AMIS SIG Introduction Microservices and Implementing Technology with Docker, Kubernetes, Node.js and Kafka –Hands-on

April 2019

The goal of this workshop is to get familiar with some microservices concepts and learn how to use some enabling technologies such as Docker Containers, Kubernetes container platform and Redis cache as well as Kafka event bus. We will be implementing microservices as Dockerized Node.JS applications that run on Kubernetes and leverage microservice platform facilities such as a cache and an event bus.

You will go through a number of steps in this workshop – that have you work with (and install) Kubernetes, Redis, Node.js, Apache Kafka. The first steps however we will make on the Katacoda platform – a browser based cloud platform where we can easily get access to playground and tutorial environments.

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

https://github.com/lucasjellema/2019-fontys-business-and-IT-agility-through-microservice-architecture

Step 1 – Running the Request Counter Node application (on Katacoda)

Note: you can also go through step 1 in a local Node environment if you prefer.

Katacoda is an online platform that offers hundreds of scenarios and sandbox environments to learn about and play with different kinds of technologies. Katacoda is special in that it not only offers the handson instructions – it also provides the runtime environment in which these steps can be executed instantly. Examples of such environments: Linux server, Docker engine, Kubernetes cluster and almost anything that can be run as container. We will make use of a Node environment I this first step.

In order to get to the desired environment and start the hands on, you have to take these steps:

1. Go to https://www.katacoda.com and sign up – using your GitHub account.
2. Open https://www.katacoda.com/courses/nodejs/playground - the Katacoda Node playground
3. Open https://github.com/lucasjellema/2019-fontys-business-and-IT-agility-through-microservice-architecture/tree/master/1-node and work your way through the README.md.

   You will git clone sources from GitHub, then run a Node application that counts HTTP requests. Questions that come up: can the application scale? Can it fail over? Does it remember state – across restarts?

Step 2 – Running the Request Counter Node application from a Docker Container (on Katacoda)

Note: this step too can be done locally. If you have a Docker environment set up on your laptop, it is perfectly fine to do the hands on steps locally.

In order to get to the desired environment and start the hands on, you have to open https://www.katacoda.com/courses/docker/playground - the Katacoda Docker playground

Go to https://github.com/lucasjellema/2019-fontys-business-and-IT-agility-through-microservice-architecture/tree/master/2-docker and work your way through the README.md.

You will git clone sources from GitHub, then run a Docker container that runs the Node Request Counter application, still counting HTTP requests. Questions that come up: how can we run a containerized application? How much do we need to know about the application inside the container in order to run it? How can we check logging from the container? Can we observe and influence the application inside the container?

Can the application scale? Can it fail over? Does it remember state – across restarts? Can we easily run additional components – such as a Redis Cache – and link it into the application container?

Step 3 – Enter: Kubernetes – for Running Containers – such as a containerized Request Counter Node application (on Katacoda)

Note: this step too can be done locally. If you have a Kubernetes environment set up on your laptop, it is perfectly fine to do the hands on steps locally.

In order to get to the desired environment and start the hands on, you have to open https://www.katacoda.com/courses/kubernetes/launch-single-node-cluster scenario - a Katacoda Kubernetes scenario

Go to https://github.com/lucasjellema/2019-fontys-business-and-IT-agility-through-microservice-architecture/tree/master/3-kubernetes and work your way through the README.md.

Next Steps

Learn more about Docker and Kubernetes by exploring scenarios on Katacoda.

- Kubernetes Introduction - https://www.katacoda.com/courses/kubernetes (17 scenarios)
- Istio – Service Mesh - https://www.katacoda.com/courses/istio
- Run applications on Kubernetes - https://www.katacoda.com/javajon/courses/kubernetes-applications

and more

Step 4 – Enter: Kafka – the Event Bus on Kubernetes (on Katacoda)

Note: this step too can be done locally. If you have a Kubernetes environment set up on your laptop, it is perfectly fine to do the hands on steps locally.

In order to get to the desired environment and start the hands on, you have to continue with the Katacoda environment from the previous step, or open again https://www.katacoda.com/courses/kubernetes/launch-single-node-cluster scenario - a Katacoda Kubernetes scenario

Go to https://github.com/lucasjellema/2019-fontys-business-and-IT-agility-through-microservice-architecture/tree/master/4-kafka and work your way through the README.md.

Appendix A - Prepare a local Kubernetes environment

We will work with 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 part of the 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.

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 other 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  --disk-size 50g --memory 6096 --cpus=4
```

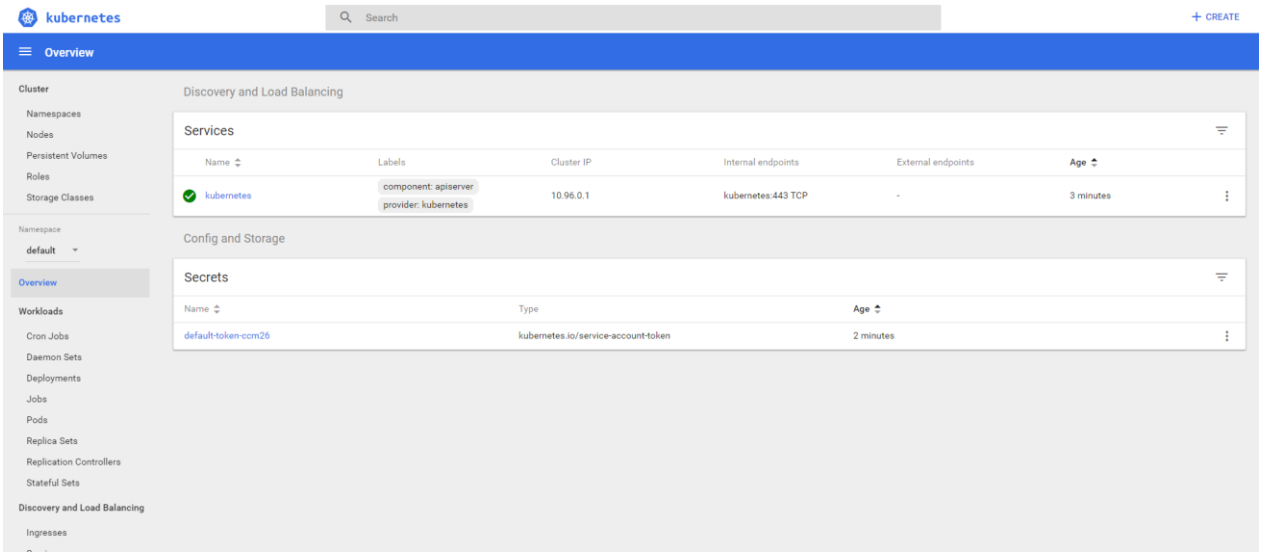
Note: if we do not provide flags to override the default values for disk-size and memory size – as shown below – nor edit minikube config file to override the defaults all the time – we get a fairly small cluster size, not big enough for our purposes today.

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/ .

```
C:\Users\lucas_j>minikube start  --disk-size 50g --memory 6096
o   minikube v0.35.0 on windows (amd64)
>   Creating virtualbox VM (CPUs=2, Memory=6096MB, Disk=50000MB) ...
-   "minikube" IP address is 192.168.99.102
o   Found network options:
    - NO_PROXY=192.168.99.100
-   Configuring Docker as the container runtime ...
-   Preparing Kubernetes environment ...
-   Pulling images required by Kubernetes v1.13.4 ...
-   Launching Kubernetes v1.13.4 using kubeadm ...
:   Waiting for pods: apiserver proxy etcd scheduler controller addon-manager dns
-   Configuring cluster permissions ...
-   Verifying component health .....
+   kubectl is now configured to use "minikube"
=   Done! Thank you for using minikube!
```

```
minikube dashboard --url=true
```

```
C:\Users\lucas_j>minikube dashboard --url=true
-   Enabling dashboard ...
-   Verifying dashboard health ...
-   Launching proxy ...
-   Verifying proxy health ...
http://127.0.0.1:64454/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/
```




```
minikube status
```

```
C:\Users\lucas_j>minikube status
host: Running
kubelet: Running
apiserver: Running
kubectl: Correctly Configured: pointing to minikube-vm at 192.168.99.101
```

To get the IP address of the Minikube cluster:

```
minikube ip
```

```
C:\Users\lucas_j>minikube ip
192.168.99.101
```

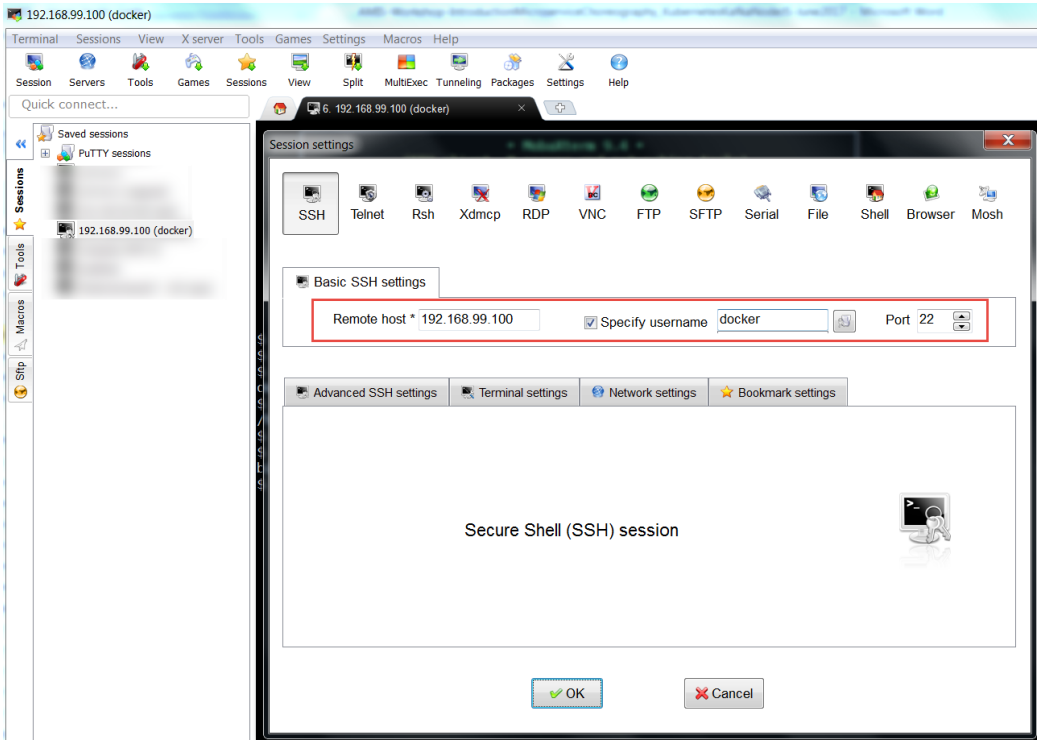
To check on the nodes of the cluster, we can do:

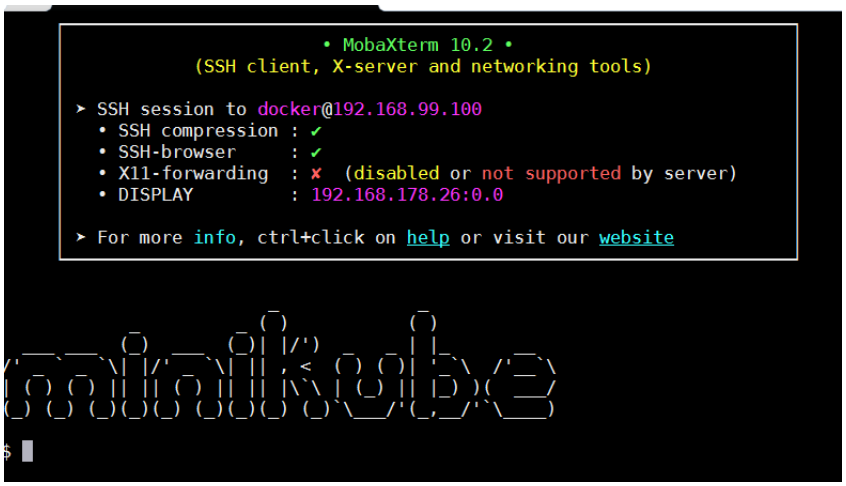
```
kubectl get nodes
```

