

Part 2

[slide 11]

(you can copy the files from part 1 to another folder, and start from there)

Let's build a web app.

Go to <http://flask.pocoo.org/>!

```
$ pip install Flask
```

Change `app.py` to:

```
from flask import Flask
app = Flask(__name__)
```

```
@app.route("/")
def hello():
    return "Hello World!"
```

```
app.run(host="0.0.0.0", debug=True)
```

Run the app and check <http://0.0.0.0:5000/>:

```
$ python app.py
```

Remove the `sleeper` from `docker-compose.yml`:

```
version: "3"
services:
  app:
    build: .
```

Run `docker-compose up --build` and check <http://0.0.0.0:5000/>.

It doesn't work. Why?

We need to expose and publish the container's port 5000 to the host (our machine):

```
version: "3"
services:
  app:
    build: .
    ports:
      - "3333:5000"
```

The above publishes container's port 5000 on host's port 3333.

Run `docker-compose up` and <http://0.0.0.0:3333/>.

Deploy it on Kubernetes

[slide 12]

Create file `app.yml` (simply based on Kubernetes 101):

```
apiVersion: v1
kind: Pod
metadata:
  name: app
  labels:
    foo: vitor
spec:
  containers:
  - name: app
    image: vitorenesduarte/tutorial
```

(Compared to 101, we added `foo: vitor` as a label, because we'll need it later, when we want to have this pod behind a service)

And deploy it on Kubernetes.

(for that you need `CONFIG`, a Kubernetes configuration file, which I will provide)

```
$ kubectl --kubeconfig=CONFIG create -f app.yml
```

QUESTION: will this work?

```
$ kubectl --kubeconfig=CONFIG get pods
NAME          READY   STATUS             RESTARTS   AGE
app           0/1     ErrImagePull       0           4s
```

Ups. The docker image is still local.

Let's push it to Docker Hub.

Create an account there, and login with `docker login`. Then:

```
$ docker build -t vitorenesduarte/tutorial .
$ docker push vitorenesduarte/tutorial
```

Before anything else, let's avoid always having to specify `--kubeconfig`.

Let's check the manual.

```
$ kubectl config --help | sed -n '5,7p'
```

We can simply have `$KUBECONFIG` environment variable pointing to the `CONFIG` file, e.g.:

```
$ export KUBECONFIG=$(pwd)/CONFIG
```

Now, let's delete the `app` pod and deploy again.

```
$ kubectl delete pod app
$ kubectl get pods
$ kubectl create -f app.yml
$ kubectl get pods --watch
$ kubectl logs -f app
```

Add RUN apk update && apk add curl to the Dockerfile, build and deploy again, so that you can:

```
$ kubectl exec app curl localhost:5000
```

Create a load balancer so that we can access our app:

[slide 13]

(Something similar to what we're doing next, would be:

```
$ kubectl expose -f app.yml \
    --name=app-service \
    --type=LoadBalancer \
    --port 3333 \
    --target-port 5000
```

)

Create file app-service.yml:

```
apiVersion: v1
kind: Service
metadata:
  name: app-service
spec:
  type: LoadBalancer
  ports:
    - port: 3333
      targetPort: 5000
  selector:
    foo: vitor
```

```
$ kubectl create -f app-service.yml
$ kubectl get service app-service
```

Watch until EXTERNAL-IP is no longer ‘:

```
$ kubectl get service app-service --watch
```

And then go to <http://EXTERNAL-IP:3333>:

Does the load balancing work?

Let's slightly change our app, so that each pod has an identifier.

```
from flask import Flask
app = Flask(__name__)

import sys
id = sys.argv[1] if len(sys.argv) > 1 else "ups!"

@app.route("/")
def hello():
    return "Hello World! (from " + id + ")"

app.run(host="0.0.0.0", debug=True)
```

Change the Dockerfile, so that we can pass the pod identifier as an environment variable \$ID:

```
FROM python:alpine
```

```
RUN pip install flask
```

```
COPY app.py /
```

```
CMD python app.py $ID
```

Change app.yml so that we run two pods with different \$ID:

```
apiVersion: v1
kind: Pod
metadata:
  name: app-1
  labels:
    foo: vitor
spec:
  containers:
  - name: app
    image: vitorenesduarte/tutorial
    imagePullPolicy: Always
    env:
    - name: ID
      value: "1"
---
apiVersion: v1
kind: Pod
```

```
metadata:
  name: app-2
  labels:
    foo: vitor
spec:
  containers:
  - name: app
    image: vitorenesduarte/tutorial
    imagePullPolicy: Always
    env:
    - name: ID
      value: "2"
```

(Note imagePullPolicy: Always: this will force Kubernetes to pull a new image, even if it already has it)

Let's build a new image, push it, delete the previous pod, and deploy the pods again:

```
$ docker build -t vitorenesduarte/tutorial .
$ docker push vitorenesduarte/tutorial
$ kubectl delete pod app
$ kubectl create -f app.yml
$ kubectl get pods --watch
```

In two different terminals:

```
$ kubectl logs -f app-1
$ kubectl logs -f app-2
```

Now go to <http://EXTERNAL-IP:3333>, and see the identifier changing, and see the logs of the two pods.

[slide 14]

[slide 15]