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

May 2018

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) Kubernetes, Redis, Node.js, Apache Kafka.

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

https://github.com/lucasjellema/workshop-event-bus-microservice-choreography-may-2018

1. Prepare a local Kubernetes environment

We will work with Docker Containers in this workshop. You will be running multiple Docker Containers inside a Kubernetes cluster and have them interact. We will be using Kubernetes in the minikube cluster incarnation.

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:

- VirtualBox
- Kubectl
- Minikube
- Moba Xterm
- Postman

Windows and MacOS: VirtualBox

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.

Linux: VirtualBox

If your operating system is Linux to start with, you still need to install VirtualBox as well as Kubectl.

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 MobaXTerm on Windows

MobaXterm is not a required tool for this workshop. However, for opening SSH sessions into the minikube host VM and the individual Pods, it is very convenient if you are on Windows. You will find details, installation instructions and downloadable software at: https://mobaxterm.mobatek.net/

Install Postman

Postman is a tool for API development – in particular for testing. You have probably worked with it and perhaps it is already set up on your system. Postman is very convenient for making calls to REST APIs – it is among others things a GUI alternative for cURL.

Postman – like MobaXterm - is not mandatory for this workshop. However, the workshop resources contain a Postman Test Collection that is very handy to make use of, so I would recommend that you get Postman going in order to leverage that Test Collection. You will find the details on Postman at this site: https://www.getpostman.com/ .

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*.

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
```

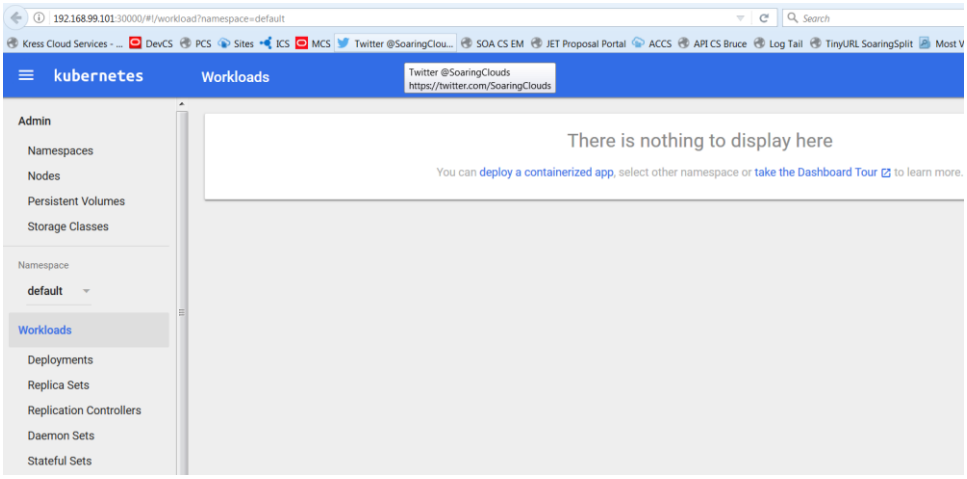
Note: we can provide flags to override the default values for disk-size and memory size – as shown below – or edit minikube config file to override the defaults all the time. The config file is located at ~/.minikube/config/config.json, until explicitly changed. For an overview of all configurable properties, check out: https://darkowlzz.github.io/post/minikube-config/ .

```
minikube start  --disk-size 50g --memory 6096

(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
```

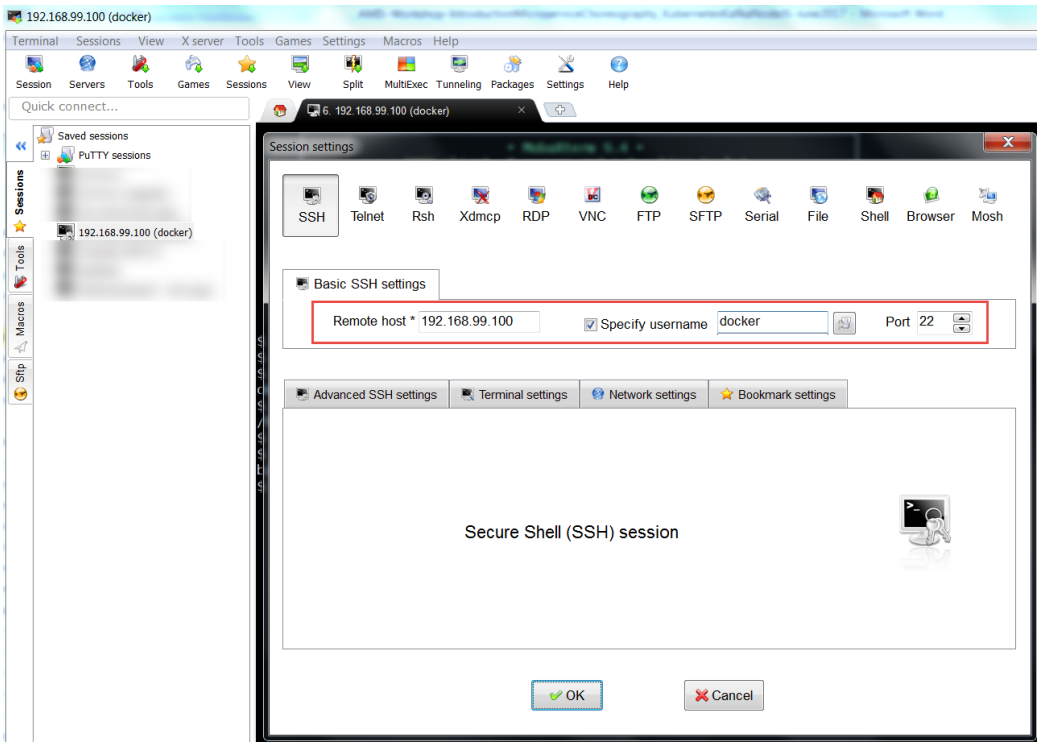
To check on the nodes of the cluster, we can do:

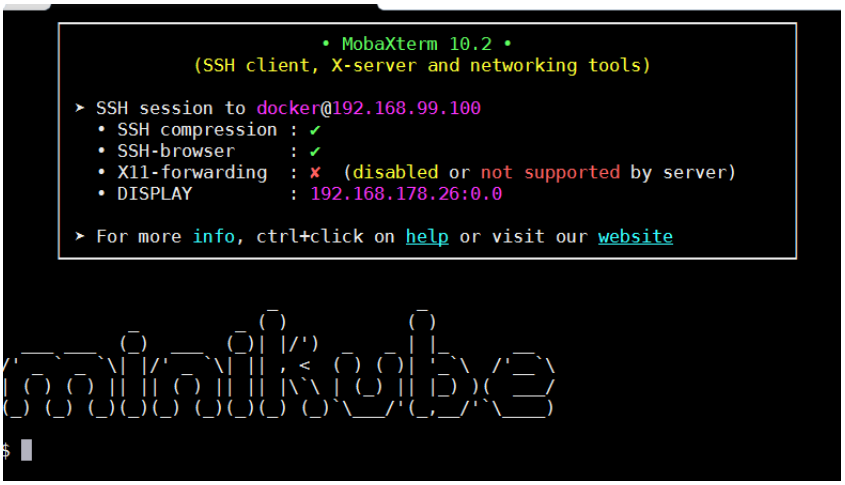
```
kubectl.exe get nodes
```

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

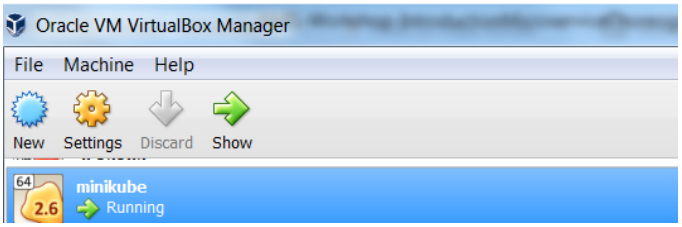
Test connect into the minikube VM through an SSH session:

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

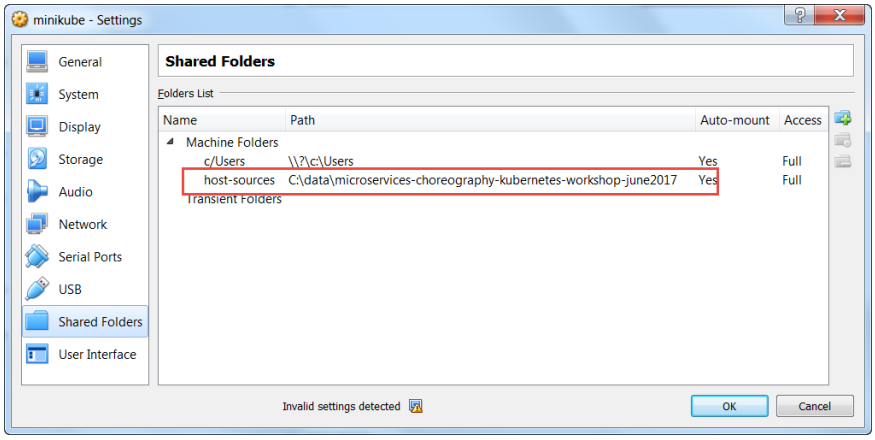




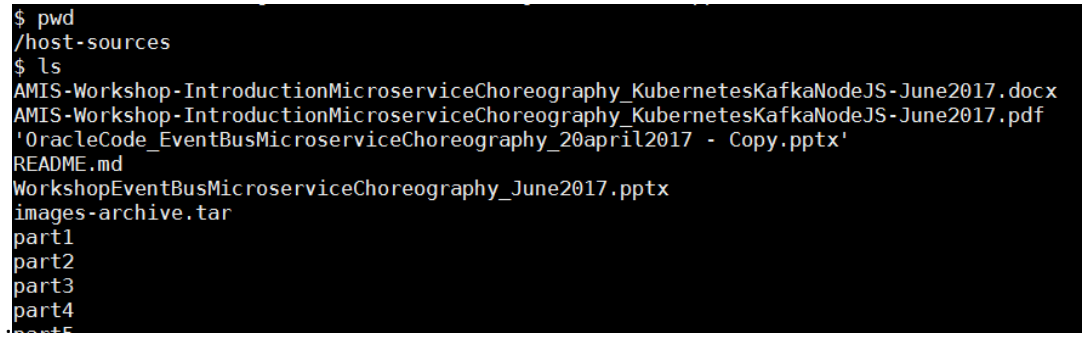
Note: after running for the first time, a new VM called minikube will show up in the Virtual Box GUI.



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:



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
```

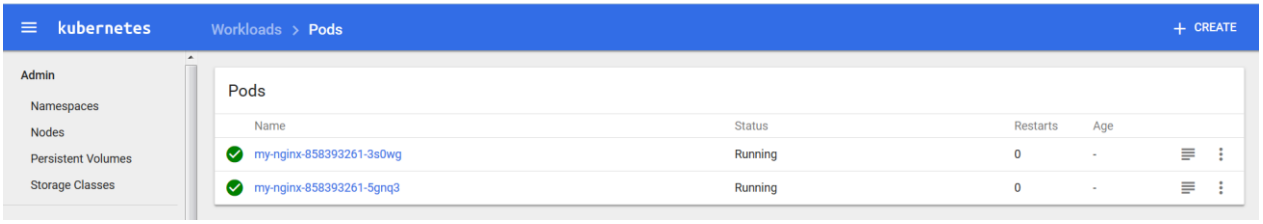
```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl run my-nginx --image=nginx --replicas=2 --port=80
deployment "my-nginx" created
```

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:



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

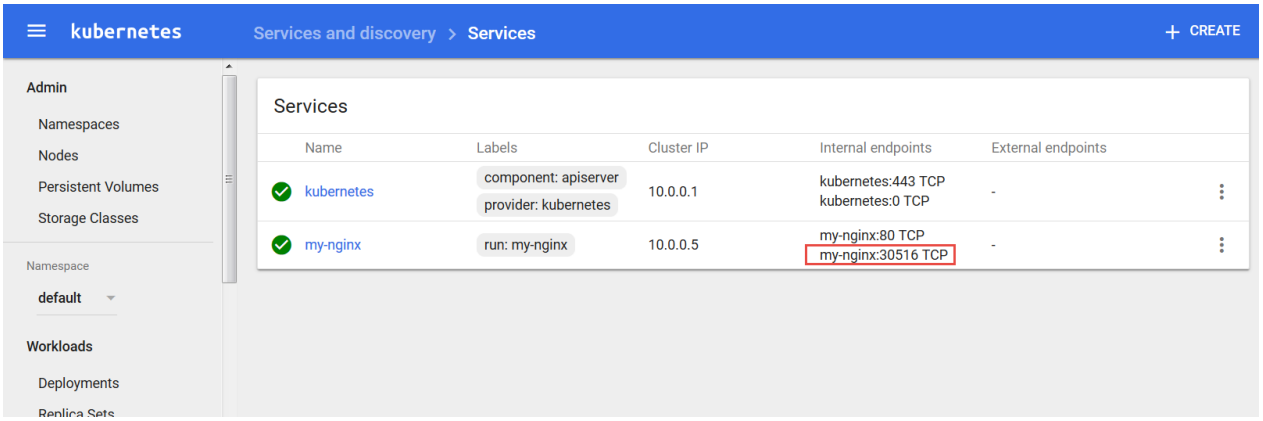
```
# expose your deployment as a service
kubectl expose deployment my-nginx --type=NodePort
```

```
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
```

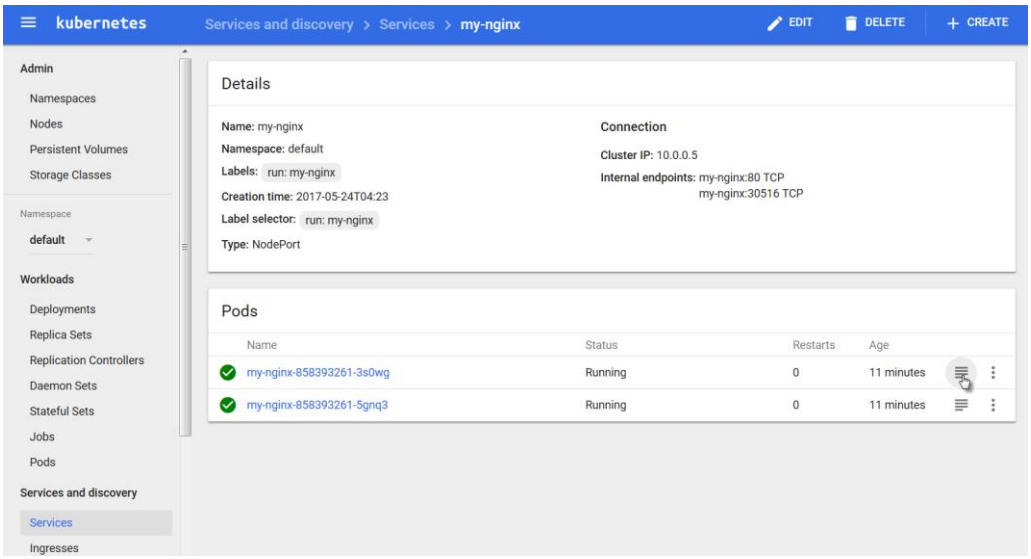
```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl get services
NAME         CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE
kubernetes   10.0.0.1     <none>        443/TCP          7h
my-nginx     10.0.0.5     <nodes>       80:30516/TCP     43s
```

And in the Dashboard:

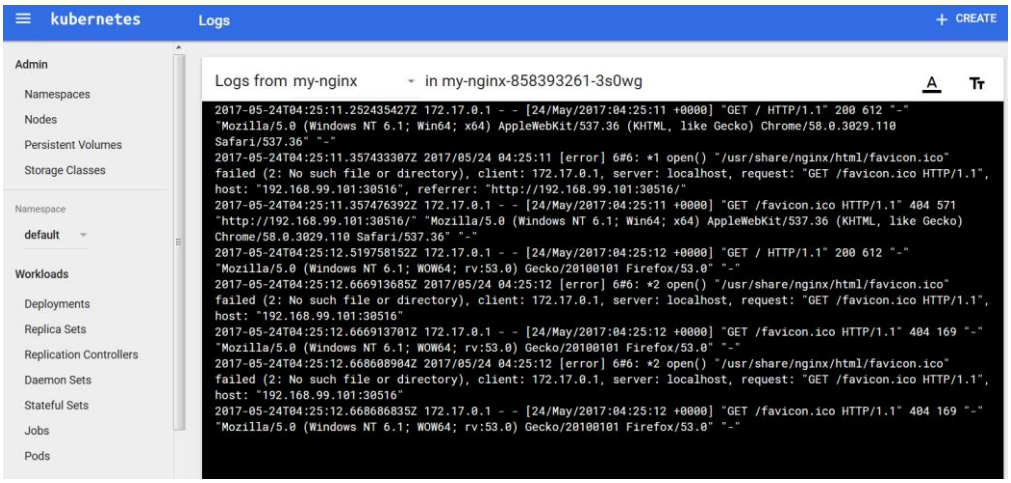



```
# Access your service from your default browser
minikube service my-nginx
```

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:



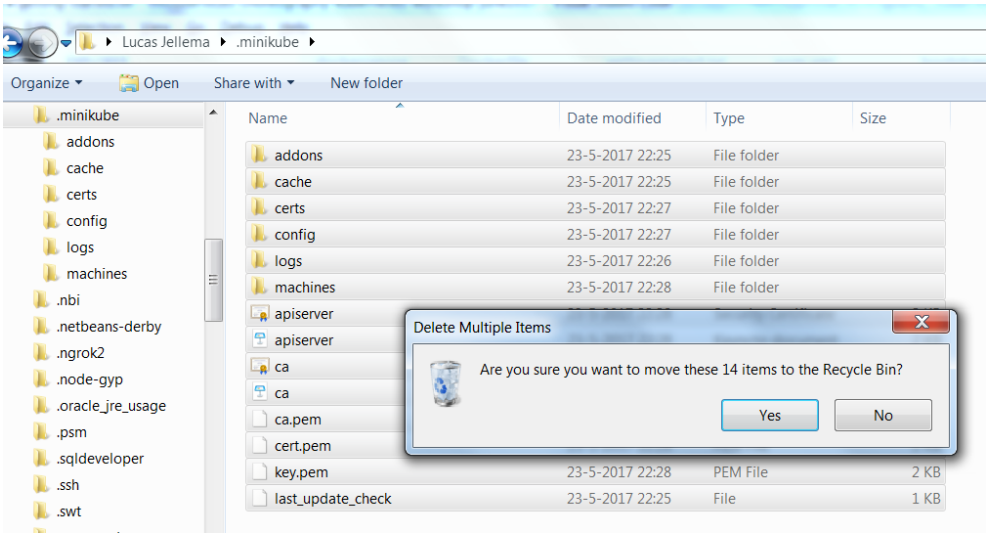
The logging will be shown:



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

Tip

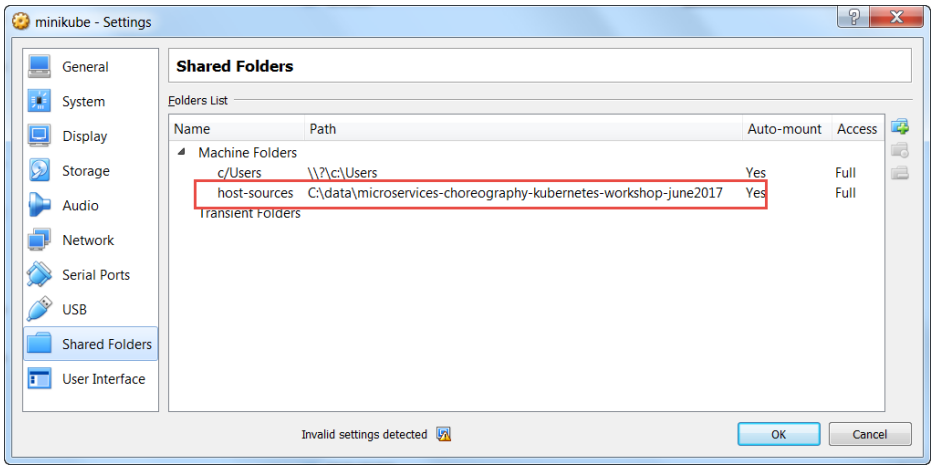
Note: if you have trouble creating, running or restarting minikube, it may help to clear the directory .minikube under the current user directory:



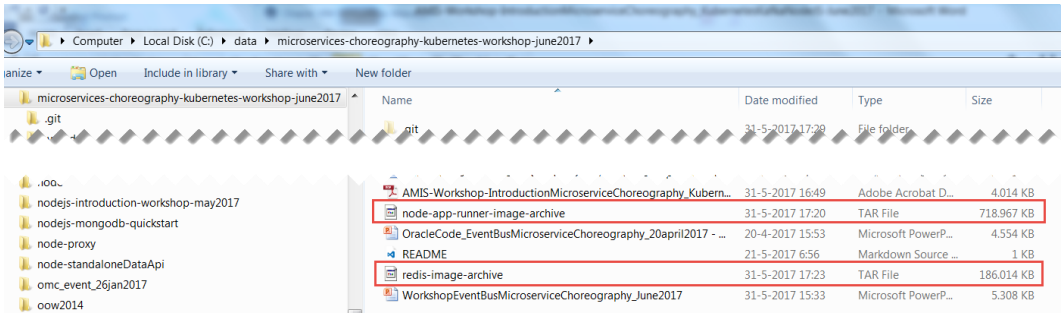
Tip 2 – No Internet Connection:

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
c59fa6cbcbd9: 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:

```
$ docker 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 v1.6.0              418701f962f2        10 weeks ago        108.6 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 MB
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, 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 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 be restarted upon failure is not easy with only Docker. Enter Kubernetes, a container management platform.

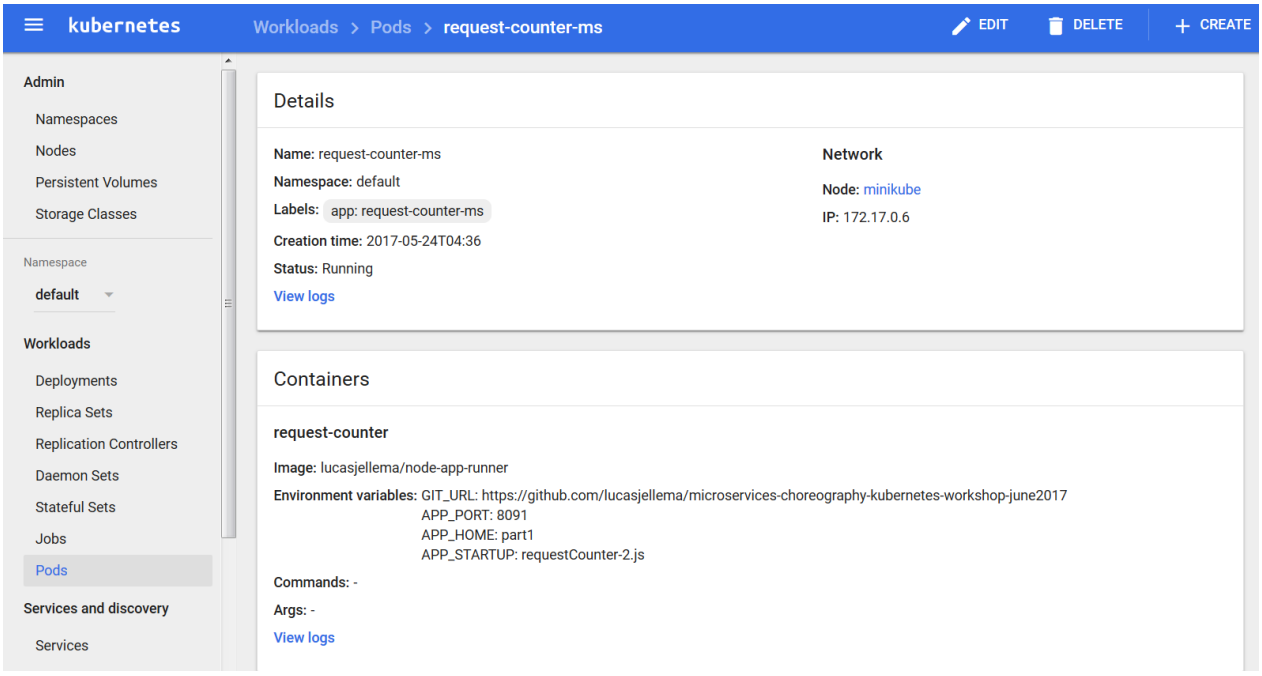
Let's run one instance of one standalone microservice on Kubernetes. From workshop directory part1/kubernetes, run

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

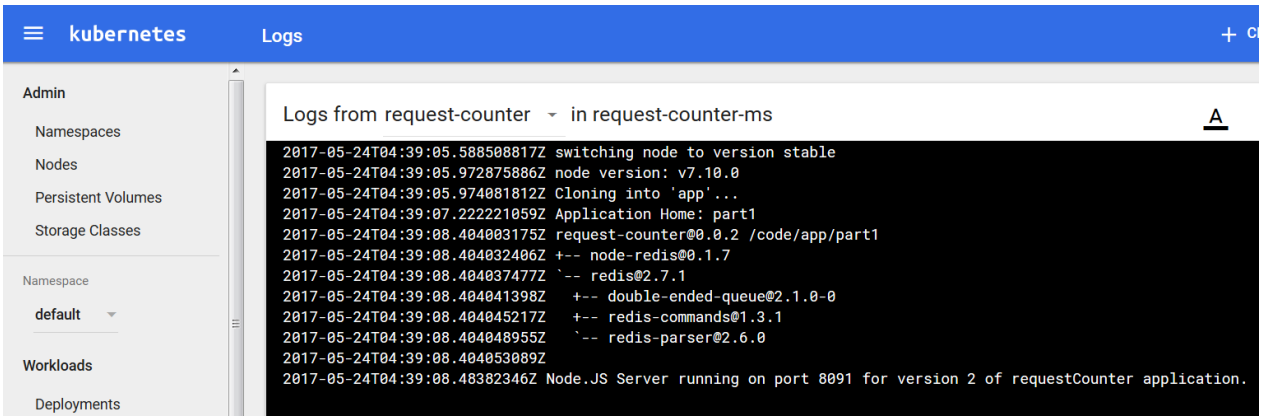
```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl create -f pod.yaml
pod "request-counter-ms" created
```

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):



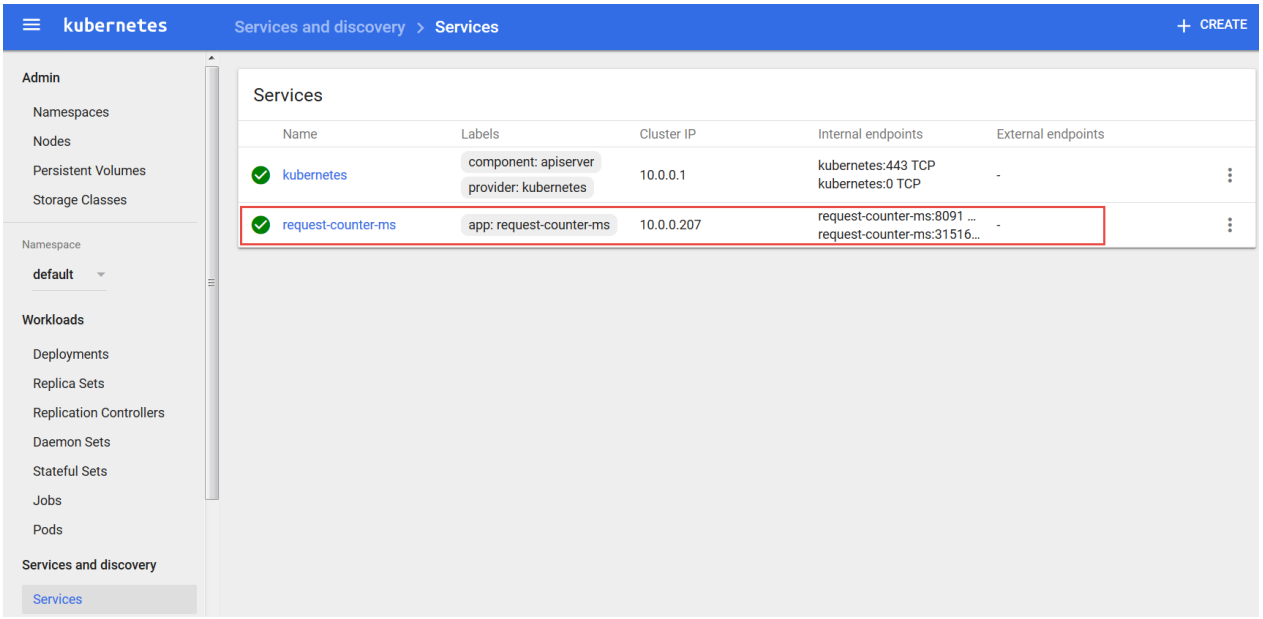
and verify the logging:



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
```



Using *kubectl get services* to inspect the service:

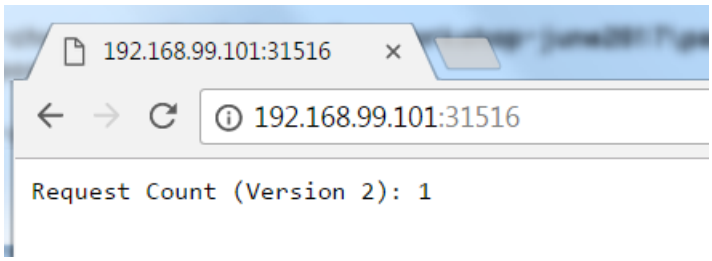
```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl get services
NAME                CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE
kubernetes          10.0.0.1     <none>        443/TCP          8h
request-counter-ms  10.0.0.207   <nodes>       8091:31516/TCP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

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

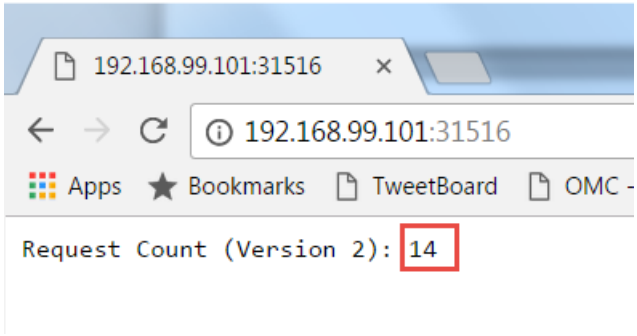
```
minikube.exe service --url=true request-counter-ms
```

```
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.



You may notice that with each request from the browser, the counter increases by two. Do you know why (not just by one)? Try to invoke the same URL using curl from the command line.

Does the same increase by two phenomenon occur?

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 in directory part1\kubernetes.

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

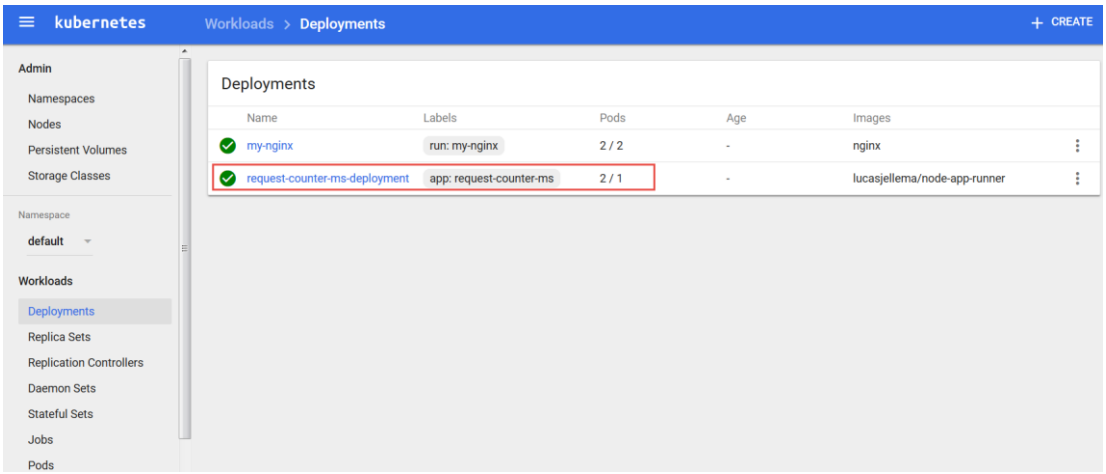
```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl create -f deployment.yaml
deployment "request-counter-ms-deployment" created
```

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-cc
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:



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
```

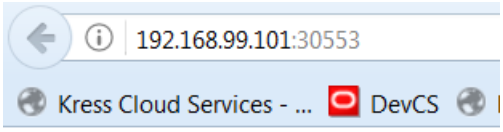
```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl.exe expose deployment request-counter-ms-deployment --type=NodePort
service "request-counter-ms-deployment" exposed
```

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
```

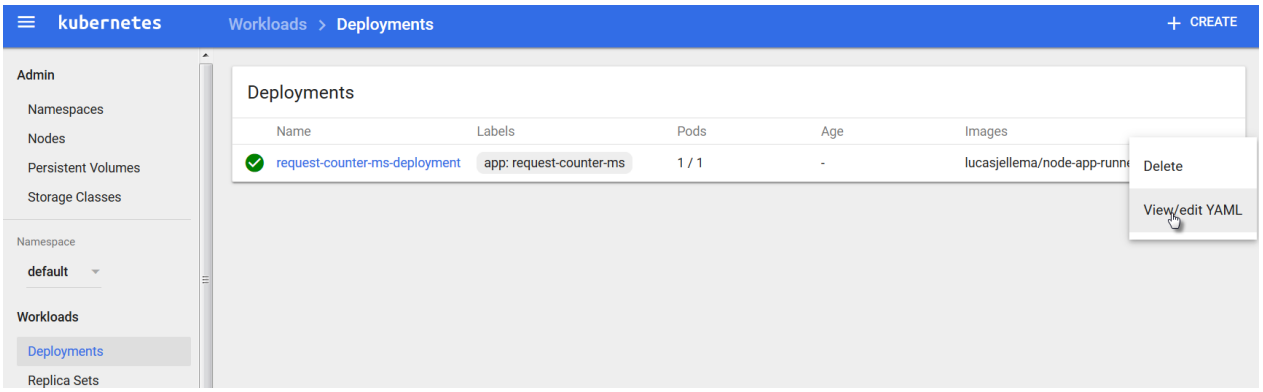



```
Request Count (Version 2): 1
```

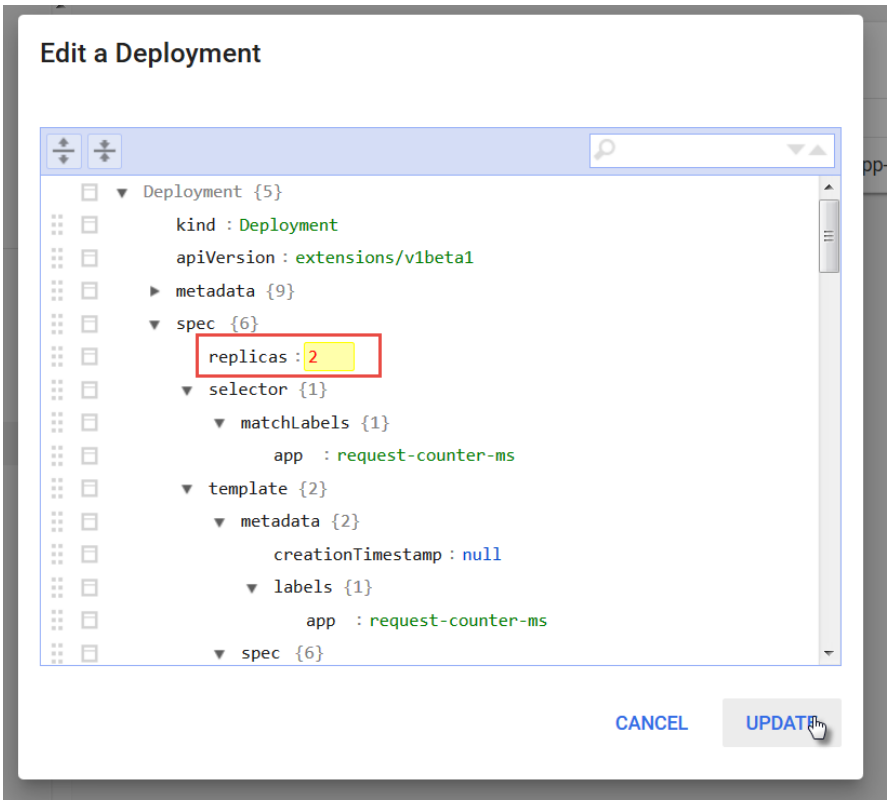
Hit the url a few more times, increasing the counter as you go.

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*.

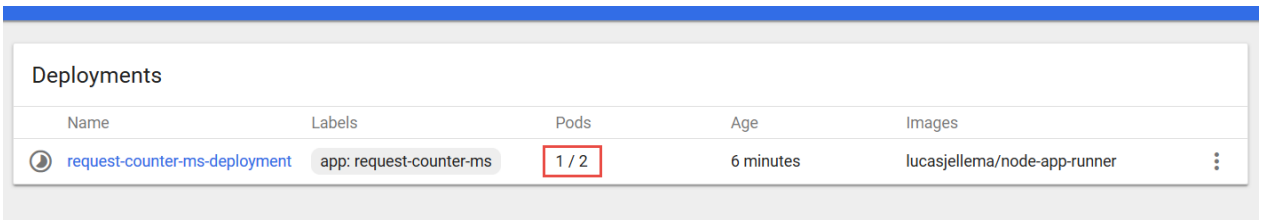


Change the value of the attribute replicas from 1 to 2:

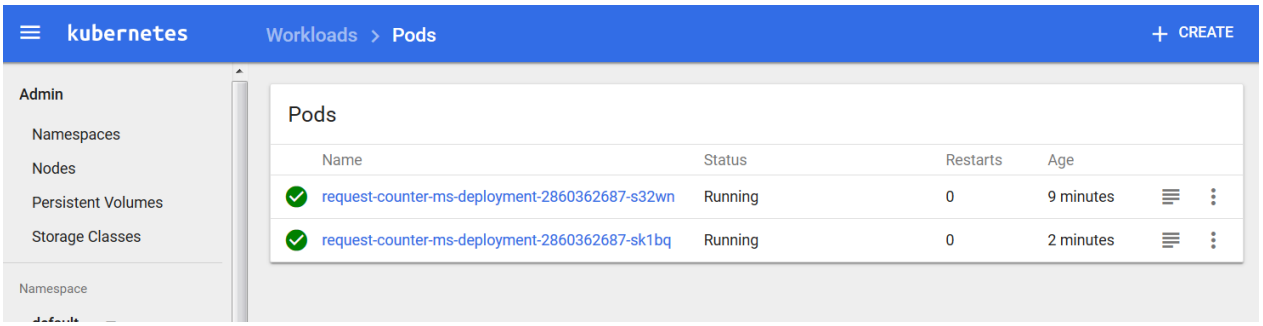


and click on Update.

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



After a little while, two pods are running:



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>
```

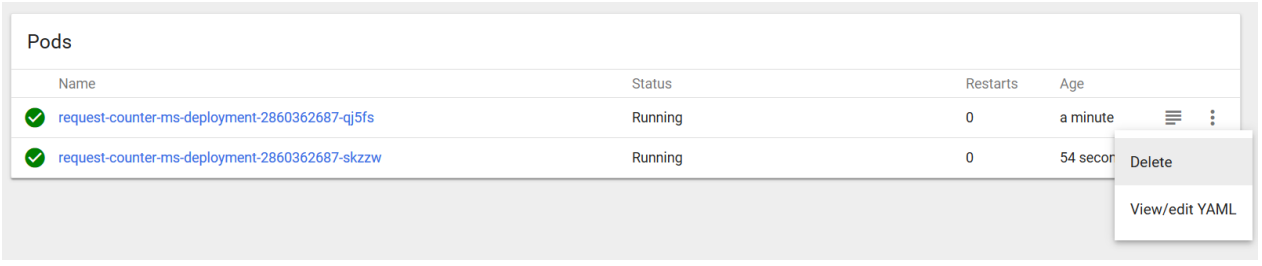
Your results will be similar to the following:

```
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 an endpoint exposed from the cluster, and traffic is distributed over the available Pods.

