

# AMIS SIG Introduction Microservice Choreography with Docker, Kubernetes Node.js and Kafka –Hands-on

---

1<sup>st</sup> June 2017

The ultimate goal of this workshop is to achieve microservice choreography. We will get there by implementing microservices as Dockerized Node.JS applications that run on Kubernetes and leverage microservice platform facilities such as a cache and an event bus. This event bus (Apache Kafka) provides the backbone for the choreography that will have microservices participate in a dance that no one orchestrates.

You will go through a number of steps in this workshop – that have you work with (and install) Docker, Kubernetes, Redis, Node.js, Apache Kafka and MongoDB.

You can get access to the sources for the practices from the GitHub repository:

<https://github.com/lucasjellema/microservices-choreography-kubernetes-workshop-june2017> .

## 1. Prepare a local Kubernetes and Docker environment

We will work with Docker in this workshop. You will be running multiple Docker Containers and have them interact. Subsequently, we will be using Kubernetes and its minikube cluster.

### Installation

The installation we have to do before getting started with this workshop depends a little on your operating system – and of course the software you may already have set up on it. What we need to work with is at least:

- Docker
- VirtualBox
- Kubectl
- Minikube
- Moba Xterm

On Windows and MacOS – without native support for Docker – we also need Docker Toolbox (or a VM running Linux) and MobaXterm (or another SSH client).

#### Windows and MacOS: Docker Toolbox and VirtualBox

I will assume an older version of Windows or MacOS that does not have native Docker support. If that is your case, You need to:

- Download & Install the latest [Docker Toolbox](#) (this is not the same thing as just installing Docker)
- Download & install [VirtualBox](#) for Windows or OS X. Direct link to the binaries [here](#) (minikube relies on some of the drivers)

VirtualBox: go to the VirtualBox Downloads page: <https://www.virtualbox.org/wiki/Downloads> .

Download the latest installer for Windows or MacOS. Run the installer to install VirtualBox.

Docker Toolbox: [https://docs.docker.com/toolbox/toolbox\\_install\\_windows/](https://docs.docker.com/toolbox/toolbox_install_windows/) or  
[https://docs.docker.com/toolbox/toolbox\\_install\\_mac/](https://docs.docker.com/toolbox/toolbox_install_mac/) (the steps: download the installer, run the installer, run the Docker Quickstart Terminal window to create the *default* machine)

## Linux: Docker and VirtualBox

If your operating system is Linux to start with, you still need to install Docker and VirtualBox as well as Kubectl; you do not need Docker Toolbox.

Go to <https://www.docker.com/community-edition> and get the installer for your Linux distribution. Run the installer to set up Docker in your environment.

VirtualBox: go to the VirtualBox Downloads page: <https://www.virtualbox.org/wiki/Downloads> .

Download the latest installer for Linux. Run the installer to install VirtualBox.

## Install Minikube and Kubectl on all operating systems

For Minikube and Kubectl – follow instructions at <https://kubernetes.io/docs/tasks/tools/install-minikube/>. These involve the installation of *kubectl* and *minikube*. The

- kubectl from
    - MacOS: <http://storage.googleapis.com/kubernetes-release/release/v1.4.0/bin/darwin/amd64/kubectl>
    - Linux: <http://storage.googleapis.com/kubernetes-release/release/v1.4.0/bin/linux/amd64/kubectl>

- Windows: <http://storage.googleapis.com/kubernetes-release/release/v1.4.0/bin/windows/amd64/kubectl.exe>
- minikube from <https://github.com/kubernetes/minikube/releases>

## Some resources:

Tutorial : Getting Started with Kubernetes on your Windows Laptop with Minikube -

<https://rominirani.com/tutorial-getting-started-with-kubernetes-on-your-windows-laptop-with-minikube-3269b54a226>

<https://codefresh.io/blog/kubernetes-snowboarding-everything-intro-kubernetes/>

Minikube on Windows7: <https://quip.com/1TYDAdJowAgJ>

Running Kubernetes Locally via Minikube - <https://kubernetes.io/docs/getting-started-guides/minikube/>

Kubernetes Cheat Sheet: <https://kubernetes.io/docs/user-guide/kubectl-cheatsheet/>

## Run Minikube Single Node Cluster

To run minikube – and have the one-node cluster initialized (in a VirtualBox VM):

```
minikube start
```

```
(minikube start --insecure-registry localhost:5000)
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>minikube.exe start
Starting local Kubernetes cluster...
Starting VM...
SSH-ing files into VM...
Setting up certs...
Starting cluster components...
Connecting to cluster...
Setting up kubeconfig...
Kubectl is now configured to use the cluster.
```

```
minikube dashboard --url=true
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>minikube dashboard --url=true
http://192.168.99.101:30000
```

minikube status

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>minikube status
minikubeVM: Running
localkube: Running
```

To get the IP address of the Minikube cluster:

```
minikube.exe ip
```

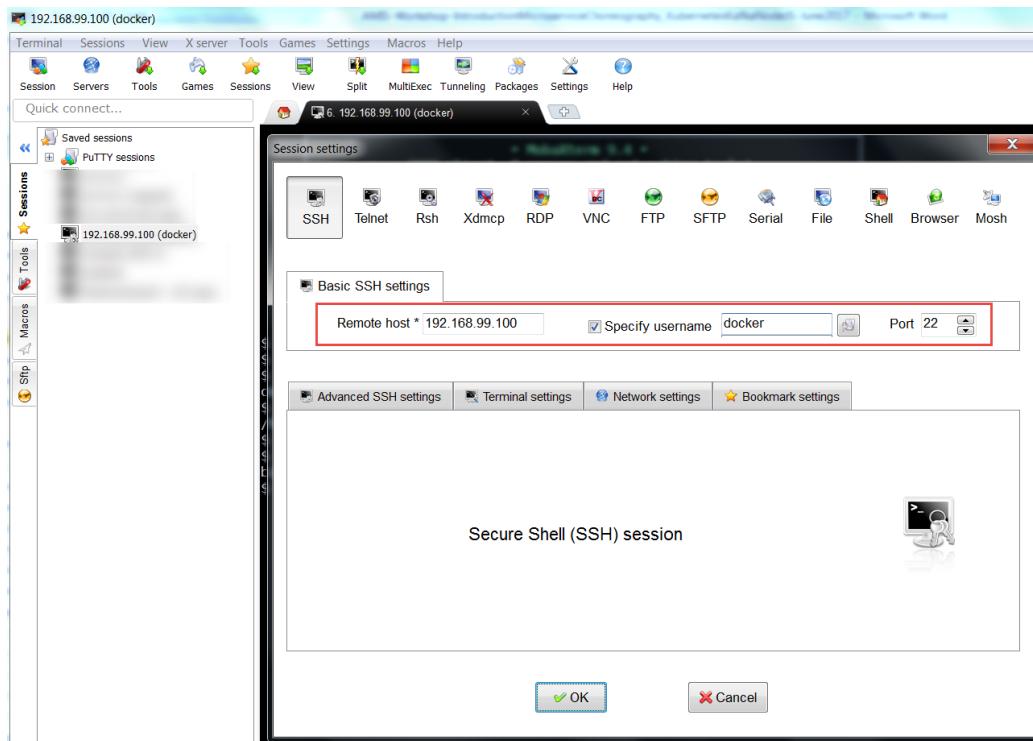
```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>minikube.exe ip
192.168.99.101
```

To check on the nodes of the cluster, we can do:

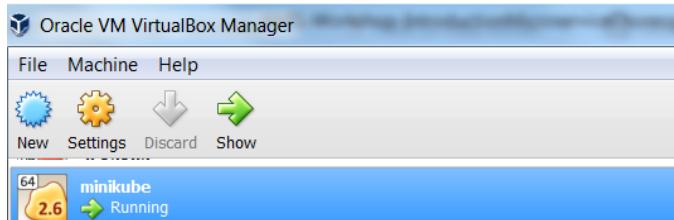
```
kubectl.exe get nodes
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl.exe get nodes
NAME      STATUS    AGE      VERSION
minikube  Ready     7h       v1.6.0
```

Test connect into the minikube VM through an SSH session:

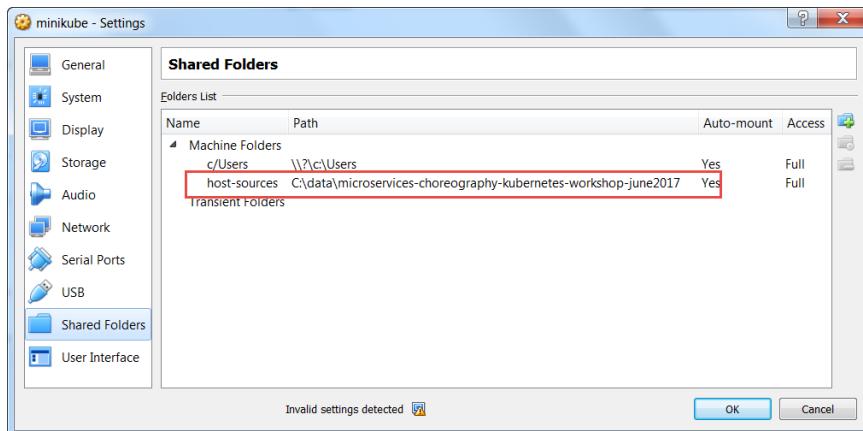
The IP address was retrieved above. The port to use is 22 and username is *docker*, password is *tcuser*.



Note: after running for the first time, a new VM called minikube will show up in the Virtual Box GUI.



When the minikube cluster is down – minikube stop – you can configure this VM definition, for example to add shared folders:



That can be accessed later on in an SSH session:

```
$ pwd  
/host-sources  
$ ls  
AMIS-Workshop-IntroductionMicroserviceChoreography_KubernetesKafkaNodeJS-June2017.docx  
AMIS-Workshop-IntroductionMicroserviceChoreography_KubernetesKafkaNodeJS-June2017.pdf  
'OracleCode_EventBusMicroserviceChoreography_20april2017 - Copy.pptx'  
README.md  
WorkshopEventBusMicroserviceChoreography_June2017.pptx  
images-archive.tar  
part1  
part2  
part3  
part4  
part5
```

## Run Something on the MiniKube Cluster

Now in order to run a first [Docker container image in a Kubernetes] Pod on the cluster:

```
# deploy Docker container image nginx:  
kubectl run my-nginx --image=nginx --replicas=2 --port=80
```

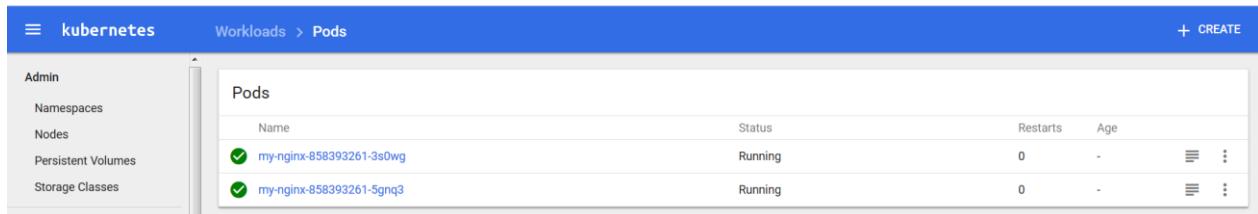
```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl run my-nginx --image=nginx --replicas=2 --port=80  
deployment "my-nginx" created
```

A Pod is started on the cluster with a single container based on the nginx Docker container image. Two replicas of the Pod will be kept running.

```
# as the result of the above, you will see pods and deployments  
kubectl get pods  
kubectl get deployments
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl get pods  
NAME          READY   STATUS    RESTARTS   AGE  
my-nginx-858393261-3s0wg  1/1     Running   0          4m  
my-nginx-858393261-5gnq3  1/1     Running   0          4m  
  
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl get deployments  
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE  
my-nginx  2          2         2           2           6m
```

In the Dashboard:



Name	Status	Restarts	Age
my-nginx-858393261-3s0wg	Running	0	4m
my-nginx-858393261-5gnq3	Running	0	4m

At this moment, the my-nginx containers cannot be accessed from outside the cluster. They need to be exposed through a Service:

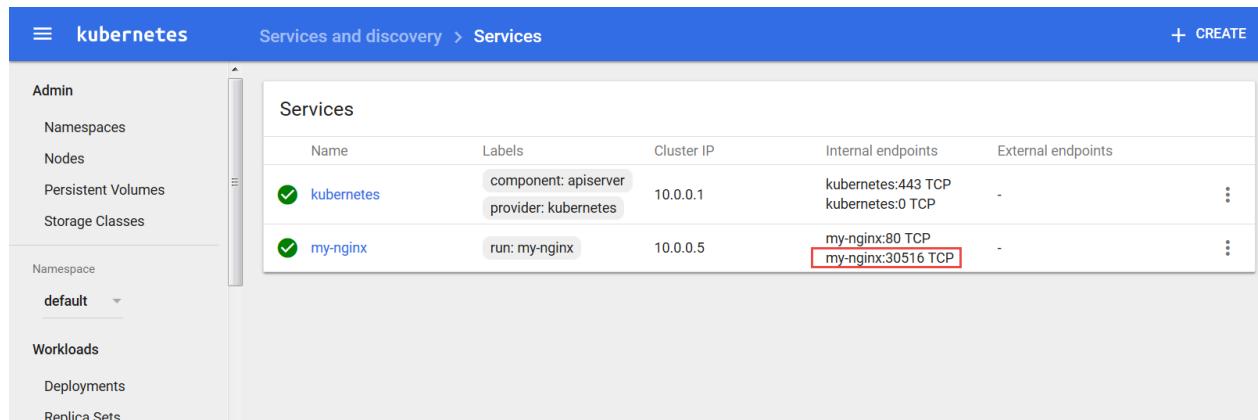
```
# expose your deployment as a service
kubectl expose deployment my-nginx --type=NodePort
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl expose deployment my-nginx --type=NodePort
service "my-nginx" exposed
```

```
# check your service is there
kubectl get services
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl get services
NAME      CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
kubernetes  10.0.0.1    <none>        443/TCP       7h
my-nginx   10.0.0.5    <nodes>       80:30516/TCP  43s
```

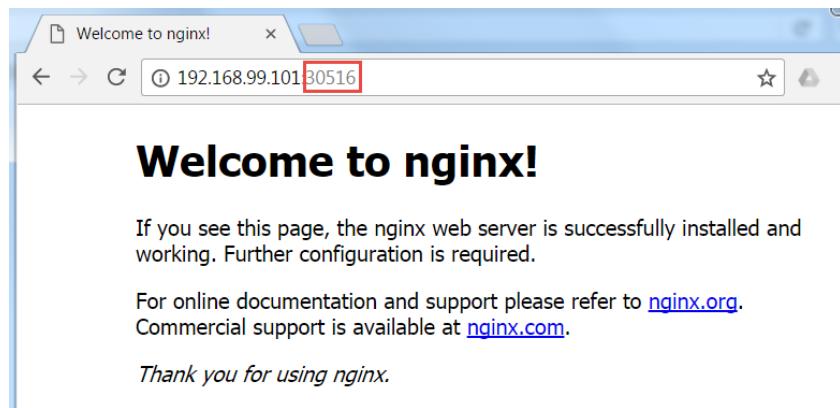
And in the Dashboard:



The screenshot shows the Kubernetes Dashboard's Services page. On the left, a sidebar lists Admin, Namespaces, Nodes, Persistent Volumes, Storage Classes, Namespace (set to default), Workloads, Deployments, and Replica Sets. The main area is titled 'Services' and contains a table with the following data:

Name	Labels	Cluster IP	Internal endpoints	External endpoints
kubernetes	component: apiserver provider: kubernetes	10.0.0.1	kubernetes:443 TCP kubernetes:0 TCP	-
my-nginx	run: my-nginx	10.0.0.5	my-nginx:80 TCP	my-nginx:30516 TCP

```
# Access your service from your default browser
minikube service my-nginx
```



If you are interested in the logging from one of the Pods, you can get to that logging in the dashboard. From the Services tab, drill down to a specific Service. Then click on the icon for the Pod that you are interested in:

The screenshot shows the Kubernetes UI for a service named 'my-nginx'. In the 'Details' section, it lists the service's name, namespace, labels, creation time, and connection information (Cluster IP: 10.0.0.5, Internal endpoints: my-nginx:80 TCP, my-nginx:30516 TCP). Below this, the 'Pods' section shows two pods: 'my-nginx-858393261-3s0wg' and 'my-nginx-858393261-5gnq3', both of which are running.

The logging will be shown:

The screenshot shows the Kubernetes UI for logs. It displays log entries from the 'my-nginx' pod. The logs show multiple requests from various clients (Mozilla/5.0, Safari/537.36, etc.) for favicon.ico files, with some errors indicating file not found.

