CRC – sesja 4 – Kubernetes

1. Instalacja klastra Kubernetesa na pojedynczym hoście za pomocą instalatora RKE

Maszyna posiada już zainstalowaną usługę *docker*, także wykonanie tego kroku nie jest wymagane na potrzeby tego laboratorium. W celu ułatwienia i przyspieszenia omawianych tutaj ćwiczeń zalecanie jest pobranie plików konfiguracyjnych Yaml z poniższego repozytorium Git:

```
$ git clone https://github.com/przemo1934/crc_session_4.git
```

Na początek wygeneruj parę kluczy SSH dla swojego użytkownika za pomocą polecenia (zostaw wszystkie odpowiedzi domyślne):

```
$ ssh-keygen -t rsa -b 4096
```

Następnie dodaj wygenerwany w ten sposób klucz publiczny do pliku *authorized_keys*

```
$ ssh-copy-id <nazwa_użytkownika>@<adres_IP_maszyny>
```

Korzystając ze swojego użytkownika pobierz instalator RKE do swojego katalogu */home/<nazwa_użytkownika>/bin* za pomocą polecenia:

```
$ mkdir -p ~/bin && wget https://github.com/rancher/rke/releases/download/v0.2.1/rke_linux-amd64 -O ~/bin/rke && chmod +x ~/bin/rke
```

Wygeneruj plik konfiguracyjny *k8s-lab.yml* potrzebny do zbudowania nowego klastra korzystając z polecenia, dla ułatwienia podążaj za przykładowymi odpowiedziami:

```
$ rke config --name k8s-lab.yml
[+] Cluster Level SSH Private Key Path [~/.ssh/id_rsa]: /home/<nazwa_użytkownika>/.ssh/id_rsa
[+] Number of Hosts [1]: 1
[+] SSH Address of host (1) [none]: <Adres IP maszyny>
[+] SSH Port of host (1) [22]: 22
[+] SSH Private Key Path of host (158.177.175.60) [none]: /home/<nazwa_użytkownika>/.ssh/id_rsa
[+] SSH User of host (158.177.175.60) [ubuntu]: <nazwa_użytkownika>
[+] Is host (158.177.175.60) a Control Plane host (y/n)? [y]: y
[+] Is host (158.177.175.60) a Worker host (y/n)? [n]: y
[+] Is host (158.177.175.60) an etcd host (y/n)? [n]: y
[+] Override Hostname of host (158.177.175.60) [none]:
[+] Internal IP of host (158.177.175.60) [none]:
[+] Docker socket path on host (158.177.175.60) [/var/run/docker.sock]: /var/run/docker.sock
[+] Network Plugin Type (flannel, calico, weave, canal) [canal]: flannel
[+] Authentication Strategy [x509]:
[+] Authorization Mode (rbac, none) [rbac]:
[+] Kubernetes Docker image [rancher/hyperkube:v1.13.5-rancher1]: rancher/hyperkube:v1.12.7-rancher1
[+] Cluster domain [cluster.local]:
[+] Service Cluster IP Range [10.43.0.0/16]:
[+] Enable PodSecurityPolicy [n]:
[+] Cluster Network CIDR [10.42.0.0/16]:
[+] Cluster DNS Service IP [10.43.0.10]:
[+] Add addon manifest URLs or YAML files [no]:
```

Zbuduj klaster z wykorzystaniem wygenerowanej konfiguracji:

```
$ rke up --config k8s-lab.yml
```

Po zbudowaniu klastra w katalogu powinny pojawić się dodatkowe pliki:

kube_config_k8s-lab.yml – zawierający dane oraz certyfikaty służące do połączenia z klastrem

k8s-lab.rkestate – stan w jakim instalator RKE zostawił klaster.

