Citrix ADC ingress controller
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>3</td>
</tr>
<tr>
<td>Getting started</td>
<td>5</td>
</tr>
<tr>
<td>Deployment topologies</td>
<td>7</td>
</tr>
<tr>
<td>Deploy Citrix ingress controller using YAML</td>
<td>14</td>
</tr>
<tr>
<td>Deploy the Citrix ingress controller using Helm charts</td>
<td>26</td>
</tr>
<tr>
<td>Deploy Citrix ingress controller using kops</td>
<td>29</td>
</tr>
<tr>
<td>Deploy the Citrix ingress controller on a Rancher managed Kubernetes cluster</td>
<td>30</td>
</tr>
<tr>
<td>Deploy the Citrix ingress controller on a PKS managed Kubernetes cluster</td>
<td>33</td>
</tr>
<tr>
<td>Expose services of type LoadBalancer</td>
<td>34</td>
</tr>
<tr>
<td>Deploy Citrix ADC-Integrated Canary Deployment Solution</td>
<td>53</td>
</tr>
<tr>
<td>Deploy Citrix API gateway using Red Hat OpenShift Operators</td>
<td>80</td>
</tr>
<tr>
<td>Deploying Citrix API gateway using Rancher</td>
<td>86</td>
</tr>
<tr>
<td>Deploy API Gateway with GitOps</td>
<td>88</td>
</tr>
<tr>
<td>Multi-cluster ingress and load balancing solution using the Citrix ingress controller</td>
<td>99</td>
</tr>
<tr>
<td>Service Mesh lite</td>
<td>117</td>
</tr>
<tr>
<td>Deploy the Citrix ingress controller as an OpenShift router plug-in</td>
<td>124</td>
</tr>
<tr>
<td>Deploy the Citrix ingress controller with OpenShift router sharding support</td>
<td>138</td>
</tr>
<tr>
<td>Deploy the Citrix ingress controller using Red Hat OpenShift Operators</td>
<td>142</td>
</tr>
<tr>
<td>Deploy Citrix ADC CPX as an Ingress device in an Azure Kubernetes Service cluster</td>
<td>169</td>
</tr>
<tr>
<td>Deploy Citrix ingress controller in an Azure Kubernetes Service cluster with Citrix ADC VPX</td>
<td>172</td>
</tr>
<tr>
<td>Deploy Citrix ADC CPX as an Ingress device in Google Cloud Platform</td>
<td>176</td>
</tr>
<tr>
<td>Deploy the Citrix ingress controller in Anthos</td>
<td>179</td>
</tr>
</tbody>
</table>
Deploy Citrix ADC VPX in active-active high availability in EKS environment using Amazon ELB and Citrix ingress controller:页188
Deploy the Citrix ingress controller for Citrix ADC with admin partitions:页201
Deploy Citrix solution for service of type LoadBalancer in AWS:页214
Multi-cloud and multi-cluster ingress and load balancing solution with Amazon EKS and Microsoft AKS clusters:页217
Annotations:页234
ConfigMap support for the Citrix ingress controller:页253
Ingress configurations:页257
Ingress class support:页262
Service class for services of type LoadBalancer:页269
Configure HTTP, TCP, or SSL profiles on Citrix ADC:页270
Log levels:页290
TCP profile support for services of type LoadBalancer:页291
SSL certificate for services of type LoadBalancer through the Kubernetes secret resource:页294
BGP advertisement of external IP addresses for type LoadBalancer services and Ingresses using Citrix ADC CPX:页298
Citrix ADC CPX integration with MetalLB in layer 2 mode for on-premises Kubernetes clusters:页311
Advanced content routing for Kubernetes Ingress using the HTTPRoute CRD:页316
Profile support for the Listener CRD:页320
IP address management using the Citrix IPAM controller for Ingress resources:页328
Apply CRDs through annotations:页333
Listener CRD support for Ingress through annotation:页336
Configuring consistent hashing algorithm using Citrix ingress controller:页342
Add DNS records using Citrix ADC ingress controller:页343
Configure static route on Ingress Citrix ADC VPX or MPX  345

Establish network between Kubernetes nodes and Ingress Citrix ADC using Citrix node controller  348

Exposé Service of type NodePort using Ingress  349

Configure pod to pod communication using Calico  352

Enhancements for Kubernetes service of type LoadBalancer support in the Citrix ingress controller  358

TLS certificates handling in Citrix ingress controller  365

TLS client authentication support in Citrix ADC  370

TLS server authentication support in Citrix ADC using the Citrix ingress controller  372

Install, link, and update certificates on a Citrix ADC using the Citrix ingress controller  374

Configure SSL passthrough using Kubernetes Ingress  377

Automated certificate management with cert-manager  379

Deploy HTTPS web application on Kubernetes with the Citrix ingress controller and Let’s Encrypt using cert-manager  381

Deploy an HTTPS web application on Kubernetes with Citrix ingress controller and HashiCorp Vault using cert-manager  403

Enable Citrix ADC certificate validation in the Citrix ingress controller  423

Disable API server certificate verification  426

View metrics of Citrix ADCs using Prometheus and Grafana  426

Analytics and observability  437

Analytics configuration support using ConfigMap  441

Troubleshooting  445

Troubleshooting the Citrix ingress controller during runtime  455

Call Home enablement for the Citrix ingress controller in Citrix ADC  457
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Rewrite and Responder policies in Kubernetes</td>
<td>604</td>
</tr>
<tr>
<td>VIP CustomResourceDefinitions</td>
<td>627</td>
</tr>
<tr>
<td>Advanced content routing for Kubernetes with Citrix ADC</td>
<td>628</td>
</tr>
<tr>
<td>Configure web application firewall policies with the Citrix ingress controller</td>
<td>635</td>
</tr>
<tr>
<td>Configure bot management policies with the Citrix ingress controller</td>
<td>652</td>
</tr>
<tr>
<td>Configure cross-origin resource sharing policies with Citrix ingress controller</td>
<td>661</td>
</tr>
<tr>
<td>Enable request retry feature using AppQoE for Citrix ingress controller</td>
<td>664</td>
</tr>
<tr>
<td>Entity name change</td>
<td>666</td>
</tr>
<tr>
<td>Licensing</td>
<td>668</td>
</tr>
<tr>
<td>Deployment using Helm charts and Citrix deployment builder</td>
<td>669</td>
</tr>
</tbody>
</table>
Overview

What is an Ingress Controller in Kubernetes

When you are running an application inside a Kubernetes cluster, you need to provide a way for external users to access the applications from outside the Kubernetes cluster. Kubernetes provides an object called Ingress which allows you to define the rules for accessing the services within the Kubernetes cluster. It provides the most effective way to externally access multiple services running inside the cluster using a stable IP address.

An Ingress Controller is an application deployed inside the cluster that interprets rules defined in the Ingress. The Ingress Controller converts the Ingress rules into configuration instructions for a load balancing application integrated with the cluster. The load balancer can be a software application running inside your Kubernetes cluster or a hardware appliance running outside the cluster.

What is Citrix ADC ingress controller

Citrix provides an implementation of the Kubernetes Ingress Controller to manage and route traffic into your Kubernetes cluster using Citrix ADCs (Citrix ADC CPX, VPX, or MPX).

Using Citrix ADC ingress controller, you can configure Citrix ADC CPX, VPX, or MPX according to the Ingress rules and integrate your Citrix ADCs with the Kubernetes environment.

Why Citrix ADC ingress controller

This topic provides information about some of the key benefits of integrating Citrix ADCs with your Kubernetes cluster using Citrix ADC ingress controller.

Support for TCP and UDP traffic

Standard Kubernetes Ingress solutions provide load balancing only at layer 7 (HTTP or HTTPS traffic). Sometimes, you need to expose many legacy applications which rely on TCP or UDP applications and need a way to load balance those applications. Citrix Kubernetes Ingress solution using Citrix ADC ingress controller provides TCP, TCP-SSL, and UDP traffic support apart from the standard HTTP or HTTPS Ingress. Also, it works seamlessly across multiple clouds or on-premises data centers.

Advanced traffic management policies

Citrix ADC provides enterprise-grade traffic management policies like rewrite and responder policies for efficiently load balancing traffic at layer 7. However, Kubernetes Ingress lacks such enterprise-grade traffic management policies. With the Kubernetes Ingress solution from Citrix, you can apply
Citrix ADC ingress controller

rewrite and responder policies for application traffic in a Kubernetes environment using CRDs provided by Citrix.

Flexible deployment topologies

Citrix provides flexible and powerful topologies such as Single-Tier and Dual-Tier depending on how you want to manage your Citrix ADCs and Kubernetes environment. For more information on the deployment topologies, see the Deployment topologies page.

Layer 7 load balancing support for East-West traffic

For traffic between microservices inside the Kubernetes cluster (East-West traffic), Kubernetes natively provides only limited layer 4 load balancing. Using Citrix ADC CPX along with the Ingress controller, you can achieve advanced layer 7 load balancing for East-West traffic.

Service of type LoadBalancer on bare metal clusters

There may be several situations where you want to deploy your Kubernetes cluster on bare metal or on-premises rather than deploy it on public cloud. When you are running your applications on bare metal Kubernetes clusters, it is much easier to route TCP or UDP traffic using a service of type LoadBalancer than using Ingress. Even for HTTP traffic, it is sometimes more convenient than Ingress. However, there is no load balancer implementation natively available for bare metal Kubernetes clusters. Citrix provides a way to load balance such services using the Ingress controller and Citrix ADC. For more information, see Expose services of type LoadBalancer.

Deploy Citrix ADC ingress controller

You can deploy Citrix ADC ingress controller in the following deployment modes:

1. As a standalone pod: This mode is used when managing ADCs such as Citrix ADC MPX, or VPX that is outside the Kubernetes cluster.

2. As a sidecar in a pod along with the Citrix ADC CPX in the same pod: The controller is only responsible for the Citrix ADC CPX that resides in the same pod.

You can deploy the ingress controller provided by Citrix using Kubernetes YAML or Helm charts. For more information, see Deploy Citrix ADC ingress controller using YAML or Deploy Citrix ADC ingress controller using Helm charts.
Getting started

February 3, 2022

This guide helps you to quickly evaluate Citrix ADC ingress controller for Kubernetes if you are new to Citrix ingress controller. If you are an advanced user, see What is Next.

Before you begin

Ensure that you have installed and set up a Minikube cluster.

Getting started with Citrix ADC ingress controller

In this procedure you perform the following steps:

- Deploy Citrix ADC CPX (a containerized version of Citrix ADC) along with ingress controller
- Deploy Guestbook, a sample application
- Deploy Ingress rules to route traffic to the Guestbook application
- Send some traffic to the application and verify

Deploy Citrix ADC CPX with Citrix ADC ingress controller

Perform the following to deploy Citrix ADC CPX with Citrix ADC ingress controller.

1. Deploy Citrix ADC CPX as an Ingress proxy in the Minikube cluster.

```
```

2. Verify the installation using the following command.

```
1  kubectl get pods -l app=cpx-ingress
```

Deploy a sample application

In this step, you deploy Guestbook which is a multi-tier PHP-based web application that uses Redis.

1. Deploy the Guestbook application in Minikube.
Citrix ADC ingress controller


2. Verify the installation using the following:

   1. **kubectl get pods -l 'app in (guestbook, redis)'**

**Deploy an Ingress for the sample application**

To deploy ingress rules for the sample application and verify the functionality, perform the following steps.

1. Deploy an Ingress rule that sends traffic to the Guestbook application([http://www.guestbook.com](http://www.guestbook.com)).


2. Get the Ingress IP address using the following command.

   1. **kubectl get ingress**
   2. **kubectl get svc cpx-service**

3. Send traffic to the Guestbook microservice application and verify that traffic to this URL gets the Guestbook page:


**Expected output:**

1. `<title>Guestbook</title>`
2. `<h2>Guestbook</h2>`
Citrix ADC ingress controller

**What is next**

The getting started section helps a beginner to evaluate Citrix ADC ingress controller quickly and the installation covers only the basic functionality. You can see the following topics for comprehensive information on deploying Citrix ADC ingress controller and customize your installation accordingly.

- **Deployment topologies**: Provides information on various topologies supported by Citrix ADC ingress controller.
- **Supported platforms**: Provides information about the different platforms supported including bare metal and cloud platforms.
- **Deploy Citrix ingress controller**: Provides information on how to deploy Citrix ADC ingress controller for different flavors of Citrix ADC like Citrix ADC CPX, VPX, and MPX.

**Deployment topologies**

February 4, 2022

Citrix ADCs can be combined in powerful and flexible topologies that complement organizational boundaries. Dual-tier deployments employ high-capacity hardware or virtualized Citrix ADCs (Citrix ADC MPX and VPX) in the first tier to offload security functions and implement relatively static organizational policies while segmenting control between network operators and Kubernetes operators.

In Dual-tier deployments, the second tier is within the Kubernetes Cluster (using the Citrix ADC CPX) and is under control of the service owners. This setup provides stability for network operators, while allowing Kubernetes users to implement high-velocity changes. Single-tier topologies are suited to organizations that need to handle high rates of change.

**Single-Tier topology**

In a Single-Tier topology, Citrix ADC MPX or VPX devices proxy the (North-South) traffic from the clients to microservices inside the cluster. The Citrix ingress controller is deployed as a standalone pod in the Kubernetes cluster. The controller automates the configuration of Citrix ADCs (MPX or VPX) based on the changes to the microservices or the Ingress resources.
Dual-Tier topology

In Dual-Tier topology, Citrix ADC MPX or VPX devices in Tier-1 proxy the traffic (North-South) from the client to Citrix ADC CPXs in Tier-2. The Tier-2 Citrix ADC CPX then routes the traffic to the microservices in the Kubernetes cluster. The Citrix ingress controller deployed as a standalone pod configures the Tier-1 devices. And, the sidecar controller in one or more Citrix ADC CPX pods configures the associated Citrix ADC CPX in the same pod.
Cloud topology

Kubernetes clusters in public clouds such as Amazon Web Services (AWS), Google Cloud, and Microsoft Azure can use their native load balancing services such as, AWS Elastic Load Balancing, Google Cloud Load Balancing, and Microsoft Azure NLB as the first (relatively static) tier of load balancing to a second tier of Citrix ADC CPX. Citrix ADC CPX operates inside the Kubernetes cluster with the sidecar Ingress controller. The Kubernetes clusters can be self-hosted or managed by the cloud provider (for example, AWS EKS, Google GKE and Azure AKS) while using the Citrix ADC CPX as the Ingress. If the cloud-based Kubernetes cluster is self-hosted or self-managed, the Citrix ADC VPX can be used as the first tier in a Dual-tier topology.

Cloud deployment with Citrix ADC (VPX) in tier-1:
Cloud deployment with Cloud LB in tier-1:

Service mesh lite

An Ingress solution (either hardware or virtualized or containerized) typically performs L7 proxy functions for north-south (N-S) traffic. The Service Mesh lite architecture uses the same Ingress solution
to manage east-west traffic as well.

In a standard Kubernetes deployment, east-west (E-W) traffic traverses the built-in KubeProxy deployed in each node. Kube-proxy being a L4 proxy can only do TCP/UDP based load balancing without the benefits of L7 proxy.

Citrix ADC (MPX, VPX, or CPX) can provide such benefits for E-W traffic such as:

- Mutual TLS or SSL offload
- Content based routing, allow or block traffic based on HTTP or HTTPS header parameters
- Advanced load balancing algorithms (for example, least connections, least response time and so on.)
- Observability of east-west traffic through measuring golden signals (errors, latencies, saturation, or traffic volume). Citrix ADM’s Service Graph is an observability solution to monitor and debug microservices.

For more information, see Service mesh lite.

**Services of type LoadBalancer**

Services of type LoadBalancer in Kubernetes enables you to directly expose services to the outside world without using an ingress resource. It is made available only by cloud providers, who spin up their own native cloud load balancers and assign an external IP address through which the service is accessed. This helps you to deploy microservices easily and expose them outside the Kubernetes cluster.

By default, in a bare metal Kubernetes cluster, service of type LoadBalancer simply exposes NodePorts for the service. And, it does not configure external load balancers.

The Citrix ingress controller supports the services of type LoadBalancer. You can create a service of type LoadBalancer and expose it using the ingress Citrix ADC in Tier-1. The ingress Citrix ADC provisions a load balancer for the service and an external IP address is assigned to the service. The Citrix ingress controller allocates the IP address using the Citrix IPAM controller.

For more information, see Expose services of type LoadBalancer.
Citrix ADC ingress controller

**Services of type NodePort**

By default, Kubernetes services are accessible using the cluster IP address. The cluster IP address is an internal IP address that can be accessed within the Kubernetes cluster. To make the service accessible from the outside of the Kubernetes cluster, you can create a service of the type **NodePort**.

The Citrix ingress controller supports services of type **NodePort**. Using the Ingress Citrix ADC and Citrix ingress controller, you can expose the service of type **NodePort** to the outside world.

For more information, see [Expose services of type NodePort](#).
Deployment using Helm charts and the Citrix deployment builder

For deploying Citrix cloud native topologies, there are various options available using YAML and Helm charts. Helm charts are one of the easiest ways for deployment in a Kubernetes environment. When you deploy using the Helm charts, you can use a `values.yaml` file to specify the values of the configurable parameters instead of providing each parameter as an argument.

You can generate the `values.yaml` file for Citrix cloud native deployments using the Citrix deployment builder, which is a GUI.

The following topologies are supported by the Citrix deployment builder:

- Single-Tier
  - Ingress
  - Service type LoadBalancer
- Dual-Tier
  - Citrix ADC CPX as NodePort
  - Citrix ADC CPX as service of type LoadBalancer
- Multi-cluster Ingress
- Service mesh
Deploy Citrix ingress controller using YAML

July 5, 2022

You can deploy Citrix ingress controller in the following modes on your bare metal and cloud deployments:

- As a standalone pod in the Kubernetes cluster. Use this mode if you are controlling Citrix ADCs (Citrix ADC MPX or Citrix ADC VPX) outside the cluster. For example, with dual-tier topologies, or single-tier topology where the single tier is a Citrix ADC MPX or VPX.

- As a sidecar (in the same pod) with Citrix ADC CPX in the Kubernetes cluster. The sidecar controller is only responsible for the associated Citrix ADC CPX within the same pod. This mode is used in dual-tier or cloud topologies.

Deploy Citrix ingress controller as a standalone pod in the Kubernetes cluster for Citrix ADC MPX or VPX appliances

Use the citrix-k8s-ingress-controller.yaml file to run the Citrix ingress controller as a standalone pod in your Kubernetes cluster.

Note:

The Citrix ADC MPX or VPX can be deployed in standalone, high-availability, or clustered modes.

Prerequisites

- Determine the NS_IP IP address needed by the controller to communicate with the appliance. The IP address might be anyone of the following depending on the type of Citrix ADC deployment:
  - (Standalone appliances) NSIP - The management IP address of a standalone Citrix ADC appliance. For more information, see IP Addressing in Citrix ADC
  - (Appliances in High Availability mode) SNIP - The subnet IP address. For more information, see IP Addressing in Citrix ADC
  - (Appliances in Clustered mode) CLIP - The cluster management IP (CLIP) address for a clustered Citrix ADC deployment. For more information, see IP addressing for a cluster

- The user name and password of the Citrix ADC VPX or MPX appliance used as the Ingress device. The Citrix ADC appliance must have a system user account (non-default) with certain privileges.
so that Citrix ingress controller can configure the Citrix ADC VPX or MPX appliance. For instructions to create the system user account on Citrix ADC, see Create System User Account for Citrix ingress controller in Citrix ADC.

You can directly pass the user name and password as environment variables to the controller, or use Kubernetes secrets (recommended). If you want to use Kubernetes secrets, create a secret for the user name and password using the following command:

```
kubectl create secret generic nslogin --from-literal=username='cic' --from-literal=password='mypassword'
```

**Create System User Account for Citrix ingress controller in Citrix ADC**

Citrix ingress controller configures the Citrix ADC appliance (MPX or VPX) using a system user account of the Citrix ADC. The system user account should have certain privileges so that the Citrix ingress controller has permission to configure the following on the Citrix ADC:

- Add, Delete, or View Content Switching (CS) virtual server
- Configure CS policies and actions
- Configure Load Balancing (LB) virtual server
- Configure Service groups
- Configure SSL certkeys
- Configure routes
- Configure user monitors
- Add system file (for uploading SSL certkeys from Kubernetes)
- Configure Virtual IP address (VIP)
- Check the status of the Citrix ADC appliance

To create the system user account, perform the following:

1. Log on to the Citrix ADC appliance. Perform the following:
   a) Use an SSH client, such as PuTTY, to open an SSH connection to the Citrix ADC appliance.
   b) Log on to the appliance by using the administrator credentials.

2. Create the system user account using the following command:

```
add system user <username> <password>
```

For example:
3. Create a policy to provide required permissions to the system user account. Use the following command:

```
add system user cic mypassword
```

Note:
The system user account would have privileges based on the command policy that you define.

The command policy mentioned in step 3 is similar to the built-in `sysAdmin` command policy with additional permission to upload files.

In the command policy specification provided, special characters which need to be escaped are already omitted to easily copy-paste into the Citrix ADC command line.

For configuring the command policy from the Citrix ADC configuration wizard (GUI), use the following command policy specification:

```
add cmdpolicy cic-policy ALLOW `^(!\!shell)(?!\!sftp)(?!\!scp)(?!\!batch)(?!\!source)(?!\!*superuser)(?!\!*nsroot)(?!\!*install)(?!\!show\s+system\s+(user|cmdPolicy|file))(?!\!(set|add|rm|create|export|kill)\s+system)(?!\!(unbind|bind)\s+system\s+(user|group))(?!\!diff\s+ns\s+config)(?!\!(set|unset|add|rm|bind|unbind|switch)\s+ns\s+partition).*\!(^\!\!install\s*(wi|wf))\!(^\!\!s+system\s+file)`
```
4. Bind the policy to the system user account using the following command:

```
bind system user cic cic-policy 0
```

**Deploy Citrix ingress controller as a pod**

Perform the following:

1. Download the `citrix-k8s-ingress-controller.yaml` using the following command:

```
wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/baremetal/citrix-k8s-ingress-controller.yaml
```

2. Edit the `citrix-k8s-ingress-controller.yaml` file and enter the values for the following environmental variables:

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Mandatory or Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS_IP</td>
<td>Mandatory</td>
<td>The IP address of the Citrix ADC appliance. For more details, see Prerequisites.</td>
</tr>
<tr>
<td>NS_USER and NS_PASSWORD</td>
<td>Mandatory</td>
<td>The user name and password of the Citrix ADC VPX or MPX appliance used as the Ingress device. For more details, see Prerequisites.</td>
</tr>
<tr>
<td>EULA</td>
<td>Mandatory</td>
<td>The End User License Agreement. Specify the value as Yes.</td>
</tr>
<tr>
<td>Kubernetes_url</td>
<td>Optional</td>
<td>The kube-apiserver url that Citrix ingress controller uses to register the events. If the value is not specified, Citrix ingress controller uses the internal kube-apiserver IP address.</td>
</tr>
</tbody>
</table>
## Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Mandatory or Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOGLEVEL</td>
<td>Optional</td>
<td>The log levels to control the logs generated by Citrix ingress controller. By default, the value is set to DEBUG. The supported values are: CRITICAL, ERROR, WARNING, INFO, and DEBUG. For more information, see <a href="#">Log Levels</a>.</td>
</tr>
<tr>
<td>NS_PROTOCOL and NS_PORT</td>
<td>Optional</td>
<td>Defines the protocol and port that is used by the Citrix ingress controller to communicate with Citrix ADC. By default, Citrix ingress controller uses HTTP on port 80. You can also use HTTPS on port 443.</td>
</tr>
<tr>
<td>ingress-classes</td>
<td>Optional</td>
<td>If multiple ingress load balancers are used to load balance different ingress resources. You can use this environment variable to specify the Citrix ingress controller to configure Citrix ADC associated with specific ingress class. For information on Ingress classes, see <a href="#">Ingress class support</a>.</td>
</tr>
</tbody>
</table>
### Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Mandatory or Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS_VIP</td>
<td>Optional</td>
<td>Citrix ingress controller uses the IP address provided in this environment variable to configure a virtual IP address to the Citrix ADC that receives Ingress traffic. <strong>Note:</strong> NS_VIP acts as a fallback when the <code>frontend-ip</code> annotation is not provided in Ingress yaml. Not supported for Type Loadbalancer service.</td>
</tr>
<tr>
<td>NS_APPS_NAME_PREFIX</td>
<td>Optional</td>
<td>By default, the Citrix ingress controller adds “<strong>k8s</strong>” as prefix to the Citrix ADC entities such as, content switching (CS) virtual server, load balancing (LB) virtual server and so on. You can now customize the prefix using the <code>NS_APPS_NAME_PREFIX</code> environment variable in the Citrix ingress controller deployment YAML file. You can use alphanumeric characters for the prefix and the prefix length should not exceed 8 characters.</td>
</tr>
<tr>
<td>Environment Variable</td>
<td>Mandatory or Optional</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NAMESPACE</td>
<td>Optional</td>
<td>While running a Citrix ingress controller with Role based RBAC, you must provide the namespace which you want to listen or get events. This namespace must be same as the one used for creating the service account. Using the service account, the Citrix ingress controller can listen on a namespace. You can use the NAMESPACE environment variable to specify the namespace. For more information, see Deploy the Citrix ingress controller for a namespace.</td>
</tr>
<tr>
<td>POD_IPS_FOR_SERVICEGROUP</td>
<td>Optional</td>
<td>By default, while configuring services of type LoadBalancer and NodePort on an external tier-1 Citrix ADC the Citrix ingress controller adds NodeIP and NodePort as service group members. If this variable is set as True, pod IP address and port are added instead of NodeIP and NodePort as service group members.</td>
</tr>
</tbody>
</table>
### Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Mandatory or Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGNORE_NODE_EXTERNAL_IP</td>
<td>Optional</td>
<td>While adding NodeIP for services of type LoadBalancer or NodePort on an external tier-1 Citrix ADC, the Citrix ingress controller prioritizes an external IP address over an internal IP address. When you want to prefer an internal IP address over an external IP address for NodeIP, you can set this variable to True.</td>
</tr>
<tr>
<td>NS_DNS_NAMESERVER</td>
<td>Optional</td>
<td>Enables adding DNS nameservers on Citrix ADC VPX.</td>
</tr>
<tr>
<td>NS_CONFIG_DNS_REC</td>
<td>Optional</td>
<td>Enables adding DNS records on Citrix ADC for Ingress resources. This variable is configured at the boot time and cannot be changed at runtime. Possible values are true or false. The default value is false and you need to set it as true to enable the DNS server configuration. When you set the value as 'true', the corresponding command add dns addrec &lt;abc.com 1.1.1.1&gt; is executed on Citrix ADC and an address record (mapping of the domain name to IP address) is created. For more information, see Create address records for a domain name.</td>
</tr>
</tbody>
</table>
Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Mandatory or Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS_SVC_LB_DNS_REC</td>
<td>Optional</td>
<td>Enables adding DNS records on Citrix ADC for services of type LoadBalancer. This variable is configured at the boot time and cannot be changed at runtime. Possible values are true or false. The default value is false and you need to set it as true to enable the DNS server configuration.</td>
</tr>
</tbody>
</table>

3. Once you update the environment variables, save the YAML file and deploy it using the following command:

```
kubectl create -f citrix-k8s-ingress-controller.yaml
```

4. Verify if Citrix ingress controller is deployed successfully using the following command:

```
kubectl get pods --all-namespaces
```

**Deploy Citrix ingress controller as a sidecar with Citrix ADC CPX**

Use the citrix-k8s-cpx-ingress.yaml file to deploy a Citrix ADC CPX with Citrix ingress controller as a sidecar. The YAML file deploys a Citrix ADC CPX instance that is used for load balancing the North-South traffic to the microservices in your Kubernetes cluster.

Perform the following:

1. Download the citrix-k8s-cpx-ingress.yaml using the following command:

```
```
2. Deploy the `citrix-k8s-cpx-ingress.yaml` file using the following command:

```bash
kubectl create -f citrix-k8s-cpx-ingress.yaml
```

3. Verify if Citrix ingress controller is deployed successfully using the following command:

```bash
kubectl get pods --all-namespaces
```

**Deploy Citrix ADC CPX with the Citrix ingress controller as sidecar without the default credentials**

Earlier, when you deploy Citrix ADC CPX with the Citrix ingress controller as a sidecar without specifying the login credentials, the Citrix ingress controller considers `nsroot/nsroot` as the default credentials.

With the latest Citrix ADC CPX versions (Citrix ADC CPX 13.0.64.35 and later), the default credentials are removed. So, when you deploy the Citrix ingress controller as a sidecar with the latest versions of Citrix ADC CPX, the Citrix ingress controller can get the credentials from Citrix ADC CPX through the `/var/deviceinfo/random_id` file in the Citrix ADC CPX. This file can be shared between the Citrix ADC CPX and the Citrix ingress controller through the volume mount.

Depending on whether you are using the latest Citrix ADC CPX version or an older version, you need to choose one of the following deployment YAML files. For older versions of Citrix ADC CPX, you need to specify the credentials in the YAML file.

