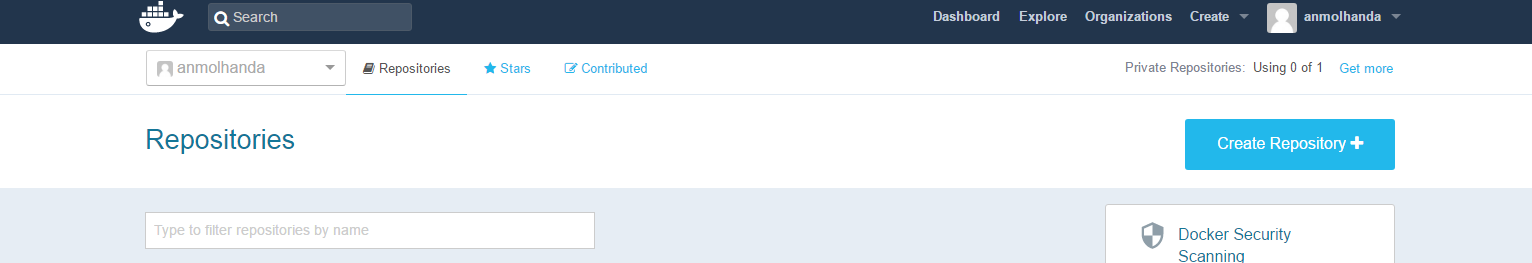
**Kubernetes Cluster App Deployment**

1. First, create a node application suppose the product-master-service and add a DockerFile to it which will help in containerization of this node.js service.   
   You can see the whole code along with the Dockerfile at this link:

<https://github.com/handaanmol/product-service>

1. After this, create a linux vm or if you already have, connect to it via putty and install git and configure it accordingly and clone the above repository.

After this, go to <https://hub.docker.com>, create your account and remember your credentials



1. Install docker in the machine where you are planning to build the image and further from where you are pushing the image to dockerhub. For installing docker, first check the distribution and release of the linux by following command : **lsb\_release –a**

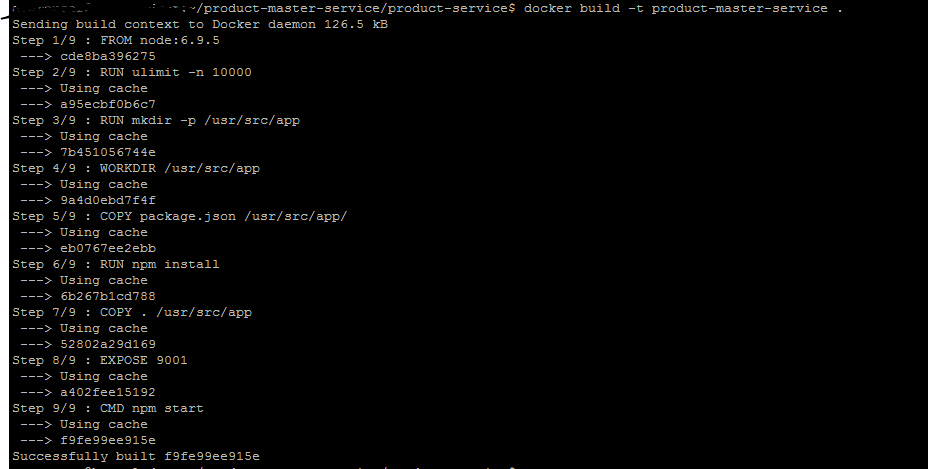
After checking, find the steps to install docker for that distribution and release.

*(The steps to install and run docker on Ubuntu 16.04 are given here:* [*https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-16-04*](https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-16-04)*)*

1. Don’t follow steps 4 to 7 if you are not planning to build your own docker image and push it to docker hub. You can directly go to step 8 and pull the image created by me directly available on docker hub and use it to run on Kubernetes Cluster.
2. Once that’s done, go back to the VM and go to the directory inside which you cloned the code

And run this command

**docker build –t product-master-service .**



It will build the image for you in the VM

1. After this run the command **docker images** and you will see your docker image listed there.



1. After this , run the following commands in order to get authenticated by dockerhub from VM

**export DOCKER\_ID\_USER="your dockerhub username "**

**docker login**

(It will ask for your credentials to verify and then you will be authenticated for dockerhub from the VM)

