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

1st June 2017

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

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

You can get access to the sources for the practices from the GitHub repository: <https://github.com/lucasjellema/microservices-choreography-kubernetes-workshop-june2017> .

# 1. Prepare a local Kubernetes and Docker environment

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

How your environment is set up depends on your starting point. Are you running a laptop with Windows or MacOS? Does your version of Windows or MacOS offer native support for Docker? Do you have Linux as your operating system?

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

* Download & Install the latest [Docker Toolbox](https://www.docker.com/products/docker-toolbox) (this is not the same thing as just installing Docker)
* Download & install [VirtualBox](https://www.virtualbox.org/wiki/Downloads) for OS X. Direct link to the binaries [here](http://download.virtualbox.org/virtualbox/5.1.14/VirtualBox-5.1.14-112924-OSX.dmg) (minikube relies on some of the drivers)
* Install Kubectl (command line tool for kubernetes)

See below for details on the installation.

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

### Linux:

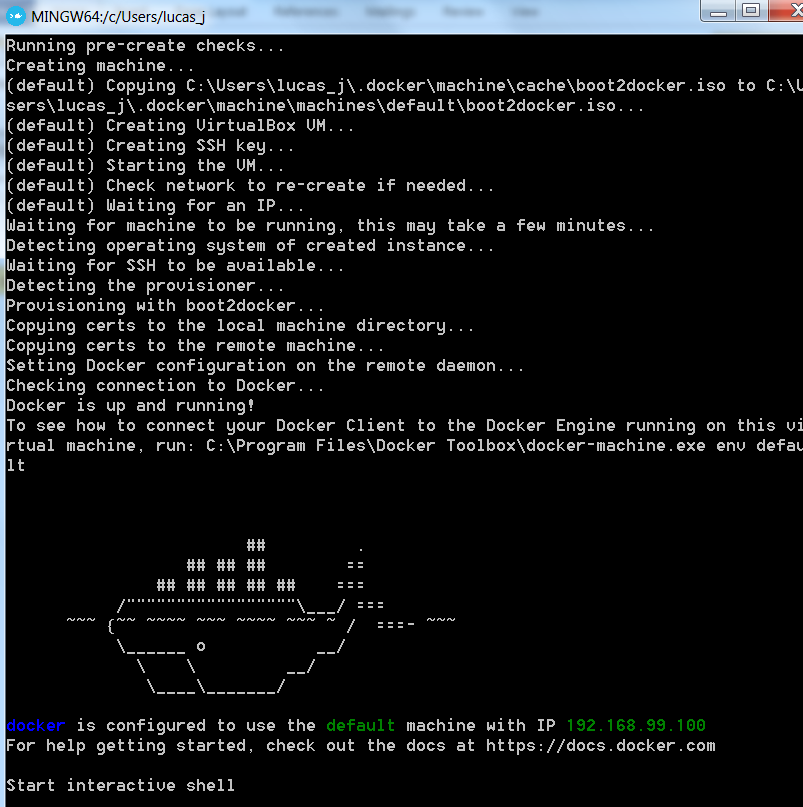
Go to <https://www.docker.com/community-edition> and get the installer for your Linux distribution. Run the installer to set up Docker in your environment.

For Kubernetes – and specifically kubectl - follow instructions at: <https://kubernetes.io/docs/tasks/tools/install-kubectl/> .

### Windows:

VirtualBox: go to the VirtualBox Downloads page: <https://www.virtualbox.org/wiki/Downloads> . Download the latest installer for Windows. Run the installer to install VirtualBox.

Docker Toolbox: <https://docs.docker.com/toolbox/toolbox_install_windows/> (the steps: download the installer, run the installer, run the Docker Quickstart Terminal window to create the *default* machine)



For Kubernetes – and specifically kubectl - follow instructions at: <https://kubernetes.io/docs/tasks/tools/install-kubectl/> .

Download two things:

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

### MacOS:

VirtualBox: goto the VirtualBox Downloads page: <https://www.virtualbox.org/wiki/Downloads> . Download the latest installer for MacOS. Run the installer to install VirtualBox.

Docker Toolbox: <https://docs.docker.com/toolbox/toolbox_install_mac/> (the steps: download the installer, run the installer, run the Docker Quickstart Terminal window to create the *default* machine)

For Kubernetes – and specifically kubectl - follow instructions at: <https://kubernetes.io/docs/tasks/tools/install-kubectl/> .

### 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/>

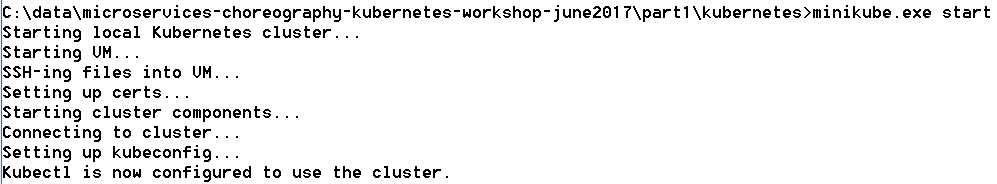
Minikube on Windows7: <https://quip.com/1TYDAdJowAgJ>

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

## Run Minikube Single Node Cluster

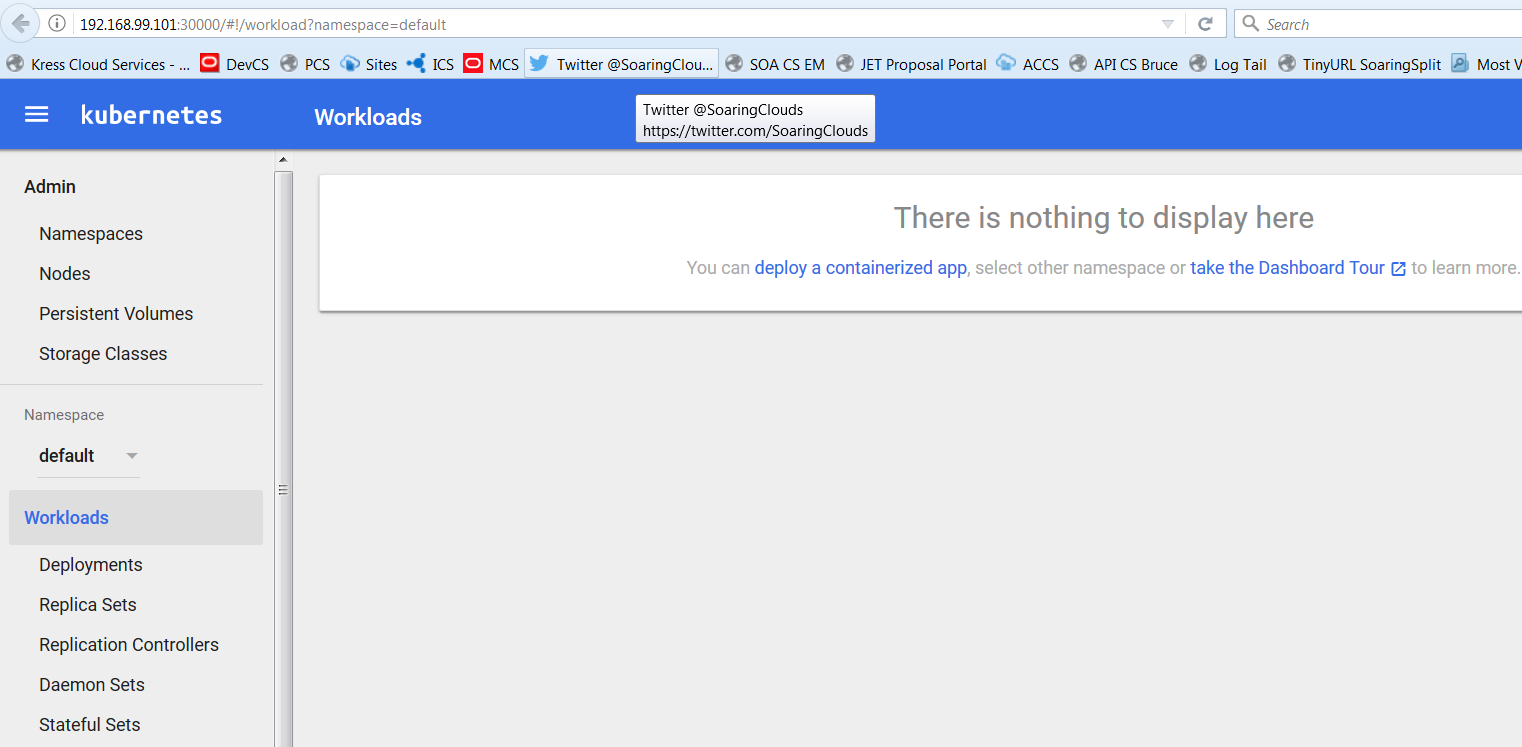
To run minikube – and have the one-node cluster initialized (in a VirtualBox VM):

minikube start

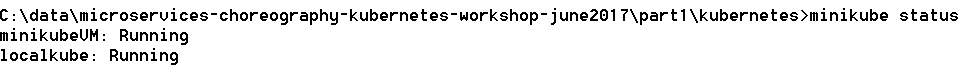


minikube dashboard --url=true





minikube status



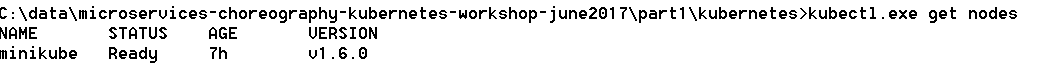
To get the IP address of the Minikube cluster:

minikube.exe ip



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

kubectl.exe get nodes



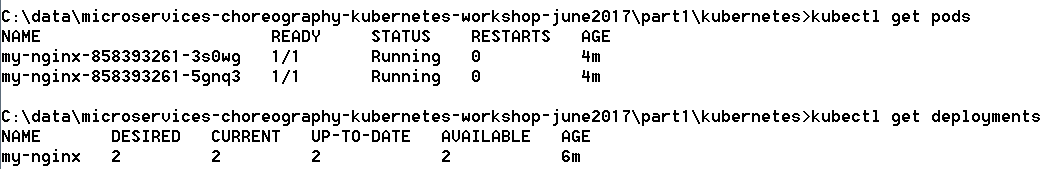
### Run Something on the MiniKube Cluster