2. Instalacja narzędzia kubectl i zapoznanie się z podstawowymi operacjami na klastrze

Jest wiele sposobów instalacji narzedzia *kubectl*. Na potrzeby tego laboratorium posłużymy się binarną wersją przeznaczoną dla systemu Linux:

```
$ wget https://storage.googleapis.com/kubernetes-release/release/v1.12.7/bin/linux/amd64/kubectl -O ~/bin/kubectl && chmod +x ~/bin/kubectl
```

W celu uzyskania dostępu do klastra za pomocą *kubectl* można:

a) wyeksportować zmienną *KUBECONFIG* zawierającą ścieżkę do konfiguracji – żeby zmiana była trwała i obecna po uruchomieniu nowej sesji SSH, należy dodać tę linię to pliku ~/.bashrc

```
export KUBECONFIG=~/kube_config_k8s-lab.yml
```

b) podawać tę ścieżkę za każdym razem wykonując konkretną komendę *kubectl*, np:

```
$ kubectl --kubeconfig ~/kube_config_k8s-lab.yml get namespaces
```

Mamy mamy już działający klaster oraz zainstalowane polecenie *kubectl*. Wykonamy teraz kilka operacji na naszym klastrze z użyciem tej komendy. Pod każdą komendą wklejona została przykładowa odpowiedź z rzeczywistego systemu.

Wyświetl podstawowe informacje o klastrze:

```
$ kubectl cluster-info
Kubernetes master is running at https://158.177.175.60:6443
KubeDNS is running at https://158.177.175.60:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Wyświetl istniejące systemowe namespace’y:

```
$ kubectl get namespaces
NAME           STATUS   AGE
default        Active   19m
ingress-nginx   Active   18m
kube-public     Active   19m
kube-system     Active   19m
```

Wyświetl pody w namespace *kube-system*:

```
$ kubectl --namespace kube-system get pods
NAME                                    READY   STATUS      RESTARTS   AGE
kube-dns-9846dcf88-wbr7n                3/3     Running     0          19m
kube-dns-autoscaler-59cc69d67f-w9nkh    1/1     Running     0          19m
kube-flannel-46t57                      2/2     Running     0          19m
metrics-server-844bd95c7b-l8xb5         1/1     Running     0          19m
rke-ingress-controller-deploy-job-5qpcd 0/1     Completed   0          19m
rke-kube-dns-addon-deploy-job-zsf5q     0/1     Completed   0          19m
rke-metrics-addon-deploy-job-fvrlw      0/1     Completed   0          19m
```

```
rke-network-plugin-deploy-job-z4bqt          0/1     Completed    0          19m
```

Wyświetl logi z poda metrics-server-xxxxxxxxxx-xxxxx (**musisz wziąć nazwę poda z wyniku poprzedniej komendy, gdyż są one unikalne dla każdego poda**):

```
$ kubectl --namespace kube-system logs metrics-server-XXXXXXXXXX-XXXXX
```

Wykonaj to samo dla poda *kube-dns*. Co poszło nie tak?

```
$ kubectl --namespace kube-system logs kube-dns-xxxxxxxxxx-xxxx
Error from server (BadRequest): a container name must be specified for pod kube-dns-9846dcf88-zpzrv, choose one of: [kubedns dnsmasq sidecar]
```

Zauważ, że pod ten składa się z trzech kontenerów – należy podać nazwę konkretnego kontenera, aby dostać się do jego logów, np:

```
$ kubectl --namespace kube-system logs kube-dns-xxxxxxxxxx-xxxx kubedns
```

Spróbujmy teraz usunąć jeden z podów, np. metrics-server-xxxxxxxxxx-xxxxx

```
$ kubectl --namespace kube-system delete pods metrics-server-xxxxxxxxxx-xxxxx
pod "metrics-server-xxxxxxxxxx-xxxxx" deleted
```

Wyświetlmy jeszcze raz wszystkie pody z namespace'u *kube-system*:

