Containers: How do I expose a Kubernetes service to an external network in Bright 8.1?

There are many ways of exposing a Kubernetes service to an external network.

This article describes two methods of exposing Kubernetes to an external network.

The methods described were tested with Bright Cluster Manager 8.1. They assume that Bright was configured with a “Type 1” network topology.

Prerequisites

Both methods described in this article require Kubernetes to be deployed previously, as described
at http://support.brightcomputing.com/manuals/8.1/admin-manual.pdf#subsection.8.3.1.

Both methods will configure the service in such a way that it will be accessible on every node of the Kubernetes cluster. It is therefore a good idea to make sure that one of the nodes has a network interface connected to the external network of the cluster. This way the users of the service will be able to reach the external network without having to connect to the head node.

Method 1: NGinx Ingress Controller

This method uses the Ingress resource as described
in https://kubernetes.io/docs/concepts/services-networking/ingress/ and for that Ingress Controllers are deployed as described in
the https://github.com/kubernetes/ingress-nginx/blob/master/docs/deploy/index.md article.

This method only supports HTTP/HTTPS services.

This example shows how it can be done:

1 - Deploy Ingress Controllers

First install git on the head node.

```
# yum install git
```

Then clone the repository with the Ingress Controller that will be deployed.

```
# git clone https://github.com/kubernetes/ingress-nginx.git
```

Run the following commands in order to deploy the Ingress Controller.

```
# module load kubernetes
# cd ingress-nginx/deploy/
# kubectl apply -f namespace.yaml
# kubectl apply -f default-backend.yaml
# kubectl apply -f configmap.yaml
# kubectl apply -f tcp-service-configmap.yaml
# kubectl apply -f udp-service-configmap.yaml
# kubectl apply -f rbac.yaml
# kubectl apply -f with-rbac.yaml
```

Edit the provider/baremetal/service-nodeport.yaml file to specify explicitly the nodePort property, as in the following example:

```
apiVersion: v1
kind: Service
metadata:
  name: ingress-nginx
  namespace: ingress-nginx
spec:
  type: NodePort
```

```
  ports:
  - name: http
port: 80
targetPort: 80
protocol: TCP
nodePort: 30080
  - name: https
port: 443
targetPort: 443
protocol: TCP
nodePort: 30443
  selector:
app: ingress-nginx
```

Run the following command:

```
# kubectl apply -f provider/baremetal/service-nodeport.yaml
```

Verify in which ports the Ingress Controller can be accessed:

```
# kubectl describe service ingress-nginx --namespace=ingress-nginx | grep NodePort
```

To check if the pods with the Ingress Controllers are running, the following command can be run on the head node:

```
# kubectl get pods --all-namespaces -l app=ingress-nginx
```

To test that the ingress controllers are accessible, run the following command, where <IP> is the IP address in the external network of any of the nodes in the Kubernetes cluster

and <PORT> is the HTTP port of the ingress-nginx service, as seen in the previous command.

curl http://<IP>:<PORT>

default backend - 404

The above examples mean that traffic sent to the ports specified by nodePort for HTTP and HTTPS of any node should be forwarded to port 80 and 443 of the Ingress Controller respectively. The Ingress Controller should answer with HTTP code 404 if no Ingress rules are defined yet.

2 - Run a pod

Create a test-pod.yaml 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 apply -f test-pod.yaml
```

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

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

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

```
  targetPort: 80
  selector:
app: test
```

Create the service by running the following commands:

```
# module load kubernetes
# kubectl apply -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
  annotations:
nginx.ingress.kubernetes.io/ssl-redirect: "false"
spec:
  rules:
-
  http:
    paths:
```

```
-
  backend:
    serviceName: test-service
    servicePort: 80
  path: /
```

Create the Ingress resource by running the following commands:

```
# module load kubernetes
# kubectl apply -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 in the Kubernetes cluster.

In this method for each service that has to be exposed, one or more Ingress resources 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:

Containers: How do I expose a Kubernetes service to an external network in Bright 8.1?

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 apply -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 apply -f test-service.yml
```

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>

Annotations:            kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"v1","kind":"Se
rvice","metadata":{"annotations":{},"name":"test-service","namespace":"default"},"spec":{"ports":[
{"port":80,"protocol":"T...

Selector:               app=test

Type:                   NodePort

IP:                     10.150.108.152

Port:                   <unset>  80/TCP

TargetPort:             80/TCP

NodePort:               <unset>  32640/TCP

Endpoints:              172.29.0.7:80

Session Affinity:       None

External Traffic Policy:  Cluster

Events:                 <none>
```

The service can then be accessed on the assigned NodePort on any of the nodes in the

Kubernetes cluster. 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>
```

Thank you for using nginx.

</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 and it also supports non-HTTP services. 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 (especially 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.

Briefly, 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.

Containers: How do I expose a Kubernetes service to an external network in Bright 8.1?

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:
Default backend:  default-http-backend:80 (<none>)
Rules:
  Host  Path Backends
  ----  ---- --------
  *
    /   test-service:80 (<none>)
Annotations:
Events:
  TypeReason  Age      From               Message
```