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

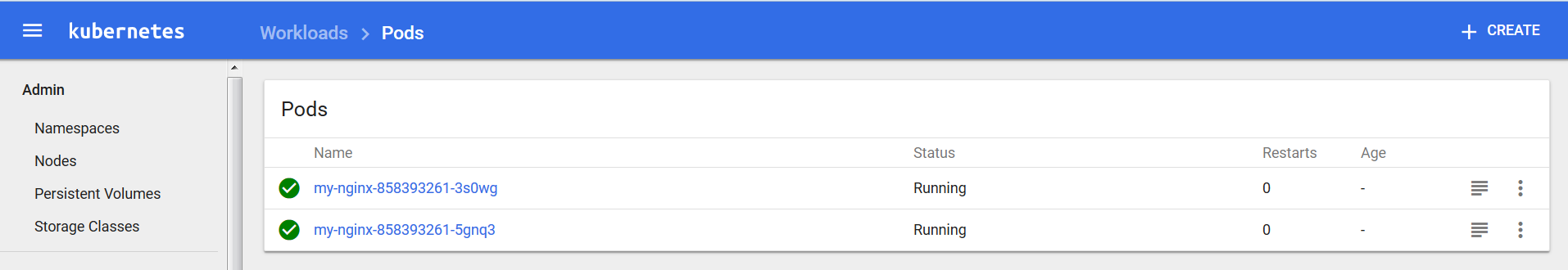
# deploy Docker container image nginx:  
kubectl run my-nginx --image**=**nginx --replicas**=**2 --port**=**80

  
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



In the Dashboard:

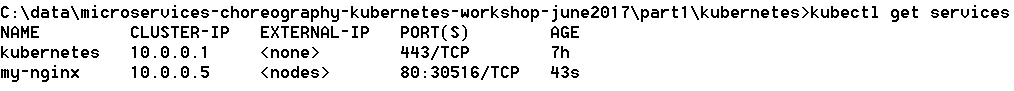


At this moment, the my-nginx containers cannot be accessed from outside the cluster. They need to be exposed through a Service:

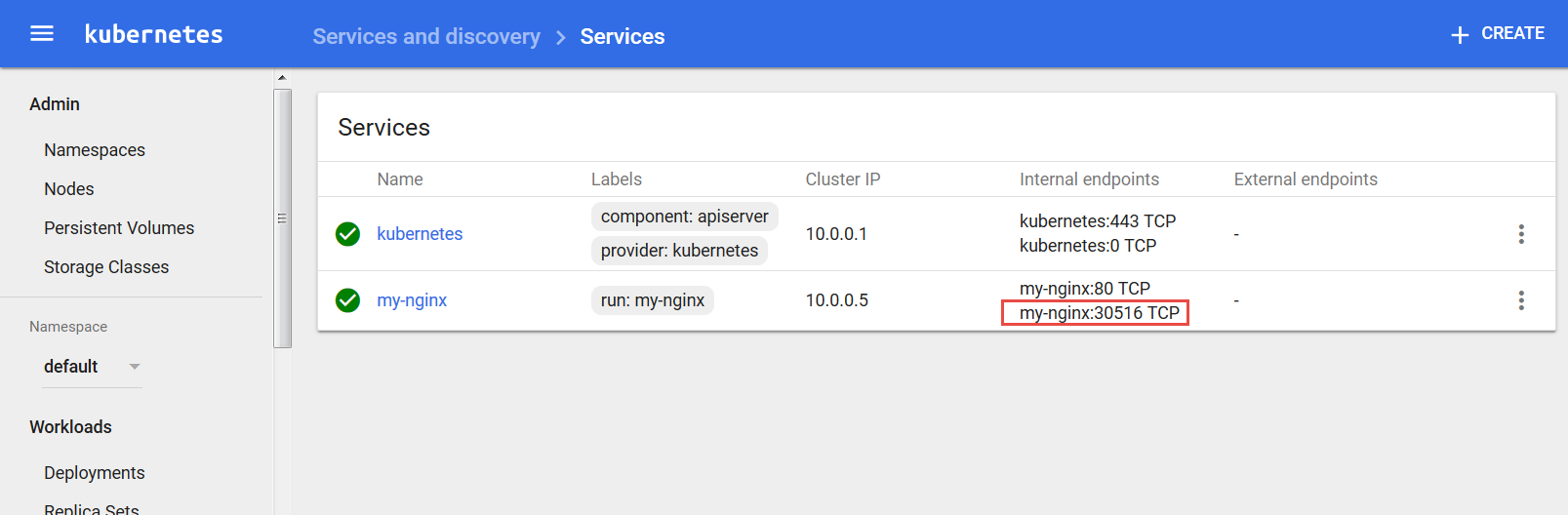
# expose your deployment as a service  
kubectl expose deployment my-nginx --type=NodePort



# check your service is there  
kubectl get services



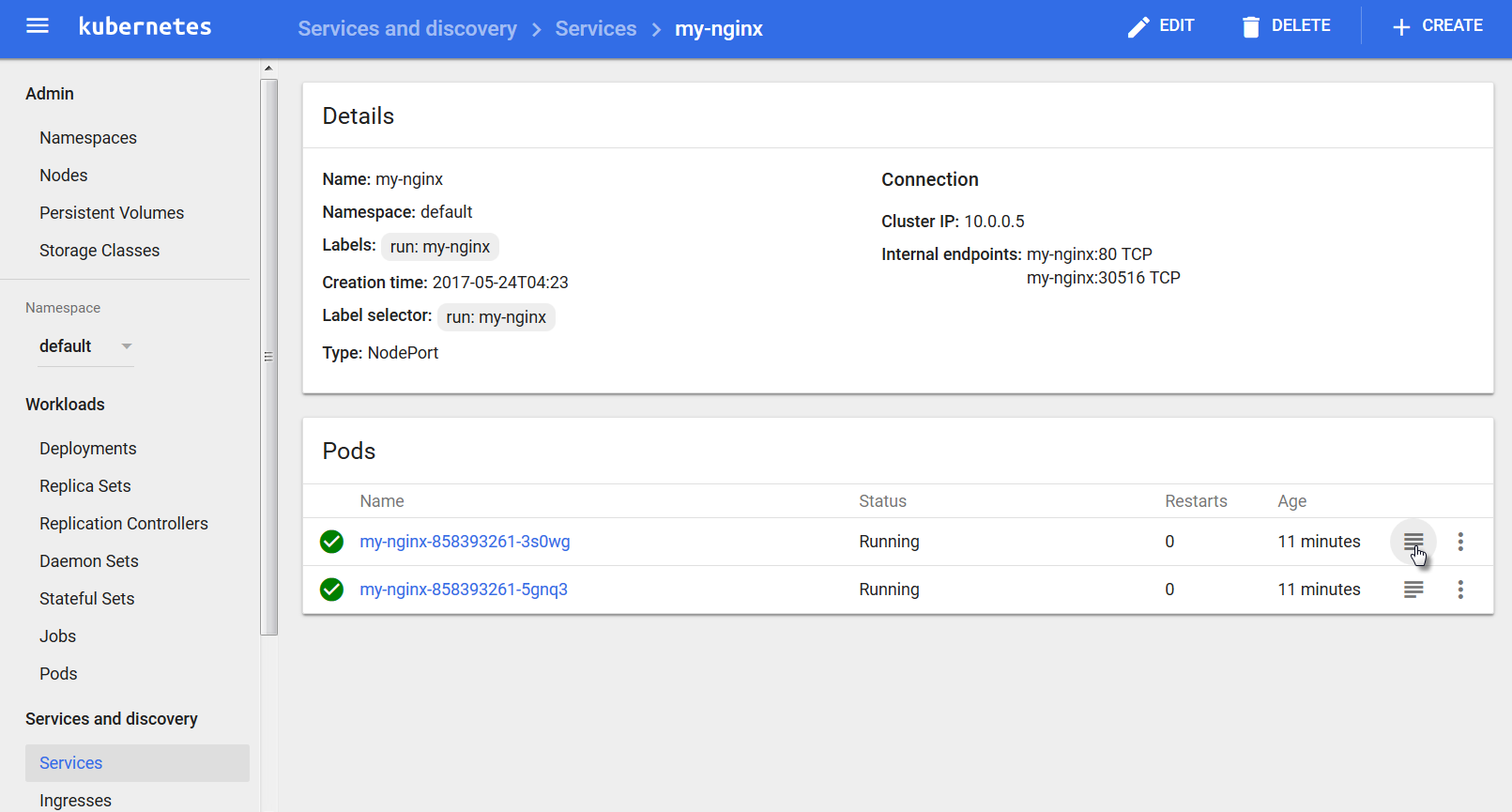
And in the Dashboard:



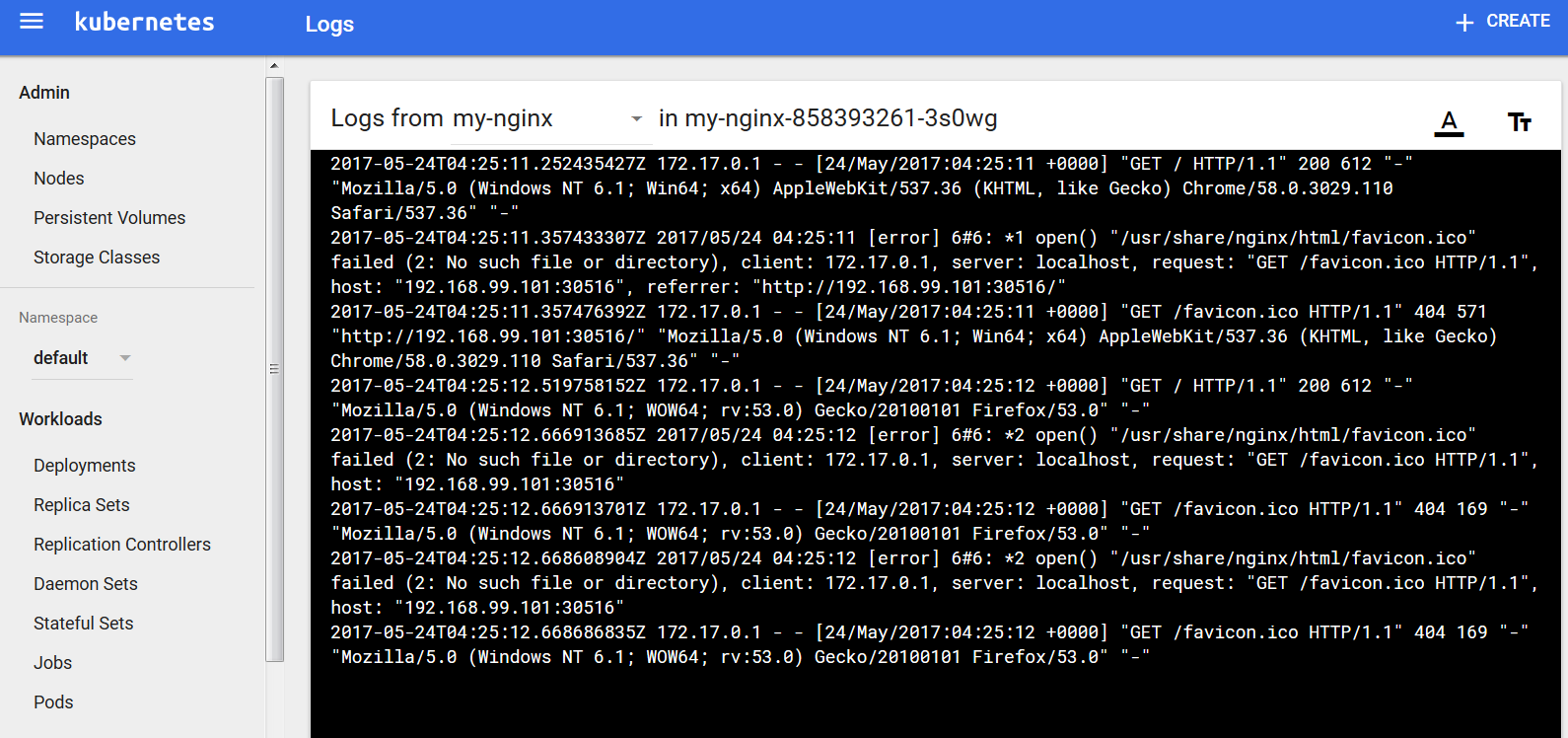
# Access your service from your default browser  
minikube service my-nginx



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:



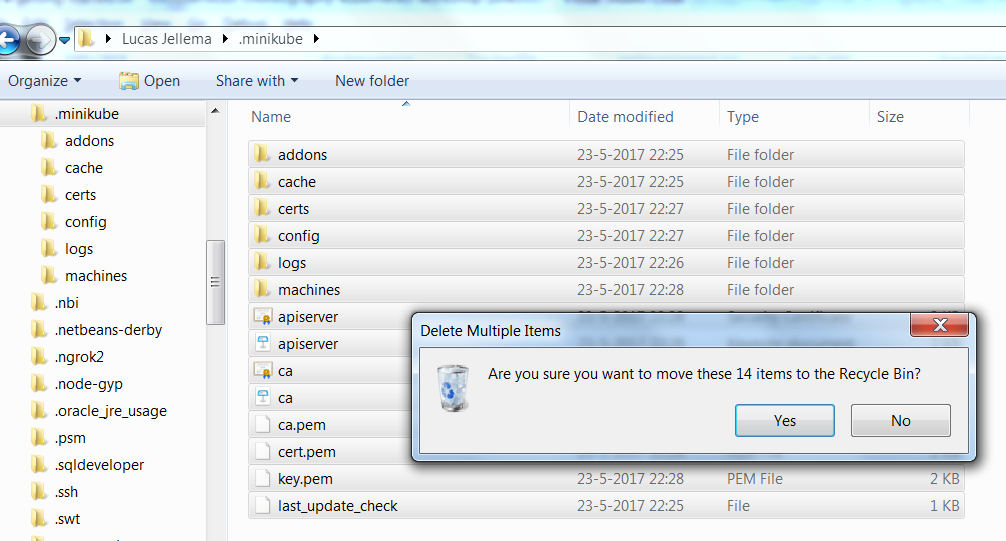
The logging will be shown:



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:



# 2. Run your first Microservice

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

## Run RequestCounter in Docker

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

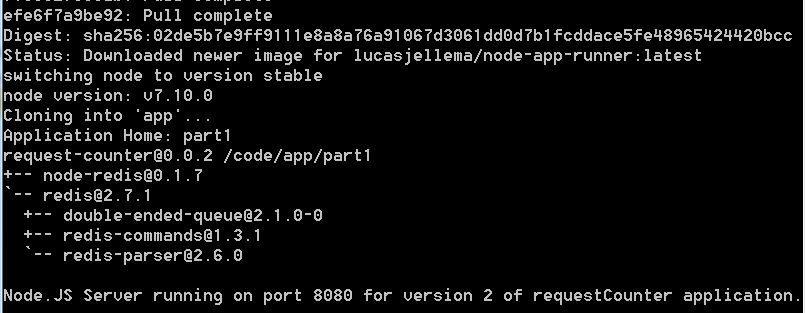
docker-machine ip default

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



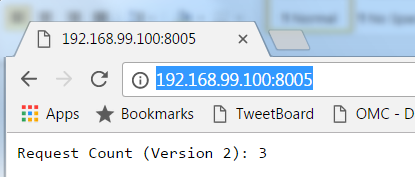
On Docker command line – either in Linux server with Docker installed or in Docker Quickstart Terminal – run a container with the requestCounter.js Node.JS application:

docker run -e "GIT\_URL=https://github.com/lucasjellema/microservices-choreography-kubernetes-workshop-june2017" -e "APP\_PORT=8080" -p 8005:8080 -e "APP\_HOME=part1" -e "APP\_STARTUP=requestCounter-2.js" lucasjellema/node-app-runner



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

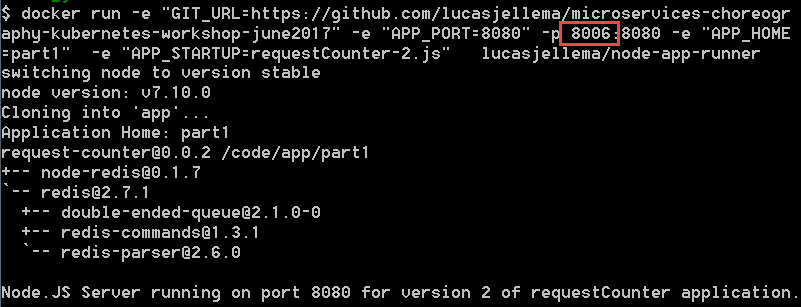
http://192.168.99.100:8005



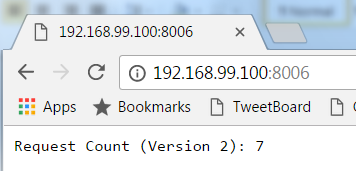
Execute the Docker run command again, with a small twist: change 8005 to 8006:

docker run -e "GIT\_URL=https://github.com/lucasjellema/microservices-choreography-kubernetes-workshop-june2017" -e "APP\_PORT=8080" -p **8006**:8080 -e "APP\_HOME=part1" -e "APP\_STARTUP=requestCounter-2.js" lucasjellema/node-app-runner

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



http://192.168.99.100:8006

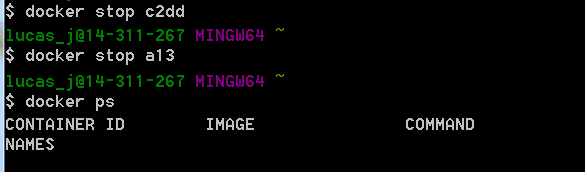


Use docker ps to check on the two containers currently running:



Stop these containers using

docker stop <first three characters of container id>



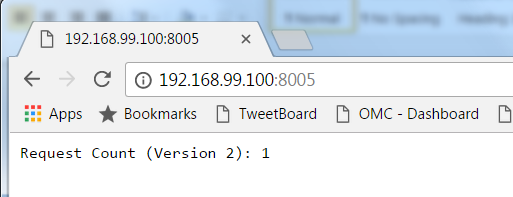
Then use docker run to start a fresh container :

docker run -e "GIT\_URL=https://github.com/lucasjellema/microservices-choreography-kubernetes-workshop-june2017" -e "APP\_PORT=8080" -p 8005:8080 -e "APP\_HOME=part1" -e "APP\_STARTUP=requestCounter-2.js" lucasjellema/node-app-runner

Access the microservice offered by this container once more.

http://192.168.99.100:8005

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



## Run and Expose the microservice on Kubernetes

Run one on Kubernetes

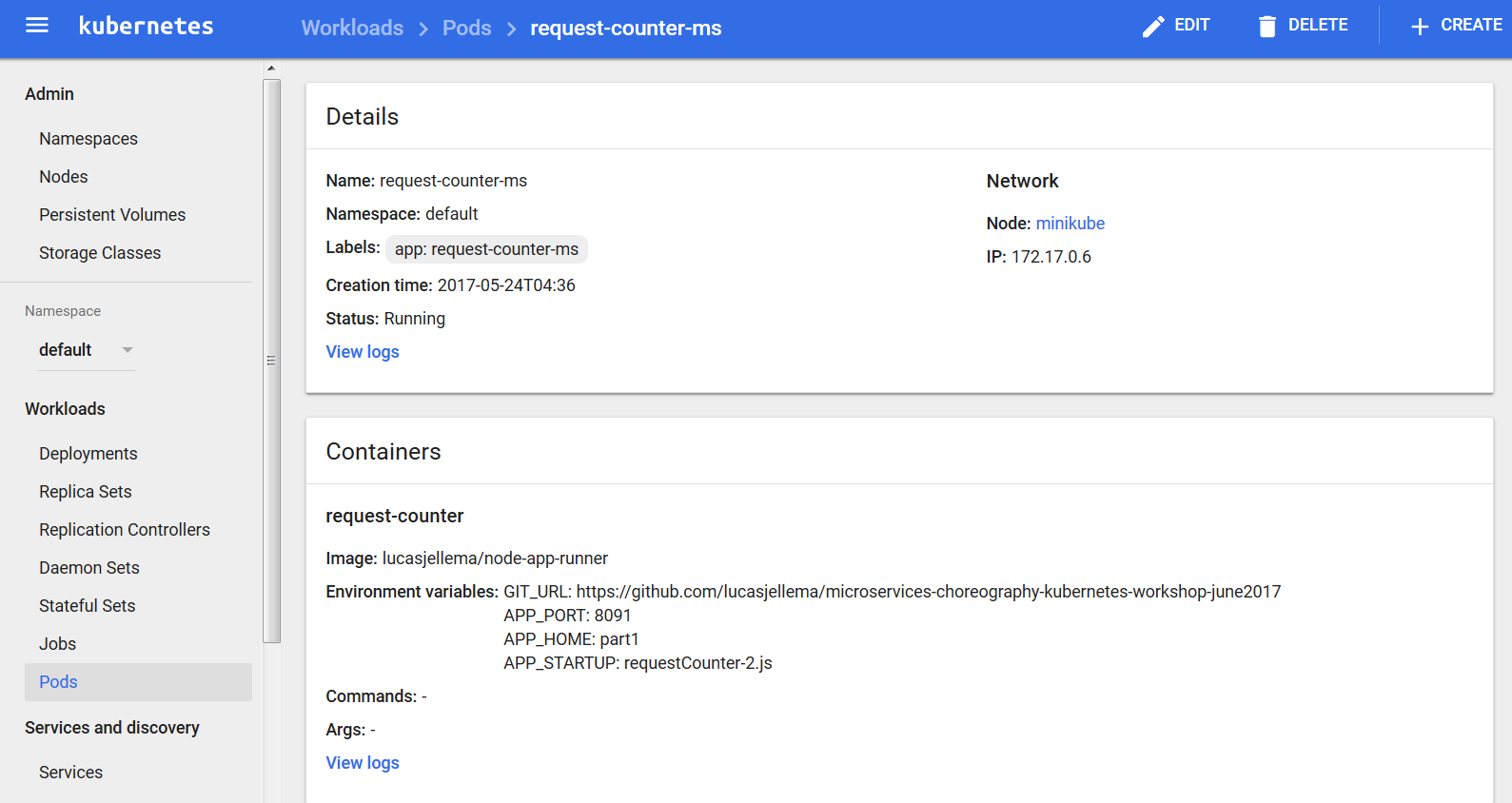
From workshop directory part1/kubernetes, run

kubectl create -f pod.yaml

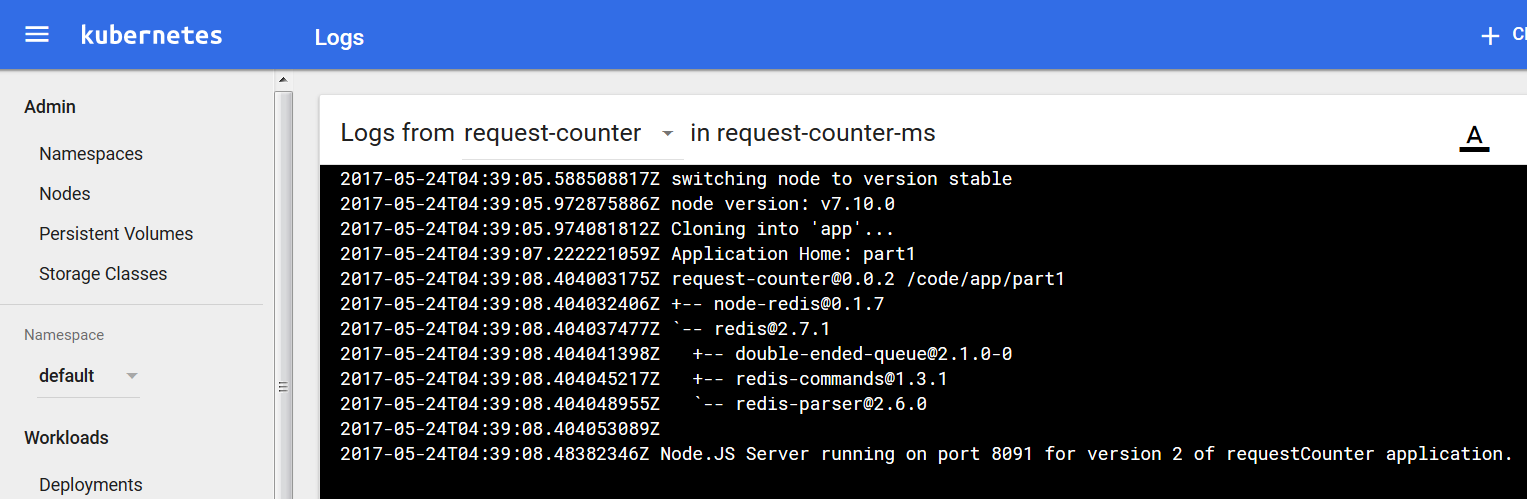


This will create a Pod according to the specification in the file pod.yaml. Open this file and review the specification for the Pod. Crucial elements are the name of the Pod, the name of the underlying Docker Container Image and the values of the environment variables to be passed into the container.

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):



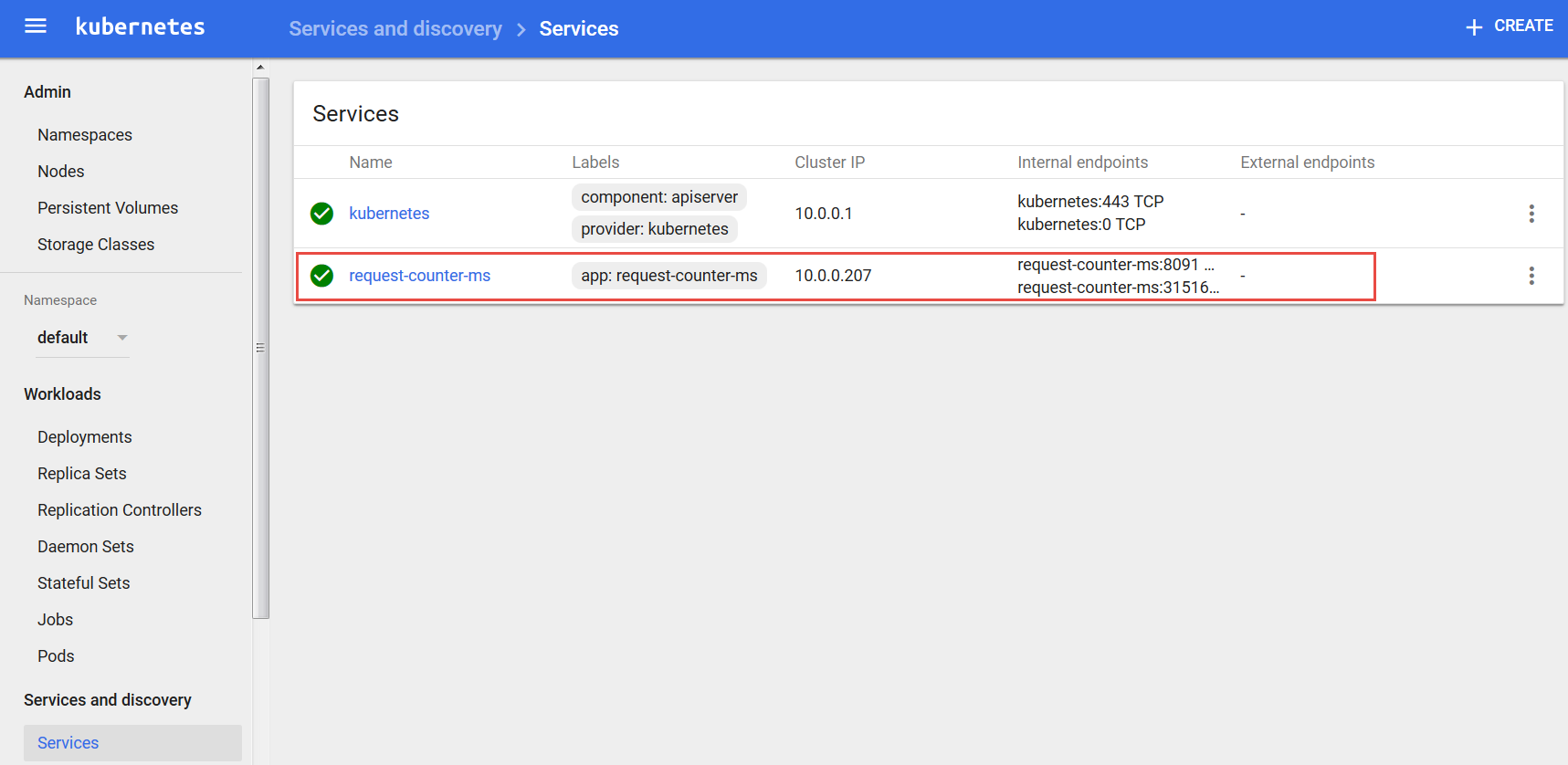
and verify the logging:



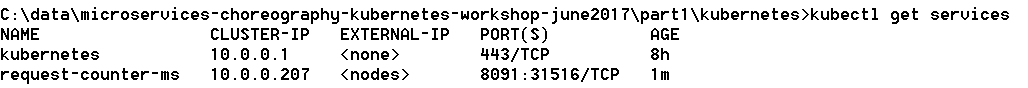
Expose the Pod using a Service to consumers outside the cluster:

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





Using kubectl get services to inspect the service:

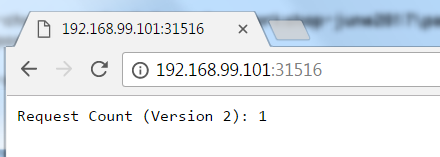


Also to get the url for accessing the service created for pod request-counter-ms:

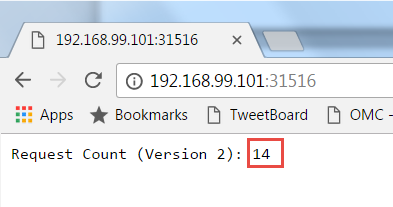
minikube.exe service --url=true request-counter-ms



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



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



Delete the service and pod you have created for request-counter-ms:

## Handle Scaling Up and Down

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

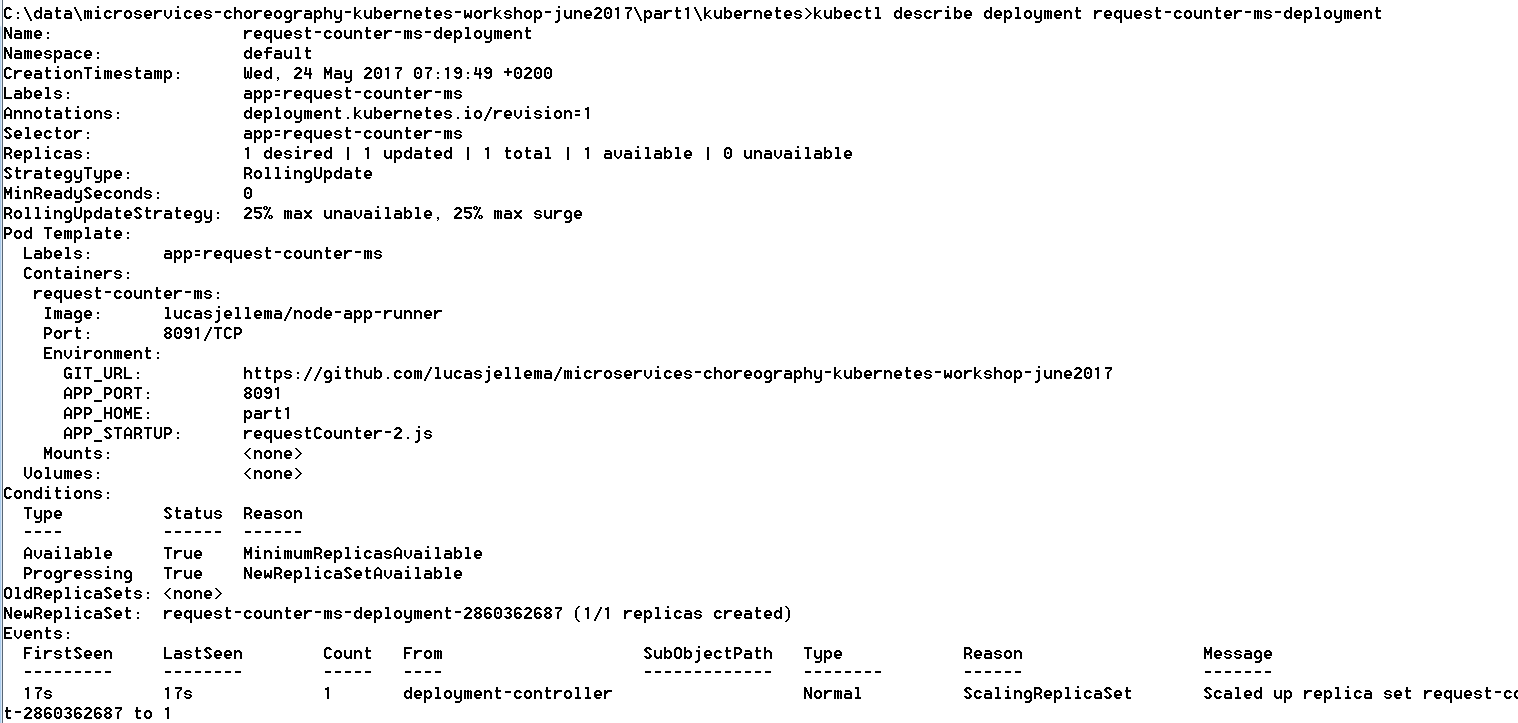
Let’s run the request counter microservice again, through a deployment object – as described by file deployment.yaml.

kubectl create -f deployment.yaml

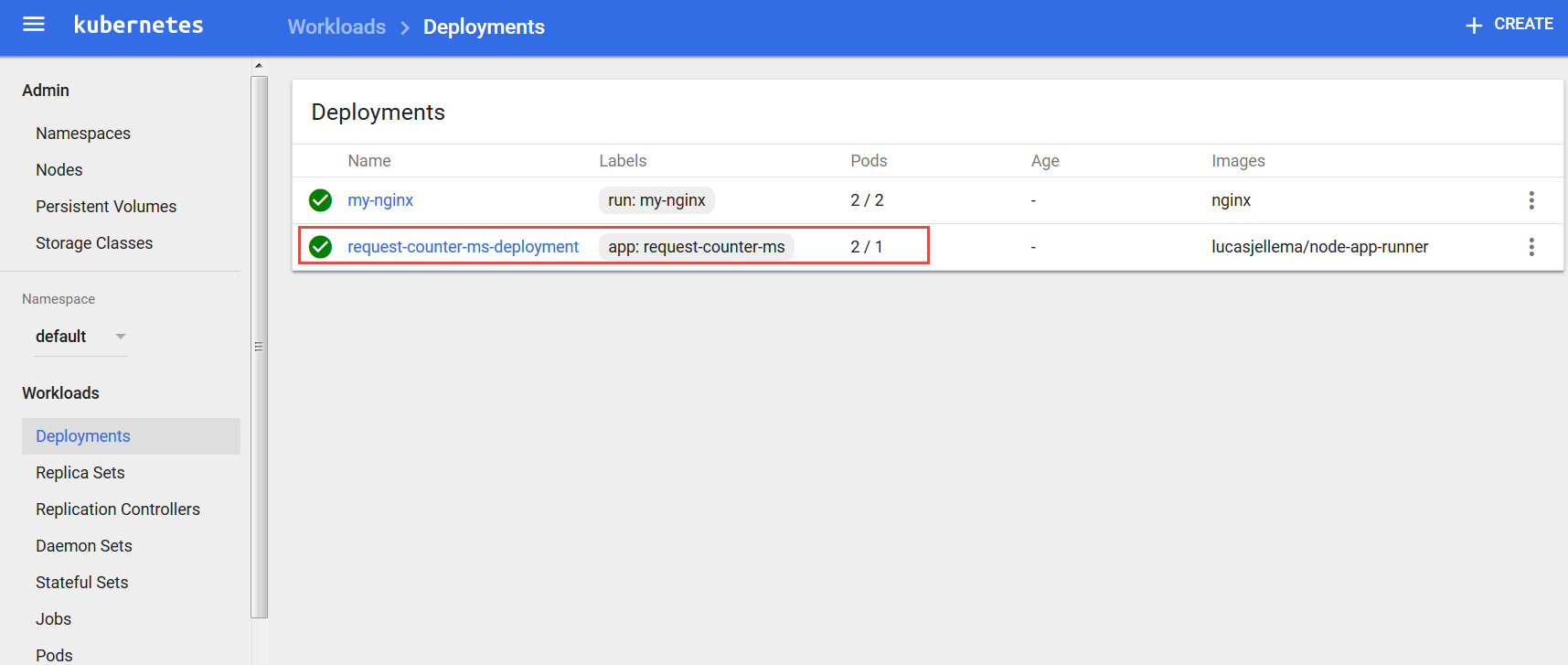


Display information about the Deployment:

kubectl describe deployment request-counter-ms-deployment



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

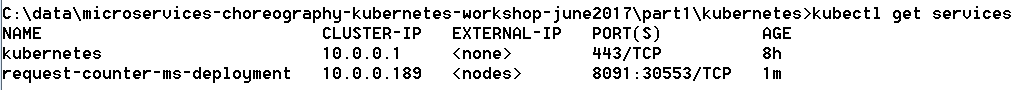


Instead of exposing a single Pod through a Service we can also expose the Deployment. Do so now, using:

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

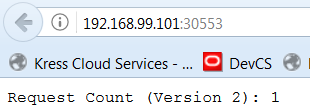


With kubectl get services:



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

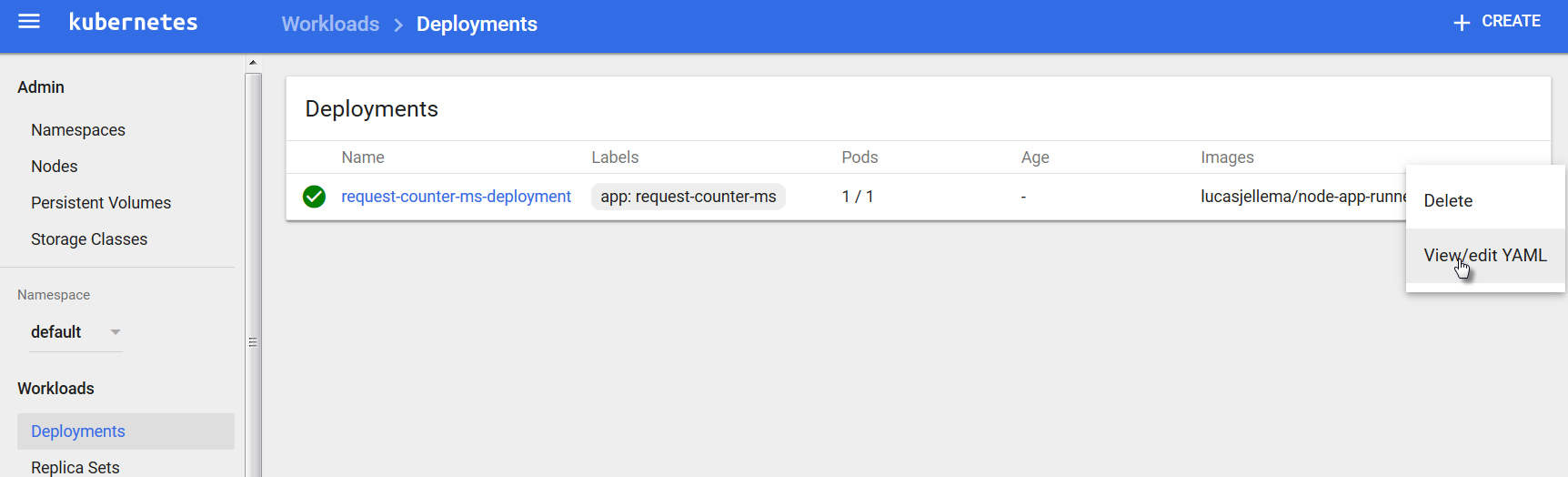
minikube.exe service request-counter-ms-deployment



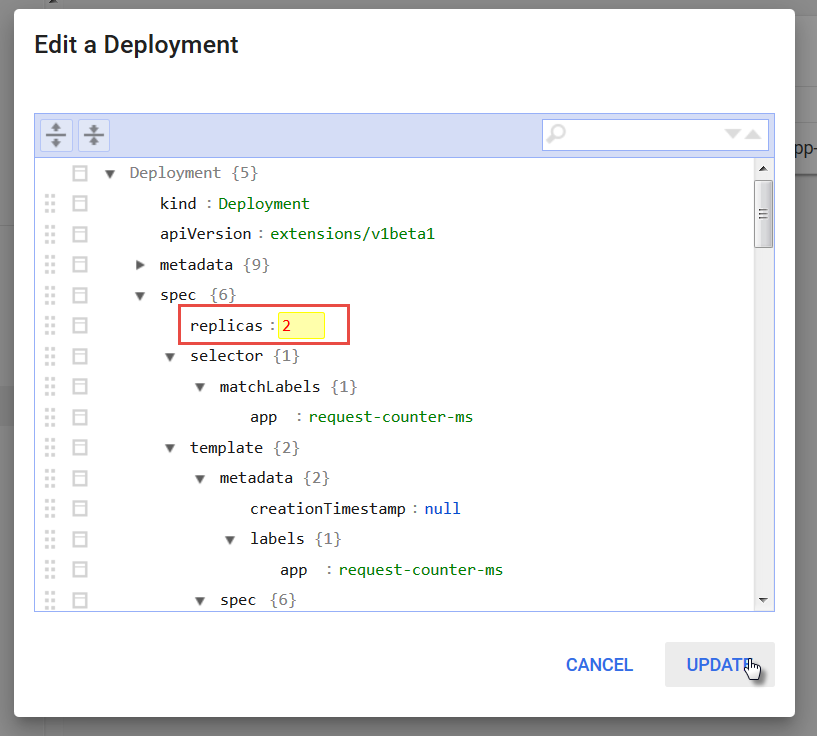
Hit the url a few more times, increasing the counter as you go.

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

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

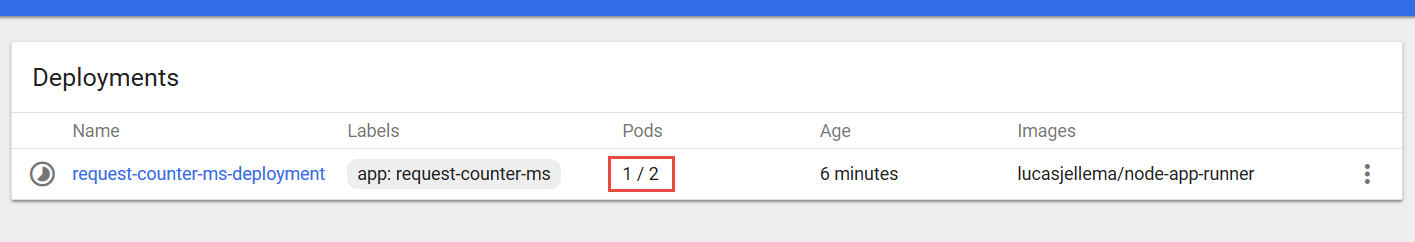


Change the value of the attribute replicas from 1 to 2:

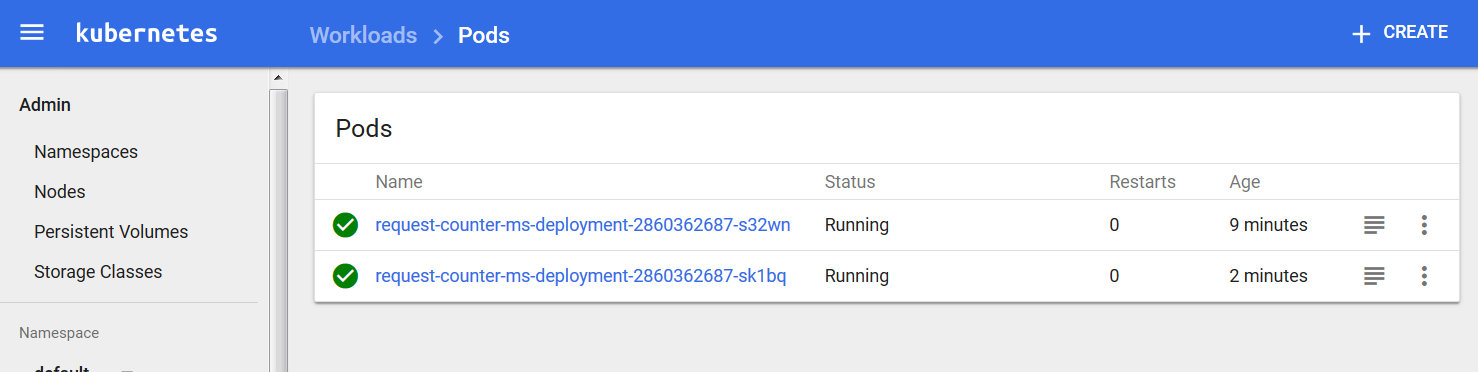


and click on Update.

