community | learn ask attend tools

# Configure and Manage a Kubernetes HAProxy Ingress Controller

November 5th 2016 kubernetes, ingress controller, haproxy, nginx

Matt Baldwin

## Table of Contents

## Introduction

Almost everyone who is deploying an application would like the app to be accessible to other people on the public internet.

- If you are an independent developer working with a cloud provider, then you'll ask, what's my public IP address, and what ports are exposed?

- If you are a developer in a corporation with private data center resources, you'll ask, how do I get the load-balancer rules configured for my set of internal hosts (and who do I need to talk to)?
- If you are working an application deployed inside a Kubernetes cluster, you'll ask, what's going on here at all? How do I let people get in?

If you need to expose your Kubernetes services to the world, Ingresses are the way to go. At the time of this writing Ingress is only available in beta, so let's see the alternatives first.

## Exposing services to the world

Your application in a Kubernetes cluster probably contains these items:

### Pod

Your carefully-designed set of applications has already been put into a set of containers-sharing-resources as a Kubernetes pod, so how do those processes communicate with others?

The pod exposes endpoints usually at the overlay network. That network is not reachable from outside the cluster. You can also use `hostPort` to expose the endpoint at the host IP, or set `hostNetwork: true` to use the host's network interface from your pod. But in both scenarios you should take extra care to avoid port conflicts at the host, and possibly some issues with packet routing and name resolutions. So let's assume that we want a risk-free and scalable kubernetes cluster, and we are properly using pods exposing their functions to the container ports.

### Service

Kubernetes services primarily work to interconnect different pod-based applications. They're designed to help make a microservices system scale well internally. They are not primarily intended for external access, but there are some accepted ways to expose services to external clients.

Let's simplify services as a routing, balancing and discovery mechanism for the pod's endpoints. Services target pods using selectors, and can map container ports to service ports. A service exposes one or more ports, although usually you will find that only one is defined.

A service can be exposed using 3 `ServiceType` choices:

- `ClusterIP`: The default, only exposed within the cluster.
- `NodePort`: Uses a port from a cluster defined range (usually 30000-32767) at every cluster node to expose the service.
- `LoadBalancer`: Setting your cluster kubelet with the parameter `--cloud-provider=the-cloud-provider` brings some goodies, including the ability to use the provider's load balancer.

If we choose `NodePort` to expose our services, we will still need an external proxy that uses DNAT to expose more friendly ports. That element will also need to balance on the cluster nodes, which in turn will balance again through the service to the internal pods. There is no easy way of adding TLS or simple host header routing rules to the external service.

Choosing `LoadBalancer` is probably the easiest of all methods to get your service exposed to the internet. The problem is that there is no standard way of telling a Kubernetes service about the elements that a balancer requires, again TLS and host headers are left out.

## Endpoints

Endpoints are usually automatically created by services, unless you are using headless services and adding the endpoints manually. An endpoint is a host:port tuple registered at Kubernetes, and in the service context it is used to route traffic. The service tracks the endpoints as pods that match the selector are created, deleted and modified. Individually, endpoints are not useful to expose services, since they are to some extent ephemeral objects.

- If you can rely on your cloud provider to correctly implement the LoadBalancer for their API, to keep up-to-date with Kubernetes releases, and you are happy with their management interfaces for DNS and certificates, then setting up your services as type `LoadBalancer` is quite acceptable. On the other hand, if you want to manage load balancing systems manually and set up port mappings yourself, `NodePort` is a low-complexity solution. If you are directly using `Endpoints` to expose external traffic, perhaps you already know what you are doing (but consider that you might have made a mistake, there could be another option).

Given that none of these elements has been originally designed to expose services to the internet, their functionality may seem limited for this purpose.

## Introducing Ingress

The Ingress resource is a set of **rules** that map to Kubernetes **services**. Sure, there will be a more complex definition, and I'll be missing some points, but I still think that "rules to services mapping" is the best way to understand what an Ingress does.

Ingress resources are defined purely within Kubernetes as a object that other entities can watch and respond to.

## What rules?

Supported rules at this beta stage are:

- host header: So we can forward traffic based on domain names.
- paths: Looks for a match at the the beginning of the path.

- TLS: If the ingress adds TLS, HTTPS and a certificate configured through a secret will be used.

  When no host header rules are included at an Ingress, requests without a match will use that Ingress and be mapped to the backend service. You will usually do this to send a 404 page to requests for sites/paths which are not sent to the other services.