```
$ kubectl --namespace kube-system get pods
NAME                                    READY   STATUS      RESTARTS   AGE
kube-dns-9846dcf88-wbr7n                3/3     Running     0          68m
kube-dns-autoscaler-59cc69d67f-w9nkh    1/1     Running     0          68m
kube-flannel-46t57                      2/2     Running     0          68m
metrics-server-844bd95c7b-brbbp         1/1     Running     0          38s
rke-ingress-controller-deploy-job-5qpcd 0/1     Completed   0          68m
rke-kube-dns-addon-deploy-job-zsf5q     0/1     Completed   0          68m
rke-metrics-addon-deploy-job-fvrlw      0/1     Completed   0          68m
rke-network-plugin-deploy-job-z4bqt     0/1     Completed   0          68m
```

Widac wyraźnie, że pod metric-server-xxxxxxxxxx-xxxxx ciągle tam jest. Zwróć uwage na inne ID – to nie jest ten sam pod. Stary po usunięciu został natychmiast zastąpiony nowym (można to zaobserwować, jeżeli podczas usuwania będziemy mieć w innym terminalu uruchomioną komendę: watch *kubectl -n kube-system get pods*).

Gdzie zatem zapisana jest instrukcja, która odpowiada za ponowne utworzenie tego poda? Odpowiedź na to pytanie znajduje się w definicji deploymentu. Wyświetl, zatem wszystkie deploymenty w namespace *kube-system:*

```
$ kubectl --namespace kube-system get deployments
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
kube-dns            1         1         1            1           73m
kube-dns-autoscaler 1         1         1            1           73m
metrics-server      1         1         1            1           73m
```

Pod jest docelowym kontenerem/kontenerami zawierającymi usługę, może być zdefiniowany w ramach *deployment, daemonset.* Także tworzony tymczasowy w celu wykonania jakiejś akcji typu *job*. Obiekty w Kubernetesie definiowane są za pomocą plików Yaml. Spróbujmy, zatem uzyskać taki plik dla naszego deploymentu *metrics-server.*

Można tego dokonać za pomocą komendy:

```
$ kubectl --namespace kube-system get deployments metrics-server -o yaml
```
albo
```
$ kubectl --namespace kube-system describe deployments metrics-server
```

To tylko kilka przykładowych komend przydatnych w codziennej pracy z klastrami Kubernetesa. Więcej można znaleźć w oficjalnej dokumentacji, a także po wykonaniu komendy *kubectl --help*

3. Kubernetes Dashboard – instalacja oraz przegląd możliwości

Kubernetes Dashboard jest bardzo przydatnym narzędziem, nieinstalowanym domyślnie. Pozwala na wygodne przeglądanie zasobów naszego klastra oraz tworzenie nowych. Jest świetnym uzupełnieniem komendy *kubectl* w codziennej administracji klastrem.

Instalacja

W celu instalacji Dashboarda należy wykonać polecenie:

```
$ kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/master/aio/deploy/recommended/kubernetes-dashboard.yaml
```

Teraz należy zweryfikować instalację Dashboarda – jest on domyślnie instalowany w namespace *kube-system*. Powyższe polecanie *kubectl apply* aplikuje plik Yaml, w którym znajduje się definicja deploymentu i service'u. Poprawność instalacji sprawdzimy za pomocą poniższych komend:

```
$ kubectl -n kube-system get pods
$ kubectl -n kube-system get deployments
$ kubectl -n kube-system get services
```

Czy widoczne są dodatkowe objekty związane z dashboardem?

Jeżeli tak, możemy teraz przejść do kolejnego kroku jakim jest uzyskanie dostępu do naszej nowo zainstalowanej aplikacji. Sposobów na dokonanie tego jest kilka, natomiast my skupimy się tutaj na jednym z nich.

Sposób ten polega na edycji service'u z dashboardem i wystawieniu usługi poprzez tzw. *NodePort*, czyli port zewnętrzny maszyny wirtualnej, na której jest zainstalowany klaster. W celu edycji usługi należy wykonać poniższą komendę:

```
$ kubectl -n kube-system edit service kubernetes-dashboard
```

W otworzonym edytorze należy zmienić "*type: ClusterIP*" na "*type: NodePort*", a następnie zapisać zmiany:

```
apiVersion: v1
...
  name: kubernetes-dashboard
  namespace: kube-system
  resourceVersion: "343478"
  selfLink: /api/v1/namespaces/kube-system/services/kubernetes-dashboard-head
  uid: 8e48f478-993d-11e7-87e0-901b0e532516
spec:
  clusterIP: 10.100.124.90
  externalTrafficPolicy: Cluster
  ports:
  - port: 443
```

```
    protocol: TCP
    targetPort: 8443
  selector:
    k8s-app: kubernetes-dashboard
  sessionAffinity: None
  type: NodePort
status:
  loadBalancer: {}
```

Sprawdź port na którym jest udostępniony dashboard:

```
$ kubectl -n kube-system get services kubernetes-dashboard
NAME                   TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)     AGE
kubernetes-dashboard   NodePort 10.43.78.251   <none>     443:31581/TCP   13m
```

Przetestuj dostępność dashboardu poprzez przeglądarkę:

https://<Adres_IP_maszyny_wirtualnej>:<port>

Odpowiedni port sprawdzaliśmy w poprzednim poleceniu – w naszym przykładzie jest to 31581.

Dodatek:

Jeżeli posiadamy *kubectl* skonfigurowany na naszym laptopie możemy użyć prostszej metody uzyskania dostepu do Dashboardu. W tym celu wystarczy wykonać polecenie:

```
$ kubectl proxy
```

Następnie w przeglądarce otworzyć stronę:

http://localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/

Autoryzacja/logowanie do Dashboardu

Istnieją dwa sposoby autoryzacji do Dashboardu:

a) poprzez kubeconfig – plik musimy ściągnąć na swoją stację roboczą

b) token – wymaga odpowiedniej konfiguracji na klastrze oraz wygenerowania jednorazowego tokena za każdym razem

Na potrzeby tych zajęć skupimy się na sposobie b) gdyż wymaga on więcej uwagi.

Utwórz plik *admin-user.yaml* z zawartością:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kube-system
```

Utwórz plik *cluster-role-binding.yaml* z zawartością:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
```

```
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kube-system
```

Utworzone zostały dwa pliki Yaml, jeden odpowiedzialny za utworzenie objektu *ServiceAccount*, natomiast drugi za utworzenie *ClusterRoleBinding*. W celu dokonania deploymentu wspomnianych objektów należy wykonać poniższe polecenia:

```
$ kubectl apply -f admin-user.yaml
$ kubectl apply -f cluster-role-binding.yaml
```

Pobierz token potrzebny do zalogowania się:

```
$ kubectl -n kube-system describe secret $(kubectl -n kube-system get secret | grep admin-user | awk '{print $1}')
```

Zaloguj się do Dashboardu z użyciem uzyskanego w ten sposób tokena. Przejrzyj dostepne zasoby oraz opcje jakie oferuje Dashboard:

– namespaces

– pods

– deployments

– daemonsets

Dashboard umożliwia również przeglądanie logów, statystyk oraz tworzenie/deployment nowych obiektów z gotowych plików Yaml.

4. Tworzenie własnych obiektów w nowo utworzonym klastrze

a) Utwórz namespace o nazwie *test1 i test2*.

Najłatwiej dokonać tego poprzez utworzenie plików Yaml.