```

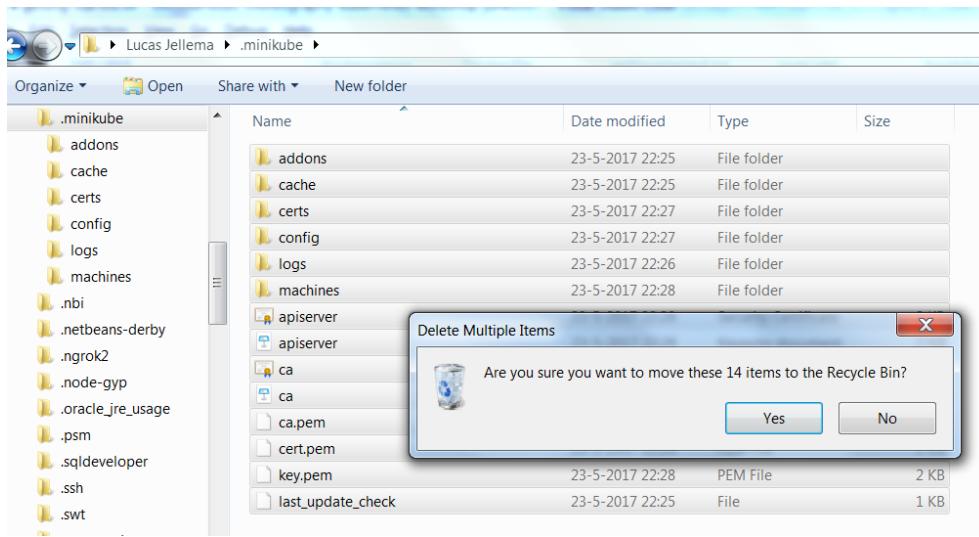
Logs from my-nginx in my-nginx-858393261-3s0wg
A Tr
2017-05-24T04:25:11.252435427Z 172.17.0.1 - - [24/May/2017:04:25:11 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"
2017-05-24T04:25:11.357433307Z 2017/05/24 04:25:11 [error] #6#6: *1 open() "/usr/share/nginx/html/favicon.ico" failed (2: No such file or directory), client: 172.17.0.1, server: localhost, request: "GET /favicon.ico HTTP/1.1", host: "192.168.99.101:30516", referer: "http://192.168.99.101:30516/"
2017-05-24T04:25:11.357476392Z 172.17.0.1 - - [24/May/2017:04:25:11 +0000] "GET /favicon.ico HTTP/1.1" 404 571 "http://192.168.99.101:30516/" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"
2017-05-24T04:25:12.519758152Z 172.17.0.1 - - [24/May/2017:04:25:12 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:53.0) Gecko/20100101 Firefox/53.0"
2017-05-24T04:25:12.666913685Z 2017/05/24 04:25:12 [error] #6#6: *2 open() "/usr/share/nginx/html/favicon.ico" failed (2: No such file or directory), client: 172.17.0.1, server: localhost, request: "GET /favicon.ico HTTP/1.1", host: "192.168.99.101:30516"
2017-05-24T04:25:12.666913701Z 172.17.0.1 - - [24/May/2017:04:25:12 +0000] "GET /favicon.ico HTTP/1.1" 404 169 "-" "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:53.0) Gecko/20100101 Firefox/53.0"
2017-05-24T04:25:12.668688904Z 2017/05/24 04:25:12 [error] #6#6: *2 open() "/usr/share/nginx/html/favicon.ico" failed (2: No such file or directory), client: 172.17.0.1, server: localhost, request: "GET /favicon.ico HTTP/1.1", host: "192.168.99.101:30516"
2017-05-24T04:25:12.668686935Z 172.17.0.1 - - [24/May/2017:04:25:12 +0000] "GET /favicon.ico HTTP/1.1" 404 169 "-" "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:53.0) Gecko/20100101 Firefox/53.0"

```

At this point, you can remove the Deployment, Service and Pods for nginx.

### Tip

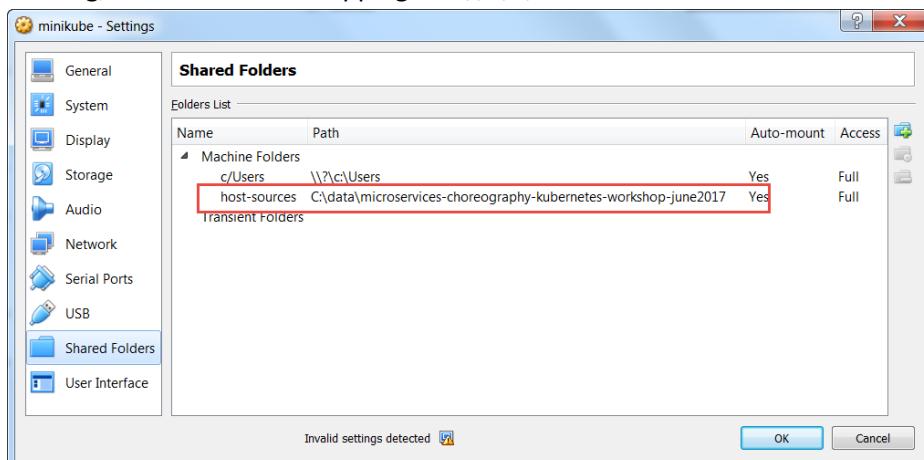
Note: if you have trouble creating, running or restarting minikube, it may help to clear the directory .minikube under the current user directory:



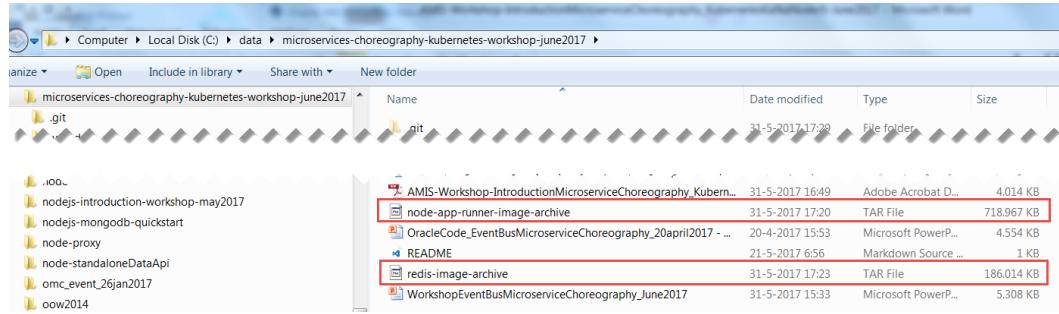
### **Tip 2 - No Internet Connection:**

You need an internet connection to download Docker Container images. If none is available, we can load images directly from a tar ball into the Docker Server. The steps for this are:

- create a shared folder mapping from a local directory on your laptop into the minikube virtual machine (by configuring a shared folder in VirtualBox when minikube is down). Note: remove the existing, incorrect folder mapping for \\?\c\Users.



- copy tar balls for images to that local directory



- start minikube
- retrieve ip address for minikube vm: minikube ip
- open an SSH session into minikube ; The IP address was retrieved above. The port to use is 22 and username is docker, password is tcuser.

```
$ cd /host-sources/
$ cd microservices-choreography-kubernetes-workshop-june2017/
$ ls -l *.tar
-rwxrwxrwx 1 docker docker 736222208 May 31 15:19 node-app-runner-image-archive.tar
-rwxrwxrwx 1 docker docker 190478336 May 31 15:23 redis-image-archive.tar
```

- load tar balls into docker using docker load --input <tar ball>

docker load --input node-app-runner-image-archive.tar  
 docker load --input redis-image-archive.tar  
 docker load --input nginx-image-archive.tar

```
$ docker load --input node-app-runner-image-archive.tar
8d4d1ab5ff74: Loading layer [=====] 129.4 MB/129.4 MB
c59fa6cbcd9: Loading layer [=====] 45.52 MB/45.52 MB
445ed6ee6867: Loading layer [=====] 126.9 MB/126.9 MB
e7b0b4cd055a: Loading layer [=====] 330.9 MB/330.9 MB
2607b744b89d: Loading layer [=====] 352.3 kB/352.3 kB
0f20784b55ff: Loading layer [=====] 137.2 kB/137.2 kB
a8d5a17bf5cc: Loading layer [=====] 49.57 MB/49.57 MB
3e88edcc5f79: Loading layer [=====] 3.731 MB/3.731 MB
56569c4031cc: Loading layer [=====] 2.56 kB/2.56 kB
ac5c421a7340: Loading layer [=====] 2.56 kB/2.56 kB
54d7cead58bf: Loading layer [=====] 133.1 kB/133.1 kB
f6121c79f746: Loading layer [=====] 49.57 MB/49.57 MB
```

show the images:

REPOSITORY	TAG	IMAGE_ID	CREATED	SIZE
Lucasjellema/node-app-runner	latest	6bb8effa98f4	10 days ago	713.1 MB
redis	latest	a858478874d1	12 days ago	183.7 MB
gcr.io/google_containers/kubernetes-dashboard-amd64	v1.0.0	416701f962f2	10 weeks ago	168.0 MB
gcr.io/google-containers/kube-addon-manager	v6.4-alpha.1	9da55e306d47	3 months ago	67.57 MB
gcr.io/google_containers/kubedns-amd64	1.9	26cf1ed9b144	6 months ago	47 kB
gcr.io/google_containers/kube-dnsmasq-amd64	1.4	3ec65756a89b	8 months ago	5.126 MB
gcr.io/google_containers/exechealthz-amd64	1.2	93a43bfb39bf	8 months ago	8.375 MB
gcr.io/google_containers/pause-amd64	3.0	99e59f495ffa	13 months ago	746.9 kB

- at this point, Pods based on these images can be started, even when no internet connection is available

## 2. Run your first Microservice

In this section, we will run a simple microservice. First as stand-alone Docker container, then in a Pod on the Kubernetes cluster. The microservice is implemented by a Node.js application – requestCounter – that accepts HTTP requests, counts request and returns the current count as the HTTP response.

### Run RequestCounter in Docker

If you are using Docker Tools, then run the Docker Quickstart Terminal. On the command line in Docker Quickstart Terminal, run this command:

```
docker-machine ip default
```

It will return the IP address on your laptop assigned to the Docker VM that will run the Docker containers.

```
$ docker-machine ip default  
192.168.99.100
```

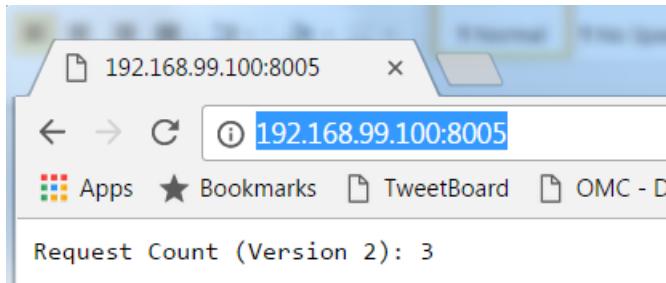
On Docker command line – either in Linux server with Docker installed or in Docker Quickstart Terminal – run a container with the requestCounter.js Node.JS application:

```
docker run -e "GIT_URL=https://github.com/lucasjellema/microservices-choreography-kubernetes-workshop-june2017" -e "APP_PORT=8080" -p 8005:8080 -e "APP_HOME=part1" -e "APP_STARTUP=requestCounter-2.js" lucasjellema/node-app-runner
```

```
efef6f7a9be92: Pull complete  
Digest: sha256:02de5b7e9ff9111e8a8a76a91067d3061dd0d7b1fcddace5fe48965424420bcc  
Status: Downloaded newer image for lucasjellema/node-app-runner:latest  
switching node to version stable  
node version: v7.10.0  
Cloning into 'app'...  
Application Home: part1  
request-counter@0.0.2 /code/app/part1  
+-- node-redis@0.1.7  
`-- redis@2.7.1  
  +-- double-ended-queue@2.1.0-0  
  +-- redis-commands@1.3.1  
  '-- redis-parser@2.6.0  
  
Node.JS Server running on port 8080 for version 2 of requestCounter application.
```

When the microservice is running, it can be accessed on your laptop in the browser, at the IP address returned by docker-machine ip default and at port 8005 – the host port mapped to the container port 8080 in the docker run startup command.

<http://192.168.99.100:8005>



Execute the Docker run command again, with a small twist: change 8005 to 8006:

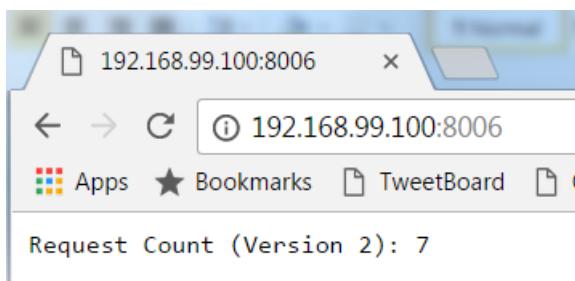
```
docker run -e "GIT_URL=https://github.com/lucasjellema/microservices-choreography-kubernetes-workshop-june2017" -e "APP_PORT=8080" -p 8006:8080 -e "APP_HOME=part1" -e "APP_STARTUP=requestCounter-2.js" lucasjellema/node-app-runner
```

A second container will be started for you, running the same application, listening at the same port – inside the container – and mapped to a different port at the host:

```
$ docker run -e "GIT_URL=https://github.com/lucasjellema/microservices-choreography-kubernetes-workshop-june2017" -e "APP_PORT=8080" -p 8006:8080 -e "APP_HOME=part1" -e "APP_STARTUP=requestCounter-2.js" lucasjellema/node-app-runner
switching node to version stable
node version: v7.10.0
Cloning into 'app'...
Application Home: part1
request-counter@0.0.2 /code/app/part1
+-- node-redis@0.1.7
`-- redis@2.7.1
  +-- double-ended-queue@2.1.0-0
  +-- redis-commands@1.3.1
  `-- redis-parser@2.6.0