- For Citrix ADC CPX 13.0.64.35 and later versions, use the following YAML:

  `citrix-k8s-cpx-ingress.yml`

As provided in the YAML, the following is a snippet of the volume mount configuration required in the YAML file both for the Citrix ingress controller and Citrix ADC CPX:

```yaml
volumeMounts:
  - mountPath: /var/deviceinfo
    name: shared-data
```

Following is a snippet of the shared volume configuration common for the Citrix ADC CPX and the Citrix ingress controller.
Citrix ADC ingress controller

```yaml
volumes:
- name: shared-data
  emptyDir: {}
```

- For earlier Citrix ADC CPX versions (versions earlier than 13.0.64.35), use the following YAML:

  cpx-ingress-previous.yaml

  Following is a snippet of the credential section in the Citrix ingress controller:

```yaml
- name: "NS_USER"
  valueFrom:
    secretKeyRef:
      name: nslogin
      key: username
- name: "NS_PASSWORD"
  valueFrom:
    secretKeyRef:
      name: nslogin
      key: password
```

**Deploy the Citrix ingress controller for a namespace**

In Kubernetes, a role consists of rules that define a set of permissions that can be performed on a set of resources. In an RBAC enabled Kubernetes environment, you can create two kinds of roles based on the scope you need:

- Role
- ClusterRole

A role can be defined within a namespace with a `Role`, or cluster-wide with a `ClusterRole`. You can create a `Role` to grant access to resources within a single namespace.

In Kubernetes, you can create multiple virtual clusters on the same physical cluster. Namespaces provides a way to divide cluster resources between multiple users and useful in environments with many users spread across multiple teams, or projects.

By default, the Citrix ingress controller monitors Ingress resources across all namespaces in the Kubernetes cluster. If multiple teams want to manage the same Citrix ADC, they can deploy a `Role` based Citrix ingress controller to monitor only ingress resources belongs to a specific namespace. This namespace must be same as the namespace you have provided for creating the service account. You need to create a `Role` and bind the role to the service account for the Citrix ingress controller. In
Citrix ADC ingress controller

div class="highlight"

```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: cic-k8s-role
rules:
  - apiGroups: [""]
    resources: ["endpoints", "ingresses", "pods", "secrets", "nodes", "routes", "namespaces"]
    verbs: ["get", "list", "watch"]
# services/status is needed to update the loadbalancer IP in service status for integrating
# service of type LoadBalancer with external-dns
  - apiGroups: [""]
    resources: ["services/status"]
    verbs: ["patch"]
  - apiGroups: [""]
    resources: ["services"]
    verbs: ["get", "list", "watch", "patch"]
  - apiGroups: ["extensions"]
    resources: ["ingresses", "ingresses/status"]
    verbs: ["get", "list", "watch"]
  - apiGroups: ["apps"]
    resources: ["deployments"]
    verbs: ["get", "list", "watch"]
---

kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: cic-k8s-role
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: cic-k8s-role
subjects:
```

this case, the Citrix ingress controller listens only for events from the specified namespace and then configure the Citrix ADC accordingly.

The following example shows a sample YAML file which defines a Role and RoleBinding for deploying a Citrix ingress controller for a specific namespace.
Restrictions

When the Citrix ingress controller runs with a Role (scope with in a namespace), the following functionalities are not supported as they require global scope.

- configuring static routes
- watching on all namespaces
- CRDs

Deploy the Citrix ingress controller using Helm charts

February 4, 2022

You can deploy the Citrix ingress controller in the following modes on your bare metal and cloud deployments:

- As a standalone pod in the Kubernetes cluster. Use this mode if you are controlling Citrix ADCs (Citrix ADC MPX or Citrix ADC VPX) outside the cluster. For example, with dual-tier topologies, or single-tier topology where the single tier is a Citrix ADC MPX or VPX.

- As a sidecar (in the same pod) with Citrix ADC CPX in the Kubernetes cluster. The sidecar controller is only responsible for the associated Citrix ADC CPX within the same pod. This mode is used in dual-tier or cloud) topologies.

The helm charts for the Citrix ingress controller are available on Artifact Hub.

When you deploy using the Helm charts, you can use a values.yaml file to specify the values of the configurable parameters instead of providing each parameter as an argument. For ease of use, Citrix provides the Citrix deployment builder which is a GUI for generating the values.yaml file for Citrix cloud native deployments.

Deploy the Citrix ingress controller as a standalone pod in the Kubernetes cluster

Use the citrix-ingress-controller chart to run the Citrix ingress controller as a pod in your Kubernetes cluster. The chart deploys the Citrix ingress controller as a pod in your Kubernetes cluster and configures the Citrix ADC VPX or MPX ingress device.
Prerequisites

- Determine the NS_IP address needed by the controller to communicate with the appliance. The IP address might be any of the following depending on the type of Citrix ADC deployment:
  - (Standalone appliances) NSIP - The management IP address of a standalone Citrix ADC appliance. For more information, see IP Addressing in Citrix ADC.
  - (Appliances in High Availability mode) SNIP - The subnet IP address. For more information, see IP Addressing in Citrix ADC.
  - (Appliances in Clustered mode) CLIP - The cluster management IP (CLIP) address for a clustered Citrix ADC deployment. For more information, see IP addressing for a cluster.
- The user name and password of the Citrix ADC VPX or MPX appliance used as the Ingress device. The Citrix ADC appliance needs to have a system user account (non-default) with certain privileges so that the Citrix ingress controller can configure the Citrix ADC VPX or MPX appliance. For instructions to create the system user account on Citrix ADC, see Create System User Account for Citrix ingress controller in Citrix ADC.

You can directly pass the user name and password or use Kubernetes secrets. If you want to use Kubernetes secrets, create a secret for the user name and password using the following command:

```
kubectl create secret generic nslogin --from-literal=username='cic' --from-literal=password='mypassowrd'
```

Create a system user account for the Citrix ingress controller in Citrix ADC

The Citrix ingress controller configures the Citrix ADC using a system user account of the Citrix ADC. The system user account should have certain privileges so that the Citrix ingress controller has permission to configure the following on the Citrix ADC:

- Add, delete, or view content switching (CS) virtual server
- Configure CS policies and actions
- Configure Load Balancing (LB) virtual server
- Configure service groups
- Configure SSL certkeys
- Configure routes
- Configure user monitors
- Add system file (for uploading SSL certkeys from Kubernetes)
- Configure Virtual IP address (VIP)
- Check the status of the Citrix ADC appliance
To create the system user account, perform the following:

1. Log on to the Citrix ADC appliance. Perform the following:
   a) Use an SSH client, such as PuTTy, to open an SSH connection to the Citrix ADC appliance.
   b) Log on to the appliance by using the administrator credentials.

2. Create the system user account using the following command:

   ```shell
   add system user <username> <password>
   ```

   For example:

   ```shell
   add system user cic mypassword
   ```

3. Create a policy to provide required permissions to the system user account. Use the following command:

   ```shell
   add cmdpolicy cic-policy ALLOW '^(?!)shell(?!)sftp(?!)scp(?!)batch(?!)source(?!)superuser(?!)nsroot(?!)install(?!)show(?!)system(?!)user(?!)cmdPolicy(?!)file(?!)set(?!)add(?!)rm(?!)create(?!)export(?!)kill(?!)unbind(?!)bind(?!)switch(?!)ns(?!)partition.*|(?!)install(?!)s+(wi|wf)|(?!)S+(system|user|cmdPolicy|file)\^(?!)shell(?!)sftp(?!)scp(?!)batch(?!)source(?!)superuser(?!)nsroot(?!)install(?!)show(?!)system(?!)user(?!)cmdPolicy|file(?!)set(?!)add(?!)rm(?!)create(?!)export(?!)kill(?!)unbind(?!)bind(?!)switch(?!)ns(?!)partition.*|(?!)install(?!)s+(wi|wf)|(?!)S+(system|user|file)'```

**Note:** The system user account would have privileges based on the command policy that you define.

The command policy mentioned in step 3 is similar to the built-in sysAdmin command policy with additional permission to upload files.

In the command policy specification provided, special characters which need to be escaped are already omitted to easily copy-paste into the Citrix ADC command line.

For configuring the command policy from Citrix ADC configuration wizard (GUI), use the following command policy specification.
4. Bind the policy to the system user account using the following command:

```
bind system user cic cic-policy 0
```

To deploy the Citrix ingress controller as a standalone pod:

To deploy the Citrix ingress controller as a standalone pod, follow the instructions provided in the Citrix ingress controller Artifact Hub.

**Deploy the Citrix ingress controller as a sidecar with Citrix ADC CPX in the Kubernetes cluster**

Use the citrix-cpx-with-ingress-controller chart to deploy a Citrix ADC CPX with Citrix ingress controller as a sidecar. The chart deploys a Citrix ADC CPX instance that is used for load balancing the North-South traffic to the microservices in your Kubernetes cluster. The sidecar Citrix ingress controller configures the Citrix ADC CPX.

To deploy Citrix ADC CPX with the Citrix ingress controller as a sidecar, follow the instruction provided in the Citrix ingress controller Helm Hub.

**Deploy Citrix ingress controller using kops**

February 1, 2022

Kops (Kubernetes Operations) is a set of tools for creating and maintaining Kubernetes clusters in the cloud. Using kops, you can also deploy and manage cluster add-ons which extend the functionality of Kubernetes. Citrix provides a kops add-on for deploying Citrix ingress controller.
Citrix ADC ingress controller

**Deploy Citrix ingress controller using kops during cluster creation**

Perform the following steps to deploy Citrix ingress controller using kops while creating a cluster.

1. Edit the cluster YAML manifest before creating the cluster.

   ```
   kops edit cluster <cluster-name>
   ```

2. Add the Citrix ingress controller add-on specification to the cluster YAML manifest in the section `spec.addons`.

   ```
   addons:
   - manifest: ingress-citrix
   ```

For more information on how to enable an add-on during Kubernetes cluster creation, see `kops addon`.

**Deploy Citrix ingress controller using kops after cluster creation**

You can use the `kubectl` command to deploy the Citrix ingress controller add-on with kops after creating the cluster.

1. `kubectl create secret generic nslogin --from-literal=username='nsroot' --from-literal=password=nsroot`
2. `kubectl create -f https://raw.githubusercontent.com/kubernetes/kops/master/addons/ingress-citrix/v1.1.1.yaml`

**Deploy the Citrix ingress controller on a Rancher managed Kubernetes cluster**

February 8, 2022

Rancher is an open-source platform with an intuitive user interface that helps you to easily deploy and manage Kubernetes clusters. Rancher supports Kubernetes clusters on any infrastructure be on cloud or on-premises deployment. Rancher also allows you to centrally manage multiple clusters running across your organization.
The Citrix ingress controller is built around the Kubernetes Ingress and it can automatically configure one or more Citrix ADCs based on the Ingress resource configuration. You can deploy the Citrix ingress controller in a Rancher managed Kubernetes cluster to extend the advanced load balancing and traffic management capabilities of Citrix ADC to your cluster.

Prerequisites
You must create a Kubernetes cluster and import the cluster on the Rancher platform.

Deployment options
You can either deploy Citrix ADC CPXs as pods inside the cluster or deploy a Citrix ADC MPX or VPX appliance outside the Kubernetes cluster.

Based on how you want to use Citrix ADC, there are two ways to deploy the Citrix ingress controller in a Kubernetes cluster on the Rancher platform:

- As a sidecar container alongside Citrix ADC CPX in the same pod: In this mode, Citrix ingress controller configures the Citrix ADC CPX.
- As a standalone pod in the Kubernetes cluster: In this mode, you can control the Citrix ADC MPX or VPX appliance deployed outside the cluster.

Deploy the Citrix ingress controller as a sidecar with Citrix ADC CPX
In this deployment, you can use the Citrix ADC CPX instance for load balancing the North-South traffic to microservices in your Kubernetes cluster. Citrix ingress controller is deployed as a sidecar alongside the Citrix ADC CPX container in the same pod using the `citrix-k8s-cpx-ingress.yaml` file.

Perform the following steps to deploy the Citrix ingress controller as a sidecar with Citrix ADC CPX on the Rancher platform.

1. Download the `citrix-k8s-cpx-ingress.yaml` file using the following command.

   ```
   ```

2. On the Rancher GUI cluster page, select Clusters from Global view.
3. From the Clusters page, open the cluster that you want to access.
4. Click Launch `kubectl` to open a terminal for interacting with your Kubernetes cluster.
5. Create a file named `cpx.yaml` in the launched terminal and then copy the contents of the modified `citrix-k8s-cpx-ingress.yaml` file to the `cpx.yaml` file.

6. Deploy the newly created YAML file using the following command.

```
kubectl create -f cpx.yaml
```

7. Verify if Citrix ingress controller is deployed successfully using the following command.

```
kubectl get pods --all-namespaces
```

**Deploy the Citrix ingress controller as a standalone pod**

In this deployment, Citrix ingress controller which runs as a stand-alone pod allows you to control the Citrix ADC MPX, or VPX appliance from the Kubernetes cluster. You can use the `citrix-k8s-ingress-controller.yaml` file for this deployment.

**Before you begin:** Ensure that you complete all the prerequisites required for deploying the Citrix ingress controller.

To deploy the Citrix ingress controller as a standalone pod on the Rancher platform:

1. Download the `citrix-k8s-ingress-controller.yaml` file using the following command:

```
wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/baremetal/citrix-k8s-ingress-controller.yaml
```

2. Edit the `citrix-k8s-ingress-controller.yaml` file and enter the values of the environment variable using the information in Deploy Citrix ingress controller as a pod.

**Note:** To update the `Status.LoadBalancer.Ingress` field of the Ingress resources managed by the Citrix ingress controller with the allocated IP addresses, you must specify the command line argument `--update-ingress-status yes` when you start the Citrix ingress controller. For more information, see Updating the Ingress status for the Ingress resources with the specified IP address.

3. On the Rancher GUI cluster page, select Clusters from Global view.

4. From the Clusters page, open the cluster that you want to access.

5. Click Launch `kubectl` to open a terminal for interacting with your Kubernetes cluster.
Citrix ADC ingress controller

6. Create a file named `cic.yaml` in the launched terminal and then copy the content of the modified `citrix-k8s-ingress-controller.yaml` file to `cic.yaml`.

7. Deploy the `cic.yaml` file using the following command.

   ```
   kubectl create -f cic.yaml
   ```

8. Verify if the Citrix ingress controller is deployed successfully using the following command.

   ```
   kubectl get pods --all-namespaces
   ```

---

**Deploy the Citrix ingress controller on a PKS managed Kubernetes cluster**

February 3, 2022

**Pivotal Container Service (PKS)** enables operators to provision, operate, and manage enterprise-grade Kubernetes clusters using BOSH and Pivotal Ops Manager.

The **Citrix ingress controller** is built around the Kubernetes Ingress and it can automatically configure one or more Citrix ADCs based on the Ingress resource configuration. You can deploy the Citrix ingress controller in a PKS managed Kubernetes cluster to extend the advanced load balancing and traffic management capabilities of Citrix ADC to your cluster.

**Prerequisites**

Before creating the Kubernetes cluster using PKS. Make sure that for all the plans available on the Pivotal Ops Manager, the following options are set:

- Enable Privileged Containers
- Disable DenyEscalatingExec

For detailed information on PKS Framework and other documentation, see Pivotal Container Service documentation.

After you have set the required options, create a Kubernetes cluster using the PKS CLI framework and set the context for the created cluster.
**Deployment options**

You can either deploy Citrix ADC CPXs as pods inside the cluster or deploy a Citrix ADC MPX or VPX appliance outside the Kubernetes cluster.

Based on how you want to use Citrix ADC, there are two ways to deploy the Citrix ingress controller in a Kubernetes cluster on the PKS:

- As a sidecar container alongside Citrix ADC CPX in the same pod: In this mode, Citrix ingress controller configures the Citrix ADC CPX.
- As a standalone pod in the Kubernetes cluster: In this mode, you can control the Citrix ADC MPX or VPX appliance deployed outside the cluster.

**Deploy Citrix ingress controller as a pod**

Follow the instruction provided in topic: Deploy Citrix ingress controller as a standalone pod in the Kubernetes cluster for Citrix ADC MPX or VPX appliances.

**Deploy Citrix ingress controller as a sidecar with Citrix ADC CPX**

Follow the instruction provided in topic: Deploy Citrix ingress controller as a sidecar with Citrix ADC CPX.

**Network Configuration**

For seamless functioning of the services deployed in the Kubernetes cluster, it is essential that Ingress Citrix ADC device should be able to reach the underlying overlay network over which Pods are running. The Citrix ingress controller allows you to configure network connectivity between the Citrix ADC device and service using Static Routing, Citrix node controller, services of type NodePort, or services of type LoadBalancer.

**Exposé services of type LoadBalancer**

July 5, 2022

**Overview of services of type LoadBalancer**

In a Kubernetes environment, a microservice is deployed as a set of pods that are created and destroyed dynamically. Since the set of pods that refer to a microservice are constantly changing, Ku-
Citrix ADC ingress controller

Kubernetes provides a logical abstraction known as service to expose your microservice running on a set of pods. A service defines a logical set of pods, as well as policies to access them.

A service of type **LoadBalancer** is the simplest way to expose a microservice inside a Kubernetes cluster to the external world. Services of type LoadBalancer are natively supported in Kubernetes deployments on public clouds such as, AWS, GCP, or Azure. In cloud deployments, when you create a service of type LoadBalancer, a cloud managed load balancer is assigned to the service. The service is then exposed using the load balancer.

**Citrix solution for services of type LoadBalancer**

There may be several situations where you want to deploy your Kubernetes cluster on bare metal or on-premises rather than deploy it on public cloud. When you are running your applications on bare metal Kubernetes clusters, it is much easier to route TCP or UDP traffic using a service of type **LoadBalancer** than using Ingress. Even for HTTP traffic, it is sometimes more convenient than Ingress. However, there is no load balancer implementation natively available for bare metal Kubernetes clusters. Citrix provides a way to load balance such services using the Citrix ingress controller and Citrix ADC.

In the Citrix solution for services of type **LoadBalancer**, the Citrix ingress controller deployed inside the Kubernetes cluster configures a Citrix ADC deployed outside the cluster to load balance the incoming traffic. Using the Citrix solution, you can load balance the incoming traffic to the Kubernetes cluster regardless of whether the deployment is on bare metal, on-premises, or public cloud. Since the Citrix ingress controller provides flexible IP address management that enables multi-tenancy for Citrix ADCs, you can use a single Citrix ADC to load balance multiple services as well as to perform Ingress functions. Hence, you can maximize the utilization of load balancer resources and significantly reduce your operational expenses.

**Services of type LoadBalancer VS Kubernetes Ingress**

The following table summarizes a comparison between the Kubernetes Ingress and services of type LoadBalancer that helps you to choose the right option based on your requirements:

<table>
<thead>
<tr>
<th>Services of type LoadBalancer</th>
<th>Ingress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simpler and faster way to expose a service. You only need to specify the service type as <code>type=LoadBalancer</code> in the service definition.</td>
<td>Ingress provides advanced features but implementation requires more steps. You need to write an Ingress object in addition to the service definition. Also, the chances of making mistakes while defining the Ingress is more.</td>
</tr>
<tr>
<td>Needs a separate IP address for each service.</td>
<td>Provides a way to expose multiple services using a single IP address.</td>
</tr>
</tbody>
</table>
Services of type **LoadBalancer**

Forwards all kinds of traffic arriving on the specified port to the service regardless of it is HTTP, TCP, or UDP. There is no filtering or options to perform advanced routing.

**Ingress**

Feature rich and powerful compared to services of type LoadBalancer. Ingress provides multiple routing options. For example, using ingress you can perform path-based and sub domain-based routing to back-end services.

---

**How does the Citrix solution for services of type LoadBalancer work on bare-metal clusters**

By default, a service of type **LoadBalancer** simply exposes NodePorts for the service in a bare-metal Kubernetes cluster. It does not configure external load balancers.

Citrix offers an end-to-end solution for services of type **LoadBalancer** in a bare-metal Kubernetes cluster by providing both IP management and external load balancer configuration. With the Citrix solution, when a service of type **LoadBalancer** is created in the bare-metal cluster, the Citrix ingress controller configures the Citrix ADC outside the Kubernetes cluster (Tier-1) with a load balancing virtual server. The load balancing virtual server is configured with an IP address either automatically assigned by the Citrix IPAM controller or manually specified in the service definition using the `spec.loadBalancerIP` field. Once the IP address is configured for a service, you can use the configured IP address to access the service externally.

**IP address management using the IPAM controller**

The IPAM controller is a container provided by Citrix for IP address management and it runs in parallel to the Citrix ingress controller a pod in the Kubernetes cluster. For services of type **LoadBalancer**, you can use the IPAM controller to automatically allocate IP addresses to services from a specified IP address range. You can specify this IP range in the YAML file while deploying the IPAM controller using YAML. The Citrix ingress controller configures the IP address allocated to the service as a virtual IP address (VIP) in Citrix ADC MPX or VPX.

Using this IP address, you can externally access the service.

The IPAM controller requires the VIP **CustomResourceDefinition** (CRD) provided by Citrix. The VIP CRD contains fields for service-name, namespace, and IP address. The VIP CRD is used for internal communication between the Citrix ingress controller and the IPAM controller.

The following diagram shows a deployment of service type load balancer where the IPAM controller is used to assign an IP address to a service.
When a new service of type `LoadBalancer` is created, the following events occur:

1. The Citrix ingress controller creates a VIP CRD object for the service whenever the `loadBalancerIP` field in the service is empty.
2. The IPAM controller assigns an IP address for the VIP CRD object.
3. Once the VIP CRD object is updated with the IP address, the Citrix ingress controller automatically configures the Citrix ADC.

**Note:**

Custom resource definitions (CRDs) offered by Citrix also supports services of type `LoadBalancer`. That means, you can specify a service of type `LoadBalancer` as a service name when you create a CRD object and apply the CRD to the service.

The IPAM controller solution is designed in such a way that you can easily integrate the solution with ExternalDNS providers such as Infoblox. For more information on ExternalDNS, see Interoperability with ExternalDNS.
**Exposé services of type LoadBalancer with IP addresses assigned by the IPAM controller**

This topic provides information on how to expose services of type LoadBalancer with IP addresses assigned by the IPAM controller.

To expose a service of type load balancer with an IP address from the IPAM controller, perform the following steps:

1. Deploy the VIP CRD
2. Deploy the Citrix ingress controller
3. Deploy the IPAM controller.
4. Deploy a sample application.
5. Create a service of type `LoadBalancer` to expose the application.
6. Access the service.

**Step 1: Deploy the VIP CRD**

Perform the following step to deploy the Citrix VIP CRD which enables communication between the Citrix ingress controller and the IPAM controller.

```
kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/crd/vip/vip.yaml
```

For more information on VIP CRD see, the [VIP CustomResourceDefinition](#).

**Step 2: Deploy the Citrix ingress controller**

Perform the following steps to deploy the Citrix ingress controller with the IPAM controller argument.

1. Download the `citrix-k8s-ingress-controller.yaml` using the following command:

   ```
   wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/baremetal/citrix-k8s-ingress-controller.yaml
   ```

2. Edit the Citrix ingress controller YAML file:
   - Specify the values of the environment variables as per your requirements. For more information on specifying the environment variables, see the [Deploy Citrix ingress controller](#).
Citrix ADC ingress controller

- Specify the IPAM controller as an argument using the following:

1. `args:
   2.  --ipam
   3.  citrix-ipam-controller`

Here is a snippet of a sample Citrix ingress controller YAML file with the IPAM controller argument:

```
apiVersion: v1
kind: Pod
metadata:
  name: cic-k8s-ingress-controller
  labels:
    app: cic-k8s-ingress-controller
spec:
  serviceAccountName: cic-k8s-role
  containers:
  - name: cic-k8s-ingress-controller
    image: "quay.io/citrix/citrix-k8s-ingress-controller:1.26.7"
    env:
      # Set Citrix ADC NSIP/SNIP, SNIP in case of HA (mgmt has to be enabled)
      - name: "NS_IP"
        value: "x.x.x.x"
      # Set the username
      - name: "NS_USER"
        valueFrom:
          secretKeyRef:
            name: nslogin
            key: username
      # Set user password
      - name: "NS_PASSWORD"
        valueFrom:
          secretKeyRef:
            name: nslogin
            key: password
      # Set log level
```

Note:
This YAML is for demonstration purpose only and not the full version. Always, use the latest version of the YAML and edit as per your requirements.
Deploy the Citrix ingress controller using the edited YAML file with the following command:

```
kubectl create -f citrix-k8s-ingress-controller.yaml
```

For more information on how to deploy the Citrix ingress controller, see the Deploy Citrix ingress controller.

**Step3: Deploy the IPAM controller**

Perform the following steps to deploy the IPAM controller.

1. Download the `citrix-ipam-controller.yaml` file.

   The manifest contains two environment variables, `VIP_RANGE` and `VIP_NAMESPACES`. You can specify the appropriate routable IP range with a valid CIDR under the `VIP_RANGE`. If necessary, you can also specify a set of namespaces under `VIP_NAMESPACES` so that the IPAM controller allocates addresses only for services from specific namespaces.

   For more information, see `VIP_RANGE` and `VIP_NAMESPACES`.

2. Deploy the IPAM controller using the following command:

   ```
kubectl create -f citrix-ipam-controller.yaml
```

**Step 4: Deploy a sample application**

Perform the following to deploy an `apache` application in your Kubernetes cluster.

**Note:**

In this example, an `apache` application is used. You can deploy a sample application of your
1. Create a file named `apache-deployment.yaml` with the following configuration:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache
labels:
  name: apache
spec:
  selector:
    matchLabels:
      app: apache
  replicas: 8
  template:
    metadata:
      labels:
        app: apache
    spec:
      containers:
        - name: apache
          image: httpd:latest
          ports:
            - name: http
              containerPort: 80
          imagePullPolicy: IfNotPresent
```

2. Deploy the `apache` application using the following command:

```
kubectl create -f apache-deployment.yaml
```

3. Verify if the pods are running using the following:

```
kubectl get pods
```

Output:

```
NAME                       READY STATUS     RESTARTS AGE
apache-7db8f797c7-2x6jc   1/1    Running   0       8s
```
### Step 5: Expose the sample application using service of type LoadBalancer

Perform the following to create a service (`apache`) of type `LoadBalancer`.

1. Create a file named `apache-service.yaml` with the following configuration:

```yaml
apiVersion: v1
group: v1
kind: Service
metadata:
  name: apache
  labels:
    name: apache
spec:
  externalTrafficPolicy: Local
  type: LoadBalancer
  ports:
  - name: http
    port: 80
    targetPort: http
  selector:
    app: apache
```

2. Deploy the service using the following command:

```
kubectl create -f apache-service.yaml
```

When you create the service, the IPAM controller assigns an IP address to the `apache` service from the IP address range you had defined in the IPAM controller deployment. The IP address allocated by the IPAM controller is provided in the `status.loadBalancer.ingress:` field of the service definition. The Citrix ingress controller configures the IP address allocated to the service as a virtual IP (VIP) in the Citrix ADC.

3. View the service using the following command:
Step 6: Access the service

You can access the apache service using the IP address assigned by the IPAM controller to the service. You can find the IP address in the `status.loadBalancer.ingress:` field of the service definition. Use the `curl` command to access the service:

```
1 curl <IP_address>
```
Exposing services of type LoadBalancer by specifying an IP address

You can also expose a service of type LoadBalancer manually by specifying an IP address in your service definition.

To expose a service of type LoadBalancer manually, you can specify the IP address in the service definition YAML file as follows.

```yaml
spec:
  type: LoadBalancer
  loadBalancerIP: "<ip-address>"
```

When you create a service of type LoadBalancer, the Citrix ingress controller configures the IP address you have defined in the `spec.loadBalancerIP` field as a virtual IP (VIP) address in Citrix ADC.