```
vim test1.yaml
```

```
apiVersion: v1
kind: Namespace
metadata:
  name: test1
  labels:
    name: test1
```

```
vim test2.yaml
```

```
apiVersion: v1
kind: Namespace
metadata:
  name: test2
  labels:
    name: test2
```

Dodaj zdefiniowane w ten sposób zasoby do klastra za pomocą komendy:

```
$ kubectl apply -f test1.yaml
$ kubectl apply -f test2.yaml
```

albo

```
$ kubectl create -f test1.yaml
$ kubectl create -f test2.yaml
```

Różnica między *kubectl apply* a *kubectl create* polega na tym, że *kubectl create* służy do tworzenia nowych obiektów i użytkownik musi posiadać informację, czy taki obiekt istnieje, czy też nie (podejście imperatywne). W przypadku *kubectl apply* wydajemy po prostu komendę, w której podajemy jak ma wyglądać stan klastra po jej wykonaniu (podejście deklaratywne) nie martwiąc się jego stanem początkowym. Z racji tego, że taki zasób nie istnieje jeszcze na klastrze, możemy użyć tych dwóch komend zamiennie.

Dodatek:

Utwórz namespace o dowolnej nazwie za pomocą Dashboardu. Następnie przygotowany na te potrzeby plik Yaml spróbuj zdeployować ponownie za pomocą komendy *kubectl create*, a nastepnie za pomocą *kubectl apply*. Co można było zaobserwować?

b) Utwórz nowy job o nazwie *oblicz-pi* w namespace *test1*

Job jest to objekt, który tworzy jeden lub większą liczbę podów, które przestają być aktywne w momencie poprawnego wykonania określonego zadania. Może być to backup danych, wykonanie określonych obliczeń, dokonanie migracji wersji bazy danych w czasie deploymentu nowszej wersji aplikacji, a także wiele innych. Kubernetes posiada również mechanizm podobny do uniksowego *crona* pozwalający na okresowe wykonywanie określonych akcji.

Utworzymy teraz job, który obliczy liczbę pi to 2000 miejsc po przecinku, poniżej definicja joba:

```
vim oblicz-pi.yaml
```

```
apiVersion: batch/v1
kind: Job
metadata:
  name: oblicz-pi
  namespace: test1
spec:
  ttlSecondsAfterFinished: 100
  template:
    spec:
      containers:
      - name: oblicz-pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

W powyższej definicji zwróć uwage na:

`namespace: test1` – definiuje do jakiego namespace przynależy job I utworzony przez niego pod/kontener
ttlSecondsAfterFinished: 100 – definiuje przedział czasu po którym zostaną usunięte pody utworzone w ramach joba, bez tego pody w statusie Completed utworzone przez joba, będą cały czas zajmować miejsce na klastrze za każdym razem, gdy go uruchomimy

Tworzenie joba wykonamy za pomocą dobrze znanej komendy *kubectl apply*:

```
$ kubectl apply -f oblicz-pi.yaml
job.batch/oblicz-pi created
```

Job został utworzony, zweryfikujmy to zatem:

```
$ kubectl -n test1 get jobs
NAME        COMPLETIONS   DURATION   AGE
oblicz-pi   1/1           10s        7m29s

$ kubectl -n test1 get pods --show-all
NAME              READY   STATUS      RESTARTS   AGE
oblicz-pi-glm4j   0/1     Completed   0          21m

$ kubectl -n test1 describe jobs/oblicz-pi
Name:           oblicz-pi
Namespace:      test1
Selector:       controller-uid=9fd19ac2-59af-11e9-982a-06a892d03969
Labels:         controller-uid=9fd19ac2-59af-11e9-982a-06a892d03969
                job-name=oblicz-pi
Annotations:    kubectl.kubernetes.io/last-applied-configuration:
                  {"apiVersion":"batch/v1","kind":"Job","metadata":{"annotations":{},"name":"oblicz-
pi","namespace":"test1"},"spec":{"template":{"spec":{"co...
Parallelism:    1
Completions:    1
Duration:       10s
Pods Statuses:  0 Running / 1 Succeeded / 0 Failed
Pod Template:
  Labels:  controller-uid=9fd19ac2-59af-11e9-982a-06a892d03969
           job-name=oblicz-pi
  Containers:
   oblicz-pi:
    Image:      perl
    Port:       <none>
    Host Port:  <none>
    Command:
      perl
      -Mbignum=bpi
      -wle
      print bpi(2000)
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From            Message
  ----    ------            ----  ----            -------
  Normal  SuccessfulCreate  23m   job-controller  Created pod: oblicz-pi-glm4j
```