## What services?

Ingress tries to match requests to rules, and forwards them to backends, which are composed of a service and a port (remember that a service can contain multiple ports.

To summarize: An Ingress defines how to take a request and (based on host/path/tls) send it to a backend.

## Introducing Ingress Controllers

In order to grant (or remove) access, some entity must be watching and responding to changes in the services, pods and Ingresses. That entity is the Ingress controller. While the Ingress controller does work directly with the Kubernetes API, watching for state changes, it is not coupled as closely to the Kubernetes source code as the cloud-provider LoadBalancer implementations.

Ingress controllers are applications that watch Ingresses in the cluster and configure a balancer to apply those rules. Typically you will use an existing Ingress controller that controls a third party balancer like HAProxy, NGINX, Vulcand or Traefik, updating configuration as the Ingress and their underlying elements change.

Ingress controllers will usually track and communicate with endpoints behind services instead of using services directly. This way some network plumbing is avoided, and we can also manage the balancing strategy from the balancer.

## Ingress Controller extensions

As of this writing, Ingress doesn't support TCP balancing, balancing policies, rewriting, SNI, and many other common balancer configuration parameters. Ingress controller developers have extended the Ingress definition using annotations, but be aware that those annotations are bound to the controller implementation. This means that the Ingress resource will evolve, and might make some of these annotations obsolete in the future.

## Using Ingresses

This example will be a demonstration of Ingress usage. For the demo, we will create in our cluster:

- A backend that will receive requests for `domain1.io`
- A pair of backends that will receive request for `domain2.io`
  - One whose path begins with `/path1`
  - One whose path begins with `/path2`
- A default backend that shows a 404 page

## Backends

You can use any website or service in a container as a backend. We will create three backends.

**echoheaders**: pod/service which contains a webpage that shows information about the received request.

```
---
apiVersion: v1
kind: ReplicationController
```

```
metadata:
  name: echoheaders
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: echoheaders
    spec:
      containers:
      - name: echoheaders
        image: gcr.io/google_containers/echoserver:1.4
        ports:
        - containerPort: 8080
        readinessProbe:
          httpGet:
            path: /healthz
            port: 8080
          periodSeconds: 1
          timeoutSeconds: 1
          successThreshold: 1
          failureThreshold: 10
---
apiVersion: v1
kind: Service
metadata:
  name: echoheaders
  labels:
    app: echoheaders
spec:
  ports:
  - port: 80
    targetPort: 8080
    protocol: TCP
    name: http
  selector:
    app: echoheaders
```

**default-http-backend**: pod/service which is a simple 404 static page.

```
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: default-http-backend
spec:
  replicas: 2
  selector:
    app: default-http-backend
  template:
    metadata:
      labels:
        app: default-http-backend
    spec:
      terminationGracePeriodSeconds: 60
      containers:
      - name: default-http-backend
        # Any image is permissable as long as:
        # 1. It serves a 404 page at /
        # 2. It serves 200 on a /healthz endpoint
        image: gcr.io/google_containers/defaultbackend:1.0
        livenessProbe:
          httpGet:
            path: /healthz
            port: 8080
            scheme: HTTP
          initialDelaySeconds: 30
          timeoutSeconds: 5
        ports:
        - containerPort: 8080
        resources:
          limits:
            cpu: 10m
            memory: 20Mi
          requests:
            cpu: 10m
            memory: 20Mi
---
apiVersion: v1
kind: Service
metadata:
  name: default-http-backend
  labels:
    app: default-http-backend
spec:
  ports:
  - port: 80
    targetPort: 8080
    protocol: TCP
    name: http
  selector:
    app: default-http-backend
```

**game2048**: A pod/service which contains a static web page game

```
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: game2048
  labels:
    name: game2048
spec:
  replicas: 2
  selector:
    name: game2048
  template:
    metadata:
      labels:
        name: game2048
        version: stable
    spec:
      containers:
      - name: game2048
        image: alexwhen/docker-2048
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: game2048
  labels:
    name: game2048
spec:
  ports:
  - port: 80
    targetPort: 80
  selector:
    name: game2048
```

Create the backends of your choice with `kubectl create -f ...`

## Ingress Controller

For this example we will be using standard functionality only, so you should be able to use any Ingress controller. I'm using the HAProxy Ingress controller, which is included by default with every StackpointCloud cluster build at stackpoint.io

The HAProxy Ingress controller at Stackpoint is already configured and listening to changes in your Kubernetes cluster. If you prefer to get NGINX, it can also be deployed in any cluster following these instructions

## Ingresses

Let's start by creating the `game2048` service at the balancer.

```
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: game-ingress
spec:
  rules:
  - host: game.domain1.io
    http:
      paths:
      - path:
        backend:
          serviceName: game2048
          servicePort: 80
```

That's it. We need now to reach HAPRoxy node using the host header `game.domain1.io`:

- Use your own domain, and point the A record to the HAProxy node's IP address.
- Use `curl -H "Host: game.domain1.io" real.server.address`
- Install a browser plugin to add host header, like Virtual-Hosts for Chrome.