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



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

Create a new 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:

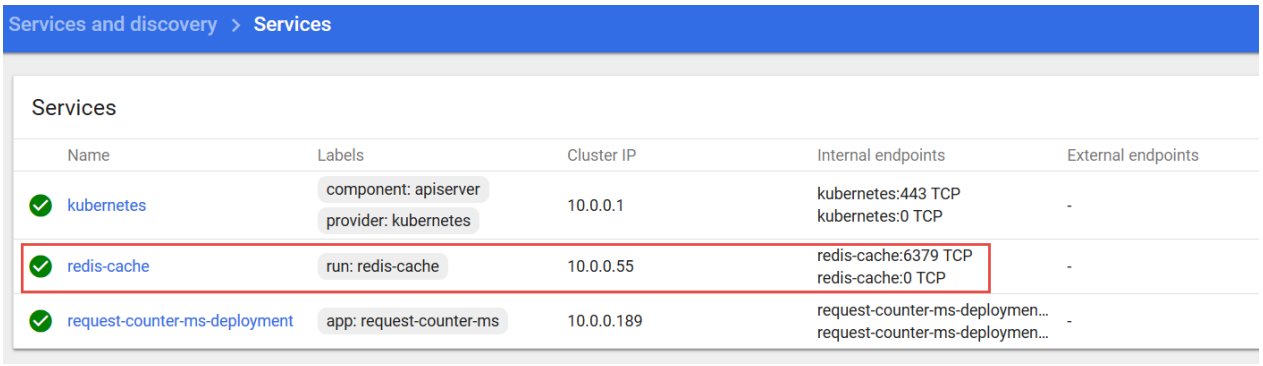
```
kubectl get services
```

```
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.



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
```

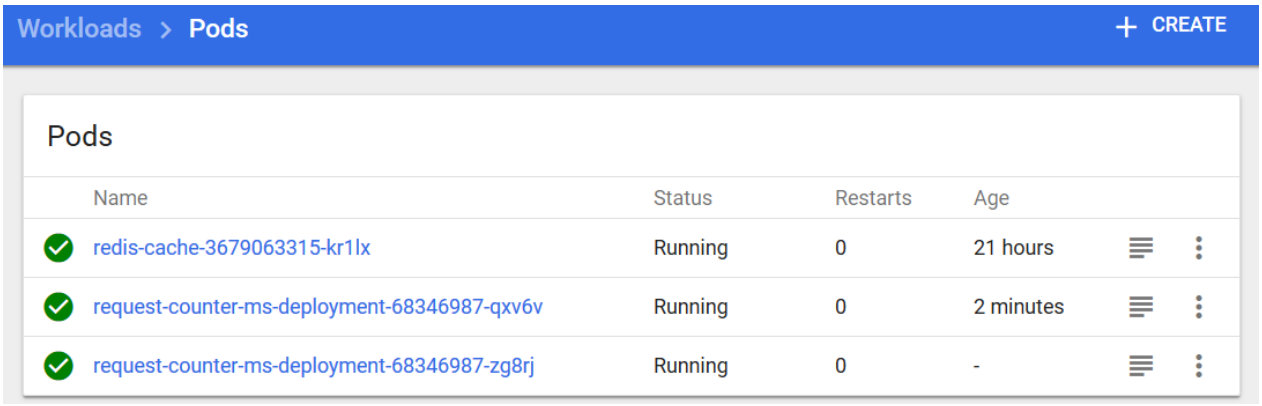
```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl rollout status deployment request-counter-ms-deployment
deployment "request-counter-ms-deployment" successfully rolled out
```

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

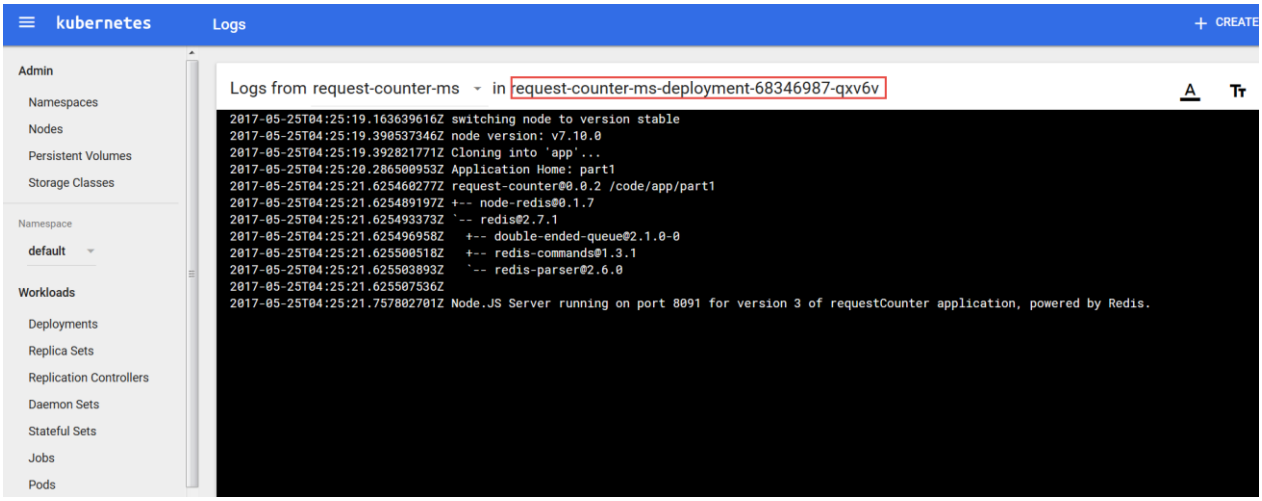
```
kubectl rollout history deployment request-counter-ms-deployment
```

```
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
```

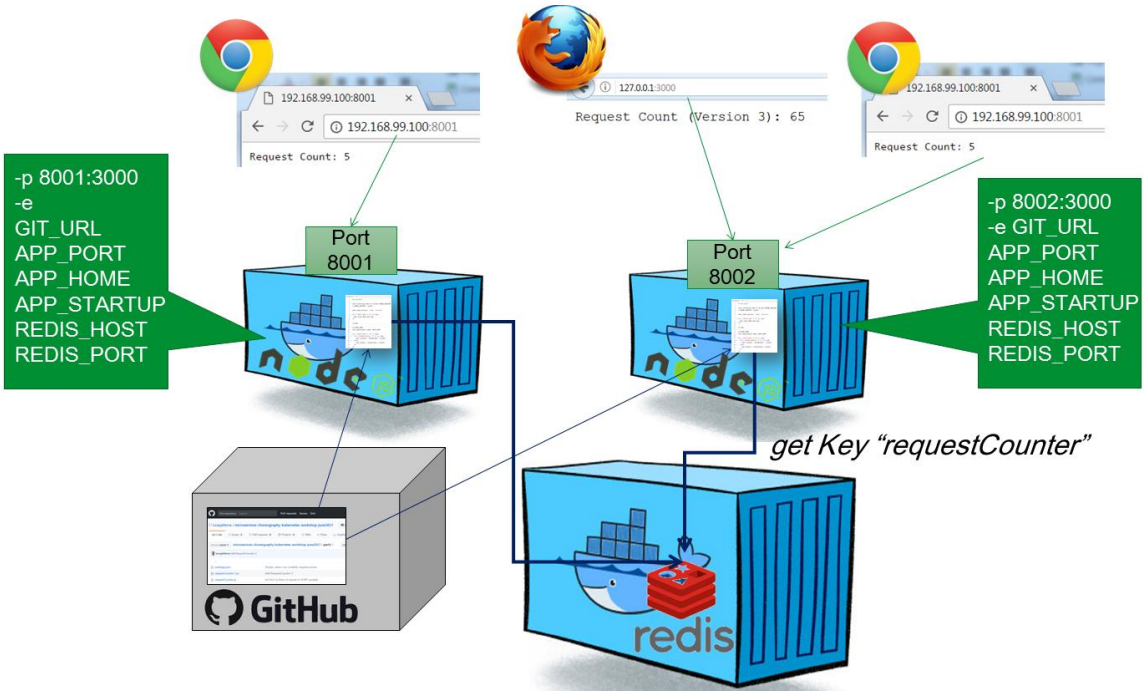
In the dashboard, if we are quick we can see the pods being restarted. If we are not so fast, we can see both pods running (again) – having been restarted fairly recently:



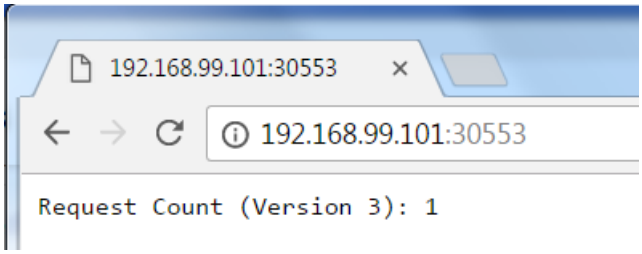
Both Pods are running again, from the logs you can tell that both are running the new version of the application – powered by Redis.



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
```

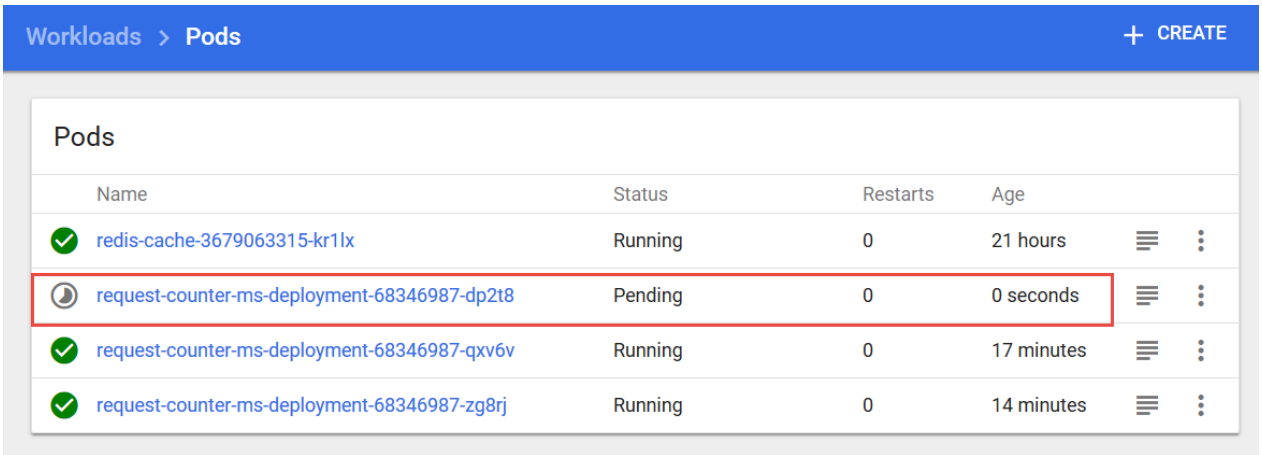
This time round, we do not get two counters incrementing in the background. Just a single incrementing value. And supposedly, that value is stored in the Redis cache where both instances access and manipulate it.

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)

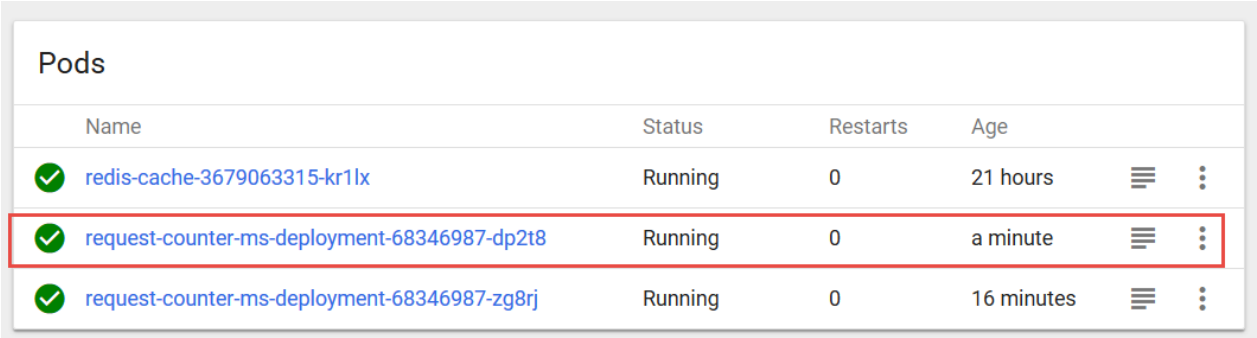
```
kubectl delete pod request-counter-ms-deployment-68346987-qxv6v --grace-period=60
```

```
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 :



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



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-kr11x               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-dp2t8   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-kr11x
pod "redis-cache-3679063315-kr11x" 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-68346987-5w16v   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 <name of 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 <name of 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 <name of pod> command
```

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

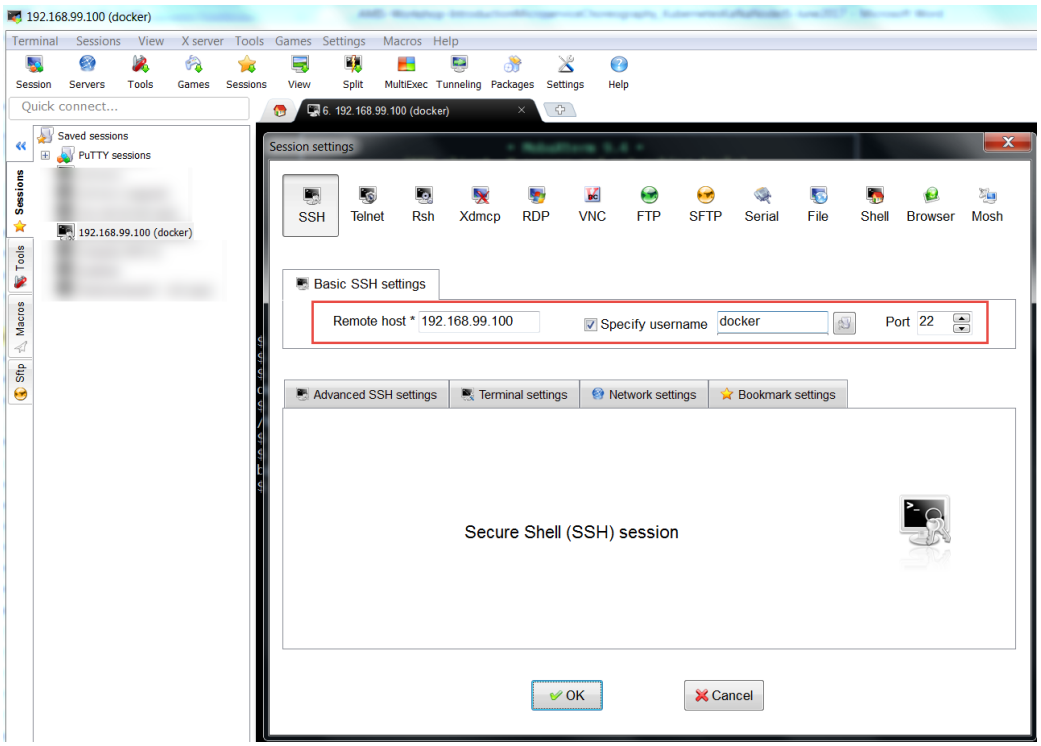
To open a shell in a running container in a pod with multiple containers:

```
kubectl exec –it <name of pod> -c <name of 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

```
$ docker ps
CONTAINER ID        IMAGE                                  COMMAND                  CREATED             STATUS              POR
TS                  NAMES
acb0ffd5e64b        9c5af559228c                           "/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        9d0c4eabab4d                           "/entrypoint.sh /etc/"   3 minutes ago       Up 3 minutes
                    k8s_registry_kube-registry-v0-35hnv_kube-system_12ff239c-460c-11e7-9206-080027d2b34f_0
f34c0f67edf2        6bb8effa98f4                           "/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        6bb8effa98f4                           "/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        6bb8effa98f4                           "/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        6bb8effa98f4                           "/code/bootstrap.sh"     4 minutes ago       Up 4 minutes
                    k8s_tweetreceiver_tweetreceiver-ms_default_d4aaf2c4-45cd-11e7-ad0e-080027d2b34f_2
26129839f1dc        6bb8effa98f4                           "/code/bootstrap.sh"     4 minutes ago       Up 4 minutes
                    k8s_microservice-workflow-launcher_workflow-launcher-ms_default_ea081b01-45ff-11e7-ad0e-080027d2b34f_2
849e1e8f12f9        6bb8effa98f4                           "/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        93a43bfb39bf                           "/exechealthz '--cmd="   4 minutes ago       Up 4 minutes
                    k8s_healthz_kube-dns-v20-t2r34_kube-system_c94122ee-41de-11e7-b254-080027d2b34f_4
81fcaae0a231        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
8fe80b44fd70        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_d4aaf2c4-45cd-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
```

Optionally, you could save images from local docker server to an archive file using these commands (note: there is no need to do so now for the workshop):

```
docker save -o /home/images-archive.tar image1 image2 lucasjellema/node-appp-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 in Docker and in Kubernetes. Some resources are listed below. I was most happy with the with the Kubernetes YAML files found in https://github.com/Yolean/kubernetes-kafka. You will make use of this repo to get a Kafka Cluster up and running.

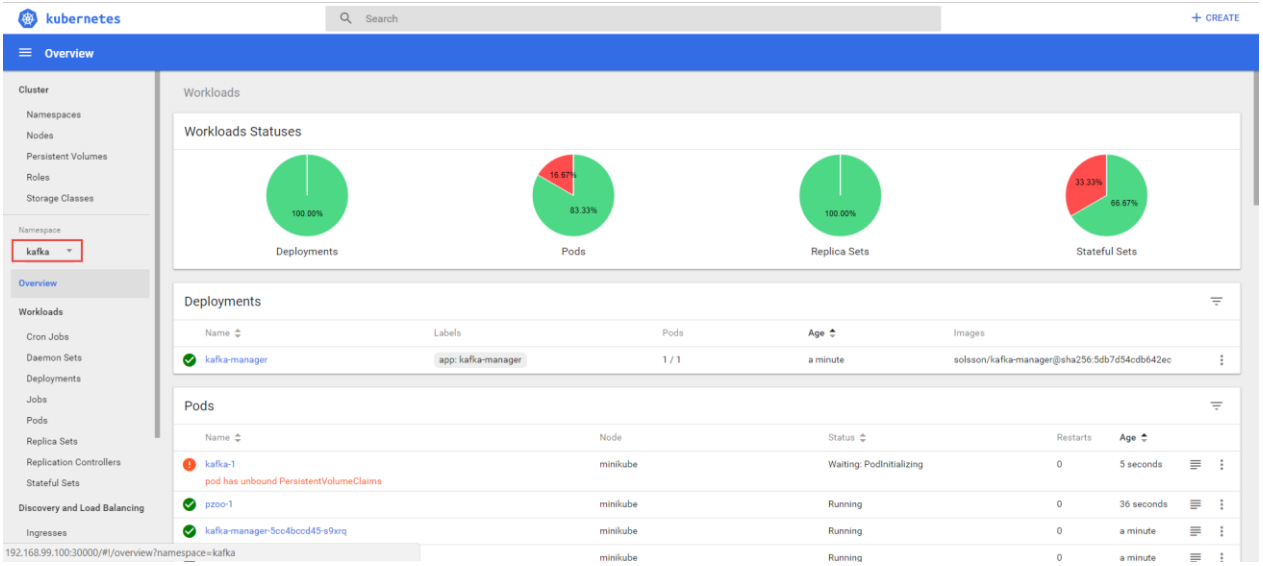
A very interesting alternative is to work with Kafka in the Cloud. Various providers offer a Kafka PaaS service – for example IBM BlueMix and Oracle Event Hub. An interesting option is CloudKarafka which offers a free tier and which is very easy to get going with. See https://www.cloudkarafka.com/plans.html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

Getting started with the Kafka on Kubernetes

Follow the instructions in https://technology.amis.nl/2018/04/19/15-minutes-to-get-a-kafka-cluster-running-on-kubernetes-and-start-producing-and-consuming-from-a-node-application/ for getting a Kafka Cluster running on your Kubernetes cluster.

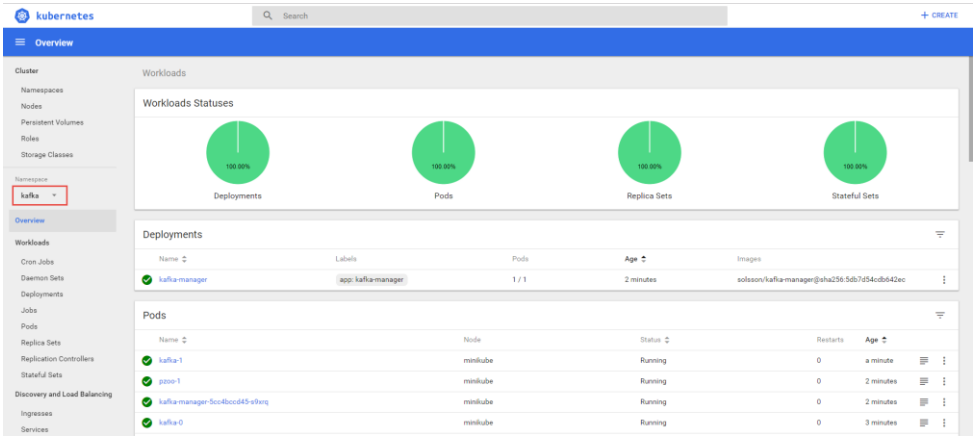
```
C:\temp\kubernetes-kafka>kubectl apply -f ./configure/minikube-storageclass-broker.yml
storageclass "kafka-broker" created

