How to develop with Kubernetes

Developer workstation's tools

Damien Ribeiro

Tinkou consulting

February 4, 2020

Outline

1. First worker
2. Working in Kubernetes
3. Managing different configurations
4. Batch and scheduling
5. Web service

Pre-requirements

- a little of bash-fu
- a computer with docker installed (official documentation)

Terminals

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

Terminals

In this workshop we will use up to four terminals at once.

To keep things clear, these terminals will be named:

- Command terminal
- Monitor terminal
- Logs terminal
- Second command terminal

If the command to run needs to be in a specific folder, it will be indicated as well.

Folder tree

For the following, all path will be assume to be under a *training* folder.

For example, assuming the path of *training* is */training*,
the mention *folder/scrip.sh*
will be equivalent to */training/folder/script.sh*.

Using workstation tools

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

Warning about Kustomize and Skaffold

As it is now, Kustomize and Skaffold are still under construction.
These tools give a great help but the version evolve quickly and the files version can change.
To avoid compatibility issues, there is a need to define the update policy for those tool's version.

As these problematic depends on the organisation of the team or company, it will not be talked here.

Using 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.

Using minikube

For the following, make sure that minikube is stopped:

Command terminal

```
minikube stop
```

Check minikube status:

Command terminal

```
minikube status
```

Objectives

- Our first task is to create a wonderful worker that will keep telling us that it is alive.
- To keep things simple, this worker will be a shell script.
- Then we will run this worker in a container on our local docker installation.

Creating a simple worker

First we place ourselves in the folder "`worker`". And we create the worker script:

worker/worker.sh

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

Testing our worker

Command terminal in folder worker

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

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

Embedding our worker in a container

We need to write a Dockerfile:

worker/Dockerfile

```
FROM alpine

COPY worker.sh .

CMD /worker.sh
```

Creating our image

We create the image of our worker:

Command terminal in folder worker

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

This image can be seen locally in docker:

Command terminal

```
docker images
```

Running our container

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

Command terminal

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

We can see our worker output.
We can kill our container with **Ctrl+C**.

Playing a little with our container

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

Command terminal

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

We can still take a look to the logs:

Command terminal

```
docker logs -f worker
```

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

Playing a little with our container

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

Command terminal

```
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 terminal

```
docker ps -a
```

Cleaning up

We stop and remove the containers and then remove the images (*beware it will clean everything*):

Command terminal

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

We can check that nothing remain with:

Command terminal

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

Going further...

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

Command terminal

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

Questions?

Objectives

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.

Using minikube

For the following, make sure that minikube is started:

Command terminal

```
minikube start
```

Check minikube status:

Command terminal

```
minikube status
```

Accessing a Kubernetes cluster

To access a kubernetes cluster, we use kubectl client configured with the file *$home/.kube/config*
More information here.

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

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

Running our first container in the cluster

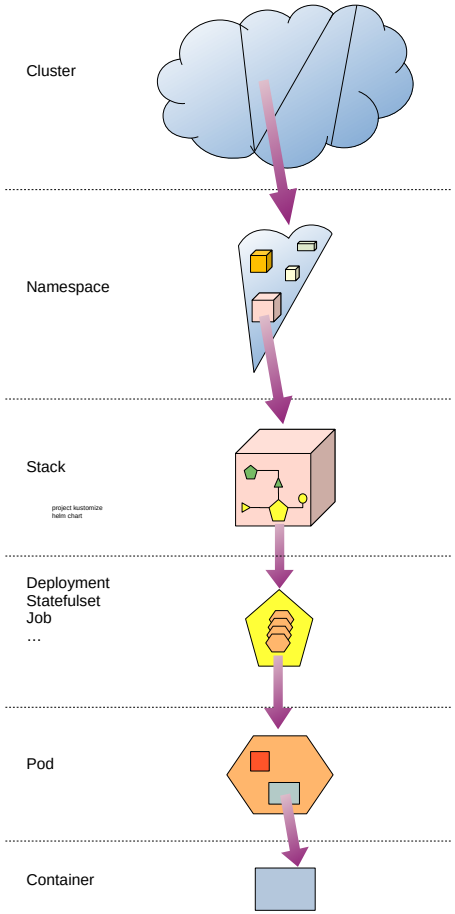
To begin, we are just going to deploy a container sending a ping:

Command terminal

