HAProxy in Kubernetes

Supercharge Your Ingress Routing

© 2020 HAProxy Technologies

Table of Contents

Introduction 3

The Ingress Pattern 5

Install the HAProxy Kubernetes Ingress Controller with Helm 13

Routing Traffic 22

TLS with Let's Encrypt 28

Multi-tenant Kubernetes Clusters 39

Kubernetes Deployment Patterns 56

Where Next 75

Introduction

By now, it's clear that Kubernetes is a force within the technology landscape, enabling sophisticated orchestration of containerized services across a cluster of servers. It's also clear that organizations need simple, yet powerful, components to solve problems like routing traffic into their Kubernetes clusters. From its inception, the platform was built for this kind of extensibility, with the foresight that users would desire ways to integrate their favorite tools.

In this book, you will learn how to combine the power of the HAProxy load balancer with Kubernetes. HAProxy has been recast as a Kubernetes ingress controller, which is a Kubernetes-native construct for traffic routing. The controller is maintained as a distinct project, with a regular release cycle, a growing community of developers, and an increasing number of converts who favor it over other ingress controllers.

As you read this book, consider your use case. Do you require enhanced security? The HAProxy Kubernetes Ingress Controller lets you enforce rate limits and you can whitelist client IP addresses to better control access. Do you want to supercharge your Kubernetes Ingress routing? HAProxy is known as the world's fastest software load balancer and has been benchmarked against alternatives like NGINX, Envoy, and Traefik. Do you need detailed observability in order to graph trends and make informed decisions? HAProxy comes with a built-in Stats page that

provides many counters and gauges for monitoring traffic, and these are also exposed by a Prometheus endpoint.

With 20 years of ongoing development under its belt, the HAProxy project is stronger than ever, and it's no surprise that it's found a niche within the Kubernetes platform. We hope that this book gives you all of the knowledge you need to start using it. Community members like you drive us to make it even better, so we welcome you to join us on Slack, follow us on Twitter, and send us your questions!

slack.haproxy.org/

twitter.com/haproxy

github.com/haproxytech/kubernetes-ingress/

The Ingress Pattern

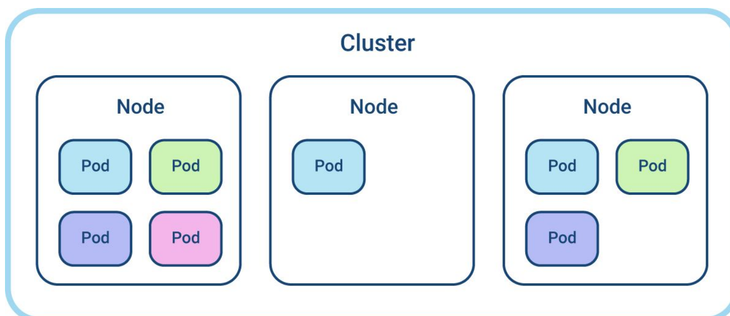
Containers allow cross-functional teams to share a consistent view of an application as it flows through engineering, quality assurance, deployment and support phases. The Docker CLI gives you a common interface through which you can package, version, and run your applications. What's more, with orchestration platforms like Kubernetes, the same tools can be leveraged to manage services across isolated Dev, Staging and Production environments.

This technology is especially convenient for teams working with a microservices architecture, which produces many small, but specialized, services. However, the challenge: How do you expose your containerized services outside of the container network?

In this chapter, you'll learn about the HAProxy Kubernetes Ingress Controller, which is built upon HAProxy, the world's fastest and most widely used software load balancer. As you'll see, using an ingress controller solves several tricky problems and provides an efficient, cost-effective way to route requests to your containers. Having HAProxy as the engine gives you access to many of the advanced features you know and love.

Kubernetes Basics

First, let's review the common approaches to routing external traffic to a pod in Kubernetes. Since there are many, thorough explanations available online that describe Kubernetes in general, we will skip the basics. In essence, Kubernetes consists of physical or virtual machines—called nodes—that together form a cluster. Within the cluster, Kubernetes deploys pods. Each pod wraps a container (or more than one container) and represents a service that runs in Kubernetes. Pods can be created and destroyed as needed.



To maintain the desired state of the cluster, such as which containers, and how many, should be deployed at a given time, we have additional objects in Kubernetes. These include ReplicaSets, StatefulSets, Deployments, DaemonSets, and more. For our current discussion, let's skip ahead and get to the meat of the topic: accessing pods via Services and Controllers.

Services

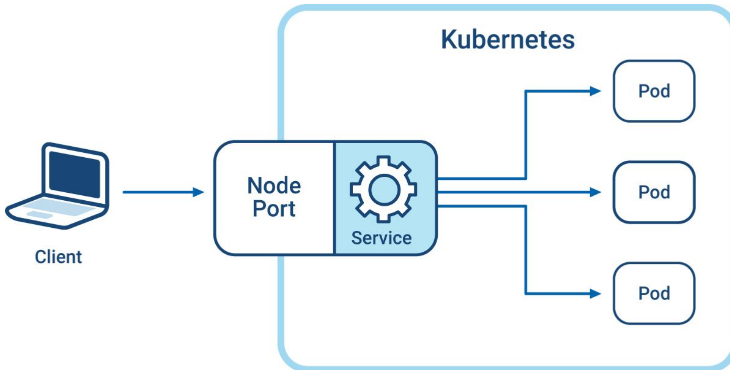
A service is an abstraction that allows you to connect to pods in a container network without needing to know a pod's location (i.e. which node is it running on?) or to be concerned about a pod's lifecycle. A service also allows you to direct external traffic to pods. Essentially, it's a primitive sort of reverse proxy. However, the mechanics that determine how traffic is routed depend on the service's type, of which there are four options:

- ClusterIP
- ExternalName
- NodePort
- LoadBalancer

When using Kubernetes services, each type has its pros and cons. We won't discuss ClusterIP because it doesn't allow for external traffic to reach the service—only traffic that originates within the cluster. ExternalName is used to route to services running outside of Kubernetes, so we won't cover it either. That leaves the NodePort and LoadBalancer types.

NodePort

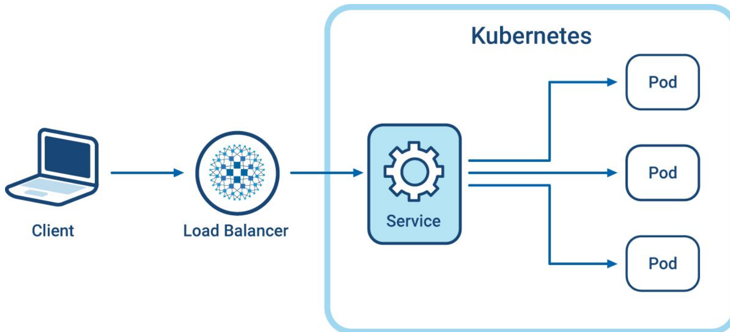
When you set a service's type to NodePort, that service begins listening on a static port on every node in the cluster. So, you'll be able to reach the service via any node's IP and the assigned port. Internally, Kubernetes does this by using L4 routing rules and Linux IPTables.