C:\temp\kubernetes-kafka>kubectl apply -f ./configure/minikube-storageclass-zookeeper.yml
storageclass "kafka-zookeeper" created

C:\temp\kubernetes-kafka>kubectl apply -f ./zookeeper
namespace "kafka" created
configmap "zookeeper-config" created
service "pzoo" created
service "zoo" created
service "zookeeper" created
statefulset "pzoo" created
statefulset "zoo" created

C:\temp\kubernetes-kafka>kubectl apply -f ./kafka
namespace "kafka" configured
configmap "broker-config" created
service "broker" created
service "bootstrap" created
statefulset "kafka" created

C:\temp\kubernetes-kafka>kubectl apply -f ./outside-services
service "outside-0" created
service "outside-1" created
service "outside-2" created

C:\temp\kubernetes-kafka>kubectl apply -f ./yahoo-kafka-manager
service "kafka-manager" created
deployment "kafka-manager" created
```

In the Kubernetes dashboard, switch to the Kafka namespace and look what has been created:



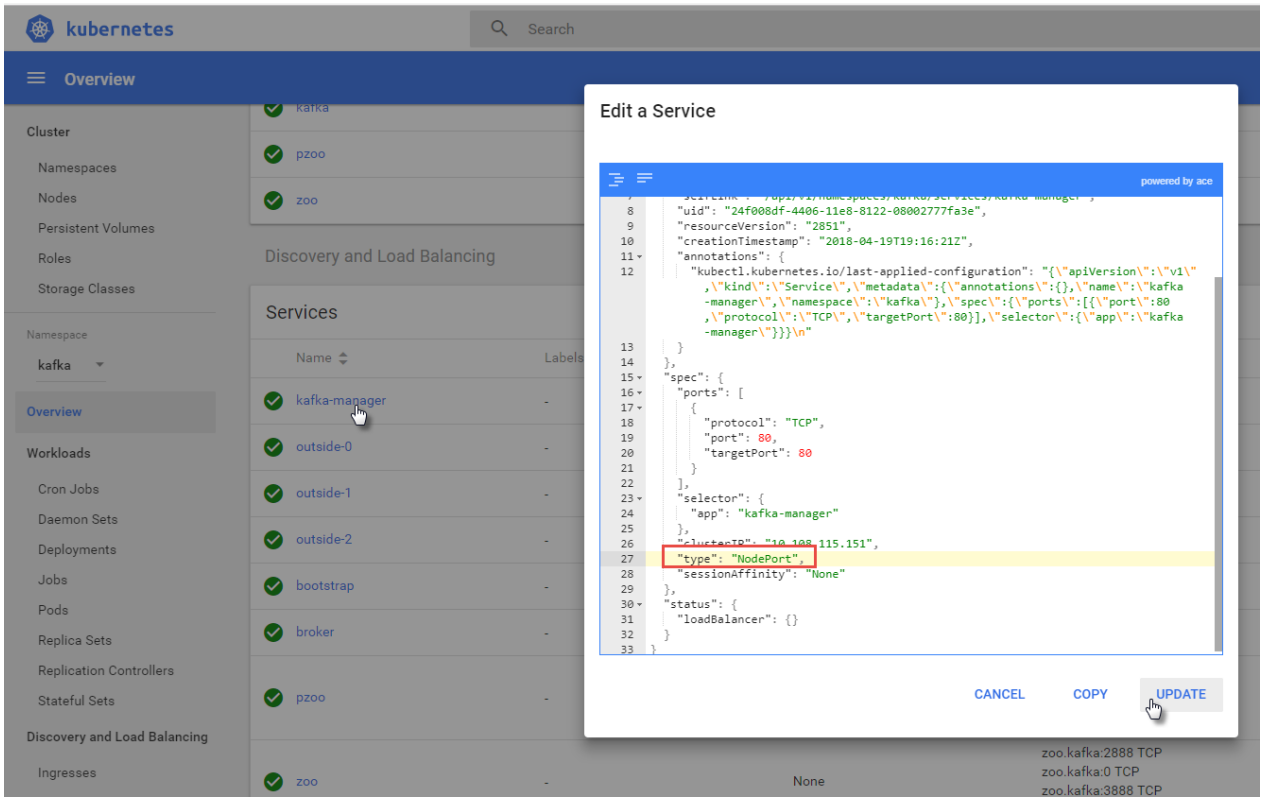
Pods were created as well as services, deployments and stateful sets. Zookeeper has been installed as well as two Kafka Broker Pods. Together these form the Kafka Cluster – a pretty powerful event hub.

Note: After a minute or two, what was red should become green as well

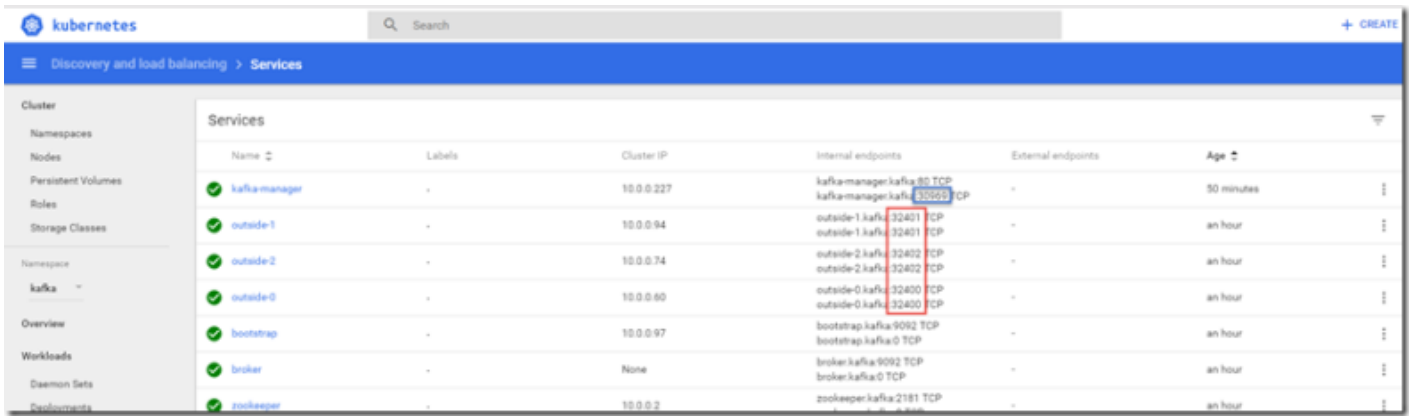


At this point, the Kafka Cluster is running. I can check the pods and services in the Kubernetes Dashboard as well as through kubectl on the command line.

In order to be able to access the Kafka Manager browser UI, you have to update the *type* attribute in the *spec* section of the kafka-manager service, from ClusterIP to NodePort – see below.

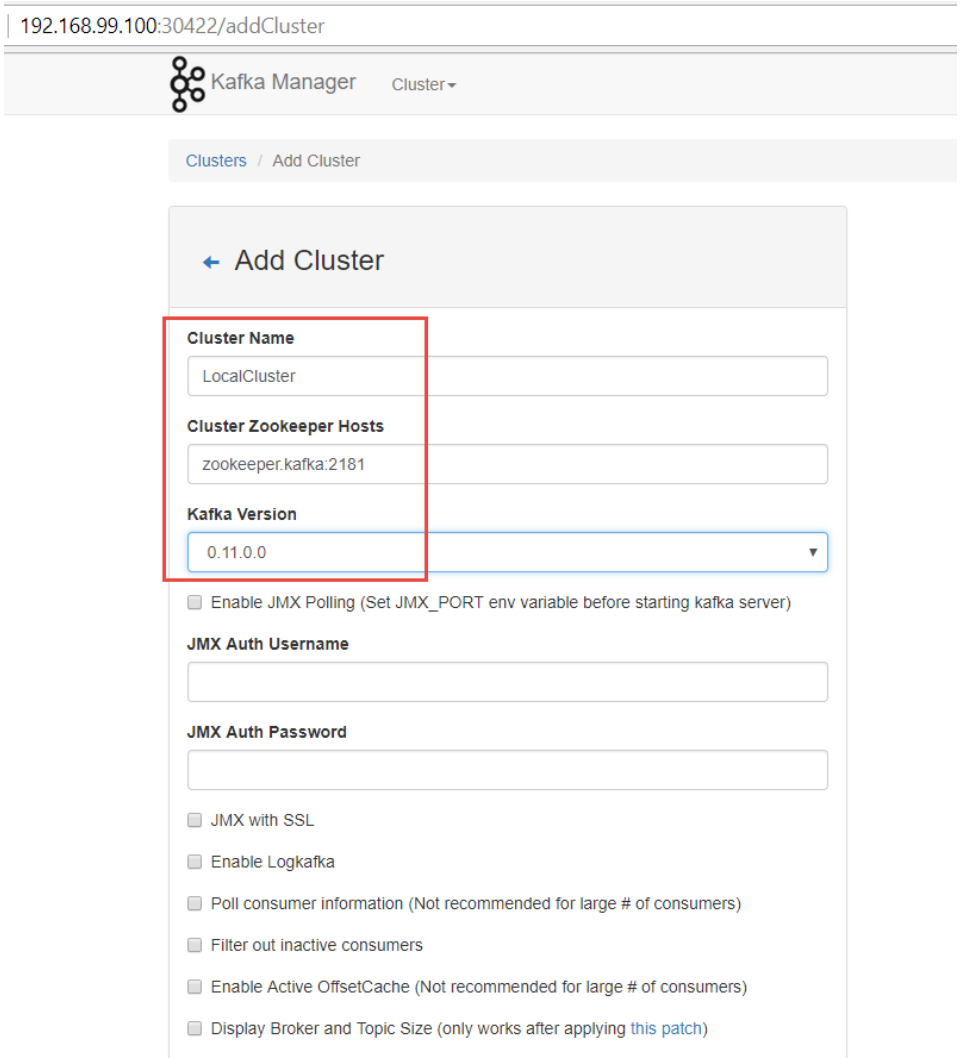


When the update is processed I can now get the Port at which I can access the Kafka Brokers as well as the Kafka-Manager:



I can access the Kafka Manager at the indicated Port.

Provide the indicated information:



And press Save. Now you can use Kafka Manager to manage and monitor the Kafka Cluster.

Finally the eating of the pudding: production and consumption of messages to and from the cluster.

Get a shell in one of the Kafka Broker pods:

```
kubectl exec -it kafka-0 --namespace kafka -- bash
```

Create a new Kafka Topic called event-bus:

```
unset JMX_PORT;bin/kafka-topics.sh --create --zookeeper
zookeeper.kafka:2181 --replication-factor 1 --partitions 1 --topic
event-bus
```

Confirm the topic is created:

```
unset JMX_PORT;bin/kafka-topics.sh --list --zookeeper
zookeeper.kafka:2181
```

Now to produce a message:

```
unset JMX_PORT;bin/kafka-console-producer.sh \

    --broker-list broker.kafka:9092 \

    --topic event-bus
```

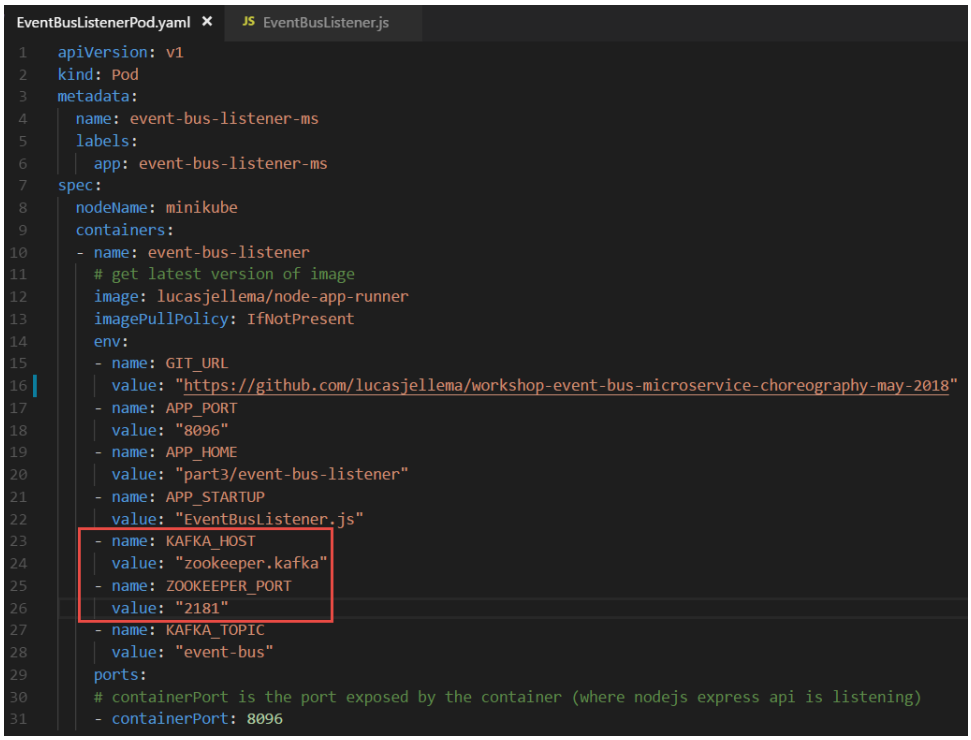
This will open a command line terminal where each line of text that you enter is sent as a message to topic event-bus. Close this terminal with ctrl+c.

Now to read the messages from the topic:

```
unset JMX_PORT;bin/kafka-console-consumer.sh \

--zookeeper zookeeper.kafka:2181    --topic event-bus    --from-
beginning
```

Run a Microservice with Kafka Interaction

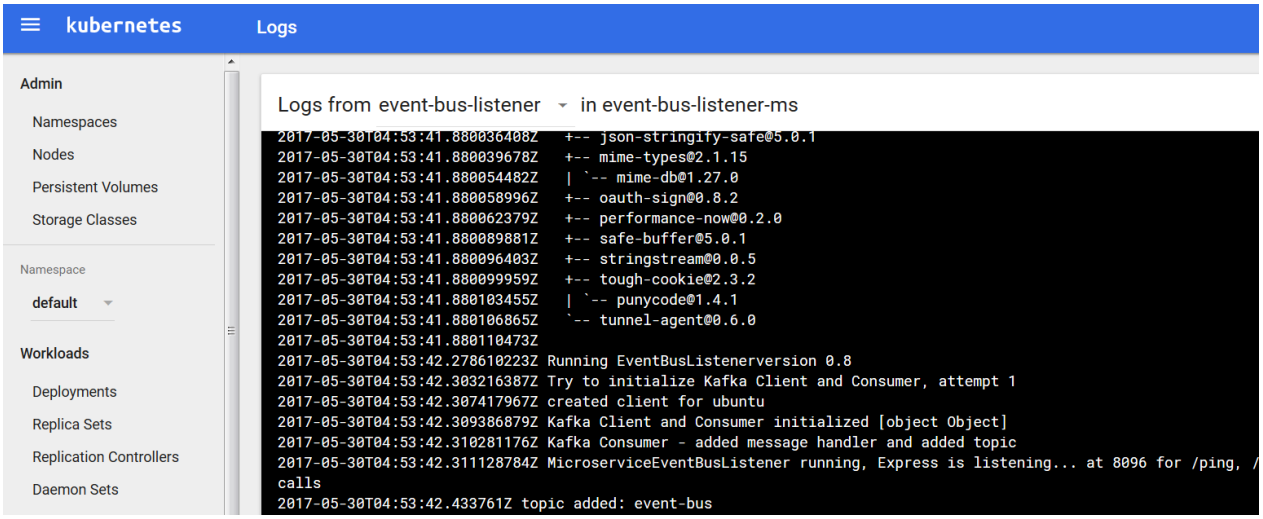
Run a Pod on Kubernetes that starts a Node.JS application that listens to the Event Bus topic on the Kafka Broker. You do this using the EventBusListener.yaml file in directory part3/event-bus-listener. Before you run the yaml file, check out its contents. Take note of the ports, the environment variables such as KAFKA_TOPIC (set to event-bus) and KAFKA_HOST, set to zookeeper.kafka.



Now 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:

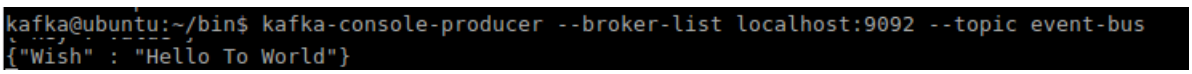


Then, publish an event to topic event-bus from the command line in the shell into the Kafka Broker:

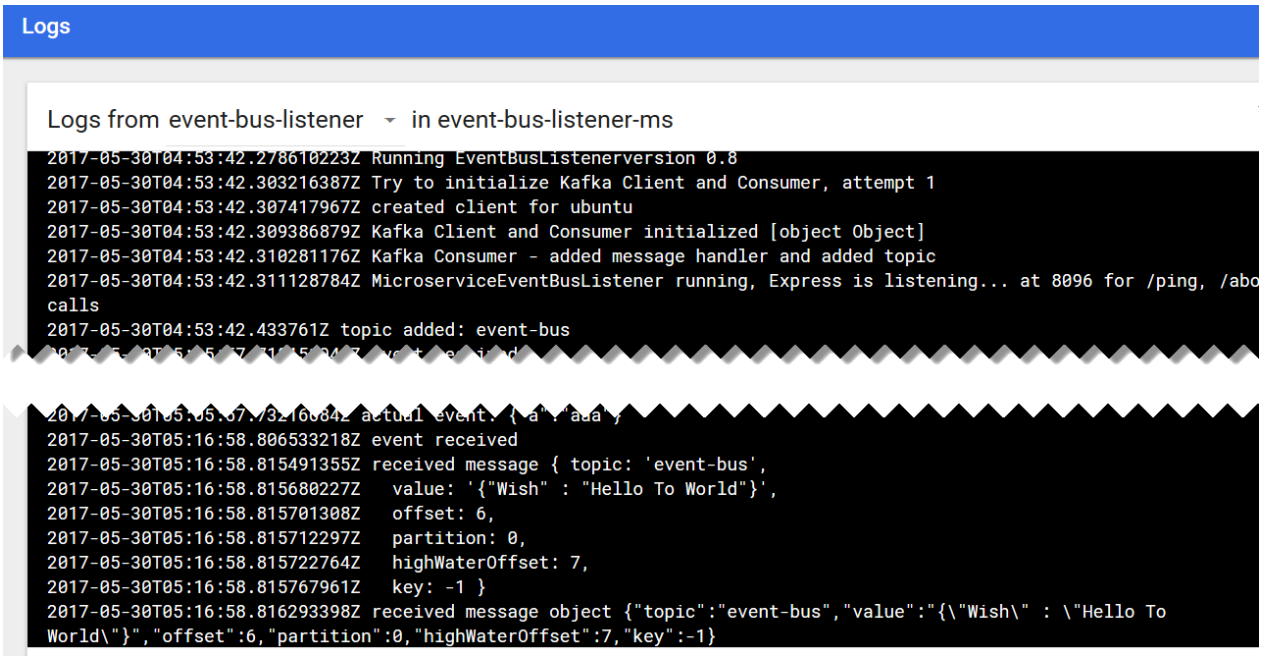
```
unset JMX_PORT;bin/kafka-console-producer.sh \
    --broker-list broker.kafka:9092 \
    --topic event-bus
```

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"}
```



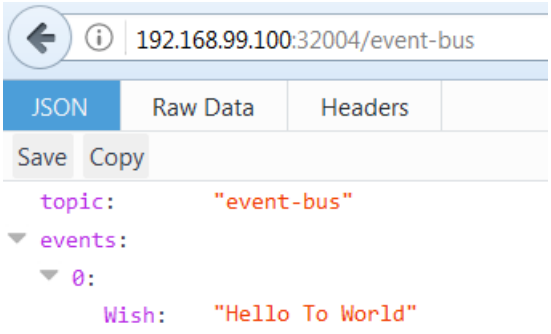
If you check the logs in the Kubernetes Dashboard, you should find that the event has been received:



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:



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.