Node.JS Server running on port 8080 for version 2 of requestCounter application.
```

<http://192.168.99.100:8006>



Use docker ps to check on the two containers currently running:

CONTAINER ID	IMAGE NAMES	COMMAND	CREATED	STATUS	PORTS
c2dd31dd3d20	lucasjellema/node-app-runner	"/code/bootstrap.sh"	5 minutes ago	Up 5 minutes	0.0.0.
0:8006->8080/tcp	practical_visvesvaraya				
a13f8eaeef363	lucasjellema/node-app-runner	"/code/bootstrap.sh"	9 minutes ago	Up 8 minutes	0.0.0.
0:8005->8080/tcp	relaxed_bartik				

Stop these containers using

docker stop <first three characters of container id>

```
$ docker stop c2dd
$ docker stop a13
$ docker ps
CONTAINER ID        IMAGE               COMMAND
NAMES
```

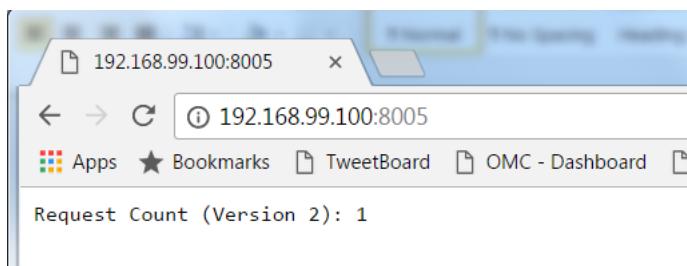
Then use docker run to start a fresh container :

```
docker run -e "GIT_URL=https://github.com/lucasjellema/microservices-choreography-
kubernetes-workshop-june2017" -e "APP_PORT=8080" -p 8005:8080 -e "APP_HOME=part1" -e
"APP_STARTUP=requestCounter-2.js"  lucasjellema/node-app-runner
```

Access the microservice offered by this container once more.

<http://192.168.99.100:8005>

Verify the value returned by the request counter. Is it in line with the total number of calls you have made to the service?



## Run and Expose the microservice on Kubernetes

Docker can run individual containers just fine. However, creating microservices that may consist of multiple containers, that may interact with other containers and that may need to scale and restarted upon failure is not easy with only Docker. Enter Kubernetes, a container management platform.

Let's run one instance of one standalone microservice on Kubernetes – basically the same thing as running a single Docker Container:

From workshop directory part1/kubernetes, run

```
kubectl create -f pod.yaml
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl create -f pod.yaml
pod "request-counter-ms" created
```

This will create a Pod according to the specification in the file pod.yaml. Open this file and review the specification for the Pod. Crucial elements are the name of the Pod, the name of the underlying Docker Container Image and the values of the environment variables to be passed into the container.

Check on the Pod in the Dashboard (note: it will take up to a few minutes to get running for the first time because of the downloading of container images):

The screenshot shows the Kubernetes Dashboard interface. The left sidebar has a tree view with 'Admin' expanded, showing 'Namespaces', 'Nodes', 'Persistent Volumes', and 'Storage Classes'. Under 'Namespaces', 'default' is selected. The 'Workloads' section is expanded, showing 'Deployments', 'Replica Sets', 'Replication Controllers', 'Daemon Sets', 'Stateful Sets', 'Jobs', and 'Pods'. 'Pods' is selected. The main area shows the 'request-counter-ms' Pod details. The 'Details' tab shows the following information:

Name: request-counter-ms	Namespace: default	Labels: app: request-counter-ms	Network
Creation time: 2017-05-24T04:36	Node: minikube	IP: 172.17.0.6	
Status: Running			

Below the Details tab is a 'Containers' section for the 'request-counter' container. It shows the image 'lucasjellema/node-app-runner' and environment variables:

Image: lucasjellema/node-app-runner
Environment variables: GIT_URL: https://github.com/lucasjellema/microservices-choreography-kubernetes-workshop-june2017 APP_PORT: 8091 APP_HOME: part1 APP_STARTUP: requestCounter-2.js

At the bottom of the container section are 'Commands:', 'Args:', and a 'View logs' link.

and verify the logging:

```

Logs from request-counter  in request-counter-ms
A
2017-05-24T04:39:05.588508817Z switching node to version stable
2017-05-24T04:39:05.972875886Z node version: v7.10.0
2017-05-24T04:39:05.974881812Z Cloning into 'app'...
2017-05-24T04:39:07.22221059Z Application Home: part1
2017-05-24T04:39:08.404003175Z request-counter@0.0.2 /code/app/part1
2017-05-24T04:39:08.404032406Z +- node-redis@0.1.7
2017-05-24T04:39:08.404037477Z `-- redis@2.7.1
2017-05-24T04:39:08.404041398Z +- double-ended-queue@2.1.0-0
2017-05-24T04:39:08.404045217Z `-- redis-commands@1.3.1
2017-05-24T04:39:08.404048955Z `-- redis-parser@2.6.0
2017-05-24T04:39:08.404053089Z
2017-05-24T04:39:08.48382346Z Node.JS Server running on port 8091 for version 2 of requestCounter application.

```

Expose the Pod using a Service to consumers outside the cluster:

```
kubectl.exe expose pod request-counter-ms --type=NodePort
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl.exe expose pod request-counter-ms --type=NodePort
service "request-counter-ms" exposed
```

Name	Labels	Cluster IP	Internal endpoints	External endpoints
kubernetes	component: apiserver provider: kubernetes	10.0.0.1	kubernetes:443 TCP kubernetes:0 TCP	-
request-counter-ms	app: request-counter-ms	10.0.0.207	request-counter-ms:8091 ... request-counter-ms:31516...	-

Using kubectl get services to inspect the service:

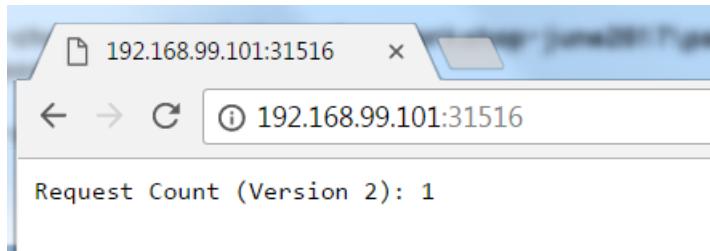
```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl get services
NAME            CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
kubernetes      10.0.0.1    <none>        443/TCP       8h
request-counter-ms  10.0.0.207 <nodes>      8091:31516/TCP 1m
```

Also to get the url for accessing the service created for pod request-counter-ms:

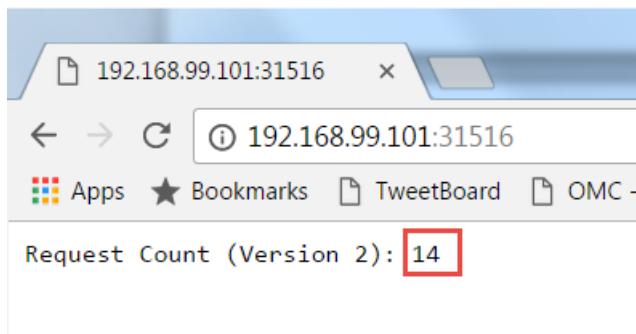
```
minikube.exe service --url=true request-counter-ms
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>minikube.exe service --url=true request-counter-ms
http://192.168.99.101:31516
```

Access the microservice at the URL retrieved (cluster ip: <port exposed for pod>):



Refresh the browser a few times. See how the counter value increases.



Delete all Pods and Services you have created on Kubernetes:

```
kubectl delete po,svc --all
```

## Handle Scaling Up and Down

We will look at deployments on Kubernetes - you can run an application by creating a Kubernetes Deployment object, and you can describe a Deployment in a YAML file. Through Deployments, multiple replicas of a Pod can be started and managed by Kubernetes. These instances can be exposed under a single external IP, with Kubernetes taking care of routing incoming traffic to one of the Pods.

Let's run the request counter microservice again, through a deployment object – as described by file deployment.yaml.

```
kubectl create -f deployment.yaml
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl create -f deployment.yaml
deployment "request-counter-ms-deployment" created
```

Display information about the Deployment:

```
kubectl describe deployment request-counter-ms-deployment
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl describe deployment request-counter-ms-deployment
Name:           request-counter-ms-deployment
Namespace:      default
CreationTimestamp:   Wed, 24 May 2017 07:19:49 +0200
Labels:          app=request-counter-ms
Annotations:    deployment.kubernetes.io/revision=1
Selector:        app=request-counter-ms
Replicas:       1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:   RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:      app=request-counter-ms
  Containers:
    request-counter-ms:
      Image:      lucasjellema/node-app-runner
      Port:       8091/TCP
      Environment:
        GIT_URL:      https://github.com/lucasjellema/microservices-choreography-kubernetes-workshop-june2017
        APP_PORT:     8091
        APP_HOME:     part1
        APP_STARTUP:  requestCounter-2.js
      Mounts:        <none>
      Volumes:       <none>
  Conditions:
    Type        Status  Reason
    ----        ----  -----
    Available   True    MinimumReplicasAvailable
    Progressing  True    NewReplicaSetAvailable
OldReplicaSets: <none>
NewReplicaSet:  request-counter-ms-deployment-2860362687 (1/1 replicas created)
Events:
  FirstSeen  LastSeen  Count  From            SubObjectPath  Type        Reason               Message
  ----        ----  -----  ----            ---          ----        ----               ---
  17s        17s       1    deployment-controller      Normal      ScalingReplicaSet  Scaled up replica set request-co
t-2860362687 to 1
```

You can also check in the Dashboard for the Deployment, the Pods created as part of it and the Replica Set:

Name	Labels	Pods	Age	Images
my-nginx	run: my-nginx	2 / 2	-	nginx
request-counter-ms-deployment	app: request-counter-ms	2 / 1	-	lucasjellema/node-app-runner

Instead of exposing a single Pod through a Service we can also expose the Deployment. Do so now, using:

```
kubectl expose deployment request-counter-ms-deployment --type=NodePort
```

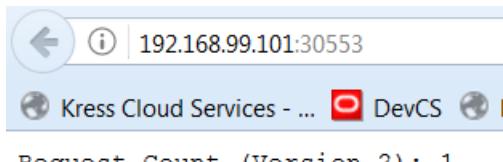
```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl.exe expose deployment request-counter-ms-deployment --type=NodePort
service "request-counter-ms-deployment" exposed
```

With kubectl get services:

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl get services
NAME           CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
kubernetes     10.0.0.1    <none>        443/TCP       8h
request-counter-ms-deployment 10.0.0.189  <nodes>      8091:30553/TCP 1m
```

Access the microservice in the browser at the specified port (30553 in this case) or simply using:

```
minikube.exe service request-counter-ms-deployment
```



Hit the url a few more times, increasing the counter as you go.

Now we will scale up the number of replicas – i.e. the number of container instances that is running on the cluster. This can be done by editing the deployment.yaml file and using kubectl apply -f deployment.yaml – or through the Dashboard:

Click on the node Deployments in the navigator. Open the drop down menu for the deployment request-counter-ms-deployment and click on *View/edit YAML*.

The screenshot shows the Kubernetes UI with the navigation bar 'kubernetes' and 'Workloads > Deployments'. On the left, there's a sidebar with 'Admin' and 'Namespaces' sections, and a dropdown for 'Namespace' set to 'default'. Under 'Workloads', 'Deployments' is selected. In the main area, a table titled 'Deployments' lists one item: 'request-counter-ms-deployment' with 'Labels' 'app: request-counter-ms', 'Pods' '1 / 1', 'Age' '-' (empty), and 'Images' 'lucasjellema/node-app-runner'. A context menu is open over the first row, with 'Delete' and 'View/edit YAML' options visible.

Change the value of the attribute replicas from 1 to 2:

The screenshot shows the 'Edit a Deployment' dialog. The deployment configuration tree is displayed, with the 'replicas' field under the 'spec' section highlighted with a red box and containing the value '2'. At the bottom right, the 'UPDATE' button is highlighted with a mouse cursor.

and click on Update.

You will see that the deployment has now 2 pods. The change is pending – as indicated by the icon on the left hand side.

The screenshot shows the Deployments table again. The 'request-counter-ms-deployment' row now shows '1 / 2' in the 'Pods' column, indicating a pending update. The other columns show 'Labels' 'app: request-counter-ms', 'Age' '6 minutes', and 'Images' 'lucasjellema/node-app-runner'.

After a little while, two pods are running:

Name	Status	Restarts	Age
request-counter-ms-deployment-2860362687-s32wn	Running	0	9 minutes
request-counter-ms-deployment-2860362687-sk1bq	Running	0	2 minutes

You can access the microservice again from the browser, or you can do so using curl from the command line:

```
curl http://<ip of cluster>:<port exposed by service>
```

Your results will be similar to the following:

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 2): 1
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 2): 10
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 2): 2
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 2): 11
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 2): 3
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 2): 4
```

Instead of a smoothly increasing counter value, we see values that are not part of the same sequence and we even get duplicate values. If you look at the code of /part1/requestCounter-2.js – the application running in the pods – it is not hard to spot the flaw: these applications are not stateless and therefore do not support horizontal scaling.

Note how Kubernetes takes care of load balancing for us. We access a endpoint exposed from the cluster, and traffic is distributed over the available Pods.

Delete one of the pods, for example in the Dashboard:

Name	Status	Restarts	Age
request-counter-ms-deployment-2860362687-qj5fs	Running	0	a minute
request-counter-ms-deployment-2860362687-skzzw	Running	0	54 seconds

You will see a new Pod being created immediately. In the deployment definition we stated two replicas are required – and that is what Kubernetes ensures.

Now once more access the microservice a few times, for example using curl:

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/  
Request Count (Version 2): 1  
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/  
Request Count (Version 2): 12  
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/  
Request Count (Version 2): 2  
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/  
Request Count (Version 2): 3
```

The value 1 is obviously returned by the newly instantiated Pod.

## Adding a Cache to the Microservices Platform and running a Stateless version of RequestCounter

Add a second pod with Redis in it

```
kubectl run redis-cache --image=redis --port=6379
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl run redis-cache --image=redis --port=6379  
deployment "redis-cache" created
```

Expose redis within cluster:

```
kubectl expose deployment redis-cache --type=ClusterIP
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl expose deployment redis-cache --type=ClusterIP  
service "redis-cache" exposed
```

From the command line we can inspect the services exposed on our minikube cluster:

```
kubectl get services
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl get services  
NAME           CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE  
kubernetes     10.0.0.1    <none>        443/TCP       10h  
redis-cache    10.0.0.55   <none>        6379/TCP      1m  
request-counter-ms-deployment 10.0.0.189 <nodes>        8091:30553/TCP 1h
```

This time, since redis-cache is exposed only inside the cluster, this command:

```
minikube.exe service redis-cache
```

produces no output.

Services and discovery > Services					
Services					
Name	Labels	Cluster IP	Internal endpoints	External endpoints	
kubernetes	component: apiserver provider: kubernetes	10.0.0.1	kubernetes:443 TCP kubernetes:0 TCP	-	
redis-cache	run: redis-cache	10.0.0.55	redis-cache:6379 TCP redis-cache:0 TCP	-	
request-counter-ms-deployment	app: request-counter-ms	10.0.0.189	request-counter-ms-deployment... request-counter-ms-deployment...	-	

The hostname for the Redis cache service inside the cluster is redis-cache and the port is 6379 – at this endpoint, other pods in the cluster can access the service.