```
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



Preparing 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:

Logs terminal

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

Something happens and we need to recreate the pod:

Monitor terminal

```
kubectl get pods -w
```

Commands terminal

```
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

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 terminal

```
stern pingpong
```

Monitor terminal

```
watch kubectl get all
```

Second command terminal

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

Cleaning up the namespace

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

Command terminal

```
kubectl delete deployment pingpong
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:

Logs terminal

```
stern worker
```

Monitor terminal

```
kubectl get pods -w
```

Command terminal

```
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 terminal

```
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 terminal

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

Using a registry

After a moment, the pod and stern starts displaying logs. What happened?

Using a registry

After a moment, the pod and stern starts displaying logs. What happened?

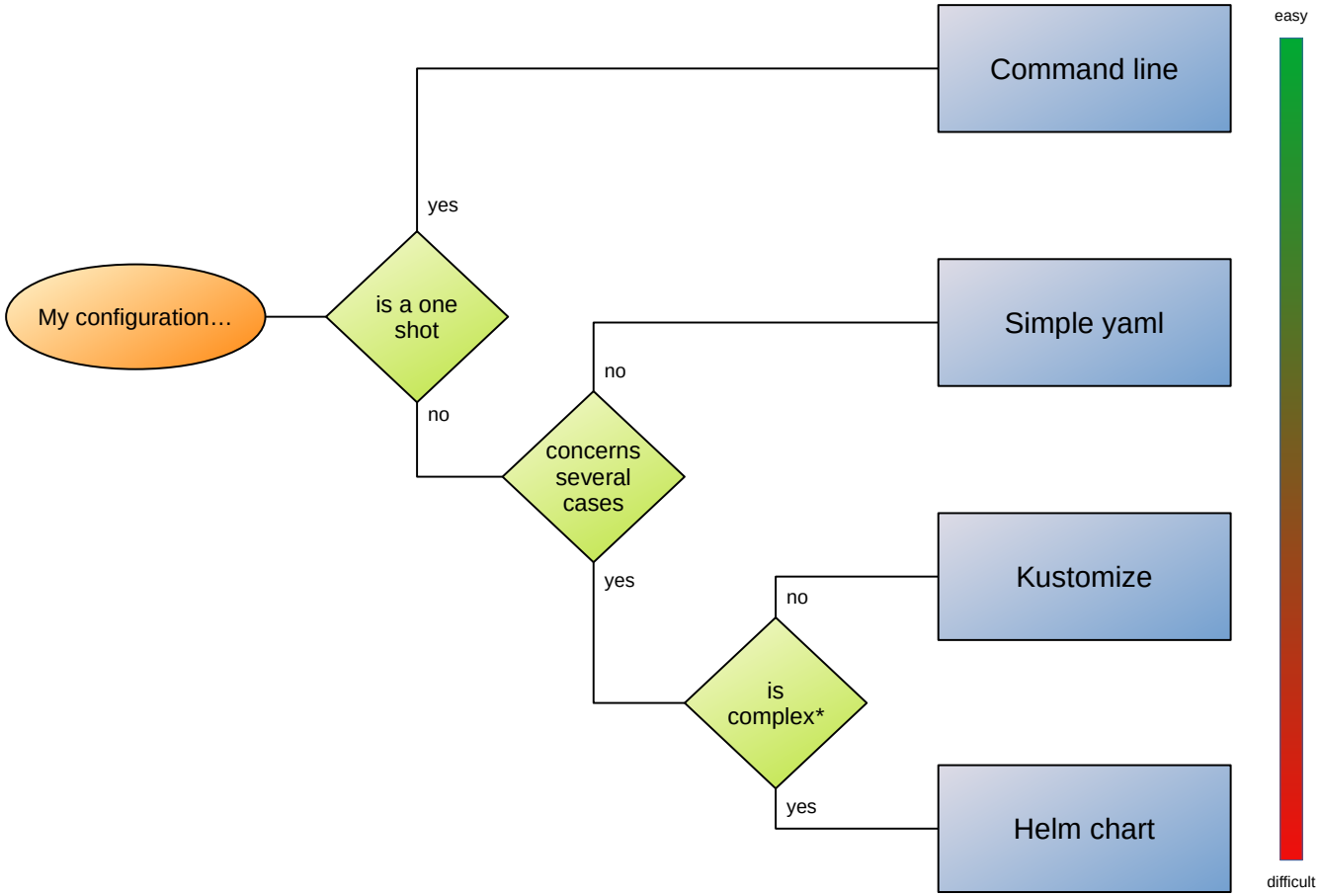
The different elements of kubernetes retry periodically. And as the image is now available, the pods can successfully start.