Example: Expose an Apache application using service of type LoadBalancer by specifying an IP address

Perform the following:

1. Create a file named `apache-deployment.yaml` with the following configuration:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache
labels:
  name: apache
spec:
  selector:
    matchLabels:
      app: apache
  replicas: 8
template:
  metadata:
    labels:
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
2. Deploy the *apache* application using the following command:

```
kubectl create -f apache-deployment.yaml
```

3. Verify if the pods are running using the following:

```
kubectl get pods
```

Output:

```
<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache-7db8f797c7-2x6jc</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8s</td>
</tr>
<tr>
<td>apache-7db8f797c7-cdgmw</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8s</td>
</tr>
<tr>
<td>apache-7db8f797c7-lh447</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8s</td>
</tr>
<tr>
<td>apache-7db8f797c7-m7mhd</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8s</td>
</tr>
<tr>
<td>apache-7db8f797c7-m9rn7</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8s</td>
</tr>
<tr>
<td>apache-7db8f797c7-r9jgz</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8s</td>
</tr>
<tr>
<td>apache-7db8f797c7-vwhc8</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8s</td>
</tr>
<tr>
<td>apache-7db8f797c7-zslwv</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8s</td>
</tr>
</tbody>
</table>
```

4. Create a service (*apache*) of type *LoadBalancer*. Create a file name *apache-service.yaml* with the following configuration:

```
apiVersion: v1
class: Service
metadata:
  name: apache
type: LoadBalancer
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
Citrix ADC ingress controller

```yaml
spec:
  externalTrafficPolicy: Local
  loadBalancerIP: "10.217.212.16"
  type: LoadBalancer
  ports:
    - name: http
      port: 80
      targetPort: http
  selector:
    app: apache
```

5. Deploy the service using the following command:

```
kubectl create -f apache-service.yaml
```

When you create the service (`apache`), the Citrix ingress controller configures 192.217.212.16 as a virtual IP address (VIP) in Citrix ADC VPX.

6. Access the `apache` service using the IP address (192.217.212.16) that you had assigned to the service. Use the `curl` command to access the service:

```
curl 192.217.212.16
```

The response should be:

```
<html><body><h1>It works!</h1></body></html>
```

Example use case: Expose microservices using services of type LoadBalancer in a Citrix ADC dual-tier deployment

This example shows how to expose microservices deployed in Kubernetes to clients outside the cluster using services of type LoadBalancer in a Citrix ADC dual-tier deployment.

You can deploy Citrix ADC VPX, MPX, or CPX as a load balancer in Tier-1 to manage high scale North-South traffic to the microservices. In Tier-2, you can deploy Citrix ADC CPX as an intelligent L7 microservices router for North-South and East-West traffic. In this example, a Citrix ADC VPX (service of type `LoadBalancer`) is used in Tier-1 and a Citrix ADC CPX (Ingress) is used in Tier-2.

The following diagram depicts the microservice deployment in this example. The deployment contains three services that are highlighted in blue, red, and green colors respectively. The deployment
contains 12 pods running across two worker nodes. These deployments are logically categorized using Kubernetes namespaces.

Prerequisites

Ensure that you have:

- Deployed a Kubernetes cluster. For more information, see Kubernetes Documentation.
- Set up the Kubernetes dashboard for deploying containerized applications. For more information, see https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/.
- The route configuration present in the Tier-1 Citrix ADC so that the Ingress Citrix ADC is able to reach the Kubernetes pod network for seamless connectivity. For detailed instructions, see manually configure a route on the Citrix ADC instance.

Deploy microservices using Kubernetes service of type LoadBalancer solution

1. Clone the GitHub repository to your master node using the following command:

   ```bash
   git clone https://github.com/citrix/example-cpx-vpx-for-kubernetes-2-tier-microservices.git
   ```

2. Using the CLI console of the master node, create namespaces using the following command:

   ```bash
   kubectl create -f namespace.yaml
   ```
Verify if the namespaces are created in your Kubernetes cluster using the following command:

```bash
kubectl get namespaces
```

The output of the command should be:

```
NAME     STATUS  AGE
default  Active 7d23h
kube-public Active 7d23h
kube-system Active 7d23h
monitoring Active 17h
team-coldrink Active 17h
team-guestbook Active 17h
team-hotdrink Active 17h
tier-2-adc  Active 17h
```

3. From the Kubernetes dashboard, deploy the `rbac.yaml` in the default namespace using the following command:

```bash
kubectl create -f rbac.yaml
```

4. Deploy the VIP CRD and IPAM controller for automatically assigning IP addresses to the Kubernetes services. Use the following command:

```bash
kubectl create -f vip.yaml
2 kubectl create -f ipam_deploy.yaml
```

5. Deploy the Citrix ADC CPX for `hotdrink`, `coldrink`, and `guestbook` microservices using the following commands:

```bash
1 kubectl create -f cpx.yaml -n tier-2-adc
2 kubectl create -f hotdrink-secret.yaml -n tier-2-adc
```

6. Deploy three types of `hotdrink` beverage microservices using the following commands:

```bash
1 kubectl create -f team_hotdrink.yaml -n team-hotdrink
2 kubectl create -f hotdrink-secret.yaml -n team-hotdrink
```
7. Deploy the `coldrink` beverage microservice using the following commands:

1. `kubectl create -f team_colddrink.yaml -n team-coldrink`
2. `kubectl create -f colddrink-secret.yaml -n team-coldrink`

8. Deploy the `guestbook` microservice using the following commands:

1. `kubectl create -f team_guestbook.yaml -n team-guestbook`

9. Log on to the Tier-1 Citrix ADC to verify that configuration is not pushed from the Citrix ingress controller before automating the Tier-1 Citrix ADC.

10. Deploy the Citrix ingress controller to push the Citrix ADC CPX configuration to the Tier-1 Citrix ADC automatically. In the `cic_vpx.yaml`, change the value of the `NS_IP` environment variable with the NS IP of your Citrix ADC VPX. For more information on the Citrix ingress controller deployment, see Deploy the Citrix ingress controller using YAML.

    After you update the `cic_vpx.yaml` file, deploy the file using the following command:

1. `kubectl create -f cic_vpx.yaml -n tier-2-adc`

11. Verify if the IPAM controller has assigned IP addresses to Citrix ADC CPX services using the following command:

    `kubectl get svc -n tier-2-adc`

12. Add the following DNS entries in your local machine host files to access the microservices using the Internet:

1. `<frontend-ip from ingress_vpx.yaml> hotdrink.beverages.com`
2. `<frontend-ip from ingress_vpx.yaml> colddrink.beverages.com`
3. `<frontend-ip from ingress_vpx.yaml> guestbook.beverages.com`

You can now access the microservices using the following URL: `https://hotdrink.beverages.com`
Environment variables: IPAM controller

This section provides information about the environment variables in the IPAM controller.

VIP_RANGE

The VIP_RANGE environment variable allows you to define the IP address range. You can either define an IP address range or an IP address range associated with a unique name.

IP address range

You can define the IP address range from a subnet or multiple subnets. Also, you can use the – character to define the IP address range. The IPAM controller assigns the IP address from this IP address range to the service.

The following examples demonstrate the various ways you can define the IP address range in the VIP_RANGE environment variable:

```bash
To define the IP address range from a subnet:
  - name: "VIP_RANGE"
  value: '10.xxx.xxx.18/31'

To define the IP address range from multiple subnets, ensure that the values are valid CIDRs for the subnets:
```
IP address range associated with a unique name

You can assign a unique name to the IP address range and define the range in the VIP_RANGE environment variable. This way of assigning the IP address range enables you to differentiate between the IP address ranges. When you create the services of type LoadBalancer you can use the service. citrix.com/ipam-range annotation in the service definition to specify the IP address range to use for IP address allocation.

For example, there are three domains namely, Dev, Test, and Prod that have dedicated workloads to manage. If each team wants a separate range of IP addresses to load balance the microservice traffic, you can assign unique names to the IP address ranges. Then, you can define the names in the service.citrix.com/ipam-range annotation in your service definition. The service defined with service.citrix.com/ipam-range = 'Dev' is allocated with an IP address from the IP address range associated with Dev.

The following examples demonstrate the various ways you can define the IP address range associated with a unique name in the VIP_RANGE environment variable:

```
- name: "VIP_RANGE"
  value: '["10.217.212.18/31", "10.217.212.20/31",
           "10.217.212.16/30", "10.217.212.0/24"]'
```

Also, you can use the - character to define the IP address range:

```
- name: "VIP_RANGE"
```
The following is a sample service definition for demonstrating the usage of the `service.citrix.com/ipam-range` annotation. In this example, the annotation is used to allocate an IP address from the IP address range associated with a unique name `Dev` to the service.

```yaml
apiVersion: v1
kind: Service
metadata:
  annotations:
    service.citrix.com/ipam-range: "Dev"
  name: apache
labels:
  name: apache
spec:
  externalTrafficPolicy: Local
type: LoadBalancer
selector:
  name: apache
ports:
  - name: http
    port: 80
  targetPort: http
selector:
  app: apache
```

**VIP_NAMESPACES**

The `VIP_NAMESPACES` environment variable enables you to define the IPAM controller to work only for a set of namespaces. The IPAM controller allocates IP addresses only to services created from namespaces specified in the environment variable.

The following example demonstrates how you can specify namespaces in the `VIP_NAMESPACES` environment variable:
The IPAM controller allocates IP addresses to services created from `default` and `kube-system` namespaces.

**Note:**
If you do not use the `VIP_NAMESPACES` environment variable or do not set a value, then the IPAM controller allocates IP addresses to services created from all namespaces.

---

## Deploy Citrix ADC-Integrated Canary Deployment Solution

July 6, 2022

Canary release is a technique to reduce the risk of introducing a new software version in production by first rolling out the change to a small subset of users. After the user validation, the application is rolled out to the larger set of users.

Citrix provides the following options for canary deployment using the Citrix ingress controller.

- Deploy canary using the Canary CRD
- Deploy canary using Ingress annotations

In a deployment using the Canary CRD, canary configuration is applied using a Kubernetes CRD. Citrix also supports a much simpler option for canary deployment using Ingress annotations.

### Deploy canary using the Canary CRD

This section provides information about how to perform Canary deployment using the Canary CRD.

Citrix ADC-Integrated Canary Deployment solution stitches together all components of continuous delivery (CD) and makes canary deployment easier for the application developers. This solution uses Spinnaker as the continuous delivery platform and Kayenta as the Spinnaker plug-in for canary analysis. Kayenta is an open-source canary analysis service that fetches user-configured metrics from their sources, runs statistical tests, and provides an aggregate score for the canary. The score from statistical tests and counters along with the success criteria is used to promote or fail the canary.

Citrix ADC comes with a rich application-centric configuration module and provides complete visibility to application traffic and health of application instances. The capabilities of Citrix ADC to generate accurate performance statistics can be leveraged for Canary analysis to take better decisions about the Canary deployment. In this solution, Citrix ADC is integrated with the Spinnaker platform and acts as a source for providing accurate metrics for analyzing Canary deployment using Kayenta.
**Citrix ADC ingress controller** exports the application performance metrics to the open-source monitoring system Prometheus and you can configure Kayenta to fetch the metrics for canary deployment. Traffic distribution to the canary version can be regulated using the Citrix ADC policy infrastructure. If you want to divert a specific kind of traffic from production to baseline and canary, you can use match expressions to redirect traffic to baseline and canary leveraging the rich Citrix ADC policy infrastructure.

For example, you can divert traffic from production to canary and baseline using the match expression `HTTP.REQ.URL.CONTAINS("citrix india")`. The traffic which matches the expression is diverted to canary and baseline and the remaining traffic goes to production.

The components which are part of the Citrix-Integrated Canary Deployment Solution and their functionalities are explained as follows:

- **GitHub**: GitHub offers all the distributed version control and source code management functionalities provided by Git and has extra features. GitHub has many utilities available for integrating with other tools that form part of your CI/CD pipeline like Docker Hub and Spinnaker.

- **Docker Hub**: Docker Hub is a cloud-based repository service provided by Docker for sharing and finding Docker images. You can integrate GitHub with Docker Hub to automatically build images from the source code in GitHub and push the built image to Docker Hub.

- **Spinnaker**: Spinnaker is an open source, multi-cloud continuous delivery platform for releasing software changes with high velocity and reliance. You can use Spinnaker’s application deployment features to construct and manage continuous delivery workflows. The key deployment management construct in Spinnaker is known as a pipeline. Pipelines in Spinnaker consist of a sequence of actions, known as stages. Spinnaker provides various stages for deploying an application, running a script, performing canary analysis, removing the deployment, and so on. You can integrate Spinnaker with many third-party tools to support many extra functionalities.

- **Prometheus**: Prometheus is an open-source systems monitoring and alerting toolkit. Prometheus is a monitoring system which can maintain a huge amount of data in a time series database. Citrix ADC Metrics exposes the performance metrics to Spinnaker through Prometheus.

- **Jenkins**: Jenkins is an open source automation server which helps to automate all sorts of tasks related to building, testing, and delivering or deploying software. Jenkins also supports running custom scripts as part of your deployment cycle.

- **Citrix ingress controller**: Citrix provides an Ingress Controller for Citrix ADC MPX (hardware), Citrix ADC VPX (virtualized), and Citrix ADC CPX (containerized) for bare metal and cloud deployments. The Citrix ingress controller is built around Kubernetes Ingress and automatically configures one or more Citrix ADCs based on the Ingress resource configuration.

Following Citrix software versions are required for Citrix-Integrated Canary Deployment Solution:
Citrix ADC ingress controller

- Citrix ADC CPX version: quay.io/citrix/citrix-k8s-cpx-ingress:13.0-83.27.
- Citrix ADC Metrics Exporter version: quay.io/citrix/netscaler-metrics-exporter:1.4.0.

Workflow of a Spinnaker pipeline for Citrix ADC-Integrated Canary Deployment Solution

The following diagram explains the workflow of a Spinnaker pipeline for Citrix ADC-Integrated Canary Deployment Solution.

The following steps explain the workflow specified in the diagram.

1. Developers maintain the source code in GitHub, make changes whenever required, and commit the changes to GitHub.

2. A webhook is configured in GitHub to listen for the source code changes. Whenever the source code is checked in to GitHub, the webhook is triggered and informs Docker Hub to build the image with the new source code. Once the docker image is created, a separate webhook configured in Docker Hub triggers a Spinnaker pipeline.

3. Once the Spinnaker pipeline is triggered, canary and baseline versions of the image are deployed.

4. Once the canary and baseline versions are deployed, some percentage of traffic from production is diverted to the canary and baseline versions. Citrix ADC collects the performance statistics and exports the statistics to Prometheus with the help of Citrix ADC Metrics Exporter. Prometheus feeds these statistics to Kayenta for canary analysis.
5. Kayenta performs a canary analysis based on the performance statistics and generates a score. Based on the score, the canary deployment is termed as success or failure and the image is rolled out or rolled back.

**Deploy the Citrix ADC-Integrated Canary Deployment Solution in Google Cloud Platform**

This section contains information on setting up Spinnaker, how to create a Spinnaker pipeline, and a sample canary deployment.

**Deploy Spinnaker in Google Cloud Platform**

This topic contains information about deploying Spinnaker and how to integrate plug-ins with Spinnaker for canary deployment on Google Cloud Platform (GCP).

Perform the following steps to deploy Spinnaker and integrate plug-ins in GCP.

1. Set up the environment and create a GKE cluster using the following commands.

   ```sh
   export GOOGLE_CLOUD_PROJECT=[PROJECT_ID]
   gcloud config set project $GOOGLE_CLOUD_PROJECT
   gcloud config set compute/zone us-central1-f
   gcloud services enable container.googleapis.com
   gcloud beta container clusters create kayenta-tutorial
   --machine-type=n1-standard-2 --enable-stackdriver-kubernetes
   ```

2. Install the plug-in for integrating Prometheus with Stackdriver using the following command.

   ```sh
   kubectl apply --as=admin --as-group=system:masters -f \
   https://storage.googleapis.com/stackdriver-prometheus-\n   documentation/rbac-setup.yaml
   curl -sS "https://storage.googleapis.com/stackdriver-prometheus-\n   documentation/prometheus-service.yaml" | \n   sed "s/_stackdriver_project_id:.*/_stackdriver_project_id: \n   $GOOGLE_CLOUD_PROJECT/" | \n   sed "s/_kubernetes_cluster_name:.*/_kubernetes_cluster_name: \n   kayenta-tutorial/" | \n   sed "s/_kubernetes_location:.*/_kubernetes_location: us-central1-\n   f/" | \n   kubectl apply -f -
   ```

3. Deploy Spinnaker in the GKE cluster using the following steps.
a) Download the quick-install.yml file for Spinnaker from Spinnaker website.

b) Update the quick-install.yml file to integrate different components starting with Docker Hub. To integrate Spinnaker with Docker Hub, update the values of address, user name, password, email, and repository under ConfigMap in quick-install.yml file.

c) (Optional) Perform the following steps to set up Jenkins.

```bash
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install openjdk-8-jdk
wget -q -O - https://jenkins-ci.org/debian/jenkins-ci.org.key | sudo apt-key add -
sudo sh -c 'echo deb http://pkg.jenkins-ci.org/debian binary/ > /etc/apt/sources.list.d/jenkins.list'
sudo apt-get update
sudo apt-get install jenkins git
sudo apt-get install software-properties-common python-
```
Note:
If Jenkins is installed in one of the nodes of Kubernetes, you must update the firewall rules for that node for public access.

d) Update the following values in the quick-install.yml file for integrating Jenkins with Spinnaker.

```yaml
data:igor.yml: |
  enabled: true
  skipLifeCycleManagement: false
ci:jenkins:
  enabled: true
  masters:
  - name: master
    address: <endpoint>
    username: <username>
    password: <password>
```

e) To set up Prometheus and Grafana, see the Prometheus and Grafana Integration section in Citrix ADC Metrics Exporter and perform the steps.

f) To integrate Prometheus with Spinnaker, update the following values in the quick-install.yml file.

```yaml
data:
  config: |
    deploymentConfigurations:
    canary:
    enabled: true
    serviceIntegrations:
    - name: prometheus
      enabled: true
      accounts:
      - name: my-prometheus
        endpoint:
        baseUrl: prometheus-endpoint
        supportedTypes:
        - METRICS_STORE
```
g) To integrate Slack for notification with Spinnaker, update the following values in the quick-install.yml file.

```yaml
1 data: 2   config: | 3     deploymentConfigurations: 4       metricStores: 5       prometheus: 6         enabled: true 7         add_source_metalabels: true 8       stackdriver: 9         enabled: true 10      period: 30 11      enabled: true
```

h) Once all the required components are integrated, deploy Spinnaker by performing the following step.

```
kubectl apply -f quick-install.yaml
```

i) Verify the progress of the deployment using the following command. Once the deployment is complete, this command outputs all the pods as Ready x/x.

```
watch kubectl -n spinnaker get pods
```

4. Once you deploy Spinnaker, you can test the deployment using the following steps:

a) Enable Spinnaker access by forwarding a local port to the deck component of Spinnaker using the following command:
b) To access Spinnaker, in the Cloud Shell, click the Web Preview icon and select Preview on port 8080.

Note:
You can access Spinnaker securely or via HTTP. To expose Spinnaker securely, use the spin-ingress-ssl.yaml file to deploy the Ingress. Once the Spinnaker application is publicly exposed, you can use the domain assigned for Spinnaker or the IP address of the Ingress to access it.

Create a Spinnaker pipeline and configure automated canary deployment

Once you deploy Spinnaker, create a Spinnaker pipeline for an application and configure the automated canary deployment.

1. Create an application in Spinnaker.
2. Create a Spinnaker pipeline. You can edit the pipeline as a JSON file using the sample file provided in Sample JSON files.
3. Create an automated canary configuration in Spinnaker for automated canary analysis. You can use the configuration provided in the JSON file as a sample for automated canary configuration Sample JSON files.

Deploy a sample application for canary

This example shows how to run the canary deployment of a sample application using Citrix ADC-Integrated Canary Deployment Solution. In this example, Citrix ADC CPX, MPX, or VPX is deployed as an Ingress device for a GKE cluster. Citrix ADC generates the performance metrics required for canary analysis.

As a prerequisite, you must complete the following step before deploying the sample application.

• Install Spinnaker and the required plug-ins in Google cloud platform using Deploy Spinnaker in Google Cloud Platform.
Deploy the sample application

Perform the following steps to deploy a sample application as a canary release.

1. Create the necessary RBAC rules for Citrix ADC by deploying the `rbac.yaml` file.

   ```bash
   kubectl apply -f rbac.yaml
   ```

2. You can either deploy the Citrix ingress controller as a sidecar with Citrix ADC CPX or as a standalone pod which controls Citrix ADC VPX or MPX.

   Use the `cpx-with-cic-sidecar.yaml` file to deploy the Citrix ingress controller as a sidecar with Citrix ADC CPX. It also deploys Citrix ADC Metrics Exporter on the same pod.

   ```bash
   kubectl apply -f cpx-with-cic-sidecar.yaml
   ```

   To deploy the Citrix ingress controller as a stand-alone pod for Citrix ADC VPX or MPX use the `cic-vpx.yaml` file. In this deployment, you should use the `exporter.yaml` file to deploy Citrix ADC Metrics Exporter.

   ```bash
   kubectl apply -f cic-vpx.yaml
   kubectl apply -f exporter.yaml
   ```

   **Note:**
   Depending on how you are deploying the Citrix ingress controller, you must edit the YAML file for Citrix ingress controller deployment and modify values for the environmental variables as provided in deploying Citrix ingress controller.

3. Deploy the Ingress for securely exposing Spinnaker using the `spin-ingress-ssl.yaml` file.

   ```bash
   kubectl apply -f spin-ingress-ssl.yaml
   ```

   **Note:**
   For more information on creating a TLS certificate for Ingress, see TLS certificates in Citrix Ingress Controller.

4. Once Spinnaker is exposed using Citrix ADC, access Spinnaker and perform the steps in Create a Spinnaker pipeline and configure automated canary deployment if the steps are not already done.
5. Deploy the production version of the application using the `production.yaml` file.

```
kubectl apply -f production.yaml
```

6. Create the Ingress resource rule to expose traffic from outside the cluster to services inside the cluster using the `ingress.yaml` file.

```
kubectl apply -f ingress.yaml
```

7. Create a Kubernetes service for the application that needs canary deployment using the `service.yaml` file.

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

8. Deploy the canary CRD that defines the canary configuration using the `canary-crd-class.yaml` file.

```
kubectl apply -f canary-crd-class.yaml
```

Note:
Once you create the CRD, wait for 10 seconds before you apply the CRD object.

9. Create a CRD object `canary-crd-object.yaml` based on the canary CRD for customizing the canary configuration.

```
kubectl apply -f canary-crd-object.yaml
```

The following table explains the fields in the canary CRD object.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>serviceNames</td>
<td>List of services on which this CRD has to be applied</td>
</tr>
<tr>
<td>deployment</td>
<td>Specifies the deployment strategy as Kayenta.</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>percentage</td>
<td>Specifies the percentage of traffic to be diverted from production to baseline and canary.</td>
</tr>
<tr>
<td>matchExpression (optional)</td>
<td>Any Citrix ADC supported policy that can be used to define the subset of users to be directed to canary and baseline versions. If x percentage of traffic is configured, then from within subset of users which matches the matchExpression only x percentage of users are diverted to baseline and canary. Remaining users are diverted to production.</td>
</tr>
</tbody>
</table>

**Spinnaker**

<table>
<thead>
<tr>
<th>Spinnaker</th>
<th>Specifies the Spinnaker pipeline configurations you want to apply for your services.</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain</td>
<td>IP address or domain name of the Spinnaker gate.</td>
</tr>
<tr>
<td>port</td>
<td>Port number of the Spinnaker gate.</td>
</tr>
<tr>
<td>applicationName</td>
<td>The name of the application in Spinnaker.</td>
</tr>
<tr>
<td>pipelineName</td>
<td>The name of the pipeline under the Spinnaker application.</td>
</tr>
<tr>
<td>serviceName</td>
<td>Specifies the name of the service to which you want to apply the Spinnaker configuration.</td>
</tr>
</tbody>
</table>

10. Deploy canary and baseline versions of the application.

**Note:**

If you are fully automating the canary deployment, deploy canary and baseline versions using the **Deploy (Manifest) stage** in Spinnaker pipeline and there is no need to perform this step.

For manually deploying canary and baseline versions, use **canary.yaml** and **baseline.yaml** files.

```bash
1  kubectl apply -f canary.yaml
2  kubectl apply -f baseline.yaml
```
Citrix ADC ingress controller

Troubleshooting

For troubleshooting the deployment, perform the following steps.

1. Check the pod logs for the respective components like Spinnaker, Prometheus, Kayenta, Citrix ADC CPX, Citrix ADC Metrics Exporter, Citrix ingress controller.
2. Check the pod logs of the Citrix ingress controller for any configuration-related errors while configuring the Citrix proxy.
3. Search for the exception/Exception keyword in the Citrix ingress controller pod logs to narrow down the issues.
4. Check for the logs preceding the search. Check for the configuration that failed and caused the issue.
5. Check for the reason of failures during configuration.
6. If the failure happened because of incorrect configuration, correct the configuration.

Sample JSON files

This topic contains sample JSON files for Spinnaker pipeline configuration and automated canary configuration. These files can be used as a reference while creating Spinnaker pipeline and automated canary configuration.

A sample JSON file for Spinnaker pipeline configuration**

```json
{
  "appConfig": {
  },
  "description": "This pipeline deploys a canary version of the application, and a baseline (identical to production) version. It compares them, and if the canary is OK, it triggers the production deployment pipeline.",
  "executionEngine": "v2",
  "expectedArtifacts": [
    {
      "defaultArtifact": {
        "kind": "custom"
      },
      "id": "ac842617-988f-48dc-a7a4-7f020d93cc42",
```
"matchArtifact": {
  "kind": "docker",
  "name": "index.docker.io/sample/demo",
  "type": "docker/image"
},
"useDefaultArtifact": false,
"usePriorExecution": false
],
"keepWaitingPipelines": false,
"lastModifiedBy": "anonymous",
"limitConcurrent": true,
"parallel": true,
"parameterConfig": [],
"stages": [
  {
    "account": "my-kubernetes-account",
    "cloudProvider": "kubernetes",
    "kinds": [
      "Deployment",
      "ConfigMap"
    ],
    "labelSelectors": {
      "selectors": [
        {
          "key": "version",
          "kind": "EQUALS",
          "values": [
            "canary"
          ]
        }
      ]
    },
    "location": "default",
    "name": "Delete Canary",
    "options": {
      "canary": "true",
      "dryRun": false
    }
  }
]
```json
{
  "cascading": true,
  "refId": "12",
  "requisiteStageRefIds": [
    "19",
    "26"
  ],
  "type": "deleteManifest"
},
{
  "account": "my-kubernetes-account",
  "cloudProvider": "kubernetes",
  "kinds": [
    "Deployment"
  ],
  "labelSelectors": {
    "selectors": [
      {
        "key": "version",
        "kind": "EQUALS",
        "values": ["baseline"]
      }
    ],
    "location": "default",
    "name": "Delete Baseline",
    "options": {
      "cascading": true
    },
    "refId": "13",
    "requisiteStageRefIds": [
      "19",
      "26"
    ],
```
"type": "deleteManifest"

"name": "Successful deployment",
"preconditions": [],
"refId": "14",
"requisiteStageRefIds": [
  "12",
  "13"
],
"type": "checkPreconditions"

"application": "sampleapplication",
"expectedArtifacts": [
  {
    "defaultArtifact": {
      "kind": "custom"
    },
    "id": "9185c756-c6cd-49bc-bее-е3f7118f3412",
    "matchArtifact": {
      "kind": "docker",
      "name": "index.docker.io/sample/demo",
      "type": "docker/image"
    },
    "useDefaultArtifact": false,
    "usePriorExecution": false
  },
  "failPipeline": true,
  "name": "Deploy to Production",
  "pipeline": "7048e5ac-246-4557-a05a-bece8bdf868fc",
  "refId": "19",
  "requisiteStageRefIds": [
    "25"
Citrix ADC ingress controller

```
],
  "stageEnabled": {
    "expression": "\$\{stage('Canary Analysis')['status'].toString() == 'SUCCEEDED' \}"
  },
  "type": "expression"
},
  "type": "pipeline",
  "waitForCompletion": true
},
{
  "account": "my-kubernetes-account",
  "cloudProvider": "kubernetes",
  "manifestArtifactAccount": "embedded-artifact",
  "manifests": [
    {
      "apiVersion": "apps/v1",
      "kind": "Deployment",
      "metadata": {
        "labels": {
          "name": "sampleapplication-prod",
          "version": "baseline"
        },
        "name": "sampleapplication-baseline-deployment",
        "namespace": "default"
      },
      "spec": {
        "replicas": 4,
        "strategy": {
          "rollingUpdate": {
            "maxSurge": 10,
            "maxUnavailable": 10
          }
        }
      }
    }
  ]
}© 1999–2022 Citrix Systems, Inc. All rights reserved. 72
"type": "RollingUpdate"

"template": {

"metadata": {

"labels": {

"name": "sampleapplication-prod"
}

}

"spec": {

"containers": [

{

"image": "index.docker.io/sample/demo:v1",
"imagePullPolicy": "Always",
"name": "sampleapplication-prod",
"ports": [

{

"containerPort": 8080,
"name": "port-8080"
}

]

}

"