Upgrade deployment RequestCounter to version with Redis backing. V3: no lock, v4: optimistic lock

```
kubectl apply -f deployment-v2.yaml --record
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl apply -f deployment-v2.yaml --record
Warning: kubectl apply should be used on resource created by either kubectl create --save-config or kubectl apply deployment "request-counter-ms-deployment" configured
```

Check on the status of the rollout of the change:

```
kubectl rollout status deployment request-counter-ms-deployment
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl rollout status deployment request-counter-ms-deployment
deployment "request-counter-ms-deployment" successfully rolled out
```

We can inspect the history of a specific deployment and see which changes have been applied to it:

```
kubectl rollout history deployment request-counter-ms-deployment
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl rollout history deployment request-counter-ms-deployment
deployments "request-counter-ms-deployment"
REVISION      CHANGE-CAUSE
1            <none>
2            kubectl apply --filename=deployment-v2.yaml --record=true
```

In the dashboard, if we are quick we can see the pods being restarted. If we are not so fast, we can see both pods running (again) – having been restarted fairly recently:

Workloads > Pods

+ CREATE

### Pods

Name	Status	Restarts	Age	Actions
<span>✓</span> redis-cache-3679063315-kr1lx	Running	0	21 hours	<span>⋮</span>
<span>✓</span> request-counter-ms-deployment-68346987-qxv6v	Running	0	2 minutes	<span>⋮</span>
<span>✓</span> request-counter-ms-deployment-68346987-zg8rj	Running	0	-	<span>⋮</span>

Both Pods are running again, from the logs you can tell that both are running the new version of the application – powered by Redis.

☰ kubernetes Logs + CREATE

Admin

- Namespaces
- Nodes
- Persistent Volumes
- Storage Classes

Namespace

- default

Workloads

- Deployments
- Replica Sets
- Replication Controllers
- Daemon Sets
- Stateful Sets
- Jobs
- Pods

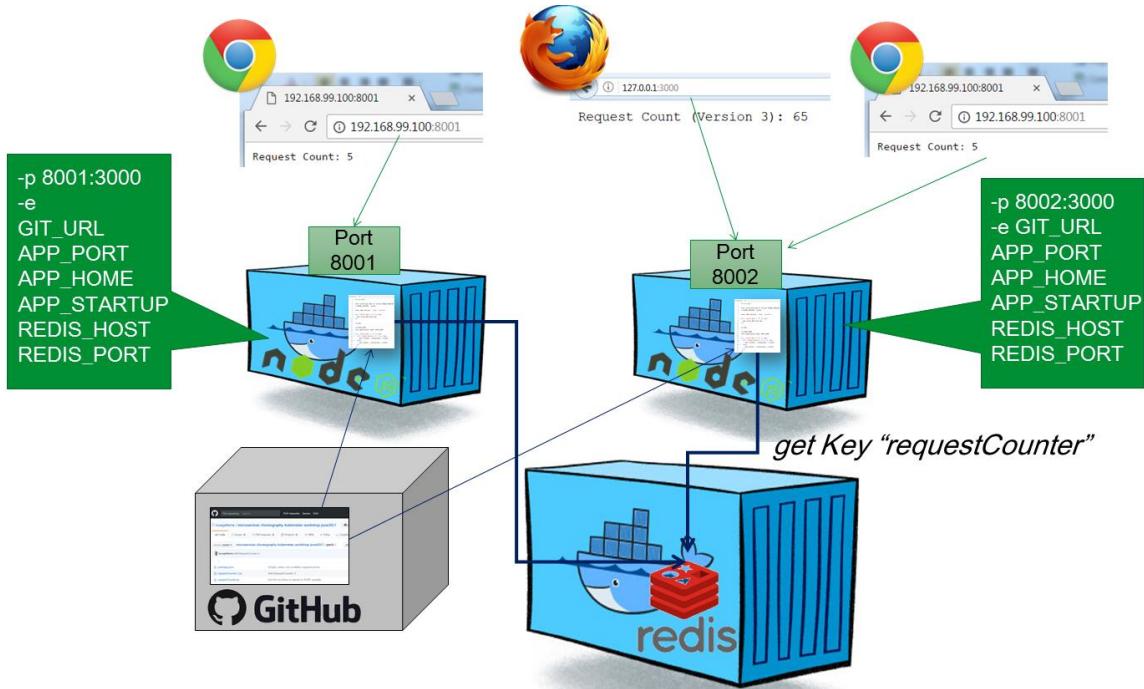
Logs from request-counter-ms in request-counter-ms-deployment-68346987-qxv6v

```

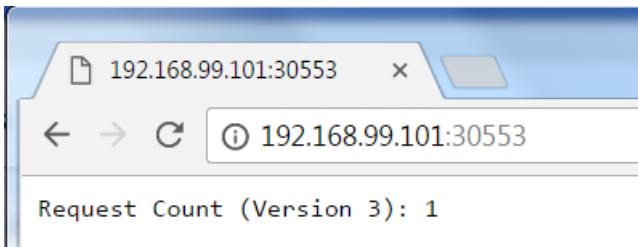
2017-05-25T04:25:19.163639616Z switching node to version stable
2017-05-25T04:25:19.3998537346Z node version: v7.10.0
2017-05-25T04:25:19.392821771Z Cloning into 'app'...
2017-05-25T04:25:20.286508953Z Application Home: part1
2017-05-25T04:25:21.625460277Z request-counter@0.0.2 /code/app/part1
2017-05-25T04:25:21.625489197Z --- node-redis@0.1.7
2017-05-25T04:25:21.625493379Z '--- redis@2.7.1
2017-05-25T04:25:21.625496958Z '--- double-ended-queue@2.1.0-0
2017-05-25T04:25:21.625508518Z '--- redis-commands@1.3.1
2017-05-25T04:25:21.625508399Z '--- redis-parser@2.6.0
2017-05-25T04:25:21.757802701Z Node.js Server running on port 8091 for version 3 of requestCounter application, powered by Redis.

```

The situation we have now established is visualized below:



We can access the microservice (again) in the browser:



And from the command line we can do the curl thing a few times:

```
curl http://<ip of cluster>:<port exposed by service>
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 3) 3
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 3) 4
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 3) 5
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 3) 6
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 3) 7
```

This time round, we do not get two counters incrementing in the background. Just a single incrementing value. And supposedly, that value is stored in the Redis cache where both instances access and manipulate it.

That should mean that we can stop and (re)start one or even both pods and then continue with the count as though nothing happened. Let's try that out. Stop one of the pods (note: get the name of the Pod in your case from either the command line using `kubectl get pods` or through the dashboard)

```
kubectl delete pod request-counter-ms-deployment-68346987-qxv6v --  
grace-period=60
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl delete pod request-counter-ms-deployment-68346987-qxv6v --grace-period=60  
pod "request-counter-ms-deployment-68346987-qxv6v" deleted
```

In the dashboard we can see how a new pod is started – because in the deployment we specified 2 replicas to always be running :

Name	Status	Restarts	Age	Actions
redis-cache-3679063315-kr1lx	Running	0	21 hours	⋮
request-counter-ms-deployment-68346987-dp2t8	Pending	0	0 seconds	⋮
request-counter-ms-deployment-68346987-qxv6v	Running	0	17 minutes	⋮
request-counter-ms-deployment-68346987-zg8rj	Running	0	14 minutes	⋮

and a little bit later the Pod we asked to be deleted is gone:

Name	Status	Restarts	Age	Actions
redis-cache-3679063315-kr1lx	Running	0	21 hours	⋮
request-counter-ms-deployment-68346987-dp2t8	Running	0	a minute	⋮
request-counter-ms-deployment-68346987-zg8rj	Running	0	16 minutes	⋮

We can also delete our other pod, to be sure any state lingering in the original pods is gone:

```
kubectl delete pod request-counter-ms-deployment-68346987-zg8rj
```

```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl delete pod request-counter-ms-deployment-68346987-zg8rj  
pod "request-counter-ms-deployment-68346987-zg8rj" deleted  
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl get pods  
NAME          READY   STATUS    RESTARTS   AGE  
redis-cache-3679063315-kr1lx  1/1     Running   0          21h  
request-counter-ms-deployment-68346987-5w16v  1/1     Running   0          19s  
request-counter-ms-deployment-68346987-dp2t8  1/1     Running   0          5m  
request-counter-ms-deployment-68346987-zg8rj  1/1     Terminating   0          19m
```

Again, we see a new pod running and the old instance being terminated. At this point, both original pods are gone. If we access the request counter microservice, we will find that the counting continues where it had left off:

```

request-counter-ms-deployment-68346987-dpzt8 1/1      Running   0      5m
request-counter-ms-deployment-68346987-zg8rj  1/1      Terminating   0      19m

C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 3): 8
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 3): 9
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 3): 10
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 3): 11
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 3): 12
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 3): 13

```

Of course now we have moved the burden of state to the Redis pod. If we restart that pod, we will get a reset in the counter value, because currently the cache is not persisted outside the container and its contents vanishes when the container dies. Note that we can have Redis be persisted to files outside container.

```
kubectl delete pod <name of redis-cache-... pod>
```

A new pod will be started after a few seconds and the old one terminated. When we then access the microservice requestcounter, the inevitable has happened: a counter reset. The old value was lost when the Redis cache pod was terminated:

```

C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl delete pod redis-cache-3679063315-kr1lx
pod "redis-cache-3679063315-kr1lx" deleted

C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
redis-cache-3679063315-pxz02  1/1     Running   0          1m
request-counter-ms-deployment-68346987-5w16u  1/1     Running   0          7m
request-counter-ms-deployment-68346987-dp2t8  1/1     Running   0          12m

C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 3): 1
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 3): 2
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 3): 3
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>curl http://192.168.99.101:30553/
Request Count (Version 3): 4

```

## Resources

See: <https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services---service-types> on services in Kubernetes

Port forwarding from host to Redis Pod: <https://kubernetes.io/docs/tasks/access-application-cluster/port-forward-access-application-cluster/>

*Rolling updates* with K8s deployment: <https://tachingchen.com/blog/Kubernetes-Rolling-Update-with-Deployment/>

Graceful shutdown of pods: <https://pracucci.com/graceful-shutdown-of-kubernetes-pods.html>

## Useful Kubectl commands

To see the logs from a specific pod:

```
kubectl logs my-pod
```

To open a shell in the running container in a pod with only one container (see:

<https://kubernetes.io/docs/tasks/debug-application-cluster/get-shell-running-container/>) :

```
kubectl exec -it my-pod -- bash
```

```
c:\data\microservices-choreography-kubernetes-workshop-june2017\part4\WorkflowLauncher>kubectl exec -it workflow-launcher-ms -- /bin/bash
root@workflow-launcher-ms:/code# ls
app bootstrap.sh
root@workflow-launcher-ms:/code# cd app
root@workflow-launcher-ms:/code/app# ls
AMIS-Workshop-IntroductionMicroserviceChoreography_KubernetesKafkaNodeJS-June2017.docx part1E.m part3 part4
```

Type exit in the shell to return.

Run a command in the container in a Pod:

```
kubectl exec my-pod command
```

```
c:\data\microservices-choreography-kubernetes-workshop-june2017\part4\WorkflowLauncher>kubectl exec workflow-launcher-ms ls
app
bootstrap.sh
```

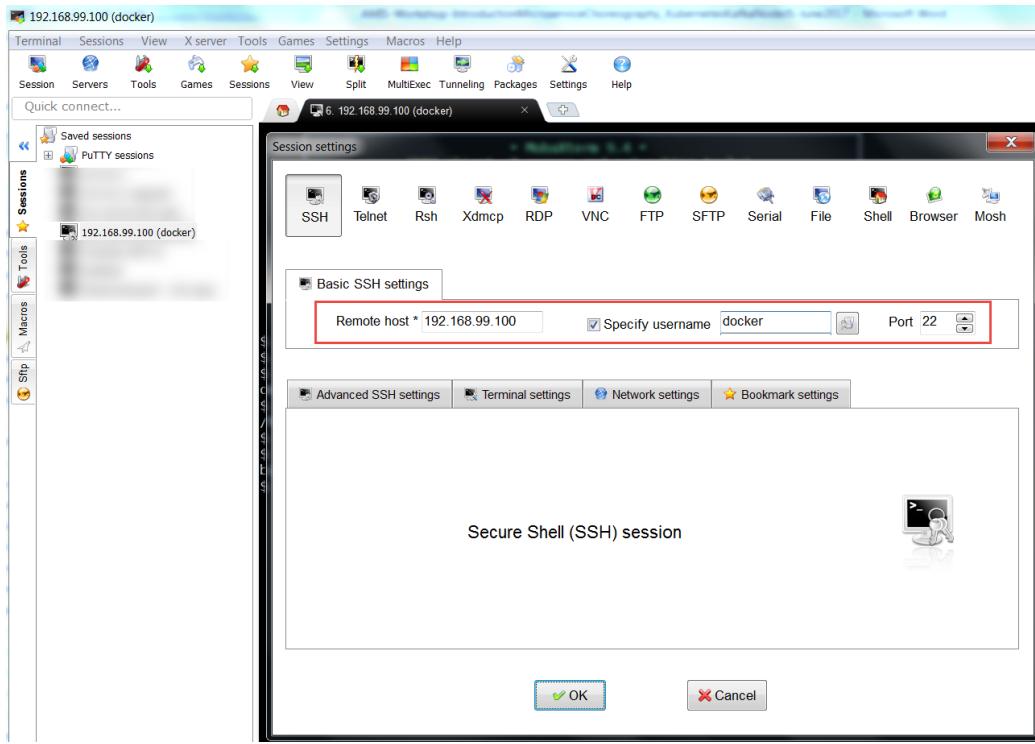
To open a shell in a running container in a pod with multiple containers:

```
kubectl exec -it my-pod -c my-container -- bash
```

See the API reference for Kubectl exec for more details: <https://kubernetes.io/docs/user-guide/kubectl/v1.6/#exec>

An SSH connection can be created into the VM running the minikube cluster:

Username is Docker, password is tcuser.