While this is the simplest solution, it can be inefficient and also doesn't provide the benefits of L7 routing. It also requires downstream clients to have awareness of your nodes' IP addresses, since they will need to connect to those addresses directly. In other words, they won't be able to connect to a single, proxied IP address.

LoadBalancer

When you set a service's type to LoadBalancer, it exposes the service externally. However, to use it, you need to have an external load balancer. The external load balancer needs to be connected to the internal Kubernetes network on one end and opened to public-facing traffic on the other in order to route incoming requests. Due to the dynamic nature of pod lifecycles, keeping an external load balancer configuration valid is a complex task, but this does allow L7 routing.



Oftentimes, when using Kubernetes with a platform-as-a-service, such as with AWS's EKS, Google's GKE, or Azure's AKS, the load balancer you get is automatic. It's the cloud provider's load balancer solution. If you create multiple Service objects, which is common, you'll be creating a hosted load balancer for each one. This can be expensive in terms of resources. You also lose the ability to choose your own preferred load balancer technology.

There needed to be a better way. The limited, and potentially costly, methods for exposing Kubernetes services to external traffic led to the invention of Ingress objects and ingress controllers.

Controllers

The official definition of a controller, not specific to ingress controllers, is:

a control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state.

For example, a Deployment is a type of controller used to manage a set of pods. It is responsible for replicating and scaling of applications. It watches the state of the cluster in a continuous loop. If you manually kill a pod, the Deployment object will take notice and immediately spin up a new one so that it keeps the configured number of pods active and stable.

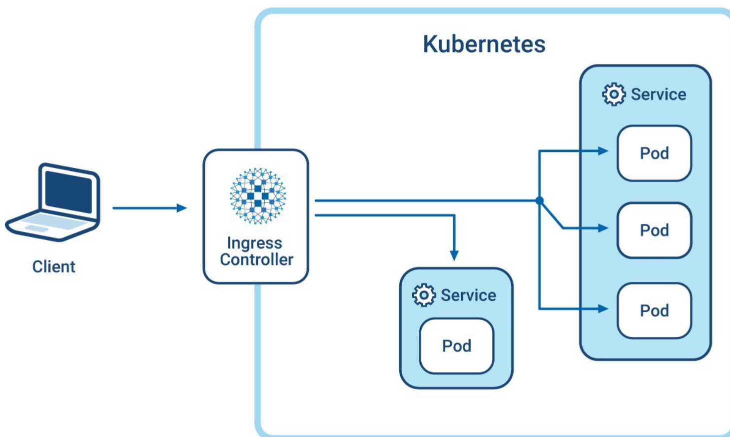
Other types of controllers manage functions related to persistent storage, service accounts, resource quotas, and cronjobs. So, in general, controllers are the watchers, ensuring that the system remains consistent. An ingress controller fits right in. It watches for new services within the cluster and is able to dynamically create routing rules for them.

Ingress

An Ingress object is an independent resource, apart from Service objects, that configures external access to a service's pods. This means you can define the Ingress later, after the Service has been deployed, to hook it up to

external traffic. That is convenient because you can isolate service definitions from the logic of how clients connect to them. This approach gives you the most flexibility.

L7 routing is one of the core features of Ingress, allowing incoming requests to be routed to the exact pods that can serve them based on HTTP characteristics such as the requested URL path. Other features include terminating TLS, using multiple domains, and, most importantly, load balancing traffic.



In order for Ingress objects to be usable, you must have an ingress controller deployed within your cluster that implements the Ingress rules as they are detected. An ingress controller, like other types of controllers, continuously watches for changes. Since pods in Kubernetes have arbitrary IPs and ports, it is the responsibility of an ingress controller to hide all internal networking from you, the operator. You only need to

define which route is designated to a service and the system will handle making the changes happen.

It’s important to note that ingress controllers still need a way to receive external traffic. This can be done by exposing the ingress controller as a Kubernetes service with either type NodePort or LoadBalancer. However, this time, when you add an external load balancer, it will be for the one service only and the external load balancer’s configuration can be more static.

Conclusion

In this chapter, you learned about Ingress objects and ingress controllers and how they solve the problem of routing external traffic into your cluster in a more flexible and cost-effective way than exposing services with NodePorts and LoadBalancers alone.

In the next chapter, you'll learn how to install the HAProxy Kubernetes Ingress Controller so that you can benefit from its blazing fast performance and mature set of features.

Install the HAProxy Kubernetes Ingress Controller with Helm

Helm is the Kubernetes package manager, resembling apt and yum, but born into the world of containers. It grew up alongside Kubernetes and was introduced early on, at the first KubeCon. Its job is to bundle up an application's Kubernetes resources into a package, called a chart, making it convenient to store, distribute, version, and upgrade those resources. That includes pods, services, config maps, roles, service accounts, and any other type available within the Kubernetes ecosystem.

Helm charts let you calibrate their behavior during install, such as to toggle from a Deployment to a Daemonset or to publish the ingress controller through an external load balancer, simply by setting a parameter during the install, which makes them perfect for delivering sophisticated services with lots of moving parts. You can use the HAProxy Kubernetes Ingress Controller Helm chart to install the ingress controller, streamlining the install process and making it easier to get started routing external traffic into your cluster. Our ingress controller is built around HAProxy, the fastest and most widely used load balancer. Having that foundation means that there are plenty of powerful features that you get right away, while benefiting from HAProxy's legendary performance.

It's easier to set up Helm than it used to be. You no longer need to install Tiller, the component that had been responsible for executing API commands and storing state within your cluster. Helm version 3 removed Tiller and has been rearchitected to use built-in Kubernetes constructs instead. That has made Helm simpler to use. It also makes it more secure due to its tighter integration with the Kubernetes role-based access controls.
In this chapter, you'll see how to install the HAProxy Kubernetes Ingress Controller using Helm, and how to customize its settings.

First, The Basics

Helm is now boringly simple to install. You need only to download the pre-built Helm binary and store it on your PATH. Unlike previous versions, there are no steps to install any server-side components like Tiller into your Kubernetes cluster prior to use. There are several good options to get a small Kubernetes cluster up and running, such as Minikube, MicroK8s and Kind.

Helm charts are stored in repositories. The main one is Helm Hub, which is hosted by the Helm project. However, you can add other, third-party repositories too. The HAProxy Kubernetes Ingress Controller is available by adding the HAProxy Technologies repository via the helm repo add command, like this:

```
$ helm repo add haproxytech \
  https://haproxytech.github.io/helm-charts

"haproxytech" has been added to your repositories
```

The next step is to refresh your list of charts by using the *helm repo update* command.

```
$ helm repo update

Hang tight while we grab the latest from your
chart repositories...
...Successfully got an update from the
"haproxytech" chart repository
...Successfully got an update from the "stable"
chart repository
Update Complete. ⎈ Happy Helming!⎈
```

Get an overview of available charts by invoking the *helm search repo* command:

```
$ helm search repo haproxy

NAME  CHART  VERSION  APP VERSION  DESCRIPTION
haproxytech/kubernetes-ingress    0.7.3
1.3.2          A
```

This shows the latest version of a chart, but you can also see older versions by including the versions argument. To install this chart, run *helm install*.

```
$ helm install haproxy \
  haproxytech/kubernetes-ingress

NAME: haproxy
LAST DEPLOYED: Tue Mar 10 14:57:41 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
HAProxy Kubernetes Ingress Controller has been
successfully installed.
```

The install command takes two parameters. The first, which I've set to haproxy, assigns a name to this release; The second identifies the chart that you want to install. Here's how the Helm documentation defines a release:

A Release is an instance of a chart running in a Kubernetes cluster. One chart can be installed many times into the same cluster. And each time it is installed, a new release is created.

