Exposing pods to the cluster

Create an nginx Pod, and note that it has a container port specification:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
```

This makes it accessible from any node in your cluster. Check the nodes the Pod is running on:

```
$ kubectl create -f ./run-my-nginx.yaml
$ kubectl get pods -l run=my-nginx -o wide
NAME                        READY     STATUS    RESTARTS   AGE       IP            NODE
my-nginx-3800858182-jr4a2   1/1       Running   0          13s       10.244.3.4
kubernetes-minion-905m
my-nginx-3800858182-kna2y   1/1       Running   0          13s       10.244.2.5
kubernetes-minion-ljyd
```

Check your pods' IPs:

```
$ kubectl get pods -l run=my-nginx -o yaml | grep podIP
    podIP: 10.244.3.4
    podIP: 10.244.2.5
```

Creating a Service

So we have pods running nginx in a flat, cluster wide, address space. In theory, you could talk to these pods directly, but what happens when a node dies? The pods die with it, and the Deployment will create new ones, with different IPs. This is the problem a Service solves.

You can create a Service for your 2 nginx replicas with **kubectl expose:**

```
$ kubectl expose deployment/my-nginx
service/my-nginx exposed
```

This is equivalent to kubectl create -f the following yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
```

Concepts

Overview

What is Kubernetes?

Kubernetes Components

The Kubernetes API

Working with Kubernetes Objects

Understanding Kubernetes Objects

Names

Namespaces

Labels and Selectors

Annotations

Field Selectors

Recommended Labels

Object Management Using kubectl

Kubernetes Object Management

Managing Kubernetes Objects Using Imperative Commands

Imperative Management of Kubernetes Objects Using Configuration Files

Declarative Management of Kubernetes Objects Using Configuration Files

Kubernetes Architecture

Nodes

Master-Node communication

Concepts Underlying the Cloud Controller Manager

Containers

Images

Container Environment Variables

Runtime Class

Container Lifecycle Hooks

Workloads

Pods

Pod Overview

Pods

Pod Lifecycle

Init Containers

Pod Preset

Disruptions

Controllers

ReplicaSet

ReplicationController

Deployments

StatefulSets

DaemonSet

Garbage Collection

TTL Controller for Finished Resources

Jobs - Run to Completion

CronJob

Services, Load Balancing, and Networking

Services

DNS for Services and Pods

Connecting Applications with Services

Ingress

Network Policies

Adding entries to Pod /etc/hosts with HostAliases

Storage

Volumes

Persistent Volumes

Volume Snapshots

Storage Classes

Volume Snapshot Classes

Dynamic Volume Provisioning

Node-specific Volume Limits

Configuration

Configuration Best Practices

Managing Compute Resources for Containers

Assigning Pods to Nodes

Taints and Tolerations

Secrets

Organizing Cluster Access Using kubeconfig Files

Pod Priority and Preemption

Scheduler Performance Tuning

Policies

Resource Quotas

Pod Security Policies

Cluster Administration

Cluster Administration Overview

Certificates

Cloud Providers

Managing Resources

Cluster Networking

Logging Architecture

Configuring kubelet Garbage Collection

Federation

Proxies in Kubernetes

Controller manager metrics

Installing Addons

Extending Kubernetes

Extending your Kubernetes Cluster

Extending the Kubernetes API

Extending the Kubernetes API with the aggregation layer

Custom Resources

Compute, Storage, and Networking Extensions

Network Plugins

Device Plugins

Service Catalog

Edit This Page

Connecting Applications with Services

The Kubernetes model for connecting containers

Now that you have a continuously running, replicated application you can expose it on a network. Before discussing the Kubernetes approach to networking, it is worthwhile to contrast it with the “normal” way networking works with Docker.

By default, Docker uses host-private networking, so containers can talk to other containers only if they are on the same machine. In order for Docker containers to communicate across nodes, there must be allocated ports on the machine’s own IP address, which are then forwarded or proxied to the containers. This obviously means that containers must either coordinate which ports they use very carefully or ports must be allocated dynamically.