Questions?

Objectives

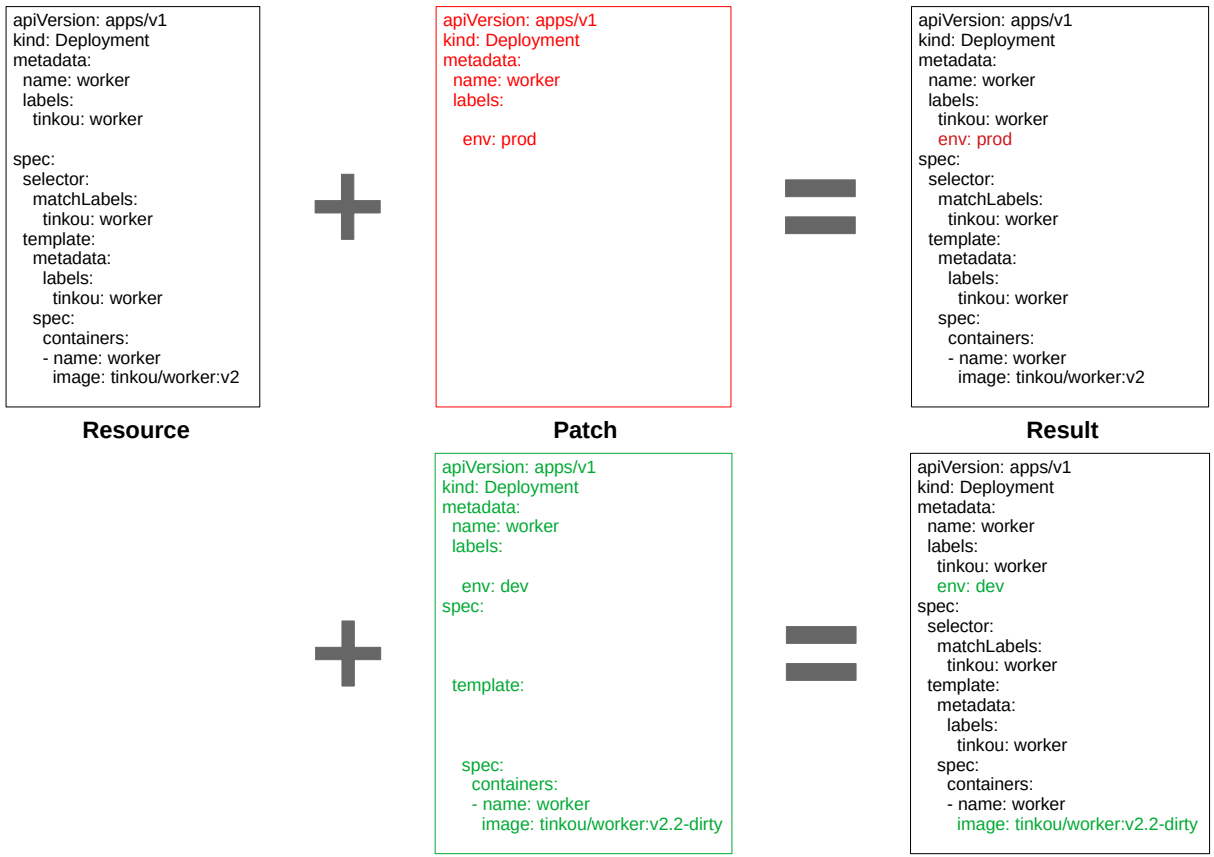
Now that we have a worker in kubernetes, we want to be able to configure it.
Also, we want to be able to differentiate between development and production configuration.