```
C:\Users\lucas_j>kubectl get nodes
NAME       STATUS    ROLES     AGE       VERSION
minikube   Ready     master    6m        v1.13.4
```

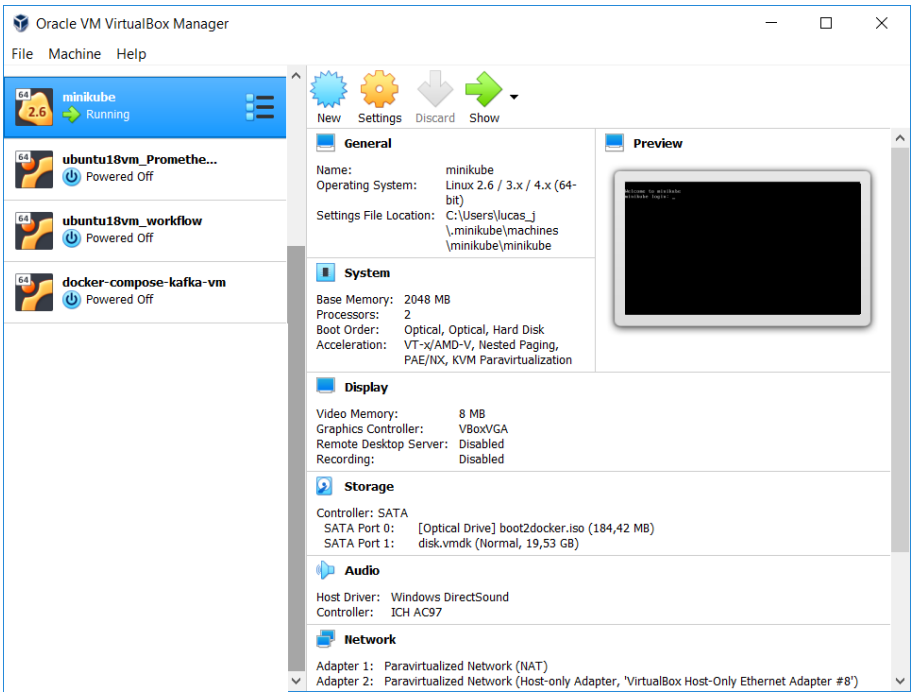
Test connect into the minikube VM through an SSH session:

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

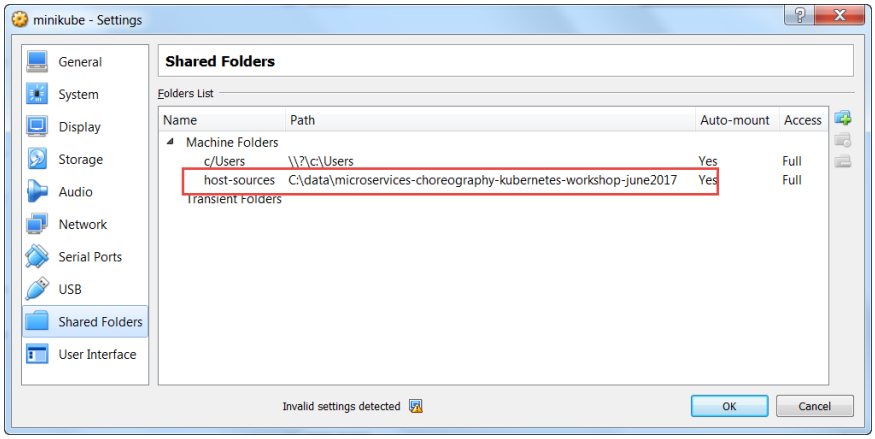




Note: after running for the first time, a new VM called minikube will show up in the Virtual Box GUI.



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



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
```

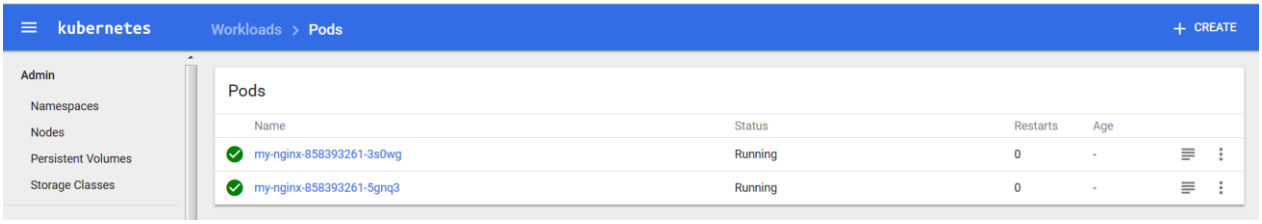
```
C:\data\microservices-choreography-kubernetes-workshop-june2017\part1\kubernetes>kubectl run my-nginx --image=nginx --replicas=2 --port=80
deployment "my-nginx" created
```

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:\Users\lucas_j>kubectl run my-nginx --image=nginx --replicas=2 --port=80
deployment.apps "my-nginx" created

C:\Users\lucas_j>kubectl get pods
NAME                        READY     STATUS    RESTARTS   AGE
my-nginx-64fc468bd4-5mwwh   1/1       Running   0          29s
my-nginx-64fc468bd4-8jb9g   1/1       Running   0          29s

C:\Users\lucas_j>kubectl get deployments
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
my-nginx   2         2         2            2           29s
```

In the Dashboard:

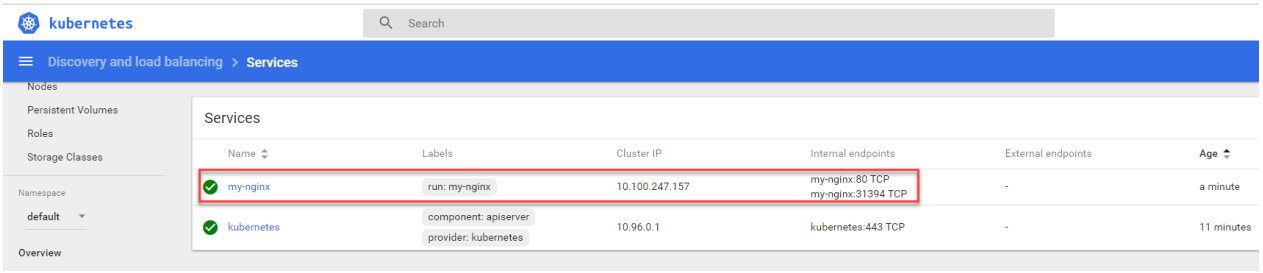


At this moment, the my-nginx containers cannot be accessed from outside the cluster. They need to be exposed through a Service:

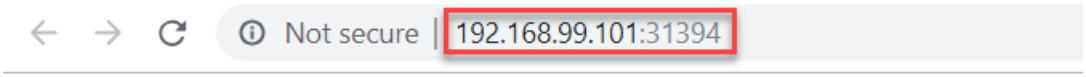
```
# expose your deployment as a service
kubectl expose deployment my-nginx --type=NodePort


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

```
C:\Users\lucas_j>kubectl expose deployment my-nginx --type=NodePort
service "my-nginx" exposed

C:\Users\lucas_j>kubectl get svc
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP        10m
my-nginx     NodePort    10.100.247.157  <none>        80:31394/TCP   22s
```

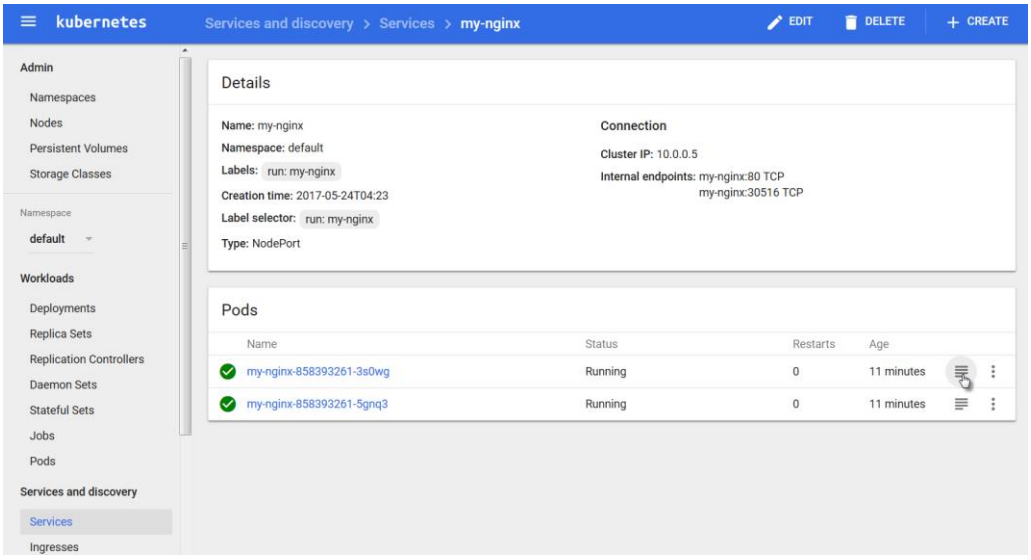
And in the Dashboard:



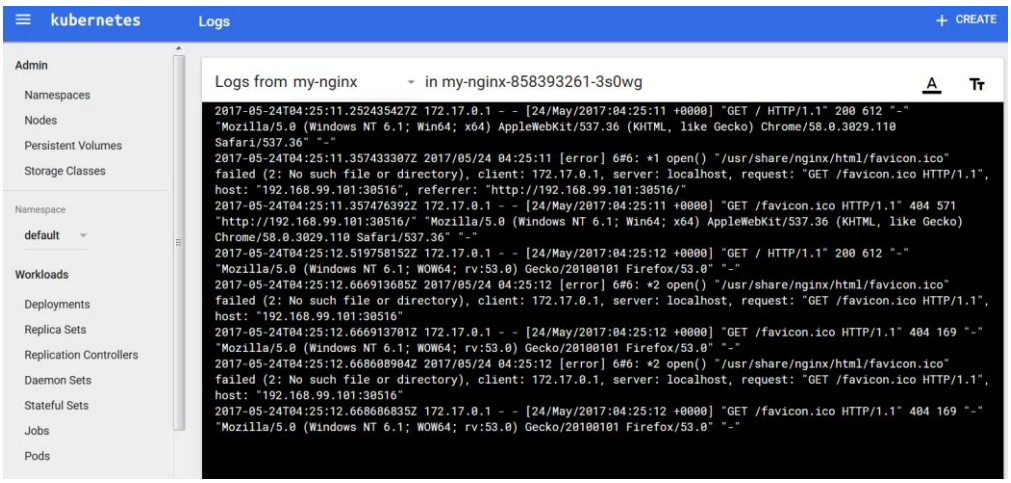

```
# Access your service from your default browser
minikube service my-nginx
```



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:



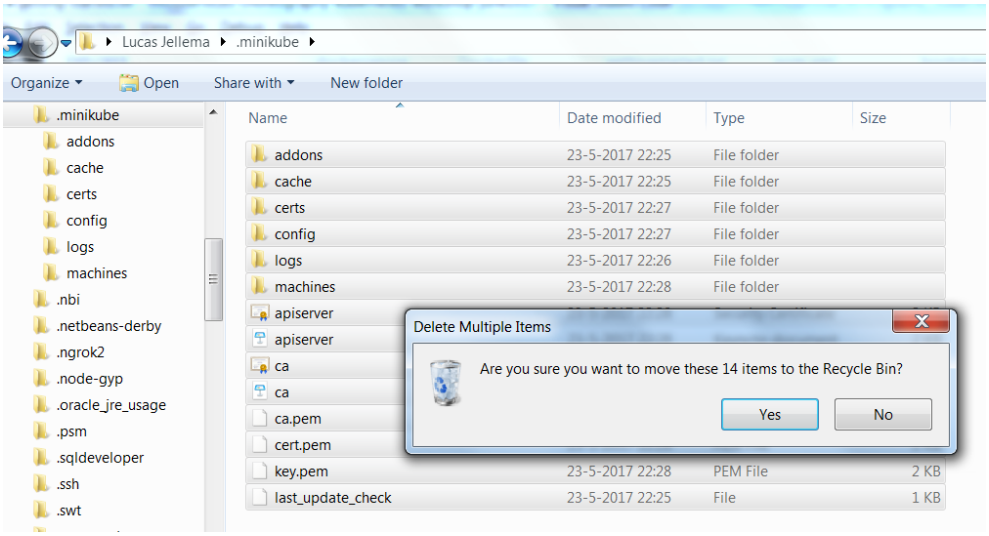
The logging will be shown:



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:



Appendix B - Introducing Kafka the Event Bus into Kubernetes (locally)

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. For a quick introduction to Apache Kafka, check out: https://kafka.apache.org/intro.