The concept of a release is what makes Helm a vital addition to Kubernetes, since it lets you manage the delivery cycle of an application in a more controlled, less error-prone, way. Compare this to editing Kubernetes YAML files by hand and you'll no doubt appreciate the safety this offers. Having a repository of versioned releases gives you a way to handle upgrades and rollbacks with ease, since Helm can track which version is currently

deployed into your environment and can access older and newer versions instantly.

Use the *helm list* command to check which releases are deployed in your cluster:

```
$ helm list

NAME   NAMESPACE   REVISION   UPDATED
STATUS   CHART   APP VERSION
haproxy  default      1          2020-03-10
15:07:00.463855042 -0400 EDT    deployed
kubernetes-ingress-0.7.3    1.3.2
```

Once a new version of the chart has been published to the repository, you can get it by refreshing your list with *helm repo update* and then invoking *helm upgrade*:

```
$ helm repo update
$ helm upgrade haproxy \
  haproxytech/kubernetes-ingress
```

Uninstall the chart with the *helm uninstall* command:

```
$ helm uninstall haproxy

release "mycontroller" uninstalled
```

After the installation, you can execute *kubectl get service* to see that the ingress controller is running in your cluster:

```
$ kubectl get service

NAME  TYPE  CLUSTER-IP  EXTERNAL-IP  PORT(S)
AGE
haproxy-kubernetes-ingress  NodePort
10.101.232.155  <none>
80:32371/TCP,443:30110/TCP,1024:32052/TCP  21h
```

Notice that, by default, the internal service ports 80, 443, and 1024 are mapped to randomly assigned NodePorts. You can change this to use hardcoded NodePort numbers during the Helm install, as shown here:

```
$ helm install haproxy \
  haproxytech/kubernetes-ingress \
  --set controller.service.nodePorts.http=30000 \
  --set controller.service.nodePorts.https=30001 \
  --set controller.service.nodePorts.stat=30002
```

Or, you can install the controller as a DaemonSet instead of a Deployment by setting the *controller.kind* field. At the same time, set the *controller.daemonset.useHostPort* field to true to expose ports 80, 443 and 1024 directly on the host.

```
$ helm install haproxy \
  haproxytech/kubernetes-ingress \
  --set controller.kind=DaemonSet
  --set controller.daemonset.useHostPort=true
```

Or, use a cloud provider's load balancer in front of your ingress controller by setting the field *controller.service.type* to LoadBalancer:

```
$ helm install haproxy \
  haproxytech/kubernetes-ingress \
  --set controller.service.type=LoadBalancer
```

Forwarding Logs

You may also want to configure the controller to forward its traffic logs to standard out on the container, which can be done by setting the *syslog-server* field during the installation.

```
$ helm install haproxy \
  haproxytech/kubernetes-ingress \
  --set-string
"controller.config.syslog-server=address:stdout\,
format:raw\, facility:daemon"
```

You can also forward logs to a remote Syslog server. Note that you must escape commas that appear in the value by prefixing them with a backslash.

```
$ helm install mycontroller \
  haproxytech/kubernetes-ingress \
  --set-string
"controller.config.syslog-server=address:10.105.98
.88\, facility:local0\, level:info"
```

Any of the options listed in the controller’s documentation can be set in this way. When you have many keys to set, you can store them in a YAML file and then pass the name of the file to the helm install command. For example, suppose you created the following file and named it **overrides.yaml**:

```
controller:
  config:
    ssl-redirect: "true"
    syslog-server: "address:10.105.98.88,
facility:local0, level:info"
    defaultTLSSecret:
      enabled: true
      secret: default/mycert
```

You would reference this file by using the values flag, as shown:

```
$ helm install mycontroller \
  haproxytech/kubernetes-ingress \
  --values overrides.yaml
```

This approach allows you to save the file in version control and makes the helm install command more concise and the process more repeatable. Now that you've learned how to use Helm, you are assured an error-proof deployment of the HAProxy Kubernetes Ingress Controller!

Conclusion

In this chapter, we introduced the Helm chart for the HAProxy Kubernetes Ingress Controller, making it easier to begin routing traffic into your cluster using the powerful HAProxy load balancer. Helm facilitates deploying software by providing streamlined package management. You can use it to customize features like SSL termination and log forwarding.

Routing Traffic

Now that you have the ingress controller installed, it's time to spin up a sample application and add an Ingress resource that will route traffic to your pod. You will create the following:

- A Deployment to launch your pods
- A Service to group your pods
- An Ingress that defines routing to your pods

Before setting up an application, you can see that the default service returns *404 Not Found* responses for all routes. If you are using Minikube, you can get the IP of the cluster with the *minikube ip* command.

Each of the ingress controller's ports is mapped to a NodePort port via a Service object. After installing the ingress controller, you can use *kubectl get svc* to see the ports that were mapped:

```
$ kubectl get svc

NAME   TYPE   CLUSTER-IP   EXTERNAL-IP   PORT(S)
AGE
haproxy-kubernetes-ingress
NodePort      10.104.180.6    <none>
80:30267/TCP,443:31566/TCP,1024:30256/TCP   4s
```

In this instance, the following ports were mapped:

- Container port 80 to NodePort 30279
- Container port 443 to NodePort 30775
- Container port 1024 to NodePort 31912

Use *curl* to send a request with a Host header of foo.bar and get back a 404 response:

```
$ curl -I -H 'Host: foo.bar' \
  'http://192.168.99.100:30279'

HTTP/1.1 404 Not Found
date: Thu, 27 Jun 2019 21:45:20 GMT
content-length: 21
content-type: text/plain; charset=utf-8
```

Launch the Application

Let's launch a sample application. Prepare the Deployment object YAML file by creating a file named **deployment.yaml** and add the following markup. This will deploy two replicas of the sample application:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: app
  name: app
spec:
  replicas: 2
  selector:
    matchLabels:
      run: app
  template:
    metadata:
      labels:
        run: app
    spec:
      containers:
      - name: app
        image: jmalloc/echo-server
        ports:
        - containerPort: 8080
```

Next, define a Service object YAML file named **service.yaml** to group the pods. Notice that you can define annotations in the Service that enable health checks, forward the client's IP address to the pod, and choose the load balancing algorithm.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    run: app
  name: app
  annotations:
    haproxy.org/check: "enabled"
    haproxy.org/forwarded-for: "enabled"
    haproxy.org/load-balance: "roundrobin"
spec:
  selector:
    run: app
  ports:
  - name: port-1
    port: 80
    protocol: TCP
    targetPort: 8080
```