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



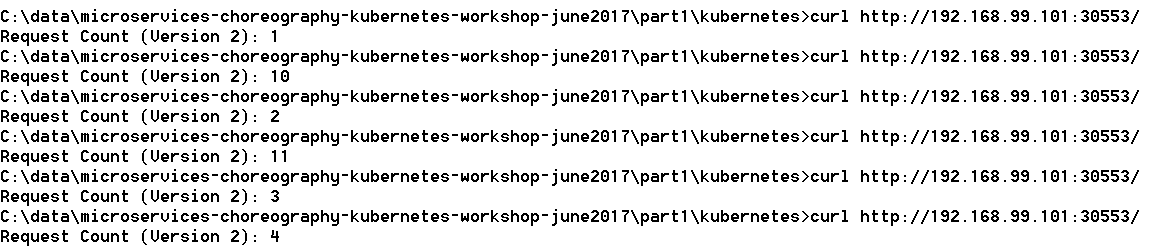
After a little while, two pods are running:



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>

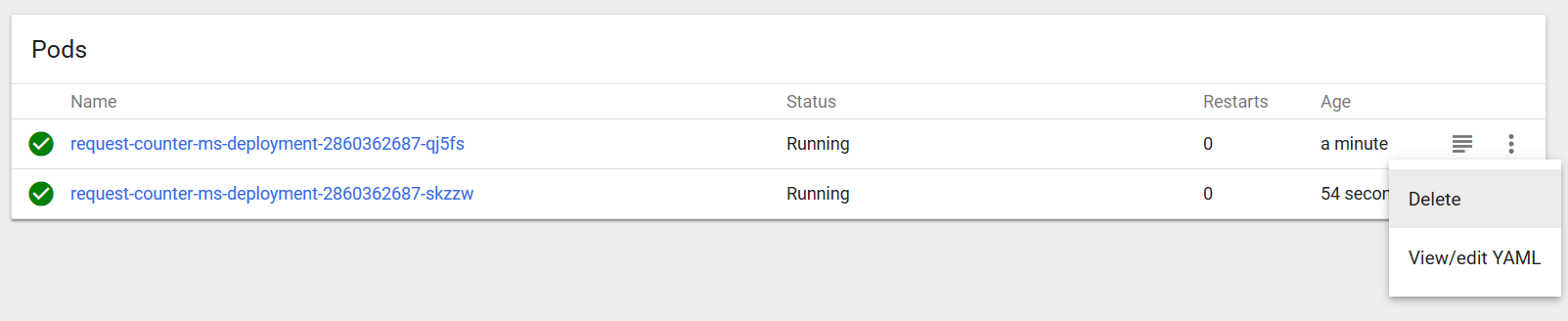
Your results will be similar to the following:



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

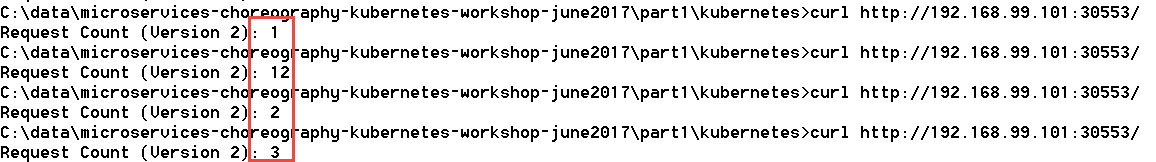
Note how Kubernetes takes care of load balancing for us. We access a endpoint exposed from the cluster, and traffic is distributed over the available Pods.

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



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:



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

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

Add a second pod with Redis in it

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



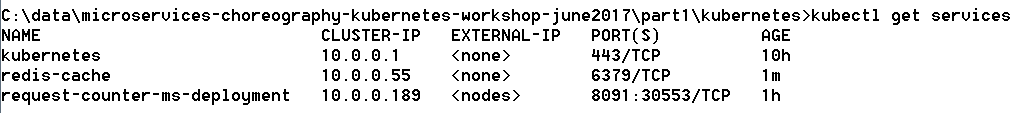
Expose redis within cluster:

kubectl expose deployment redis-cache --type=ClusterIP



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

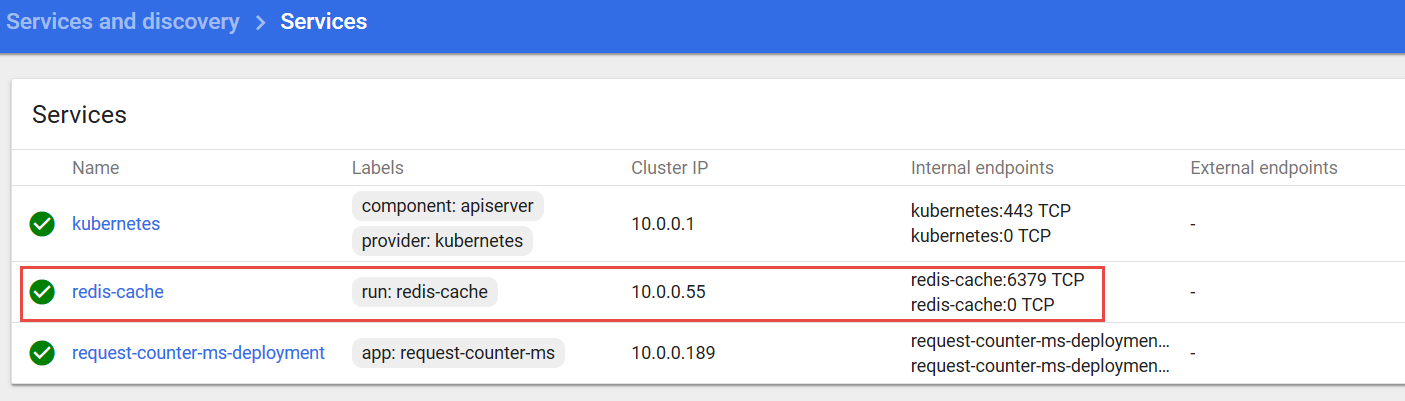
kubectl get services



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

minikube.exe service redis-cache

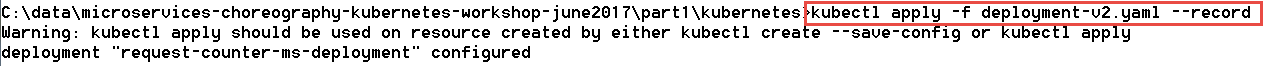
produces no output.



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



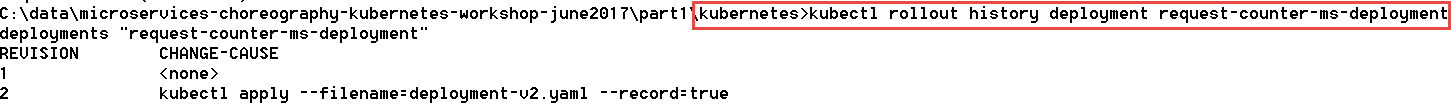
Check on the status of the rollout of the change:

kubectl rollout status deployment request-counter-ms-deployment

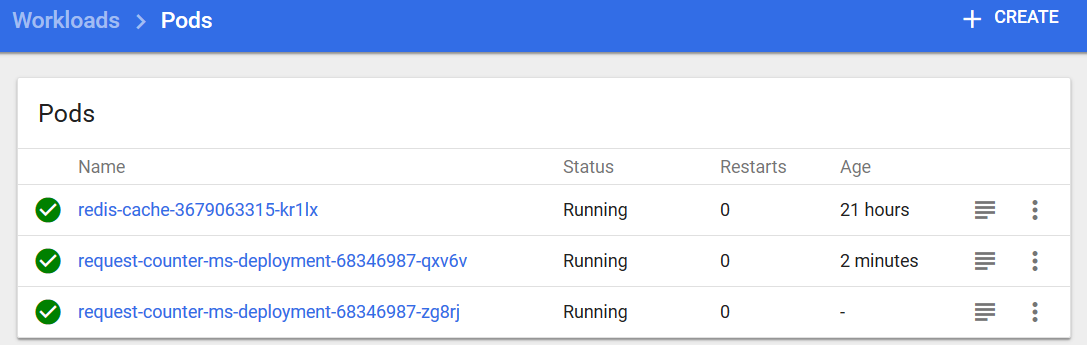


We can inspect the history of a specific deployment and see which changes have been applied to it:

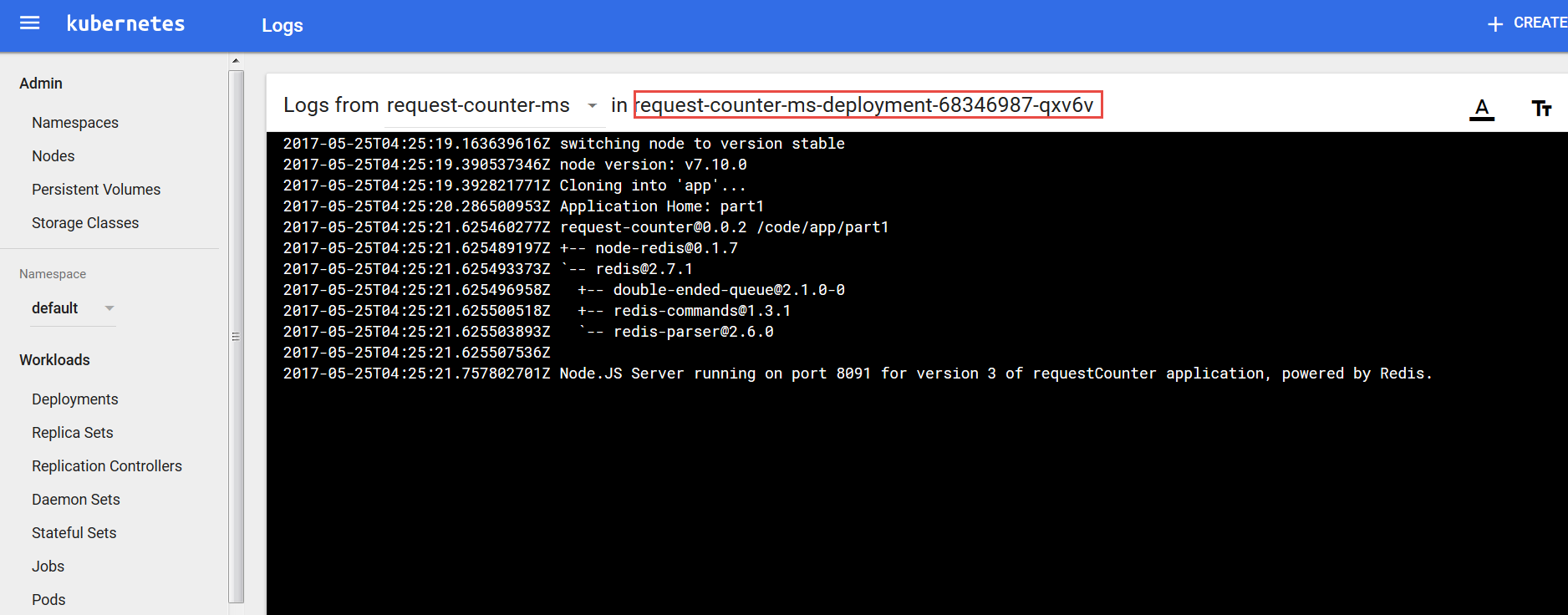
kubectl rollout history deployment request-counter-ms-deployment