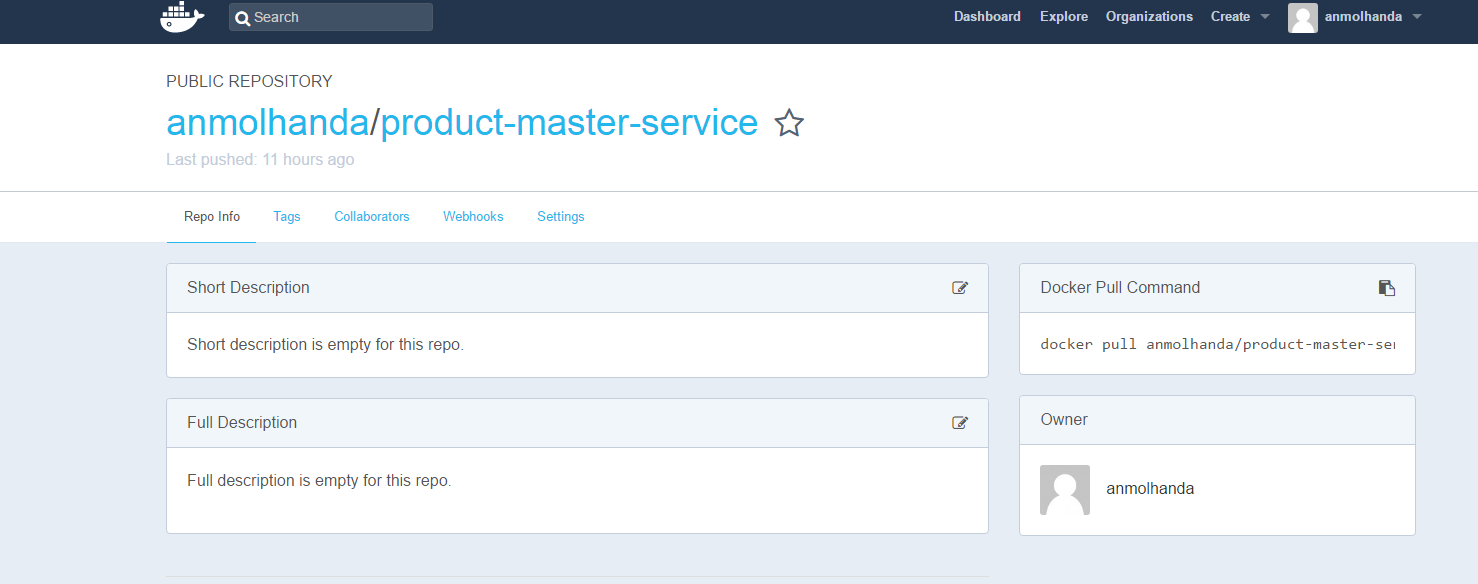
1. Now tag the image which we build before pushing it with the following command:

**docker tag product-master-service $DOCKER\_ID\_USER/ product-master-service**

And once the above command runs successfully, we will push the image to our dockerhub Registry by writing the following command

**docker push $DOCKER\_ID\_USER/ product-master-service**

This command will push the image to your dockerhub account from where it can be accessed by anyone as shown in the below screenshot



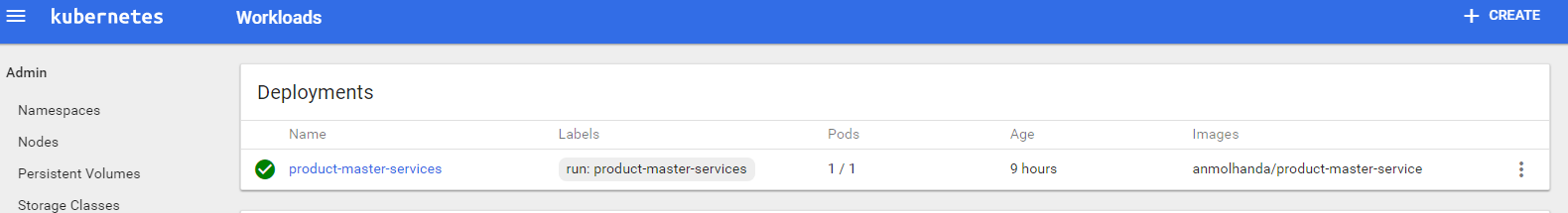
1. After pushing image, just check whether the image is running as docker container fine or not by just writing the following command in the VM

**docker run -d -p 9090:9001 --name product-master-services anmolhanda/product-master- service**  
(Since I have run the application on port on 9001 so I am exposing it on port 9090 externally outside the docker container here)

If it works fine that means nothing is wrong with your Dockerfile and you can proceed forward with Deployment of this container on Kubernetes. It’s better to delete this container from the VM otherwise it will utilize VM resources continuously. Use **docker rm product-master-service** to remove the container

