head>
  <body>
    <h1>Hello World!</h1>
  </body>
</html>
```

Create the web service image

Dockerfile

```
FROM nginx:alpine
COPY index.html /usr/share/nginx/html
```

Command line 1

```
docker build -t tinkou/web:1
```

Create kubernetes deployment configuration

deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  labels:
    tinkou: web
spec:
  selector:
    matchLabels:
      tinkou: web
  template:
    metadata:
      labels:
        tinkou: web
    spec:
      containers:
      - name: service
        image: tinkou/web:v1
```

Create kubernetes deployment configuration

Command line 2

```
stern -l tinkou=web
```

Command line 3

```
watch kubectl get all
```

Command line 1 in folder web

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

Create kubernetes deployment configuration

A deployment, a replicaset and a pod are created.

But how to accede to this web service?

Exposing a web service

Pods have an IP. . .

Command line 1

```
kubectl get pod -o wide
```

. . . but this IP is internal to the cluster.

Exposing a web service

Pods have an IP. . .

Command line 1

```
kubectl get pod -o wide
```

. . . but this IP is internal to the cluster.

—

In kubernetes, this exposition in done via services.

Exposing a web service

web/service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    tinkou: web
spec:
  type: NodePort
  selector:
    tinkou: web
  ports:
  - port: 80
    protocol: TCP
    nodePort: 32001
```

Exposing a web service

Command line 1 in folder web

```
kubectl apply -f service.yaml
curl http://$(minikube ip):32001
```

Now we can access to the service.
It is also possible to display it in a web browser (by replacing `$(minikube ip)` by its value).

How does it work?

- the minikube VM has the IP given by `minikube ip`
- as a NodePort, the service listen to the port 32001 of all nodes
- the service redirect requests in round robin to pods matching the selector
- pods listen to the same ports as their containers
- at least the container can answer to requests