We need two identical Docker images with installed Flask (in requirements.txt Flask==0.10.1). Dockerfile is as below:

FROM ubuntu:latest

RUN apt-get update -y && \

apt-get install -y python-pip python-dev

COPY ./app/requirements.txt /app/requirements.txt

WORKDIR /app

RUN pip install -r requirements.txt

COPY ./app/app.py /app

ENTRYPOINT [ "python" ]

CMD [ "app.py" ]

Above nodes are called diginex-front and diginex-back. Back is one which does reverting. Code of backend is below:

import flask

import json

from flask import request

from random import seed

from random import random

app = flask.Flask(\_\_name\_\_)

app.config["DEBUG"] = True

def reverting\_message(message):

reversedstring = ''.join(reversed(message))

return reversedstring

@app.route('/reverse', methods=['POST'])

def api\_post():

content = request.get\_json()

print ("Received json: ",content)

reverted = reverting\_message(content['message'])

response = app.response\_class(

response=json.dumps({

"message":reverted,

}),

status=200,

mimetype='application/json'

)

return response

app.run(host='0.0.0.0', port=6000)

It just listen in port 6000 and on received POST on /reverse kick function reverting\_message. This is returned back as JSON.

Frontend code is slightly longer.

import flask

import json

import requests

from flask import request

from random import seed

from random import random

app = flask.Flask(\_\_name\_\_)

app.config["DEBUG"] = True

#generating random number

seed(1)

def random\_number():

for \_ in range(1):

value = random()

return value

random\_number = random\_number()

def reverting\_message\_remote(message):

try:

res = requests.post('http://jak.slave.arkdevs.ee:6000/reverse', json={"message":message})

if res.ok:

print ("Response from backend ", res.json())

return res.json()

except:

print "Can't call backend"

finally:

res.close

@app.route('/', methods=['GET'])

def home():

return "<h1>Test</h1><p>This site is Diginex Test</p>"

@app.route('/api', methods=['POST'])

def api\_post():

content = request.get\_json()

reverted = reverting\_message\_remote(content['message'])

response = app.response\_class(

response=json.dumps({

"message":reverted['message'],

"random":random\_number

}),

status=200,

mimetype='application/json'

)

return response

app.run(host='0.0.0.0', port=5000)

This one is listening on port 5000. Now couple of comments:

1. Above two applications are running in my datacenter, you can test it with simply curl (curl -XPOST -H "Accept: application/json" -H "Content-Type: application/json" -d '{"message":"abcdef"}' jak.slave.arkdevs.ee:5000/api)
2. This is basic version of code, there is much more things you can do to make it better and more resilient but I am not sure how far we can go with that. For instance I would set maximum length of received message. Secondary hardcoding of ports and hosts is poor idea, this should be passed as env variables. Thirdly I would add more validators, of received and sent messages.
3. I used Flask as it is most lightweight solution I know. In real live (I have my own company – Arkdevs OU, Tallin, Estonia) I am using Django as API and admin panel for backend for mobiles apps I am developing.

To build and start above two images, and run this without orchestrator I am using below commands.

docker build -t diginex-front:latest .

docker run -it -p 5000:5000 diginex-front

docker build -t diginex-back:latest .

docker run -it -p 6000:6000 diginex-back

Please note that I am using “latest” as I will go to CI/CD below.

I am using CI/CD in two steps, first step is to rebuild image. It can be automated with SCM Git listener. Above two jobs of course are running on my Jenkins.

node {

println "${BRANCH}"

println "${params.BRANCH}"

def newImage = null

stage("clean up workspace") {

cleanWs()

}

stage('Clone sources') {

git branch: "${params.BRANCH}", credentialsId: \*\*\*, url: 'git@github.com:drjak/\*\*\*-API.git'

}

stage("create image tag") {

withCredentials([usernameColonPassword(credentialsId: '2f124f95-cee9-419a-902a-7206df43b1e7', variable: 'CREDENTIALS')]) {

env.VERSION = sh (returnStdout: true, script:'''

curl -s -u $CREDENTIALS -X GET "http://jak.slave.arkdevs.ee:8084/v2/${IMAGE}/tags/list" -H "accept: application/json" | jq -r '.tags // ["1.0.0"] | sort\_by(. | split(".") |map(tonumber) )' | jq -r '.[-1]'

''').trim()

versions = env.VERSION.split('\\.')

def patch = versions[2].toInteger() + 1

imageVersion = versions[0] + '.' + versions[1] + '.' + patch

println("imageVersion=${imageVersion}")

}

}

stage("build image") {

newImage = docker.build("jak.slave.arkdevs.ee:8082/${IMAGE}:${imageVersion}")

}

stage("push image") {

docker.withRegistry('https://jak.slave.arkdevs.ee:8082', \*\*\*\*) {

newImage.push()

}

}

stage("remove local image") {

sh "docker rmi jak.slave.arkdevs.ee:8082/${IMAGE}:${imageVersion}"

}

}

Sensitive information is starred. I am hosting my own Nexus on the same host as hardcoded in python. Above are doing simple steps:

1. Cloning new code from git
2. Creating new tag
3. Building image
4. Pushing image
5. Cleaning

My Jenkins and Nexus are available at: <http://jak.slave.arkdevs.ee:8081/#browse/welcome>

And <http://jak.slave.arkdevs.ee:8080/>

To deploy this I would use Kubernetes Rolling Deployment, so updating one by one with new image. Currently my process is not entirely automated and consist some manual steps, once apps for my customers will grow up it will be fully automated.

1. Remove node from Cloud Flare load balancer, so it is not receiving traffic
2. Taint and in Kubernetes so it is out of cluster
3. Delete PODs by reducing number of replicas in Replica Set, or delete it manually with kubectl
4. Re-recreate with increasing “Desired number of pods” in Dashboard (or manually) (link to mine <https://k8sm.slave.arkdevs.ee:30003>)
5. Make sure POD is recreated and balanced eventually and host is back to cluster.
6. Put back host to Cloud Flare loadbalancer
7. Repeat steps for all other nodes

With small environments like mine it is not very time consuming to do this manually but with larger environments it would need to be up automated.

Infrastructure: As mentioned I have couple of servers (both cloud and hardware) and Cloud Flare account. All requests are going via CF load balancers and hosts are in LB pool. We set health check so if any problem we getting immediate notification. Our nodes are using GlusterFS and Heketi, so data persist during rollout all are running Ubuntu. Application is accessed via Ingress.

I haven’t been using AWS or GPC lately, but I would deploy to AWS EC2 with Terraform (this is what we are using in Sky) and Ansible to install software/set node.