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

Normal UPDATE 21m (x4 over 2h) nginx-ingress-controller Ingress default/test-ingress

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=ingress-nginx | grep nginx-ingress-controller

nginx-ingress-controller-777686bfbc-jmsgb 1/1 Running 2 14d

Then run the cat command inside it:

kubectl --namespace=ingress-nginx exec nginx-ingress-controller-777686bfbc-jmsgb 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=ingress-nginx exec -it nginx-ingress-controller-777686bfbc-jmsgb -- /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 there is a service defined as follows:

kubectl describe service test-service

Name: test-service

Namespace: default

Labels: <none>

Annotations: kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"v1","kind":"Service","metadata":{"annotations":{},"name":"test-service","namespace":"default"},"spec":{"ports":[{"port":80,"protocol":"T...

Selector: app=test

Type: ClusterIP

IP: 10.150.76.151

Port: <unset> 80/TCP

TargetPort: 80/TCP

Endpoints: 172.29.0.7:80

Session Affinity: None

Events: <none>

Then in the Nginx configuration file an upstream section is defined like this:

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

        keepalive 32;

        server 172.29.0.7:80 max_fails=0 fail_timeout=0;

}
```

If for that service an Ingress resource is defined like this:

```
# kubectl describe ing test-ingress
Name:         test-ingress
Namespace:    default
Address:
Default backend:  default-http-backend:80 (<none>)
Rules:
  Host  Path Backends
  ----  ---- --------
  *
    /   test-service:80 (<none>)
Annotations:
Events:  <none>
```

then in the Nginx configuration you will find:

```
location / {
        log_by_lua_block {
        }
        if ($scheme = https) {
                more_set_headers                "Strict-Transport-Security:
max-age=15724800; includeSubDomains";
        }
```

```
port_in_redirect off;

set $proxy_upstream_name "default-test-service-80";

set $namespace  "default";
set $ingress_name   "test-ingress";
set $service_name   "test-service";

client_max_body_size                "1m";

proxy_set_header Host               $best_http_host;

# Pass the extracted client certificate to the backend

# Allow websocket connections
proxy_set_header                    Upgrade       $http_upgrade;

proxy_set_header                    Connection    $connection_upgrade;

proxy_set_header X-Real-IP          $the_real_ip;

proxy_set_header X-Forwarded-For    $the_real_ip;

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;
```

```
            # Pass the original X-Forwarded-For

            proxy_set_header X-Original-Forwarded-For $http_x_forwarded_for;


            # mitigate HTTPoxy Vulnerability

            # https://www.nginx.com/blog/mitigating-the-httpoxy-vulnerability-with-nginx/

            proxy_set_header Proxy            "";


            # Custom headers to proxied server


            proxy_connect_timeout             5s;

            proxy_send_timeout                60s;

            proxy_read_timeout                60s;


            proxy_buffering                   "off";

            proxy_buffer_size                 "4k";

            proxy_buffers                     4 "4k";

            proxy_request_buffering           "on";


            proxy_http_version                1.1;


            proxy_cookie_domain               off;

            proxy_cookie_path                 off;


            # In case of errors try the next upstream server before returning an error

            proxy_next_upstream               error timeout invalid_header http_502 http_503
http_504;

            proxy_next_upstream_tries         0;


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

```
            proxy_redirect                  off;

        }
```

The service can then service can be accessed in this way:

```
# curl http://node001:30080
<!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 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?

In case a firewall is configured on the node, a couple of rules have to be configured. For the following explanation we assume that the user needs to access a service which was exposed as NodePort. The Ingress Controller is a particular case of this, as it is a NodePort service itself.

First examine the description of the service:

kubectl describe service ingress-nginx --namespace=ingress-nginx

Name: ingress-nginx

Namespace: ingress-nginx

Labels: <none>

Annotations: kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"v1","kind":"Service","metadata":{"annotations":{},"name":"ingress-nginx","namespace":"ingress-nginx"},"spec":{"ports":[{"name":"http","n...

Selector: app=ingress-nginx

Type: NodePort

IP: 10.150.147.104

Port: http 80/TCP

TargetPort: 80/TCP

NodePort: http 30080/TCP

Endpoints: 172.29.0.9:80

Port: https 443/TCP

TargetPort: 443/TCP

NodePort: https 30443/TCP

Endpoints: 172.29.0.9:443

Session Affinity: None

External Traffic Policy: Cluster

Events: <none>

The preceding service receives traffic on ports 30080 and 30443, and then forwards that traffic to ports 80 and 443 on the 172.29.0.9 host (using a DNAT rule configured by the kube-proxy service). This means that in the firewall of the node, the following traffic has to be allowed:

1 - incoming traffic to ports 30080 and 30443

2 - forward traffic to ports 80 and 443 of the 172.29.0.9 host

Please note that the kube-proxy service running on the node already creates rules in the FORWARD chain to allow packets to be forwarded to the pods network. The administrator just has to beware of configuring the firewall in a way that would not allow this traffic.

Unique solution ID: #1430
Author: Carlos Pintado
Last update: 2018-05-28 17:12