Check for example all running containers with: docker ps

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
TS	NAMES				
acb0ffd5e64b	9c5af59228c	"/usr/bin/run_proxy"	2 minutes ago	Up 2 minutes	
	k8s_kube-registry-proxy_kube-registry-proxy_kube-system_132964b6-460c-11e7-9206-080027d2b34f_0				
3c11935a3429	9d0c4ebab4d	"/entrypoint.sh /etc/"	3 minutes ago	Up 3 minutes	
	k8s_registry_kube-registry-v0-35hnv_kube-system_12ff239c-460c-11e7-9206-080027d2b34f_0				
f34c0f67edf2	6bb8ffa98f4	"/code/bootstrap.sh"	4 minutes ago	Up 4 minutes	
	k8s_cache-inspector_cache-inspector-ms_default_b5da8716-45f9-11e7-ad0e-080027d2b34f_2				
e8a7c1a8c240	gcr.io/google_containers/pause-amd64:3.0	"/pause"	4 minutes ago	Up 4 minutes	0.0
,0.0:5000->5000/tcp	k8s_POD_kube-registry-proxy_kube-system_132964b6-460c-11e7-9206-080027d2b34f_0				
2dd406d2990b	gcr.io/google_containers/pause-amd64:3.0	"/pause"	4 minutes ago	Up 4 minutes	
	k8s_POD_kube-registry-v0-35hnv_kube-system_12ff239c-460c-11e7-9206-080027d2b34f_0				
779b0442f45a	6bb8ffa98f4	"/code/bootstrap.sh"	4 minutes ago	Up 4 minutes	
	k8s_tweet-enricher_tweet-enricher-ms_default_e1856767-45ec-11e7-ad0e-080027d2b34f_2				
e2ca2f797916	gcr.io/google_containers/pause-amd64:3.0	"/pause"	4 minutes ago	Up 4 minutes	
	k8s_POD_cache-inspector-ms_default_b5da8716-45f9-11e7-ad0e-080027d2b34f_2				
2376e35b723d	6bb8ffa98f4	"/code/bootstrap.sh"	4 minutes ago	Up 4 minutes	
	k8s_tweet-board_tweet-board-ms_default_09b10924-45f3-11e7-ad0e-080027d2b34f_2				
f0ea410110cd	a858478874d1	"docker-entrypoint.sh"	4 minutes ago	Up 4 minutes	
	k8s_redis-cache_redis-cache-3679063315-8m9jl_default_c11b1a16-45cf-11e7-ad0e-080027d2b34f_2				
40530ee3c7ff	6bb8ffa98f4	"/code/bootstrap.sh"	4 minutes ago	Up 4 minutes	
	k8s_tweetreceiver_tweetreceiver-ms_default_d4aaef2c4-45cd-11e7-ad0e-080027d2b34f_2				
26129839f1dc	6bb8ffa98f4	"/code/bootstrap.sh"	4 minutes ago	Up 4 minutes	
	k8s_microservice-workflow-launcher_workflow-launcher-ms_default_ea081b01-45ff-11e7-ad0e-080027d2b34f_2				
849e1e8f12f9	6bb8ffa98f4	"/code/bootstrap.sh"	4 minutes ago	Up 4 minutes	
	k8s_tweet-validator_tweet-validator-ms_default_ca887a2a-45dd-11e7-ad0e-080027d2b34f_2				
a517b83fe69a	gcr.io/google_containers/pause-amd64:3.0	"/pause"	4 minutes ago	Up 4 minutes	
	k8s_POD_tweet-enricher-ms_default_e1856767-45ec-11e7-ad0e-080027d2b34f_2				
da09c4243710	gcr.io/google_containers/pause-amd64:3.0	"/pause"	4 minutes ago	Up 4 minutes	
	k8s_POD_tweet-board-ms_default_09b10924-45f3-11e7-ad0e-080027d2b34f_2				
0aa8c4fe30c4	93a43fbfb39bf	"/exechealthz '--cmd='	4 minutes ago	Up 4 minutes	
	k8s_healthz_kube-dns-v20-t2r34_kube-system_c94122ee-41de-11e7-b254-080027d2b34f_4				
81fcacae0a231	416701f962f2	"/dashboard --port=90"	4 minutes ago	Up 4 minutes	
	k8s_kubernetes-dashboard_kubernetes-dashboard-vfhnq_kube-system_c9305543-41de-11e7-b254-080027d2b34f_4				
5eb697c24d5c	3ec65756a89b	"/usr/sbin/dnsmasq --"	4 minutes ago	Up 4 minutes	
	k8s_dnsmasq_kube-dns-v20-t2r34_kube-system_c94122ee-41de-11e7-b254-080027d2b34f_4				
8fe800b4fd70	gcr.io/google_containers/pause-amd64:3.0	"/pause"	4 minutes ago	Up 4 minutes	
	k8s_POD_redis-cache-3679063315-8m9jl_default_c11b1a16-45cf-11e7-ad0e-080027d2b34f_2				
99ee585d4f8c	gcr.io/google_containers/pause-amd64:3.0	"/pause"	4 minutes ago	Up 4 minutes	
	k8s_POD_tweetreceiver-ms_default_d4aaef2c4-45cd-11e7-ad0e-080027d2b34f_2				
4b967e331444	gcr.io/google_containers/pause-amd64:3.0	"/pause"	4 minutes ago	Up 4 minutes	
	k8s_POD_workflow-launcher-ms_default_ea081b01-45ff-11e7-ad0e-080027d2b34f_2				

Save images from local docker server to archive:

```
docker save -o /home/images-archive.tar image1 image2 lucasjellema/node-app-runner
```

```
docker save -o /home-sources/images-archive.tar redis lucasjellema/node-app-runner
```

### 3. Introducing the Event Bus Microservice into the Microservices Platform

Microservice orchestration through an event bus is one of the ultimate goals of this workshop. Let's add an event bus to our microservice platform – using Apache Kafka.

Note: there are many articles and resources for running Kafka & Zookeeper in Docker and in Kubernetes. Some resources are listed below. Unfortunately, none of these options really worked well for me. In the time I had available for preparation, I could not get a reproducible case to work as desired.

Therefore, you will work with a lean Virtual Machine that runs in VirtualBox.

A very interesting alternative is to work with Kafka in the Cloud. Various providers offer a Kafka PaaS service – for example IBM BlueMix and Oracle Event Hub. An interesting option is CloudKarafka which offers a free tier and which is very easy to get going with. See <https://www.cloudkarafka.com/plans.html> for details - and the free Developer Duck Plan.

#### Getting started with the Kafka VM

Running the VM requires 2Gb of RAM and approximately 7Gb of free HD space

Start VirtualBox

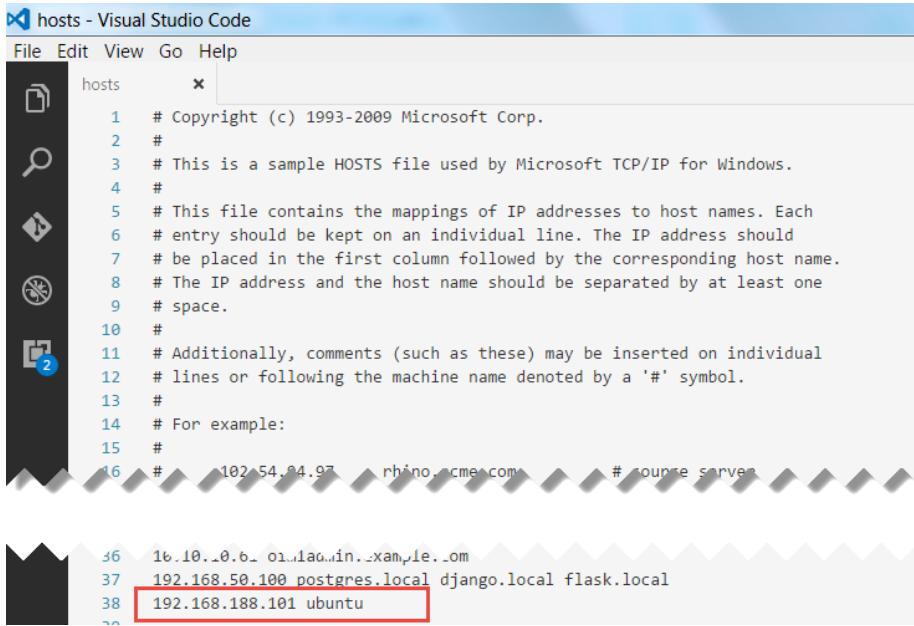
Import the VM image – KafkaWorkshop.ova - by going to File, Import Virtual Appliance and browse to the image file. The VM requires 2Gb of RAM to run.

#### Configure Networking in VM

To access the Kafka instance running inside the VM from your laptop, you need to ensure that you enable network access to the VM. The instructions in this article (AMIS Technology Blog:

<https://technology.amis.nl/2017/01/29/network-access-to-ubuntu-virtual-box-vm-from-host-laptop/>) show what you have to configure in order to be able to access the VM on its own IP address.

After arranging access to the VM, you may want to define a logical host name for the Virtual Machine – by adding an entry to the hosts file: C:\Windows\System32\drivers\etc\hosts, as shows below:



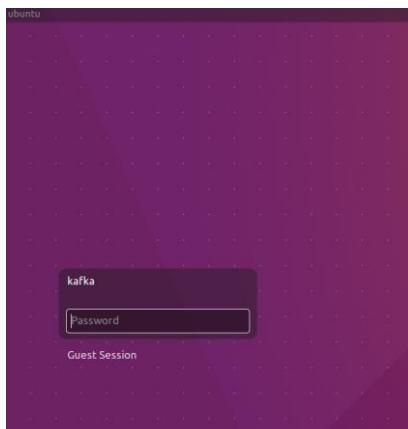
The screenshot shows the Visual Studio Code interface with a dark theme. The title bar says "hosts - Visual Studio Code". The menu bar includes File, Edit, View, Go, Help. The left sidebar has icons for file operations like Open, Save, Find, and a folder with two files. The main editor area contains the Windows hosts file. Lines 36, 37, and 38 are highlighted with a red box:

```
1 # Copyright (c) 1993-2009 Microsoft Corp.
2 #
3 # This is a sample HOSTS file used by Microsoft TCP/IP for Windows.
4 #
5 # This file contains the mappings of IP addresses to host names. Each
6 # entry should be kept on an individual line. The IP address should
7 # be placed in the first column followed by the corresponding host name.
8 # The IP address and the host name should be separated by at least one
9 # space.
10 #
11 # Additionally, comments (such as these) may be inserted on individual
12 # lines or following the machine name denoted by a '#' symbol.
13 #
14 # For example:
15 #
16 #       102.54.0.97    rhino.cme.com    # source server
17 #
18 #
19 #
20 #
21 #
22 #
23 #
24 #
25 #
26 #
27 #
28 #
29 #
30 #
31 #
32 #
33 #
34 #
35 #
36 16.10.0.6 01.liaowin.example.com
37 192.168.50.100 postgres.local django.local flask.local
38 192.168.188.101 ubuntu
```

Note: *ubuntu* is the (advertised) host name of the Kafka in the VM; to play nice with Zookeeper it seems convenient to use that same name as the host name in our hosts file.

## Start the VM

The VM will boot to a graphical login screen



The VM has two users: *root* and *kafka*. Both have password Welcome01.

The VM contains:

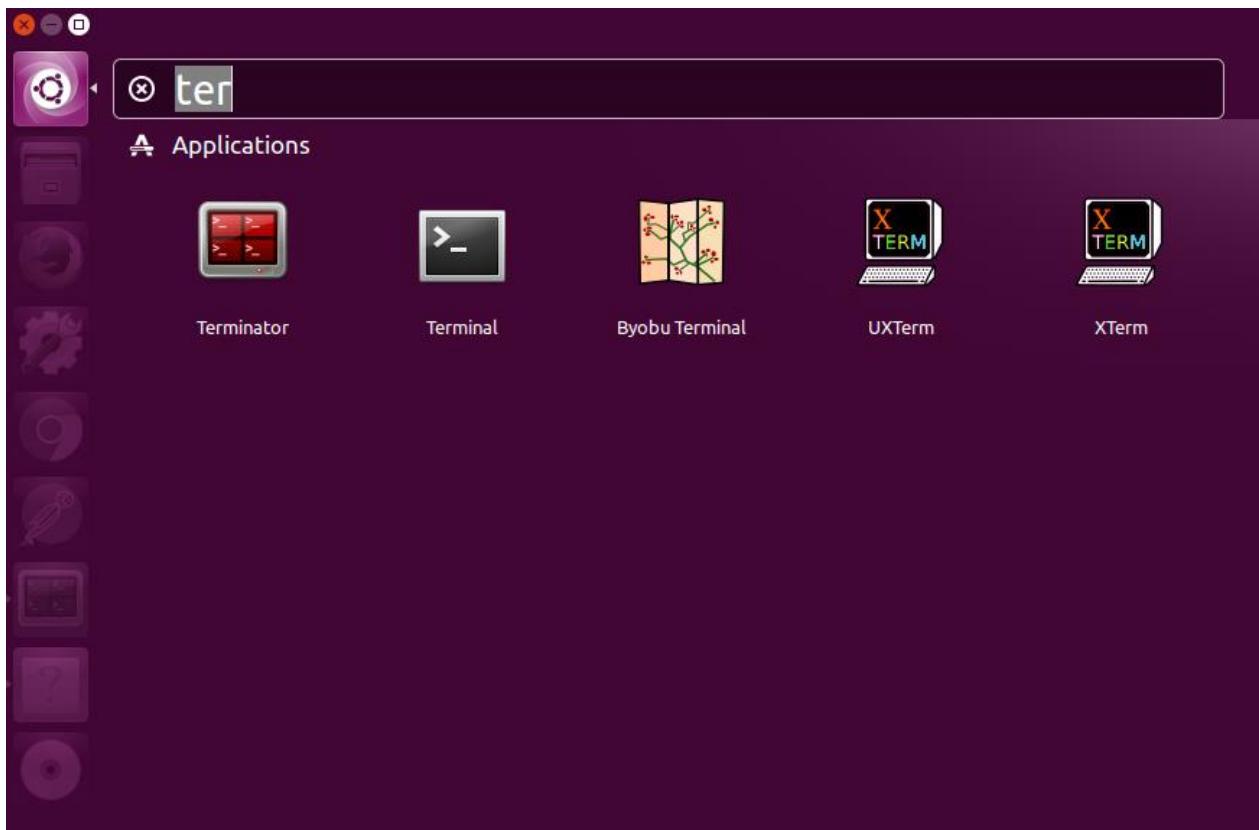
- Ubuntu 16.04 LTS
- Oracle JDK 8
- Eclipse Neon.2
- Node 7.4
- Zookeeper
- Firefox

- Chrome
- Postman
- confluent-platform-oss-2.11
- kafka-manager 1.3.2.1. When started runs on <http://localhost:9000>
- kafkatoold 1.0.1