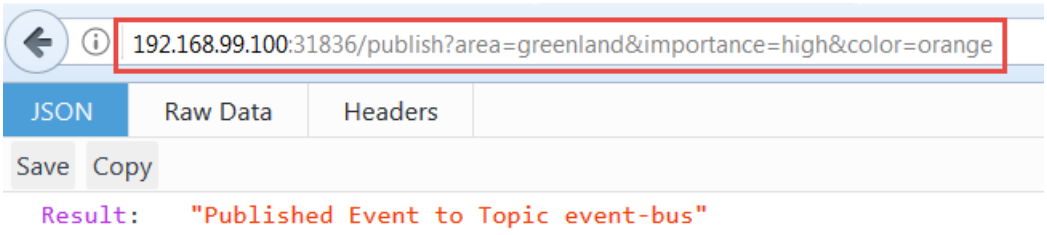
```
kubectl create -f EventBusPublisherPod.yaml  -f
EventBusPublisherService.yaml
```

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*.

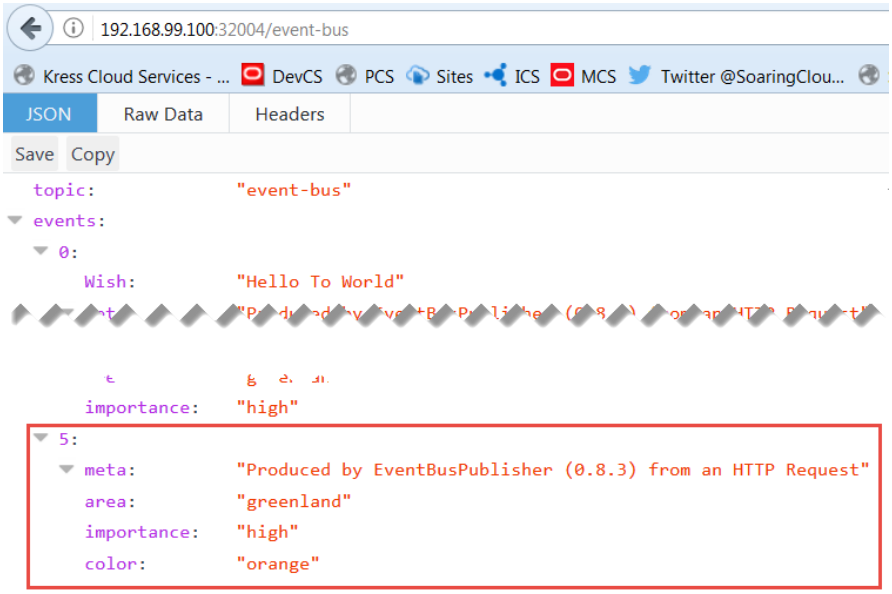
```
c:\data\microservices-choreography-kubernetes-workshop-june2017\part3\event-bus-publisher>kubectl get services
NAME                        CLUSTER-IP   EXTERNAL-IP   PORT(S)                                        AGE
eventbuslistenerservice     10.0.0.249   <nodes>       8096:32004/TCP          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

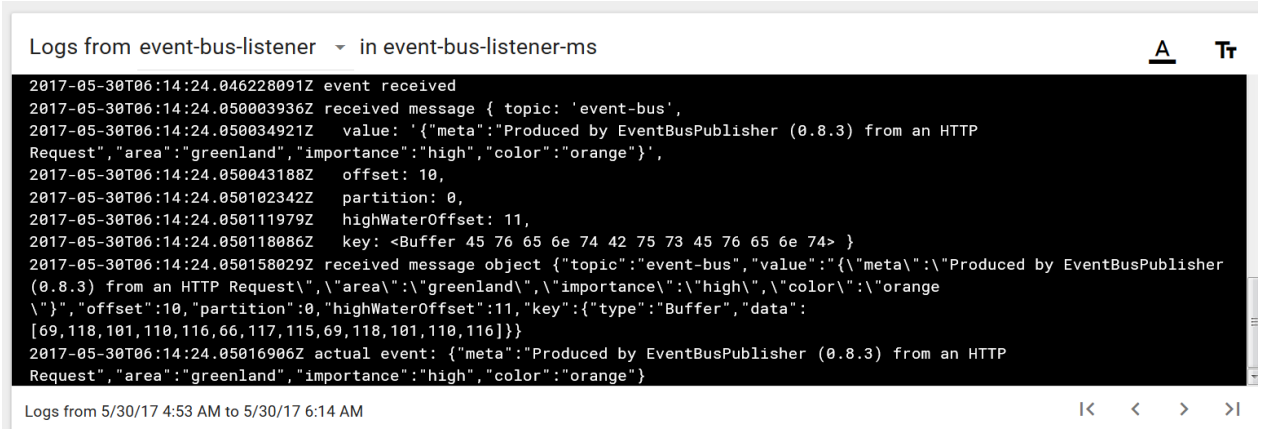
With http requests from your browser - or using CURL or Postman – you can trigger the Event Bus Publisher microservice into publishing events to the Event Bus:

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



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.





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 in the *default* namespace; we like to keep he *kafka* namespace artefacts that are running just fine. A quick way to tear all components down is:

```
kubectl delete po,svc,rc,deploy --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 microservice 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, still running on your Kubernetes cluster. In this Kafka Cluster, create two new topics (in a shell into the Kafka Broker Container):

Get a shell in one of the Kafka Broker pods:

```
kubectl exec -it kafka-0 --namespace kafka -- bash
```

Create two new Kafka Topics called workflowEvents and logTopic respectively:

```
unset JMX_PORT;bin/kafka-topics.sh --create --zookeeper
zookeeper.kafka:2181 --replication-factor 1 --partitions 1 --topic
workflowEvents

unset JMX_PORT;bin/kafka-topics.sh --create --zookeeper
zookeeper.kafka:2181 --replication-factor 1 --partitions 1 --topic
logTopic
```

Confirm the topics have been created:

```
unset JMX_PORT;bin/kafka-topics.sh --list --zookeeper
zookeeper.kafka:2181
```

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 to only allow access to other microservices (pods) on the cluster.

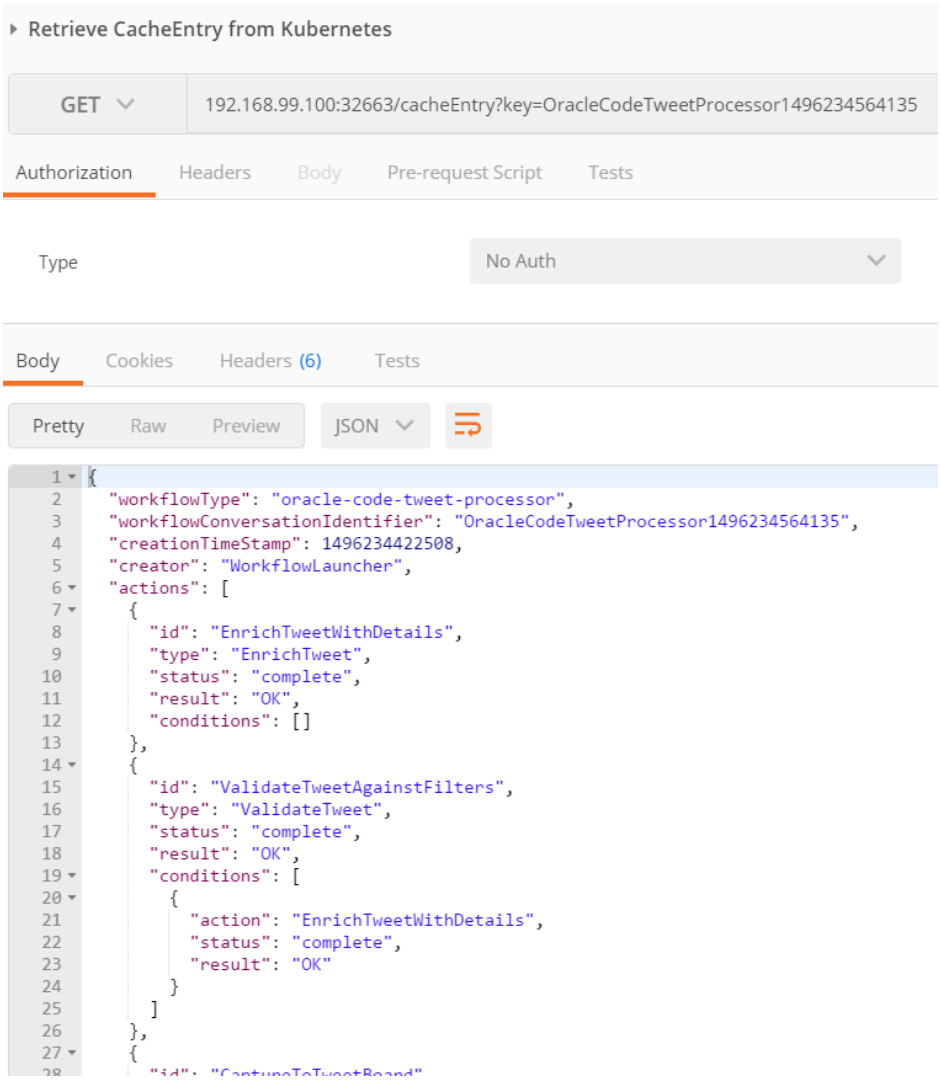
Inspect the cache contents

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 will be able a little bit later on to 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. Note: there are no workflow instances yet.



Run the LogMonitor to inspect the logging produced in microservices

Being able to keep track of what is going on inside the microservices – on a technical, non functional level and on a functional level – is pretty important in order to detect malfunctions and analyze and ultimately resolve any issues. The microservices you will deploy today all produce logging – to a Kafka logTopic. The LogMonitor consumes the log events on this topic and exposes them through a simple web UI. Let's install the LogMonitor now.

In directory part4/LogMonitor, run:

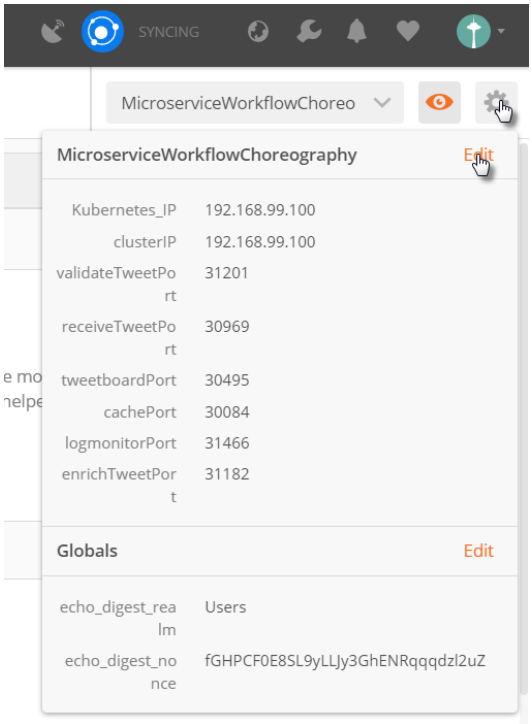
```
kubectl create -f LogMonitorPod.yaml  -f LogMonitorService.yaml
```

With this command, a new Pod is launched that consumes the log events on the logTopic and keeps them in memory to make them available to anyone requesting them through a simple HTTP request:

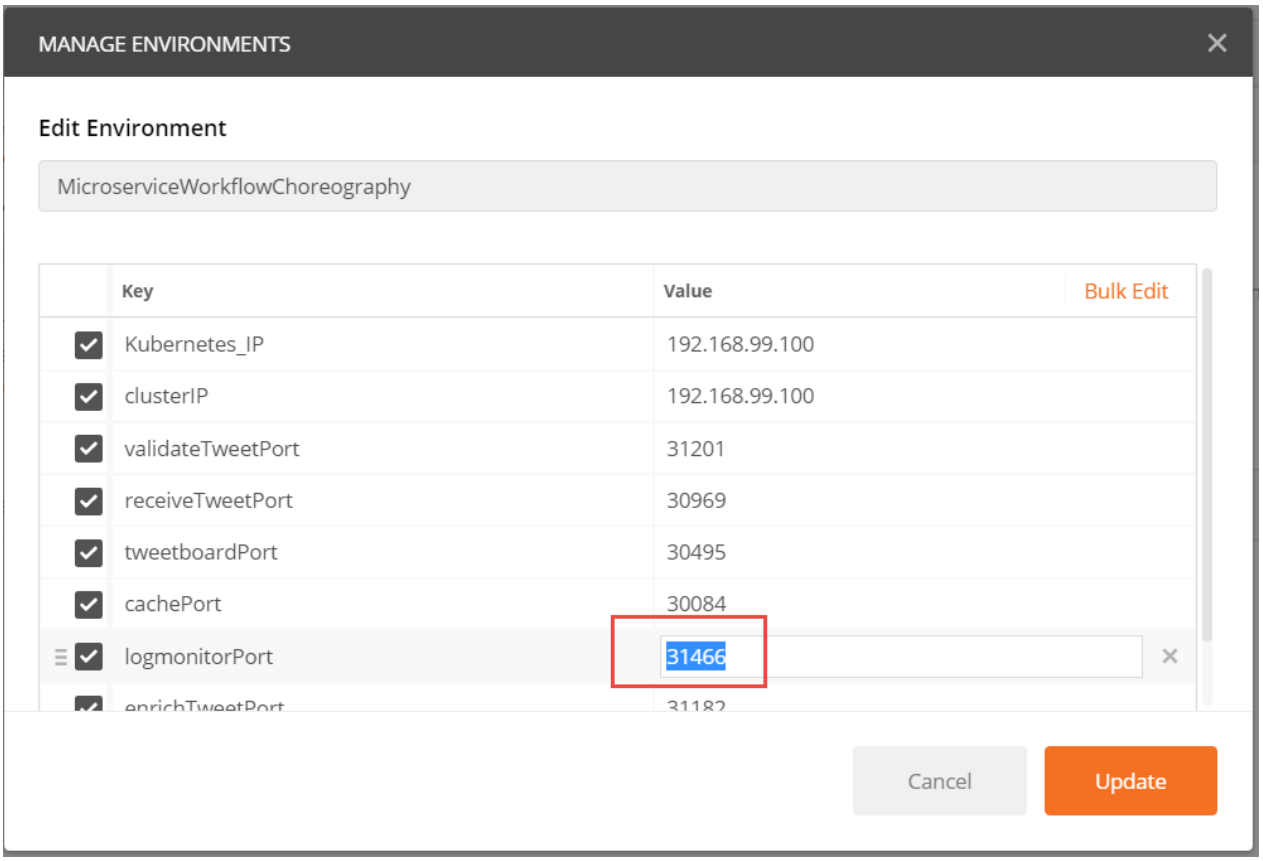
http://kubernetesIP:logMonitorPort/logs

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

Open Postman and edit the environment variables:

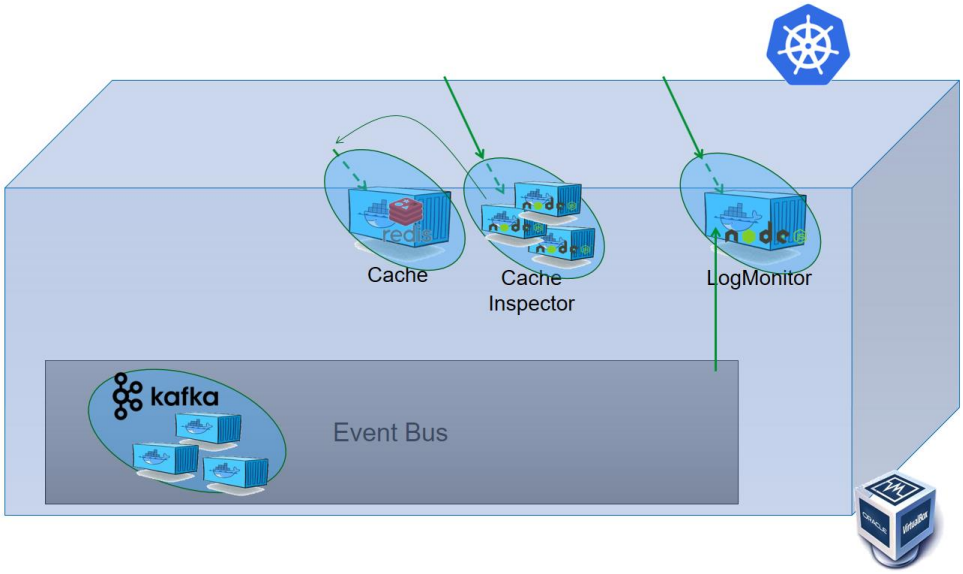


Update the logMonitorPort variable to the value retrieved using *kubectl get services*.



Then press Update to save the configuration changes.

We have initialized a few key pieces of the microservices runtime platform:



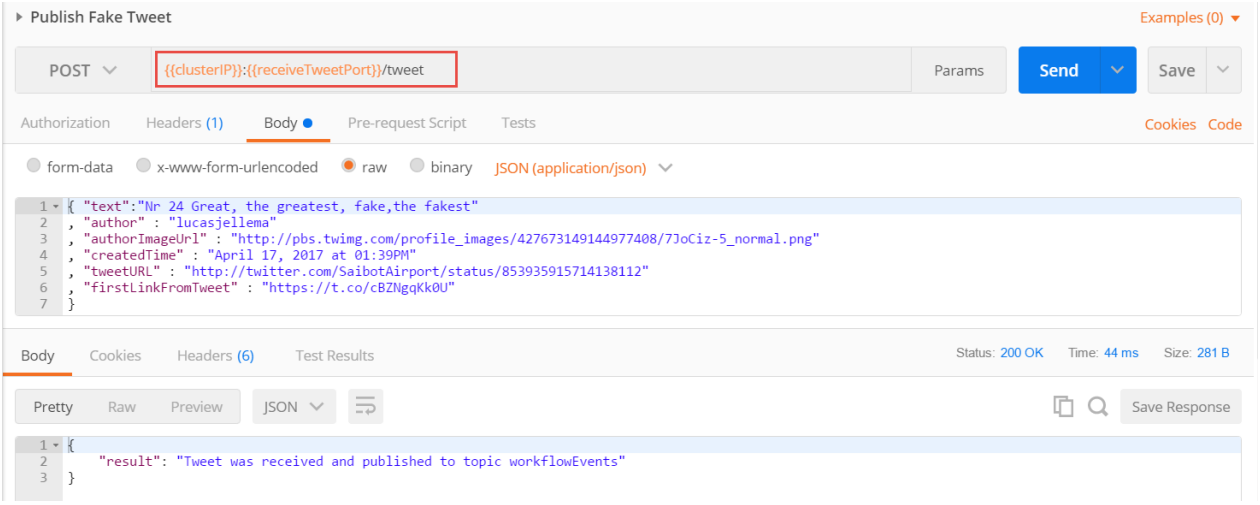
Run Microservice TweetReceiver

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 reporting could be done through a recipe from IFTTT or with a simple call from any HTTP client such as Postman.



