Containers: How do I expose a Kubernetes service to an external network in Bright 7.3/8.0?

There are many ways of exposing a Kubernetes service to an external network. In this article, two methods are described. Each has its advantages and disadvantages.

The methods described were tested with Bright Cluster Manager 7.3 and 8.0. They assume that Bright was configured with a “Type 1” network topology.

Prerequisites

The two methods require the following conditions to be met:

- At least one of the nodes that will be used to run pods has to be connected to the external network.
- Kubernetes must be deployed.
- The flanneld service is configured to always use the internal network interface. This is necessary so all traffic between Kubernetes nodes is done through the internal network, even when the nodes have interfaces in the external network. This can be configured in the Flannel::Host role.

  For example, to configure the head node and all nodes in the default category to always use the eth0 interface for flanneld, the following commands can be run on the head node:

  ```
  # cmsh -c "device roles master; set flannel::host interfacename eth0; commit"

  # cmsh -c "category roles default; set flannel::host interfacename eth0; commit"
  ```

Method 1: Ingress Controller

This method uses the Ingress resource as described in https://kubernetes.io/docs/concepts/services-networking/ingress/.

Containers: How do I expose a Kubernetes service to an external network in Bright 7.3/8.0?

We will show how this is done with an example:

1 - Configure Ingress Controllers

Choose which of nodes from among the Kubernetes nodes are going to be used as Ingress Controllers. Create a category for those nodes by cloning the category which is being used for Kubernetes.

```
# cmsh -c "category; clone default ingress; commit"
```

Assign a label to the ingress category.

```
# cmsh -c "category; roles ingress; set kubernetes::node labels nodeType=ingress; commit"
```

The nodeType=ingress is an arbitrary label. This will be used later to run the IngressController pod only in these nodes.

Enable host networking for the nodes:

```
# cmsh -c "category; roles ingress; set kubernetes::node hostnetworksources *; commit"
```

Assign the ingress category to the nodes with interfaces in the external network.

```
# cmsh -c "device use node001; set category ingress; commit"
```

The nodes have to be rebooted after this operation.

Create an ingress-controller.yml file with the following content:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    k8s-app: nginx-ingress-controller
  name: nginx-ingress-controller
  namespace: kube-system
spec:
  replicas: 1
  template:
    metadata:
      annotations:
        prometheus.io/port: "10254"
        prometheus.io/scrape: "true"
      labels:
        k8s-app: nginx-ingress-controller
    spec:
      containers:
        -
```

```
  args:
    - /nginx-ingress-controller
    - "--default-backend-service=$(POD_NAMESPACE)/default-http-backend"
    - "--kubeconfig=/cm/local/apps/kubernetes/var/etc/kubeconfig"
  env:
    -
      name: POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    -
      name: POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
  image: "gcr.io/google_containers/nginx-ingress-controller:0.9.0-beta.5"
  livenessProbe:
    httpGet:
      path: /healthz
      port: 10254
      scheme: HTTP
    initialDelaySeconds: 10
    timeoutSeconds: 1
  name: nginx-ingress-controller
```

```
    ports:
      -
        containerPort: 80
        hostPort: 80
      -
        containerPort: 443
        hostPort: 443
    readinessProbe:
      httpGet:
        path: /healthz
        port: 10254
        scheme: HTTP
    volumeMounts:
      -
        mountPath: /cm/local/apps/kubernetes/var/etc
        name: kubeconfig
hostNetwork: true
nodeSelector:
  nodeType: ingress
terminationGracePeriodSeconds: 60
volumes:
  -
    hostPath:
      path: /cm/local/apps/kubernetes/var/etc/
```

```
        name: kubeconfig

---

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

  labels:

    k8s-app: default-http-backend

  name: default-http-backend

  namespace: kube-system

spec:

  replicas: 1

  template:

    metadata:

      labels:

        k8s-app: default-http-backend

    spec:

      containers:

        -

          image: "gcr.io/google_containers/defaultbackend:1.0"

          livenessProbe:

            httpGet:

              path: /healthz

              port: 8080

              scheme: HTTP
```

```
        initialDelaySeconds: 30
        timeoutSeconds: 5
      name: default-http-backend
      ports:
      -
        containerPort: 8080
      resources:
        limits:
          cpu: 10m
          memory: 20Mi
        requests:
          cpu: 10m
          memory: 20Mi
    terminationGracePeriodSeconds: 60
---
apiVersion: v1
kind: Service
metadata:
  labels:
    k8s-app: default-http-backend
  name: default-http-backend
  namespace: kube-system
spec:
  ports:
```

```
-
  port: 80
  targetPort: 8080
selector:
  k8s-app: default-http-backend
```

Deploy the Ingress Controller by running the following command:

```
# kubectl create -f ingress-controller.yml
```

To test that the ingress controllers were deployed, run the following command, where <IP> is the IP address in the external network of any of the nodes used as Ingress Controllers.

```
# curl http://<IP>
default backend - 404
```

2 - Run a pod

Create a test-pod.yml file with the following content:

```
 ---
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: test
  name: test
spec:
  containers:
    -
      image: nginx
      imagePullPolicy: IfNotPresent
      name: test
      ports:
        -
          containerPort: 80
```

Run the pod by running the following commands:

```
# module load kubernetes

# kubectl create -f test-pod.yml
```

3 - Create a service exposed as ClusterIP (the default option)

Create a test-service.yml file with the following content:

```
---
apiVersion: v1
kind: Service
metadata:
  name: test-service
spec:
  ports:
```

```
  -
    port: 8888
    protocol: TCP
    targetPort: 80
  selector:
    app: test
```

Create the service by running the following commands:

```
# module load kubernetes

# kubectl create -f test-service.yml
```

4 - Configure an Ingress resource for the service

Create a test-ingress.yml file with the following content:

```
 ---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
spec:
  rules:
    -
      http:
        paths:
          -
            backend:
              serviceName: test-service
              servicePort: 80
            path: /
```

Create the Ingress resource by running the following commands:

```
# module load kubernetes

# kubectl create -f test-ingress.yml
```

In order to access the service from the external network, users can make an HTTP request to <hostname>, where <hostname> must resolve to the IP address on the external network of any of the nodes that are running Ingress Controllers.

This method requires more configuration than method 2, and makes use of Ingress Controllers which are still a beta feature of Kubernetes (as of May 2017). For each service that has to be exposed, one or more Ingress resource has to be defined and added. Configuring Ingress resources is very similar to configuring rules in a Reverse Proxy and allows for great flexibility.

Method 2: NodePort

This method requires exposing the service as NodePort (described at https://kubernetes.io/docs/concepts/services-networking/service/#type-nodeport). When a service is exposed in this way, then a port number is assigned to it. The service can then be accessed on any of the nodes, at the assigned port.

An example follows:

1 - Run a pod

Create a test-pod.yml file with the following content:

```
---
apiVersion: v1
```

```
kind: Pod
metadata:
  labels:
    app: test
  name: test
spec:
  containers:
    -
      image: nginx
      imagePullPolicy: IfNotPresent
      name: test
      ports:
        -
          containerPort: 80
```

Run the pod by running the following commands:

```
# module load kubernetes

# kubectl create -f test-pod.yml
```

2 - Create a service exposed as NodePort

Create a test-service.yml file with the following content:

```
---
apiVersion: v1
kind: Service
metadata:
  name: test-service
spec:
  ports:
    -
      port: 80
```

```
    protocol: TCP
    targetPort: 80
  selector:
    app: test
  type: NodePort
```

Create the service by running the following commands:

```
# module load kubernetes

# kubectl create -f test-service.yml
```

The following message is then displayed:

```
You have exposed your service on an external port on all nodes in your

cluster.  If you want to expose this service to the external internet, you may

need to set up firewall rules for the service port(s) (tcp:32284) to serve traffic.