Powyżej możliśmy uzyskać kilka szczegółów na temat naszego joba. Najważniejszy jednak jest wynik jego działania. W celu poznania tego wyniku (a jest nim obliczenie liczby pi z dokładnością do 2000 miejsc po przecinku), należy spojrzeć w logi poda, który został utworzony w procesie uruchamiania naszego joba:

```
$ kubectl -n test1 logs oblicz-pi-glm4j
3.14159265358979323846264338327950288419716939937510.....
```

W celu usunięcia joba możemy wykonać polecenie:

```
$ kubectl -n test1 delete job oblicz-pi
job.batch "oblicz-pi" deleted

$ kubectl -n test1 get pods -a
No resources found.
```

c) Utwórz nowy *deployment* w namespace *test1* i *test2* serwujący usługę WWW poprzez nginx

Deployment to podstawowy objekt w Kubernetesie pozwalajacy w deklaratywny sposób definiować Pody oraz ReplicaSets. Dzięki niemu możemy definiować pody serwujące różnego rodzaju aplikacje oraz sterować ilością replik danego rodzaju poda.

Stwórzmy zatem prosty *Deployment*, który wygeneruje trzy pody z serwerem Nginx w namespace *test1*.

```
vim nginx-deployment.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: test1
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

```
$ kubectl apply -f nginx-deployment.yaml
deployment.apps/nginx-deployment created
```

```
$ kubectl -n test1 get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment         3         3         3           3          8s
```

```
$ kubectl -n test1 get pods
NAME                               READY   STATUS    RESTARTS   AGE
nginx-deployment-5c689d88bb-5hs9d   1/1     Running   0          53s
nginx-deployment-5c689d88bb-npwwz   1/1     Running   0          53s
nginx-deployment-5c689d88bb-pqpj7   1/1     Running   0          53s
```

Zróbmy to samo dla namespace *test2, ale wykorzystując serwer Apache:*

```
vim apache-deployment.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache-deployment
  namespace: test2
  labels:
    app: apache
spec:
  replicas: 3
  selector:
```

```
    matchLabels:
      app: apache
  template:
    metadata:
      labels:
        app: apache
    spec:
      containers:
      - name: apache
        image: httpd:2.4.39
        ports:
          - containerPort: 80
```

```
$ kubectl apply -f apache-deployment.yaml
deployment.apps/apache-deployment created
```

Powyższe kroki doprowadziły to utworzenia dwóch deploymentów o nazwie *nginx-deployment* oraz *apache-deployment* znajdujących się w izolowanych namespacach. Każdy deployment dba oto, żeby w danym momencie były utworzone 3 pody z serwerem nginx/apache, które serwują WWW na porcie 80.

Na potrzeby tego laboratorium nie bedziemy udostępniać tych usług na zewnątrz – podobną operację zrobiliśmy w dziale 3 udostępniając Dashboard poprzez NodePort. NodePort jest najprostszą opcją udostępnienia usługi na świat. Inne możliwości to Proxy, Loadbalancer albo Ingress, zainteresowanych odsyłam do porównania

Aby mieć możliwość odpytywania usług w wewnętrznej wirtualnej sieci Kubernetesa utworzymy objekt typu *Service*. Użyjemy tutaj *ClusterIP* jako najprostszy sposób na udostępnienie usługi dla innych usług wewnątrz klastra.

Utworzymy zatem dwie usługi wewnątrz klastra:

```
vim nginx-service.yaml
```

```
kind: Service
apiVersion: v1
metadata:
  name: nginx-service
  namespace: test1
spec:
  type: ClusterIP
  ports:
  - name: nginx-service
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
```

```
$ kubectl apply -f nginx-service.yaml
```

```
$ kubectl -n test1 get services
NAME            TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
nginx-service       ClusterIP   10.43.148.69   <none>       80/TCP    27s
```

Wykonajmy to samo dla deploymentu z apache w namespace test2.