Add an Ingress

Next, define an Ingress object YAML file named **ingress.yaml**. This file controls how external traffic will be routed to your pods. It finds the pods by the Service that we used to group them.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: web-ingress
  namespace: default
spec:
  rules:
  - host: foo.bar
    http:
      paths:
      - path: /
        backend:
          serviceName: app
          servicePort: 80
```

Apply the configuration with the *kubectl apply* command:

```
$ kubectl apply -f deployment.yaml
$ kubectl apply -f service.yaml
$ kubectl apply -f ingress.yaml
```

The ingress controller will automatically detect the Ingress and add a new route for it to HAProxy's underlying configuration. This will allow all traffic for **http://foo.bar/** to go to our application. Next, define a ConfigMap to tune other settings. Use *curl* again and you'll get a successful response:

```
$ curl -I -H 'Host: foo.bar' \
  'http://192.168.99.100:30279'

HTTP/1.1 200 OK
content-type: text/plain
date: Thu, 27 Jun 2019 21:46:30 GMT
content-length: 136
```

Conclusion

In this chapter, you learned how to deploy an application to Kubernetes and publish it using an Ingress object. By allowing routing rules to be defined separately from Deployment and Service objects, you are able to publish applications later and with a greater degree of control. You can even delete the Ingress to un-publish the application without affecting the pods themselves.

TLS with Let's Encrypt

When it comes to TLS in Kubernetes, the first thing to appreciate when you use the HAProxy Ingress Controller is that all traffic for all services travelling to your Kubernetes cluster passes through HAProxy. Requests are then routed towards the appropriate backend services depending on metadata in the request, such as the Host header. So, by enabling TLS in your ingress controller, you're adding secure communication to all of your services at once. HAProxy is known for its advanced support of the important performance-oriented features available in TLS.

In this chapter, you'll learn how to configure TLS in the ingress controller using a self-signed certificate. Then, you'll see how to get a certificate automatically from Let's Encrypt, which can be used in Production. Using Let's Encrypt requires version 1.4.6 or later of the HAProxy Kubernetes Ingress Controller.

A Default TLS Certificate

When you install the ingress controller with Helm, it creates a self-signed TLS certificate, which is useful for non-production environments. Run *kubectl get secret* to see that it exists:

```
$ kubectl get secret

NAME   TYPE   DATA   AGE
haproxy-kubernetes-ingress-default-cert
kubernetes.io/tls  2        2m22s
```

View the certificate's details by running the same command with the name of the secret and the output parameter set to yaml:

```
$ kubectl get secret \
  haproxy-kubernetes-ingress-default-cert -o yaml

apiVersion: v1
data:
  tls.crt: ABCDEFG123456...
  tls.key: ABCDEFG123456...
```

Straight away, you can access your services externally over HTTPS using this certificate. However, you'll want to replace it with your own, trusted one for production environments, which you can do by creating a new Secret object in Kubernetes that contains your certificate and then updating the ingress controller to use it.

To see how it works, let's create a self-signed certificate of our own. Here's how to create a self-signed certificate using OpenSSL for a website named test.local:

```
$ openssl req -x509 \
  -newkey rsa:2048 \
  -keyout test.local.key \
  -out test.local.crt \
  -days 365 \
  -nodes \
  -subj "/C=US/ST=Ohio/L=Columbus/O=MyCompany/
CN=test.local"
```

Use the *kubectl create secret* command to save your TLS certificate and key as a Secret in the cluster. The *key* and *cert* fields reference local files where you've saved your certificate and private key.

```
$ kubectl create secret tls test-cert \
  --key="test.local.key" \
  --cert="test.local.crt"
```

When you installed the HAProxy Ingress Controller, it also generated an empty ConfigMap object named haproxy-kubernetes-ingress, where haproxy is the name you gave when installing the Helm chart. Update this ConfigMap with a field named ssl-certificate that points to the Secret object you just created.

Did You Know? The HAProxy Ingress Controller depends on having a ConfigMap defined. You can add and delete fields from it, but you should not delete it from the cluster.

Here is an example ConfigMap object that sets the *ssl-certificate* field to the Secret named my-cert. Use the *kubectl apply -f [FILE]* command to update the ConfigMap in your cluster.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: haproxy-kubernetes-ingress
  namespace: default
data:
  ssl-certificate: "default/test-cert"
```

Now, when you access your services over HTTPS, they'll use this TLS certificate.

Choose a Different Certificate Per Ingress

The benefit of an ingress controller is that it proxies traffic for all of the services you'd like to publish externally. The certificate you added to the ConfigMap applies across the board, but you can override it with a different certificate for each service. In that case, HAProxy uses SNI to find the right certificate.

Create a new certificate to use for a particular domain, such as api.test.local. Create a new certificate using OpenSSL:

```
$ openssl req -x509 \
  -newkey rsa:2048 \
  -keyout api.test.local.key \
  -out api.test.local.crt \
  -days 365 \
  -nodes \
  -subj "/C=US/ST=Ohio/L=Columbus/O=MyCompany/
CN=api.test.com"
```

Next, add the certificate and key files to your cluster by creating a Secret object:

```
$ kubectl create secret tls api-test-cert \
  --key="api.test.local.key" \
  --cert="api.test.local.crt"
```

Then, define an Ingress object where the *rules* stanza applies to any request for api.test.local. Any requests for that hostname will be routed to the backend service named api-service. We're also defining a *tls* stanza that configures which TLS certificate to use for this service. Its *secretName* field points to our new Secret object.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
 name: api-ingress
 namespace: default
spec:
 rules:
 - host: api.test.local
   http:
     paths:
     - path: /
       backend:
         serviceName: api-service
         servicePort: 80
 tls:
 - secretName: api-test-cert
   hosts:
   - api.test.local
```

Apply this with the *kubectl apply -f [FILE]* command and you'll see that requests for api.test.local use this certificate rather than the one you set in the ConfigMap. Note that you can update your **/etc/hosts** file to resolve **test.local** and api.test.local to your ingress controller's IP address. Technically, HAProxy chooses the correct certificate by using SNI, which means that once the certificate is added by one Ingress, HAProxy will use it for other routes too if they match that hostname.

Let's Encrypt Certificates

Now that you've seen how to define which TLS certificate to use for a particular service, you can take this a step further by having the Secret populated automatically with a certificate from Let's Encrypt. There's an open-source tool called cert-manager that you'll install into your cluster to handle communicating with the Let's Encrypt servers.

First, be sure to deploy your cluster with a public IP address, such as by using a managed Kubernetes service like Amazon EKS and then deploying the ingress controller with a service type of LoadBalancer, which will create a cloud load balancer in front of the cluster that has a public IP. Then, create a DNS record that resolves your domain name to that IP address. You can use a service like NS1 to set up a DNS record, once you've purchased a domain from a domain registrar. Let's Encrypt will need access to your service at its domain name address to send the ACME challenges. In particular, Let's Encrypt expects your website to be listening on port 80 and will issue certificates that match your domain name.

Next, deploy cert-manager into your cluster:

```
$ kubectl apply --validate=false -f \
https://github.com/jetstack/cert-manager/releases/
download/v0.15.1/cert-manager.yaml
```

Then, deploy a cert-manager issuer, which is responsible for getting certificates from Let's Encrypt and validating your domain by answering [ACME HTTP-01 challenges](). Here's an example YAML file to create a ClusterIssuer that's taken, in part, from the cert-manager documentation:

```
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: letsencrypt-staging
spec:
  acme:
    email: myemail@company.com
    server:
https://acme-staging-v02.api.letsencrypt.org/direc
tory
    privateKeySecretRef:
      # Secret used to store the account's private
key.
      name: example-issuer-account-key
    # Add a ACME HTTP01 challenge solver
    solvers:
    - http01:
        ingress: {}
```

In this example, you are creating a ClusterIssuer that can set up certificates for ingress controllers regardless of the namespace in which they run. It is configured to use the Let's Encrypt staging server, which is the best place to work out your implementation without contacting the Let's Encrypt production servers. Later, you can create a

different ClusterIssuer that has its *server* field set to the real Let's Encrypt server, **https://acme-v02.api.letsencrypt.org/directory**.

Next, add an Ingress object that includes the cert-manager annotation, which points to your ClusterIssuer. The cert-manager program will communicate with Let's Encrypt and store the certificate it receives in the Secret referred to by the *secretName* field in the *tls* stanza.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    # add an annotation indicating the issuer to
use
    cert-manager.io/cluster-issuer:
letsencrypt-staging
  name: mysite-ingress
  namespace: default
spec:
  rules:
  - host: mysite.com
    http:
      paths:
      - path: /
        backend:
          serviceName: mysite-service
          servicePort: 80
  tls:
  - secretName: mysite-cert
    hosts:
    - mysite.com
```

You can specify more than one host in the *rules* and *tls* sections to handle different domain names, such as mysite.com and www.mysite.com. A temporary cert-manager pod and ingress resource will be created for you to handle the HTTP-01 challenge, but are removed afterwards. You can inspect this pod's logs in case of any trouble:

```
$ kubectl logs -f <cert-manager-pod> -n
cert-manager
```

Once set up, you won't have to worry about manually installing certificates again!

Conclusion

In this chapter, you learned how to configure TLS with the HAProxy Ingress Controller, making it easy to provide secure communication for all of the clients accessing your Kubernetes services. To take it a step further, you can use cert-manager to configure Let's Encrypt certificates automatically.

Multi-tenant Kubernetes Clusters

It's a rare bird, a Kubernetes cluster that serves only a single tenant. In the wild, you'll likely encounter clusters where tenants are packed in close: QA and Dev, Team A and Team B, Java application and .NET application—environments, teams, and technology stacks declare their stakes on resources. It's essential that you plan ahead for multiple tenants, set up the proper namespaces, define access controls, set resource quotas, and configure ingress routing.

Sharing resources in a Kubernetes cluster is a logical way to save money on the cost of infrastructure. In this chapter, we'll share tips for setting up multiple tenants and, in particular, how to configure the HAProxy Kubernetes Ingress Controller to serve traffic to multiple tenants.

Namespaces are Key

A Kubernetes namespace groups objects inside of a shared scope, providing a sandbox where objects created by one tenant don't overlap with objects created by another. Take for example a Dev and a QA environment. You can host both environments inside of a single Kubernetes cluster where they can share server resources, yet remain oblivious to one another. Each environment, or "tenant",

can duplicate your entire application stack. Building walls around each tenant avoids accidentally exposing an experimental Dev service within the QA environment, or deleting the wrong object, or applying a breaking change to the wrong application

Declare a new namespace by adding a YAML file that defines a Namespace object, like this:

```
apiVersion: v1
kind: Namespace
metadata:
  name: dev
```

In this instance, the namespace is named dev. Use *kubectl* to apply this change to your cluster:

```
$ kubectl apply -f dev-namespace.yaml
```

Once created, add objects to the namespace by referencing its name within the object's metadata. In the following example, a ConfigMap object is added to the dev namespace:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap
  namespace: dev
data:
  foo: 'bar'
```

Only objects within the same dev namespace will have access to this ConfigMap. Also, when using the *kubectl* command-line utility to view objects, you will need to include the --namespace argument or else the returned list will come up empty:

```
$ kubectl get configmaps --namespace=dev
```

Use the *kubectl get namespaces* command to view all of your defined namespaces:

```
$ kubectl get namespaces
NAME      STATUS   AGE
default   Active   4m33s
dev       Active   2m38s
```

Managing User Access to a Namespace

Once you've defined a namespace, you can configure role-based access control (RBAC) to limit who has access to it. Out of the box, there are already a few roles defined, including admin, edit, and view. In the following sections, a new user login is created and given the edit role in the dev namespace, which gives it read/write access to that namespace only.

Add a RoleBinding

To begin granting access to a user, first create a new RoleBinding object that assigns the edit role to a user named bob.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: dev-edit
  namespace: dev
subjects:
- kind: User
  name: bob # permissions for a user named bob
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: edit # read/write access
  apiGroup: rbac.authorization.k8s.io
```

The edit ClusterRole is already defined and can be scoped to the dev namespace by setting the namespace metadata field. Who is Bob? It's a user who isn't represented as an object per se (there is no User object in Kubernetes), but who will authenticate to the cluster using a client certificate that contains a CN field set to bob. You can also grant permissions to a group of users. In the following RoleBinding object, a group named dev-group is granted edit access to the dev namespace:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: dev-edit
  namespace: dev
subjects:
- kind: Group
  name: dev-group # permissions for the group
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: edit # read/write access
  apiGroup: rbac.authorization.k8s.io
```

For group permissions, when you create the client certificate, its O field must match the subject name, which is dev-group in this case. Use *kubectl* to create the object in Kubernetes:

```
$ kubectl apply -f dev-rolebinding.yaml
```

Create a Client Certificate

The next step is to create a certificate signing request (CSR) for a new client certificate. There are a number of tools that you can use to do this, such as the openssl command-line utility. In the following example, I use openssl to create a CSR for a user named bob with a group of dev-group to demonstrate setting the CN and O fields:

```
# Create a CSR with CN=bob and O=dev-group
# This creates bob.csr and bob.key
$ openssl req -newkey rsa:2048 -nodes \
  -keyout bob.key -out bob.csr \
  -subj "/CN=bob/O=dev-group"
```

Next, you'll need to sign the CSR with your cluster's CA certificate in order to get a client certificate. I'm using Minikube in my test lab, so I could sign the certificate signing request with Minikube's CA certificate and key, which can be found in the .minikube directory. I would use the following OpenSSL command to create a client certificate named bob.crt.