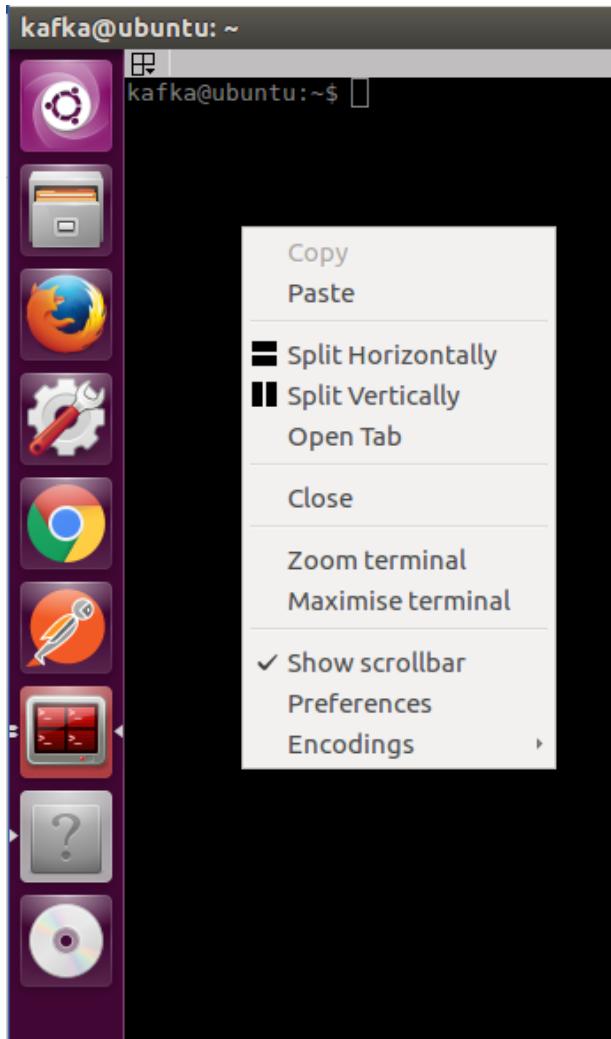
## Getting to know the Kafka VM

Login to the VM: kafka/Welcome01

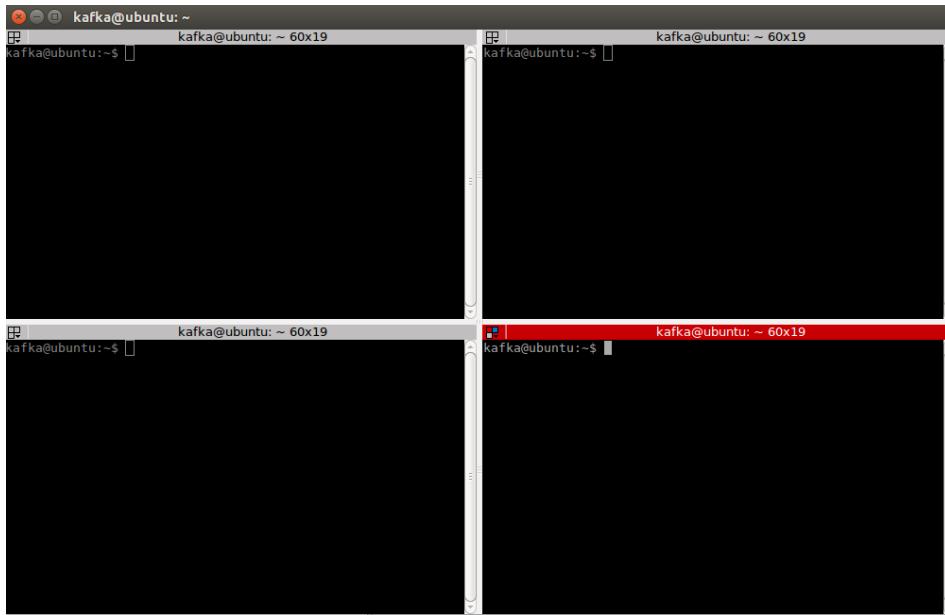
Start Terminator from the Unity menu



Split the screen horizontally – as shown in the next figure - and both screens vertically by right clicking in the window.



You will end up with four terminal windows.



In the first terminal window start a Kafka broker:

```
kafka-server-start /etc/kafka/server.properties
```

Start in the second terminal window start the kafka manager:

```
kafka-manager
```

Start kafkatool in the third terminal window:

```
kafkatool
```

Minimize the tool. You will use it at a later time.

Continue in the fourth terminal window

### Create a topic

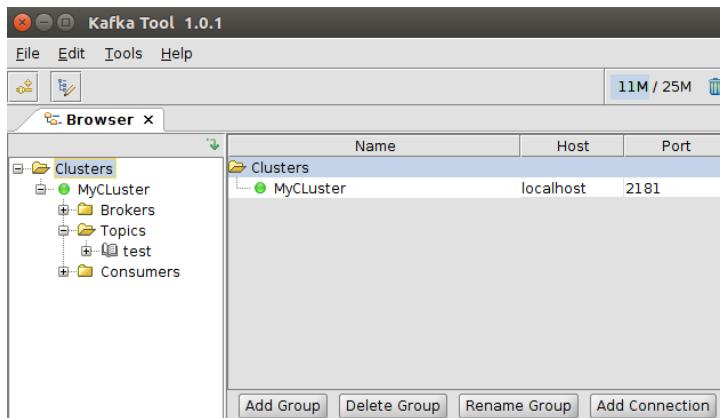
Create a topic:

```
kafka-topics --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test
```

Confirm the topic is created:

```
kafka-topics --list --zookeeper localhost:2181
```

Open kafkatool and confirm the topic is created



## Produce a message

Focus on the fourth terminal window again and type:

```
kafka-console-producer --broker-list localhost:9092 --topic test
```

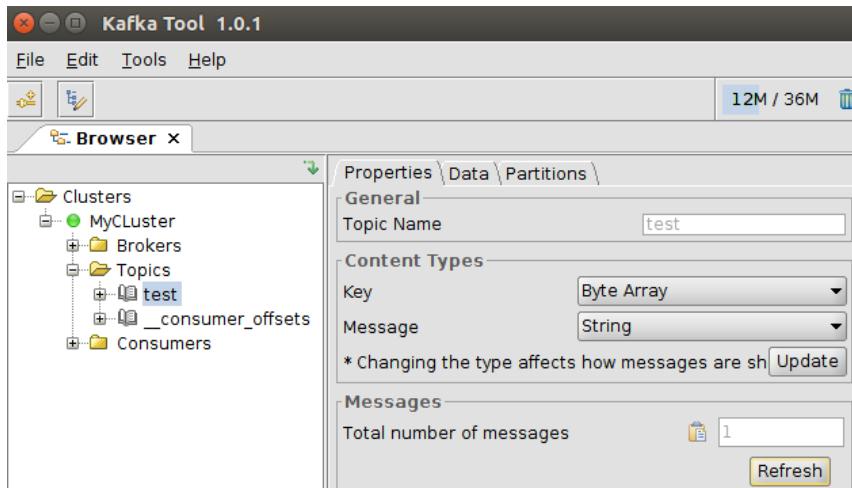
Press enter. No feedback appears; the cursor sits blinking and waiting for your input.

Type a message and press enter:

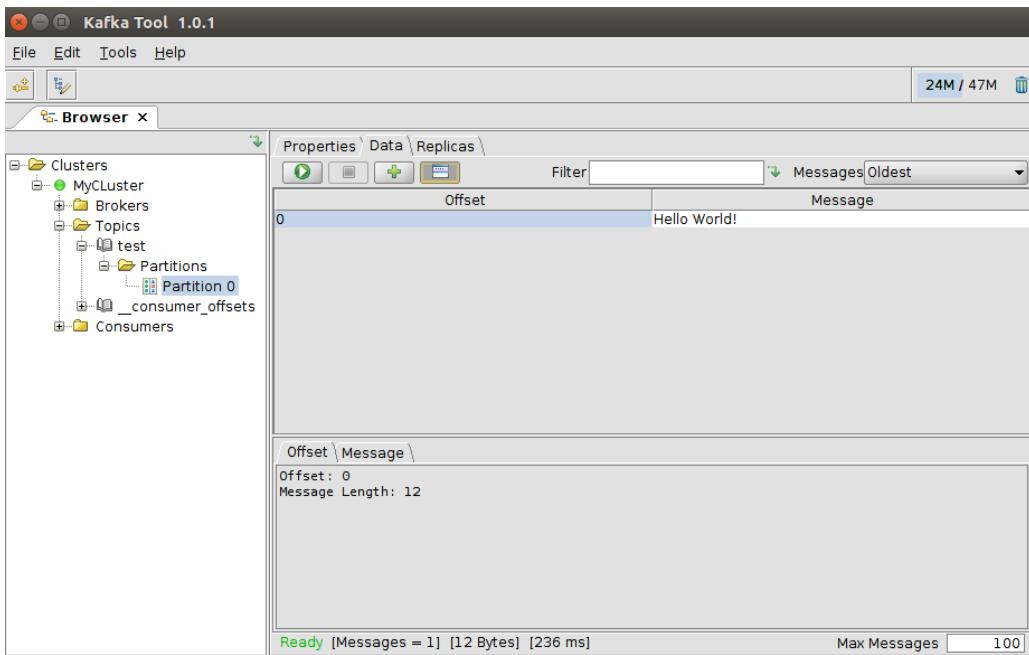
```
Hello World!
```

Check the message is created using kafka-manager and kafkatoold.

Open kafkatoold. Indicate the messages on the topic are string. Click the update button. Click the refresh button to view how many messages are present on the topic. Confirm the number is 1.



Open the partition under the test topic. Click the green play button. Confirm the Hello World! message has arrived.



## Consume a message

Consume a message

```
kafka-console-consumer --bootstrap-server localhost:9092 --topic test --from-beginning
```

```
kafka@ubuntu:~$ kafka-console-consumer --bootstrap-server localhost:9092 --topic test --from-beginning
Hello World!
```

Confirm that the previously posted message has been consumed.

## Run a Microservice with Kafka Interaction

In one of the terminal windows in your Kafka VM – determine the IP address that is assigned to the VM:

ifconfig

```
kafka@ubuntu:~$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:86:d4:19
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe86:d419/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:199 errors:0 dropped:0 overruns:0 frame:0
          TX packets:246 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:61565 (61.5 KB)  TX bytes:30803 (30.8 KB)

enp0s8    Link encap:Ethernet  HWaddr 08:00:27:25:ec:80
          inet addr:192.168.188.101  Bcast:192.168.188.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe25:ec80/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:5342045 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8251249 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:539053379 (539.0 MB)  TX bytes:682081625 (682.0 MB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
```

This is the address for the Kafka Host that we can use in applications that interact with Kafka. Take note of this IP address.

Create a new Kafka Topic called event-bus:

```
kafka-topics --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic event-bus
```

```
kafka@ubuntu:~/bin$ kafka-topics --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic event-bus
Created topic "event-bus".
```

Confirm the topic is created:

```
kafka-topics --list --zookeeper localhost:2181
```

Run a Pod on Kubernetes that starts a Node.JS application that listens to the Event Bus topic on the Kafka Broker. You do this using the EventBusListener.yaml file in directory part3/event-bus-listener. Before you run the yaml file, check out its contents. Take note of the ports, the environment variables such as KAFKA\_TOPIC (set to event-bus) and KAFKA\_HOST. The *lifecycle* element in the yaml file ensures that the Kafka Host (the VM running the Kafka Broker) is known as a host inside the EventBusListener container. Ensure that the IP address used in the command executed is the correct one – that is: the IP address returned inside the Kafka Host using ifconfig.

```
EventBusListenerPod.yaml x package.json EventBusPublisherService.yaml EventBusListener.js (Index) E
3  metadata:
4    name: event-bus-listener-ms
5    labels:
6      app: event-bus-listener-ms
7  spec:
8    nodeName: minikube
9    containers:
10      - name: event-bus-listener
11        # get latest version of image
12        image: lucasjellema/node-app-runner
13        env:
14          - name: GIT_URL
15            value: "https://github.com/lucasjellema/microservices-choreography-kubernetes-w
16          - name: APP_PORT
17            value: "8096"
18          - name: APP_HOME
19            value: "part3/event-bus-listener"
20          - name: APP_STARTUP
21            value: "EventBusListener.js"
22          - name: KAFKA_HOST
23            value: "ubuntu"
24          - name: ZOOKEEPER_PORT
25            value: "2181"
26          - name: KAFKA_TOPIC
27            value: "event-bus"
28    ports:
29      # containerPort is the port exposed by the container (where nodejs express api is
30      - containerPort: 8096
31    lifecycle:
32      postStart:
33        exec:
34          # add advertised host (ubuntu) of VM running Kafka to hosts file - with tha
            command: ["sh","-c","echo 192.168.188.101 ubuntu > /etc/hosts"]
```

When the yaml file is correct, then run:

```
kubectl create -f EventBusListenerPod.yaml -f  
EventBusListenerService.yaml
```

You can check in the Kubernetes Dashboard whether the Pod has started running successfully and is now listening to a Kafka Topic:

The screenshot shows the Kubernetes Dashboard interface. The left sidebar has sections for Admin, Namespaces, Nodes, Persistent Volumes, Storage Classes, Namespace (set to default), and Workloads (Deployment). The main area is titled "Logs from event-bus-listener" and shows log entries for the pod "event-bus-listener-ms". The logs include initialization messages for Kafka Client and Consumer, adding a topic named "event-bus", and a message handler being added.

```
Logs from event-bus-listener in event-bus-listener-ms
2017-05-30T04:53:41.880036408Z +++ json-stringify-safe@5.0.1
2017-05-30T04:53:41.880039678Z +++ mime-types@2.1.15
2017-05-30T04:53:41.880054482Z | `-- mime-db@1.27.0
2017-05-30T04:53:41.880058996Z +++ oauth-sign@0.8.2
2017-05-30T04:53:41.880062379Z +++ performance-now@0.2.0
2017-05-30T04:53:41.880089881Z +++ safe-buffer@5.0.1
2017-05-30T04:53:41.880096483Z +++ stringstream@0.0.5
2017-05-30T04:53:41.880099959Z +++ tough-cookie@2.3.2
2017-05-30T04:53:41.880183455Z | `-- punycode@1.4.1
2017-05-30T04:53:41.880196865Z '-- tunnel-agent@0.6.0
2017-05-30T04:53:41.88019473Z
2017-05-30T04:53:42.278610223Z Running EventBusListenerversion 0.8
2017-05-30T04:53:42.303216387Z Try to initialize Kafka Client and Consumer, attempt 1
2017-05-30T04:53:42.307417967Z created client for ubuntu
2017-05-30T04:53:42.309336879Z Kafka Client and Consumer initialized [object Object]
2017-05-30T04:53:42.310281176Z Kafka Consumer - added message handler and added topic
2017-05-30T04:53:42.311128784Z MicroserviceEventBusListener running, Express is listening... at 8096 for /ping, /calls
2017-05-30T04:53:42.433761Z topic added: event-bus
```

Then, publish an event to topic event-bus from the command line in the VM running the Kafka Broker:

```
kafka-console-producer --broker-list localhost:9092 --topic event-bus
```

Press enter. No feedback appears; the cursor sits blinking and waiting for your input. Type a valid JSON message, something like:

```
{"Wish" : "Hello To World"}
```

```
kafka@ubuntu:~/bin$ kafka-console-producer --broker-list localhost:9092 --topic event-bus
{"Wish" : "Hello To World"}
```

If you check the logs in the Kubernetes Dashboard, you should find that the event has been received:

Logs

```
Logs from event-bus-listener in event-bus-listener-ms
2017-05-30T04:53:42.278610223Z Running EventBusListenerversion 0.8
2017-05-30T04:53:42.303216387Z Try to initialize Kafka Client and Consumer, attempt 1
2017-05-30T04:53:42.307417967Z created client for ubuntu
2017-05-30T04:53:42.309386879Z Kafka Client and Consumer initialized [object Object]
2017-05-30T04:53:42.310281176Z Kafka Consumer - added message handler and added topic
2017-05-30T04:53:42.311128784Z MicroserviceEventBusListener running, Express is listening... at 8096 for /ping, /abo
calls
2017-05-30T04:53:42.433761Z topic added: event-bus
2017-05-30T05:16:58.806533218Z actual event: { "a": "aaa" }
2017-05-30T05:16:58.815491355Z event received
2017-05-30T05:16:58.815680227Z received message { topic: 'event-bus',
2017-05-30T05:16:58.815701308Z value: '{"Wish" : "Hello To World"}',
2017-05-30T05:16:58.815712297Z offset: 6,
2017-05-30T05:16:58.815722764Z partition: 0,
2017-05-30T05:16:58.815767961Z highWaterOffset: 7,
2017-05-30T05:16:58.816293398Z key: -1 }
2017-05-30T05:16:58.816293398Z received message object {"topic":"event-bus","value":"{\"Wish\" : \"Hello To
World\"}","offset":6,"partition":0,"highWaterOffset":7,"key":-1}
```

Using the port assigned to service

```
c:\data\microservices-choreography-kubernetes-workshop-june2017\part3\event-bus-listener>kubectl get services
NAME           CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE
eventbuslistenerservice  10.0.0.249   <nodes>        8096,32004/TCP   5m
```

You can access the event bus listener microservice from the browser on your laptop (using the IP address of the Docker machine) and get a list of all events consumed thusfar from topic event-bus:

The screenshot shows a browser window with the URL `192.168.99.100:32004/event-bus`. The response is a JSON object:

```

topic: "event-bus"
events:
  0:
    Wish: "Hello To World"
  
```

## Microservice on Kubernetes that Publishes Events

As a next step, we will run a microservice that takes input from you – the user – through simple HTTP requests and that turns each request into an event published on the event bus. The *event bus listener* microservice that we launched in the previous section will consume all those events and make them available through the browser. That means we realize communication between two microservices that are completely unaware of each other and only need to know about the common microservices platform event-bus capability.