See http://releases.k8s.io/release-1.3/docs/user-guide/services-firewalls.md for more details.

service "test-service" created
```

To verify which port was assigned to the service, the following command can be run:

```
# kubectl describe service test-service

Name:           test-service
```

```
Namespace:        default

Labels:           <none>

Selector:         app=test

Type:             NodePort

IP:               10.150.96.51

Port:             <unset>  8888/TCP

NodePort:         <unset>  30844/TCP

Endpoints:        172.29.40.2:80

Session Affinity:  None

No events.
```

The service can then be accessed on the assigned NodePort on any of the Kubernetes nodes. For example:

```
# curl http://node001:32674

<!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>
```

In order to access the service from the external network, the users can
use <IP>:<NodePort> where <IP> is the IP address on the external network of any of the nodes.

This method is easy to implement. However, it is not scalable, as each service will need to use a different port and users need to be aware of which port they have to use for each service.

Troubleshooting

In this section we mention some common problems that could occur when following these steps (specially when using method 1)

1 - How can I troubleshoot the Ingress Controller?

Please note that you should refer to the Kubernetes and to the Nginx ingress controller documentation for details on how it works. The documentation could contain updated information that is not available here.

In a nutshell the Nginx ingress controller is a container running Nginx. This Nginx server is configured as a reverse proxy to pass the requests to the required services. The container pulls the Ingress resources defined in Kubernetes (by accessing the API server) and writes the corresponding configuration to the Nginx configuration file.

To verify what Ingress resources are defined, run the following commands in the head node:

```
# module load kubernetes
# kubectl get ing
NAME          HOSTS     ADDRESS     PORTS     AGE
test-ingress  *         10.141.0.1  80        22h
```

```
test-ingress2   *         10.141.0.1   80        1h
```

To view the details of a particular Ingress resource:

```
# kubectl describe ing test-ingress

Name:             test-ingress

Namespace:        default

Address:          10.141.0.1

Default backend:  default-http-backend:80 (172.29.83.2:8080)

Rules:

  Host  Path  Backends

  ----  ----  --------

  *

        /     test-service:80 (<none>)

Annotations:

Events:  <none>
```

To view the Nginx configuration file inside the container, first get the name of the pod which runs the ingress controller:

```
# kubectl get pods --namespace=kube-system | grep nginx-ingress-controller

nginx-ingress-controller-3976493878-1zqs8   1/1       Running   0          4d
```

Then run the cat command inside it:

```
# kubectl --namespace=kube-system exec nginx-ingress-controller-3976493878-1zqs8 cat /etc/nginx/nginx.conf
```

You can redirect the previous output to a file in order to analyze it in detail. You can also get a shell inside the container, with:

```
# kubectl --namespace=kube-system exec -it nginx-ingress-controller-3976493878-1zqs8 -- /bin/bash
```

With a shell inside the container you could, for example, redirect the Nginx log files to a different file in order to analyze them (by default the log files are symbolic links to /dev/null)

If in Kubernetes you have a service defined like this:

```
# kubectl describe service test-service

Name:              test-service

Namespace:         default

Labels:            <none>

Annotations:       <none>

Selector:          app=test

Type:              ClusterIP

IP:                10.150.107.161

Port:              <unset>  8888/TCP
```

```
Endpoints:        172.29.83.3:80

Session Affinity:    None

Events:           <none>
```

You can find in the Nginx configuration file an upstream defined like this:

```
   upstream default-test-service-80 {

       least_conn;

       server 172.29.83.3:80 max_fails=0 fail_timeout=0;

   }
```

If for that service you have an Ingress resource defined like this:

```
# kubectl describe ing test-ingress

Name:             test-ingress

Namespace:        default

Address:          10.141.0.1

Default backend:    default-http-backend:80 (172.29.83.2:8080)

Rules:

  Host    Path    Backends

  ----    ----    --------

  *

        /     test-service:80 (<none>)
```

```
Annotations:

Events:    <none>
```

In the Nginx configuration you will find:

```
        # enforce ssl on server side
        if ($pass_access_scheme = http) {
            return 301 https://$best_http_host$request_uri;
        }
        location / {
            set $proxy_upstream_name "default-test-service-80";
            port_in_redirect off;

            client_max_body_size                    "1m";

            proxy_set_header Host                   $best_http_host;