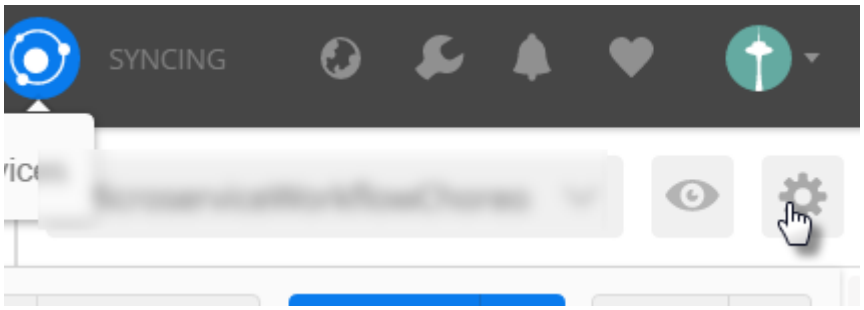
Using *kubectl get services* you can inspect the service that is created and the port at which it is exposed.

Run Postman, Load Test Collection, Create Environment plus Variables and Test TweetReceiver

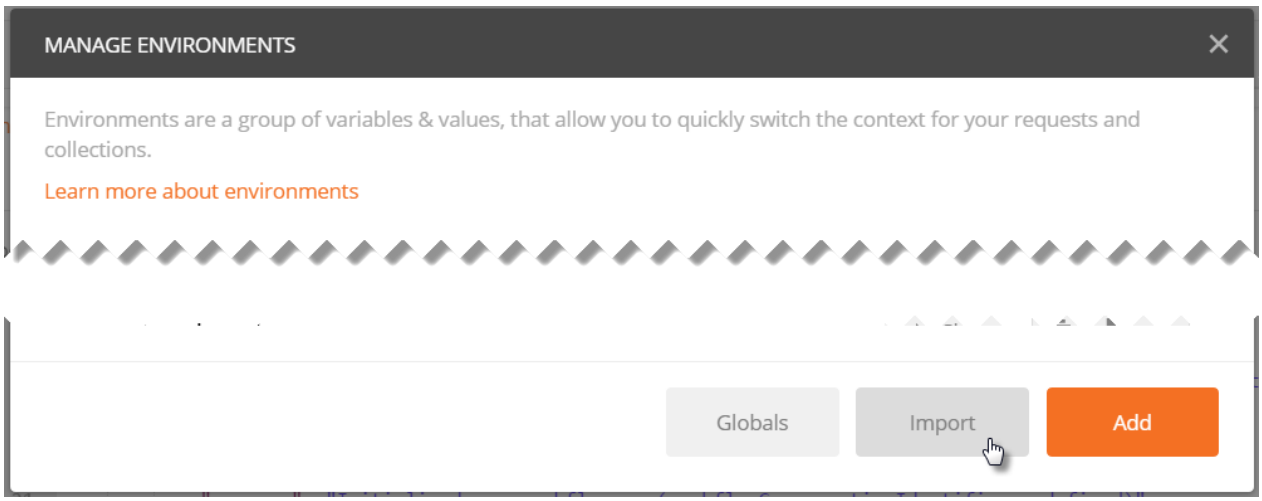
Run Postman.

Import the Postman Collection MicroServiceTestSet.postman_collection.json from directory part4 in the workshop resources. This will create the collection MicroServiceTestSet with a number of test requests to the TweetReceiver and other microservices.

Now click on the *manage environments* icon in the upper right hand corner:



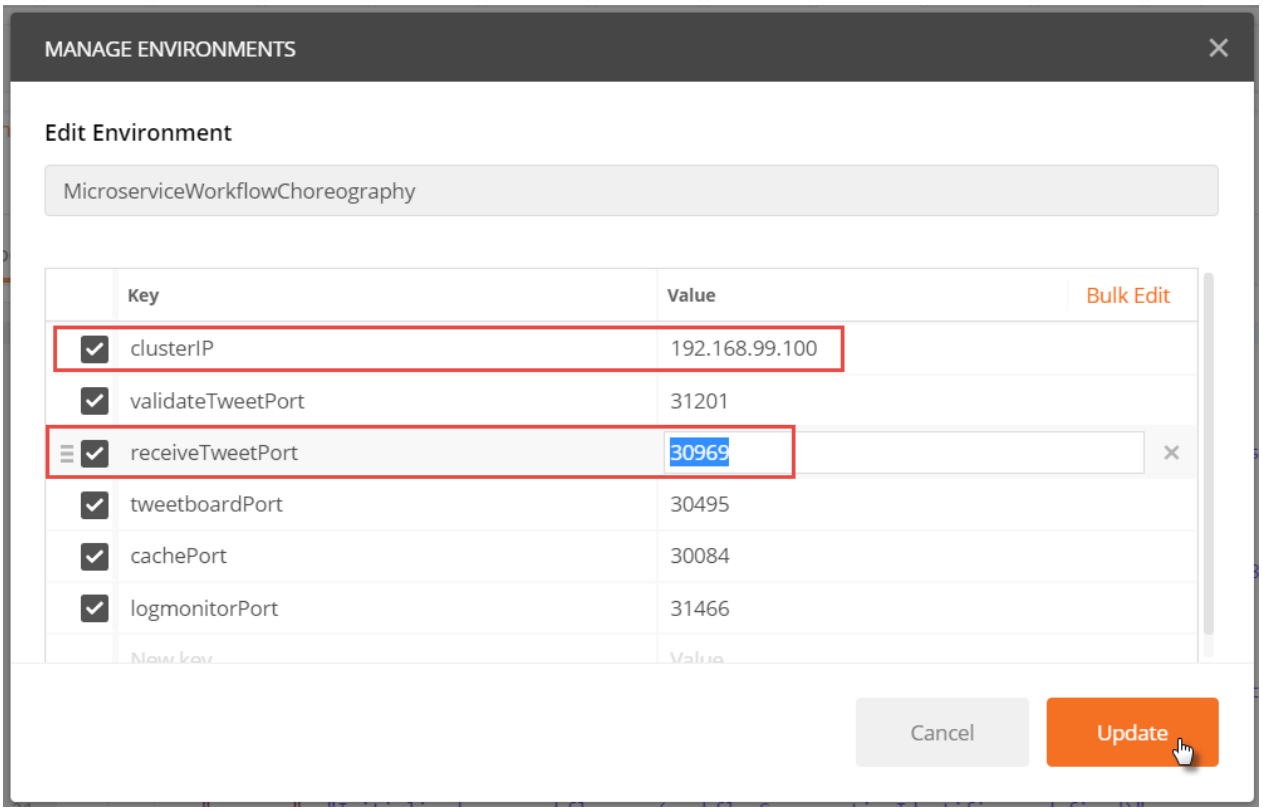
A popup window opens. Click on Import:



And load the file MicroserviceWorkflowChoreography.postman_environment.json. This file contains the environment MicroserviceWorkflowChoreography, a set of environment variables used in the Postman Collection. These variables define the IP address of the Kubernetes cluster – the host for all microservices – as well as the port for each of the microservices that you install.

Open the environment, and edit the value for both the clusterIP variable and the receiveTweetPort. The first can be retrieved using *minikube ip* and the second with *kubectl get services*.

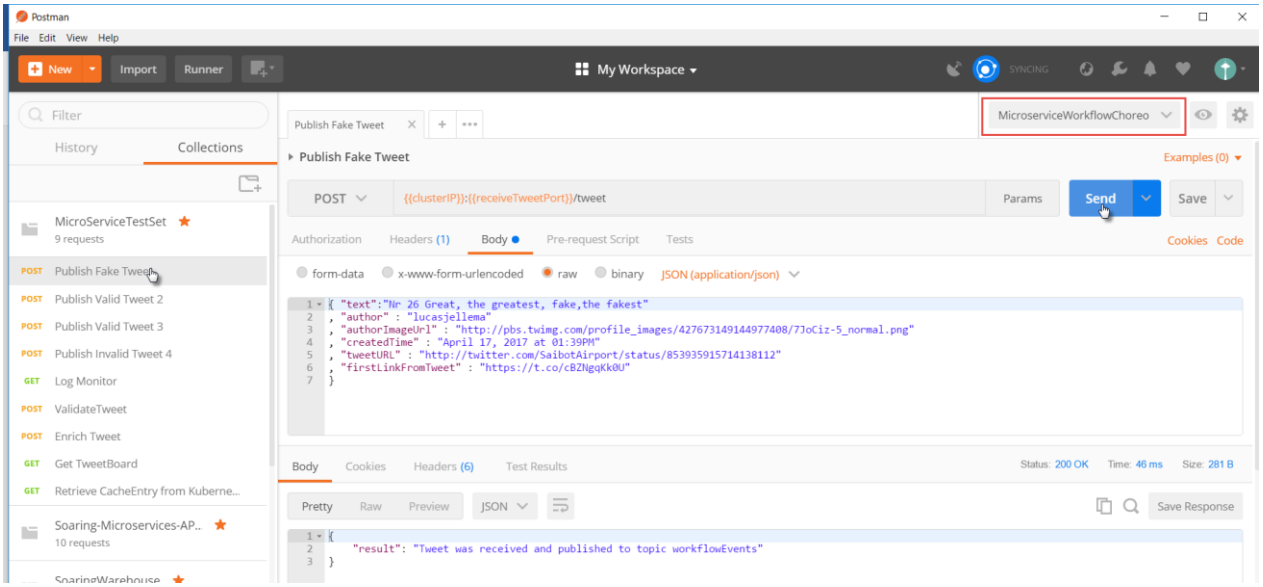
```
c:\data\1-FontysHogeschool-9may-2018\workshop-event-bus-microservice-choreography\part4\TweetReceiver>minikube ip
192.168.99.100

c:\data\1-FontysHogeschool-9may-2018\workshop-event-bus-microservice-choreography\part4\TweetReceiver>kubectl get services
NAME                   TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
kubernetes             ClusterIP   10.96.0.1        <none>        443/TCP          1d
tweetreceiverservice   NodePort    10.105.110.215   <none>        8101:30969/TCP   1d
```



For now, just ignore all the other variables. Press Update and close the popup.

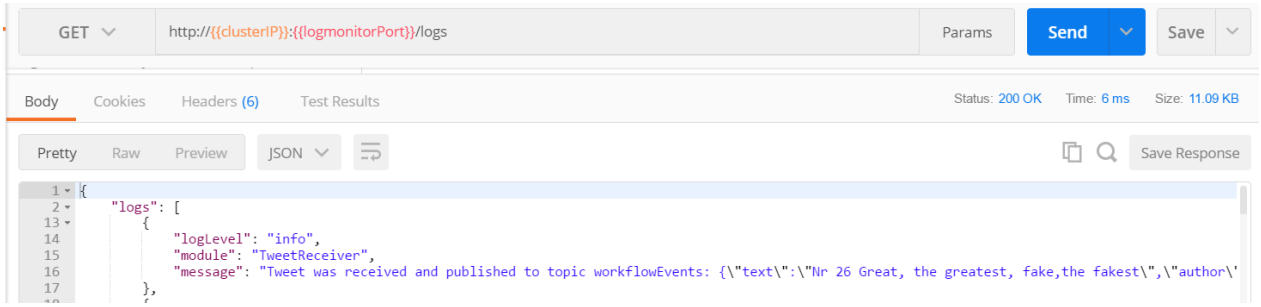
Ensure that the currently active environment is indeed *MicroserviceWorkflowChoreography*.

Open the request called Publish Fake Tweet. Inspect the request – the URL and the Body. Then press Send. A response should be visible momentarily.



This proves that the microservice TweetReceiver is available – up and running.

Open the request LogMonitor and press Send.



You should now see a log entry for each request to the TweetReceiver microservice API.

Run Microservice Workflow Launcher

The Workflow Launcher listens to the *workflowEvents* 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:

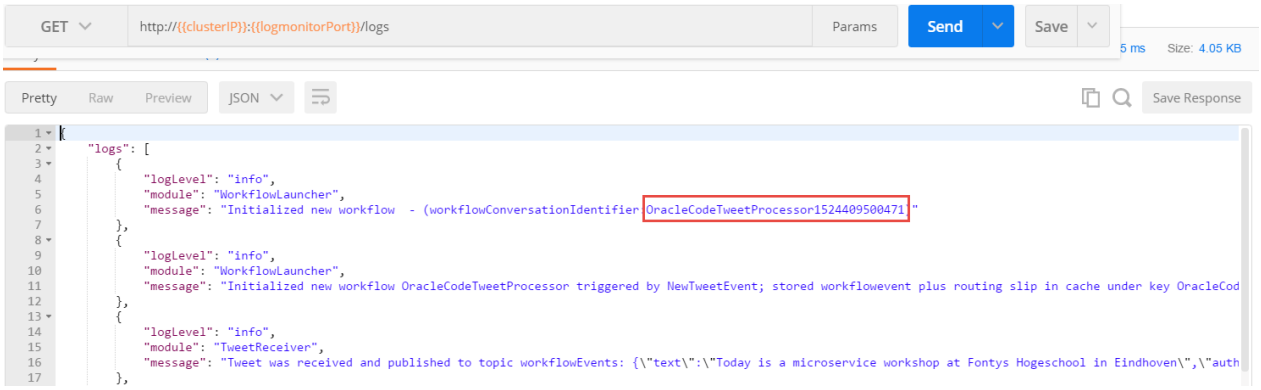
```
kubectl create -f WorkflowLauncherPod.yaml
```

Note: the values for the environment variable KAFKA_HOST, ZOOKEEPER_PORT and REDIS_HOST and REDIS_PORT in the yaml file link this microservice to some of its dependencies.

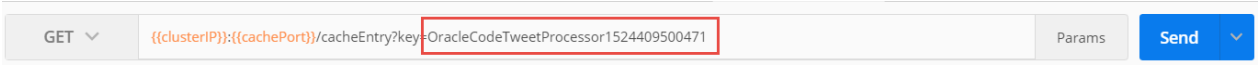
This microservices does not expose an external service – all it does is listen to events on the Kafka Topic [and turn a NewTweetEvent into a Workflow Event with a routing slip for the workflow that has to be executed]. This routing slip is defined in the *message* variable at the bottom of the file WorkflowLauncher.js.

```
message =
  {
   "workflowType": "oracle-code-tweet-processor"
   , "workflowConversationIdentifier": "oracle-code-tweet-processor" + new Date().getTime()
   , "creationTimeStamp": new Date().getTime()
   , "creator": "WorkflowLauncher"
   , "actions":
   [{
    "id": "ValidateTweetAgainstFilters"
    , "type": "ValidateTweet"
    , "status": "new"  // new, inprogress, complete, failed
    , "result": "" // for example OK, 0, 42, true
    , "conditions": [] // a condition can be {"action":"<id of a step in the routingslip>", "status":"complete"
   }
    , {
    "id": "EnrichTweetWithDetails"
    , "type": "EnrichTweet"
    , "status": "new"  // new, inprogress, complete, failed
    , "result": "" // for example OK, 0, 42, true
    , "conditions": [{ "action": "ValidateTweetAgainstFilters", "status": "complete", "result": "OK" }]
   }
    , {
    "id": "CaptureToTweetBoard"
    , "type": "TweetBoardCapture"
    , "status": "new"  // new, inprogress, complete, failed
    , "result": "" // for example OK, 0, 42, true
    , "conditions": [{ "action": "EnrichTweetWithDetails", "status": "complete", "result": "OK" }]
   }
   ]
```

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



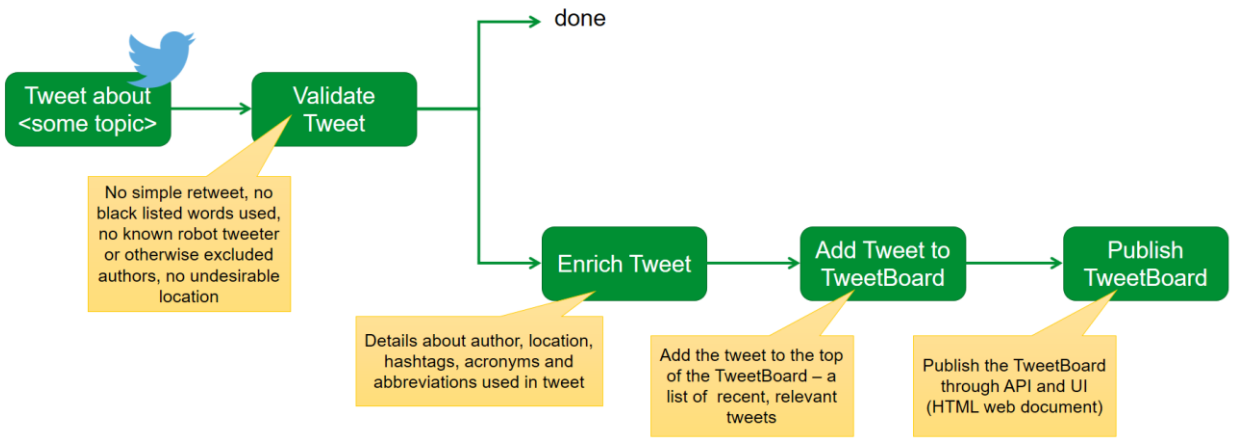
The WorkflowLauncher creates a workflow instance for which we can read the identifier in the LogMonitor output. Using this identifier, we can retrieve the routing slip from the CacheInspector service:



The result - with the activities that should be performed and the condition that apply to each activity:

```
{
    "workflowType": "oracle-code-tweet-processor",
    "workflowConversationIdentifier": "OracleCodeTweetProcessor1524409500471",
    "creationTimeStamp": 1524409335790,
    "creator": "WorkflowLauncher",
    "actions": [
        {
            "id": "ValidateTweetAgainstFilters",
            "type": "ValidateTweet",
            "status": "new",
            "result": "",
            "conditions": []
        },
        {
            "id": "EnrichTweetWithDetails",
            "type": "EnrichTweet",
            "status": "new",
            "result": "",
            "conditions": [
                {
                    "action": "ValidateTweetAgainstFilters",
                    "status": "complete",
                    "result": "OK"
                }
            ]
        },
        {
            "id": "CaptureToTweetBoard",
            "type": "TweetBoardCapture",
            "status": "new",
            "result": "",
            "conditions": [
                {
                    "action": "EnrichTweetWithDetails",
                    "status": "complete",
                    "result": "OK"
                }
            ]
        }
    ],
    "audit": [
        {
            "when": 1524409335790,
            "who": "WorkflowLauncher",
            "what": "creation",
            "comment": "initial creation of workflow"
        }
    ],
    "payload": {
        "text": "Today is a microservice workshop at Fontys Hogeschool in Eindhoven",
        "author": "lucasjellema",
        "authorImageUrl": "http://pbs.twimg.com/profile_images/427673149144977408/7JoCiz
        "createdTime": "May 9th, 2018 at 08:39AM",
        "tweetURL": "http://twitter.com/SaibotAirport/status/853935915714138112",
        "firstLinkFromTweet": "https://t.co/cBZNgqKk0U"
    }
}
```


The intended workflow is the following:



Can you see how the routing slip created by the Workflow Launcher microservice describes this workflow – especially the individual activities – validate, enrich, add to tweet board – and the proper sequence?

Note that at this point the workflow instance that is created by the workflow launcher is not processed. There are no microservices available yet to process the activities described in the routing slip. That is about to change.