```
# Sign it with the cluster's CA certificate
# This creates bob.crt
$ openssl x509 -req -in bob.csr \
  -CA ~/.minikube/ca.crt \
  -CAkey ~/.minikube/ca.key -CAcreateserial \
  -out bob.crt -days 1000
```

Another way to sign the CSR and get a bob.crt file is to use the Kubernetes Certificates API, wherein you create a CertificateSigningRequest object. You will need to store the CSR data in a YAML file as a base64-encoded string and then apply the YAML file to your cluster, so it's easiest to do it from the command line, like this:

```
$ cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: bob
spec:
  request: $(cat bob.csr | base64 | tr -d '\n')
  usages:
  - digital signature
EOF
```

Then, approve the CSR:

```
$ kubectl certificate approve bob
```

You can then download the signed certificate with the *kubectl get csr* command:

```
$ kubectl get csr bob -o \
  jsonpath='{.status.certificate}' |
  base64 --decode > bob.crt
```

Add a Cluster Context

Next, add a new cluster context that lets you log in as bob, using the bob certificate.

```
$ kubectl config set-credentials bob \
  --client-certificate=bob.crt \
  --client-key=bob.key

$ kubectl config set-context minikube-bob \
  --cluster=minikube --user=bob

$ kubectl config  use-context minikube-bob
```

You’re now using the minikube-bob context to access your Minikube Kubernetes cluster. If you try accessing or creating objects in the dev namespace, it will work, but you’ll get an error if you try to access an object in any other namespace.

```
$ kubectl get pods --namespace=dev

NAME   READY   STATUS   RESTARTS   AGE
app-66d9457bf5-vpbnw                      1/1
Running   1          22h

$ kubectl get pods --namespace=default

Error from server (Forbidden): pods is forbidden:
User "bob" cannot list resource "pods" in API
group "" in the namespace "default"
```

You can switch back to the normal Minikube context, which has admin privileges, like this:

```
$ kubectl config  use-context minikube
```

An Ingress Controller that Watches a Namespace

Now that you've created a namespace and given limited access to it, let's see how to manage HTTP traffic going into the environment by levering the HAProxy Kubernetes Ingress Controller.

Be sure to switch back to the normal admin context before going further. Without any special configuration, the HAProxy Kubernetes Ingress Controller will watch over all namespaces. When a pod is added or removed anywhere within the cluster, the controller is notified, which means that any of your teams can use it for ingress traffic routing. That's great news if you want to set up routing quickly for all of your teams (i.e. tenants). However, there are a few reasons why you may decide to deploy multiple ingress controllers.

For one thing, by creating multiple ingress controllers, you can apply a walled garden approach for each tenant. By creating a distinct ingress controller for each one, you can:

1. collect distinct HAProxy metrics per tenant, such as request rates and error rates.
2. set rate limits per tenant to prevent "noisy neighbors" syndrome.

3. define custom timeouts per tenant to accommodate varying SLAs.
4. reuse the same URL paths to keep your applications consistent between tenants.

When you deploy an HAProxy Kubernetes Ingress Controller using Helm, add *--namespace-whitelist* to the *controller.extraArgs* field to set the namespace to watch, as shown:

```
$ helm install onlydev \
  haproxytech/kubernetes-ingress \
  --set-string
"controller.extraArgs={--namespace-whitelist=dev}"
```

You can specify more than one namespace to watch:

```
$ helm install onlydev \
  haproxytech/kubernetes-ingress \
  --set-string
"controller.extraArgs={--namespace-whitelist=dev-t
eam-a,--namespace-whitelist=dev-team-b}"
```

This Ingress object is created within the dev namespace and, therefore, is picked up automatically by an ingress controller that has its whitelist set to *dev*:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: app-ingress
  namespace: dev
spec:
  rules:
  - http:
      paths:
      - path: /app-service
        backend:
          serviceName: app-service
          servicePort: 80
```

You could create an identical Ingress object in your *qa* namespace, but it won't route through this particular ingress controller because its namespace is different. Each ingress controller can be exposed on a unique IP address so that tenants can be given their own subdomain. It won't be possible for one tenant's traffic to mix with that of another.

Note that it would still be possible for services running in one namespace to call services running in another. Although we won't cover it here, you can use Network Policy objects to restrict access between services inside the cluster.

An Ingress Controller that You Target

Another way to manage ingress routing is to use ingress classes. Whereas *--namespace-whitelist* tells the ingress controller to watch a specific namespace for changes, an ingress class flips that responsibility around, giving an Ingress object a chance to target the controller it wants by name. To set this up, add class to the list of arguments when defining your ingress controller. Here, the *--ingress.class* argument is set to intranet:

```
$ helm install intranet \
  haproxytech/kubernetes-ingress \
  --set controller.ingressClass=intranet
```

Maybe this ingress controller exposes services only to the company's intranet. You may have another ingress controller that has a class of public, exposing services to external customers, for example. An Ingress object targets its desired controller by setting its *haproxy.org/ingress.class* annotation, as shown:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: app-ingress-internal
  namespace: default
  annotations:
    haproxy.org/ingress.class: "intranet"
spec:
  rules:
  - http:
      paths:
      - path: /app-service
        backend:
          serviceName: app-service-internal
          servicePort: 80
```

This puts the control into the hands of your service developers. They can choose which ingress controller to use and it gives them a greater degree of autonomy. You can even use this with multiple tenants, if you don't mind giving them a common IP address for accessing their services.

Resource Quotas

As a final tip, each namespace can be assigned its own allotment of resources. For example, QA might be allocated more or less CPU and memory than Dev. This lets you prioritize which tenants receive the resources, or lets you simply keep things equal for everybody. If you don't do this, then you risk one tenant utilizing more than their fair

share and leaving other tenants squabbling over the scraps.

It is essential then that every pod defines how much CPU and memory it needs so that Kubernetes knows when a tenant is about to exceed its resource limits. We won't go into detail about this, but this is accomplished by setting requests and limits on a pod, which you can learn more about on theManaging Compute Resources page. You can also create defaults for a namespace, in case a pod doesn't set its own limits, by creating a LimitRange object.

Let's cover how to set the resource quota for a namespace, which determines the cap on resources. If a tenant requests more resources than what you've allowed here, their objects won't be created. Resource quotas let you restrict:

- the total CPU that can be used
- the total memory that can be used
- the amount of harddrive storage that can be used
- the number of objects that can be created

The following ResourceQuota object sets limits for CPU and memory in the dev namespace:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev-resources
  namespace: dev
spec:
  hard:
    requests.cpu: "3"
    requests.memory: 40Gi
    limits.cpu: "4"
    limits.memory: 50Gi
```

Use *kubectl* to apply the quota:

```
$ kubectl apply -f dev-quota.yaml
```

Then, you can view how much has been used so far:

```
$ kubectl describe resourcequota dev-resources \
  -n dev

Name:            dev-resources
Namespace:       dev
Resource         Used  Hard
--------         ----  ----
limits.cpu       0     4
limits.memory    0     50Gi
requests.cpu     500m  3
requests.memory  50Mi  40Gi
```

Quotas will help keep tenants from overusing resources and allows you to see how much a particular tenant has used so far, which is great when planning whether you need to expand the cluster. It's a vital step when planning for multiple tenants.

Conclusion

In this chapter, you learned some tips for managing multiple tenants that share resources within a Kubernetes cluster. The HAProxy Kubernetes Ingress Controllers lets you whitelist certain namespaces to watch so that each namespace can be routed through a specific controller. You can also target specific ingress controllers by using ingress classes.