}

}

"moniker": {

},

© 1999–2022 Citrix Systems, Inc. All rights reserved.


```json
"app": "sampleapplication"

,

"name": "Deploy Baseline",
"refId": "20",
"relationships": {

  "loadBalancers": [],
  "securityGroups": []

},

"requisiteStageRefIds": [],
"source": "text",
"type": "deployManifest"

,

{

  "account": "my-kubernetes-account",
  "cloudProvider": "kubernetes",
  "manifestArtifactAccount": "embedded-artifact",
  "manifests": [

    {

      "apiVersion": "apps/v1",
      "kind": "Deployment",
      "metadata": {

        "labels": {

          "name": "sampleapplication-prod",
          "version": "canary"

        }

      },

      "name": "sampleapplication-canary-deployment",
      "namespace": "default"

    },

    "spec": {

      "replicas": 4,
      "strategy": {

        "rollingUpdate": {

        }

    }

}
"maxSurge": 10,
"maxUnavailable": 10
}

"type": "RollingUpdate"
}

"template": {

"metadata": {

"labels": {

"name": "sampleapplication-prod"
}

}

"spec": {

"containers": [

{

"image": "index.docker.io/sample/demo",
"imagePullPolicy": "Always",
"name": "sampleapplication-prod",
"ports": [

{

"containerPort": 8080,
"name": "port-8080"
}

]

}

]}
"moniker": {
  "app": "sampleapplicaion"
},

"name": "Deploy Canary",
"refId": "21",
"relationships": {
  "loadBalancers": [],
  "securityGroups": []
},

"requiredArtifactIds": [
  "ac842617-988f-48dc-a7a4-7f020d93cc42"
],

"requisiteStageRefIds": [],
"source": "text",
"type": "deployManifest"
},

{
  "analysisType": "realTime",
  "canaryConfig": {
    "beginCanaryAnalysisAfterMins": "2",
    "canaryAnalysisIntervalMins": "",
    "canaryConfigId": "7bd4ab4-f933-4a41-865f-6d3e9c786351",
    "combinedCanaryResultStrategy": "LOWEST",
    "lifetimeDuration": "PT0H5M",
    "metricsAccountName": "my-prometheus",
    "scopes": [
      {
        "controlLocation": "default",
        "controlScope": "k8s-sampleapplicaion.default.80.k8s-sampleapplicaion.default.8000.svc-baseline",
        "experimentLocation": "default",
        "experimentScope": "k8s-sampleapplicaion.default.80.k8s-sampleapplicaion.default.8080.svc-canary",
        "extendedScopeParams": {
        }
      }
    ]
  }
}
"scopeName": "default"

},
"scoreThresholds": {
"marginal": "0",
"pass": "70"
}

,"storageAccountName": "kayenta-minio"
}

,"name": "Canary Analysis",
"refId": "25",
"requisiteStageRefIds": [
"20",
"21"
],
"type": "kayentaCanary"
}

{"continuePipeline": false,
"failPipeline": true,
"job": "NJob",
"master": "master",
"name": "Auto Cleanup: GCR Image and code revert",
"parameters": {
}

,"refId": "26",
"requisiteStageRefIds": [
"25"
],
"stageEnabled": {
"type": "expression"
}

,"type": "jenkins"
}]

}
A sample JSON file for automated canary configuration

Following is a sample JSON file for automated canary configuration.
"marginal": 75,
"pass": 95
}

"configVersion": "1",
"createdTimestamp": 1552650414234,
"createdTimestampIso": "2019-03-15T11:46:54.234Z",
"description": "Canary Config",
"judge": {

"judgeConfigurations": {
)

, "name": "NetflixACAJudge-v1.0"
).
,
"metrics": [
{

"analysisConfigurations": {

"canary": {

"direction": "increase"

}

}

,
"groups": [
"Group 1"
],
"name": "Server Response Errors - 5XX",
"query": {

"customFilterTemplate": "tot_requests",
"metricName": "netscaler_lb_vserver_svr_busy_err_rate",
"serviceType": "prometheus",
"type": "prometheus"

,
"scopeName": "default"

},

}
{  
  "analysisConfigurations": {  
    "canary": {  
      "direction": "either",
      "nanStrategy": "replace"
    }
  },
  "groups": [  
    "Group 2"
  ],
  "name": "Server Response Latency - TTFB",
  "query": {  
    "customFilterTemplate": "ttfb",
    "metricName": "netscaler_lb_vserver_hits_total",
    "serviceType": "prometheus",
    "type": "prometheus"
  },
  "scopeName": "default"
},
"name": "canary-config",
"templates": {  
  "tot_requests": "lb_vserver_name = \"${scope} \"",
  "ttfb": "lb_vserver_name = \"${scope} \"
},
"updatedTimestamp": 1553098513495,
"updatedTimestampIso": "2019-03-20T16:15:13.495Z"
}  

<!--NeedCopy-->
Simplified canary deployment using Ingress annotations

This topic provides information about the simplified Canary deployment using Ingress annotations. While Citrix provides multiple options to support canary deployment, this is a simpler type of Canary deployment.

Canary using Ingress annotations is a rule based canary deployment. In this approach, you need to define an additional Ingress object with specific annotations to indicate that the application request needs to be served based on the rule based canary deployment strategy. In the Citrix solution, Canary based traffic routing at the Ingress level can be achieved by defining various sets of rules as follows:

- Applying the canary rules based on weight
- Applying the canary rules based on the HTTP request header
- Applying the canary rules based on the HTTP header value

The order of precedence of the canary rules is as follows:

Canary by HTTP request header value -> canary by HTTP request header -> canary by weight

Canary deployment based on weight

Weight based canary deployment is a widely used canary deployment approach. In this approach, you can set the weight as a range from 0 to 100 which decides the percentage of traffic to be directed to the canary version and the production version of an application.

Following is the workflow for the weight based canary deployment:

- Initially the weight can be set to zero which indicates that the traffic is not forwarded to the canary version.
- Once you decide to start canary deployment, change the weight to the required percentage to make sure the traffic is directed to canary version as well.
- Finally, when you determine that the canary version is ready to be released, change the weight to 100 to ensure that all the traffic is being directed to the canary version.

For deploying weight based canary using the Citrix ingress controller, create a new Ingress with a canary annotation ingress.citrix.com/canary-weight: and specify the percentage of traffic to be directed to the canary version.

Canary deployment based on the HTTP request header

You can configure canary deployment based on the HTTP request header which is controlled by clients. The request header notifies the Ingress to route the request to the service specified in the canary Ingress. When the request header contains the value mentioned in the Ingress annotation ingress.citrix.com/canary-by-header:, the request is routed to the service specified in the canary Ingress.
Canary deployment based on the HTTP request header value

You can also configure canary deployment based on values of the HTTP request header which is an extension of canary by header. In this deployment, along with the `ingress.citrix.com/canary-by-header:` annotation, you also specify the `ingress.citrix.com/canary-by-header-value:` annotation. When the request header value matches with the value specified in the Ingress annotation `ingress.citrix.com/canary-by-header-value:` the request is routed to the service specified in the canary Ingress. You can specify multiple header values as a list of strings.

Following is a sample annotation for canary deployment based on the HTTP request header values:

```
ingress.citrix.com/canary-by-header-value: ['value1','value2','value3','value4']
```

Configure canary deployment using Ingress annotations

Perform the following steps to deploy a sample application as a canary release.

1. Deploy the Citrix ingress controller using the steps in deploy the Citrix ingress controller. You can either deploy the Citrix ingress controller as a sidecar with Citrix ADC CPX or as a standalone pod which controls Citrix ADC VPX or MPX.

2. Deploy the Guestbook application using the `guestbook-deploy.yaml` file.

```
kubectl apply -f guestbook-deploy.yaml
```

3. Deploy a service to expose the Guestbook application using the `guestbook-service.yaml` file.

```
kubectl apply -f guestbook-service.yaml
```

4. Deploy the Ingress object for the Guestbook application using the `guestbook-ingress.yaml` file.

```
kubectl apply -f guestbook-ingress.yaml
```

5. Deploy a canary version of the Guestbook application using the `canary-deployment.yaml` file.

```
kubectl apply -f canary-deployment.yaml
```

6. Deploy a service to expose the canary version of the Guestbook application using the `canary-service.yaml` file.
7. Deploy an Ingress object with annotations for the canary version of the Guestbook application using the canary-ingress.yaml file.

```yaml
kubectl apply -f canary-ingress.yaml
```

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    ingress.citrix.com/canary-weight: "10"
    kubernetes.io/ingress.class: citrix
    name: canary-by-weight
spec:
  rules:
  - host: webapp.com
    http:
      paths:
      - backend:
          service:
            name: guestbook-canary
            port:
              number: 80
      path: /
      pathType: Prefix
```

Here, the annotation `ingress.citrix.com/canary-weight: "10"` is the annotation for the weight based canary. This annotation specifies the Citrix ingress controller to configure the Citrix ADC in such a way that 10 percent of the total requests destined to `webapp.com` is sent to the `guestbook-canary` service. This is the service for the canary version of the Guestbook application.

For deploying the HTTP header based canary using the Citrix ingress controller, replace the canary annotation `ingress.citrix.com/canary-weight: "10"` with the `ingress.citrix.com/canary-by-header: annotation` in the canary-ingress.yaml file.

For deploying the HTTP header value based canary using the Citrix ingress controller, replace the `ingress.citrix.com/canary-weight: annotation` with the `ingress.citrix.com/canary-by-header: and ingress.citrix.com/canary-by-header-value: annotations in the`
Deploy Citrix API gateway using Red Hat OpenShift Operators

February 3, 2022

OpenShift Operator is an open-source toolkit designed to package, deploy, and manage Kubernetes native applications in a more effective, automated, and scalable way.

An API gateway acts as the single entry point for your APIs and ensures secure and reliable access to multiple APIs and microservices in your system. Citrix provides an enterprise grade API gateway for North-South API traffic into the OpenShift cluster. The API gateway integrates with the OpenShift cluster through the Citrix ingress controller and the Citrix ADC (Citrix ADC MPX, VPX, or CPX) deployed as the Ingress Gateway for on-premises or cloud deployments.

This topic covers information on how to deploy the API gateway using OpenShift Operators.

The Citrix API gateway operator installs the CRDs required for the API Gateway functionality. Since this operator leverages the Citrix ingress controller, you need to create an instance of the API Gateway CRD, which is the Citrix ingress controller. If you already have a Citrix ingress controller instance in the cluster you need to install it again.

API gateway supports the following CRDs:

- **Auth CRD** (authpolicies.citrix.com)
  
  Authentication policies are used to enforce access restrictions to resources hosted by an application or an API server.

- **Content routing CRDs** (httproutes.citrix.com and listeners.citrix.com)
  
  Kubernetes native Ingress supports only basic host and path based routing. Using content routing CRDs, you can expose the advanced content routing abilities provided by Citrix ADC like routing based on header values or query strings.

- **Rate limit CRD** (ratelimits.citrix.com)
  
  In a Kubernetes deployment, you can rate limit the requests to the resources on the back end server or services using the rate limiting feature provided by the ingress Citrix ADC.

- **Rewrite and responder CRD** (rewritepolicies.citrix.com)
  
  In a Kubernetes environment, to deploy specific layer 7 policies (such as redirecting HTTP traffic
Citrix ADC ingress controller

to a specific URL) requires you to add appropriate libraries within the microservices and manually configure the policies. Instead, you can use the Rewrite and Responder features provided by the Ingress Citrix ADC device to deploy these policies.

- VIP CRD (vips.citrix.com)

Citrix provides a VIP CRD for asynchronous communication between the IPAM controller and the Citrix ingress controller. The IPAM controller is provided by Citrix for IP address management.

**Deploy Citrix API Gateway using OpenShift Operator**

Perform the following steps to deploy the API gateway using the OpenShift Operator.

1. In the OperatorHub, search for **Citrix API Gateway Operator**.

2. Select the **Citrix API Gateway Operator**, and click **Install**.
3. Choose the namespace where you want the operator to be installed and click **Subscribe**.

4. Once the Citrix API gateway operator is installed, you can see it listed under the **Installed Operators** section on the left panel. At this point, all CRDs are installed in the cluster.
5. To create instances of each CRD, select the appropriate CRD and click **Create Instance**.

6. Once you create the instance you can see all your instances by selecting the **All instances** tab.
Citrix ADC ingress controller

Update the CRD instance

To update any values of the created CRD instance, click the instance and go to the YAML tab and edit the YAML file and click Save.

Upgrade CRD versions

For more information on upgrading CRD versions, see the OpenShift documentation.
Delete the instance and operator

To delete an instance of the CRD you have to delete the Citrix ingress controller instance from the user interface. Select the three vertical dots on the right side of the instance as shown in the following image and then click **Delete CitrixIngressController**.

To delete the operator, click the **Actions** drop-down list and then click **Uninstall operator**.

**Note:**
Make sure that all instances are deleted manually before uninstalling the operator. The instances are not automatically deleted if you uninstall the operator.
Deploying Citrix API gateway using Rancher

February 1, 2022

Citrix API gateway provides a single entry point for APIs by ensuring secure and reliable access to APIs and microservices on your system. Citrix provides an enterprise-grade API gateway for North-South API traffic for Kubernetes clusters.

Citrix API gateway integrates with Kubernetes through the Citrix ingress controller and the Citrix ADC (Citrix ADC MPX, VPX, or CPX) deployed as the Ingress Gateway for on-premises and cloud deployments.

You can use the Rancher platform to deploy Citrix API gateway. Rancher provides a catalog of application templates that help you to deploy Citrix API gateway.

Prerequisites

You must import the cluster, in which you want to deploy the API gateway, to the Rancher platform.

Import the cluster to the Rancher platform

Perform the following steps to import your cluster to the Rancher platform:

1. Log in to the Rancher platform.
2. In the Clusters page, click **Add Cluster**.

3. In the Add Cluster - Select Cluster Type page, choose the **Import an existing cluster** option.

4. Specify the **Cluster Name**.

5. Specify **Member Roles, Labels**, and **Annotations**.

6. Click **Create**.

### Deploy Citrix API gateway using the Rancher platform

Perform the following steps to deploy the API gateway on the cluster using the Rancher platform:

1. Log in to the Rancher platform.

2. From the **Global** drop-down list, select the cluster that you have imported.

3. Select the **Apps** tab and click **Launch**.

4. From the Catalog page, choose the **citrix-api-gateway** template.
5. Specify the mandatory and required fields under **Configuration Options** (includes deployment settings, ADC settings, the Citrix ingress controller image settings, and exporter settings).

The mandatory fields include:

- **Namespace:** Specify the namespace where you want to create the Citrix ingress controller. You can also use the **Edit as YAML** option to specify the same in the YAML file.
- **Accept License:** Select **Yes** to accept the terms and conditions of the Citrix license.
- **Login File Name:** Specify the name of the Kubernetes secret. The secret file is used for the Citrix ADC login.
- **Citrix ADC IP:** It is the NSIP or SNIP of the Citrix ADC device. For high availability, specify the SNIP as the IP address.

6. Click **Preview** to verify the information and click **Launch**.

### Deploy API Gateway with GitOps

February 3, 2022

Custom Resource Definitions (CRDs) are the primary way of configuring API gateway policies in cloud native deployments. Operations teams create the configuration policies (routing, authentication, rewrite, Web Application Firewall (WAF), and so on) and apply them in the form of CRDs. In an API Gateway context, these policies are applied on the specific APIs and upstream hosting these APIs.

API developers document the API details in an Open API specification format for the client software developers and peer service implementation teams for using the API details. API documents contain information such as base path, path, method, authentication, and authorization.

Operation teams can use the information in an API specification document to configure the API Gate-
Citrix ADC ingress controller

way. Git, a source control solution, is used extensively by developers and operations teams. The GitOps solution makes the collaboration and communication that take place between development and operations teams easier. GitOps helps to create a faster, more streamlined, and continuous delivery for Kubernetes without losing stability.

The API Gateway deployment with the GitOps solution enables operations teams to use the API specification document created by software developers in the API gateway configuration. This solution automates the tasks and information exchange between API development and operations teams.

About the GitOps solution for API Gateway

The GitOps solution is constituted mainly by three entities:

- Open API specification document
- Policy template CRDs
- API Gateway deployment CRD

Open API Specification document

Created by API developers or API designers, the document provides an API information. The GitOps solution uses the following details from an Open API specification document:

- Base path
- Path
- Method
- Tags
- Authentication
- Authorization

The following is a sample Open API specification file with the details (in red) that are used to automatically create policies.
Policy template CRDs

CRDs are the primary way of configuring an API gateway instance. The operations team creates and manages the CRD implementations. In the traditional workflow, as part of creating the policies, the operations team manually fills the target details such as upstream and API path in the CRD instances. In the GitOps solution, the API path and upstream details are derived automatically. Operations team creates the CRDs without any target details and the solution refer to such CRD instances as policy templates.

The GitOps solution supports the following policy templates:

- Rewrite policy
- Rate limit policy
- Authentication policy
- WAF

The following is a sample rewrite policy template:

**Note:** For information on how to create a CRD instance, see the individual CRDs.
Citrix ADC ingress controller

API Gateway deployment CRD

API Gateway deployment CRD binds the API specification document with policy templates. This CRD enables mapping of API resources with upstream services and API gateway policies related to routing and security. The API Gateway deployment CRD is maintained by the operations team with the data received from the development team.

The API Gateway deployment CRD configures the following:

- Git repository details
- Endpoint listener
- API to upstream mapping
- API to policy mapping
- Open API authentication policy references to authentication policy template mapping

Alternatively, API Gateway CRD supports non-Git sources for fetching OpenAPI Specification (OAS) doc-
Citrix ADC ingress controller

Currently, both HTTP and HTTPS URL sources are supported. These URLs can be password protected and basic HTTP authentication is supported. Credentials can be configured using the same fields as that of Git based OAS file sources.

The following image shows the API Gateway deployment CRD binding the API specification with policy templates using the API selectors and policy mappings.

APIs that start with the /pet regular expression is selected with the path regexp pattern and APIs with /play is selected with the play tag. Security definitions in the API specification document are mapped with the available authentication, authorization, and auditing configurations in the authentication CRD template.

Configure API Gateway CRD

The API Gateway CRD binds the API resources defined in the Swagger specification with policies defined in the other CRDs.

Prerequisites

Apply CRD definitions for the following CRD objects:

- Listener
- HTTP route
- Rate limit
- Rewrite
Citrix ADC ingress controller

- **Authentication**
- **WAF**

The following sections provide information about the various elements in the API Gateway CRD configuration file:

**API definition**

It provides information about the Git repository in which the Git watcher monitors for the Open API specification files.

**API definition: Git repository access details**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repository</td>
<td>Specifies the Git repository URL.</td>
</tr>
<tr>
<td>Branch</td>
<td>Specifies the Git branch name (By default, master).</td>
</tr>
<tr>
<td>oas_secret_ref</td>
<td>Specifies the Git access secret reference as a Kubernetes secret object name. <strong>Note:</strong> When creating a secret, keep the <strong>username</strong> and <strong>password</strong> as the secret field names for Git access credentials.</td>
</tr>
<tr>
<td>Files</td>
<td>The credentials for these OAS URLs can be accessed from the oas_secret_ref field or user_name and password field combinations.</td>
</tr>
</tbody>
</table>

**API proxy**

It provides information about the endpoint (VIP) configuration that is used to expose the APIs on the API Gateway front end.

**api_proxy: VIP details**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip_address</td>
<td>Specifies the IP address of the end point (VIP).</td>
</tr>
<tr>
<td>port</td>
<td>Specifies the endpoint port.</td>
</tr>
</tbody>
</table>
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocol</td>
<td>Specifies the protocol (HTTP/HTTPs).</td>
</tr>
<tr>
<td>secret</td>
<td>Specifies the SSL certificate secret for the endpoint configuration.</td>
</tr>
</tbody>
</table>

### Policy mappings

It maps the API resources with the upstream services and policy templates. Some information in this section is collected from the developers when the operations team creating the CRD.

<table>
<thead>
<tr>
<th>Section</th>
<th>Sub section</th>
<th>Field</th>
<th>Sub field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Policies</td>
<td></td>
<td></td>
<td></td>
<td>Specifies the policy and upstream mapping.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Name</td>
<td></td>
<td>Specifies the name of the policy. It is unique in a CRD instance.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Selector</td>
<td></td>
<td>A list of filters for selecting the API resources.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>API</td>
<td></td>
<td>Specifies the Regexp pattern for the API selection. All the APIs that match with this pattern are selected for applying policies from this block.</td>
</tr>
<tr>
<td>Section</td>
<td>Sub section</td>
<td>Field</td>
<td>Sub field</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------</td>
<td>-----------</td>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>method</td>
<td></td>
<td>A list of HTTP verbs, if the API resource verb matches with ANY in the list, it is selected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tags</td>
<td></td>
<td>A list of tags to match with an API. These tags are matched with tags in the API specification document. You can use either regex based path patterns or tags to match a policy.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Upstream</td>
<td></td>
<td>Specifies the upstream for the selected policy.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Service</td>
<td></td>
<td>Specifies the back-end service name.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Port</td>
<td></td>
<td>Specifies the back-end service port.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Policy-binding</td>
<td></td>
<td>Specifies the policy list to be applied on the selected API.</td>
</tr>
</tbody>
</table>
### Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>Section</th>
<th>Sub section</th>
<th>Field</th>
<th>Sub field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type of the policy template</td>
<td>Specifies the exact type of policy. Supported types are WAF, rewrite policy, and rate limit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Name</td>
<td>Specifies the name of the policy template.</td>
</tr>
</tbody>
</table>

#### AAA mappings

It maps the authentication references in the API specification document with the available policy definition sections in the authentication CRD template.

<table>
<thead>
<tr>
<th>Section</th>
<th>Sub section</th>
<th>Field</th>
<th>Sub field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaa</td>
<td></td>
<td></td>
<td>Crd_name</td>
<td>Authentication, authorization, and auditing policy section mappings.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Specifies the name of the authentication CRD template.</td>
</tr>
</tbody>
</table>
Perform the following steps to deploy the API Gateway CRD:

1. Download the API Gateway CRD.

2. Deploy the API Gateway CRD using the following command:

   ```bash
   kubectl create -f apigateway-crd.yaml
   ```

The following is an example API Gateway CRD configuration:

```yaml
apiVersion: citrix.com/v1beta1
kind: apigatewaypolicy
metadata:
  name: apigatewaypolicyinstance
spec:
  api_definition:
    repository: "https://code.citrite.net/scm/cnn/cic-gitops.git"
    branch: "modify-test-branch"
```
oas_secret_ref: "mysecret"

files:
- "test_gitwatcher/petstore.yaml"
- "test_gitwatcher/playstore.yaml"

api_proxy:
ipaddress: "10.106.172.83"
port: 80
protocol: "http"
secret: "listener-secret"
policies:
- name: "p1"
  selector:
    - api: "/pet.*"
      method: ["GET", "POST"]
  upstream:
    service: "pet-service"
    port: 80
  policy_bindings:
    ratelimit:
      name: "ratelimit-gitops-slow"
- name: "p2"
  selector:
    - api: "/user.*"
      method: ["GET", "POST"]
  upstream:
    service: "user-service"
    port: 80
  policy_bindings:
    ratelimit:
      name: "ratelimit-gitops-slow"
- name: "p3"
  selector:
    - tags: ["play"]
  upstream:
    service: "play-service"
    port: 80
  policy_bindings:
    ratelimit:
      name: "ratelimit-gitops"
  rewritepolicy:
    name: "prefixurl"
  waf:
    name: "buffoverflow"
  aaa:
    crd_name: authgitops
Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>mappings:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- petstore_auth: jwt-auth-provider</td>
</tr>
<tr>
<td>- api_key: introspect-auth-provider</td>
</tr>
</tbody>
</table>

Support for web insight based analytics

Web insight based analytics is now supported with the API gateway CRD. When you use GitOps, the following web insight parameters are enabled by default:

- httpurl
- httpuseragent
- httphost
- httpmethod
- httpcontenttype

Multi-cluster ingress and load balancing solution using the Citrix ingress controller

May 5, 2022

Overview

For ensuring high availability, proximity based load balancing, and scalability, you may need to deploy an application in multiple distributed Kubernetes clusters. When the same application is deployed in multiple Kubernetes clusters, a load balancing decision has to be taken among application instances dispersed across clusters.

For implementing a load balancing solution for distributed Kubernetes clusters, the health of applications across clusters needs to be monitored globally. You need to monitor application availability and performance, update the application status across clusters, collect statistics from endpoints in data centers, and share the statistics across data centers.

Citrix provides a multi-cluster ingress and load balancing solution which globally monitors applications, collect, and share metrics across different clusters, and provides intelligent load balancing decisions. It ensures better performance and reliability for your Kubernetes services that are exposed using Ingress or using type LoadBalancer.
**Deployment topology**

The following diagram explains a deployment topology for the multi-cluster ingress and load balancing solution for Kubernetes clusters.

**Note:**

Services of type LoadBalancer (available for bare metal clusters using Citrix ADC) are also supported.

This diagram shows a sample topology with two data centers and each data center contains multiple Kubernetes clusters. For data center 1, Citrix ADC CPX is deployed as the Ingress load balancer in each Kubernetes cluster. For data center 2, HAProxy is deployed as the load balancer in each Kubernetes cluster. Citrix multi-cluster ingress and load balancing solution for Kubernetes load balances across the ingresses.

**Note:**

Any ingress solution, including third party solutions such as Istio ingress gateway as well as the Citrix ingress controller with Citrix ADC MPX, VPX, or BLX is supported. This topology is just a sample deployment.

A Citrix global server load balancing (GSLB) device is configured for each data center. In each Citrix ADC which acts as a global load balancer, one site is configured as a local site representing the local data center. The other sites are configured as remote sites for each remote data center. The Citrix ADC (MPX or VPX) used as the GSLB can be also used as the Ingress appliance with the Citrix ingress controller.

The global server load balancing (GSLB) configuration synchronization option of the Citrix ADC is used to synchronize the configuration across the sites. The Citrix ADC appliance from which you use the
Citrix ADC ingress controller

synchronization is referred as the master node and the site where the configuration is copied as the subordinate site.

Each cluster in the deployment runs an instance of the GSLB Kubernetes controller. GSLB controller is the module responsible for the configuration of the Citrix ADC GSLB device. The GSLB controller configures the GSLB master device for the applications deployed in their respective cluster. The GSLB master device pushes the GSLB configuration to the remaining GSLB subordinate devices using the GSLB sync feature. When you synchronize a GSLB configuration, the configurations on all the GSLB sites participating in the GSLB setup are made similar to the configuration on the master site.

The multi-cluster ingress and load balancing solution can be applied for any Kubernetes object which is used to route traffic into the cluster.

The following global load balancing methods are supported:

- Round trip time (RTT)
- Static proximity
- Round robin (RR)

The following deployment types are supported:

- Local first: In a local first deployment, when an application wants to communicate with another application it prefers a local application in the same cluster. When the application is not available locally, the request is directed to other clusters or regions.

- Canary: Canary release is a technique to reduce the risk of introducing a new software version in production by first rolling out the change to a small subset of users. In this solution, canary deployment can be used when you want to roll out new versions of the application to selected clusters before moving it to production.

- Failover: A failover deployment is used when you want to deploy applications in an active/passive configuration when they cannot be deployed in active/active mode.

- Round trip time (RTT): In an RTT deployment, the real-time status of the network is monitored and dynamically directs the client request to the data center with the lowest RTT value.

- Static proximity: In a static proximity deployment, an IP-address based static proximity database is used to determine the proximity between the client's local DNS server and the GSLB sites. The requests are sent to the site that best matches the proximity criteria.

- Round robin: In a round robin deployment, the GSLB device continuously rotates a list of the services that are bound to it. When it receives a request, it assigns the connection to the first service in the list, and then moves that service to the bottom of the list.

Note:
Currently, IPv6 is not supported.
CRDs for configuring Multi-cluster ingress and load balancing solution for Kubernetes clusters

The following CRDs are introduced to support the Citrix ADC configuration for performing GSLB of Kubernetes applications.

- Global traffic policy (GTP)
- Global service entry (GSE)

**GTP CRD**

GTP CRD accepts the parameters for configuring GSLB on the Citrix ADC including deployment type (canary, failover, local-first), GSLB domain, health monitor for the Ingress, and service type.

The GTP CRD spec is available in the Citrix ingress controller GitHub repo at: grp-crd.yaml.

**GSE CRD**

GSE CRD dictates the endpoint information (any Kubernetes object which routes traffic into the cluster) in each cluster.

The GSE CRD Spec is available in the citrix ingress controller GitHub repo at: gse-crd.yaml

The GSE CRD is auto generated for an Ingress object if the service specified in the Ingress resource is referred in the GTP CRD instance and the `status-loadbalancer-ip/hostname` field is already populated. For a service of type `LoadBalancer`, the GSE CRD is auto generated if the service is referred in the GTP CRD instance and the `status-loadbalancer-ip/hostname` field is already populated.

**Note:**

For GSE CRD auto generation in the case of Ingress, host name should exactly match with the host name specified in the GTP CRD instance. For both Ingress and service of type `LoadBalancer`, the GSE CRD is generated only for the first port specified.

**Deploy Citrix Multi-cluster ingress and load balancing solution**

The following prerequisites apply before performing the steps to deploy the Citrix global load balancing solution:

- You should configure GSLB sites on the Citrix ADC which acts as the GSLB device.
- Features like content switching and SSL should be enabled in the GSLB device
- For static proximity, the location database has to be applied externally

Perform the following steps to deploy the Citrix global load balancing solution for geographically distributed Kubernetes clusters.
1. Create the RBAC permissions required to deploy the GSLB controller using the `gslb-rbac.yaml` file.

   ```bash
   kubectl apply -f gslb-rbac.yaml
   ```

2. Create the secrets required for the GSLB controller to connect to GSLB devices and push the configuration from the GSLB controller.

   ```bash
   kubectl create secret generic secret-1 --from-literal=username=<username for gslb device1> --from-literal=password=<password for gslb device1>
   kubectl create secret generic secret-2 --from-literal=username=<username for gslb device2> --from-literal=password=<password for gslb device2>
   ```

   **Note:**
   These secrets are used in the GSLB controller YAML file for the respective sites. The `username` and `password` in the command specifies the user name and password of the Citrix GSLB ADC.

3. Download the GSLB controller YAML file `gslb-controller.yaml`.

4. Edit the GSLB controller YAML file and update the following values as per the requirements of each cluster.

   - `LOCAL_REGION` and `LOCAL_CLUSTER`: Specify the region and cluster name where this controller is deployed.
   - `SITENAMES`: Provide site names separated by commas and the configuration should be the same as the site configured on GSLB devices.
   - IP address, region, user name, and password for each site should start with the corresponding site name.
     For example: For site1 in `SITENAMES`, fields should be `site1_ip`, `site1_region`, `site1_username`, and `site1_password`.
   - argument section in the specification should include `--config-interface` and `gslb-endpoint`.

   The following is a snippet of the YAML file for deploying the GSLB controller.

   ```yaml
   env:
   - name: "LOCAL_REGION"
   ```
Note:
The order of the GSLB site information should be the same in all clusters. First site in the order is considered as the master site for pushing the configuration. When that master site goes down, the next site in the list will be the new master. Hence, the order of the sites should be the same in all Kubernetes clusters. For example, if the order of sites is site1 followed by site2 in cluster1 all other clusters should follow
5. Deploy the modified GSLB controller YAML file specific to each cluster on the corresponding cluster.

```bash
kubectl apply -f gslb-controller.yaml
```

6. Deploy the GTP CRD definition YAML file, using the following command.

```bash
kubectl create -f gtp-crd.yaml
```

7. Deploy the GSE CRD definition YAML file using the following command.

```bash
kubectl create -f gse-crd.yaml
```

8. Define the GTPs for your domain as YAML files and apply GTP instances.

```bash
kubectl create -f gtp-example.yaml
```

**Note:**

GTP CRD should be applied across all clusters with the same configuration for the same domain.