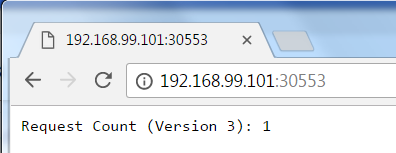
In the dashboard, if we are quick we can see the pods being restarted. If we are not so fast, we can see both pods running (again) – having been restarted fairly recently:



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

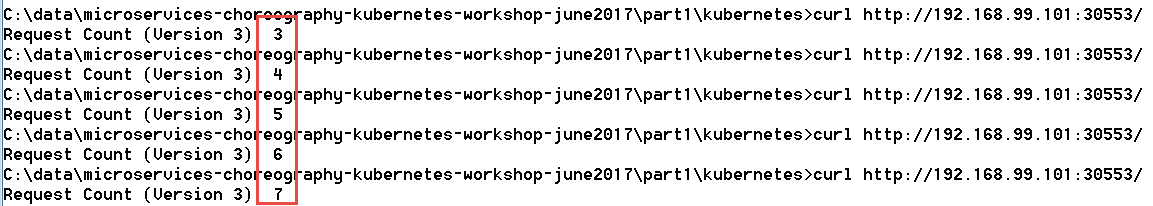


We can access the microservice (again) in the browser:



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

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



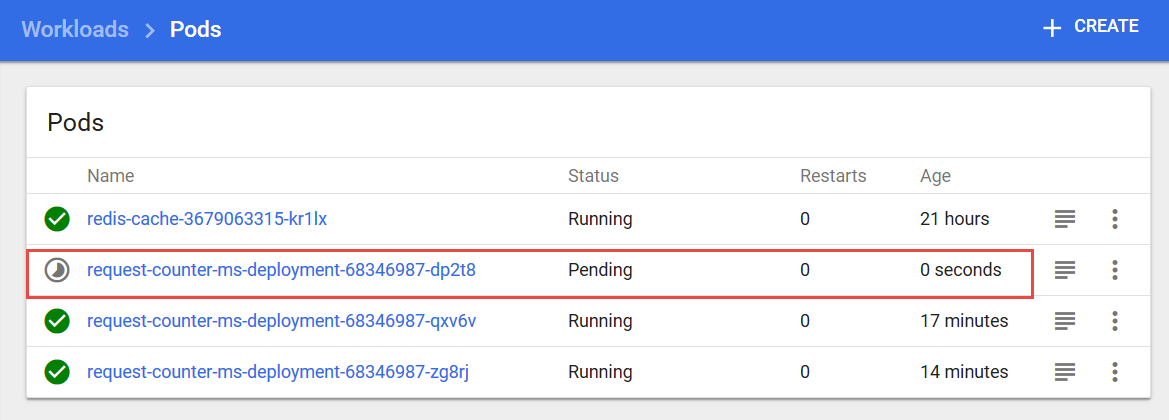
This time round, we do not get two counters incrementing in the background. Just a single incrementing value. And supposedly, that value is stored in the Redis cache where both instances access and manipulate it.

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)

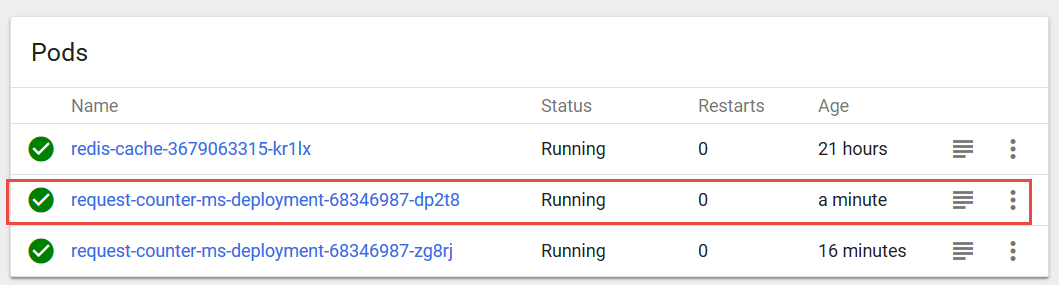
kubectl delete pod request-counter-ms-deployment-68346987-qxv6v --grace-period=60



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

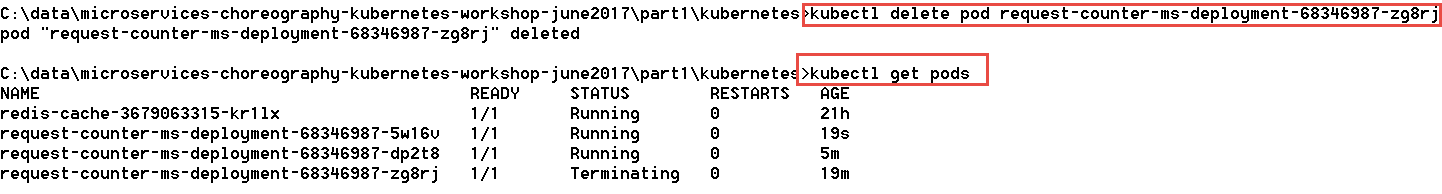


and a little bit later the Pod we asked to be deleted is gone:

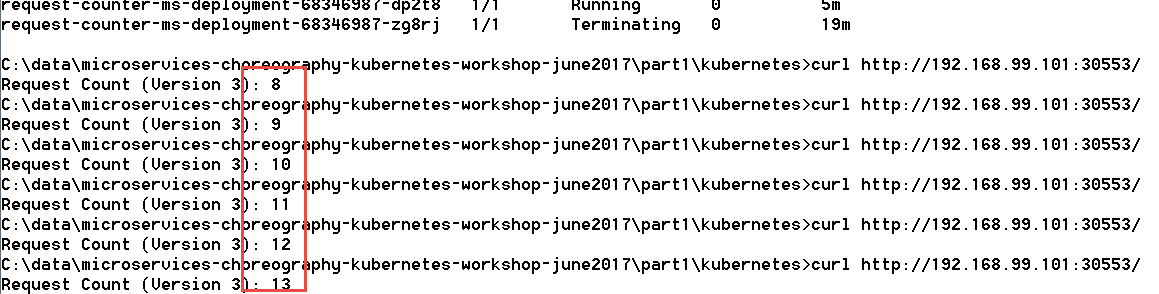


We can also delete our other pod, to be sure any state lingering in the original pods is gone:

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



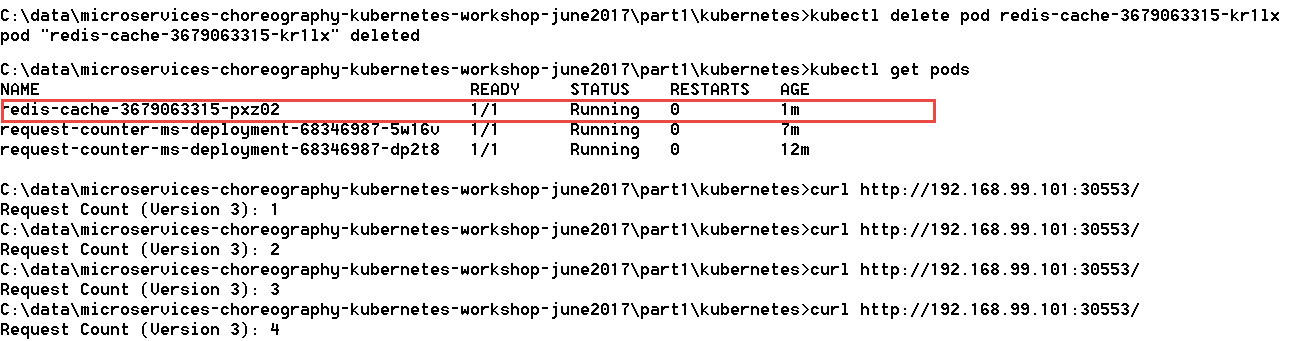
Again, we see a new pod running and the old instance being terminated. At this point, both original pods are gone. If we access the request counter microservice, we will find that the counting continues where it had left off:



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:



### 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/>