Kubernetes workshop jFall

Environment

Server	Adres
WLAN	Guest-Valk-Veenendaal
Kubernetes Dashboard	https://104.199.60.253/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard
Workshop server	104.199.53.243
SSH Keys (per user)	Will be given. Login: ssh -i ~/.ssh/user-1-ssh-key user-1@104.199.53.243

Kubectl

`kubectl config view` shows which clusters can be used and which authentication to use. Note that also a namespace is specified.

```
$ kubectl config view
```

`kubectl get nodes` shows which cluster nodes are registered along with its status.

```
$ kubectl get nodes
```

`kubectl get namespaces` shows which namespaces are registered.

```
$ kubectl get namespaces
```

Kubectl has much more to offer check kubectl help for all options.

2. Kubernetes

Running a container on kubernetes

An easy way to test the cluster is by running a simple docker image like the `nginx` one. `kubectl run` can be used to run the image as a container in a pod. `kubectl get pods` shows the pods that are registered along with its status. `kubectl describe` gives more detailed information on a specific resource. Each pod (container) get a unique ip-address assigned wihtin the cluster and is accessible on that througout the cluster, thanks to flannel overlay network. A pod can be deleted using kubectl delete pod/deployment/service <name>. Note that the run command creates a **deployment** (http://kubernetes.io/docs/user-guide/deployments/) which will ensure a crashed or deleted pod is restored. To remove your deployment, use kubectl delete deployment nginx. **(kubectl help is your friend!)** The --port flag exposes the pods to the internal network. The --labels flag ensures your pods are visible. Please use both flags.
Note that your nginx pod may seem to be stuck at "ContainerCreating" because it has to download the image first.

```
$ kubectl run nginx --image=awassink/nginx:v1 --port=80
--labels="run=nginx,visualize=true"
deployment "nginx" created

$ kubectl get pods -o wide
NAME                     READY     STATUS    RESTARTS   AGE       IP
NODE
nginx-1665122148-4amzl   1/1       Running   0          47s
10.1.87.2    10.150.42.100
$ kubectl describe pod nginx-1665122148-4amzl
Name:  nginx-1665122148-4amzl
Namespace: rpi-node-61
Node:  10.150.42.100/10.150.42.100
Start Time: Wed, 19 Oct 2016 02:18:38 +0200
Labels:  pod-template-hash=1665122148
  run=nginx
  visualize=true
Status:  Running
IP:  10.1.87.2
Controllers: ReplicaSet/nginx-1665122148
Containers:
$ kubectl get rs -o=wide
NAME               DESIRED   CURRENT   AGE       CONTAINER(S)   IMAGE(S)
SELECTOR
nginx-1665122148   1         1         3m        nginx
buildserver:5000/rpi-nginx
pod-template-hash=1665122148,run=nginx,visualize=true

$ kubectl get deployment -o=wide
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx     1         1         1            1           4m
```

Now the container is running with kubernetes, the NGINX application is directly accessible via its IP address within the kubernetes cluster. Note that this is an IP address within the cluster network and is not accessible from outside the cluster. Also note that we do not have to specify any port mappings from the container to the host.

```
$ curl http://10.1.87.2/
<html>
<head>
<title>Welcome to nginx!</title>
</head>
<body bgcolor="white" text="black">
<center><h1>Welcome to nginx!</h1></center>
</body>
</html>
```

Exposing containers on kubernetes

Now the pod is running, but the application is not generally accessible. That can be achieved by creating a service in kubernetes. The service will

have a cluster IP-address assigned, which is the IP-address the service is avalailable at within the cluster (`10.0.0.*`). Use the IP-address of your Raspberry Pi node as external IP and the service becomes available outside of the cluster (e.g.10.150.42.103 in my case). Check in your browser that `http://<ip-address-of-your-node>:90/` is available.

```
$ kubectl expose deployment nginx --port=80 --target-port=80
--type=LoadBalancer
service "nginx" exposed

$ kubectl get svc
NAME      CLUSTER-IP   EXTERNAL-IP     PORT(S)   AGE
nginx     10.0.0.102   10.150.42.103   90/TCP    57s
$ curl http://10.150.42.103/
```

The cluster IP address is only accessible within the cluster, therefor go into a container and access the service.

```
$ kubectl exec -ti nginx-1665122148-4amzl bash

# curl http://10.0.0.102/
<html>
<head>
<title>Welcome to nginx!</title>
</head>
<body bgcolor="white" text="black">
<center><h1>Welcome to nginx!</h1></center>
</body>
</html>
```

Service discovery in kubernetes

Kubernetes runs its own service discovery and makes service and pods accessible via DNS.

```
$ kubectl exec -ti nginx-1665122148-4amzl bash

# nslookup nginx

# nslookup kubernetes.default
```