Following is an example for a global traffic policy configuration where traffic policy is specified as local first for the domain app2.com. When your application prefers services local to it, you can use this option. The CIDR of the local cluster (cluster1) is specified using the CIDR field. The weight field is used to direct more client requests to any particular cluster than other clusters when the GSLB decision is taken by the Citrix ADC. The load balancing method is specified using the secLbMethod field as round robin.

**Note:**

You can specify the load balancing method for local first, canary, and failover deployments.

```yaml
apiVersion: "citrix.com/v1beta1"
kind: globaltrafficpolicy
metadata:
  name: gtp1
```
namespace: default
spec:
  serviceType: 'HTTP'
  hosts:
    - host: 'app2.com'
  policy:
    trafficPolicy: 'LOCAL-FIRST'
    secLbMethod: 'ROUNDROBIN'
  targets:
    - destination: 'app2.default.east.cluster1'
      CIDR: '10.102.217.69/24'
      weight: 1
    - destination: 'app2.default.west.cluster2'
      weight: 1
  monitor:
    - monType: tcp
      uri: ''
      respCode: 200

For more information on other GTP deployment options like canary and failover, see Examples: Global traffic policy deployments.

9. Apply GSE instances manually for GSLB of ingress.

```
kubectl create -f gse-example.yaml
```

Note:

GSE CRD is applied in a specific cluster based on the cluster endpoint information. The global service entry name should be the same as the target destination name in the global traffic policy.

Following is an example for a global service entry.

```
apiVersion: "citrix.com/v1beta1"
kind: globalserviceentry
metadata:
  name: 'app2.default.east.cluster1'
namespace: default
spec:
  endpoint:
    ipv4address: 10.102.217.70
    monitorPort: 33036
```
Citrix ADC ingress controller

In this example, the global service entry name `app2.default.east.cluster1` is one of the target destination names in the global traffic policy created in step 8.

10. Apply service YAML for GSLB of services of type LoadBalancer.

```
kubectl create -f service-example.yaml
```

Following is a sample service.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: cold
  namespace: default
spec:
  type: LoadBalancer
  ports:
    - name: port-8080
      port: 8080
targetPort: 8080
selector:
  dep: citrixapp
status:
  loadBalancer:
    ingress:
      - ip: 10.102.217.72
```

For a sample configuration of multi-cloud ingress and load balancing solution for Amazon EKS and Microsoft AKS clusters using Citrix ADC, see the Multi-cloud and multi-cluster ingress and load balancing solution with Amazon EKS and Microsoft AKS clusters.

**How to direct the DNS resolution of pods to Citrix GSLB ADC**

**Note:**

This procedure is optional and needed only if the pods within the cluster need to resolve the DNS through the Citrix GSLB ADC.

**When pods are within the Kubernetes cluster**

When you want the pods in a Kubernetes cluster to use the GSLB solution, the ConfigMap of the DNS provider should be updated to forward the request for a domain (for which GSLB is required) to Citrix
The following example shows how to update the ConfigMap if the DNS provider is CoreDNS.

```yaml
# kubectl edit configmap coredns -n kube-system
apiVersion: v1
data:
  Corefile: |
  cluster.local:53 {
    errors
    health
    kubernetes cluster.local in-addr.arpa ip6.arpa {
      pods insecure
      upstream
      fallthrough in-addr.arpa ip6.arpa
      ttl 60
    }
    prometheus :9153
    forward . /etc/resolv.conf
    cache 90
    loop
    reload
    loadbalance
  }

  app2.com.:53 {
    errors
    cache 30
    forward . 10.102.217.149
  }

kind: ConfigMap
metadata:
  creationTimestamp: "2019-08-30T10:59:36Z"
  name: coredns
  namespace: kube-system
  resourceVersion: "43569751"
  selfLink: /api/v1/namespaces/kube-system/configmaps/coredns
  uid: ac1d92e4-260f-45bd-8708-5f8194482885
```
As shown in the example, you need to add the required configuration for your domain if you want a pod to have a GSLB decision for applications hosted behind a domain. Here, the domain name is `app2.com`.

```
app2.com.:53 {
  errors
  cache 30
  forward . 10.102.217.149
}
```

The IP address specified (`forward . 10.102.217.149`) is a DNS service configured in the Citrix GSLB ADC. You can specify the multiple IP addresses of different GSLB sites by separating them with spaces as shown as follows.

```
forward . ip1 ip2 ip3
```

**When pods are within the OpenShift cluster**

When you want the pods in an OpenShift cluster to use the GSLB solution, the DNS operator should be updated to forward the request for a domain (for which GSLB is required) to Citrix GSLB ADC.

```
# oc edit dns.operator/default
apiVersion: operator.openshift.io/v1
kind: DNS
metadata:
  name: default
spec:
  servers:
    - name: gslb-app2
      zones:
        - app2.com
      forwardPlugin:
        upstreams:
          - 10.102.217.149
          - 10.102.218.129:5353
```
As shown in the example, you need to add the required configuration for your domain if you want a pod to have a GSLB decision for applications hosted behind a domain. Here, the domain name is app2.com.

```
servers:
  - name: gslb-app2
    zones:
      - app2.com
    forwardPlugin:
      upstreams:
        - 10.102.217.149
        - 10.102.218.129:5353
```

The IP addresses specified (10.102.217.149 and 10.102.218.129:5353) are DNS services configured in the Citrix GSLB ADC.

This configuration can be verified using the following command:

```
# oc get configmap/dns-default -n openshift-dns -o yaml
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  labels:
    dns.operator.openshift.io/owning-dns: default
    manager: dns-operator
    name: dns-default
    namespace: openshift-dns
data:
  Corefile: |
    # gslb-app2
    app2.com:5353 {
      forward . 10.102.217.149 10.102.218.129:5353
      errors
      bufsize 512
    }
  .:5353 {
    bufsize 512
    errors
```
Citrix ADC ingress controller

```yaml
health {
  lameduck 20s
}
ready
kubernetes cluster.local in-addr.arpa ip6.arpa {
  pods insecure
  fallthrough in-addr.arpa ip6.arpa
}
prometheus 127.0.0.1:9153
forward . /etc/resolv.conf {
  policy sequential
}
cache 900 {
  denial 9984 30
}
reload
}

GTP CRD definition

GTP CRD definition is available at gtp-crd.yaml

The following table explains the GTP CRD attributes.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ipType</td>
<td>Specifies the DNS record type as A or AAAA. Currently, only A record type is supported</td>
</tr>
<tr>
<td>serviceType:</td>
<td>Specifies the protocol to which multi-cluster support is applied.</td>
</tr>
<tr>
<td>host</td>
<td>Specifies the domain for which multi-cluster support is applied.</td>
</tr>
<tr>
<td>trafficPolicy</td>
<td>Specifies the traffic distribution policy supported in a multi-cluster deployment.</td>
</tr>
</tbody>
</table>

© 1999–2022 Citrix Systems, Inc. All rights reserved.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sourceIpPersistenceId</td>
<td>Specifies the unique source IP persistence ID. This attribute enables</td>
</tr>
<tr>
<td></td>
<td>persistence based on the source IP address for the inbound packets. The</td>
</tr>
<tr>
<td></td>
<td>sourceIpPersistenceId attribute should be a multiple of 100 and should be</td>
</tr>
<tr>
<td></td>
<td>unique. For a sample configuration, see Example: source IP persistence.</td>
</tr>
<tr>
<td>secLbMethod</td>
<td>Specifies the traffic distribution policy supported among clusters under a</td>
</tr>
<tr>
<td></td>
<td>group in local-first, canary, or failover.</td>
</tr>
<tr>
<td>destination</td>
<td>Specifies the Ingress or LoadBalancer service endpoint in each cluster.</td>
</tr>
<tr>
<td></td>
<td>The destination name should match with the name of GSE.</td>
</tr>
<tr>
<td>weight</td>
<td>Specifies the proportion of traffic to be distributed across clusters. For</td>
</tr>
<tr>
<td></td>
<td>canary deployment, the proportion is specified as percentage.</td>
</tr>
<tr>
<td>CIDR</td>
<td>Specifies the CIDR to be used in local-first to determine the scope of</td>
</tr>
<tr>
<td></td>
<td>locality.</td>
</tr>
<tr>
<td>primary</td>
<td>Specifies whether the destination is a primary cluster or a backup cluster</td>
</tr>
<tr>
<td></td>
<td>in the failover deployment.</td>
</tr>
<tr>
<td>monType</td>
<td>Specifies the type of probe to determine the health of the multi-cluster</td>
</tr>
<tr>
<td></td>
<td>endpoint. When the monitor type is HTTPS, SNI is enabled by default during</td>
</tr>
<tr>
<td></td>
<td>the TLS handshake.</td>
</tr>
<tr>
<td>uri</td>
<td>Specifies the path to be probed for the health of the multi-cluster endpoint</td>
</tr>
<tr>
<td></td>
<td>for HTTP and HTTPS.</td>
</tr>
<tr>
<td>respCode</td>
<td>Specifies the response code expected to mark the multi-cluster endpoint as</td>
</tr>
<tr>
<td></td>
<td>healthy for HTTP and HTTPS.</td>
</tr>
</tbody>
</table>
GSE CRD definition

GSE CRD definition is available at gse-crd.yaml

Examples: Global traffic policy deployments

Canary deployment

You can use the canary deployment when you want to roll out new versions of the application to selected clusters before moving it to production.

This section explains a sample global traffic policy with Canary deployment, where you need to roll out a newer version of an application in stages before deploying it in production.

In this example, a stable version of an application is deployed in a cluster `cluster2` in the `west` region. A new version of the application is getting deployed in `cluster1` of the `east` region. Using the `weight` field you can specify how much traffic is redirected to each cluster. Here, `weight` is specified as 40 percent. Hence, only 40 percent of the traffic is directed to the new version. If the `weight` field is not mentioned for a destination, it is considered as part of the production which takes the majority traffic. When the newer version of the application is found as stable, the new version can be rolled out to other clusters as well.

```yaml
apiVersion: "citrix.com/v1beta1"
kind: globaltrafficpolicy
metadata:
  name: gtp1
  namespace: default
spec:
  serviceType: 'HTTP'
  hosts:
    - host: 'app1.com'
      policy:
        trafficPolicy: 'CANARY'
        secLbMethod: 'ROUNDROBIN'
        targets:
          - destination: 'app1.default.east.cluster1'
            weight: 40
          - destination: 'app1.default.west.cluster2'
            monitor:
              - monType: http
                uri: ''
                respCode: 200
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
Failover deployment

You can use the failover deployment when you want to deploy applications in an active/passive configuration.

In a failover deployment, the application is deployed in multiple clusters and these clusters are grouped into an active cluster group (group1) and a passive cluster group (group2). At any time, only one set of clusters is active while the other set remains passive. When all the clusters in group1 are unavailable, the clusters in group2 moves to the active state. When any of the clusters in group1 becomes available at a later point, group1 moves to the active state and group2 moves to the passive state.

The following example shows a sample GTP configuration for failover. Using the primary field, you can specify which cluster belongs to the active group and which cluster belongs to the passive group. The default value for the field is True indicating that the cluster belongs to the active group. You can use the weight field to direct more client requests to a specific cluster within a group than the other clusters if the configured method is round robin. The monitor parameter in the global traffic policy is used to configure the monitor in the Citrix ADC. The monitor can be bound to endpoints in each cluster to probe their health.

```yaml
apiVersion: "citrix.com/v1beta1"
kind: globaltrafficpolicy
metadata:
  name: gtp1
  namespace: default
spec:
  serviceType: 'HTTP'
  hosts:
    - host: 'app1.com'
      policy:
        trafficPolicy: 'FAILOVER'
        secLbMethod: 'ROUNDROBIN'
        targets:
          - destination: 'app1.default.east.cluster1'
            weight: 1
          - destination: 'app1.default.west.cluster2'
            primary: false
            weight: 1
      monitor:
        - monType: http
          uri: '
          respCode: 200
```
RTT deployment

Following is a sample global traffic policy for round trip time deployment.

```yaml
apiVersion: "citrix.com/v1beta1"
kind: globaltrafficpolicy
metadata:
  name: gtp1
  namespace: default
spec:
  serviceType: 'HTTP'
  hosts:
    - host: 'app1.com'
      policy:
        trafficPolicy: 'RTT'
        targets:
          - destination: 'app1.default.east.cluster1'
          - destination: 'app1.default.west.cluster2'
      monitor:
        monType: tcp
```

Round robin deployment

Following is a sample traffic policy for the round robin deployment. You can use this deployment when you need to distribute the traffic evenly across the clusters.

```yaml
apiVersion: "citrix.com/v1beta1"
kind: globaltrafficpolicy
metadata:
  name: gtp1
  namespace: default
spec:
  serviceType: 'HTTP'
  hosts:
    - host: 'app1.com'
      policy:
        trafficPolicy: 'ROUNDROBIN'
        targets:
          - destination: 'app1.default.east.cluster1'
          - destination: 'app1.default.west.cluster2'
```
Citrix ADC ingress controller

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>weight: 5</td>
</tr>
<tr>
<td>17</td>
<td>monitor:</td>
</tr>
<tr>
<td>18</td>
<td>- monType: tcp</td>
</tr>
<tr>
<td>19</td>
<td>uri: ''</td>
</tr>
<tr>
<td>20</td>
<td>respCode: 200</td>
</tr>
</tbody>
</table>

**Static Proximity**

Following is a sample traffic policy for the static proximity deployment.

```yaml
apiVersion: "citrix.com/v1beta1"
kind: globaltrafficpolicy
metadata:
  name: gtp1
  namespace: default
spec:
  serviceType: 'HTTP'
  hosts:
    - host: 'app1.com'
      policy:
        trafficPolicy: 'STATICPROXIMITY'
        targets:
          - destination: 'app1.default.east.cluster1'
          - destination: 'app1.default.west.cluster2'
    - monType: http
      uri: ''
      respCode: 200
```

**Example: source IP persistence**

The following traffic policy provides an example for enabling source IP persistence support. Source IP persistence can be enabled by providing the parameter `sourceIpPersistenceId`. The source IP persistence attribute can be enabled with the supported traffic policies.

```yaml
apiVersion: "citrix.com/v1beta1"
kind: globaltrafficpolicy
metadata
  name: gtp1
  namespace: default
```
Service Mesh lite

February 4, 2022

An Ingress solution (either hardware or virtualized or containerized) typically performs L7 proxy functions for north-south (N-S) traffic. The Service Mesh lite architecture uses the same Ingress solution to manage east-west traffic as well.

In a standard Kubernetes deployment, east-west (E-W) traffic traverses the built-in kube-proxy deployed in each node. Kube-proxy is an L4 proxy that can only perform TCP/UDP based load balancing and cannot offer the benefits provided by an L7 proxy.

Citrix ADC (MPX, VPX, or CPX) can provide the benefits of L7 proxy for E-W traffic such as:

- Mutual TLS and SSL offload.
- Content based routing, allow or block traffic based on HTTP and HTTPS header parameters.
- Advanced load balancing algorithms (least connections or least response time).
- Observability of east-west traffic through measuring golden signals (errors, latencies, saturation, traffic volume). Citrix ADM Service Graph is an observability solution to monitor and debug microservices.

A Service Mesh architecture (such as Istio or LinkerD) is complex to manage. Service Mesh lite architecture is a lightweight version and much simpler to get started to achieve the same requirements.

To configure east-west communication with Citrix ADC CPX in a Service mesh lite architecture, you must first understand how the kube-proxy is configured to manage east-west traffic.
East-west communication with kubectl-proxy

When you create a Kubernetes deployment for a microservice, Kubernetes deploys a set of pods based on the replica count. To access those pods, you create a Kubernetes Service which provides an abstraction to access those pods. The abstraction is provided by assigning a Cluster IP address to the service. Kubernetes DNS gets populated with an address record that maps the service name with the Cluster IP address. So, when an application, say tea wants to access a microservice named coffee then DNS returns the Cluster IP address of the coffee Service to the tea application. The tea application initiates a connection which is then intercepted by kube-proxy to load balance it to a set of coffee pods.

East-west communication with Citrix ADC CPX in Service Mesh Lite architecture

The goal is to insert the Citrix ADC CPX in the east-west path and use the Ingress rules to control this traffic.

Perform the following steps to configure east-west communication with Citrix ADC CPX.

Step 1: Modify the coffee service definition to point to Citrix ADC CPX

For Citrix ADC CPX to manage east-west traffic, the FQDN of the microservice (for example, coffee) should point to the Citrix ADC CPX IP address instead of the Cluster IP of the target microservice (coffee). (This Citrix ADC CPX deployment can be the same as the Ingress Citrix ADC CPX device.)
After this modification, when a pod in the Kubernetes cluster resolves the FQDN for the coffee service, the IP address of the Citrix ADC CPX is returned.

**Note:**

If you are deploying service mesh lite to bring up the service graph in Citrix ADM for observability, then you should add the label `citrix-adc: cpx` in all the services of your application which are pointing to the Citrix ADC CPX IP address after modifying the service.

### Step 2: Create a headless service named `coffee-headless` for coffee microservice pods

Since you have modified the `coffee` service to point to Citrix ADC CPX, you need to create one more service that represents coffee microservice deployment.

The following is a sample headless service resource:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: coffee-headless
spec:
  #headless Service
  clusterIP: None
  ports:
  - name: coffee-443
    port: 443
    targetPort: 443
  selector:
    name: coffee-deployment
```
**Step 3: Create an Ingress resource with rules for the coffee-headless service**

With the changes in the previous steps, you are now ready to create an Ingress object that configures the Citrix ADC CPX to control the east-west traffic to the coffee microservice pods.

The following is a sample Ingress resource:

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: coffee-ingress
spec:
rules:
  - host: coffee
    http:
      paths:
      - path: /
        backend:
          serviceName: coffee-headless
          servicePort: 80
```

Using the usual Ingress load balancing methodology with these changes, Citrix ADC CPX can now load balance the east-west traffic. The following diagrams show how the Citrix ADC CPX Service Mesh Lite architecture provides L7 proxying for east-west communication between tea and coffee microservices using the Ingress rules:
East-west communication with Citrix ADC MPX or VPX in Service Mesh lite architecture

Citrix ADC MPX or VPX acting as an Ingress can also load balance east-west microservice communication in a similar way as mentioned in the previous section with slight modifications. The following procedure shows how to achieve the same.

Step 1: Create an external service resolving the coffee host name to Citrix ADC MPX/VPX IP address

There are two ways to do it. You can add an external service mapping a host name or by using an IP address.

Mapping by a host name (CNAME)

- Create a domain name for the Ingress endpoint IP address (Content Switching virtual server IP address) in Citrix ADC MPX or VPX (for example, myadc-instance1.us-east-1.mydomain.com) and update it in your DNS server.
- Create a Kubernetes service for coffee with externalName as myadc-instance1.us-east-1.mydomain.com.
Now, when any pod looks up for the coffee microservice a CNAME(myadc-instance1.us-east-1.mydomain.com) is returned.

```yaml
kind: Service
apiVersion: v1
metadata:
  name: coffee
spec:
type: ExternalName
externalName: myadc-instance1.us-east-1.mydomain.com
```

Mapping a host name to an IP address

When you want your application to use the host name coffee that will redirect to the virtual IP address hosted in Citrix ADC MPX or VPX, you can create the following:

```yaml
---
kind: "Service"
apiVersion: "v1"
metadata:
  name: "coffee"
spec:
  ports:
    -
      name: "coffee"
      protocol: "TCP"
      port: 80
---
kind: "Endpoints"
apiVersion: "v1"
metadata:
  name: "coffee"
subsets:
  -
    addresses:
    -
      ip: "1.1.1.1" # Ingress IP in MPX
    ports:
      -
        port: 80
```
Step 2: Create a headless service for microservice pods

Since you have modified the coffee service to point to Citrix ADC MPX, you need to create one more service that represents coffee microservice deployment.

Step 3: Create an Ingress resource

Create an Ingress resource using the `ingress.citrix.com/frontend-ip` annotation where the value matches the Ingress endpoint IP address in Citrix ADC MPX or VPX.

Now, you can create an Ingress object that configures the Citrix ADC MPX or VPX to control the east-west traffic to the coffee microservice pods.

The following is a sample ingress resource:

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: coffee-ingress
  annotations:
    ingress.citrix.com/frontend-ip: "1.1.1.1"
spec:
rules:
- host: coffee
  http:
    paths:
    - path: /
      backend:
        serviceName: coffee-headless
        servicePort: 80
```

Using the usual ingress load balancing methodology with these changes Citrix ADC MPX can now load balance east-west traffic. The following diagram shows a Citrix ADC MPX or VPX configured as the N-S and E-W proxy using the Ingress rules.
Automated deployment of applications in Service Mesh lite

To deploy an application in a Service Mesh lite architecture, you need to perform multiple tasks manually. However, when you want to deploy multiple applications which consist of several microservices, you have an easier way to deploy the services in a Service Mesh lite architecture. Citrix provides you an automated way to generate ready to deploy YAMLs.

This document provides information on how to generate all the necessary YAMLs for Service Mesh lite deployment from your existing YAMLs using the Citrix provided script.

Deploy the Citrix ingress controller as an OpenShift router plug-in

February 3, 2022

In an OpenShift cluster, external clients need a way to access the services provided by pods. OpenShift provides two resources for communicating with services running in the cluster: routes and Ingress.

In an OpenShift cluster, a route exposes a service on a given domain name or associates a domain name with a service. OpenShift routers route external requests to services inside the OpenShift cluster according to the rules specified in routes. When you use the OpenShift router, you must also configure the external DNS to make sure that the traffic is landing on the router.

The Citrix ingress controller can be deployed as a router plug-in in the OpenShift cluster to integrate with Citrix ADCs deployed in your environment. The Citrix ingress controller enables you to use the
advanced load balancing and traffic management capabilities of Citrix ADC with your OpenShift cluster.

OpenShift routes can be secured or unsecured. Secured routes specify the TLS termination of the route.

The Citrix ingress controller supports the following OpenShift routes:

- **Unsecured Routes**: For Unsecured routes, HTTP traffic is not encrypted.
- **Edge Termination**: For edge termination, TLS is terminated at the router. Traffic from the router to the endpoints over the internal network is not encrypted.
- **Passthrough Termination**: With passthrough termination, the router is not involved in TLS offloading and encrypted traffic is sent straight to the destination.
- **Re-encryption Termination**: In re-encryption termination, the router terminates the TLS connection but then establishes another TLS connection to the endpoint.

For detailed information on routes, see the OpenShift documentation.

You can either deploy a Citrix ADC MPX or VPX appliance outside the OpenShift cluster or deploy Citrix ADC CPXs as pods inside the cluster. The Citrix ingress controller integrates Citrix ADCs with the OpenShift cluster and automatically configures Citrix ADCs based on rules specified in routes.

Based on how you want to use Citrix ADC, there are two ways to deploy the Citrix Ingress Controller as a router plug-in in the OpenShift cluster:

- As a **sidecar** container alongside Citrix ADC CPX in the same pod: In this mode, the Citrix ingress controller configures the Citrix ADC CPX.
- As a standalone pod in the OpenShift cluster: In this mode, you can control the Citrix ADC MPX or VPX appliance deployed outside the cluster.

For information on deploying the Citrix ingress controller to control the OpenShift ingress, see the Citrix ingress controller for Kubernetes.

You can use Citrix ADC for load balancing OpenShift control plane (master nodes). Citrix provides a solution to automate the configuration of Citrix ADC using Terraform instead of manually configuring the Citrix ADC. For more information, see Citrix ADC as a load balancer for the OpenShift control plane.

**Alternate Backend Support**

OpenShift Alternate backends is now supported by Citrix ingress controller.

Citrix ADC is configured according to the weights provided in the routes definition and traffic is distributed among the service pods based on those weights.

The following is an example of a route manifest with alternate backend:
For this route, 30 percent of the traffic is sent to the service apache-1 and 20 percent is sent to the service apache-2 and 50 percent to the service apache-3 based on weights provided in the route manifest.

**Supported Citrix components on OpenShift**

<table>
<thead>
<tr>
<th>Citrix components</th>
<th>Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Citrix ingress controller</td>
<td>Latest</td>
</tr>
<tr>
<td>Citrix ADC VPX</td>
<td>12.1 50.x and later</td>
</tr>
<tr>
<td>Citrix ADC CPX</td>
<td>13.0–36.28</td>
</tr>
</tbody>
</table>

**Note:**

CRDs provided for the Citrix ingress controller is not supported for OpenShift routes. You can use
Deploy Citrix ADC CPX as a router within the OpenShift cluster

In this deployment, you can use the Citrix ADC CPX instance for load balancing the North-South traffic to microservices in your OpenShift cluster. The Citrix ingress controller is deployed as a sidecar alongside the Citrix ADC CPX container in the same pod using the `cpx_cic_side_car.yaml` file.

**Before you begin:** When you deploy Citrix ADC CPX as a router, port conflicts can arise with the default router in OpenShift. You should remove the default router in OpenShift before deploying Citrix ADC CPX as a router. To remove the default router in OpenShift, perform the following steps:

1. Back up the default router configuration using the following command.

```bash
oc get -o yaml dc/router_clusterrolebinding/router-router-role serviceaccount/router > default-router-backup.yaml
```

2. Delete the default router using the following command.

```bash
oc delete -f default-router-backup.yaml
```

Perform the following steps to deploy Citrix ADC CPX as a router with the Citrix ingress controller as a sidecar.

1. Download the `cpx_cic_side_car.yaml` file using the following command:

```bash
wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/openshift/manifest/cpx_cic_side_car.yaml
```

2. Add the service account to privileged security context constraints (SCC) of OpenShift.

```bash
oc adm policy add-scc-to-user privileged system:serviceaccount:default:citrix
```

3. Deploy the Citrix ingress controller using the following command:

```bash
oc create -f cpx_cic_side_car.yaml
```
4. Verify if the Citrix ingress controller is deployed successfully using the following command:

```
oc get pods --all-namespaces
```

**Deploy Citrix ADC MPX/VPX as a router outside the OpenShift cluster**

In this deployment, the Citrix ingress controller which runs as a stand-alone pod allows you to control the Citrix ADC MPX, or VPX appliance from the OpenShift cluster. You can use the cic.yaml file for this deployment.

**Note:** The Citrix ADC MPX or VPX can be deployed in *standalone*, *high-availability*, or *clustered* modes.

**Prerequisites**

- Determine the IP address needed by the Citrix ingress controller to communicate with the Citrix ADC appliance. The IP address might be any one of the following depending on the type of Citrix ADC deployment:
  - **NSIP (for standalone appliances):** The management IP address of a standalone Citrix ADC appliance. For more information, see [IP Addressing in Citrix ADC](#).
  - **SNIP (for appliances in High Availability mode):** The subnet IP address. For more information, see [IP Addressing in Citrix ADC](#).
  - **CLIP (for appliances in clustered mode):** The cluster management IP (CLIP) address for a clustered Citrix ADC deployment. For more information, see [IP addressing for a cluster](#).

- The user name and password of the Citrix ADC VPX or MPX appliance used as the Ingress device. If you are not using the default credentials, the Citrix ADC appliance must have a system user account with certain privileges so that the Citrix ingress controller can configure the Citrix ADC MPX, or VPX appliance. To create a system user account on Citrix ADC, see [Create a system user account for the Citrix ingress controller in Citrix ADC](#).

  You can directly pass the user name and password as environment variables to the Citrix ingress controller or use OpenShift secrets (recommended). If you want to use OpenShift secrets, create a secret for the user name and password using the following command:

```
oc create secret generic nslogin --from-literal=username='cic' --from-literal=password='mypassword'
```
Citrix ADC ingress controller

Create a system user account for the Citrix ingress controller in Citrix ADC

The Citrix ingress controller configures a Citrix ADC appliance (MPX or VPX) using a system user account of the Citrix ADC appliance. The system user account must have the permissions to configure the following tasks on the Citrix ADC:

- Add, Delete, or View Content Switching (CS) virtual server
- Configure CS policies and actions
- Configure Load Balancing (LB) virtual server
- Configure Service groups
- Configure SSL certkeys
- Configure routes
- Configure user monitors
- Add system file (for uploading SSL testkeys from OpenShift)
- Configure Virtual IP address (VIP)
- Check the status of the Citrix ADC appliance
- Configure SSL actions and policies
- Configure SSL vServer
- Configure responder actions and policies

To create the system user account, perform the following:

1. Log on to the Citrix ADC appliance using the following steps:
   a) Use an SSH client, such as PuTTY, to open an SSH connection to the Citrix ADC appliance.
   b) Log on to the appliance by using the administrator credentials.