Kustomize among other solutions



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

Kustomize: a layered base configuration



Initializing the base

We create a folder tree to sort our files

Command terminal in folder worker

```
mkdir -p kube/base
cd kube/base
```

Logs terminal still had "`stern worker`" running

Monitor terminal still had "`kubectl get pods -w`" running

Initializing the base

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

worker/kube/base/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
```

Initializing the base

Command terminal in folder worker/kube/base

```
kustomize create --autodetect
kustomize build
kubectl apply -k .
```

Initializing the base

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

Command terminal in folder worker/kube/base

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

Adding a parameter to our worker

Modify the worker to use a parameter:

worker/worker.sh

```
#!/usr/bin/env bash
while true; do
  echo "I'm $NAME and it is $(date)"
  sleep 2
done
```

Deploying the new version in our cluster

First we create a new version of the image:

Command terminal in folder worker

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

Deploying the new version in our cluster

Update source.yaml the field `.spec.template.spec`:

worker/kube/base/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
```

Deploying the new version in our cluster

And finally apply the modification:

Command terminal in folder worker

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

Deploying the new version in our cluster

And finally apply the modification:

Command terminal in folder worker

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

To clean what has been deployed:

Command terminal in folder worker

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

Deploying 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 created to simplify the packaging and deployment on kubernetes.

The Skaffold documentation can be found here.

Initializing skaffold

Let's initialize skaffold:

Command terminal in folder worker

```
skaffold init
```

Follow the application command line interface.

Initializing skaffold

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

worker/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 terminal in folder worker

```
skaffold run
```

Using Skaffold

Let's use Skaffold to build our image:

Command terminal 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 terminal

```
kubectl config current-context
```

Using skaffold

We are going to test a little skaffold commands:

Command terminal 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 terminal in folder worker

```
skaffold dev
```

It works like run, but displays logs and stays in foreground.

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 builds and deploys each time the file is saved.

Isolating the configuration in a configmap

Add a file conf.env in the folder base:

worker/kube/base/conf.env

```
NAME=Trent
```

Second command terminal in the folder worker/kube/base

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

Isolating the configuration in a configmap

worker/kube/base/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
```

Defining a local configuration

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

worker/kube/local/patch.yaml

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

Defining a local configuration

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

worker/kube/local/patch.yaml

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

Nothing happens in skaffold dev.

Defining a local configuration: kustomize

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

Second command terminal in folder worker/kube/local

```
kustomize create --resources ../base
kustomize edit add patch patch.yaml
```

Defining a local configuration: kustomize

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

Second command terminal in folder worker/kube/local

```
kustomize create --resources ../base
kustomize edit add patch patch.yaml
```

Still nothing in skaffold dev.

Defining a local configuration: skaffold

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

worker/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
```

Using skaffold dev modifications detection

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

worker/kube/base/conf.env

```
NAME=Dor
```

Using skaffold dev modifications detection

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

worker/kube/base/conf.env

```
NAME=Dor
```

Skaffold do not react.

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

Cleaning up

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

Questions?