Rollout 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:

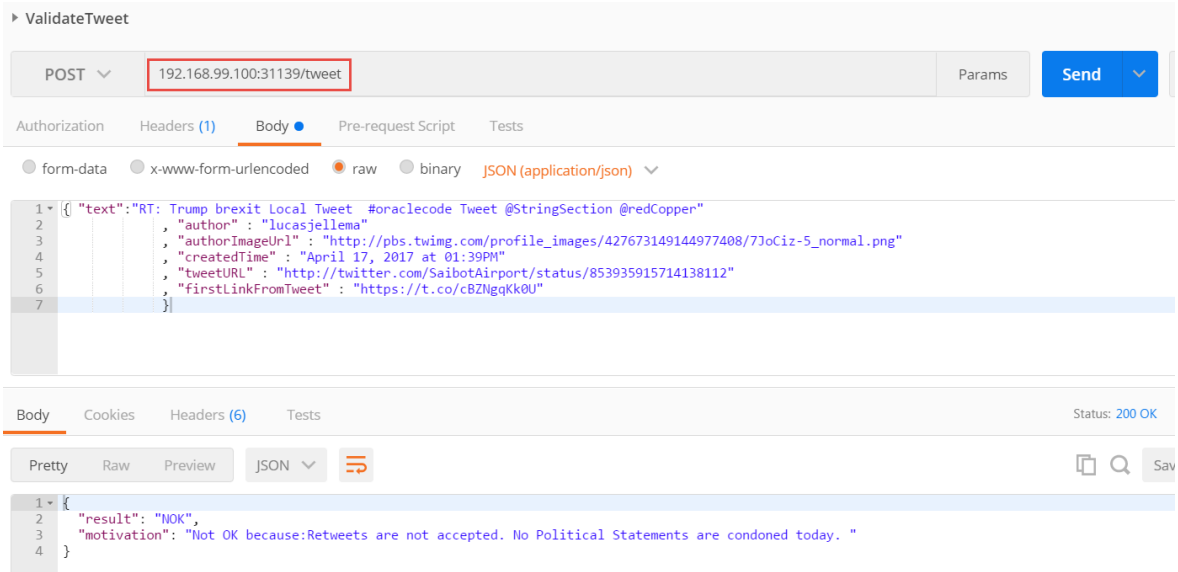
```
kubectl create -f ValidateTweetService.yaml
```

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 microservice is running:

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

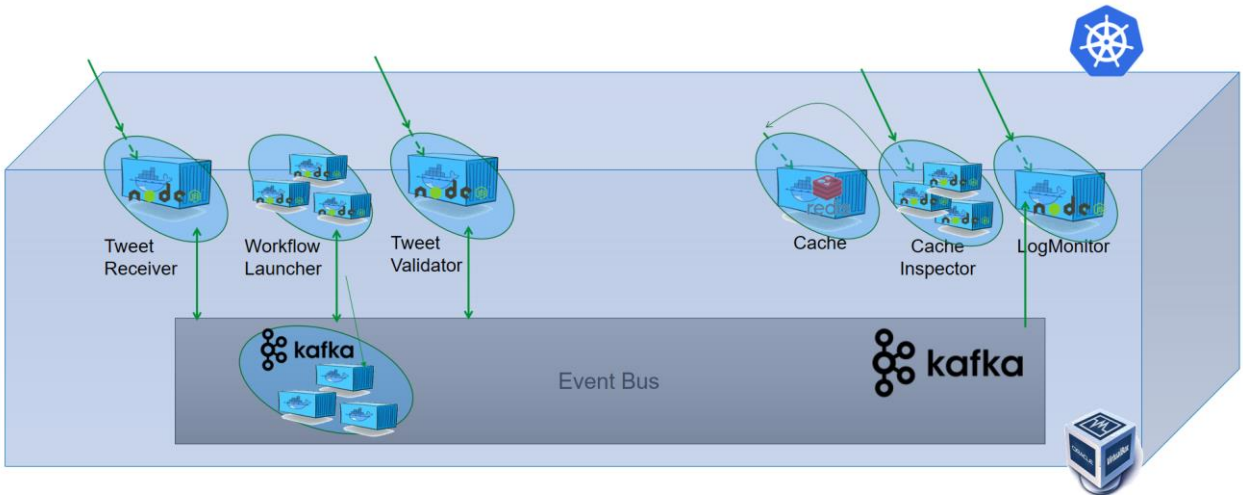
```
c:\data\microservices-choreography-kubernetes-workshop-june2017\part4\ValidateTweet>curl http://192.168.99.100:31139/about
About TweetValidator API, Version 0.8Supported URLs:/ping (GET)
;/tweet (POST)NodeJS runtime version v8.0.0incoming headers{"user-agent":"curl/7.30.0","host":"192.168.99.100:31139","accept":"*/*"}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

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 source code for this microservices is in file part4/ValidateTweet/TweetValidator.js. In this file, the function handleWorkflowEvent is invoked whenever an event is consumed from the workflowEvents topic. The logic in this function inspects the workflow event to find any action of the actionType that this microservice knows how to process – ValidateTweet – and with status *new*. If it finds such as action, it will check what the condition are that have to be fulfilled in order for this action to per performed (none at present). If an action of the right type with the right status and without unsatisfied conditions is found, the microservice will do its job and execute the action. Subsequently it will update the routing slip and publish a fresh event to the *workflowEvents* topic.

At this point, we have the following microservices platform – that still cannot complete the workflow but at least that can begin with the validation step in the workflows.



What we see in action now is one of the promises of the microservices choreography approach: without changing a central orchestration component – because there is none – or any other microservice, we have extended the capability of our application landscape with the ability to participate in the Tweet Workflow. Before too long we will have deployed to additional microservices and at that point the entire workflow can be completed. Without any impact on existing pieces and components.

After that, we will change the workflow definition – and without changing any of the microservices have the updated workflows executed by the existing microservices. We can also take down one or more of the microservices – and after a little while start the up again, with the same or a changed implementation. While they are down, the execution of the workflows ceases. But as soon as the microservice is started, it will go an consume all pending events on the *workflowEvents* topic and start processing them – if it can.

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
```

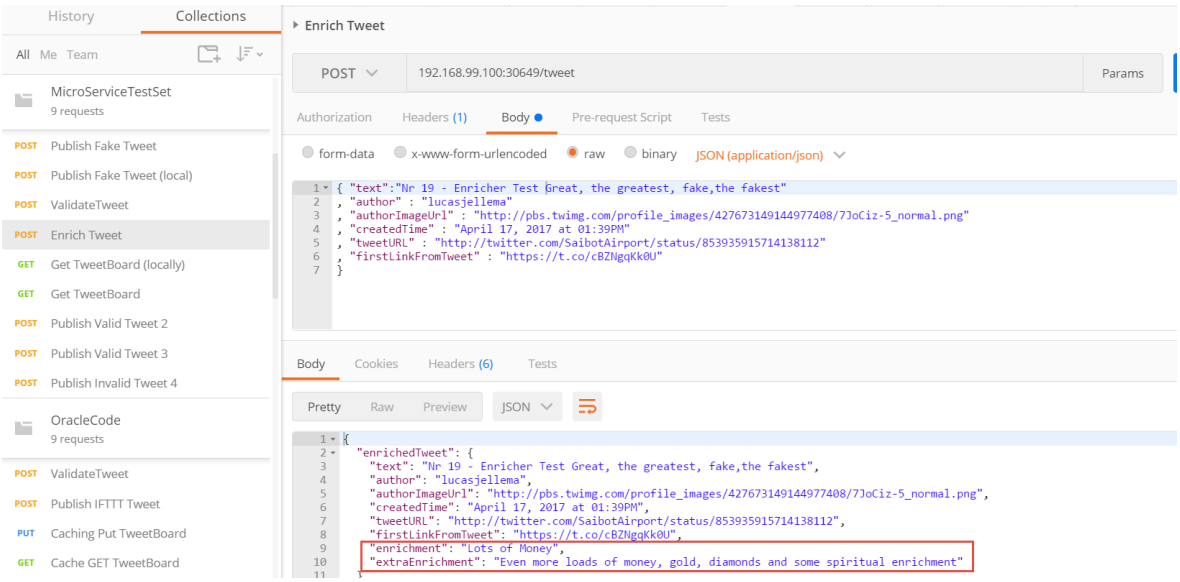
Also to expose an API for this microservice:

```
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 microservice 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:



This microservice looks for workflowEvents and searches actions of type EnrichTweet that are available for execution. If it finds such actions, it will execute them.

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:

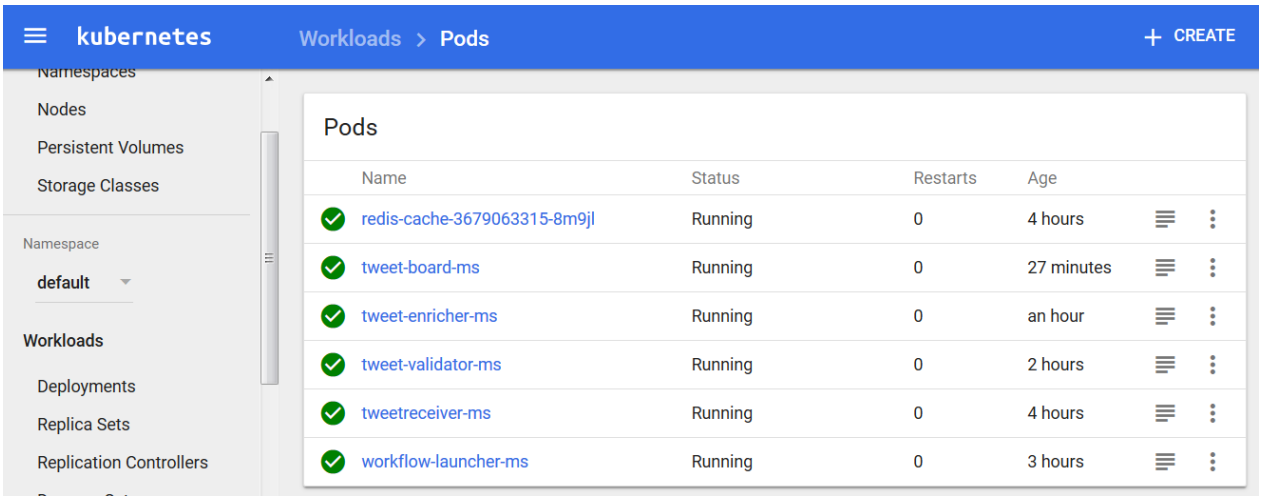
```
kubectl create -f TweetBoardPod.yaml
```

and expose the microservice as a Service:

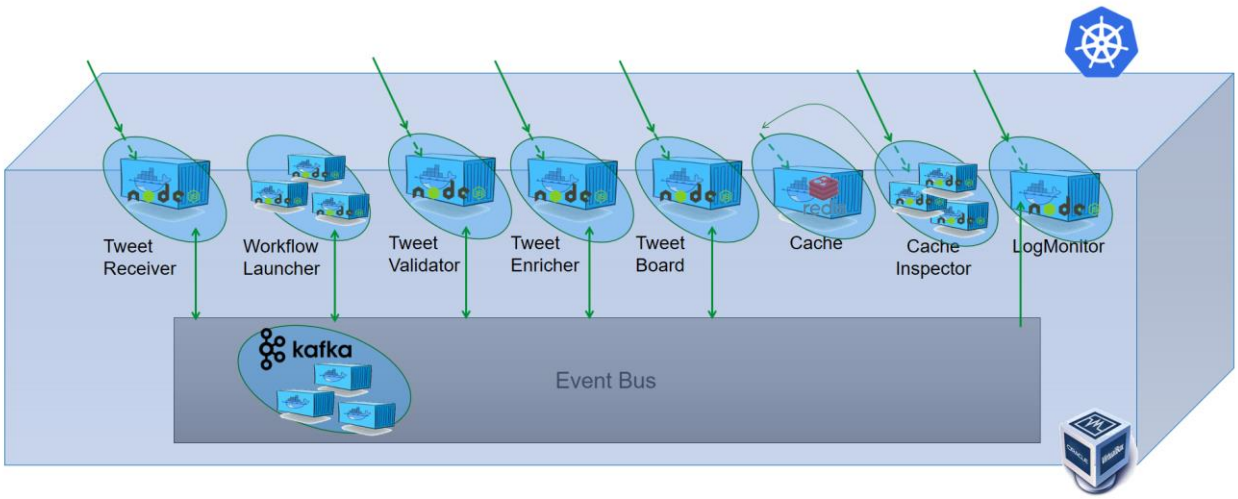
```
kubectl create -f TweetBoardService.yaml
```

This microservice looks for workflowEvents and searches actions of type *TweetBoardCapture* that are available for execution. If it finds such actions, it will execute them.

The Kubernetes Dashboard now lists at least the following Microservices in the *default* namespace :



This corresponds with this overview architecture diagram:

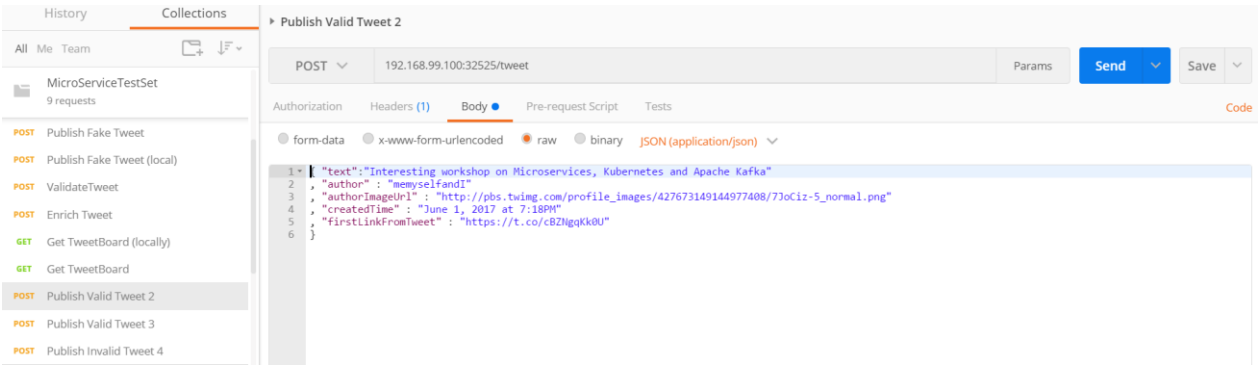


All actions in the routing slips of workflow instances can now be executed by the available microservices.

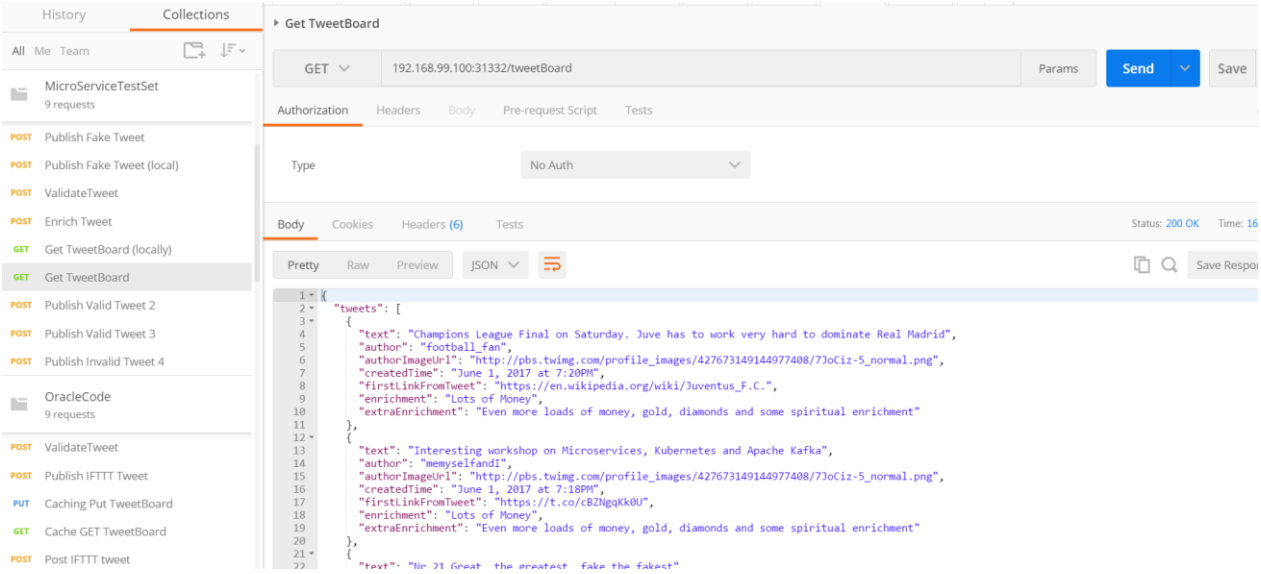
Inspect the port assigned to this microservice using *kubectl get services*. Then access the microservice at http://<docker machine IP>:<assigned port>/tweetBoard. You will not yet see any tweets on the tweet board.

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:



The Tweet Board looks like this:



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)
- 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 or Oracle EventHub) and possibly (if the local Kafka is retained as well) a bridge between the local and the cloud based event bus.

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

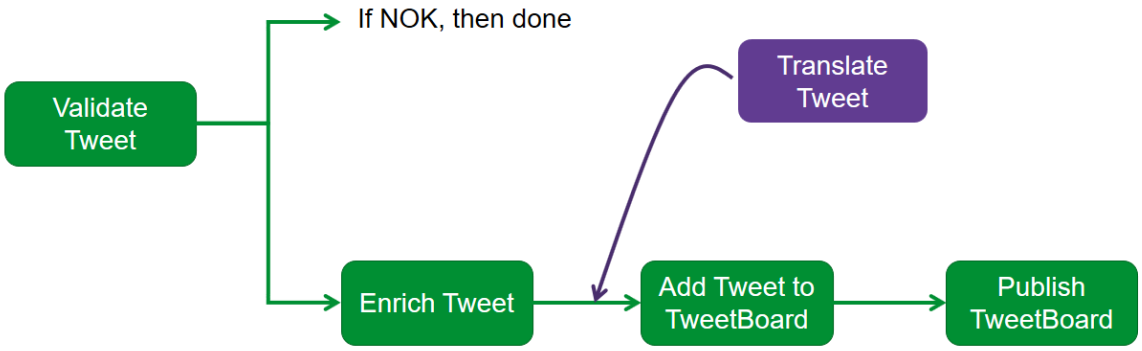
```
kubectl replace -f WorkflowLauncherV2Pod.yaml --force
```

to try this later version of the workflow.

Publish a new tweet to TweetReceiver. Now when you inspect the workflow document in the cache (through the CacheInspector) you will find that in the latest workflow, enrichment is done before validation.

Add Tweet Translation to the Workflow

A more substantial change is the next one, where we introduce a translation of the tweets, into several popular languages.



We first create a microservice called TweetTranslator that can perform the translation of tweets and that can consume workflow events from the Kafka topic. Next, we will update the workflow definition to also include the translation of the tweet.

Change into the directory part4\TweetTranslator and run

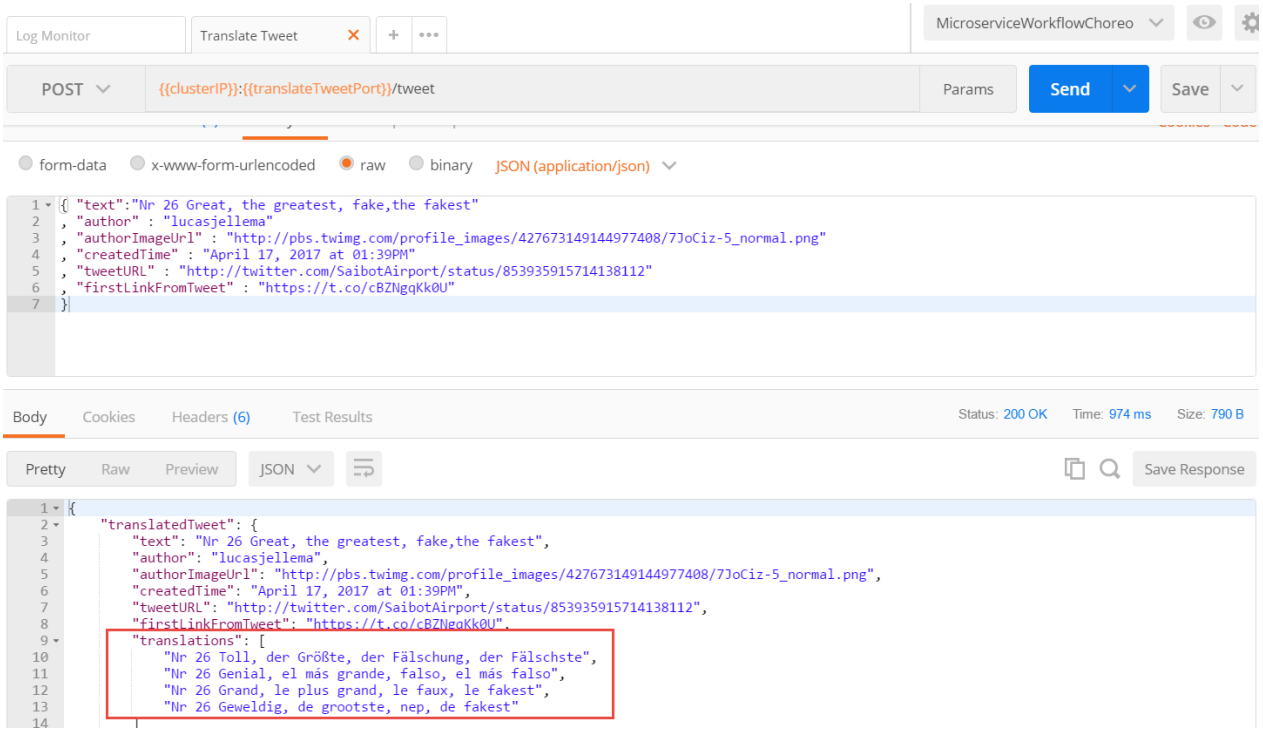
```
kubectl apply -f TweetTranslatorPod.yaml -f TweetTranslatorService.yaml
```

This creates a Pod for the TweetTranslator microservice as well as a Service that exposes the translation capabilities. The microservices listens to the *workflowEvents* topic – for any workflowEvents for a workflow instance that it can make a contribution to. And it can handle direct HTTP requests.

Use *kubectl get services* to find out the port at which the TweetTranslator microservice is exposed.

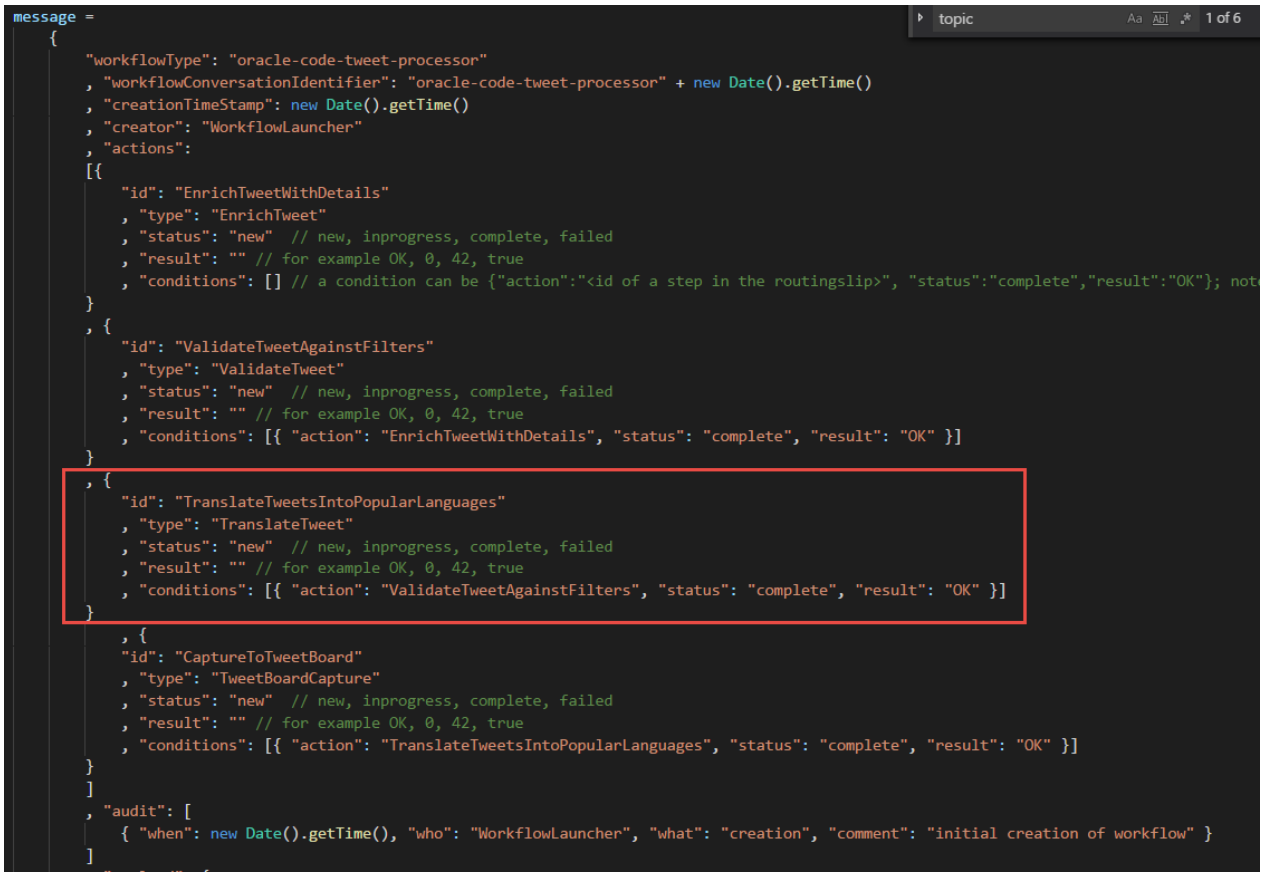
Then update variable *translateTweetPort* in the Postman Environment to the actual port for the Tweet Translator.

Open Request Translate Tweet, edit the body as you see fit and press Send. Now you should see the translation of the Tweet in action.



A somewhat updated version of the Tweet Workflow is available in part4\WorkflowLauncher\ WorkflowLauncherV3.js. In this version, Translation is added to the workflow after validation. This is

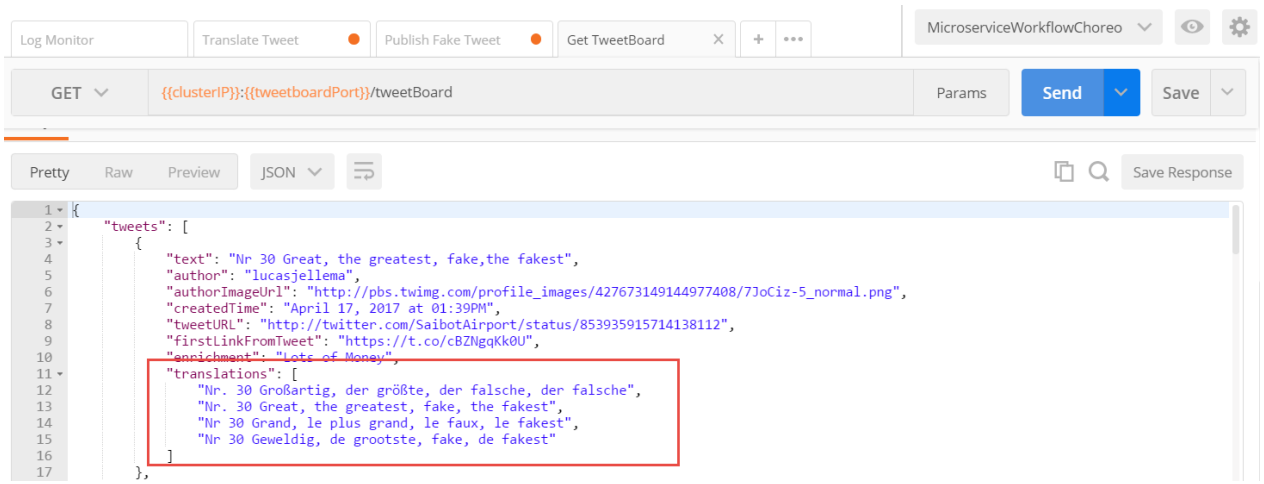
defined in the *message* variable at the bottom of the program. Take a look at definition of var message in the source of WorkflowLauncherV3.js – and try to understand the updated definition of the routing slip:



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