2. Create the system user account using the following command:

   ```
   add system user <username> <password>
   ```

   For example:

   ```
   add system user cic mypassword
   ```

3. Create a policy to provide required permissions to the system user account. Use the following command:

   ```
   add cmdpolicy cic-policy ALLOW ^\(^?!shel\(\(?!sftp\(\(?!scp\(\(?!batch\(\(?!source\(\(?!.*superuser\(\(?!.*nsroot\(\(?!install\(\(?!show\(\(?!system\(\(?!user|cmdPolicy|file))\(\(?!(set|add|rm|create|
Deploy the Citrix ingress controller as a pod in an OpenShift cluster

Perform the following steps to deploy the Citrix ingress controller as a pod:

**Step 3**

The command policy mentioned in **step 3** is similar to the built-in `sysAdmin` command policy with another permission to upload files.

In the command policy specification provided, special characters which need to be escaped are already omitted to easily copy-paste into the Citrix ADC command line.

For configuring the command policy from the Citrix ADC configuration wizard (GUI), use the following command policy specification:

```bash
^(!shell)!scp)(!batch)(!source)(!*superuser)(!*nsroot)(!install)(!show)s+system*s+(user|cmdPolicy|file))(!set|add|rm|create|export|kill)s+system*s+(user|group))(!diff|s+ns*s+config)(!set|unset|add|rm|bind|unbind|switch)s+ns*s+partition).*(^install\s*\(wi|wf\))\(^S+s+system\s+file\)
```

4. Bind the policy to the system user account using the following command:

```bash
bind system user cic cic-policy 0
```
1. Download the cic.yaml file using the following command:

   ```
   wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/openshift/manifest/cic.yaml
   ```

2. Edit the cic.yaml file and enter the values for the following environmental variables:

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Mandatory or Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS_IP</td>
<td>Mandatory</td>
<td>The IP address of the Citrix ADC appliance. For more details, see Prerequisites.</td>
</tr>
<tr>
<td>NS_USER and NS_PASSWORD</td>
<td>Mandatory</td>
<td>The user name and password of the Citrix ADC VPX or MPX appliance used as the Ingress device. For more details, see Prerequisites.</td>
</tr>
<tr>
<td>EULA</td>
<td>Mandatory</td>
<td>The End User License Agreement. Specify the value as Yes.</td>
</tr>
<tr>
<td>NS_VIP</td>
<td>Optional</td>
<td>Citrix ingress controller uses the IP address provided in this environment variable to configure a virtual IP address to the Citrix ADC that receives Ingress traffic. Note: NS_VIP acts as a fallback when the frontend-ip annotation is not provided in Ingress or Route yaml. Not supported for Type Loadbalancer service.</td>
</tr>
</tbody>
</table>

3. Add the service account to privileged security context constraints (SCC) of OpenShift.

   ```
   oc adm policy add-scc-to-user privileged system:serviceaccount:default:citrix
   ```

4. Save the edited cic.yaml file and deploy it using the following command:
5. Verify if the Citrix ingress controller is deployed successfully using the following command:

```
oc create -f cic.yaml
```

6. Configure static routes on Citrix ADC VPX or MPX to reach the pods inside the OpenShift cluster.
   a) Use the following command to get the information about host names, host IP addresses, and subnets for static route configuration.

```
oc get hostsubnet
```

   b) Log on to the Citrix ADC instance.

   c) Add the route on the Citrix ADC instance using the following command.

```
add route <pod_network> <netmask> <gateway>
```

Example:
```
oc get hostsubnet
```

```
NAME    HOST      HOST IP       SUBNET
os.example.com  os.example.com  192.168.122.46  192.1.1.0/24
```

From the output of the `oc get hostsubnet` command:

- `<pod_network> = 192.1.1.0`
- Value for subnet = 192.1.1.0/x where x = 24 that means `<netmask> = 255.255.255.0`
- `<gateway> = 192.168.122.46`

The required **static** route to add on Citrix ADC is as follows:
```
add route 10.1.1.0 255.255.255.0 192.168.122.46
Example: Deploy the Citrix ingress controller as a router plug-in in an OpenShift cluster

In this example, the Citrix ingress controller is deployed as a router plug-in in the OpenShift cluster to load balance an application.

1. Deploy a sample application (apache.yaml) in your OpenShift cluster and expose it as a service in your cluster using the following command.


   **Note:**  
   When you deploy a normal Apache pod in OpenShift, it may fail as Apache pod always runs as a root pod. OpenShift has strict security checks which block running a pod as root or binding to port 80. As a workaround, you can add the default service account of the pod to the privileged security context of OpenShift by using the following commands:

   1. `oc adm policy add-scc-to-user privileged system:serviceaccount:default:default`
   2. `oc adm policy add-scc-to-group anyuid system:authenticated`

   The content of the apache.yaml file is given as follows.

   ```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    name: apache-only-http
spec:
  replicas: 4
  selector:
    matchLabels:
      app: apache-only-http
  template:
    metadata:
      labels:
        app: apache-only-http
    spec:
```
containers:
  - image: raghulc/apache-multiport-http:1.0.0
    imagePullPolicy: IfNotPresent
    name: apache-only-http
    ports:
      - containerPort: 80
        protocol: TCP
      - containerPort: 5080
        protocol: TCP
      - containerPort: 5081
        protocol: TCP
      - containerPort: 5082
        protocol: TCP

---

apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    name: apache-only-ssl
  name: apache-only-ssl
spec:
  replicas: 4
  selector:
    matchLabels:
      app: apache-only-ssl
  template:
    metadata:
      labels:
        app: apache-only-ssl
      name: apache-only-ssl
spec:
  containers:
    - image: raghulc/apache-multiport-ssl:1.0.0
      imagePullPolicy: IfNotPresent
      name: apache-only-ssl
      ports:
        - containerPort: 443
          protocol: TCP
        - containerPort: 5443
          protocol: TCP
        - containerPort: 5444
          protocol: TCP
        - containerPort: 5445
          protocol: TCP

---

apiVersion: v1
2. Deploy the Citrix ingress controller for Citrix ADC VPX as a stand-alone pod in the OpenShift cluster using the steps in Deploy the Citrix ingress controller as a pod.

```
oc create -f cic.yaml
```

**Note:**
To deploy the Citrix ingress controller with Citrix ADC CPX in the OpenShift cluster, perform the
steps in Deploy the Citrix ingress controller as a sidecar with Citrix ADC CPX.

3. Create an OpenShift route for exposing the application.
   - For creating an unsecured OpenShift route (unsecured-route.yaml), use the following command:

```
1 oc create -f unsecured-route.yaml
```

- For creating a secured OpenShift route with edge termination (secured-edge-route.yaml), use the following command.

```
1 oc create -f secured-route-edge.yaml
```

- For creating a secured OpenShift route with passthrough termination (secured-passthrough-route.yaml), use the following command.

```
1 oc create -f secured-passthrough-route.yaml
```

- For creating a secured OpenShift route with re-encryption termination (secured-reencrypt-route.yaml), use the following command.

```
1 oc create -f secured-reencrypt-route.yaml
```

To see the contents of the YAML files for OpenShift routes in this example, see YAML files for routes.

**Note:**

For a secured OpenShift route with passthrough termination, you must include the default certificate.

4. Access the application using the following command.

```
1 curl http://<VIP of Citrix ADC VPX>/ -H 'Host: < host-name-as-per-the-host-configuration-in-route >'
2 <!--NeedCopy-->
```
YAML files for routes

This section contains YAML files for unsecured and secured routes specified in the example.

Note:
Keys used in this example are for testing purpose only. You must create your own keys for the actual deployment.

The contents of the unsecured-route.yaml file is given as follows:

```yaml
apiVersion: v1
kind: Route
metadata:
  name: unsecured-route
spec:
  host: unsecured-route.openshift.citrix-cic.com
  path: "/"
  to:
    kind: Service
    name: svc-apache-multi-http
<!-- NeedCopy-->
```

See, secured-edge-route for the contents of the secured-edge-route.yaml file.

The contents of the secured-passthrough-route is given as follows:

```yaml
apiVersion: v1
kind: Route
metadata:
  name: secured-passthrough-route
spec:
  host: secured-passthrough-route.openshift.citrix-cic.com
  to:
    kind: Service
    name: svc-apache-multi-ssl
    tls:
      termination: passthrough
<!-- NeedCopy-->
```

See, secured-reencrypt-route for the contents of the secured-reencrypt-route.yaml file.
Deploy the Citrix ingress controller with OpenShift router sharding support

February 4, 2022

OpenShift router sharding allows distributing a set of routes among multiple OpenShift routers. By default, an OpenShift router selects all routes from all namespaces. In router sharding, labels are added to routes or namespaces and label selectors to routers for filtering routes. Each router shard selects only routes with specific labels that match its label selection parameters.

Citrix ADC can be integrated with OpenShift in two ways and both deployments support OpenShift router sharding.

- Citrix ADC CPX deployed as an OpenShift router along with Citrix ingress controller inside the cluster
- Citrix ingress controller as a router plug-in for Citrix ADC MPX or VPX deployed outside the cluster

To configure router sharding for a Citrix ADC deployment on OpenShift, a Citrix ingress controller instance is required per shard. The Citrix ingress controller instance is deployed with route or namespace labels or both as environment variables depending on the criteria required for sharding.

When the Citrix ingress controller processes a route, it compares the route’s labels or route’s namespace labels with the selection criteria configured on it. If the route satisfies the criteria, the appropriate configuration is applied to Citrix ADC, otherwise it does not apply the configuration.

In router sharding, selecting a subset of routes from the entire pool of routes is based on selection expressions. Selection expressions are a combination of multiple values and operations.

For example, consider there are some routes with various labels for service level agreement (sla), geographical location (geo), hardware requirements (hw), department (dept), type, and frequency as shown in the following table.

<table>
<thead>
<tr>
<th>Label</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>sla</td>
<td>high, medium, low</td>
</tr>
<tr>
<td>geo</td>
<td>east, west</td>
</tr>
<tr>
<td>hw</td>
<td>modest, strong</td>
</tr>
<tr>
<td>dept</td>
<td>finance, dev, ops</td>
</tr>
<tr>
<td>type</td>
<td>static, dynamic</td>
</tr>
<tr>
<td>frequency</td>
<td>high, weekly</td>
</tr>
</tbody>
</table>

The following table shows selectors for route labels or namespace labels and a few sample selection expressions.
Citrix ADC ingress controller

expressions based on labels in the example. Route selection criteria is configured on the Citrix ingress
troller by using environment variables ROUTE_LABELS and NAMESPACE_LABELS.

<table>
<thead>
<tr>
<th>Type of selector</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR operation</td>
<td>ROUTE_LABELS='dept in (dev, ops)'</td>
</tr>
<tr>
<td>AND operation</td>
<td>ROUTE_LABELS='hw=strong,type=dynamic,geo=west'</td>
</tr>
<tr>
<td>NOT operation</td>
<td>ROUTE_LABELS='dept!= finance'</td>
</tr>
<tr>
<td>Exact match</td>
<td>NAMESPACE_LABELS='frequency=weekly'</td>
</tr>
<tr>
<td>Exact match with both route and namespace labels</td>
<td>NAMESPACE_LABELS='frequency=weekly' ROUTE_LABELS='sla=low'</td>
</tr>
<tr>
<td>Key based matching independent of value</td>
<td>NAMESPACE_LABELS='name'</td>
</tr>
<tr>
<td>NOT operation with key based matching independent of value</td>
<td>NAMESPACE_LABELS='!name'</td>
</tr>
</tbody>
</table>

**Note:**
The label selectors use the language supported by Kubernetes labels.

If you want, you can change route or namespace labels by editing them later. Once you change the
labels, router shard is revalidated and based on the change the Citrix ingress controller updates the
configuration on Citrix ADC.

**Deploy Citrix ADC CPX with OpenShift router sharding**

To deploy CPX with OpenShift router sharding support, perform the following steps:

1. Download the `cpx_cic_side_car.yaml` file using the following command:

   ```
   wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/openshift/manifest/cpx_cic_side_car.yaml
   ```

2. Edit the `cpx_cic_side_car.yaml` file and specify the route labels and namespace label selectors as
   environment variables.

   The following example shows how to specify a sample route label and namespace label in the
   `cpx_cic_side_car.yaml` file. This example selects routes with label “name” values as either
   abc or xyz and with namespace label as frequency=high.
Deploy the Citrix ingress controller router plug-in with OpenShift router sharding support

To deploy a Citrix ingress controller router plug-in with router sharding, perform the following steps:

1. Download the cic.yaml file using the following command:

   ```
   wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/openshift/manifest/cic.yaml
   ```

2. Edit the cic.yaml file and specify the route labels and namespace label selectors as environment variables.

   The following example shows how to specify a sample route label and namespace label in the cic.yaml file. This example selects routes with label “name” values as either abc or xyz and with namespace label as frequency=high.

   ```
   env:
   - name: "ROUTE_LABELS"
     value: "name in (abc,xyz)"
   - name: "NAMESPACE_LABELS"
     value: "frequency=high"
   ```

3. Deploy the Citrix ingress controller using the following command.

   ```
   oc create -f cic.yaml
   ```
**Example: Create an OpenShift route and verify the route configuration on Citrix ADC VPX**

This example shows how to create an OpenShift route with labels and verify the router shard configuration.

In this example, route configuration is verified on a Citrix ADC VPX deployment.

Perform the following steps to create a sample route with labels.

1. Define the route in a YAML file. Following is an example for a sample route named as `route.yaml`.

   ```yaml
   apiVersion: v1
   kind: Route
   metadata:
     name: web-backend-route
     namespace: default
     labels:
       sla: low
       name: abc
   spec:
     host: web-frontend.cpx-lab.org
     path: "/web-backend"
     port:
       targetPort: 80
     to:
       kind: Service
       name: web-backend
   
   1 oc create -f route.yaml
   
   2. Use the following command to deploy the route.
   
   ```

2. Add labels to the namespace where you create the route.

   ```bash
   oc label namespace default frequency=high
   
   3. Add labels to the namespace where you create the route.
   
   ```

**Verify route configuration**

You can verify the OpenShift route configuration on a Citrix ADC VPX by performing the following steps:
1. Log on to Citrix ADC VPX by performing the following:
   - Use an SSH client such as PuTTy, to open an SSH connection to Citrix ADC VPX.
   - Log on to Citrix ADC VPX by using administrator credentials.

2. Check if the service group is created using the following command.

   ```
   show serviceGroup
   ```

3. Verify the route configuration on Citrix ADC VPX in the `show serviceGroup` command output. Following is a sample route configuration from the `show serviceGroup` command output.

   ```
   > show serviceGroup
   k8s-web-backend-route_default_80_k8s-web-backend_default_80_svc - HTTP
   State: ENABLED   Effective State: DOWN   Monitor Threshold : 0
   Max Conn: 0   Max Req: 0   Max Bandwidth: 0 kbits
   Use Source IP: NO
   Client Keepalive(CKA): NO
   TCP Buffering(TCPB): NO
   HTTP Compression(CMP): NO
   Idle timeout: Client: 180 sec   Server: 360 sec
   Client IP: DISABLED
   Cacheable: NO
   SC: OFF
   SP: OFF
   Down state flush: ENABLED
   Monitor Connection Close : NONE
   Appflow logging: ENABLED
   ContentInspection profile name: ???
   Process Local: DISABLED
   Traffic Domain: 0
   ```

**Deploy the Citrix ingress controller using Red Hat OpenShift Operators**

July 5, 2022

An Operator is an open source toolkit designed to package, deploy, and manage Kubernetes native applications in a more effective, automated, and scalable way.

This topic covers information on how to deploy the Citrix ingress controller using OpenShift Operators.
**Deployment options**

Based on how you want to use Citrix ADC, there are two ways to deploy the Citrix ingress controller in an OpenShift cluster:

- As a standalone pod in the Kubernetes cluster: In this mode, you can control the Citrix ADC MPX or VPX appliance deployed outside the cluster.
- As a sidecar container alongside Citrix ADC CPX in the same pod: In this mode, Citrix ingress controller configures the Citrix ADC CPX.

**Deploy the Citrix ingress controller as a standalone pod in the OpenShift cluster for Citrix ADC MPX or VPX appliances**

Using the Citrix ingress controller Operator you can deploy the Citrix ingress controller as a standalone pod in the OpenShift cluster. The Citrix Ingress controller configures the Citrix ADC VPX or MPX which is deployed as an Ingress or router for an application running in the OpenShift cluster. The following diagram explains the topology:

**Prerequisites**

- Deployed Red Hat OpenShift version 4.1 or later.
- Determine the NS.IP IP address needed by the controller to communicate with the appliance. The IP address might be anyone of the following depending on the type of Citrix ADC deployment:
Citrix ADC ingress controller

- (Standalone appliances) NSIP - The management IP address of a standalone Citrix ADC appliance. For more information, see IP Addressing in Citrix ADC
- (Appliances in High Availability mode) SNIP - The subnet IP address. For more information, see IP Addressing in Citrix ADC
- (Appliances in Clustered mode) CLIP - The cluster management IP (CLIP) address for a clustered Citrix ADC deployment. For more information, see IP addressing for a cluster

- The user name and password of the Citrix ADC VPX or MPX appliance used as the Ingress device. The Citrix ADC appliance must have a system user account (non-default) with certain privileges so that the Citrix ingress controller can configure the Citrix ADC VPX or MPX appliance. For instructions to create the system user account on Citrix ADC, see Create System User Account for Citrix ingress controller in Citrix ADC

You can directly pass the user name and password as environment variables to the controller, or use Kubernetes secrets (recommended). If you want to use Kubernetes secrets, create a secret for the user name and password using the following command:

```
kubectl create secret generic nslogin --from-literal=username='cic' --from-literal=password='mypassword'
```

Specify the allowed image registries to access by operators and pods

You must specify the image registries to which operators and pods should have access for pulling the images. You can specify the allowed list of registries by updating the `image.config.openshift.io/cluster` custom resource definition file. Ensure to bring up the Openshift cluster before updating the `image.config.openshift.io/cluster` custom resource definition.

To update the `image.config.openshift.io/cluster` custom resource definition, perform the following steps:

1. Run the following command:

```
oc edit image.config.openshift.io/cluster
```

2. Edit the `spec` field such as follows:

```
spec:
  registrySources:
  allowedRegistries:
    - quay.io
```
The Citrix ingress controller configures the Citrix ADC appliance (MPX or VPX) using a system user account of the Citrix ADC. The system user account should have certain privileges so that the Citrix ingress controller has permission to configure the following on the Citrix ADC:

- Add, Delete, or View Content Switching (CS) virtual server
- Configure CS policies and actions
- Configure Load Balancing (LB) virtual server
- Configure Service groups
- Configure SSL certkeys
- Configure routes
- Configure user monitors
- Add system file (for uploading SSL certkeys from Kubernetes)
- Configure Virtual IP address (VIP)
- Check the status of the Citrix ADC appliance

To create the system user account, perform the following:

1. Log on to the Citrix ADC appliance. Perform the following:
   a) Use an SSH client, such as PuTTY, to open an SSH connection to the Citrix ADC appliance.
   b) Log on to the appliance by using the administrator credentials.
2. Create the system user account using the following command:

```
add system user <username> <password>
```
3. Create a policy to provide required permissions to the system user account. Use the following command:

```plaintext
1 add cmdpolicy cic-policy ALLOW "^(!shell)\(|!sftp)\(|!scp)\(|!batch)\(|!source)\(|!.*superuser)\(|!.*nsroot)\(|!install)\(|!show\s+system\s+(user|cmdPolicy|file))\(|!(set|add|rm|create|export|kill)\s+system)\(|!(unbind|bind)\s+system\s+(user|group)\)\(|!diff\s+ns\s+config)\(|!(set|unset|add|rm|bind|unbind|switch)\s+ns\s+partition)\s+\(\^\s+\s+(user|group)\)\(|!(unbind|bind)\s+system\s+(user|group)\)\(|!\s+\s+config)\(|!(set|unset|add|rm|bind|unbind|switch)\s+ns\s+partition)\s+\(\^\s+\s+(user|group)\)\(|!\s+\s+config)\(|!(set|unset|add|rm|bind|unbind|switch)\s+ns\s+partition)\s+\(\^\s+\s+(user|group)\)
```

**Note:** The system user account would have privileges based on the command policy that you define.

The command policy mentioned in **step 3** is similar to the built-in `sysAdmin` command policy with another permission to upload files.

In the command policy specification provided, special characters which need to be escaped are already omitted to easily copy-paste into the Citrix ADC command line.

For configuring the command policy from the Citrix ADC configuration wizard (GUI), use the following command policy specification.

```plaintext
1 ^(!shell)\(|!sftp)\(|!scp)\(|!batch)\(|!source)\(|!.*superuser)\(|!.*nsroot)\(|!install)\(|!show\s+system\s+(user|cmdPolicy|file))\(|!(set|add|rm|create|export|kill)\s+system)\(|!(unbind|bind)\s+system\s+(user|group)\)\(|!diff\s+ns\s+config)\(|!(set|unset|add|rm|bind|unbind|switch)\s+ns\s+partition)\s+\(\^\s+\s+(user|group)\)
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
**Deploy the Citrix ingress controller as a standalone pod using Operators**

Perform the following:

1. Log on to OpenShift 4.1 Cluster console.
2. Deploy an Apache application using the console. Perform the following:
   a) Navigate to **Workloads > Deployments > Create Deployment** and use the `apache.yaml` to create the deployment.

   NOTE:
   The Apache application is for the demonstration purpose only, you can modify the YAML file based on your requirement.

   b) Navigate to **Workloads > Pods** section and ensure that the Apache application pods are up and running.
3. Create a service for the Apache application. Navigate to Networking > Services > Create Service and use the apache-service.yaml to create the service.
4. Create an ingress for the Apache application. Navigate to **Networking > Ingress > Create Ingress** and use the `apache-ingress-vpx.yaml` to create the ingress. Ensure that you update VIP of the Citrix ADC VPX in the ingress YAML before applying it in the cluster.
5. Navigate to **Catalog > OperatorHUB**, select the **Citrix Ingress Controller Operator**, and click **Install**.
You have the following options to subscribe to the Citrix ingress Controller Operator:

- **All namespaces on the cluster (default)** - Allows the Citrix ingress controller operator to subscribe to every namespace present on the OpenShift cluster and hence allows you to initiate the Citrix ingress controller from any namespace on the cluster.

- **A specific namespace on the cluster** - Allows the Citrix ingress controller operator to subscribe to the selected namespace on the OpenShift cluster. You can initiate the Citrix ingress controller instance on the selected namespace only.

For the demonstration purpose, you can subscribe the Citrix ingress controller operator to the
default namespace.

6. Navigate to Workloads > Pods section and verify that the cic-operator pod is up and running.

Wait until the Citrix ingress controller operator is subscribed successfully.
7. Navigate to Catalog > Installed Operators and select the Citrix ingress controller operator.

8. Click Citrix Ingress Controller and in the Overview tab, select Create New on the option to create the CRD for the Citrix ingress controller operator.
The Citrix ingress controller YAML definition is displayed.
The following table lists the mandatory and optional parameters and their default values that you can configure also during installation. Ensure that you set the `license.accept` parameter to `yes` and provide the IP address of the Citrix ADC VPX instance in the `nsIP` parameter.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Mandatory or Optional</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>license.accept</td>
<td>Mandatory</td>
<td>no</td>
<td>Set <code>yes</code> to accept the Citrix ingress controller end user license agreement.</td>
</tr>
<tr>
<td>cic.image</td>
<td>Mandatory</td>
<td><code>quay.io/citrix/citrix-k8s-ingress-controller:1.26.7</code></td>
<td>The Citrix ingress controller image.</td>
</tr>
<tr>
<td>cic.pullPolicy</td>
<td>Mandatory</td>
<td>Always</td>
<td>The Citrix ingress controller image pull policy.</td>
</tr>
<tr>
<td>loginFileName</td>
<td>Mandatory</td>
<td>nslogin</td>
<td>The secret key to log on to the Citrix ADC VPX or MPX. For information on how to create the secret keys, see Prerequisites.</td>
</tr>
<tr>
<td>nsIP</td>
<td>Mandatory</td>
<td>N/A</td>
<td>The IP address of the Citrix ADC device. For details, see Prerequisites.</td>
</tr>
<tr>
<td>nsVIP</td>
<td>Optional</td>
<td>N/A</td>
<td>The virtual IP address on the Citrix ADC device.</td>
</tr>
<tr>
<td>nsPort</td>
<td>Optional</td>
<td>443</td>
<td>The port used by the Citrix ingress controller to communicate with Citrix ADC. You can port 80 for HTTP.</td>
</tr>
</tbody>
</table>
### Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Mandatory or Optional</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nsProtocol</td>
<td>Optional</td>
<td>HTTPS</td>
<td>The protocol used by the Citrix ingress controller to communicate with Citrix ADC. You can also use HTTP on port 80.</td>
</tr>
<tr>
<td>logLevel</td>
<td>Optional</td>
<td>DEBUG</td>
<td>The log level to control the logs generated by the Citrix ingress controller. The supported log levels are: CRITICAL, ERROR, WARNING, INFO, and DEBUG. For more information, see <a href="#">Log Levels</a>.</td>
</tr>
<tr>
<td>kubernetesURL</td>
<td>Optional</td>
<td>N/A</td>
<td>The kube-apiserver url that the Citrix ingress controller uses to register the events. If the value is not specified, the Citrix ingress controller uses the <strong>internal</strong> kube-apiserver IP address.</td>
</tr>
<tr>
<td>Parameters</td>
<td>Mandatory or Optional</td>
<td>Default value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ingressClass</td>
<td>Optional</td>
<td>N/A</td>
<td>If multiple ingress load balancers are used to load balance different ingress resources. You can use this parameter to specify the Citrix ingress controller to configure Citrix ADC associated with specific ingress class. For more information on Ingress class, see Ingress class support.</td>
</tr>
<tr>
<td>nodeWatch</td>
<td>Optional</td>
<td>false</td>
<td>Use the argument if you want to automatically configure the network route from the Ingress Citrix ADC VPX or MPX to the pods in the Kubernetes cluster. For more information, see Automatically configure a route on the Citrix ADC instance.</td>
</tr>
<tr>
<td>defaultSSLCert</td>
<td>Optional</td>
<td>N/A</td>
<td>Default SSL certificate that must be used as a non-SNI certificate in Citrix ADC.</td>
</tr>
</tbody>
</table>
## Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Mandatory or Optional</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exporter.required</td>
<td>Optional</td>
<td>false</td>
<td>Use the argument, if you want to run the <a href="https://github.com/citrix/citrix-adc-metrics-exporter">Exporter for Citrix ADC Stats</a> along with the Citrix ingress controller to pull metrics for the Citrix ADC VPX or MPX.</td>
</tr>
<tr>
<td>exporter.image</td>
<td>Optional</td>
<td>quay.io/citrix/citrix-adc-metrics-exporter:1.4.6</td>
<td>The image of Exporter for Citrix ADC Stats.</td>
</tr>
<tr>
<td>exporter.pullPolicy</td>
<td>Optional</td>
<td>Always</td>
<td>The pull policy for the Exporter for Citrix ADC Stats image.</td>
</tr>
<tr>
<td>exporter.ports.containerPort</td>
<td>Optional</td>
<td>8888</td>
<td>The container port of the Exporter for Citrix ADC Stats.</td>
</tr>
<tr>
<td>openshift</td>
<td>Optional</td>
<td>true</td>
<td>Set this argument if OpenShift environment is being used.</td>
</tr>
</tbody>
</table>

After you have updated the values of the required parameters, click **Create**.

9. Navigate to **Workloads > Pods** section and ensure that the `citrix-ingress-controller` pod is up and running.
10. Verify the deployment by sending traffic as shown:

```bash
2  
3  It works!
```

Deploy the Citrix ingress controller as a sidecar with Citrix ADC CPX

Using the Citrix ingress controller Operator you can deploy a Citrix ADC CPX with the Citrix ingress controller as a sidecar. The Citrix Ingress controller configures the Citrix ADC CPX which is deployed as an Ingress or router for an application running in the OpenShift cluster. The following diagram explains the topology.

Prerequisites

- Deployed Red Hat OpenShift version 4.1 or later.
- Installed the Prometheus Operator, if you want to view the metrics of the Citrix ADC CPX collected by the Citrix ADC metrics exporter.

Deploy the Citrix ingress controller as a sidecar with Citrix ADC CPX using Operators

Perform the following:

1. Log on to OpenShift 4.1 Cluster console.
2. Deploy an Apache application using the console. Perform the following:

   a) Navigate to **Workloads > Deployments > Create Deployment** and use the **apache.yaml** to create the deployment.

   b) Navigate to **Workloads > Pods** section and ensure that the Apache application pods are up and running.

3. Create a service for the Apache application. Navigate to **Networking > Services > Create Service** and use the **apache-service.yaml** to create the service.
4. Create an Ingress for the Apache application. Navigate to Networking > Ingress > Create Ingress and use the `apache-ingress-cpx.yaml` to create the ingress.
5. Navigate to Catalog > OperatorHUB, select the Citrix ADC CPX with Ingress Controller Operator and click Install.
You have the following options to subscribe to the Citrix ingress Controller Operator:

- **All namespaces on the cluster (default)** - Allows the Citrix ingress controller operator to subscribe to every namespace present on the OpenShift cluster and hence allows you to initiate the Citrix ADC CPX from any namespace on the cluster.

- **A specific namespace on the cluster** - Allows the Citrix ingress controller operator to subscribe to the selected namespace on the OpenShift cluster. You can initiate the Citrix ADC CPX instance on the selected namespace only.

For the demonstration purpose, you can subscribe the Citrix ingress controller operator to the default namespace.

Wait until the operator is subscribed successfully.
6. Navigate to **Workloads > Pods** section and verify that the **cic-operator** pod is up and running.

7. Navigate to **Catalog > Installed Operators** and select the Citrix ingress controller operator.

8. Click Citrix ADC CPX with Ingress Controller and in the **Overview** tab, select **Create New** on the option to create the CRD for the Citrix ADC CPX with the Ingress Controller operator.
The Citrix ADC CPX with ingress controller YAML definition is displayed.