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 Minikube Kubernetes YAML files found in https://github.com/d1egoaz/minikube-kafka-cluster. 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 Kafka on Kubernetes

Follow these instructions for getting a Kafka Cluster running on your Kubernetes cluster.

```
git clone https://github.com/d1egoaz/minikube-kafka-cluster
```

```
cd minikube-kafka-cluster
```

Then execute these four statements to create the Kubernetes resources that run the Kafka cluster.

```
kubectl apply -f 00-namespace/
```

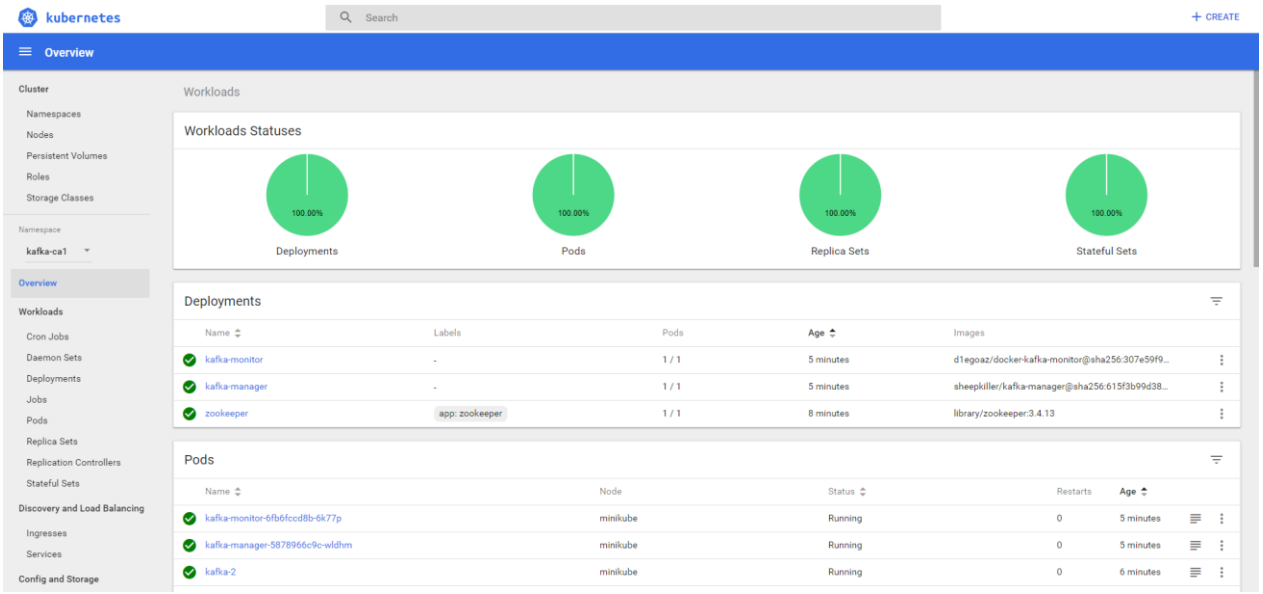
```
kubectl apply -f 01-zookeeper/
```

```
kubectl apply -f 02-kafka/
```

```
kubectl apply -f 03-yahoo-kafka-manager/
```

In the Kubernetes dashboard, switch to the kafka-ca1 namespace and look what has been created: Note: it will take some time for all Pods to be running correctly. After a little while, things are likely to get right - unless the minikube cluster is starved of physical resources (memory and disk).

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



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.

Run Kafka Manager

Execute at the command line:

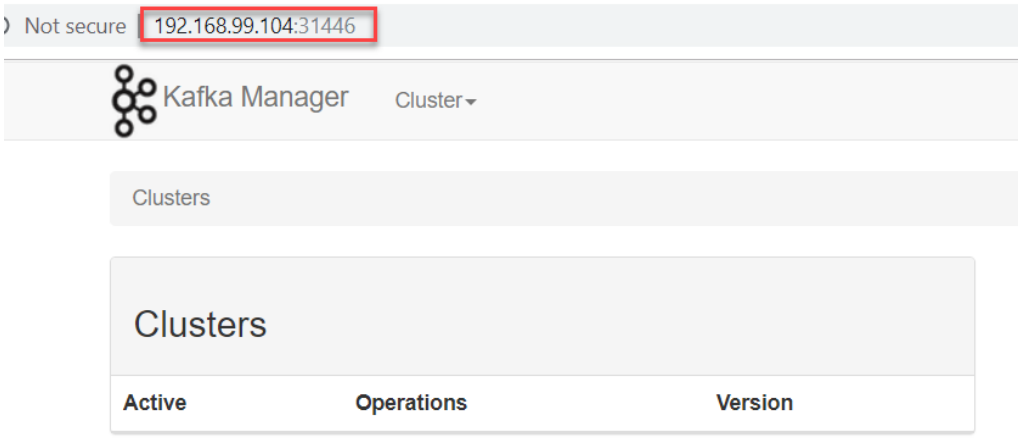
```
kubectl get svc --namespace=kafka-ca1
```

This should list the services in kafka-ca1 namespace, one of which is kafka-manager. We are looking for the port at which this service is exposed:

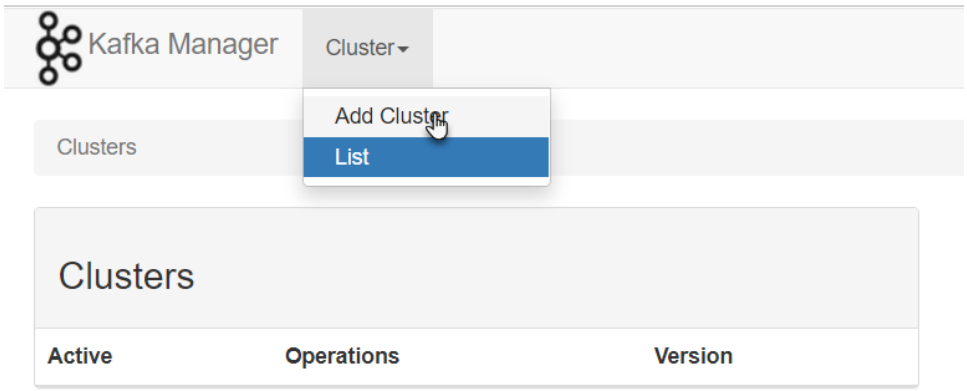
```
C:\Users\lucas_j\minikube-kafka-cluste>kubectl get svc --namespace=kafka-ca1
NAME                TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                AGE
kafka               ClusterIP   None             <none>        9092/TCP,9999/TCP      3m
kafka-manager       NodePort    10.105.242.15    <none>        9000:31446/TCP         1m
zookeeper-service   NodePort    10.105.111.207   <none>        2181:32259/TCP         3m
```

Using ‘minikube ip’ we can learn the IP address for the minikube cluster.

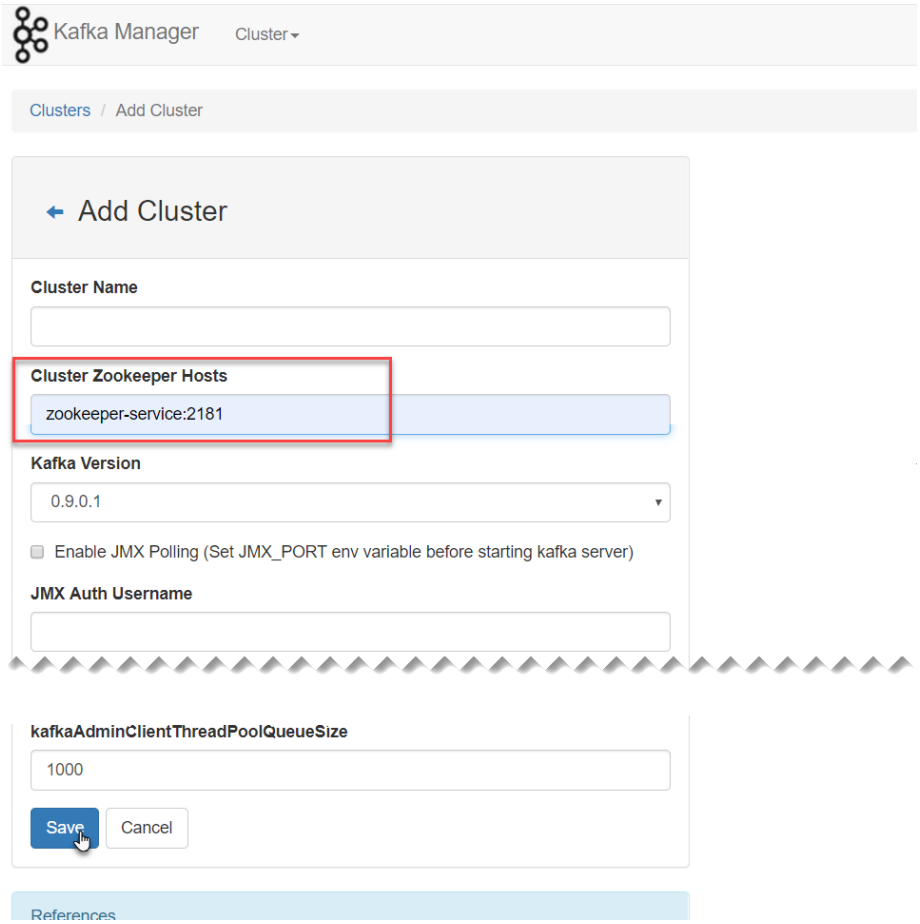
In your browser, you can open the Kafka Manager using the minikube cluster IP and the kafka-manager service port:



Open the Cluster menu and click on Add Cluster:



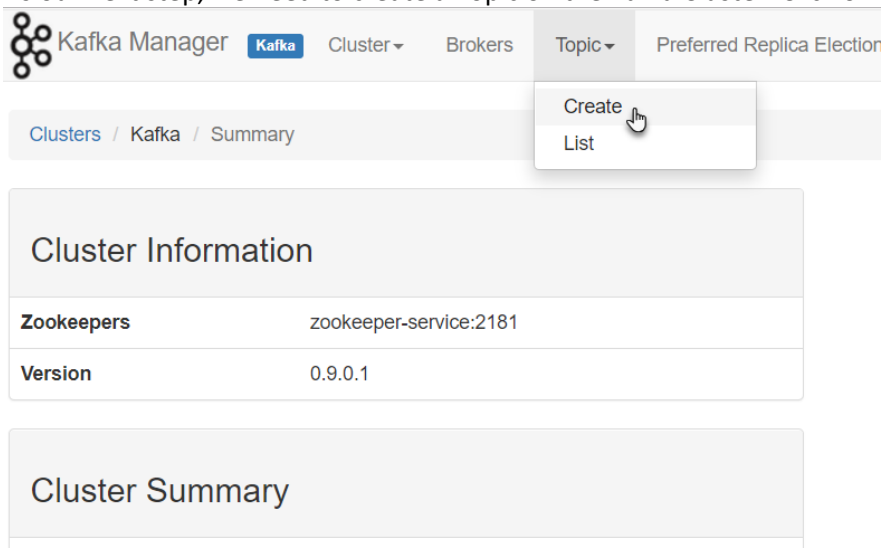
Add new cluster, and use the following data for Cluster Zookeeper Hosts: zookeeper-service:2181 and press Save. Note: this is not the creation of new Kafka cluster but instead the registration with Kafka Manager of an existing cluster.