Coordinating ports across multiple developers is very difficult to do at scale and exposes users to cluster-level issues outside of their control. Kubernetes assumes that pods can communicate with other pods, regardless of which host they land on. We give every pod its own cluster-private-IP address so you do not need to explicitly create links between pods or mapping container ports to host ports. This means that containers within a Pod can all reach each other’s ports on localhost, and all pods in a cluster can see each other without NAT. The rest of this document will elaborate on how you can run reliable services on such a networking model.

This guide uses a simple nginx server to demonstrate proof of concept. The same principles are embodied in a more complete Jenkins CI application.

- Exposing pods to the cluster
- Creating a Service
- Accessing the Service
- Securing the Service
- Exposing the Service
- What's next

Exposing pods to the cluster

We did this in a previous example, but let’s do it once again and focus on the networking perspective. Create an nginx Pod, and note that it has a container port specification:

service/networking/run-my-nginx.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
```

This makes it accessible from any node in your cluster. Check the nodes the Pod is running on:

```
$ kubectl create -f ./run-my-nginx.yaml
$ kubectl get pods -l run=my-nginx -o wide
NAME                        READY     STATUS    RESTARTS   AGE       IP           NODE
my-nginx-3800858182-jr4a2   1/1       Running   0          13s       10.244.3.4
kubernetes-minion-905m
my-nginx-3800858182-kna2y   1/1       Running   0          13s       10.244.2.5
kubernetes-minion-ljyd
```

Check your pods' IPs:

```
$ kubectl get pods -l run=my-nginx -o yaml | grep podIP
    podIP: 10.244.3.4
    podIP: 10.244.2.5
```

You should be able to ssh into any node in your cluster and curl both IPs. Note that the containers are *not* using port 80 on the node, nor are there any special NAT rules to route traffic to the pod. This means you can run multiple nginx pods on the same node all using the same containerPort and access them from any other pod or node in your cluster using IP. Like Docker, ports can still be published to the host node's interfaces, but the need for this is radically diminished because of the networking model.

You can read more about [how we achieve this] if you're curious.

Creating a Service

So we have pods running nginx in a flat, cluster wide, address space. In theory, you could talk to these pods directly, but what happens when a node dies? The pods die with it, and the Deployment will create new ones, with different IPs. This is the problem a Service solves.
A Kubernetes Service is an abstraction which defines a logical set of Pods running somewhere in your cluster, that all provide the same functionality. When created, each Service is assigned a unique IP address (also called clusterIP). This address is tied to the lifespan of the Service, and will not change while the Service is alive. Pods can be configured to talk to the Service, and know that communication to the Service will be automatically load-balanced out to some pod that is a member of the Service.
You can create a Service for your 2 nginx replicas with kubectl expose:

```
$ kubectl expose deployment/my-nginx
service/my-nginx exposed
```

This is equivalent to kubectl create -f the following yaml:

service/networking/nginx-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
```

This specification will create a Service which targets TCP port 80 on any Pod with the run: my-nginx label, and expose it on an abstracted Service port (targetPort: is the port the container accepts traffic on, port: is the abstracted Service port, which can be any port other pods use to access the Service).

```
$ kubectl get svc my-nginx
NAME       TYPE       CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
my-nginx   ClusterIP  10.0.162.149  <none>        80/TCP    21s
```