```yaml
apiVersion: charts.helm.k8s.io/v1alpha1
kind: CitrixCPXIngressController
metadata:
  name: cpx-cic
  namespace: default
spec:
  image: "quay.io/citrix/citrix-k8s-ingress-controller:1.2.0"
  pullPolicy: Always
  required: true
  cpx:
    image: "quay.io/citrix/citrix-k8s-cpx-ingress:13.0-36.29"
    pullPolicy: Always
    defaultSSLCert: null
    exporter:
      image: "quay.io/citrix/netscaler-metrics-exporter:v1.0.4"
      ports:
        - containerPort: 8888
        pullPolicy: Always
      required: false
      ingressClass: null
      license:
        accept: "no"
        lsIP: null
        lsPort: null
        openShift: true
        platform: null
```
The following table lists the mandatory and optional parameters and their default values that you can configure during installation. Ensure that you set the `license.accept` parameter to `yes`.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Mandatory or Optional</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>license.accept</td>
<td>Mandatory</td>
<td>no</td>
<td>Set <code>yes</code> to accept the Citrix ingress controller end user license agreement.</td>
</tr>
<tr>
<td>cpx.image</td>
<td>Mandatory</td>
<td>quay.io/citrix/citrix-k8s-cpx-ingress:13.0-83.27</td>
<td>The Citrix ADC CPX image.</td>
</tr>
<tr>
<td>cpx.pullPolicy</td>
<td>Mandatory</td>
<td>Always</td>
<td>The Citrix ADC CPX image pull policy.</td>
</tr>
<tr>
<td>lsIP</td>
<td>Optional</td>
<td>N/A</td>
<td>Provide the Citrix Application Delivery Management (ADM) IP address to license Citrix ADC CPX. For more information, see Licensing.</td>
</tr>
<tr>
<td>lsPort</td>
<td>Optional</td>
<td>27000</td>
<td>Citrix ADM port if a non-default port is used.</td>
</tr>
<tr>
<td>platform</td>
<td>Optional</td>
<td>N/A</td>
<td>Platform license. The platform is CP1000.</td>
</tr>
<tr>
<td>cic.image</td>
<td>Mandatory</td>
<td>quay.io/citrix/citrix-k8s-ingress-controller:1.21.9</td>
<td>The Citrix ingress controller image.</td>
</tr>
<tr>
<td>cic.pullPolicy</td>
<td>Mandatory</td>
<td>Always</td>
<td>The Citrix ingress controller image pull policy.</td>
</tr>
</tbody>
</table>
## Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Mandatory or Optional</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cic.required</td>
<td>Optional</td>
<td>true</td>
<td>Specifies this parameter to run the Citrix ingress controller as a sidecar with Citrix ADC CPX</td>
</tr>
<tr>
<td>defaultSSLCert</td>
<td>Optional</td>
<td>N/A</td>
<td>Default SSL certificate that must be used as a non-SNI certificate in Citrix ADC.</td>
</tr>
<tr>
<td>exporter.required</td>
<td>Optional</td>
<td>false</td>
<td>Use the argument if you want to run the <a href="https://example.com">Exporter for Citrix ADC Stats</a> along with the Citrix ingress controller to pull metrics for the Citrix ADC CPX.</td>
</tr>
<tr>
<td>exporter.image</td>
<td>Optional</td>
<td>quay.io/citrix/citrix-adc-metrics-exporter:1.4.6</td>
<td>The image for the Exporter for Citrix ADC Stats.</td>
</tr>
<tr>
<td>exporter.pullPolicy</td>
<td>Optional</td>
<td>Always</td>
<td>The image pull policy for the Exporter for Citrix ADC Stats.</td>
</tr>
<tr>
<td>exporter.ports.containerPort</td>
<td>Optional</td>
<td>8888</td>
<td>The container port of Exporter for the Citrix ADC Stats.</td>
</tr>
</tbody>
</table>
# Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Mandatory or Optional</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ingressClass</td>
<td>Optional</td>
<td>N/A</td>
<td>If multiple ingress load balancers are used to load balance different ingress resources. You can use this parameter to specify the Citrix ingress controller to configure Citrix ADC associated with specific ingress class.</td>
</tr>
<tr>
<td>openshift</td>
<td>Optional</td>
<td>true</td>
<td>Set this argument if OpenShift environment is being used.</td>
</tr>
</tbody>
</table>

After you have updated the values of the required parameters, click **Create**.

9. Navigate to **Workloads > Pods** section and ensure that the `cpx-ingress` pod is up and running.

10. Verify the deployment by sending traffic. Perform the following:

    a) Obtain the NodePort details using the following command:

        ```bash
        oc get svc
        ```
Citrix ADC ingress controller

b) Use `cpx-service` NodePort and send the traffic as shown in the following command:

```
2. <html><body><h1>It works!</h1></body></html>
```

Deploy Citrix ADC CPX as an Ingress device in an Azure Kubernetes Service cluster

February 7, 2022

This topic explains how to deploy Citrix ADC CPX as an ingress device in an Azure Kubernetes Service (AKS) cluster. Citrix ADC CPX supports both the Advanced Networking (Azure CNI) and Basic Networking (Kubenet) mode of AKS.

**Note:**

If you want to use Azure repository images for Citrix ADC CPX or the Citrix ingress controller instead of the default quay.io images, then see Deploy Citrix ADC CPX as an Ingress device in an AKS cluster using Azure repository images.

Deploy Citrix ADC CPX as an ingress device in an AKS cluster

Perform the following steps to deploy Citrix ADC CPX as an ingress device in an AKS cluster.

**Note:**

In this procedure, Apache web server is used as the sample application.

1. Deploy the required application in your Kubernetes cluster and expose it as a service in your cluster using the following command.

```
1. kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/azure/manifest/apache.yaml
```

**Note:**
In this example, *apache.yaml* is used. You should use the specific YAML file for your application.

2. Deploy Citrix ADC CPX as an ingress device in the cluster using the following command.

```bash
kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/azure/manifest/standalone_cpx.yaml
```

3. Create the ingress resource using the following command.

```bash
kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/azure/manifest/cpx_ingress.yaml
```

4. Create a service of type LoadBalancer for accessing the Citrix ADC CPX by using the following command.

```bash
kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/azure/manifest/cpx_service.yaml
```

This command creates an Azure load balancer with an external IP for receiving traffic.

5. Verify the service and check whether the load balancer has created an external IP. Wait for some time if the external IP is not created.

```bash
kubectl get svc
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER-IP</th>
<th>EXTERNAL-IP</th>
<th>PORT(S)</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>ClusterIP</td>
<td>10.0.103.3</td>
<td>none</td>
<td>80/TCP</td>
<td>2m</td>
</tr>
<tr>
<td>cpx</td>
<td>LoadBalancer</td>
<td>10.0.37.255</td>
<td>pending</td>
<td>80:32258/TCP,443:32084/TCP</td>
<td>2m</td>
</tr>
<tr>
<td>Kubernetes</td>
<td>ClusterIP</td>
<td>10.0.0.1</td>
<td>none</td>
<td>443/TCP</td>
<td>22h</td>
</tr>
</tbody>
</table>

6. Once the external IP for the load-balancer is available as follows, you can access your resources using the external IP for the load balancer.
Citrix ADC ingress controller

1. `kubectl get svc`

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER-IP</th>
<th>EXTERNAL-IP</th>
<th>PORT(S)</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>ClusterIP</td>
<td>10.0.103.3</td>
<td>none</td>
<td>80/TCP</td>
<td>3m</td>
</tr>
<tr>
<td>cpx-ingress</td>
<td>LoadBalancer</td>
<td>10.0.37.255</td>
<td>EXTERNAL-IP CREATED</td>
<td>80:32258/TCP, 443:32084/TCP</td>
<td>3m</td>
</tr>
<tr>
<td>Kubernetes</td>
<td>ClusterIP</td>
<td>10.0.0.1</td>
<td>none</td>
<td>443/TCP</td>
<td>22h</td>
</tr>
</tbody>
</table>

**Note:**

The health check for the cloud load-balancer is obtained from the readinessProbe configured in the Citrix ADC CPX deployment yaml file. If the health check fails, you should check the readinessProbe configured for Citrix ADC CPX. For more information, see readinessProbe and external Load balancer.

7. Access the application using the following command.

```
```

**Quick Deploy**

For the ease of deployment, you can just deploy a single all-in-one manifest that would combine the steps explained in the previous topic.

1. Deploy a Citrix ADC CPX ingress with in built Citrix ingress controller in your Kubernetes cluster using the `all-in-one.yaml`.

```
1 kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/azure/manifest/all-in-one.yaml
```

2. Access the application using the following command.

```
```
Deploy Citrix ingress controller in an Azure Kubernetes Service cluster with Citrix ADC VPX

February 3, 2022

This topic explains how to deploy the Citrix ingress controller with Citrix ADC VPX in an Azure Kubernetes Service (AKS) cluster. You can also configure the Kubernetes cluster on Azure VMs and then deploy the Citrix ingress controller with Citrix ADC VPX.

The procedure to deploy for both AKS and Azure VM is the same. However, if you are configuring Kubernetes on Azure VMs you need to deploy the CNI plug-in for the Kubernetes cluster.

Prerequisites

You should complete the following tasks before performing the steps in the procedure.

- Ensure that you have a Kubernetes cluster up and running.

  Note:

  For more information on creating a Kubernetes cluster in AKS, see Guide to create an AKS cluster.

Topology

The following is the sample topology used in this deployment.
Get a Citrix ADC VPX instance from Azure Marketplace

You can create Citrix ADC VPX from the Azure Marketplace. For more information on how to create a Citrix ADC VPX instance from Azure Marketplace, see Get Citrix ADC VPX from Azure Marketplace.

Get the Citrix ingress controller from Azure Marketplace

To deploy the Citrix ingress controller, an image registry should be created on Azure and the corresponding image URL should be used to fetch the Citrix ingress controller image.

For more information on how to create a registry and get the image URL, see Get Citrix ingress controller from Azure Marketplace.

Once a registry is created, the Citrix ingress controller registry name should be attached to the AKS cluster used for deployment.

```
az aks update -n <cluster-name> -g <resource-group-where-aks-deployed> --attach-acr <cic-registry>
```

Deploy Citrix Ingress Controller

Perform the following steps to deploy the Citrix ingress controller.
1. Create Citrix ADC VPX login credentials using Kubernetes secret.

```
kubectl create secret generic nslogin --from-literal=username='<azure-vpx-instance-username>' --from-literal=password='<azure-vpx-instance-password>'
```

**Note:**
The Citrix ADC VPX user name and password should be the same as the credentials set while creating Citrix ADC VPX on Azure.

2. Using SSH, configure a SNIP in the Citrix ADC VPX, which is the secondary IP address of the Citrix ADC VPX. This step is required for the Citrix ADC to interact with pods inside the Kubernetes cluster.

```
add ns ip <snip-vpx-instance-private-ip> <vpx-instance-primary-ip-subnet>
```

- `<snip-vpx-instance-private-ip>` is the dynamic private IP address assigned while adding a SNIP during the Citrix ADC VPX instance creation.
- `<vpx-instance-primary-ip-subnet>` is the subnet of the primary private IP address of the Citrix ADC VPX instance.

To verify the subnet of the private IP address, SSH into the Citrix ADC VPX instance and use the following command.

```
show ip <primary-private-ip-addres>
```

3. Update the Citrix ADC VPX image URL, management IP, and VIP in the Citrix ingress controller YAML file.
   a) Download the Citrix ingress controller YAML file.

```
wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/azure/manifest/azurecic/cic.yaml
```
b) Update the Citrix ingress controller image with the Azure image URL in the `cic.yaml` file.

```yaml
- name: cic-k8s-ingress-controller
  # CIC Image from Azure
  image: "<azure-cic-image-url>"
```

c) Update the primary IP address of the Citrix ADC VPX in the `cic.yaml` in the following field with the primary private IP address of the Azure VPX instance.

```yaml
# Set NetScaler NSIP/SNIP, SNIP in case of HA (mgmt has to be enabled)
- name: "NS_IP"
  value: "X.X.X.X"
```

d) Update the Citrix ADC VPX VIP in the `cic.yaml` in the following field with the private IP address of the VIP assigned during VPX Azure instance creation.

```yaml
# Set NetScaler VIP for the data traffic
- name: "NS_VIP"
  value: "X.X.X.X"
```

4. Once you have configured the Citrix ingress controller with the required values, deploy the Citrix ingress controller using the following command.

```bash
kubectl create -f cic.yaml
```

**Verify the deployment using a sample application**

1. Deploy the required application in your Kubernetes cluster and expose it as a service in your cluster using the following command.
Citrix ADC inge...
You can get your Google account details using the following command.

```
gcloud info | grep Account
```

**Deploy Citrix ADC CPX as an ingress device in Google Cloud Platform**

1. Deploy the required application in your Kubernetes cluster and expose it as a service in your cluster using the following command.

```
kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/gcp/manifest/apache.yaml
```

*Note:*

In this example, `apache.yaml` is used. You should use the specific YAML file for your application.

2. Deploy Citrix ADC CPX as an ingress device in the cluster using the following command.

```
kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/gcp/manifest/standalone_cpx.yaml
```

3. Create the ingress resource using the following command.

```
kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/gcp/manifest/cpx_ingress.yaml
```

4. Create a service of type LoadBalancer for accessing the Citrix ADC CPX by using the following command.

```
kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/gcp/manifest/cpx_service.yaml
```
Note:
This command creates a load balancer with an external IP for receiving traffic.

1. Verify the service and check whether the load balancer has created an external IP. Wait for some time if the external IP is not created.

```
kubectl get svc
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER-IP</th>
<th>EXTERNAL-IP</th>
<th>PORT(S)</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>ClusterIP</td>
<td>10.7.248.216</td>
<td>none</td>
<td>80/TCP</td>
<td>2m</td>
</tr>
<tr>
<td>cpx-ingress</td>
<td>LoadBalancer</td>
<td>10.7.241.6</td>
<td>pending</td>
<td>80:32258/TCP, 443:32084/TCP</td>
<td>2m</td>
</tr>
<tr>
<td>kubernetes</td>
<td>ClusterIP</td>
<td>10.7.240.1</td>
<td>none</td>
<td>443/TCP</td>
<td>22h</td>
</tr>
</tbody>
</table>

2. Once the external IP for the load-balancer is available as follows, you can access your resources using the external IP for the load balancer.

```
kubectl get svc
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Cluster-IP</th>
<th>External IP</th>
<th>Port(s)</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>ClusterIP</td>
<td>10.7.248.216</td>
<td>none</td>
<td>80/TCP</td>
<td>3m</td>
</tr>
<tr>
<td>cpx-ingress</td>
<td>LoadBalancer</td>
<td>10.7.241.6</td>
<td>EXTERNAL-IP CREATED</td>
<td>80:32258/TCP, 443:32084/TCP</td>
<td>3m</td>
</tr>
<tr>
<td>kubernetes</td>
<td>ClusterIP</td>
<td>10.7.240.1</td>
<td>none</td>
<td>443/TCP</td>
<td>22h</td>
</tr>
</tbody>
</table>

Note:
The health check for the cloud load-balancer is obtained from the readinessProbe configured in the Citrix ADC CPX service YAML file. If the health check fails, you should check the readinessProbe configured for Citrix ADC CPX.
For more information, see readinessProbe and external Load balancer.

3. Access the application using the following command.

```
```
Citrix ADC ingress controller

Quick Deploy

For the ease of deployment, you can just deploy a single all-in-one manifest that would combine the steps explained in the previous topic.

1. Deploy a Citrix ADC CPX ingress with in built Citrix ingress controller in your Kubernetes cluster using the all-in-one.yaml.

```bash
kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/gcp/manifest/all-in-one.yaml
```

2. Access the application using the following command.

```bash
```

Note:
To delete the deployment, use the `kubectl delete -f all-in-one.yaml` command.

Deploy the Citrix ingress controller in Anthos

February 4, 2022

Anthos is a hybrid and multi cloud platform that lets you run your applications on existing on-prem hardware or in the public cloud. It provides a consistent development and operation experience for cloud and on-premises environments.

The Citrix ingress controller can be deployed in Anthos GKE on-premises using the following deployment modes:

- Exposing Citrix ADC CPX with the sidecar ingress controller as a service of type `LoadBalancer`.
- Dual-tier Ingress deployment

Expose Citrix ADC CPX as a service of type LoadBalancer in Anthos GKE on-prem

In this deployment, Citrix ADC VPX or MPX is deployed outside the cluster at Tier-1 and Citrix ADC CPX at Tier-2 inside the Anthos cluster similar to a dual-tier deployment. However instead of using Ingress, the Citrix ADC CPX is exposed using the Kubernetes service of type `LoadBalancer`. 
The Citrix ingress controller automates the process of configuring the IP address provided in the `LoadBalancerIP` field of the service specification.

**Prerequisites**

- You must deploy a Tier-1 Citrix ADC VPX or MPX in the same subnet as the Anthos GKE on-prem user cluster.
- You must configure a subnet IP address (SNIP) on the Tier-1 Citrix ADC and Anthos GKE on-prem cluster nodes should be reachable using the IP address.
- To use a Citrix ADC VPX or MPX from a different network, use Citrix Node Controller to enable communication between the Citrix ADC and the Anthos GKE on-prem cluster.
- You must set aside a virtual IP address (VIP) to be used as a Load Balancer IP address.

**Deploy Citrix ADC CPX as service of type LoadBalancer in Anthos GKE on-premises**

Perform the following steps to deploy Citrix ADC CPX as a service of type `LoadBalancer` in Anthos GKE on-premises.

1. Deploy the required application in your Kubernetes cluster and expose it as a service in your cluster using the following command.

   ```
   kubectl --kubeconfig user-cluster-1-kubeconfig create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/anthos/manifest/service-type-lb/apache.yaml
   ```

   **Note:**
   
   In this example, `apache.yaml` is used. You should use the specific YAML file for your application.

2. Deploy Citrix ADC CPX with the sidecar Citrix ingress controller as Tier-2 Ingress device using the `cpx-cic.yaml` file.

   ```
   kubectl --kubeconfig user-cluster-1-kubeconfig create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/anthos/manifest/service-type-lb/cpx-cic.yaml
   ```

3. (Optional) Create a self-signed SSL certificate and a key to be used with the Ingress for TLS configuration.
Citrix ADC ingress controller

```bash
```

**Note:**

If you already have an SSL certificate, you can create a Kubernetes secret using the same. This is just an example command to create a self-signed certificate and also this command assumes the host name of the application to be `anthos-citrix-ingress.com`.

4. Create a Kubernetes secret with the created SSL cert-key pair.

```bash
kubectl --kubeconfig user-cluster-1-kubeconfig create secret tls anthos-citrix --cert=$PWD/anthos-citrix-certificate.crt --key=$PWD/anthos-citrix-certificate.key
```

5. Create an Ingress resource for Tier-2 using the `tier-2-ingress.yaml` file.

```bash
```

6. Create a Kubernetes secret for the Tier-1 Citrix ADC.

```bash
kubectl --kubeconfig user-cluster-1-kubeconfig create secret generic nslogin --from-literal=username='citrix-adc-username' --from-literal=password='citrix-adc-password'
```

7. Deploy the Citrix ingress controller as a Tier-1 ingress controller.

   a) Download the `cic.yaml` file.

   b) Enter the management IP address of Citrix ADC. Update the Tier-1 Citrix ADC’s management IP address in the placeholder `Tier-1-Citrix-ADC-IP` specified in the `cic.yaml` file.

   c) Save and deploy the `cic.yaml` using the following command.
8. Expose Citrix ADC CPX as a Kubernetes service of type LoadBalancer.
   a) Download the cpx-service-type-lb.yaml file.
   b) Edit the YAML file and specify the value of VIP-for-accessing-microservices as the VIP address which is to be used for accessing the applications inside the cluster. This VIP address is the one set aside to be used as a Load Balancer IP address.
   c) Save and deploy the cpx-service-type-lb.yaml file using the following command.

   ```
kubectl --kubeconfig user-cluster-1-kubeconfig create -f cpx.yaml
   ```

9. Update the DNS records with the IP address of VIP-for-accessing-microservices for accessing the microservice. In this example, to access the Apache microservice, you must have the following DNS entry.

   ```
`<VIP-for-accessing-microservices> anthos-citrix-ingress.com`
   ```

10. Use the following command to access the application.

   ```
   ```

   **Note:**

   In this command, --resolve anthos-citrix-ingress.com:443:<VIP-for-accessing-microservices> is used to override the DNS configuration part in step 9 for demonstration purpose.

---

**Clean up the installation: Expose Citrix ADC CPX as service of type LoadBalancer**

To clean up the installation, use the kubectl --kubeconfig delete command to delete each deployment.

To delete the Citrix ADC CPX service deployment (CPX+CIC service) use the following command:
To delete the Tier-2 Ingress object, use the following command.

```bash
kubectl --kubeconfig user-cluster-1-kubeconfig delete -f tier-2-ingress.yaml
```

To delete the Citrix ADC CPX deployment along with the sidecar Citrix ingress controller, use the following command.

```bash
kubectl --kubeconfig user-cluster-1-kubeconfig delete -f cpx-cic.yaml
```

To delete the stand-alone Citrix ingress controller, use the following command.

```bash
kubectl --kubeconfig user-cluster-1-kubeconfig delete -f cic.yaml
```

To delete the Apache microservice, use the following command.

```bash
kubectl --kubeconfig user-cluster-1-kubeconfig delete -f apache.yaml
```

To delete the Kubernetes secret, use the following command.

```bash
kubectl --kubeconfig user-cluster-1-kubeconfig delete secret anthos-citrix
```

To delete the `nslogin` secret, use the following command.

```bash
kubectl --kubeconfig user-cluster-1-kubeconfig delete secret nslogin
```
Dual tier Ingress deployment

In a dual-tier Ingress deployment, Citrix ADC VPX or MPX is deployed outside the Kubernetes cluster (Tier-1) and Citrix ADC CPXs are deployed inside the Kubernetes cluster (Tier-2).

Citrix ADC MPX or VPX devices in Tier-1 proxy the traffic (North-South) from the client to Citrix ADC CPXs in Tier-2. The Tier-2 Citrix ADC CPX then routes the traffic to the microservices in the Kubernetes cluster. The Citrix ingress controller deployed as a standalone pod configures the Tier-1 Citrix ADC. The sidecar Citrix ingress controller in one or more Citrix ADC CPX pods configures the associated Citrix ADC CPX in the same pod.

Prerequisites

- You must deploy a Tier-1 Citrix ADC VPX or MPX in the same subnet as the Anthos GKE on-prem user cluster.
- You must configure a subnet IP address (SNIP) on the Tier-1 Citrix ADC and Anthos GKE on-prem cluster nodes should be reachable using the IP address.
- To use a Citrix ADC VPX or MPX from a different network, use the Citrix Node Controller to enable communication between the Citrix ADC and the Anthos GKE on-prem cluster.
- You must set aside a virtual IP address to be used as a front-end IP address in the Tier-1 Ingress manifest.

Dual-tier Ingress deployment in Anthos GKE on-prem

Perform the following steps to deploy a dual-tier Ingress deployment of Citrix ADC in Anthos GKE on-prem.

1. Deploy the required application in your Kubernetes cluster and expose it as a service in your cluster using the following command.

```bash
```

Note:

In this example, `apache.yaml` is used. You should use the specific YAML file for your application.
2. Deploy Citrix ADC CPX with the Citrix ingress controller as Tier-2 Ingress using the `cpx-cic.yaml` file.

```bash
```

3. Expose Citrix ADC CPX as a Kubernetes service using the `cpx-service.yaml` file.

```bash
```

4. (Optional) Create a self-signed SSL certificate and a key to be used with the Ingress for TLS configuration.

   **Note:**
   
   If you already have an SSL certificate, you can create a Kubernetes secret using the same.

```bash
```

   **Note:**
   
   This is just an example command to create a self-signed certificate and also this command assumes that the hostname of the application to be `anthos-citrix-ingress.com`.

5. Create a Kubernetes secret with the created SSL cert-key pair.

```bash
kubectl --kubeconfig user-cluster-1-kubeconfig create secret tls anthos-citrix --cert=$PWD/anthos-citrix-certificate.crt --key=$PWD/anthos-citrix-certificate.key
```

6. Create an Ingress resource for Tier-2 using the `tier-2-ingress.yaml` file.
7. Create a Kubernetes secret for the Tier-1 Citrix ADC.

```
kubectl --kubeconfig user-cluster-1-kubeconfig create secret generic nslogin --from-literal=username='citrix-adc-username' --from-literal=password='citrix-adc-password'
```

8. Deploy the Citrix ingress controller as a Tier-1 ingress controller.

   a) Download the cic.yaml file.

   b) Enter the management IP address of Citrix ADC. Update the Tier-1 Citrix ADC's management IP address in the placeholder `Tier-1-Citrix-ADC-IP` specified in the cic.yaml file.

   c) Save and deploy the cic.yaml using the following command.

```
kubectl --kubeconfig user-cluster-1-kubeconfig create -f cic.yaml
```

9. Create an Ingress resource for Tier-1 using the tier-1-ingress.yaml file.

   a) Download the tier-1-ingress.yaml file.

   b) Edit the YAML file and replace `VIP-Citrix-ADC` with the VIP address which was set aside.

   c) Save and deploy the tier-1-ingress.yaml file using the following command.

```
kubectl --kubeconfig user-cluster-1-kubeconfig create -f tier-1-ingress.yaml
```

10. Update the DNS records with the IP address of VIP-Citrix-ADC for accessing the microservice. In this example, to access the Apache microservice, you must have the following DNS entry.

```
<VIP-Citrix-ADC> anthos-citrix-ingress.com
```
11. Use the following command to access the application.

```
1 curl -k --resolve anthos-citrix-ingress.com:443:<VIP-Citrix-ADC
> https://anthos-citrix-ingress.com/
2 <html><body><h1>It works!</h1></body></html>
```

**Note:**
In this command, `--resolve anthos-citrix-ingress.com:443:<VIP-for-accessing-microservices>` is used to override the DNS configuration part.

**Clean up the installation: Dual tier Ingress**

To clean up the installation, use the `kubectl --kubeconfig delete` command to delete each deployment.

To delete the Tier-1 Ingress object, use the following command.

```
1 kubectl --kubeconfig user-cluster-1-kubeconfig delete -f tier-1-ingress.yaml
```

To delete the Tier-2 Ingress object, use the following command.

```
1 kubectl --kubeconfig user-cluster-1-kubeconfig delete -f tier-2-ingress.yaml
```

To delete the Citrix ADC CPX deployment along with the sidecar Citrix ingress controller, use the following command.

```
1 kubectl --kubeconfig user-cluster-1-kubeconfig delete -f cpx-cic.yaml
```

To delete the Citrix ADC CPX service deployment, use the following command:

```
1 kubectl --kubeconfig user-cluster-1-kubeconfig delete -f cpx-service.yaml
```
Citrix ADC ingress controller

To delete the stand-alone Citrix ingress controller use the following command:

```
kubectl --kubeconfig user-cluster-1-kubeconfig delete -f cic.yaml
```

To delete the Apache microservice, use the following command.

```
kubectl --kubeconfig user-cluster-1-kubeconfig delete -f apache.yaml
```

To delete the Kubernetes secret, use the following command.

```
kubectl --kubeconfig user-cluster-1-kubeconfig delete secret anthos-citrix
```

To delete the `nslogin` secret, use the following command.

```
kubectl --kubeconfig user-cluster-1-kubeconfig delete secret nslogin
```

Deploy Citrix ADC VPX in active-active high availability in EKS environment using Amazon ELB and Citrix ingress controller

February 15, 2022

The topic covers a solution to deploy Citrix ADC VPX in active-active high availability mode on multiple availability zones in AWS Elastic Container Service (EKS) platform. The solution combines AWS Elastic load balancing (ELB) and Citrix ADC VPX to load balance the Ingress traffic to the microservices deployed in EKS cluster. AWS ELB handles the Layer 4 traffic and the Citrix ADC VPXs provides advanced Layer 7 functionalities such as, advanced load balancing, caching, and content-based routing.

**Solution overview**

A basic architecture of an EKS cluster would include three public subnet and three private subnets deployed across three availability zones as shown in the following diagram:
With the solution, the architecture of the EKS cluster would be as shown in the following diagram:

In the AWS cloud, AWS Elastic Load Balancing handles the Layer 4 TCP connections and load balances the traffic using a flow hash routing algorithm. The ELB can be either Network Load Balancer or a Classic Load Balancer.

AWS ELB listens for incoming connections as defined by its listeners. Each listener forwards a new connection to one of the available Citrix ADC VPX instances. The Citrix ADC VPX instance load balances the traffic to the EKS pods. It also performs other Layer 7 functionalities such as, rewrite policy, responder policy, SSL offloading and so on provided by Citrix ADC VPX.
Citrix ADC ingress controller

A Citrix ingress controller is deployed in the EKS cluster for each Citrix ADC VPX instance. The Citrix ingress controllers are configured with the same ingress class. And, it configures the Ingress objects in the EKS cluster on the respective Citrix ADC VPX instances.

AWS Elastic Load Balancing (ELB) has a DNS name to which an IP address is assigned dynamically. The DNS name can be added as Alias A record for your domain in Route53 to access the application hosted in the EKS cluster.

Deployment process

Perform the following to deploy the solution:

1. Deploy Citrix ADC VPX Instances.
2. Deploy Citrix ingress controller.
3. Set up Amazon Elastic Load Balancing. You can either set up Network Load Balancer or Classic Load Balancer.
4. Verify the solution.

Deploy Citrix ADC VPX instances

Citrix ADC VPX is available as CloudFormation Template. The CloudFormation template deploys an instance of Citrix ADC VPX with single ENI on a given subnet. It also configures the NSIP, VIP, and SNIP for the Citrix ADC VPX instance.