When setting up multiple tenants, it pays to configure RBAC and to give teams access to only their respective namespaces, which you can accomplish by using client certificates. You should also consider setting resource quotas to prevent a tenant from using more than their fair share of CPU and memory.

Kubernetes Deployment Patterns

In this chapter, we'll cover how to plan for upgrades to your applications that run in your Kubernetes cluster. It can be important to plan for this upfront so that when the time to upgrade comes, you'll be well prepared. Kubernetes accommodates a wide range of deployment methods. We'll cover two that guarantee a safe rollout while keeping the ability to revert if necessary:

- Rolling updates have first-class support in Kubernetes and allow you to phase in a new version gradually;
- Blue-green deployments avoid having two versions at play at the same time by swapping one set of pods for another.

The HAProxy Kubernetes Ingress Controller is powered by the world's fastest and most widely used software load balancer. Known to provide the utmost performance, observability, and security, it is the most efficient way to route traffic into a Kubernetes cluster. It automatically detects changes within your Kubernetes infrastructure and ensures accurate distribution of traffic to healthy pods. Its design prevents downtime even when there are rapid configuration changes. It supports both deployment patterns and reliably exposes the correct pods to clients.

Rolling Updates

A rolling update offers a way to deploy the new version of your application gradually across your cluster. It replaces pods during several phases. For example, you may replace 25% of the pods during the first phase, then another 25% during the next, and so on until all are upgraded. Since the pods are not replaced all at once, this means that both versions will be live, at least for a short time, during the rollout.

Did You Know? Because a rolling update creates the potential for two versions of your application to be deployed simultaneously, make sure that any upstream databases and services are compatible with both versions.

This deployment model enjoys first-class support in Kubernetes with baked-in YAML configuration options. Here's how it works:

1. Version 1 of your application is already deployed.
2. Push version 2 of your application to your container image repository.
3. Update the version number in the Deployment object's definition.
4. Apply the change with *kubectl*.
5. Kubernetes staggers the rollout of the new version across your pods.
6. The HAProxy Kubernetes Ingress Controller detects when the new pods are live. It automatically

updates its proxy configuration, routing traffic away from the old pods and towards the new ones.

A rolling update dodges downtime by replacing existing pods incrementally. If the new pods introduce an error that stops them from starting up, Kubernetes will pause the rollout. Also, a rolling update ensures that some pods are always up, so there's no downtime. Kubernetes keeps a minimum number of pods running during the rollout. However, this requires that you've added a readiness check to your pods so that Kubernetes knows when they are truly ready to receive traffic.

Deploy the Original Application

Kubernetes enables rolling updates by default. An update begins when you change your Deployment resource's YAML file and then use *kubectl apply*. Consider the following definition, which deploys version 1 of an application. Note that it uses the errm/versions Docker image because it displays the version of the application when you browse to its webpage, which makes it easy to see which version you're running.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: app
  name: app
spec:
  replicas: 5
  selector:
    matchLabels:
      run: app
  template:
    metadata:
      labels:
        run: app
    spec:
      containers:
      - name: app
        image: errm/versions:0.0.1
        ports:
        - containerPort: 3000
        readinessProbe:
          httpGet:
            path: /
            port: 3000
          initialDelaySeconds: 5
          periodSeconds: 5
          successThreshold: 1
```

The *readinessProbe* section tells Kubernetes to send an HTTP request to the application five seconds after it has started, and then every five seconds thereafter. No traffic

is sent to the pod until a successful response is returned. This is key to preventing downtime.

Did You Know? Consider tagging your container images with version numbers, rather than using a tag like latest. This allows you to keep track of the versions that are deployed and manage the release of new versions.

Next, define a Service object that will categorize the pods into a single group that the ingress controller will watch:

```
apiVersion: v1
kind: Service
metadata:
  name: app-service
spec:
  selector:
    run: app
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 3000
```

Next, define an Ingress object. This configures how the HAProxy Ingress Controller will route traffic to the pods:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: app-ingress
  namespace: default
spec:
  rules:
  - http:
      paths:
      - path: /
        backend:
          serviceName: app-service
          servicePort: 80
```

Use *kubectl apply* to deploy the pods, service and ingress:

```
$ kubectl apply -f app.yaml \
  -f app-service.yaml -f ingress.yaml
```

Version 1 of your application is now deployed. Run the following command to see which port the HAProxy Kubernetes Ingress Controller has mapped to port 80:

```
$ kubectl get svc haproxy-ingress \
  -n haproxy-controller

NAME              TYPE       CLUSTER-IP
EXTERNAL-IP   PORT(S)
AGE
haproxy-ingress   NodePort   10.101.75.28   <none>
80:31179/TCP,443:31923/TCP,1024:30430/TCP   98s
```

You can then see that the application is exposed on port 31179. You can see it by visiting the Minikube IP address **http://192.168.99.100:31179** in your browser.

Let's see how to upgrade it to version 2 next.

Upgrade Using a Rolling Update

After you have pushed a new version of your application to your container repository, trigger a rolling update by increasing the version number set on the Deployment definition's *spec.template.spec.containers.image* property. This tells Kubernetes that the current, desired version of your application has changed. In our example, since we're using a prebaked image, there's already a version 2 set up in the Docker Hub repository.

```
image: errm/versions:0.0.2
```

Then, use *kubectl apply* to start the rollout:

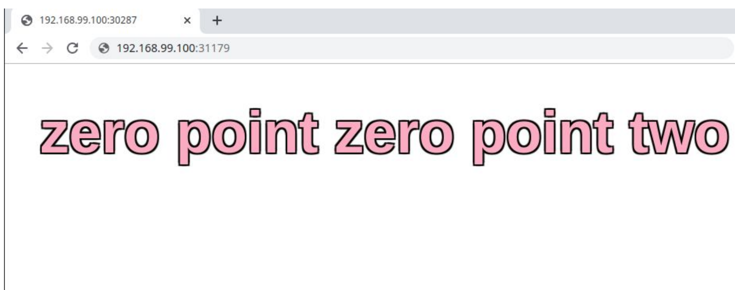
```
$ kubectl apply -f app.yaml
```

You can check the status of the rollout by using the *kubectl rollout status* command:

```
$ kubectl rollout status deployment app

deployment "app" successfully rolled out
```

Once completed, you can access the application again at the same URL, **http://192.168.99.100:31179**. It shows you a new web page signifying that version 2 has been deployed.



If you decide that the new version is faulty, you can revert to the previous one by using the *kubectl rollout undo* command, like this:

```
$ kubectl rollout undo deployment app

deployment.extensions/app rolled back
```

The HAProxy Kubernetes Ingress Controller detects pod changes quickly and can switch back and forth between versions without dropping connections. Rolling updates aren't the only way to accomplish highly-available services, though. In the next section, you'll learn about blue-green deployments, which update all pods simultaneously.