As mentioned previously, a Service is backed by a group of Pods. These Pods are exposed through endpoints. The Service's selector will be evaluated continuously and the results will be POSTed to an Endpoints object also named my-nginx. When a Pod dies, it is automatically removed from the endpoints, and new Pods matching the Service's selector will automatically get added to the endpoints. Check the endpoints, and note that the IPs are the same as the Pods created in the first step:

```
$ kubectl describe svc my-nginx
Name:              my-nginx
Namespace:         default
Labels:            run=my-nginx
Annotations:       <none>
Selector:          run=my-nginx
Type:              ClusterIP
IP:                10.0.162.149
Port:              <unset> 80/TCP
Endpoints:         10.244.2.5:80,10.244.3.4:80
Session Affinity:  None
Events:            <none>

$ kubectl get ep my-nginx
NAME       ENDPOINTS                     AGE
my-nginx   10.244.2.5:80,10.244.3.4:80   1m
```

Accessing the Service

Kubernetes supports 2 primary modes of finding a Service - environment variables and DNS. The former works out of the box while the latter requires the CoreDNS cluster addon.

Environment Variables

When a Pod runs on a Node, the kubelet adds a set of environment variables for each active Service. This introduces an ordering problem. To see why, inspect the environment of your running nginx Pods (your Pod name will be different):

```
$ kubectl exec my-nginx-3800858182-jr4a2 -- printenv | grep SERVICE
KUBERNETES_SERVICE_HOST=10.0.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
```

Note there's no mention of your Service. This is because you created the replicas before the Service. Another disadvantage of doing this is that the scheduler might put both Pods on the same machine, which will take your entire Service down if it dies. We can do this the right way by killing the 2 Pods and waiting for the Deployment to recreate them. This time around the Service exists *before* the replicas. This will give you scheduler-level Service spreading of your Pods (provided all your nodes have equal capacity), as well as the right environment variables:

```
$ kubectl scale deployment my-nginx --replicas=0; kubectl scale deployment my-nginx --replicas=2;

$ kubectl get pods -l run=my-nginx -o wide
NAME                        READY     STATUS    RESTARTS   AGE       IP            NODE
my-nginx-3800858182-e9ihh   1/1       Running   0          5s        10.244.2.7    kubernetes-minion-ljyd
my-nginx-3800858182-j4rm4   1/1       Running   0          5s        10.244.3.8    kubernetes-minion-905m
```

You may notice that the pods have different names, since they are killed and recreated.

```
$ kubectl exec my-nginx-3800858182-e9ihh -- printenv | grep SERVICE
KUBERNETES_SERVICE_PORT=443
MY_NGINX_SERVICE_HOST=10.0.162.149
KUBERNETES_SERVICE_HOST=10.0.0.1
MY_NGINX_SERVICE_PORT=80
KUBERNETES_SERVICE_PORT_HTTPS=443
```

DNS

Kubernetes offers a DNS cluster addon Service that automatically assigns dns names to other Services. You can check if it's running on your cluster:

```
$ kubectl get services kube-dns --namespace=kube-system
NAME       TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)         AGE
kube-dns   ClusterIP   10.0.0.10    <none>        53/UDP,53/TCP   8m
```

If it isn't running, you can enable it. The rest of this section will assume you have a Service with a long lived IP (my-nginx), and a DNS server that has assigned a name to that IP (the CoreDNS cluster addon), so you can talk to the Service from any pod in your cluster using standard methods (e.g. gethostbyname). Let's run another curl application to test this:

```
$ kubectl run curl --image=radial/busyboxplus:curl -i --tty
Waiting for pod default/curl-131556218-9fnch to be running, status is Pending, pod ready: false
Hit enter for command prompt
```

Then, hit enter and run nslookup my-nginx:

```
[ root@curl-131556218-9fnch:/ ]$ nslookup my-nginx

Server:    10.0.0.10
Address 1: 10.0.0.10

Name:      my-nginx
Address 1: 10.0.162.149
```

Securing the Service

Till now we have only accessed the nginx server from within the cluster. Before exposing the Service to the internet, you want to make sure the communication channel is secure. For this, you will need:

- Self signed certificates for https (unless you already have an identity certificate)
- An nginx server configured to use the certificates
- A secret that makes the certificates accessible to pods