Scaling

The number of pod serving a service can easily be scaled via kubectl. Use `kubectl scale` to do so. Check out the visualizer the moment when you execute the scale command.

```
$ kubectl scale --replicas=3 deployment nginx
deployment "nginx" scaled

$ kubectl get pods -o=wide
NAME                     READY     STATUS              RESTARTS   AGE
IP           NODE
nginx-1665122148-4amzl   1/1       Running             0          13m
10.1.87.2    10.150.42.100
nginx-1665122148-peep7   0/1       ContainerCreating   0          1m
<none>       10.150.42.176
nginx-1665122148-shm28   0/1       ContainerCreating   0          1m
<none>       10.150.42.183
```

Doing a rolling update with Kubernetes (no service downtime)

In order to demonstrate a rolling update, we will use some prepared nginx containers which serve different static html depending on the version. Please remove your current deployment and deploy version 1 of this image, with 4 replicas, exposing port 80 on the pods (tip: if you don't remove the service exposing your raspi's port 90 to the world you can reuse it for this deployment). **This step can take some time since the image has to be downloaded**.

```
$ kubectl delete deployment nginx
deployment "nginx" deleted

$ kubectl run nginx --image=awassink/nginx-withcontent:3 --port=80
--replicas=4 --labels="run=nginx,visualize=true"
deployment "nginx" created
```

Now we can see Kubernetes' full magic at work. We will edit the deployment (http://kubernetes.io/docs/user-guide/deployments/#updating-a-deployment) to start using the second version of the image, which will be rolled out by the system, replacing one pod at a time. The service will never go down, during the update a user simply gets served either the old or the new version. To kick the update off, you must edit the deployment. Take note of the different parts of this deployment file. You can write such a file yourself to deploy your applications, which is often more practical than having a bloke or gall hammer commands into a cluster with kubectl. For now, change the container image to version 4. For those unfamiliar with this editor, start editing with insert, stop editing with esc, save the result with :w and quit with :q.

```
$ kubectl edit deployment nginx
deployment nginx edited

$ kubectl get deployments
```

Creating services, replicationcontrolers and pods from configuration files

Kubernetes resources can also be created from configuration files instead of via the command line. This makes it easy to put this kubernetes configuration in version control and maintain it from there.

```
$ kubectl delete svc nginx
service "nginx" deleted

$ kubectl delete deployment nginx
deployment "nginx" deleted

$ kubectl create -f nginx-deployment.yaml
replicationcontroller "nginx" created

$ kubectl create -f nginx-svc.yaml
service "nginx" created

Now edit the deployment yaml file to use a different image version (4->3)
$ kubectl replace -f nginx-deployment.yaml
```

3. Deploying a three tier application

Creating and claiming persisted volumes

The google compute engine hosts multiple disks for mounting. In Kubernetes you can make a volume available for usage by creating a Persisted Volume.

Edit the cddb-pvc.yaml file so that the pdName **matches your account name. Also change the PV name to your account name.**

```
$ kubectl create -f cddb-pv.yaml
persistentvolume "user-1" created
$ kubectl get pv
NAME           CAPACITY   ACCESSMODES   STATUS      CLAIM
REASON    AGE
user-1         1Gi        RWO           Available
28s
```

Before the volume can be used it needs to be claimed for a certain application. This is done by creating a Persisted Volume Claim.

```
$ kubectl create -f cddb-pvc.yaml
persistentvolumeclaim "cddb-mysql-pv-claim" created

$ kubectl get pvc
NAME                  STATUS    VOLUME          CAPACITY   ACCESSMODES   AGE
cddb-mysql-pv-claim   Bound     nfs-share-61    1Gi        RWO           12s
$ kubectl get pv
NAME           CAPACITY   ACCESSMODES   STATUS    CLAIM
REASON    AGE
user-1         1Gi        RWO           Bound
user-1/cddb-mysql-pv-claim             5m
```

More information can be found here: http://kubernetes.io/docs/user-guide/persistent-volumes/

Deploy the Mysql container and use the PVC for the data storage and create a service for mysql.

```
$ kubectl create -f cddb-mysql-deployment.yaml
deployment "cddb-mysql" created

$ kubectl create -f cddb-mysql-service.yaml
service "cddb-mysql" created
```

Deploy the backend container and create a service for the backend.

```
$ kubectl create -f cddb-backend-deployment.yaml
deployment "cddb-backend" created

$ kubectl create -f cddb-backend-service.yaml
service "cddb-backend" created
```

Deploy the frontend container and create a service for the fronend.

```
$ kubectl create -f cddb-frontend-deployment.yaml
deployment "cddb-frontend" created

$ kubectl create -f cddb-backend-frontend.yaml
service "cddb-frontend" created
```

Test that the application is working using a browser and that it stores the data in the database.

You can scale up the frontend and backend layer. But the mysql layer cannot be scaled. Though Kubernetes manages the persisted volumes and remounts them on a different node when needed.