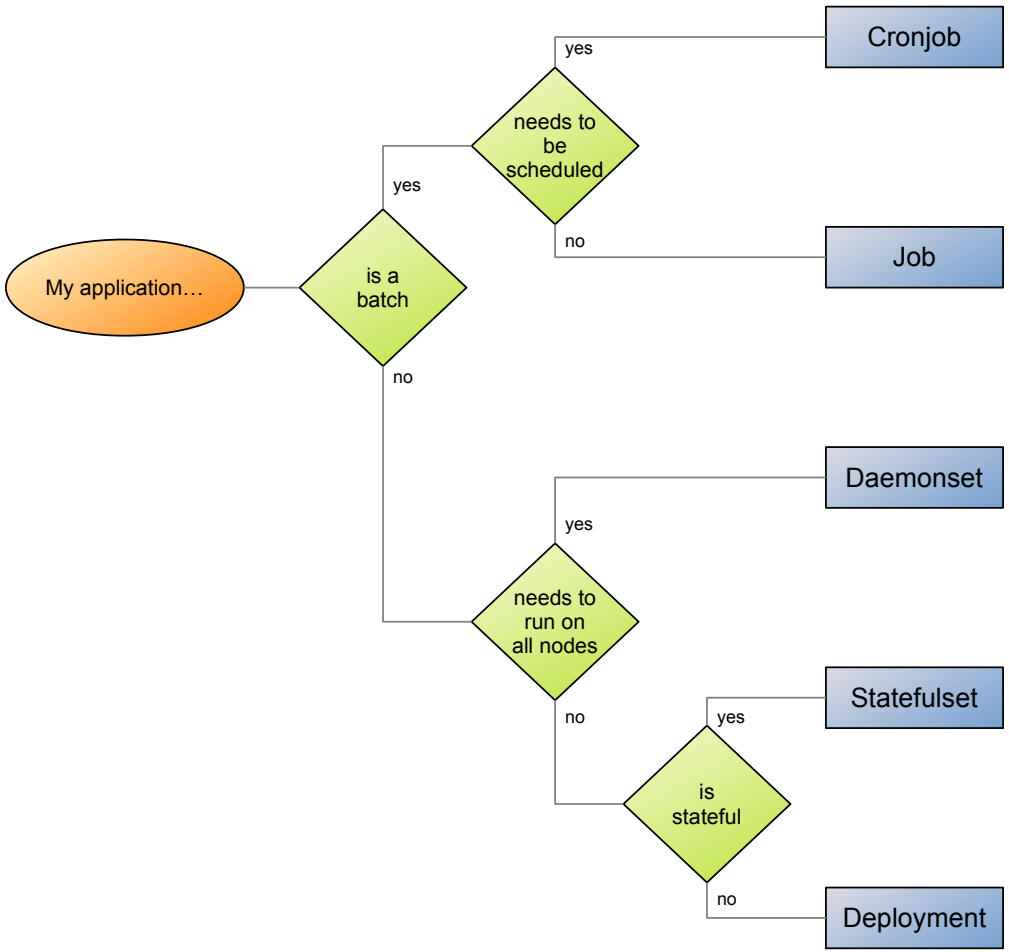
Objectives

Our worker is fine, but induces too much stress on the system.We now want 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 to use to deploy our application



Creating the batch

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

Create the batch itself:

batch/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.

A solution

batch/Dockerfile

```
FROM alpine

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

CMD /batch.sh
```

Command terminal in folder batch

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

Creating our first CronJob

batch/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
          restartPolicy: Never
```

Creating our first CronJob

Logs terminal

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

Monitor terminal

```
watch kubectl get all
```

Command terminal

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

Wait for a few batches to run...

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

Creating our first CronJob

Conclusions

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

CronJobs useful options

- A field .spec.suspend 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

Folder tree in batch folder

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

batch/kube/base/conf.env

```
NAME=Humphrey
```

A solution

batch/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 terminal in folder batch/kube/base

```
kustomize create --resources cronjob.yaml
kustomize edit add configmap batch \
                              --from-env-file conf.env
```

A solution

batch/kube/local/conf.yaml

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

A solution

Command terminal in folder batch/kube/local

```
kustomize create --resources ../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 terminal in folder batch

```
skaffold run
```

Take a quick look at the scheduling options and logs:

Command terminal

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

Clean after working:

Command terminal in folder batch

```
skaffold delete
```

Questions?