You can acquire all these from the nginx https example. This requires having go and make tools installed. If you don't want to install those, then follow the manual steps later. In short:

```
$ make keys secret KEY=/tmp/nginx.key CERT=/tmp/nginx.crt SECRET=/tmp/secret.json
$ kubectl create -f /tmp/secret.json
secret/nginxsecret created
$ kubectl get secrets
NAME                  TYPE                                  DATA      AGE
default-token-il9rc   kubernetes.io/service-account-token   1         1d
nginxsecret           Opaque                                2         1m
```

Following are the manual steps to follow in case you run into problems running make (on windows for example):

#create a public private key pair
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /d/tmp/nginx.key -out /d/tmp/nginx.crt -subj "/CN=my-nginx/O=my-nginx"
#convert the keys to base64 encoding
cat /d/tmp/nginx.crt | base64
cat /d/tmp/nginx.key | base64

Use the output from the previous commands to create a yaml file as follows. The base64 encoded value should all be on a single line.

```
apiVersion: "v1"
kind: "Secret"
metadata:
  name: "nginxsecret"
  namespace: "default"
data:
  nginx.crt:
"LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURlekNDQWdlZ0F3SUJBZ0lKQUp5
M3lQK0pzMlpJTUEwR0NTcUdTSWIzRFFFQkJRVUFNQ1I4RVRBUEJnTlYKQkFNVENHNW
5hVzU0YzNaak1SRXdEd1lEVlFRS0V3aHVaMmx1ZUhOMll6QWVGdzB4TnpFd01qWXdOek
EzTVRKYQpGdzB4T0RFd01qWXdOekEzTVRKYU1DWXhFVEFQQmdOVkJBTVRDRzVuYV
c1NGMzWmpNUkV3RHdZRFZRUUtFd2h1CloybHVlSE4yWXpDQ0FTSXdEUVlKS29aSWh2
Y05BUUVCQlFBRGdnRVBBRENDQVFvQ2dnRUJBSjFxSU1SOVdWM0lKMlZlQlRMRmtob
DRONXljMEJxYUhlQktMSnJMcy8vdzZhU3hRS29GbHlJSU94NGUrMlN5ajBFcndCLzlYTnB
wbQppeW1CL3JkRldkOXg5UWhBQUxCZkVaTmNiV3NsTVFVcnhBZW50VWt1dk1vLzgvM
HRpbGhjc3paenJEYVJ4NEo5Ci82UVRtVVl3a0ZTWUpOWTVQZkR3cGc3dlVvaDZmZ1Voa
m92VG42eHNVR0M2QURVODBpNXFlZWhNeVl1N2lmU2YKNHZpaXdlY3hnL3lZR1JBRS9
mRTRqakxCdmdONjc2SU90S01rZXV3R0ljNDFhd05tNnNTSzRqYUNGeGpYSnZaZQp2by9
kTlEybHhHWCtKT2l3SEhXbXNhdGp4WTRaNVk3R1ZoK0QrWnYvcW1mMFgvbVY0Rmo1N
zV3ajFMWVBocWtsCmdhSXZYRyt4U1FVQ0F3RUFBYU5RTUU0d0hRWURWUjBPQkJZRU
ZPNG9OWkl3YXc1OUlsYkROMzhlYkduYnhFVjcKTUl4R0ExVWRJd1FZTUJhQUZPNG9OW
kl3YXc1OUlsYkROMzhlYkduYnhFVjdNQXdHQTFVZEV3UUZNQU1CQWY4dwpEUVlKS29a
SWh2Y05BUUVGQlFBRGdnRUJBRVhTMW9FU0lFaXdyMDhWcVA0K2NwTHI3TW5FMTdu
cDBvMm14alFvCjRGb0RvRjdRZnZqeE04Tzd2TjB0clcxb2pGSW0vWDE4ZnZaL3k4ZzVaW
G40Vm8zc3hKVmRBcStNZC9jTStzUGEKNmJjTkNUekZqeFpUV0UrKzE5NS9zb2dmOUZ3V
DVDK3U2Q3B5N0M3MTZvUXRUakViV05VdEt4cXI0Nk1OZWNCMApwRFhWZmdWQTRadk
R4NFo3S2RiZDY5eXM3OVFHYmg5ZW1PZ05NZFlsSUswSGt0ejF5WU4vbVpmK3FqTkJqb
WZjCkNnMnlwbGQ0Wi8rUUNQZjI3SkoybFlrY2FnT0R4elBWcGxNSEcybzgvTHFDdnh6elZ
PUDUxeXdLZEtxaUMwSVEKQ0l5T2wwWW5scE9UNEh1b2hSUzBPOStlMm9KdFZsNUlycz
RpbDlhZ3RTVXFxUlU9Ci0tLS0tRU5EIENFUlRJRkIDQVRFLS0tLS0K"
  nginx.key:
```