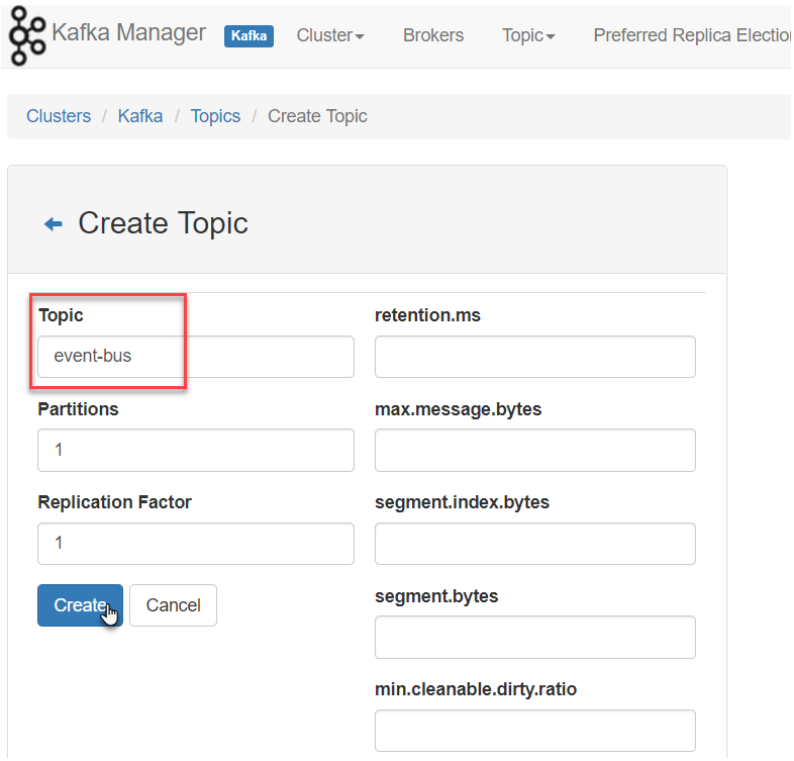
Now you can use Kafka Manager to manage and monitor the Kafka Cluster.

Create Topic

As our next step, we need to create a Topic on the Kafka Cluster. Click on Topics, the on Create:



Set the name to the new topic to *event-bus*. Accept all other default settings and press Create.



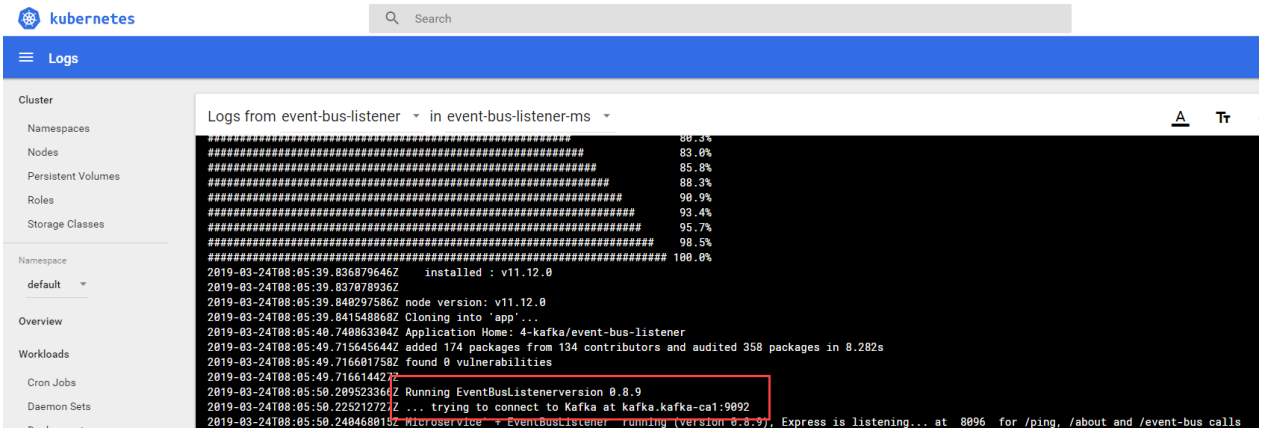
Run an Application with Kafka Interaction on Kubernetes

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 EventBusListenerPod.yaml file in directory 4-kafka/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-service.kafka-ca1, the fully qualified name of the name of the Zookeeper service).

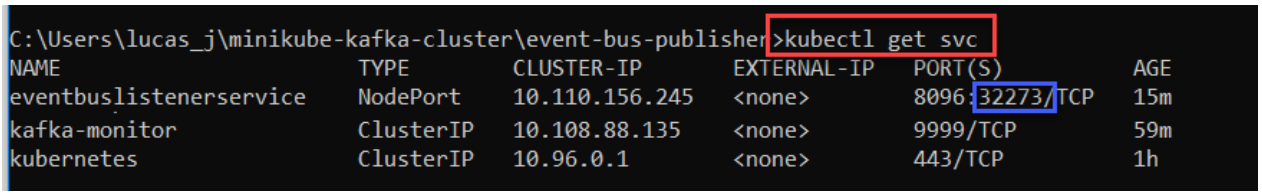
Now run from the specified directory:

```
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:

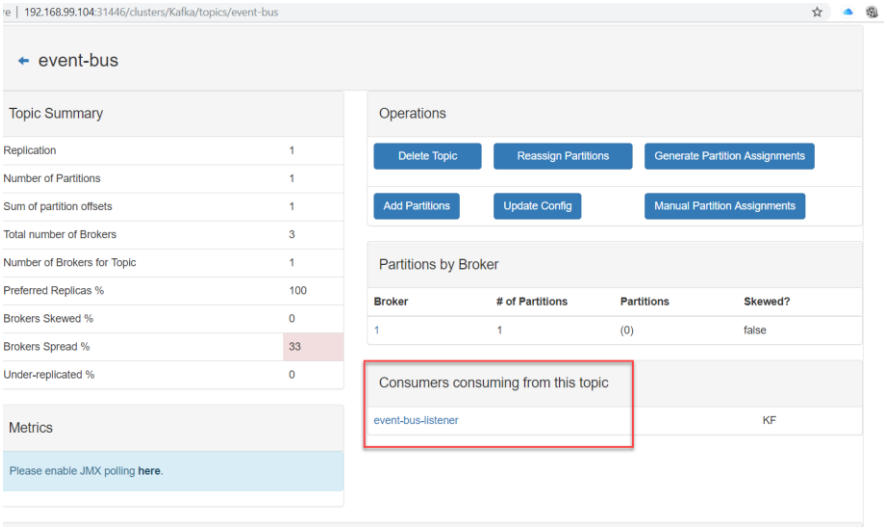


Using the port assigned to service



vou can access the event bus listener application from the browser on your laptop (using the IP address of the minikube cluster) and get a list of all events consumed thusfar from topic event-bus – which are probably none at this stage:

Kafka Manager has recognized the listener or consumer:



Application on Kubernetes that Publishes Events

As a next step, we will run an application 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* application 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 applications that are completely unaware of each other and only need to know about the common platform event-bus capability.

Directory 4-kafka\event-bus-publisher contains the sources for this application. Deploy Pod ad Services for the Event Bus Publisher:

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

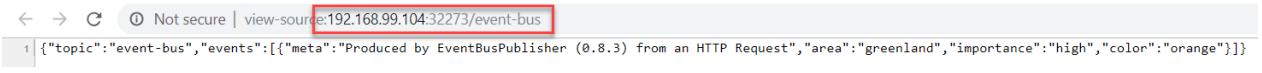
Check in Kubernetes Dashboard or through *kubernetes get pods* and *kubernetes get services* if the creation is complete – and what the port is that was assigned to the *EventBusPublisherService*.

```
C:\Users\lucas_j\minikube-kafka-cluster\event-bus-publisher>kubectl get svc
NAME                        TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
eventbuslistenerservice     NodePort    10.110.156.245   <none>        8096:32273/TCP   15m
eventbuspublisherservice    NodePort    10.104.164.34    <none>        8097:31476/TCP   11m
kafka-monitor               ClusterIP   10.108.88.135    <none>        9999/TCP         59m
kubernetes                  ClusterIP   10.96.0.1        <none>        443/TCP          1h
```

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 application 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 application.



When the HTTP Request to the Event Bus Publisher results in a message returned from the Event Bus Listener application, 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-ca1* namespace artefacts that are running just fine. A quick way to tear down all components in the Kubernetes default namespace is:

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

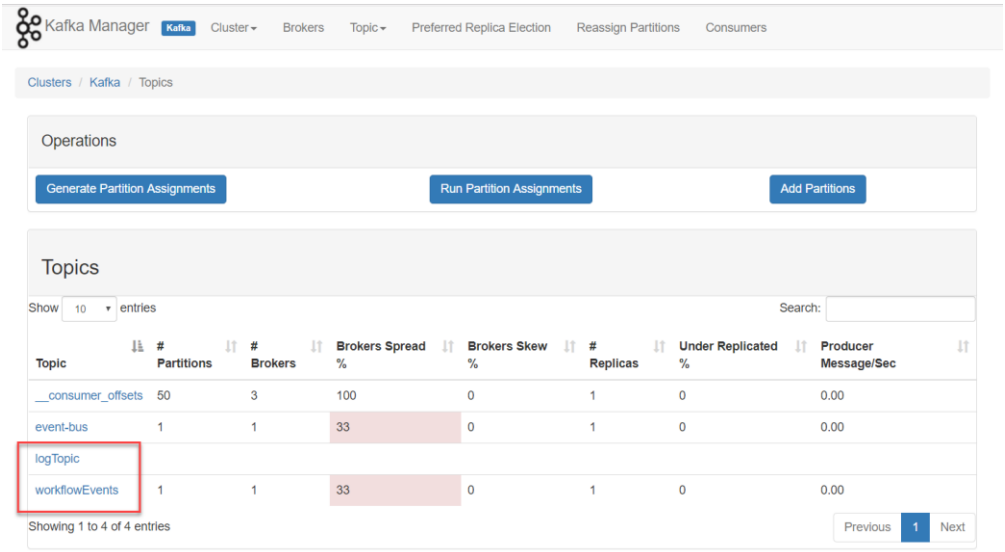
Appendix C - Microservice Choreography – on Kubernetes and Kafka (locally)

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 5 – Microservices 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 – for example through the Kafka Manager - called workflowEvents and logTopic respectively.



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

A new namespace is created for all resources associated with this *choreography* demonstration, from directory 5-microservices:

```
kubectl apply -f namespace.yml
```

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 --namespace=choreography
```

```
kubectl expose deployment redis-cache --type=NodePort  --
namespace=choreography
```

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 5-microservices\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:
http://<minikube IP>:<port assigned to CacheInspector Service>/?key=TheAnswer

we can read cache entries from the Redis Cache, for example:
http://192.168.99.106:32528/cacheEntry?key=TheAnswer

Instead of through the browser, you can also make a curl request to retrieve this information from the CacheInspector:

```
curl -X GET \
  'http://<minikube IP>:<port CacheInspector Service>
/cacheEntry?key=TheAnswer'
```

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 5-microservices/LogMonitor, run:

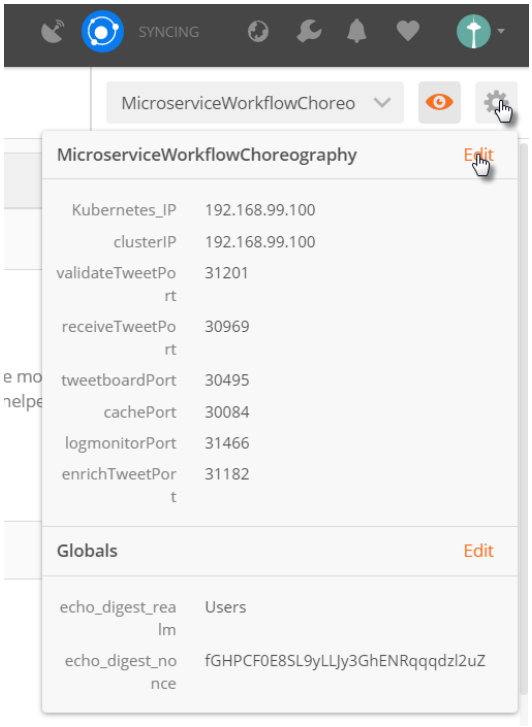
```
kubectl create -f LogMonitorPod.yaml  -f LogMonitorService.yaml
```

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:

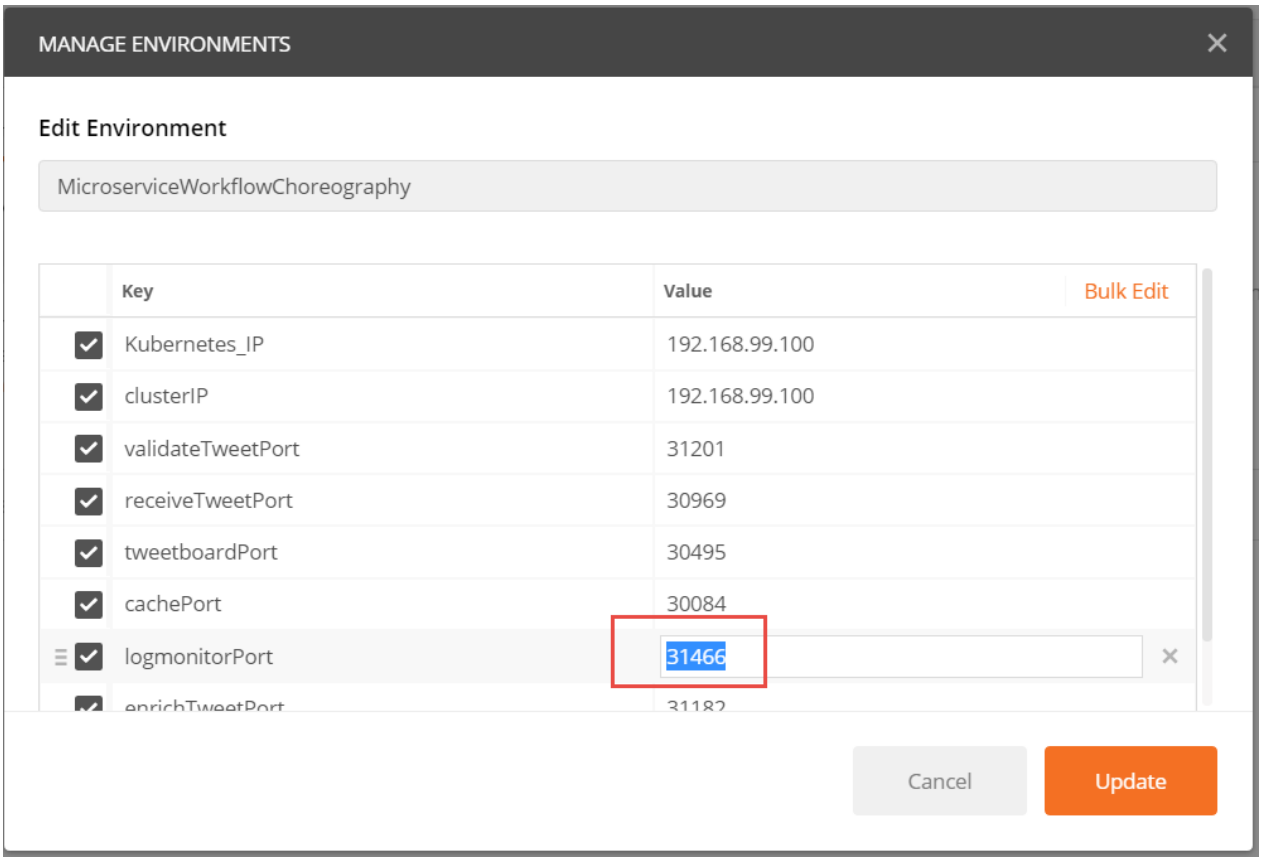
http://kubernetesIP:logMonitorPort/logs

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:

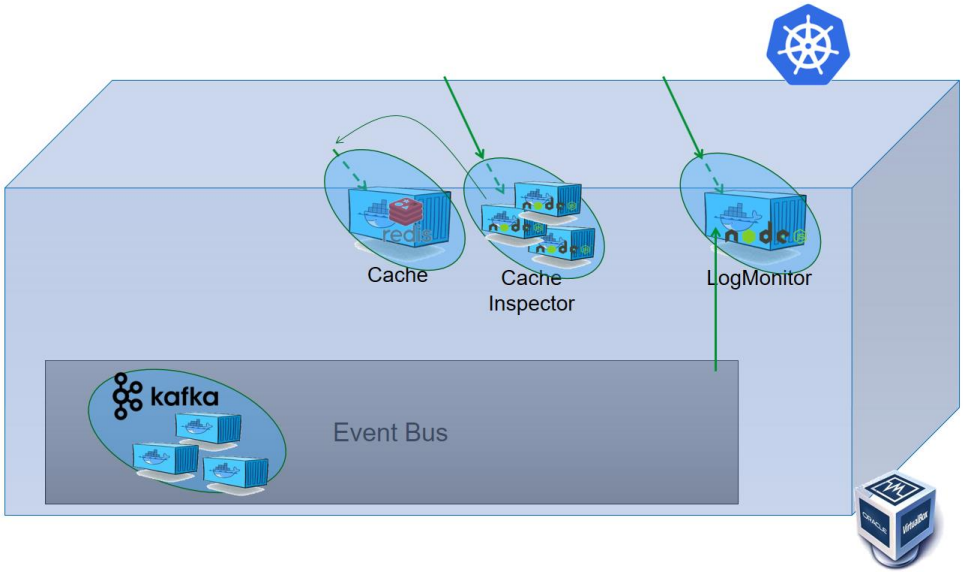


Update the logMonitorPort variable to the value retrieved using *kubectl get services*.



Then press Update to save the configuration changes.

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



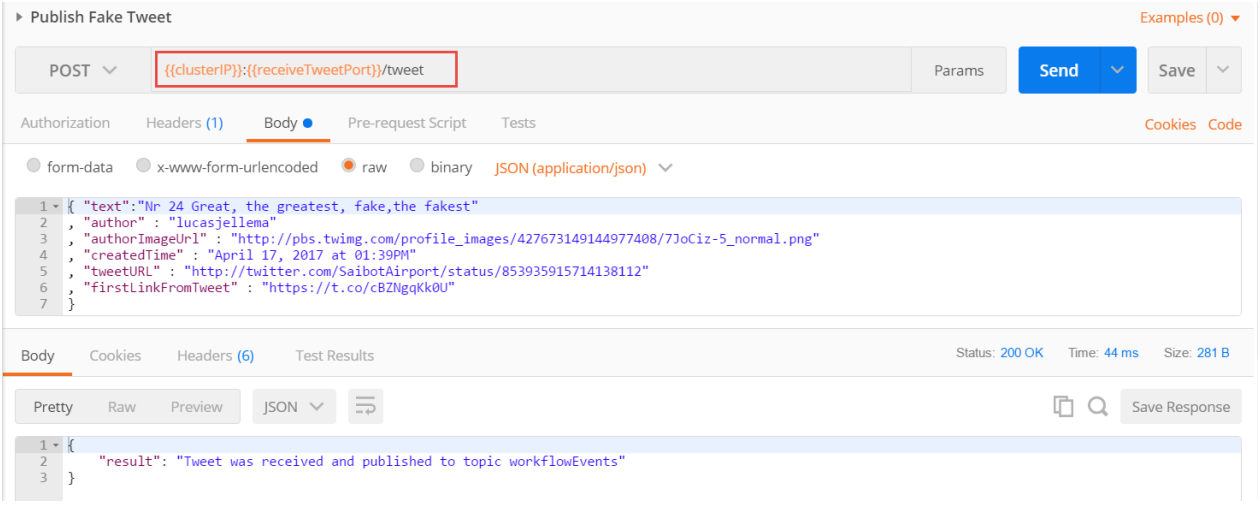
Run Microservice TweetReceiver

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 5-microservices/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.



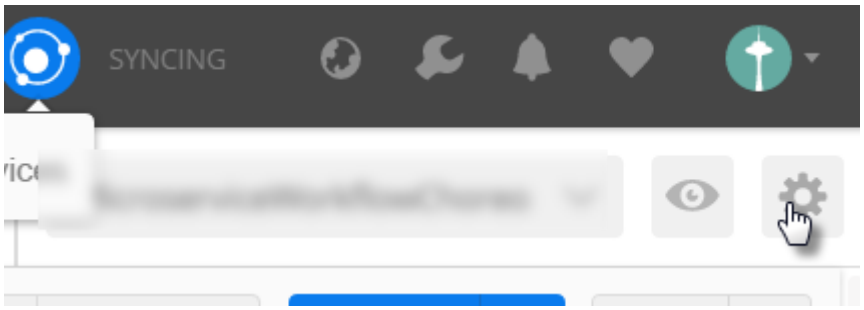
Using *kubectl get services* you can inspect the service that is created and the port at which it is exposed.

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

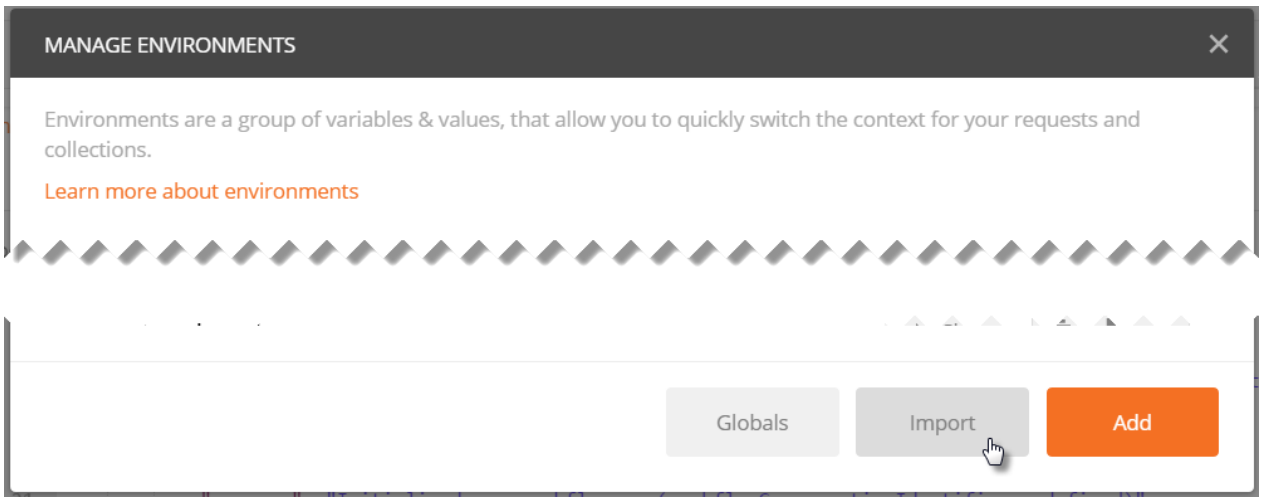
Run Postman.