1. Now from Azure CLI or local cmd with kubectl working , run the following command

**kubectl run product-master-services --image anmolhanda/product-master-service --port=9001**  
 It will start the deployment of the following image on dockerhub on kubernetes with port of node.js containerized application exposed on 9001 port of pod as shown below



Alternatively, write the following command on CLI or CMD to see whether deployment has been done successfully or not

**kubectl get deployments**

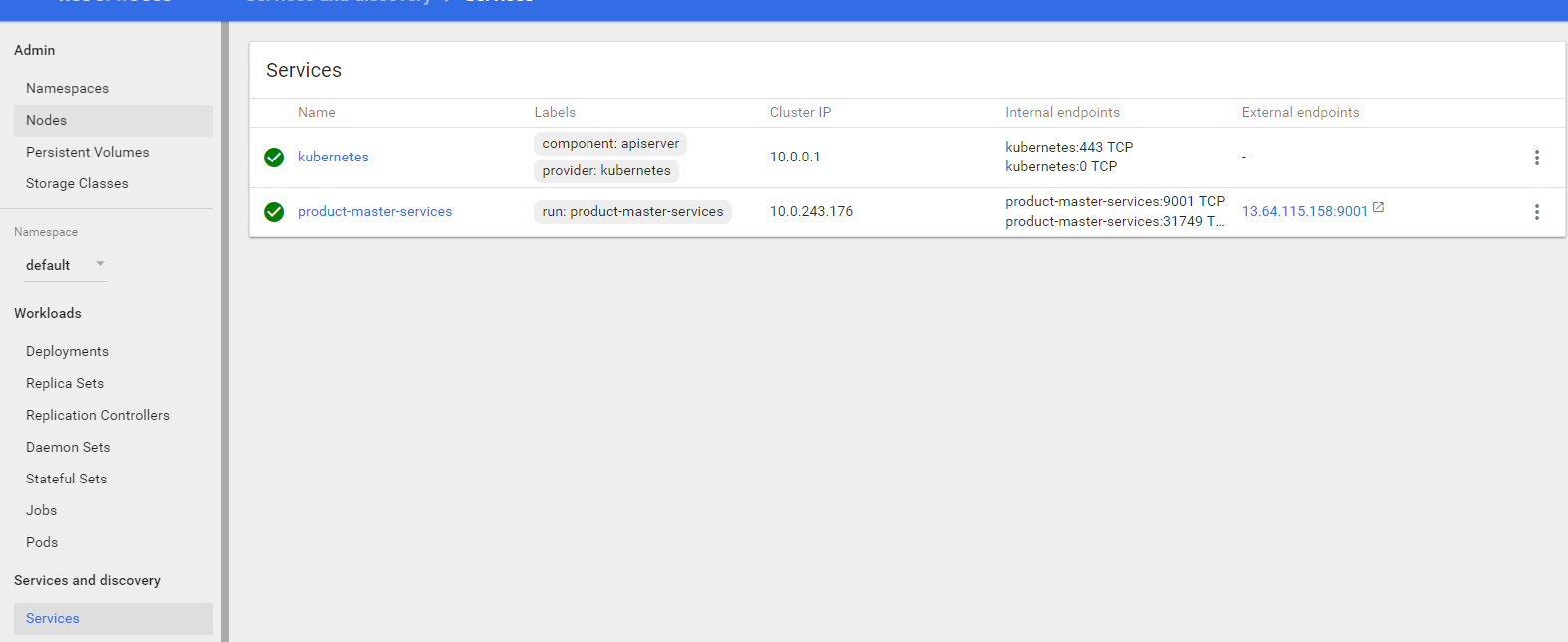


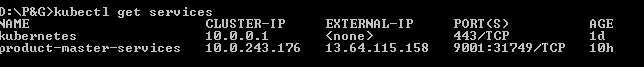
1. Then once you see the above result, run the service for the above deployment using the following command