"LS0tLS1CRUdJTiBQUklWQVRFIEtFWS0tLS0tCk1JSUV2UUlCQURBTkJna3Foa2lHOXcw
QkFRRUZBQVNDQktjd2dnU2pBZ0VBQW9JQkFRQ2RhaURFZlZsZHdkbFlKd1V5eFpJWm
VEZWNuTkFhbWh4d1NpeWF5N1AvOE9ta3NVQ3FCWmNpQ0RzZUh2dGtzbzlCSzhBZi9We
mFhWm9zcApnZjYzUlZuZmNmVUlRQUN3WHhHVFhHMXJKVEVGSzhRSHA3VkpMcnpLU
C9QOUxZcFlYTE0yYzZ3MmtjZUNmZitrCkU1bEVlNUJVbUNUV09UM3c4S1lPNzFLSWVuNE
ZJWTZMMDUrc2JGQmd1Z0ExUE5JdWFubm9UTWtlZTRuMG4rTDQKb3NCM01ZUDhtQmt
RQlAzeE9JNHl3YjREZXUraURyU2pKSHJzQmlIT05Xc0RadXJFaXVJMmdoY1kxeWlyWHl2
UAozVFVOcGNSbC9pVG9zQngxcHJHclk4V09HZVdPeGxZZmcvbWlvNnBuOUYvNWxlQlkr
ZStjSTITMkQ0YXBKWUdpCkwxeHZzVWtGQWdNQkFBRUNnZ0VBZFhCK0xkbk8ySElOTG
o5bWRsb25lUGlHWWVzZ294RGQwci9hQ1Zkank4dlEKTjlwL3FQWkUxek1yall6Ry9kVGhT
MmMwc0QxaTBXSjdwR1IGb0xtdXlWTjltY0FXUTM5SjM0VHZaU2FFSWZWNgo5TE1jUHhN
TmFsNjRLMFRVbUFQZytGam9QSFlhUUxLOERLOUtnNXNrSE5pOWNzMlY5ckd6VWlVZW
tBL0RBUlBTCll3L2ZjUFBacDRuRWVBZml3WTk1R1Ilb1p5V21SU3VKdlNyblBESGtUdW1v
VlVWdkxMRHRzaG9reUxiTWVtN3oKMmJzVmpwSW1GTHJqbGtmQXlpNHg0WjJrV3YyMF
RrdWtsZU1jaVlMbjk4QWxiRi9DSmRLM3QraTRoMTVlR2ZQegpoTnh3bk9QdlVTaDR2Q0o3
c2Q5TmtEUGJvS2JneVVHOXBYamZhRGR2UVFLQmdRRFFLM01nUkhkQ1pKNVFqZWFK
ClFGdXF4cHdnNzhZTjQyL1NwenlUYmtGcVFoQWtyczJxWGx1MDZBRzhrZzlzQkswaHkza