```
vim apache-service.yaml
```

```
kind: Service
apiVersion: v1
metadata:
  name: apache-service
  namespace: test2
spec:
  type: ClusterIP
  ports:
  - name: apache-service
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: apache
```

```
$ kubectl apply -f apache-service.yaml
```

d) Prezentacja działania wwewnętrznej usługi DNS – kube-dns.

Utwórz niezależny kontener na bazie obrazu z curl i dokonaj weryfikacji działania usług utworzonych w punkcie c).

Poniższe polecenie utworzy pod/kontener na bazie obrazu pstauffer/curl I otworzy w nim sesję terminala. Pod domyślnie utworzy się w namespace: Default

```
$ kubectl delete pods busybox && kubectl run -i --tty busybox --image=pstauffer/curl --restart=Never -- sh
```

Wewnątrz terminala kontenera sprawdź łączność do usługi z Nginx oraz Apache:

Nginx:

```
/ # curl nginx-service.test1.svc.cluster.local
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Oraz Apache:

```
/ # curl apache-service.test2.svc.cluster.local
<html><body><h1>It works!</h1></body></html>
```

Zwróć uwagę na to, że łączność z usługą odbywa się poprzez działający wewnętrznie DNS – nie trzeba podawać konkretnego ClusterIP przypisanego do usługi. W należności od namespacu wymagane jest dodanie odpowiedniej domeny do nazwy usługi:

– test1.svc.cluster.local – jest do domena dla namespace test1

– test2.svc.cluster.local – jest to domena dla namespace test2

– cluster.local – domyślna domena wewnętrzna ustawiana na etapie instalacji klastra.

Wyjdź z terminala tego kontenera poprzez wydanie komendy:

```
# exit
```

e) awaryjny dostęp do terminala konkretnego poda poprzez *kubectl exec*

Czasami w celu debugowania naszych usług I aplikacji potrzebujemy “wejść” do konsoli danego poda. Spróbujmy zatem wybrać jedne z kontenerów deploymentu nginx-deployment w namespace test1.

Listujemy pody w test1:

```
$ kubectl -n test1 get pods
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-5c689d88bb-2v2bq   1/1     Running   0          17m
nginx-deployment-5c689d88bb-fhlxx   1/1     Running   0          17m
nginx-deployment-5c689d88bb-xhc4f   1/1     Running   0          17m
```

Wybieramy ID konkretnego poda i wchodzimy w terminal (ID jest unikalne – wybierz ID swojego poda z powyższej komendy):

```
$ kubectl -n test1 exec -it nginx-deployment-5c689d88bb-2v2bq -- bash
root@nginx-deployment-5c689d88bb-2v2bq:/#
```

Sprawdź wersję Nginx, czy zgadza się z definicją w deploymencie:

```
root@nginx-deployment-5c689d88bb-2v2bq:/# nginx -v
nginx version: nginx/1.7.9
```

Spradź rozwiązywanie nazwy bez podawania całej domeny. Użyjemy tutaj polecenia PING, poniewaz *curl* nie jest dostępny w obrazie z nginx:

```
root@nginx-deployment-5c689d88bb-2v2bq:/# ping nginx-service
PING nginx-service.test1.svc.cluster.local (10.43.148.69): 48 data bytes
```

Jak widać nazwa jest poprawnie rozwiązywana do konkretnego ClusterIP bez potrzeby podawania całej nazwy domenowej – jest to możliwe jedynie przy komunikacji między servicami w obrębie tego samego namespacu.

```
$ kubectl -n test1 get services
NAME            TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
nginx-service   ClusterIP   10.43.148.69   <none>        80/TCP    24m
```