Objectives

Our project now needs a web service.

This web service needs to be monitored and scalable.

Creating 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>
```

Creating the web service image

web/Dockerfile

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

Command terminal in folder web

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

Creating kubernetes deployment configuration

web/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
```

Creating kubernetes deployment configuration

Logs terminal

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

Monitor terminal

```
watch kubectl get all
```

Command terminal in folder web

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

Creating kubernetes deployment configuration

A deployment, a replicaset and a pod have been created.

But how to access this web service?

Exposing a web service

Pods have an IP...

Command terminal

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

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

Exposing a web service

Pods have an IP. . .

Command terminal

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

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

—

In kubernetes, pods are exposed 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 terminal in folder web

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

Now we can access 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 its IP given by `minikube ip`
- as a NodePort, the service listen to the port 32001 of all nodes
- the service redirects requests in round robin to pods matching the selector
- pods listen to the same ports as their containers
- at last the container can answer to requests

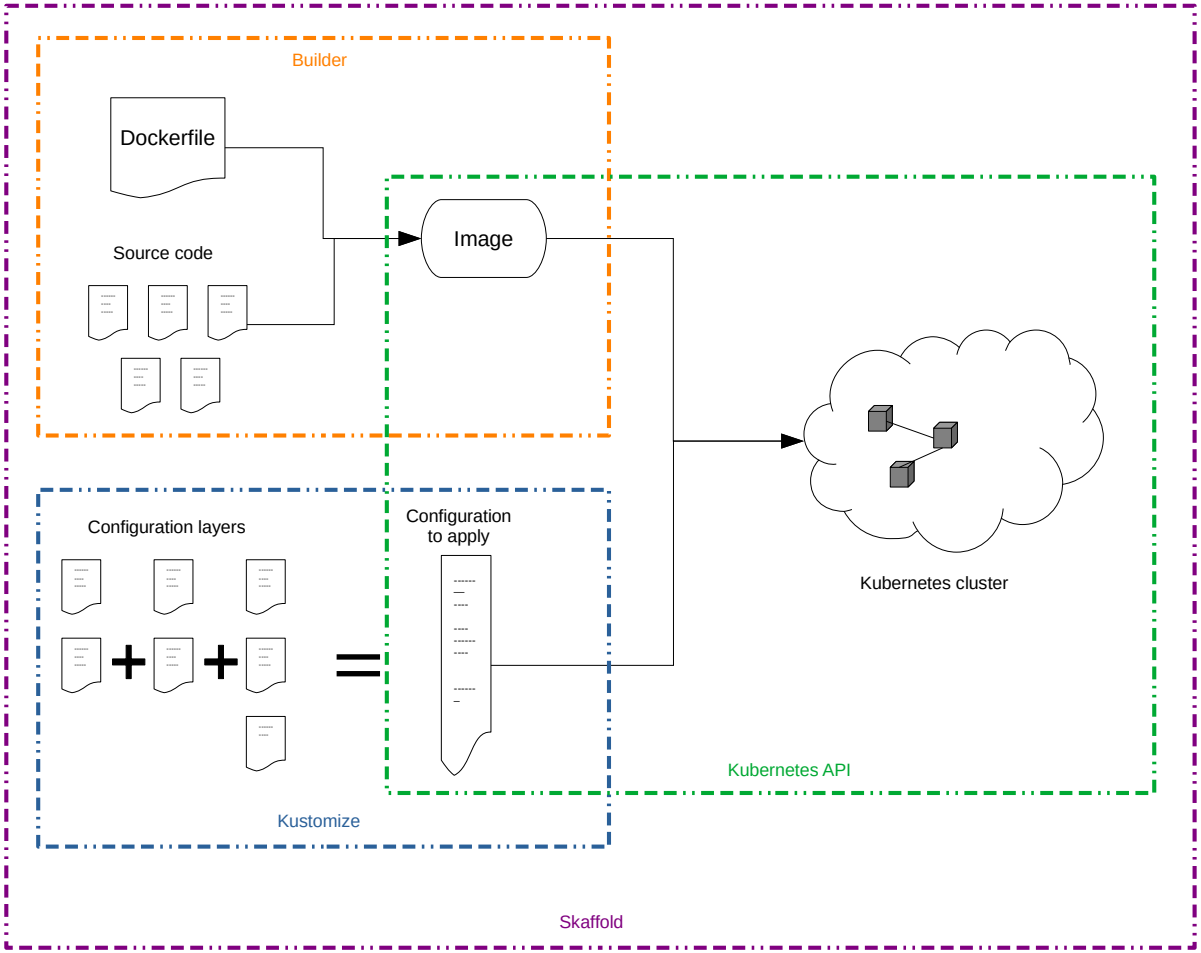
Cleaning up

Command terminal in folder web

```
kubectl delete -f deployment.yaml
kubectl delete -f service.yaml
```

Questions?

Conclusion


Builder
Dockerfile
Source code
Image
Kubernetes cluster
Configuration layers
Configuration
to apply
Kubernetes API
Kustomize
Skaffold

Going further?

With these tools it is possible:

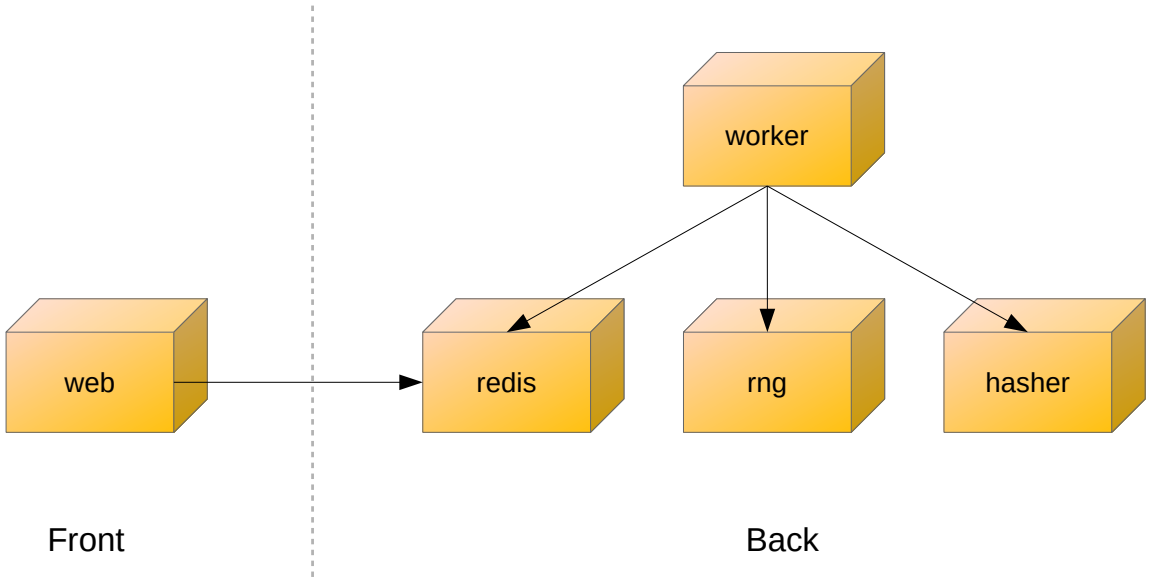
- to use an other image builder than docker (jib, kaniko, ...)
- to have stacks with multiple images
- to use alk kind of kubernetes cluster instead of minikube
- to debug inside deployed containers

A little more complex example

In this github repository, you can found an example consisting of a complete stack.

This is an educational project based on the work of jpetazzo.

A little more complex example



A little more complex example

This project is useful to demonstrate the usage of several languages and to manipulate kubernetes configuration.

Finally, in the branch *exercise*, there is the base configuration and it is possible to use it to try to create kustomize and skaffold configurations.