E9zSGgxcXRVK3NHZVAKOWRERHBsUWV0ODZsY2FlR3hoc0V0L1R6cEdtNGFKSm5oNz
VVaTVGZk9QTDhPTm1FZ3MxMVRhUldhNzZxelRyMgphRlpjQ2pWV1g0YnRSTHVwSkgrMj
ZnY0FhUUtCZ1FEQmxVSUUzTnNVOFBBZEYvL25sQVB5VWs1T3lDdWc3dmVyClUycXlrd
XFzYnBkSi9hODViT1JhM05lVmpVM25uRGpHVHBWaE9JeXg5TEFrc2RwZEFjVmxvcG9HO
DhXYk9lMTAKMUdqbnkySmdDK3JVWUZiRGtpUGx1K09lYnRnOXFYcGJMSHBzUVpsMG
hucDBYSFNYVm9CMUliQndnMGEyOFVadApCbFBtWmc2d1BRS0JnRHVlUVV2SDZHYTN
DVUsxNFdmOFhlcFFnMU16M2VvWTBPQm5iSDRvZUZKZmcraEppSXlnCm9RN3hqWldVR
3Blc3AyblRtcHErQWlSNzdyRVhsdlhtOElVU2FsbkNiRGlKY01Pc29RdFBZNS9NczJMRm5L
QTQKaENmL0pWb2FtZm1nZEN0ZGtFMXNlNE9MR2lJVHdEbTRpb0dWZGlwMllnbzFyb2ht
NUpLMUI3MkpBb0dBUW01UQpHNDhXOTVhL0w1eSt5dCsyZ3YvUHM2VnBvMjZlTzRNQ3l
JazJVem9ZWE9lYnNkODJkaC8xT2sybGdHZll2K3VuCnc1YytZUXRSTHlhQmd3MUtpbGhF
ZDBKTWU3cGpUSVpnQWJ0LzVPbnlDak9OVXN2aDJjS2lrQ1Z2dTZsZlBjNkQKckliT2ZlaHh
xV0RZK2Q1TGN1YSt2NzJ0RkxhenJsSlBsRzlOZHhrQ2dZRUF5ellzT3UyMDNRVVV6bUlCR
kwzZAp4Wm5XZ0JLSEo3TnNxcGFWb2RjL0d5aGVycjFDZzE2MmJaSjJDV2RsZkl0VEdtUj
ZZdmxTZEFOOFRwUWhFbUtKCnFBLzVzdHdxNWd0WGVLOVJmMWxXK29xNThRNTBxM
mk1NVdUTThoSDZhTjlaMTltZ0FGdE5VdGNqQUx2dFYxdEYKWSs4WFJkSHJaRnBlWll2N
WkwVW1VbGc9Ci0tLS0tRU5EIFBSSVZBVEUgS0VZLS0tLS0K"