Directory `part3\event-bus-publisher` contains the sources for this microservice.

```
kubectl create -f EventBusPublisherPod.yaml -f  
EventBusPublisherService.yaml
```

Check in Kubernetes Dashboard or through *kubernetes get pods* and *kubernetes get services* if the creation is complete – and what the port is that was assigned to the *EventBusPublisherService*.

```
c:\data\microservices-choreography-kubernetes-workshop-june2017\part3\event-bus-publisher>kubectl get services  
NAME           CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE  
eventbuslistenerservice  10.0.0.249  <nodes>      8096:32004/TCP  59m  
eventbuspublisherservice  10.0.0.172  <nodes>      8097:31836/TCP  1m
```

With http requests from your browser - or using CURL or Postman – you can trigger the Event Bus Publisher microservice into publishing events to the Event Bus:

```
http://<Docker Machine IP>:<port assigned to Kubernetes service>/publish?area=greenland&importance=high&color=orange
```

The screenshot shows a Postman interface. The URL field contains "192.168.99.100:31836/publish?area=greenland&importance=high&color=orange". The "Result" section displays the response: "Published Event to Topic event-bus".

The Event Bus Listener microservice is still running. Through the browser – or by looking at the logs for the Pod - you can check if it has consumed the event that was just published through the Event Bus Publisher microservice.

The screenshot shows a browser window with the URL "192.168.99.100:32004/event-bus". The JSON response shows an event with a Wish of "Hello To World" and an importance of "high". A red box highlights the meta information: "Produced by EventBusPublisher (0.8.3) from an HTTP Request", "area: greenland", "importance: high", and "color: orange".

```
topic: "event-bus"  
events:  
  0:  
    Wish: "Hello To World"  
    importance: "high"  
  5:  
    meta: "Produced by EventBusPublisher (0.8.3) from an HTTP Request"  
    area: "greenland"  
    importance: "high"  
    color: "orange"
```

Logs from event-bus-listener in event-bus-listener-ms

A Tr

```
2017-05-30T06:14:24.046228091Z event received
2017-05-30T06:14:24.050003936Z received message { topic: 'event-bus',
2017-05-30T06:14:24.050034921Z   value: '{"meta":"Produced by EventBusPublisher (0.8.3) from an HTTP
Request", "area": "greenland", "importance": "high", "color": "orange"}',
2017-05-30T06:14:24.050043188Z   offset: 10,
2017-05-30T06:14:24.050102342Z   partition: 0,
2017-05-30T06:14:24.050111979Z   highWaterOffset: 11,
2017-05-30T06:14:24.050118086Z   key: <Buffer 45 76 65 6e 74 42 75 73 45 76 65 6e 74>
2017-05-30T06:14:24.050158029Z received message object {"topic": "event-bus", "value": "{\"meta\": \"Produced by EventBusPublisher
(0.8.3) from an HTTP Request\", \"area\": \"greenland\", \"importance\": \"high\", \"color\": \"orange
\"}", "offset": 10, "partition": 0, "highWaterOffset": 11, "key": {"type": "Buffer", "data": [69, 118, 101, 110, 116, 66, 117, 115, 69, 118, 101, 110, 116]}}
2017-05-30T06:14:24.05016906Z actual event: {"meta": "Produced by EventBusPublisher (0.8.3) from an HTTP
Request", "area": "greenland", "importance": "high", "color": "orange"}
```

Logs from 5/30/17 4:53 AM to 5/30/17 6:14 AM

|< < > >|

When the HTTP Request to the Event Bus Publisher results in a message returned from the Event Bus Listener microservice, then we have succeeded. Microservice interaction in a most decoupled way. Time for some choreography!

Note: At this point, you can stop and delete all components currently running on Kubernetes. A quick way to tear all components down is:

```
kubectl delete po,svc,rc --all
```

## 4. Microservice Choreography – on Kubernetes and Kafa

In this section, a multi-step workflow is implemented using various microservices that dance together – albeit unknowingly. The choreography is laid down in a workflow prescription that each microservices knows how to participate in.

Note: directory part4 contains a Postman Collection: MicroServiceTestSet.postman\_collection with a few simple test calls for the microservices that we run in this section.

### Prepare Kafka Event Bus

The Kafka Event Bus was introduced in the previous section, using a Virtual Machine running on Virtual Box. We assume that this VM is (still) running. In this running VM, create two new topics (in a terminal window inside the VM):

```
kafka-topics --create --zookeeper localhost:2181 --replication-factor 1 --  
partitions 1 --topic workflowEvents
```

```
kafka-topics --create --zookeeper localhost:2181 --replication-factor 1 --  
partitions 1 --topic logTopic
```

The workflow choreography takes place through events published to and consumed from the first topic. Logging will be published to the second topic.

### Startup Cache Capability

In addition to the Kafka Event Bus, our microservices platform also provides a cache facility to the microservices. This cache facility is provided through Redis, by executing these commands:

```
kubectl run redis-cache --image=redis --port=6379  
kubectl expose deployment redis-cache --type=NodePort
```

With these commands, a deployment is created on the Kubernetes Cluster with a pod based on Docker Image *redis* and exposing port 6379. This deployment is subsequently exposed as a service, available for other microservices on the same cluster and for consumers outside the cluster, so we can check on the contents of the cache if we want to. Type ClusterIP instead of NodePort is enough to allow access to other microservices (pods) on the cluster.

### Run Microservice TweetReceiver

The microservice TweetReceiver exposes an API that expects HTTP Post Requests with the contents of a Tweet message. It will publish a workflow event to the Kafka Topic to report the tweet to the microservices cosmos to take care of.

In directory part4/TweetReceiver, run:

```
kubectl create -f TweetReceiverPod.yaml -f TweetReceiverService.yaml
```

With this command, a new Pod is launched that listens for HTTP Requests that report a Tweet. This can be done through a recipe from IFTTT or with a simple call from any HTTP client such as Postman.

Using `kubectl get services` you can inspect the service that is created and the port at which it is exposed.

## Run Microservice Workflow Launcher

The Workflow Launcher listens to the Kafka Topic for events of type NewTweetEvent. Whenever it consumes one of those, it will compose a workflow event with a workflow choreography definition for that specific tweet. The data associated with the workflow is stored in the cache and thus made available to other microservices.

Run the Workflow Launcher with:

```
kubectl create -f WorkflowLauncherPod.yaml
```

Note: ensure that the KAFKA\_HOST, ZOOKEEPER\_PORT and REDIS\_HOST and REDIS\_PORT are correct in the yaml file.

This microservices does not expose an external service – all it does is listen to events on the Kafka Topic

When the microservice is running, it will listen for NewTweetEvents published by the TweetReceiver. If you check out the logs for WorkflowLauncher, you should find that any request send to TweetReceiver will now lead to activity in the WorkflowLauncher.

## Run Microservice ValidateTweet

The next microservice takes care of validating tweets. It can do so based on workflow events or in response to direct HTTP requests.

Run the microservice in directory part4/ValidateTweet.

```
kubectl create -f ValidateTweetPod.yaml
```

Also to expose an API for this microservice:

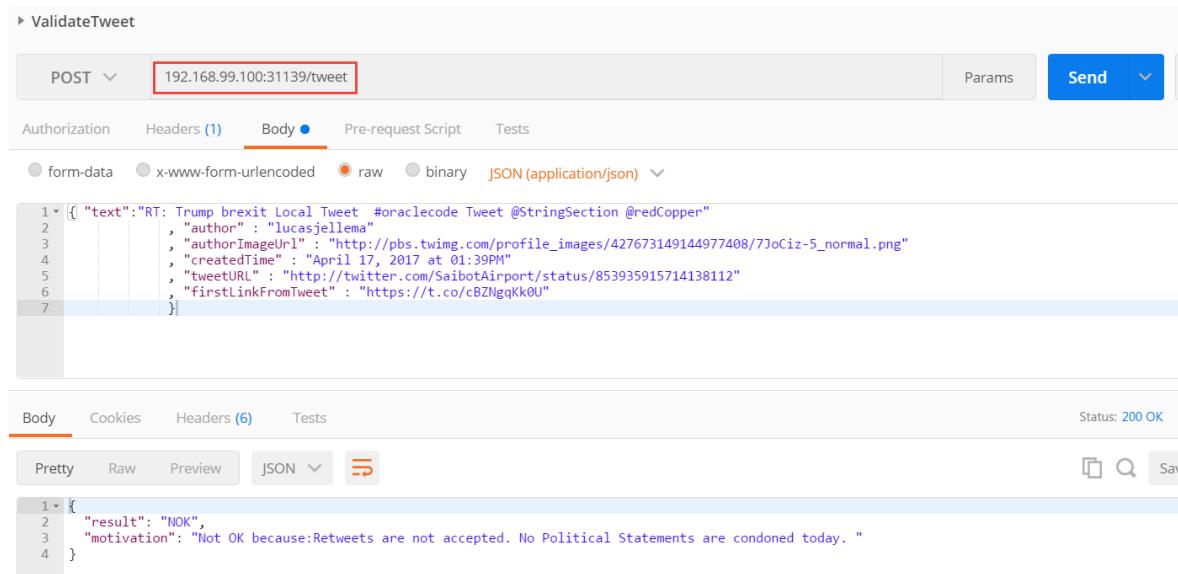
```
kubectl create -f ValidateTweetService.yaml
```

Using `kubectl get services` you can find out the port at which you can reach this microservice from your laptop. Using a simple curl, you can verify whether the microservices is running:

```
curl http://192.168.99.100:31139/about
```

```
c:\data\microservices-choreography-kubernetes-workshop-june2017\part4\ValidateTweet>curl http://192.168.99.100:31139/about
About TweetValidator API, Version 0.8$Supported URLs:/ping (GET)
;/tweet (POST)NodeJS runtime version v8.0.0.incoming headers("user-agent":"curl/7.30.0","host":"192.168.99.100:31139","accept":"*/*)
```

You can try out the functionality of this microservice with a simple POST request to the url:  
<http://192.168.99.100:31139/tweet>:



The screenshot shows a Postman interface with the following details:

- Method:** POST
- URL:** 192.168.99.100:31139/tweet
- Body:** Raw JSON (application/json)
- JSON Body:**

```
1 [ { "text": "RT: Trump brexit Local Tweet #oraclecode Tweet @StringSection @redCopper"
2 , "author" : "lucasjellema"
3 , "authorImageURL" : "http://pbs.twimg.com/profile_images/427673149144977408/7JoCiz-5_normal.png"
4 , "createdTime" : "April 17, 2017 at 01:39PM"
5 , "tweetURL" : "http://twitter.com/SaibotAirport/status/853935915714138112"
6 , "firstLinkFromTweet" : "https://t.co/BZNlgqKk0U"
7 } ]
```
- Response Status:** 200 OK
- Response Body:**

```
1 {
2   "result": "NOK",
3   "motivation": "Not OK because:Retweets are not accepted. No Political Statements are condoned today. "
4 }
```

## Run Microservice to Enrich Tweet

This microservice takes the tweet and enriches it with information about the author, any acronyms and abbreviations, related tweets and many more details. Well, that was the intent. And could still happen. But for now all the enrichment that takes place is very limited indeed. Sorry about that. However, this microservice will play its designated role in the workflow execution, according to the choreography suggested by the workflow launcher.

Run the microservice from directory part4/ TweetEnricher:

```
kubectl create -f TweetEnricherPod.yaml
```

Also to expose an API for this microservice:

```
kubectl create -f TweetEnricherService.yaml
```

Using *kubectl get services* you can find out the port at which you can reach this microservice from your laptop. Using a simple curl, you can verify whether the microservices is running:

```
curl http://192.168.99.100:30649/about
```

You can try out the functionality of this microservice with a simple POST request to the url:  
<http://192.168.99.100:31139/tweet>:

The screenshot shows the Postman application interface. The left sidebar lists various API endpoints under categories like History, Collections, and MicroServiceTestSet. The main workspace shows an 'Enrich Tweet' request. The 'Body' tab is selected, showing a raw JSON payload. The response body is displayed below, with the enrichment part highlighted in a red box.

POST 192.168.99.100:30649/tweet

Authorization Headers (1) Body Pre-request Script Tests

form-data x-www-form-urlencoded raw binary **JSON (application/json)**

```
1 { "text": "Nr 19 - Enricher Test Great, the greatest, fake,the fakest"
2 , "author": "lucasjellema"
3 , "authorImageUri": "http://pbs.twimg.com/profile_images/427673149144977408/7JoCiz-5_normal.png"
4 , "createdTime": "April 17, 2017 at 01:39PM"
5 , "tweetURL": "http://twitter.com/SaibotAirport/status/853935915714138112"
6 , "firstLinkFromTweet": "https://t.co/cBZNgqKk0U"
7 }
```

Body Cookies Headers (6) Tests

Pretty Raw Preview JSON

```
1 {
2   "enrichedTweet": {
3     "text": "Nr 19 - Enricher Test Great, the greatest, fake,the fakest",
4     "author": "lucasjellema",
5     "authorImageUri": "http://pbs.twimg.com/profile_images/427673149144977408/7JoCiz-5_normal.png",
6     "createdTime": "April 17, 2017 at 01:39PM",
7     "tweetURL": "http://twitter.com/SaibotAirport/status/853935915714138112",
8     "firstLinkFromTweet": "https://t.co/cBZNgqKk0U",
9     "enrichment": "Lots of Money",
10    "extraEnrichment": "Even more loads of money, gold, diamonds and some spiritual enrichment"
11 }
```

## Run Microservice TweetBoard

The last microservice we discuss does two things:

1. it responds to events in the workflow topic of type TweetBoardCapture (by adding an entry for the tweet in the workflow document) and
  2. it responds to HTTP Requests for the current tweet board [contents] by returning a JSON document with the most recent (maximum 25) Tweets that were processed by the workflow

This microservice is stateless. It uses the cache in the microservices platform to manage a document with the most recent tweets.