**kubectl expose deployment product-master-services --type=LoadBalancer**

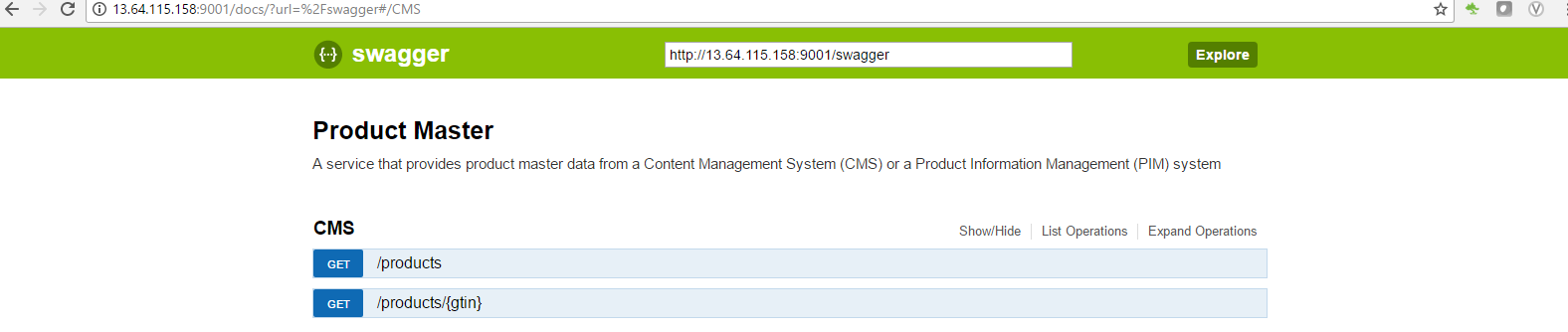
This command causes Kubernetes to create an Azure load balancer rule with a public IP address

You will observe that service gets run successfully on dashboard and also a public IP address is given to access the service as shown below



Also, through CLI or CMD, use the command **kubectl get services** to view the same thing as follows:  


1. Try to access the external IP for your API say swagger, you would see the following result.

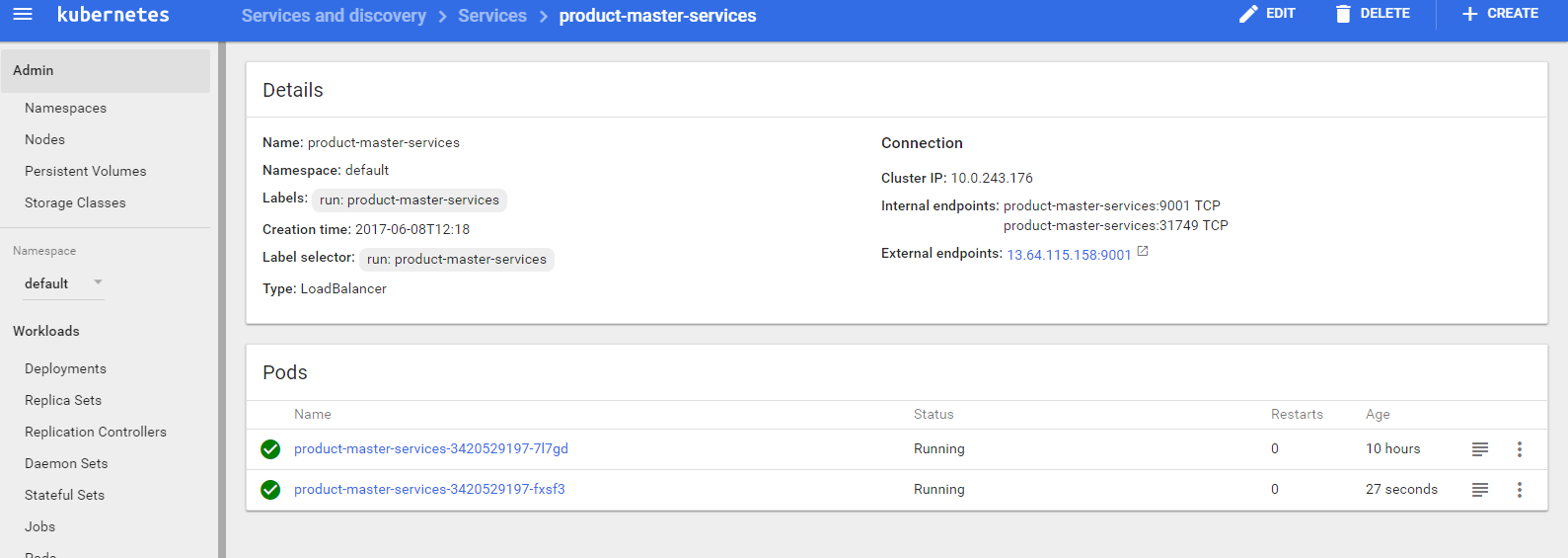


That means your app has been successfully deployed to Kubernetes Cluster

1. Now to scale it to one more instance , use command

**kubectl scale deployment product-master-services --replicas=2**

This command would bring up to replication sets or 2 pods for same service in order to promote higher availability and zero downtime as shown below



1. The application is still available one the same link that is:

<http://13.64.115.158:9001/docs> - Swagger Link

But internally its load balancing the request among the two replicas or pods which are running.

That’s how Kubernetes works.