If you use a browser, you should be shown the 2048 game. But if you try the IP address without that host header, you wil get a 503 error.

We'll be fixing that with our next Ingress, the default backend:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: default-http-backend
spec:
    backend:
      serviceName: default-http-backend
      servicePort: 80
```

Every request that doesn't match an existing rule will be served by the `default-http-backend`. Now if you try the IP address without the game host header, you should get a simple 404 page.

Finally, let's use the flexible `echoheaders` service to add some path matching:

```
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: echoheaders
spec:
  rules:
  - host: domain2.io
    http:
      paths:
      - path: /path1
        backend:
          serviceName: echoheaders
          servicePort: 80
      - path: /path2
        backend:
          serviceName: echoheaders
          servicePort: 80
```

In a real world scenario you will use different services for each path, but for us to test it's ok to use the echoheaders service.

To test all Ingresses try these requests:

- balancer node IP –> will be sent to `default-http-backend` (404 default)
- domain1.io –> will be sent to `game2048`
- domain1.io/path1 –> will be sent to `game2048` (and it will fail with 404, since game2048 has no resource at that path)
- domain2.io/path1 –> will be sent to `echoheaders`
- domain2.io/path2 –> will be sent to `echoheaders`
- domain2.io/path3 –> will be sent to `default-http-backend` (404 default)
- domain2.io –> will be sent to `default-http-backend` (404 default)

Ingresses are simple and very easy to deploy, and really fun to play with. However, when you plan your Ingresses for production some other factors arise:

- High Availability
- SSL
- Custom configuration
- Troubleshooting

But we will leave those questions for a future post. Thanks for reading!

Showing 5 comments.

Fede Diaz *November 16th 2016 18:14*

Hi there, I've been fighting with something with this for a few days. I just follow the instructions to dedploy Kubernetes over 3 Ubuntu 16.04 machines (1 master, 2 nodes) with waeve as network overlay. Everything looks great when the cluster is up and running. Then I want to deploy a NGiNX Ingress controller following the instructions https://github.com/nginxinc/kubernetes-ingress/tree/master/examples/complete-example. And again everything looks up and running. But when I try to connect to the controller throught node's external IP ends with connection refused. I don't know what I'm doing wrong or there is no chance for ingress controller on bare metal. I also opened a thread in kubernetes users group: https://groups.google.com/forum/#!topic/kubernetes-users/arfGJnxlauU Please, can you help me? Thank you!!

Michael Draper *January 18th 2017 12:34*

How do you redirect http to https using this controller?

Jude Zhu *January 28th 2017 07:03*

https://github.com/kubernetes/contrib/tree/master/ingress/controllers/nginx works for me

but you need modify the rc.yaml a little bit check the part of nginx-ingress-controller ReplicationController deployment

they use the hostport , like below ports:

```
  - containerPort: 80
    hostPort: 80
  - containerPort: 443
    hostPort: 443
```

but there is no "hostPort: true" in the yaml file , add it here as below

```
spec:
  terminationGracePeriodSeconds: 60
  hostNetwork: true
  containers:
  - image: gcr.io/google_containers/nginx-ingress-controller:0.8.3
```

redeploy it, and it should open the 80 ,433 port on the real server ip which hold the ingress-controller container

let me know if you have more question , I'll try my best

Tarun Prakash *March 6th 2017 18:01*

@Fede Diaz

I have exactly the same issue, connection refused. followed this
https://github.com/nginxinc/kubernetes-ingress/tree/master/examples/complete-example

@Jude Zhu

I made changes to nginx-controller as suggested ( hostNetwork: true ) and redeployed . But still same issue ( i am using virtual box with master and single node )

Please help :(

Fede Diaz *April 10th 2018 20:22*

Thanks @Tarun

ADD A COMMENT, LOG IN

kubernetes community

Learn
Ask
Attend
Tools

Code of conduct

STACKPOINTCLOUD