Now create the secrets using the file:

```
$ kubectl create -f nginxsecrets.yaml
$ kubectl get secrets
NAME                  TYPE                                  DATA      AGE
default-token-il9rc   kubernetes.io/service-account-token   1         1d
nginxsecret           Opaque                                2         1m
```

Now modify your nginx replicas to start an https server using the certificate in the secret, and the Service, to expose both ports (80 and 443):

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    protocol: TCP
    name: https
  selector:
    run: my-nginx
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 1
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      volumes:
      - name: secret-volume
        secret:
          secretName: nginxsecret
      containers:
```

```
    - name: nginxhttps
      image: bprashanth/nginxhttps:1.0
      ports:
      - containerPort: 443
      - containerPort: 80
      volumeMounts:
      - mountPath: /etc/nginx/ssl
        name: secret-volume
```

Noteworthy points about the nginx-secure-app manifest:

- It contains both Deployment and Service specification in the same file.
- The nginx server serves HTTP traffic on port 80 and HTTPS traffic on 443, and nginx Service exposes both ports.
- Each container has access to the keys through a volume mounted at /etc/nginx/ssl. This is setup *before* the nginx server is started.

```
$ kubectl delete deployments,svc my-nginx; kubectl create -f ./nginx-secure-app.yaml
```

At this point you can reach the nginx server from any node.

```
$ kubectl get pods -o yaml | grep -i podip
    podIP: 10.244.3.5
node $ curl -k https://10.244.3.5
...
<h1>Welcome to nginx!</h1>
```

Note how we supplied the -k parameter to curl in the last step, this is because we don't know anything about the pods running nginx at certificate generation time, so we have to tell curl to ignore the CName mismatch. By creating a Service we linked the CName used in the certificate with the actual DNS name used by pods during Service lookup. Let's test this from a pod (the same secret is being reused for simplicity, the pod only needs nginx.crt to access the Service):

service/networking/curlpod.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: curl-deploymentspec:
  selector:
    matchLabels:
      app: curlpod
  replicas: 1
  template:
    metadata:
      labels:
        app: curlpod
    spec:
      volumes:
      - name: secret-volume
        secret:
          secretName: nginxsecret
      containers:
      - name: curlpod
        command:
        - sh
        - -c
        - while true; do sleep 1; done
        image: radial/busyboxplus:curl
        volumeMounts:
        - mountPath: /etc/nginx/ssl
          name: secret-volume
```

```
$ kubectl create -f ./curlpod.yaml
$ kubectl get pods -l app=curlpod
NAME                               READY     STATUS    RESTARTS   AGE
curl-deployment-1515033274-1410r   1/1       Running   0          1m
$ kubectl exec curl-deployment-1515033274-1410r -- curl https://my-nginx --cacert
/etc/nginx/ssl/nginx.crt
...
<title>Welcome to nginx!</title>
...
```

Exposing the Service

For some parts of your applications you may want to expose a Service onto an external IP address. Kubernetes supports two ways of doing this: NodePorts and LoadBalancers. The

Service created in the last section already used NodePort, so your nginx HTTPS replica is ready to serve traffic on the internet if your node has a public IP.

```
$ kubectl get svc my-nginx -o yaml | grep nodePort -C 5
  uid: 07191fb3-f61a-11e5-8ae5-42010af00002
spec:
  clusterIP: 10.0.162.149
  ports:
  - name: http
    nodePort: 31704
    port: 8080
    protocol: TCP
    targetPort: 80
  - name: https
    nodePort: 32453
    port: 443
    protocol: TCP
    targetPort: 443
  selector:
    run: my-nginx

$ kubectl get nodes -o yaml | grep ExternalIP -C 1
    - address: 104.197.41.11
      type: ExternalIP
    allocatable:
--
    - address: 23.251.152.56
      type: ExternalIP
    allocatable:
...

$ curl https://<EXTERNAL-IP>:<NODE-PORT> -k
...
<h1>Welcome to nginx!</h1>
```

Let’s now recreate the Service to use a cloud load balancer, just change the Type of my-nginx Service from NodePort to LoadBalancer:

```
$ kubectl edit svc my-nginx
$ kubectl get svc my-nginx
NAME       TYPE       CLUSTER-IP     EXTERNAL-IP       PORT(S)              AGE
my-nginx   ClusterIP  10.0.162.149   162.222.184.144   80/TCP,81/TCP,82/TCP  21s

$ curl https://<EXTERNAL-IP> -k
```

...
<title>Welcome to nginx!</title>

The IP address in the EXTERNAL-IP column is the one that is available on the public internet. The CLUSTER-IP is only available inside your cluster/private cloud network.
Note that on AWS, type LoadBalancer creates an ELB, which uses a (long) hostname, not an IP. It's too long to fit in the standard kubectl get svc output, in fact, so you'll need to do kubectl describe service my-nginx to see it. You'll see something like this:

$ kubectl describe service my-nginx
...
LoadBalancer Ingress:
a320587ffd19711e5a37606cf4a74574-1142138393.us-east-1.elb.amazonaws.com
...