```
kubectl replace -f WorkflowLauncherV3Pod.yaml --force
```

Publish a new tweet to TweetReceiver. Now when you inspect the workflow document in the cache (through the CacheInspector using the workflowConversationIdentifier that you have learned through the LogMonitor) you will find that in the latest workflow, translation has been done. When you retrieve tweets from the TweetBoard, you will also find the translations in the final outcome of the workflow.



5. Streaming Data and Continuous Queries with Kafka KSQL

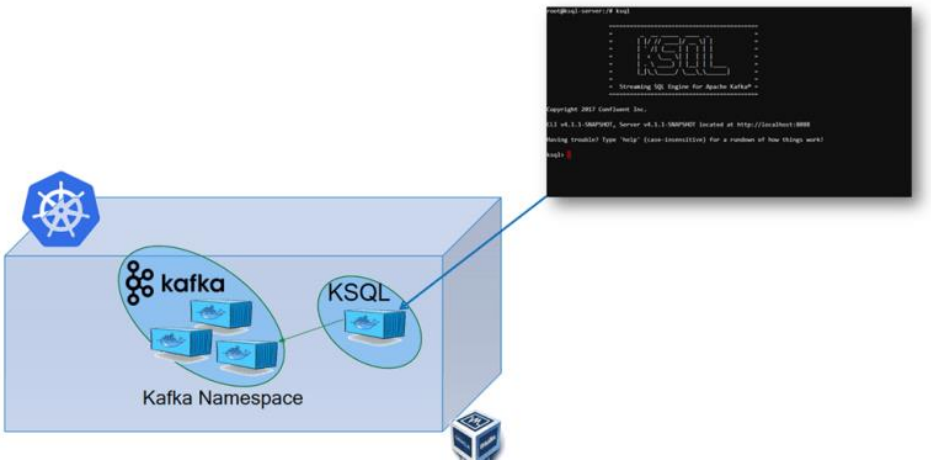
Events published on Kafka Topics are a form of streaming data. They [can] keep coming. Analyzing such 'fluid data' can be challenging. Questions such as: *how many events arrived per time period*, *what was the highest/lowest/aggregate value of all events in certain period – per category* and *tell me if an event arrives that cross thresholds* are common. They may need to be answered over a the past – or they may need to be answered continuously – for example reporting on a dashboard or by sending alert notifications for specific conditions.

Kafka provides specific support for this streaming analysis – through KSQL. With KSQL, we can define queries in a syntax that is very similar to SQL queries. However, KSQL does not query a static data set in database table but instead a stream of events. A query does not produce a single result – a query in KSQL is a continuous question that will keep on reporting on events for as long as it is not interrupted. KSQL queries are targeted at Kafka Topics or KSQL Tables or Streams. These Tables and Streams are themselves underpinned by Kafka Topics – and the product of a KSQL query that produces to those topics. A table is an updateable collection of facts. A stream is collection of immutable facts. New facts can be added to both a table and a stream, but only in a table can facts be updated.

In this section we will get going with KSQL – with a few small steps and simple examples.

Install KSQL in Kubernetes

First, we need to install the KSQL Server into the Kubernetes Cluster. This can be done by following the steps in this blog article: https://technology.amis.nl/2018/05/01/get-going-with-ksql-on-kubernetes/ .



The steps described in the article:

- Create Kubernetes YAML for KSQL [server]Pod – use file *ksqlServerPod.yaml* in directory *part5* of the resources for this workshop

- Apply YAML file with Kubectl – this creates the Pod with the running container:
  `kubectl apply -f ksqlServerPod.yaml`

- Open bash shell on the pod and run ksql [command line utility]
  `kubectl exec -it ksql-server /bin/bash`

  then:
  `ksql`

This should bring up the KSQL command line interface. To test we are in business, type:

```
list topics;
```

(this will display a list of all Kafka Topics)

To check the contents of a Kafka Topic, you can use the PRINT command. For example:

```
PRINT 'logTopic' FROM BEGINNING;
```

This shows the raw content of all messages on the logTopic.

In order to ensure that all messages are associated with a timestamp, execute the following command:

```
SET
'timestamp.extractor'='org.apache.kafka.streams.processor.WallclockTimestampEx
tractor';
```

To list all current property values, use:

```
list properties;
```

Create and Query a KSQL Table

Now you can create a KSQL Table to capture all log messages:

```
CREATE TABLE logs ( module VARCHAR, logLevel VARCHAR, message VARCHAR,
timestamp VARCHAR) WITH \
(kafka_topic='logTopic', value_format='JSON', key='module');
```

The names of the 'columns' correspond with the properties in the JSON message payload. Log Messages are constructed like this:

```
 { "logLevel": "info",

   "module": "TweetTranslator",

   "message": "handleWorkflowEvent : ",

   "timestamp": "2018-5-1 05:06:49"

 }
```

To query the contents of the table (which under the covers is a Kafka Topic itself), use:

```
select TIMESTAMPTOSTRING(ROWTIME, 'yyyy-MM-dd HH:mm:ss.SSS') AS timestring,
rowkey, module, logLevel, message, timestamp from logs;
```

Note that this query does not end. It continuous to run. Use Postman to Validate, Translate or Enrich a Tweet. All actions will produce additional logging events. As a result, the query will produce additional results.

Note: ROWTIME is the timestamp for the message added by Kafka when the message was created in the table.

Analyzing & Aggregating Workflow Events

The events published on topic *workflowEvents* have fairly complex, nested JSON structures. KSQL can deal with that. Columns can be defined using map<> and array<> types.

Use this statement to create a stream called wf_s that produces workflow events based on the Kafka Topic workflowEvents with columns as defined below. Note how *payload* is defined on a nested JSON object using map<VARCHAR,VARCHAR> and how *audit* is based on an array of nested objects using array<map<VARCHAR,VARCHAR>>.

```
create stream wf_s (workflowConversationIdentifier VARCHAR \
```

```
, payload map<VARCHAR,VARCHAR> \
, updateTimeStamp VARCHAR \
, lastUpdater VARCHAR \
, audit array<map<VARCHAR,VARCHAR>> \
) WITH (kafka_topic='workflowEvents', \
 value_format='JSON', \
key='workflowConversationIdentifier' ,\
timestamp='updateTimeStamp');
```

Results can be retrieved from this stream – for example like this:

```
select TIMESTAMPTOSTRING(ROWTIME, 'yyyy-MM-dd HH:mm:ss.SSS') AS timestring \
,  workflowConversationIdentifier \
, audit \
, lastUpdater \
, updateTimeStamp \
, creationTimeStamp \
, payload \
, payload['author'] \
, payload['workflowType'] \
, payload['lastUpdater'] \
from wf_s;
```

This is a straight query – no filtering, aggregation or enrichment yet.

In order to actually get some results from this query – publish a Tweet to the Tweet Receiver in order to kick off a workflow instance.

Alternatively, we can override the default value in KSQL for property auto.offset.reset from *latest* (meaning all the Kafka topics will be read from the current offset - aka latest available data) to *earliest* meaning all data ever published to the topic (and not yet removed).

```
SET 'auto.offset.reset'='earliest';
```

To count the number of actions performed by each updater (actor), execute this query:

```
SELECT lastUpdater, \
```

```
       count(*) \
FROM wf_s \
WINDOW TUMBLING (SIZE 1 MINUTE) \
GROUP BY lastUpdater;
```

It returns the total number of actions by each actor over the last one minute time slice. If you send a few new tweets from Postman to trigger a few more workflow instances, you will see the additional records coming back from this query.

We can create a table (i.e. an Kafka Topic) using a similar KSQL statement that records the number of actions per actor per MINUTE

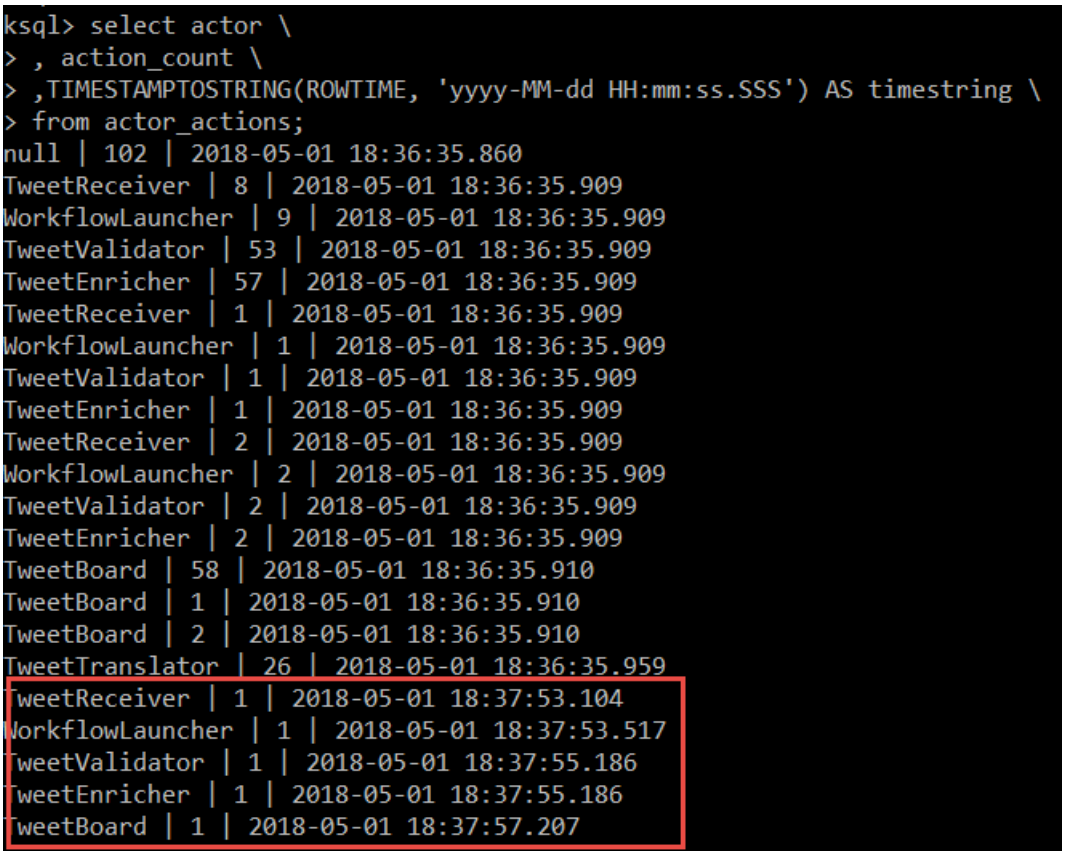
```
create table actor_actions \
 AS SELECT lastUpdater as actor \
, count(*) as action_count \
FROM wf_s \
WINDOW TUMBLING (SIZE 1 MINUTE) \
GROUP BY lastUpdater;
```

This table will be extended every time an event is published on the *workflowEvents* topic, with a record that indicates for the specific *actor* how many actions that *actor* has performed in the last 60 seconds. The records in the table (in reality also events on a Kafka Topic) can be queried like this:

```
select actor \
, action_count \
,TIMESTAMPTOSTRING(ROWTIME, 'yyyy-MM-dd HH:mm:ss.SSS') AS timestring \
from actor_actions;
```

Use Postman to publish one or a few tweets to TweetReceiver. This will trigger the table and indirectly the query.

The result could look something like this:



The red rectangle shows the most recent events, produced from the workflow instance created for a tweet.

To inspect running workflow instances and the time they have been running for, we create a stream:

```
create stream running_workflows as \

select TIMESTAMPTOSTRING(ROWTIME, 'yyyy-MM-dd HH:mm:ss.SSS') AS timestring \

,  workflowConversationIdentifier \

, lastUpdater \

, updateTimeStamp \

, creationTimeStamp \

,(updateTimeStamp - creationTimeStamp ) as runningTime \

from wf_s \

where len(workflowConversationIdentifier) <> 0;
```

and query the records from this stream:

```
select workflowConversationIdentifier \
```

```
, max(runningTime) \

from running_workflows \

WINDOW TUMBLING (SIZE 30 MINUTE) \

group by workflowConversationIdentifier;
```

This returns the maximum difference between creation time and most recent update time – which means the running time at the time of the last update, which is not necessarily the same thing as the total running time until now.