Blue-Green Deployments

A blue-green deployment lets you replace an existing version of your application across all pods at once. The name, blue-green, was coined in the book Continuous Delivery by Jez Humble and David Farley. Here's how it works:

1. Version 1 of your application is already deployed.
2. Push version 2 of your application to your container image repository.
3. Deploy version 2 of your application to a new group of pods. Both versions 1 and 2 pods are now running in parallel. However, only version 1 is exposed to external clients.

4. Run internal testing on version 2 and make sure it is ready to go live.
5. Flip a switch and the ingress controller in front of your clusters stops routing traffic to the version 1 pods and starts routing it to the version 2 pods.

This deployment pattern has a few advantages over a rolling update. For one, at no time are there ever two versions of your application accessible to external clients at the same time. So, all users will receive the same client-side Javascript files and be routed to a version of the application that supports the API calls within those files. It also simplifies upstream dependencies, such as database schemas.

Another advantage is that it gives you time to test the new version in a production environment before it goes live. You control how long to wait before making the switch. Meanwhile, you can verify that the application and its dependencies function normally.

On the other hand, a blue-green deployment is all-or-nothing. Unlike a rolling update, you aren't able to gradually roll out the new version. All users will receive the update at the same time, although existing sessions will be allowed to finish their work on the old instances. So, the stakes are a bit higher that everything should work, once you do initiate the change. It also requires allocating more server resources, since you will need to run two copies of every pod.

Luckily, the rollback procedure is just as easy: You simply flip the switch again and the previous version is swapped

back into place. That's because the old version is still running on the old pods. It is simply that traffic is no longer being routed to them. When you're confident that the new version is here to stay, you can decommission those pods. You'll need to set up your original application in a slightly different way when you expect to use a blue-green deployment. There is more emphasis on using Kubernetes metadata labels, which will become clear in the next section.

Deploy the Original Application

Consider the following definition, which deploys version 1 of your application. Note its *spec.selector* section, which specifies a label called version:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: app
  name: app-blue
spec:
  replicas: 1
  selector:
    matchLabels:
      run: app
      version: 0.0.1
  template:
    metadata:
      labels:
        run: app
        version: 0.0.1
    spec:
      containers:
      - name: app
        image: errm/versions:0.0.1
        ports:
        - containerPort: 3000
```

A Deployment object defines a spec.selector section that matches the *spec.template.metadata* section. This is how a Deployment tags pods and keeps track of them. This is the key to setting up a blue-green deployment. By using different labels, you can deploy multiple versions of the same application. Here, the *spec.selector.matchLabels* property is set to run=app,version=0.0.1. The version

should match the version tag of your Docker image, for convenience and simplicity.

The following Service definition targets that same selector:

```
apiVersion: v1
kind: Service
metadata:
  name: app-service
spec:
  selector:
    run: app
    version: 0.0.1
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 3000
```

Next, use the following Ingress definition to expose the version 1 pods to the world. It registers a route with the HAProxy Kubernetes Ingress Controller:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: app-ingress
  namespace: default
spec:
  rules:
  - http:
      paths:
      - path: /
        backend:
          serviceName: app-service
          servicePort: 80
```

Apply everything using *kubectl*:

```
$ kubectl apply -f app-v1.yaml
$ kubectl apply -f app-service-bg.yaml
$ kubectl apply -f ingress.yaml
```

At this point, you can access the application at the HTTP port exposed by the ingress controller: **http://192.168.99.100:31179**. Now, let’s see how to use a blue-green deployment to upgrade the version.

Upgrade Using a Blue-green Deployment

Now that the blue version (i.e. version 1) is released, create a green version of your Deployment object that will deploy version 2. The YAML will be the same, except that you

increase the value of the version label, as well as the Docker image tag. Also note that the name of the deployment is changed from app-blue to app-green, since you cannot have two Deployments with the same name that target different pods.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: app
  name: app-green
spec:
  replicas: 1
  selector:
    matchLabels:
      run: app
      version: 0.0.2
  template:
    metadata:
      labels:
        run: app
        version: 0.0.2
    spec:
      containers:
      - name: app
        image: errm/versions:0.0.2
        ports:
        - containerPort: 3000
```

Apply it with *kubectl*:

```
$ kubectl apply -f app-v2.yaml
```

At this point, both blue (version 1) and green (version 2) are deployed. Only the blue instance is receiving traffic, though. To make the switch, update your Service definition's version selector so that it points to the new version:

```
apiVersion: v1
kind: Service
metadata:
  name: app-service
spec:
  selector:
    run: app
    version: 0.0.2
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 3000
```

Apply it with *kubectl*:

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

Check the application again and you will see that the new version is live. If you need to roll back to the earlier version, simply change the Service definition's selector back and reapply it. The HAProxy Kubernetes Ingress

Controller detects these changes almost instantly and you can swap back and forth to your heart's content. There's no downtime during the cutover. Established TCP connections will finish normally on the instance where they began.

Testing the New Pods

You can also test the new version before it's released by registering a different ingress route that exposes the application at a new URL path. First, create another Service definition called test-service:

```
apiVersion: v1
kind: Service
metadata:
  name: test-service
  annotations:
    haproxy.org/path-rewrite: /
spec:
  selector:
    run: app
    version: 0.0.2
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 3000
```

Note that we are including the path-rewrite annotation, which rewrites the URL /test to / before it reaches the pod.

Then, add a new route to your existing Ingress object that exposes this service at the URL path /test, as shown:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: app-ingress
  namespace: default
  annotations:
    haproxy.org/ingress.class: "development"
spec:
  rules:
  - http:
      paths:
      - path: /
        backend:
          serviceName: app-service
          servicePort: 80
      - path: /test
        backend:
          serviceName: test-service
          servicePort: 80
```

This lets you check your application by visiting /test in your browser.

Conclusion

The HAProxy Kubernetes Ingress Controller is powered by the legendary HAProxy. Known to provide the utmost performance, observability, and security, it features many

benefits including SSL termination, rate limiting, and IP whitelisting. When you deploy the ingress controller into your cluster, it's important to consider how your applications will be upgraded later.

Two popular methods are rolling updates and blue-green deployments. Rolling updates allow you to phase in a new version gradually and it has first-class support in Kubernetes. Blue-green deployments avoid the complexity of having two versions at play at the same time and give you a chance to test the change before going live. In either case, the HAProxy Kubernetes Ingress Controller detects these changes quickly and maintains uptime throughout.

Where Next

Now that you've had an introduction to using the HAProxy Kubernetes Ingress Controller, what should you do next? First, know that it is continuously evolving and new features are released at a regular cadence. Check out its official GitHub repository to see the latest version and read the official documentation.

The next thing is to appreciate that we've only scratched the surface. On the whole, Kubernetes covers a lot of ground, including authentication, resilience, deployments, and job scheduling. HAProxy fills a niche for routing and load balancing, but because the platform is extensible, you'll likely find new use cases and potential integrations between HAProxy and other components.

File an issue on GitHub or consider trying to solve an existing issue. If you have questions, you'll find people quick to help on the HAProxy Slack, where there's a channel (#ingress-controller) dedicated to Kubernetes. Perhaps you will be the next mentor to someone else who is picking up Kubernetes for the first time. Whatever the case, we're glad you've joined us.

Visit us at https://www.haproxy.com