Run a Pod on Kubernetes with this microservice using this command in directory part4/TweetBoard:

```
kubectl create -f TweetBoardPod.yaml
```

and expose the microservice as a Service:

```
kubectl create -f TweetBoardService.yaml
```

The Kubernetes Dashboard now lists the following Pods or Microservices:

Name	Status	Restarts	Age
redis-cache-3679063315-8m9jl	Running	0	4 hours
tweet-board-ms	Running	0	27 minutes
tweet-enricher-ms	Running	0	an hour
tweet-validator-ms	Running	0	2 hours
tweetreceiver-ms	Running	0	4 hours
workflow-launcher-ms	Running	0	3 hours

Inspect the port assigned to this microservice using `kubectl get services`. Then access the microservice at `http://<docker machine IP>:<assigned port>/tweetBoard`. You will not yet see any tweets on the tweet board.

However, as soon as you publish a valid tweet to the TweetReceiver micro service, the workflow is initiated and through the choreographed dance that involves Workflow Launcher, TweetValidator, TweetEnricher and TweetBoard, that tweet will make its appearance on the tweet board.

Publish a few tweets for TweetReceiver and inspect the tweet board again. You can use the test messages in the Postman collection:

```

1 | "text": "Interesting workshop on Microservices, Kubernetes and Apache Kafka"
2 | "author": "memyselfandI"
3 | "authorImageURL": "http://pbs.twimg.com/profile_images/427673149144977408/7JoCiz-5_normal.png"
4 | "createdTime": "June 1, 2017 at 7:18PM"
5 | "firstLinkFromTweet": "https://t.co/cBZlNgqKk0U"
6 |

```

The Tweet Board looks like this:

```

1+ [
2+   "tweets": [
3+     {
4+       "text": "Champions League Final on Saturday. Juve has to work very hard to dominate Real Madrid",
5+       "author": "football_fan",
6+       "authorImageURL": "http://pbs.twimg.com/profile_images/427673149144977408/7JoCiz-5_normal.png",
7+       "createdTime": "June 1, 2017 at 7:20PM",
8+       "firstLinkFromTweet": "https://en.wikipedia.org/wiki/Juventus_F.C.",
9+       "enrichment": "Lots of Money",
10+      "extraEnrichment": "Even more loads of money, gold, diamonds and some spiritual enrichment"
11+    },
12+    {
13+      "text": "Interesting workshop on Microservices, Kubernetes and Apache Kafka",
14+      "author": "mehysel and I",
15+      "authorImageURL": "http://pbs.twimg.com/profile_images/427673149144977408/7JoCiz-5_normal.png",
16+      "createdTime": "June 1, 2017 at 7:18PM",
17+      "firstLinkFromTweet": "https://t.co/c8ZhigKk0U",
18+      "enrichment": "Lots of Money",
19+      "extraEnrichment": "Even more loads of money, gold, diamonds and some spiritual enrichment"
20+    },
21+    {
22+      "text": "Hm ?! Great - the greatest fake the fakest"
}

```

By checking the topic `logTopic` in the Kafka tool in the VM running Kafka, we get a feel for the workflows that were executed – and for tweets that did not get through validation:

Partition	Offset	Key	Message
0	11269	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetValidator\\",\\"message\\":\\"Validated Tweet (outcome: \\\\"result\\\":\\\"OK\\\",\\\"motivation\\\":\\\"Not OK because:No fake authors (John Doe is not acceptable)\\\")\\\"\\}"
0	11268	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetBoard\\",\\"message\\":\\"Added Tweet to TweetBoard - (workflowConversationIdentifier:OracleCodeTweetProcessor149623...\\")\\\"\\}"
0	11267	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetEnricher\\",\\"message\\":\\"Enriched Tweet - (workflowConversationIdentifier:OracleCodeTweetProcessor149623...\\")\\\"\\}"
0	11266	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetValidator\\",\\"message\\":\\"Validated Tweet (outcome: \\\\"result\\\":\\\"OK\\\",\\\"motivation\\\":\\\"perfectly ok twe...\\\")\\\"\\}"
0	11265	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetValidator\\",\\"message\\":\\"Validated Tweet (outcome: \\\\"result\\\":\\\"OK\\\",\\\"motivation\\\":\\\"perfectly ok twe...\\\")\\\"\\}"
0	11264	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetEnricher\\",\\"message\\":\\"Enriched Tweet - (workflowConversationIdentifier:OracleCodeTweetProcessor149622...\\")\\\"\\}"
0	11263	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetValidator\\",\\"message\\":\\"Validated Tweet (outcome: \\\\"result\\\":\\\"OK\\\",\\\"motivation\\\":\\\"perfectly ok twe...\\\")\\\"\\}"
0	11262	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetBoard\\",\\"message\\":\\"Added Tweet to TweetBoard - (workflowConversationIdentifier:OracleCodeTweetProcessor...\\")\\\"\\}"
0	11261	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetEnricher\\",\\"message\\":\\"Enriched Tweet - (workflowConversationIdentifier:OracleCodeTweetProcessor149622...\\")\\\"\\}"
0	11260	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetValidator\\",\\"message\\":\\"Validated Tweet (outcome: \\\\"result\\\":\\\"OK\\\",\\\"motivation\\\":\\\"perfectly ok twe...\\\")\\\"\\}"
0	11259	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetValidator\\",\\"message\\":\\"Validated Tweet (outcome: \\\\"result\\\":\\\"OK\\\",\\\"motivation\\\":\\\"perfectly ok twe...\\\")\\\"\\}"
0	11258	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetValidator\\",\\"message\\":\\"Validated Tweet (outcome: \\\\"result\\\":\\\"OK\\\",\\\"motivation\\\":\\\"perfectly ok twe...\\\")\\\"\\}"
0	11257	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetEnricher\\",\\"message\\":\\"Enriched Tweet - (workflowConversationIdentifier:OracleCodeTweetProcessor149622...\\")\\\"\\}"
0	11256	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetValidator\\",\\"message\\":\\"Validated Tweet (outcome: \\\\"result\\\":\\\"OK\\\",\\\"motivation\\\":\\\"perfectly ok twe...\\\")\\\"\\}"
0	11255	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetEnricher\\",\\"message\\":\\"Enriched Tweet - (workflowConversationIdentifier:OracleCodeTweetProcessor149622...\\")\\\"\\}"
0	11254	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetValidator\\",\\"message\\":\\"Validated Tweet (outcome: \\\\"result\\\":\\\"OK\\\",\\\"motivation\\\":\\\"perfectly ok twe...\\\")\\\"\\}"
0	11253	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetEnricher\\",\\"message\\":\\"Enriched Tweet - (workflowConversationIdentifier:OracleCodeTweetProcessor149622...\\")\\\"\\}"
0	11252	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"WorkflowLauncher\\",\\"message\\":\\"WorkflowLauncher - (WorkflowIdentifier:WorkflowIdentifierWorkflowLauncher149622061061)\\\"\\}"
0	11251	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetValidator\\",\\"message\\":\\"Validated Tweet (outcome: \\\\"result\\\":\\\"OK\\\",\\\"motivation\\\":\\\"perfectly ok twe...\\\")\\\"\\}"
0	11250	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetValidator\\",\\"message\\":\\"Validated Tweet (outcome: \\\\"result\\\":\\\"OK\\\",\\\"motivation\\\":\\\"perfectly ok twe...\\\")\\\"\\}"
0	11249	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"WorkflowLauncher\\",\\"message\\":\\"WorkflowLauncher - (WorkflowIdentifier:WorkflowIdentifierWorkflowLauncher149622061061)\\\"\\}"
0	11248	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"WorkflowLauncher\\",\\"message\\":\\"Initialized new workflow OracleCodeTweetProcessor triggered by NewTweetEvent....\\\"\\}"

Note how you can filter events in this tool, for example by the workflow identifier:

Partition	Offset	Key	Message
0	11268	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetBoard\\",\\"message\\":\\"Added Tweet to TweetBoard - (workflowConversationIdentifier:OracleCodeTweetProcessor149623...\\")\\\"\\}"
0	11267	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetEnricher\\",\\"message\\":\\"Enriched Tweet - (workflowConversationIdentifier:OracleCodeTweetProcessor149623...\\")\\\"\\}"
0	11266	logEntry	"\{\\"logLevel\\":\\"info\\",\\"module\\":\\"TweetValidator\\",\\"message\\":\\"Validated Tweet (outcome: \\\\"result\\\":\\\"OK\\\",\\\"motivation\\\":\\\"perfectly ok twe...\\\")\\\"\\}"

## Optional next steps

Some obvious next steps around this workflow implementation and the microservices platform used are listed below:

- Expose TweetReceiver to the public internet (for example using ngrok) and create an IFTTT recipe to invoke the TweetReceiver for selected tweets; this allows the workflow to act on real tweets
- Run multiple instances (replicas) of the Pods that participate in the workflow (note: they are all stateless and capable of horizontally scaling; however, here is not currently any (optimistic) locking implemented on cache access, so race conditions are although rare still possible!)
- Change the workflow
  - create a new workflow plan that for example changes the sequence of validation and enrichment or even allows them to be in parallel (see example below)
  - add a step to the workflow (and a microservice to carry out that step)
- Create microservice (or simple Node.js script) to check the contents of Redis Cache (for example the tweetboard document or the payload for a specific workflow instance) (see example below)
- Retrieve Logging from Kafka Topic
- Run Kafka inside the Kubernetes cluster
- Implement one or more microservices on a cloud platform instead of on the local Kubernetes cluster; that requires use of an event bus on the cloud (e.g. Kafka on the cloud, such as CloudKarafka) and possibly (if the local Kafka is retained as well) a bridge between the local and the cloud based event bus.

## Inspect the cache contents

A simple cache inspector is available in directory part4/CacheInspector. You can run the node application CacheInspector.js locally, or you can launch another Pod on Kubernetes using

```
kubectl create -f CacheInspectorPod.yaml -f CacheInspectorService.yaml
```

Using a URL like:

```
192.168.99.100:32663/cacheEntry?key=OracleCodeTweetProcessor1496230025295
```

you can retrieve the workflow routing slip plus payload for a specific workflow instance. Of course you need to use your own IP address, assigned port and workflow identifier.

▶ Retrieve CacheEntry from Kubernetes

```

1  {
2    "workflowType": "oracle-code-tweet-processor",
3    "workflowConversationIdentifier": "OracleCodeTweetProcessor1496234564135",
4    "creationTimeStamp": 1496234422508,
5    "creator": "WorkflowLauncher",
6    "actions": [
7      {
8        "id": "EnrichTweetWithDetails",
9        "type": "EnrichTweet",
10       "status": "complete",
11       "result": "OK",
12       "conditions": []
13     },
14     {
15       "id": "ValidateTweetAgainstFilters",
16       "type": "ValidateTweet",
17       "status": "complete",
18       "result": "OK",
19       "conditions": [
20         {
21           "action": "EnrichTweetWithDetails",
22           "status": "complete",
23           "result": "OK"
24         }
25       ]
26     },
27   {
28     "id": "ContinueToTweetRecord"
29   }
30 }
  
```

## Change the Workflow

A somewhat updated version of the Tweet Workflow is available in part4\WorkflowLauncher\WorkflowLauncherV2.js. In this version, Enrichment is done before Validation. This is defined in the *message* variable at the bottom of the program.

You can force replace the pod currently running on Kubernetes for workflow-launcher-ms

```
kubectl replace -f WorkflowLauncherV2Pod.yaml --force
```

to try this later version of the workflow.

Publish a new tweet to TweetReceiver. Now when you inspect the logTopic on Kafka, the logs for the Kubernetes pods or the workflow document in the cache (through the CacheInspector) you will find that in the latest workflow, enrichment is done before validation.



## Appendix: Running Apache Kafka and Zookeeper in Docker and in Kubernetes

<https://howtoprogram.xyz/2016/07/21/using-apache-kafka-docker/>

Using the instructions at <http://docs.confluent.io/current/cp-docker-images/docs/quickstart.html> and working either directly in Docker or through Docker Quickstart Terminal (on Windows or MacOS), we can get Apache Kafka (and Zookeeper) running, in two separate Docker containers.

```
docker run -d \
--net=host \
--name=zookeeper \
-e ZOOKEEPER_CLIENT_PORT=32181 \
confluentinc/cp-zookeeper:3.2.1
```

```
docker run -d \
--net=host \
--name=kafka \
-e KAFKA_ZOOKEEPER_CONNECT=localhost:32181 \
-e KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://localhost:29092 \
confluentinc/cp-kafka:3.2.1
```

```
docker run -d \
--net=host \
--name=schema-registry \
-e SCHEMA_REGISTRY_KAFKASTORE_CONNECTION_URL=localhost:32181 \
-e SCHEMA_REGISTRY_HOST_NAME=localhost \
-e SCHEMA_REGISTRY_LISTENERS=http://localhost:8081 \
confluentinc/cp-schema-registry:3.2.1
```

```
docker run -d \
--name=kafka-rest4 \
-p 8082:8082 \
-e KAFKA_REST_ZOOKEEPER_CONNECT=localhost:32181 \
-e KAFKA_REST_LISTENERS=http://localhost:8082 \
-e KAFKA_REST_SCHEMA_REGISTRY_URL=http://localhost:8081 \
-e KAFKA_REST_HOST_NAME=localhost \
confluentinc/cp-kafka-rest:3.2.1
```

```
docker inspect --format '{{ .NetworkSettings.IPAddress }}' kafka-rest
```

to get ip address to access REST Proxy?

Alternative:

<https://howtoprogram.xyz/2016/07/21/using-apache-kafka-docker/>

Based on <https://github.com/CloudTrackInc/kubernetes-kafka> - we get Apache Kafka running on our minikube cluster.

From directory part3/kubernetes-kafka:

Zookeeper

```
kubectl apply -f zoo-rc.yaml --record
```

This should start a Pod based on the digitalwonderland/zookeeper image for Apache Zookeeper (<https://github.com/digital-wonderland/docker-zookeeper>).

Next, expose this Pod as a Service:

```
kubectl apply -f zoo-service.yaml --record
```

Run ReplicationController for Kafka

```
kubectl apply -f kafka-rc.yaml --record
```

And expose this Pod as a service:

```
kubectl apply -f kafka-service.yaml
```

Kafka Rest Proxy

<https://hub.docker.com/r/confluentinc/cp-kafka-rest/>

Run Kafka Rest Proxy – listening at port 8082, connecting to Zookeeper and from there to Kafka broker

```
kubectl apply -f kafka-rest-proxy-rc.yaml
```

And to expose the REST proxy outside the cluster

```
kubectl apply -f kafka-rest-proxy-service.yaml
```

An overview of the three services now running to implement the Event Bus platform service:

```
kubectl get services
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kafka	10.0.0.42	<nodes>	9092:32124/TCP	50m
kafka-restproxy-svc	10.0.0.218	<nodes>	8082:32626/TCP	1m
kafka-zoo-svc	10.0.0.237	<nodes>	2181:30686/TCP,2888:32429/TCP,3888:32494/TCP	1h
kubernetes	10.0.0.1	<none>	443/TCP	1d

To try out the Event Bus configuration – we can ask the Kafka Rest Proxy for a list of topics available on the Kafka Cluster:

```
curl "http://192.168.99.100:32626/topics"
```

The response:

```
c:\data\microservices-choreography-kubernetes-workshop-june2017\part3\kubernetes-kafka>curl "http://192.168.99.100:32626/topics"  
["demo-topic"]
```

The fact that we get a response reveals that the Rest Proxy talked to Zookeeper and got in touch with the Kafka Broker. All three pods that constitute the Event Bus are in business.

```
curl "http://192.168.99.100:32626/topics/demo-topic"
```

In directory part3

```
kubectl apply -f kafka.yaml
```