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:



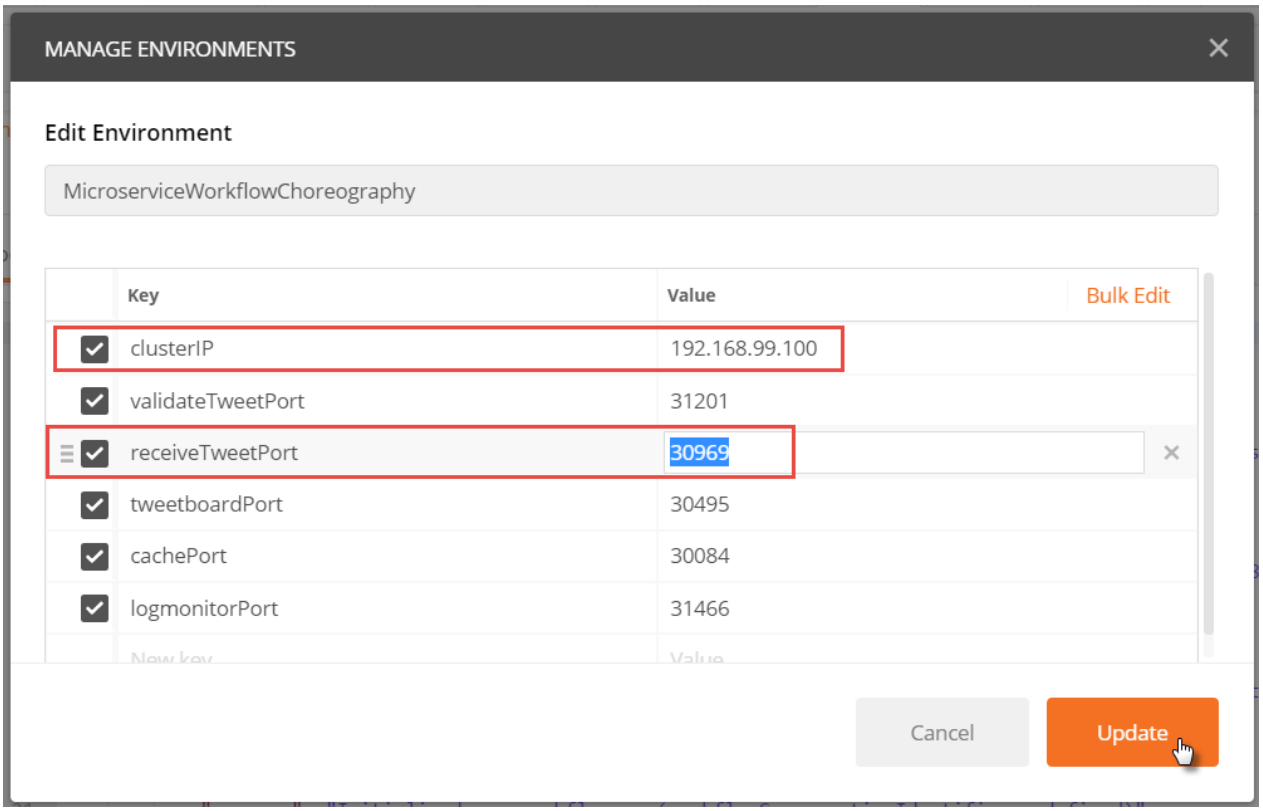
A popup window opens. Click on Import:



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*.

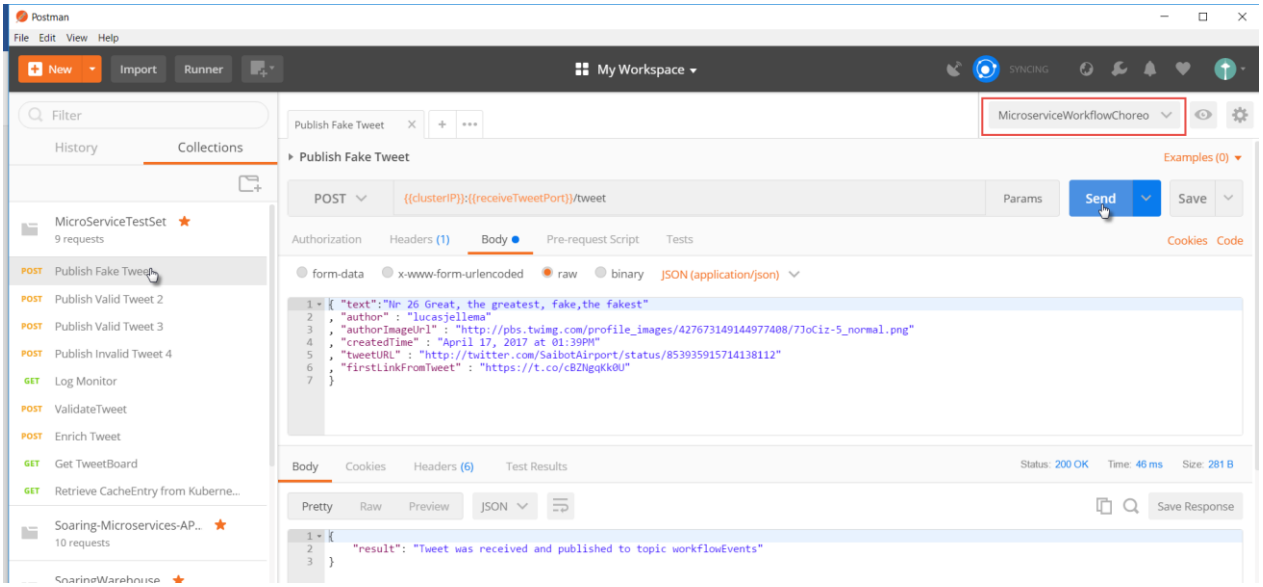
```
c:\data\1-FontysHogeschool-9may-2018\workshop-event-bus-microservice-choreography\part4\TweetReceiver>minikube ip
192.168.99.100

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.

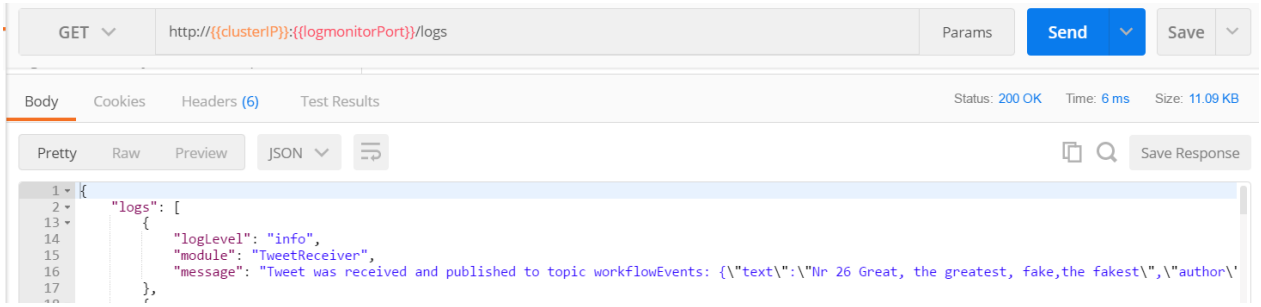
Ensure that the currently active environment is indeed *MicroserviceWorkflowChoreography*.

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:

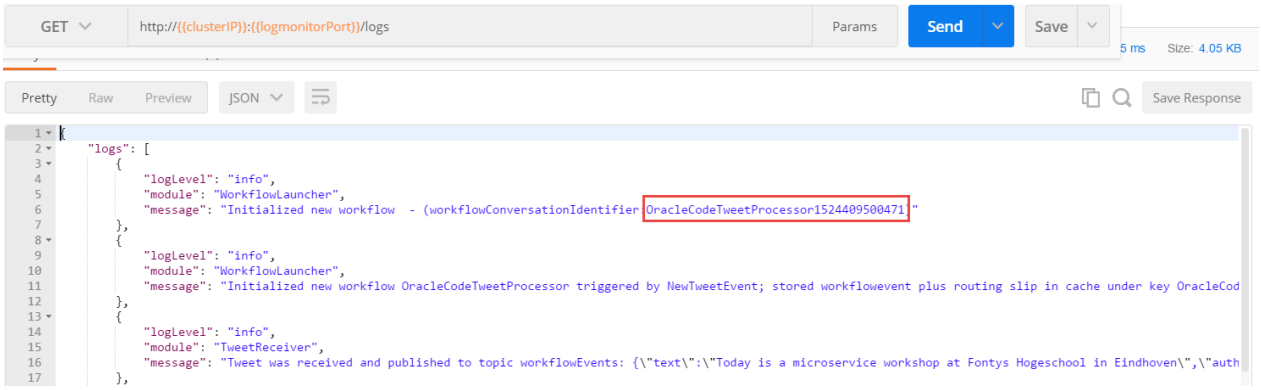
```
kubectl create -f WorkflowLauncherPod.yaml
```

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.

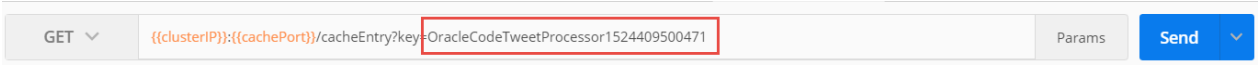
This microservices does not expose an external service – all it does is listen to events on the Kafka Topic [and turn a NewTweetEvent into a Workflow Event with a routing slip for the workflow that has to be executed]. This routing slip is defined in the *message* variable at the bottom of the file WorkflowLauncher.js.

```
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.



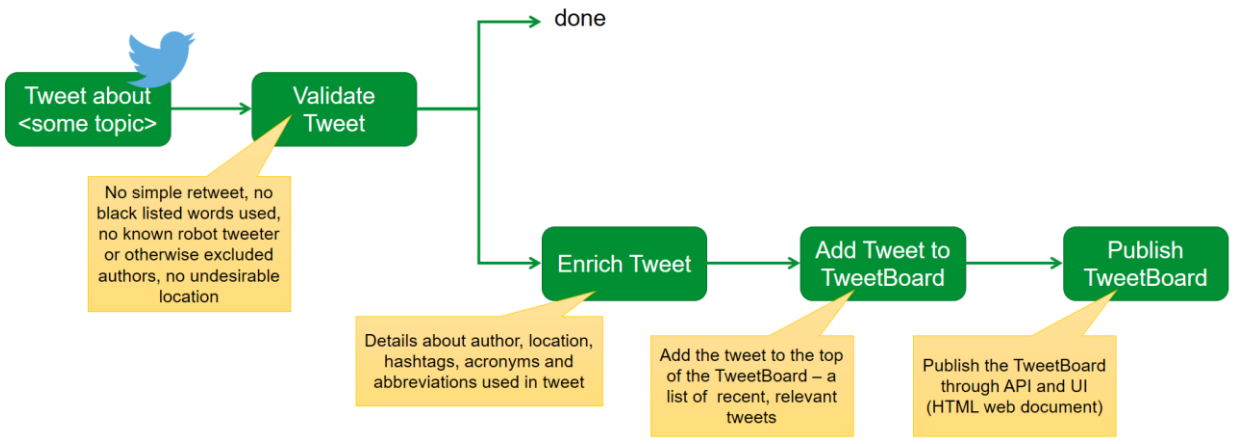
The WorkflowLauncher creates a workflow instance for which we can read the identifier in the LogMonitor output. Using this identifier, we can retrieve the routing slip from the CacheInspector service:



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 5-microservices/ValidateTweet.