            # Pass the extracted client certificate to the backend

            # Pass Real IP
            proxy_set_header X-Real-IP              $remote_addr;
```

```
        # Allow websocket connections
        proxy_set_header                    Upgrade           $http_upgrade;
        proxy_set_header                    Connection        $connection_upgrade;

        proxy_set_header X-Forwarded-For        $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Host       $best_http_host;
        proxy_set_header X-Forwarded-Port       $pass_port;
        proxy_set_header X-Forwarded-Proto      $pass_access_scheme;
        proxy_set_header X-Original-URI         $request_uri;
        proxy_set_header X-Scheme               $pass_access_scheme;

        # mitigate HTTPoxy Vulnerability
        # https://www.nginx.com/blog/mitigating-the-httpoxy-vulnerability-with-nginx/
        proxy_set_header Proxy                  "";

        # Custom headers

        proxy_connect_timeout               5s;
        proxy_send_timeout                  60s;
        proxy_read_timeout                  60s;

        proxy_redirect                      off;
```

```
        proxy_buffering                 off;

        proxy_buffer_size               "4k";

        proxy_buffers                   4 "4k";



        proxy_http_version              1.1;



        proxy_cookie_domain             off;

        proxy_cookie_path               off;



        proxy_pass http://default-test-service-80;

    }
```

According to this configuration, a HTTP request will be answered with a redirect to HTTPS. When the site is requested via HTTPS, Nginx will then pass the request to the defined service. An example of the expected behavior can be shown by using curl (in the following example we use node001 to reach the node with the ingress controller:

```
# curl http://node001 --head

HTTP/1.1 301 Moved Permanently

Server: nginx/1.11.12

Date: Tue, 27 Jun 2017 12:32:58 GMT

Content-Type: text/html

Content-Length: 186

Connection: keep-alive
```

Location: https://node001/

Strict-Transport-Security: max-age=15724800; includeSubDomains;

In the previous output we can see that the server answered with 301, specifying that we should redirect the request to https://node001/. Now we will follow the redirect:

curl http://node001 --location

curl: (60) Issuer certificate is invalid.

More details here: http://curl.haxx.se/docs/sslcerts.html

curl performs SSL certificate verification by default, using a "bundle"

of Certificate Authority (CA) public keys (CA certs). If the default

bundle file isn't adequate, you can specify an alternate file

using the --cacert option.

If this HTTPS server uses a certificate signed by a CA represented in

the bundle, the certificate verification probably failed due to a

problem with the certificate (it might be expired, or the name might

not match the domain name in the URL).

If you'd like to turn off curl's verification of the certificate, use

the -k (or --insecure) option.

In this case we get an invalid certificate error because there is no valid certificate in the server for node001. We can safely ignore this error so we run curl with the -k option:

```
# curl http://node001 --location -k

<!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
```

nginx.com.

Thank you for using nginx.

</body>

</html>

Here the server returned the HTML generated by test-service, so everything is working correctly.

2 - I configured a service using method 1. When accessing the service I get a site unavailable error.

Please verify the configuration as mentioned in the previous section. A common error is doing this:

curl http://10.141.0.1 --location -k

curl: (35) Encountered end of file

The error happens because an IP address is used in the request instead of a name. As described in the procedure, you have to use a name that resolves to the node’s external network IP address

3 - I configured one of the above methods. I have a node with an interface in the external network. Do I need to configure some special firewall rules in this node?

This depends on which method was used. If an Ingress Controller was implemented (method 1) then the

node must allow incoming traffic to ports 80 and 443.

If the service was exposed as NodePort (method 2), then for each service a firewall rule must be added to allow the node to FORWARD traffic to the pod network and to the port the container is listening to. For example, in the service we configured in Method 2:

```
# kubectl describe service test-service

Name:              test-service

Namespace:         default

Labels:            <none>

Selector:          app=test

Type:              NodePort

IP:                10.150.96.51

Port:              <unset>  8888/TCP

NodePort:          <unset>  30844/TCP

Endpoints:         172.29.40.2:80

Session Affinity:  None

No events.
```

The kube-proxy service (part of Kubernetes) will configure iptables in every node so all packets destined to port 30844 are forwarded to the IP address 172.29.40.2 (IP address of the pod) and port 80 (where

the service is listening inside of the pods network) This means that if a firewall is configured in a node, the administrator should make sure that this traffic will be allowed.

Unique solution ID: #1357
Author: Carlos Pintado
Last update: 2018-04-24 15:03