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

Also to expose an API for this microservice:

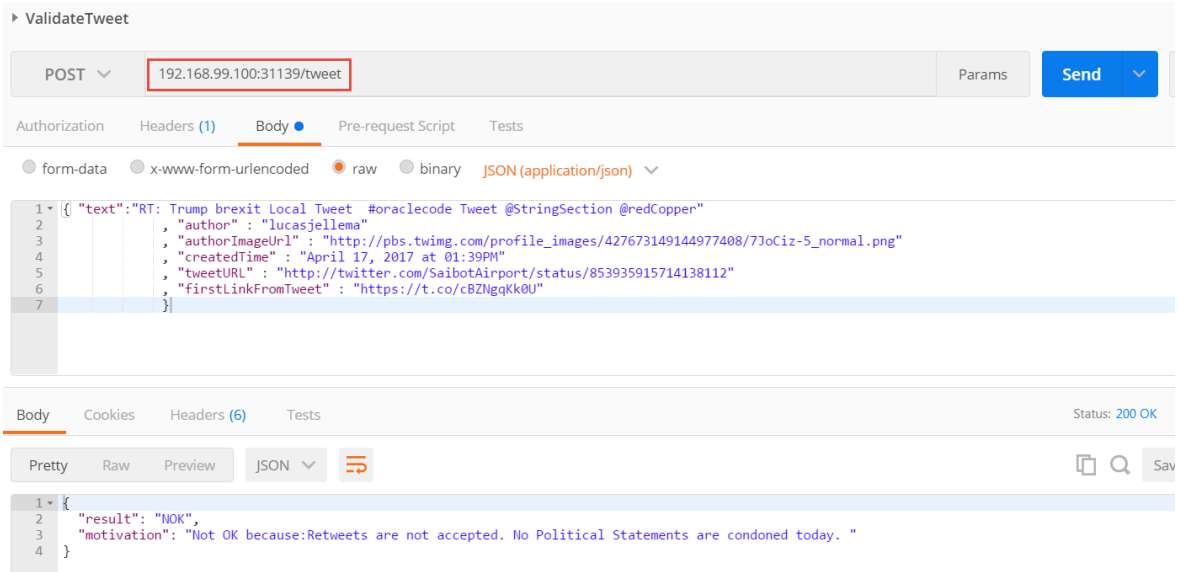
```
kubectl create -f ValidateTweetService.yaml
```

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
```

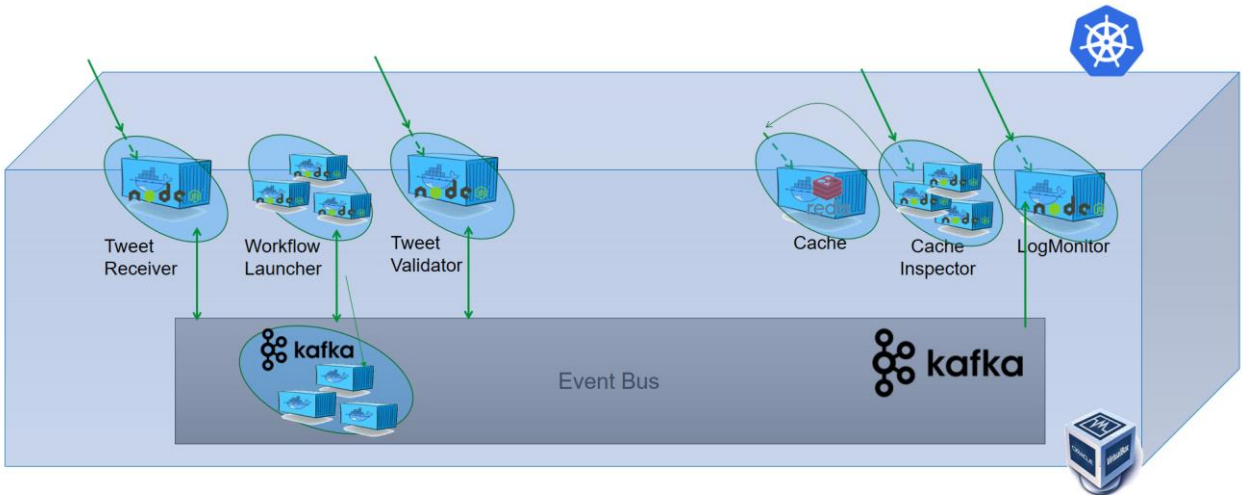
```
c:\data\microservices-choreography-kubernetes-workshop-june2017\part4\ValidateTweet>curl http://192.168.99.100:31139/about
About TweetValidator API, Version 0.8Supported URLs:/ping (GET)
;/tweet (POST)NodeJS runtime version v8.0.0incoming headers{"user-agent":"curl/7.30.0","host":"192.168.99.100:31139","accept":"*/*"}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

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 5-microservices/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 5-microservices/TweetEnricher:

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

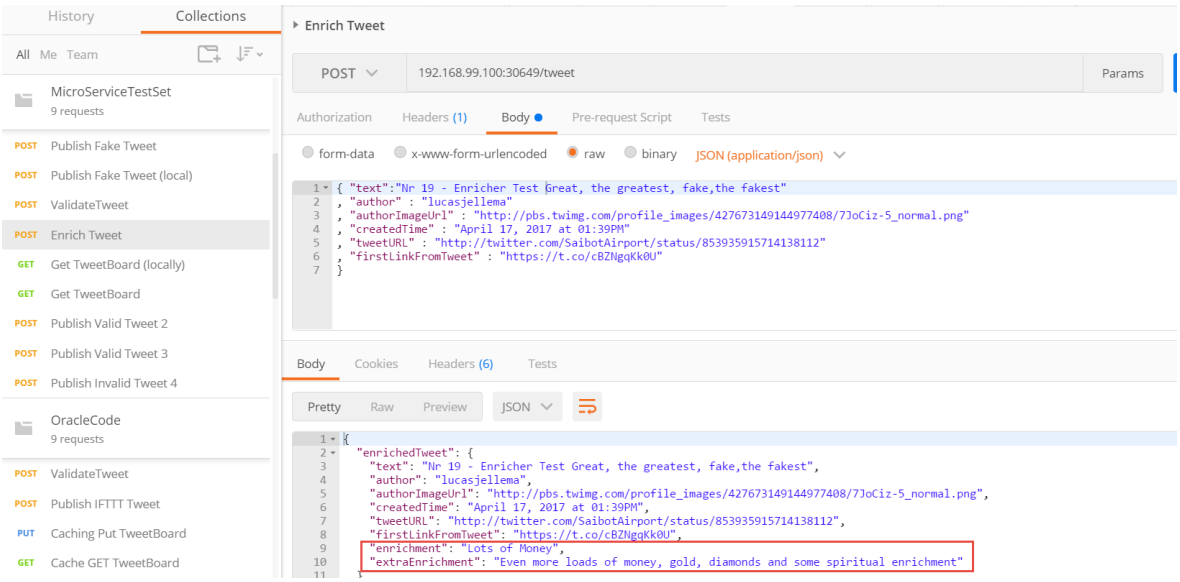
Also to expose an API for this microservice:

```
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 5-microservices/TweetBoard:

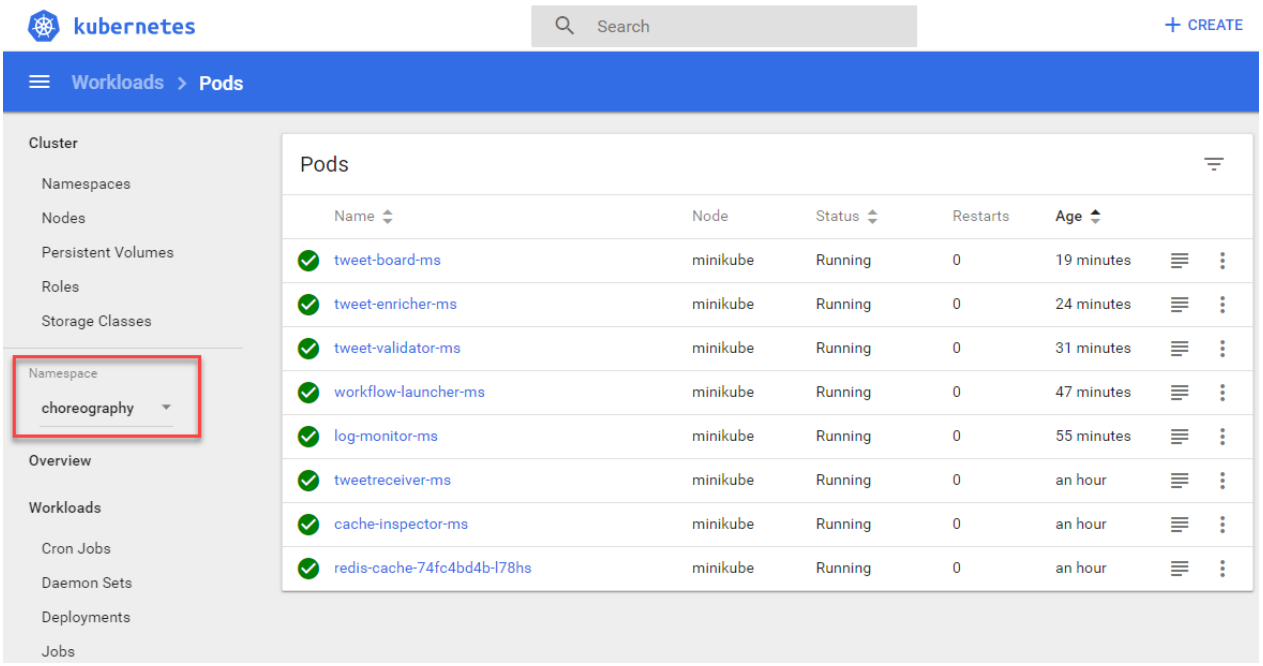
```
kubectl create -f TweetBoardPod.yaml
```

and expose the microservice as a Service:

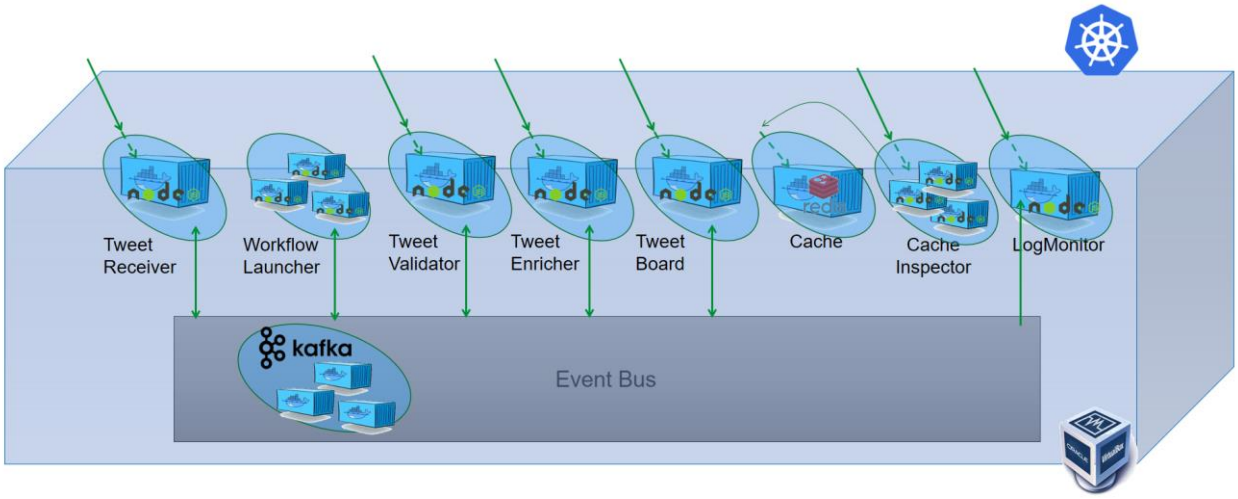
```
kubectl create -f TweetBoardService.yaml
```

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 *choreograohy* namespace :



This corresponds with this overview architecture diagram:

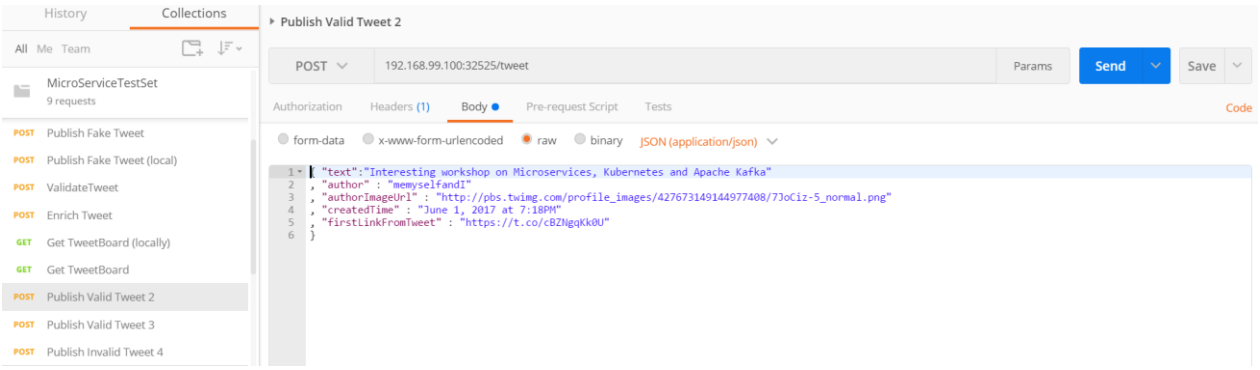


All actions in the routing slips of workflow instances can now be executed by the available microservices.

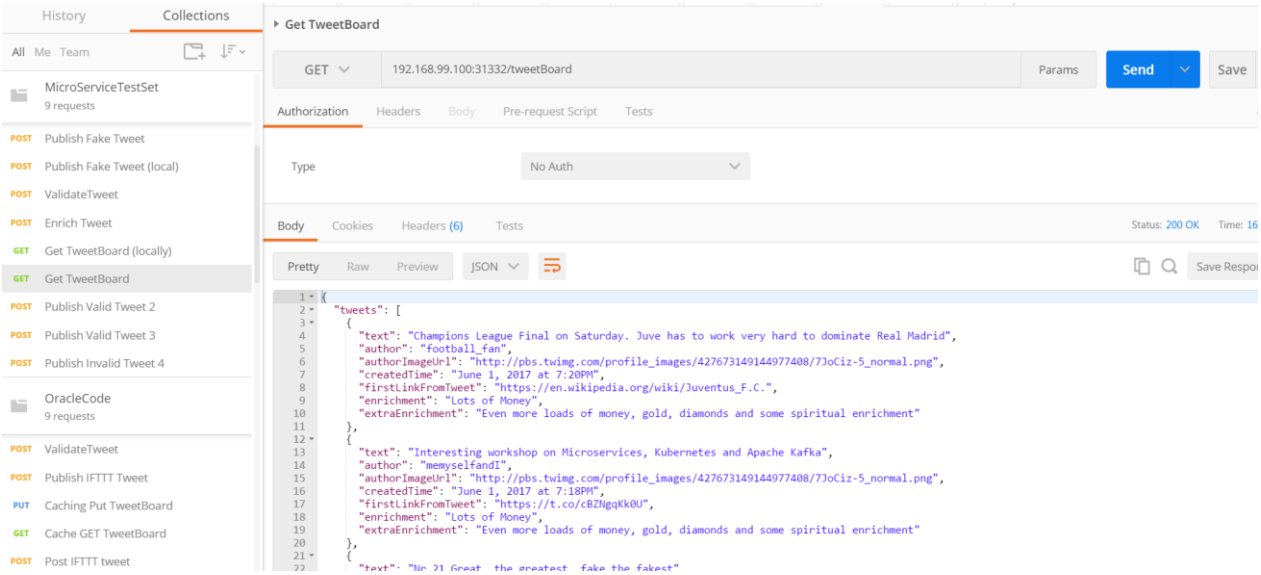
Inspect the port assigned to this microservice using *kubectl get services --namespace choreograhy*. Then access the microservice at http://<Minikube IP>:<assigned port>/tweetBoard. You will not yet see any tweets on the tweet board.

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) 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 5-microservices\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

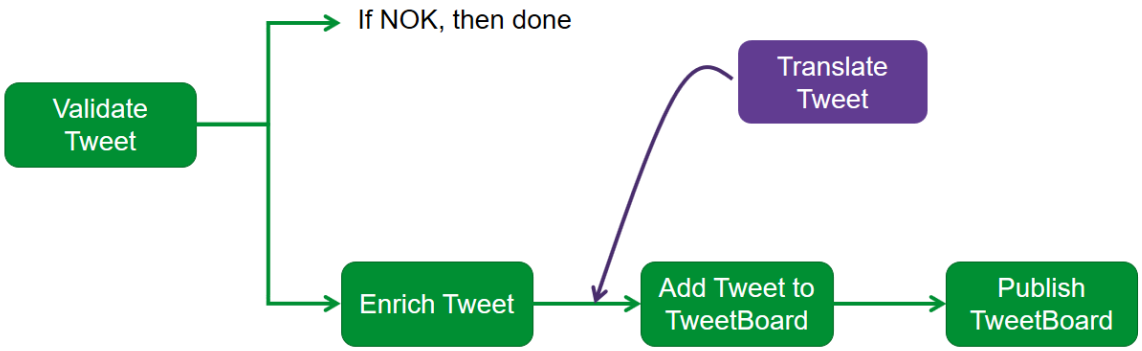
```
kubectl replace -f WorkflowLauncherV2Pod.yaml --force
```

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 5-microservices\TweetTranslator and run

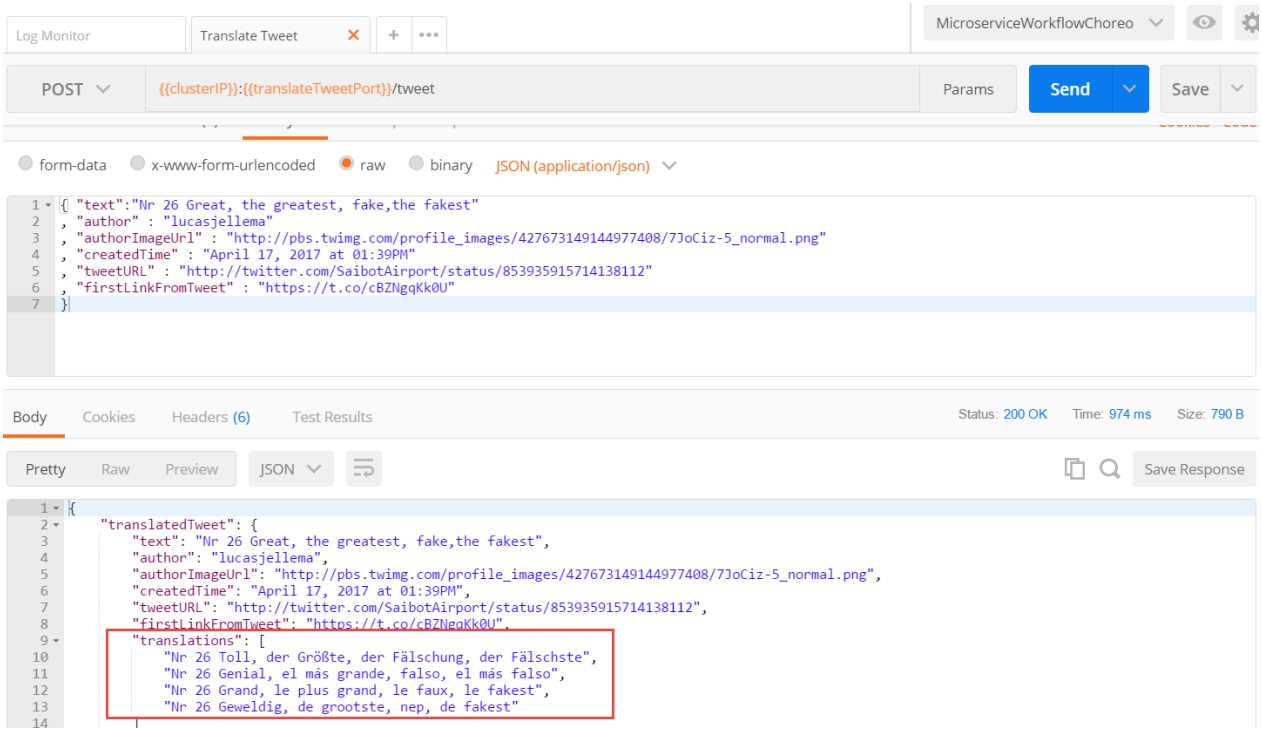
```
kubectl apply -f TweetTranslatorPod.yaml -f TweetTranslatorService.yaml
```

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.



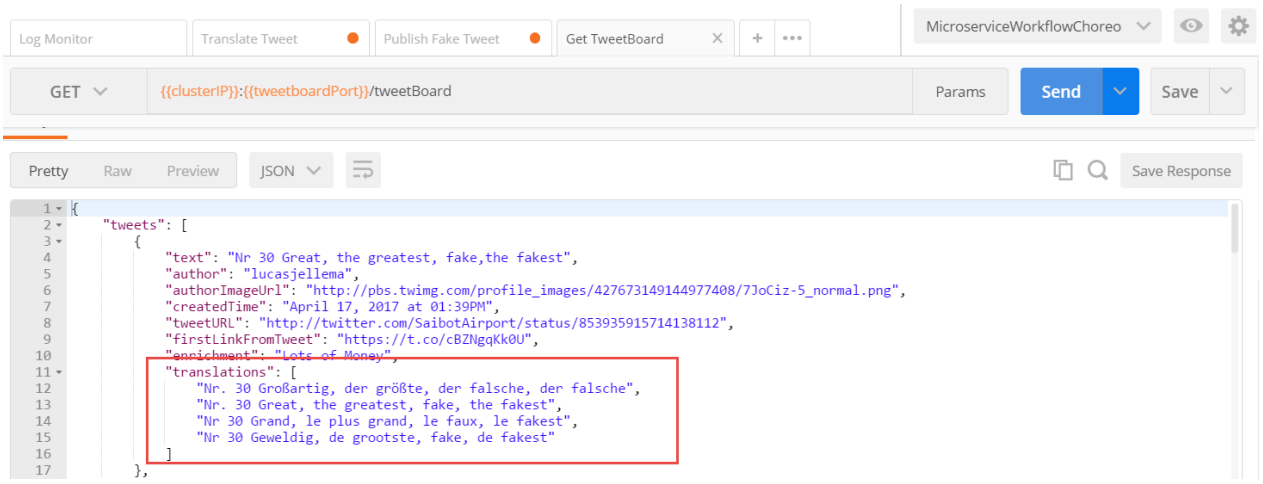
A somewhat updated version of the Tweet Workflow is available in 5-microservices\WorkflowLauncher\ WorkflowLauncherV3.js. In this version, Translation is added to the workflow after validation. This is defined in the *message* variable at the bottom of the program. Take a look at definition of var message in the source of WorkflowLauncherV3.js – and try to understand the updated definition of the routing slip:

```
message =
    {
        "workflowType": "oracle-code-tweet-processor"
        , "workflowConversationIdentifier": "oracle-code-tweet-processor" + new Date().getTime()
        , "creationTimeStamp": new Date().getTime()
        , "creator": "WorkflowLauncher"
        , "actions":
        [{
            "id": "EnrichTweetWithDetails"
            , "type": "EnrichTweet"
            , "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","result":"OK"}; not
        }
        , {
            "id": "ValidateTweetAgainstFilters"
            , "type": "ValidateTweet"
            , "status": "new"  // new, inprogress, complete, failed
            , "result": "" // for example OK, 0, 42, true
            , "conditions": [{ "action": "EnrichTweetWithDetails", "status": "complete", "result": "OK" }]
        }
        , {
            "id": "TranslateTweetsIntoPopularLanguages"
            , "type": "TranslateTweet"
            , "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": "TranslateTweetsIntoPopularLanguages", "status": "complete", "result": "OK" }]
        }
        ]
        , "audit": [
            { "when": new Date().getTime(), "who": "WorkflowLauncher", "what": "creation", "comment": "initial creation of workflow" }
        ]
```

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.



```
```