For this solution you need to deploy two instances of Citrix ADC VPX. Deploy the Citrix ADC VPX instances on two availability zones by specifying the same Citrix ADC VPX and different public subnet.

After you deploy the Citrix ADC VPX instances, you can verify the deployment by reviewing the output of the CloudFormation template as shown in the following screenshot. The output must show the various IP addresses (VIP, SNIP, and NSIP) configured for the Citrix VPX instances:

Note:
The CloudFormation template deploys the Citrix ADC VPX instance with primary IP address of the
Citrix ADC ingress controller

Citrix ADC VPX EC2 instance as VIP and secondary IP address as management IP address.

After the Citrix ADC VPX instances are successfully deployed, you must edit the security groups to allow traffic from EKS node group security group. Also, you must change the EKS node group security group to allow traffic from VPX instances.

**Deploy Citrix ingress controller**

Deploy separate instance of Citrix ingress controller for each Citrix ADC VPX instance. Follow the deployment instructions to deploy Citrix ingress controller.

After the Citrix ADC VPX instance is up, you must set up a system user account on the Citrix ADC VPX instances. The system user account is used by Citrix ingress controller to log into the Citrix ADC VPX instances. For instruction to set up the system user account, see Create System User Account for CIC in Citrix ADC.

1. Edit the Citrix ingress controller deployment YAML (`citrix-ingress-controller.yaml`).

   Replace `NS_IP` with the Private NSIP address of the respective Citrix ADC VPX instance. Also, provide the system user account user name and password that you have created on the Citrix ADC VPX instance. Once you edited the `citrix-ingress-controller.yaml` file, deploy the updated YAML file using the following command:

   ```bash
   kubectl apply -f citrix-ingress-controller.yaml
   ```

2. Perform Step 1 on the second Citrix ingress controller instance.

3. Ensure that both the pods are UP and running. Also, verify if Citrix ingress controller is able to connect to the respective Citrix ADC VPX instance using the logs:

   ```bash
   kubectl logs <cic_pod_name>
   ```

After the Citrix ingress controller pods are deployed and running in the EKS cluster. Any, Kubernetes Ingress resource configured with the `citrix` ingress class is automatically configured on both the Citrix ADC VPX instances.

**Setup elastic load balancing**

Depending upon your requirement you can configure any of the following load balancers:

- Network Load Balancers
- Classic Load Balancers
Citrix ADC ingress controller

**Set up network load balancer**

Network Load Balancer (NLB) is a good option for handling TCP connection load balancing. In this solution, NLB is used to accept the incoming traffic and route it to one of the Citrix ADC VPX instances. NLB load balances using the flow hash algorithm based on the protocol, source IP address, source port, destination IP address, destination port, and TCP sequence number.

To set up NLB:

1. Log on to the [AWS Management Console for EC2](https://aws.amazon.com/ec2/).

2. In the left navigation bar, click **Target Group**. Create two different target groups. One target group (Target-Group-80) for routing traffic on port 80 and the other target group (Target-Group-443) for routing traffic on 443 respectively.

3. Create a target group named, *Target-Group-80*. Perform the following:
   a) In the **Target group name** field, enter the target group name as Target-Group-80.
   b) In the **Target type** field, select Instance.
   c) From the **Protocol** list, select TCP.
   d) In the **Port** field, enter 80.
   e) From the **VPC** list, select you VPC where you deployed your EKS cluster.
   f) In the **Health check settings** section, use TCP for health check.
   g) Optional. You can modify the **Advance health check settings** to configure health checks.
4. Create a target group named, **Target-Group-443**. Perform the following:

   a) In the **Target group name** field, enter the target group name as **Target-Group-443**.

   b) In the **Target type** field, select **Instance**.

   c) From the **Protocol** list, select **TCP**.

   d) In the **Port** field, enter **443**.

   e) From the **VCP** list, select your VPC where you deployed your EKS cluster.

   f) In the **Health check settings** section, use TCP for health check.

   g) Optional. You can modify the **Advance health check settings** to configure health checks.

5. Once you have created the target groups, you must register the target instances.

   a) Select the created target group in the list page, click the **Target** tab, and select **edit**.

   b) In the **Instances** tab, select the two Citrix ADC VPX instances and click **Add to registered**.
6. Repeat **Step 5** for the other target group that you have created.

7. Create Network Load Balancer.
   a) In the left navigation bar, select **Load Balancers**, then click **Create Load Balancer**.
   b) In the **Select load balancer type** window, click **Create** in the Network Load balancer panel.

   ![Load Balancer Types](image)

8. In the **Configure Load Balancer** page, do the following:
   a) In the **Name** field, enter a name for the load balancer.
   b) In the **Scheme** field, select **internet-facing**.
   c) In the **Listeners** section, click **Add listener** and add two entries with **TCP** as the load balancer protocol and **80** and **443** as the load balancer port respectively as shown in the following image:

   ![Load Balancer Configuration](image)

   d) In the **Availability Zones** section, select the VPC, availability zones, and subnets where the Citrix ADC VPX instances are deployed.
9. In the **Configure routing** page, do the following:

   a) In the **Target group** list, click **Existing target group**.
   
   b) In the **Name** field, enter **Target-Group-80**.
   
   c) In the **Target type** field, select **Instance**.
   
   d) In the **Protocol** list, select **TCP**.
   
   e) In the **Port** field, enter **80**.
   
   f) Select **TCP** from the **Protocol** list in the **Health checks** section as shown in the following image:

10. In the **Review** page, review your configuration and click **Create**.
11. After the Network load balancer is created, select the load balancer that you have created for the list page. Select **Listeners** tab, select **TCP : 444** and then click **Edit**.

12. In the **Listeners** page, delete the default action and then select **Target-Group-443** in the **Forward to** list.

13. Click **Update**.
Citrix ADC ingress controller

Set up classic load balancer

Alternative to Amazon Network load balancer, you can set up Classic Load Balancer (CLB) as Tier 1 TCP load balancer.

1. Log on to the AWS Management Console for EC2.

2. In the left navigation bar, select Load Balancers, then click Create Load Balancer.

3. In the Select load balancer type window, click Create on the Classic Load balancer panel.

4. In the Define Load Balancer page, do the following:
   a) In the Load Balancer name field, enter a name for the load balancer.
   b) In the Create LB Inside list, select your Citrix ADC VPX.
   c) In the Listener Configuration section, click Add and add two entries with TCP as the load balancer protocol and 80 and 443 as the load balancer port respectively. Also, select TCP as instance protocol and 80 and 443 as the instance port respectively as shown in the
d) In the **Select Subnets** section, select two public subnets in two different availability zones for the Classic Load balancer to route the traffic. These subnets are same as where you have deployed the Citrix ADC VPX instances.

**Select Subnets**
You will need to select a Subnet for each Availability Zone where you wish traffic to be routed by your load balancer. If you have Instances in only one Availability Zone, please select at least two Subnets in different Availability Zones to create highest availability for your load balancer.

**Available subnets**

<table>
<thead>
<tr>
<th>Actions</th>
<th>Subnet ID</th>
<th>Availability Zone</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>.v2</td>
<td>subnet-0fa858835c03</td>
<td>public-3a</td>
<td>Public subnet 3</td>
</tr>
<tr>
<td>.v2</td>
<td>subnet-0fa858835c03</td>
<td>public-2a</td>
<td>Public subnet 2</td>
</tr>
<tr>
<td>.v2</td>
<td>subnet-0fa858835c03</td>
<td>public-1a</td>
<td>Public subnet 1</td>
</tr>
</tbody>
</table>

**Selected subnets**

<table>
<thead>
<tr>
<th>Actions</th>
<th>Subnet ID</th>
<th>Availability Zone</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>.v2</td>
<td>subnet-0fa858835c03</td>
<td>public-3a</td>
<td>Public subnet 3</td>
</tr>
<tr>
<td>.v2</td>
<td>subnet-0fa858835c03</td>
<td>public-2a</td>
<td>Public subnet 2</td>
</tr>
</tbody>
</table>

**Assign Security Groups** page, select a security group for the ELB instance. The security group can be same as the security group attached to Citrix ADC VPX ENI or it can be a new security group.

If you are using a new security group, make sure that you allow traffic to the Citrix ADC VPX security group from the ELB security group and conversely.

**Configure Health Check** page, select the configuration for the health check. By
default health check is set as TCP on port 80, optionally you can do the health check on port 443 as well.

**Step 4: Configure Health Check**

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>80</td>
</tr>
</tbody>
</table>

Advanced Details

- **Response Timeout:** 5 seconds
- **Interval:** 30 seconds
- **Unhealthy threshold:** 2
- **Healthy threshold:** 1

You load balancer will automatically perform health checks on your EC2 instances and only route traffic to instances that pass the health check. If an instance fails the health check, it is automatically removed from the load balancer.

In the **Add EC2 Instances** page, select two Citrix ADC VPX instances that were deployed earlier.

```
1 ![Classic ADD EC2 Instances](/en-us/citrix-k8s-ingress-controller/media/classic-add-ec2.png)
```

In the **Add Tags** page, add tags as per your requirement.

In the **Review** page, review your configurations.

Click **Create**.

**Verify the solution**

After you have successfully deployed Citrix ADC VPX, AWS ELB, and Citrix ingress controller, you can verify the solution using a sample service.

Perform the following:

1. Deploy a sample service and ingress using `app.yaml`.

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

2. Log on to the Citrix ADC VPX instance and verify if the Content Switching vserver are successfully configured on both the Citrix ADC VPX instance. Do the following:

   a) Log on to the Citrix ADC VPX instance. Perform the following:
      i. Use an SSH client, such as PuTTY, to open an SSH connection to the Citrix ADC VPX instance.
      ii. Log on to the instance by using the administrator credentials.
b) Verify if the Content Switching (cs) vserver is configured on the instance using the following command:

```bash
sh cs vserver
```

Output:

```
1) k8s-10.0.139.87:80:http (10.0.139.87:80) - HTTP Type: CONTENT
   State: UP
   Last state change was at Fri Apr 12 14:24:13 2019
   Time since last state change: 3 days, 03:09:18.920
   Client Idle Timeout: 180 sec
   Down state flush: ENABLED
   Disable Primary Vserver On Down: DISABLED
   Comment: uid=
   NNJRYQ54VM2KWCX0ERK6HR3HR4VEQYR7U3W4BNFQLTIAENMTWA
   ===
   Appflow logging: ENABLED
   Port Rewrite: DISABLED
   State Update: DISABLED
   Default: Content Precedence: RULE
   Vserver IP and Port insertion: OFF
   L2Conn: OFF Case Sensitivity: ON
   Authentication: OFF
   401 Based Authentication: OFF
   Listen Policy: NONE
   IcmpResponse: PASSIVE
   RHIsstate: PASSIVE
   Traffic Domain: 0
```

c) Access the application `test.example.com` using the DNS name of the ELB instance.

```bash
# curl -H 'Host: test.example.com' <DNS name of the ELB>
```

Example:
Citrix ADC ingress controller

d) To delete the deployment, use the following command:

```
kubectl delete -f app.yaml
```

Troubleshooting

<table>
<thead>
<tr>
<th>Problem</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>CloudFormation stack failure</td>
<td>Ensure that the IAM user or role has sufficient privilege to create EC2 instances and Lambda configurations. Ensure that you haven’t exceeded the resource quota.</td>
</tr>
<tr>
<td>Citrix ingress controller unable to communicate with the Citrix ADC VPX instances.</td>
<td>Ensure that user name and password is correct in <code>citrix-ingress-controller.yaml</code> file. Ensure that Citrix ADC VPX security group allows the traffic on port 80 and 443 from the EKS node group security group.</td>
</tr>
<tr>
<td>The services are DOWN in the Citrix ADC VPX instances.</td>
<td>Ensure that the Citrix ADC VPX traffic can reach the EKS cluster. Modify the security group of EKS node group to allow traffic from Citrix ADC VPX security group.</td>
</tr>
<tr>
<td>Traffic not routing to Citrix ADC VPX instance from ELB.</td>
<td>Ensure that security group of Citrix ADC VPX allows traffic from the ELB security group.</td>
</tr>
</tbody>
</table>

Deploy the Citrix ingress controller for Citrix ADC with admin partitions

June 2, 2022

Citrix ingress controller is used to automatically configure one or more Citrix ADC based on the Ingress resource configuration. The ingress Citrix ADC appliance (MPX or VPX) can be partitioned into logical
entities called admin partitions, where each partition can be configured and used as a separate Citrix ADC appliance. For more information, see Admin Partition. Citrix ingress controller can also be deployed to configure Citrix ADC with admin partitions.

For Citrix ADC with admin partitions, you must deploy a single instance of Citrix ingress controller for each partition. And, the partition must be associated with a partition user specific to the Citrix ingress controller instance.

**Prerequisites**

Ensure that:

- Admin partitions are configured on the Citrix ADC appliance. For instructions see, Configure admin partitions.

- Create a partition user specifically for the Citrix ingress controller. Citrix ingress controller configures the Citrix ADC using this partition user account. Ensure that you do not associate this partition user to other partitions in the Citrix ADC appliance.

  **Note:**
  
  For SSL-related use cases in the admin partition, ensure that you use Citrix ADC version 12.0–56.8 and above.

**To deploy the Citrix ingress controller for Citrix ADC with admin partitions**

1. Download the citrix-k8s-ingress-controller.yaml using the following command:

   ```bash
   wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/baremetal/citrix-k8s-ingress-controller.yaml
   ```

2. Edit the citrix-k8s-ingress-controller.yaml file and enter the values for the following environmental variables:

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Mandatory or Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS_IP</td>
<td>Mandatory</td>
<td>The IP address of the Citrix ADC appliance. For more details, see Prerequisites.</td>
</tr>
<tr>
<td>Environment Variable</td>
<td>Mandatory or Optional</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NS_USER and NS_PASSWORD</td>
<td>Mandatory</td>
<td>The user name and password of the partition user that you have created for the Citrix ingress controller. For more details, see Prerequisites.</td>
</tr>
<tr>
<td>NS_VIP</td>
<td>Mandatory</td>
<td>Citrix ingress controller uses the IP address provided in this environment variable to configure a virtual IP address to the Citrix ADC that receives the Ingress traffic. <strong>Note:</strong> NS_VIP acts as a fallback when the <code>frontend-ip</code> annotation is not provided in Ingress yaml. Only Supported for Ingress.</td>
</tr>
<tr>
<td>NS_SNIPS</td>
<td>Optional</td>
<td>Specifies the SNIP addresses on the Citrix ADC appliance or the SNIP addresses on a specific admin partition on the Citrix ADC appliance.</td>
</tr>
<tr>
<td>NS_ENABLE_MONITORING</td>
<td>Mandatory</td>
<td>Set the value <strong>Yes</strong> to monitor Citrix ADC. <strong>Note:</strong> Ensure that you disable Citrix ADC monitoring for Citrix ADC with admin partitions. Set the value to <strong>No</strong>.</td>
</tr>
<tr>
<td>EULA</td>
<td>Mandatory</td>
<td>The End User License Agreement. Specify the value as <strong>Yes</strong>.</td>
</tr>
<tr>
<td>Environment Variable</td>
<td>Mandatory or Optional</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Kubernetes_url</td>
<td>Optional</td>
<td>The kube-apiserver url that Citrix ingress controller uses to register the events. If the value is not specified, Citrix ingress controller uses the internal kube-apiserver IP address.</td>
</tr>
<tr>
<td>LOGLEVEL</td>
<td>Optional</td>
<td>The log levels to control the logs generated by Citrix ingress controller. By default, the value is set to DEBUG. The supported values are: CRITICAL, ERROR, WARNING, INFO, and DEBUG. For more information, see Log Levels</td>
</tr>
<tr>
<td>NS_PROTOCOL and NS_PORT</td>
<td>Optional</td>
<td>Defines the protocol and port that must be used by the Citrix ingress controller to communicate with Citrix ADC. By default, the Citrix ingress controller uses HTTPS on port 443. You can also use HTTP on port 80.</td>
</tr>
<tr>
<td>ingress-classes</td>
<td>Optional</td>
<td>If multiple ingress load balancers are used to load balance different ingress resources. You can use this environment variable to specify the Citrix ingress controller to configure Citrix ADC associated with a specific ingress class. For information on Ingress classes, see Ingress class support</td>
</tr>
</tbody>
</table>

3. Once you update the environment variables, save the YAML file and deploy it using the following
command:

```bash
kubectl create -f citrix-k8s-ingress-controller.yaml
```

4. Verify if the Citrix ingress controller is deployed successfully using the following command:

```bash
kubectl get pods --all-namespaces
```

**Use case: How to securely deliver multitenant microservice-based applications using Citrix ADC admin partitions**

You can isolate ingress traffic between different microservice based applications with the Citrix ADC admin partition using Citrix ingress controller. Citrix ADC admin partition enables multitenancy at the software level in a single Citrix ADC instance. Each partition has its own control plane and network plane.

You can deploy one instance of Citrix ingress controller in each namespace in a cluster.

For example, imagine you have two namespaces in a Kubernetes cluster and you want to isolate these namespaces from each other under two different admins. You can use the admin partition feature to separate these two namespaces. Create namespace 1 and namespace 2 and deploy Citrix ingress controller separately in both of these namespaces.

Citrix ingress controller instances provide configuration instructions to the respective Citrix ADC partitions using the system user account specified in the YAML manifest.
In this example, apache and guestbook sample applications are deployed in two different namespaces (namespace 1 and namespace 2 respectively) in a Kubernetes cluster. Both apache and guestbook application team wants to manage their workload independently and do not want to share resources. Citrix ADC admin partition helps to achieve multitenancy and in this example, two partitions (default, partition1) are used to manage both application workload separately.

The following prerequisites apply:

- Ensure that you have configured admin partitions on the Citrix ADC appliance. For instructions see, Configure admin partitions.
- Ensure that you create a partition user account specifically for the Citrix ingress controller. Citrix ingress controller configures the Citrix ADC using this partition user account. Ensure that you do not associate this partition user to other partitions in the Citrix ADC appliance.

Example

The following example scenario shows how to deploy different applications within different namespaces in a Kubernetes cluster and how the request can be isolated from ADC using the admin partition.
Citrix ADC ingress controller

In this example, two sample applications are deployed in two different namespaces in a Kubernetes cluster. In this example, it is used a default partition in Citrix ADC for the apache application and the admin partition p1 for the guestbook application.

Create namespaces

Create two namespaces ns1 and ns2 using the following commands:

```
1  kubectl create namespace ns1
2  kubectl create namespace ns2
```

Configurations in namespace ns1

1. Deploy the apache application in ns1.

```
1  apiVersion: v1
2  kind: Namespace
3  metadata:
4    name: ns1
5  ---
6  apiVersion: apps/v1
7  kind: Deployment
8  metadata:
9    labels:
10   app: apache-ns1
11   name: apache-ns1
12   namespace: ns1
13  spec:
14    replicas: 2
15    selector:
16      matchLabels:
17        app: apache-ns1
18  template:
19    metadata:
20      labels:
21        app: apache-ns1
22    spec:
23      containers:
24        - image: httpd
25          name: httpd
```
### Deploy Citrix ingress controller

2. **Deploy Citrix ingress controller in** `ns1`.

   You can use the YAML file to deploy Citrix ingress controller or use the Helm chart.

   Ensure that you use the user credentials that is bind to the default partition.

   ```sh
ehelm install cic-def-part-ns1 citrix/citrix-ingress-controller --set nsIP=<nsIP of ADC>,license.accept=yes,adcCredentialSecret=nslogin,ingressClass[0]=citrix-def-part-ns1 --namespace ns1
```

3. **Deploy the Ingress resource.**

   ```sh
   apiVersion: networking.k8s.io/v1
   kind: Ingress
   metadata:
   name: ingress-apache-ns1
   namespace: ns1
   annotations:
   kubernetes.io/ingress.class: "citrix-def-part-ns1"
   ingress.citrix.com/frontend-ip: "< ADC VIP IP >"
   spec:
   rules:
   - host: apache-ns1.com
     http:
     paths:
   ```
Citrix ADC ingress controller

```yaml
- backend:
  service:
    name: apache-ns1
    port:
      number: 80
    pathType: Prefix
    path: /index.html
```

4. Citrix ingress controller in ns1 configures the ADC entities in the default partition.

**Configurations in namespace ns2**

1. Deploy guestbook application in ns2.

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: ns2

---

apiVersion: v1
kind: Service
metadata:
  name: redis-master
  namespace: ns2
  labels:
    app: redis
    tier: backend
    role: master
spec:
  ports:
    - port: 6379
      targetPort: 6379
  selector:
    app: redis
    tier: backend
    role: master

---

apiVersion: apps/v1 # for k8s versions before 1.9.0 use apps/v1beta2 and before 1.8.0 use extensions/v1beta1
kind: Deployment
metadata:
  name: redis-master
  namespace: ns2
```

© 1999‒2022 Citrix Systems, Inc. All rights reserved.
```yaml
spec:
  selector:
    matchLabels:
      app: redis
      role: master
      tier: backend
  replicas: 1

template:
  metadata:
    labels:
      app: redis
      role: master
      tier: backend
  spec:
    containers:
      - name: master
        image: k8s.gcr.io/redis:e2e  # or just image: redis
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
          - containerPort: 6379

---

apiVersion: v1
kind: Service
metadata:
  name: redis-slave
  namespace: ns2
labels:
  app: redis
  tier: backend
  role: slave
spec:
  ports:
    - port: 6379
  selector:
    app: redis
    tier: backend
    role: slave

---

apiVersion: apps/v1  # for k8s versions before 1.9.0 use apps/
  v1beta2  and before 1.8.0 use extensions/v1beta1
kind: Deployment
metadata:
```
```yaml
name: redis-slave
namespace: ns2
spec:
  selector:
    matchLabels:
      app: redis
      role: slave
      tier: backend
  replicas: 2
  template:
    metadata:
      labels:
        app: redis
        role: slave
        tier: backend
    spec:
      containers:
        - name: slave
          image: gcr.io/google_samples/gb-redisslave:v1
          resources:
            requests:
              cpu: 100m
              memory: 100Mi
          env:
            - name: GET_HOSTS_FROM
              value: dns
              # If your cluster config does not include a dns service,
              # instead access an environment variable to find the master
              # service's host, comment out the 'value: dns' line above, and
              # uncomment the line below:
              # value: env
            ports:
              - containerPort: 6379
```

---

```yaml
apiVersion: v1
doctype: kind: Service
metadata:
  name: frontend
  namespace: ns2
  labels:
    app: guestbook
    tier: frontend
```
spec:
  # if your cluster supports it, uncomment the following to automatically create
  # an external load-balanced IP for the frontend service.
  # type: LoadBalancer
  ports:
    - port: 80
  selector:
    app: guestbook
    tier: frontend

---

apiVersion: apps/v1 # for k8s versions before 1.9.0 use apps/v1beta2 and before 1.8.0 use extensions/v1beta1
kind: Deployment
metadata:
  name: frontend
  namespace: ns2
spec:
  selector:
    matchLabels:
      app: guestbook
      tier: frontend
  replicas: 3
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google-samples/gb-frontend:v4
          resources:
            requests:
              cpu: 100m
              memory: 100Mi
          env:
            - name: GET_HOSTS_FROM
              value: dns
              # If your cluster config does not include a dns service, then to
              # instead access environment variables to find service host
              # info, comment out the 'value: dns' line above, and
              # uncomment the
2. Deploy Citrix ingress controller in namespace ns2.
   Ensure that you use the user credentials that is bind to the partition p1.

```
helm install cic-adm-part-p1 citrix/citrix-ingress-controller --
    set nsIP=<nsIP of ADC>,nsSNIPS='[<SNIPs in partition p1>]',
    license.accept=yes,adcCredentialSecret=admin-part-user-p1,
    ingressClass[0]=citrix-adm-part-ns2 --namespace ns2
```

3. Deploy ingress for the guestbook application.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: citrix-adm-part-ns2
    ingress.citrix.com/frontend-ip: "<VIP in partition 1>"
  name: guestbook-ingress
  namespace: ns2
spec:
  rules:
  - host: www.guestbook.com
    http:
      paths:
      - backend:
          service:
            name: frontend
            port:
              number: 80
        path: /
        pathType: Prefix
```

4. Citrix ingress controller in ns2 configures the ADC entities in partition p1.
Deploy Citrix solution for service of type LoadBalancer in AWS

February 4, 2022

A service of type LoadBalancer is a simpler and faster way to expose a microservice running in a Kubernetes cluster to the external world. In cloud deployments, when you create a service of type LoadBalancer, a cloud managed load balancer is assigned to the service. The service is, then, exposed using the load balancer. For more information about services of type LoadBalancer, see Services of type LoadBalancer.

With the Citrix solution for service of type LoadBalancer, you can use Citrix ADC to directly load balance and expose a service instead of the cloud managed load balancer. Citrix provides this solution for service of type LoadBalancer for on-prem and cloud. Services of type LoadBalancer are natively supported in Kubernetes deployments on public clouds such as AWS, GCP, and Azure.

When you deploy a service in AWS, a load balancer is created automatically and the IP address is allocated to the external field of the service. In this Citrix solution, Citrix IPAM controller allocates the IP address and that IP address is the VIP of Citrix ADC VPX. Citrix ingress controller, deployed in a Kubernetes cluster, configures a Citrix ADC deployed outside the cluster to load balance the incoming traffic. So, the service is accessed through Citrix ADC VPX instead of the cloud load balancer.

You need to specify the service type as LoadBalancer in the service definition. Setting the type field to LoadBalancer provisions a load balancer for your service on AWS.

Citrix IPAM controller is used to automatically allocate IP addresses to services of type LoadBalancer from a specified range of IP addresses. For more information about the Citrix solution for services of type LoadBalancer, see Expose services of type LoadBalancer.

You can deploy the Citrix solution for service of type LoadBalancer in AWS using Helm charts or YAML files.

Prerequisites

- Ensure that the Elastic Kubernetes Service (EKS) cluster version 1.18 or later is running.
- Ensure that Citrix ADC VPX and EKS are deployed and running in the same VPC. For information about creating Citrix ADC VPX in AWS, see Create a Citrix ADC VPX instance from AWS Marketplace.

Deploy Citrix solution for service of type LoadBalancer in AWS using Helm charts

Perform the following steps to configure the Citrix solution for service of type LoadBalancer using Helm charts.
Citrix ADC ingress controller

1. Download the `unified-lb-values.yaml` file and edit the YAML file for specifying the following details:
   - Citrix ADC VPX NSIP. For more information, see [Citrix ingress controller Helm chart](#).
   - Secret created using the Citrix ADC VPX credentials. For more information, see [Citrix ingress controller Helm chart](#).
   - List of VIPs to be used in IPAM controller. For more information, see [IPAM Helm chart](#).

2. Deploy Citrix IPAM controller and Citrix ingress controller on your Amazon EKS cluster using the edited YAML file. Use the following commands:

   ```
   1  helm repo add citrix https://citrix.github.io/citrix-helm-charts/
   2  helm install serviceLB citrix/citrix-cloud-native -f values.yaml
   ```

3. Deploy the application and service in Amazon EKS:
   a) Add the following annotation in the service manifest:

   ```
   1  beta.kubernetes.io/aws-load-balancer-type: "external"
   ```

   b) Deploy the application and service with the modified annotation using the following command:

   ```
   ```

   **Note:** The `guestbook` microservice is a sample used in this procedure. You can deploy an application of your choice. Ensure that the service should be of type LoadBalancer and the service manifest should contain the annotation.

   c) Associate an elastic IP address with the VIP of Citrix ADC VPX.

   d) Access the application using a browser. For example, `http://EIP-associated-with-vip`.

---

**Deploy Citrix solution for service of type LoadBalancer in AWS using YAML**

Perform the following steps to deploy the Citrix solution for service of type LoadBalancer using YAML.
1. Download the `citrix-k8s-ingress-controller.yaml` file and specify the following details.
   - Citrix ADC VPX NSIP
   - Secret created using the Citrix ADC VPX credentials. For information about creating the secret, see Create a secret.
   - Specify the argument for Citrix IPAM controller:

   ```
   args:
   - --ipam citrix-ipam-controller
   ```

2. Deploy the Citrix ingress controller using the modified YAML.

   ```
   kubectl create -f citrix-k8s-ingress-controller.yaml
   ```

3. Deploy the Citrix VIP CRD which enables communication between the Citrix ingress controller and the IPAM controller using the following command.

   ```
   kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/crd/vip/vip.yaml
   ```

   For more information about deploying Citrix VIP CRD, see Deploy the VIP CRD.

4. Deploy the IPAM controller. For information about deploying the IPAM controller, see Deploy the IPAM controller.

   **Note:**

   Specify the list of Citrix ADC VPX VIPs in the `VIP_RANGE` field of the IPAM deployment YAML file.

5. Deploy the application with service type LoadBalancer in Amazon EKS using the following steps:
   a) Add the following annotation in the service manifest.

   ```
   beta.kubernetes.io/aws-load-balancer-type: "external"
   ```

   b) Deploy the application and service with the modified annotation using the following command.
Citrix ADC ingress controller

```bash
```

**Note:**
The guestbook microservice is a sample used in this procedure. You can deploy an application of your choice. Ensure that the service should be of type LoadBalancer and the service manifest should contain the annotation.

c) Associate an elastic IP address with the VIP of Citrix ADC VPX.

d) Access the application using a browser. For example, `http://EIP-associated-with-vip`.

**Multi-cloud and multi-cluster ingress and load balancing solution with Amazon EKS and Microsoft AKS clusters**

March 8, 2022

You can deploy multiple instances of the same application across multiple clouds provided by different cloud providers. This multi-cloud strategy helps you to ensure resiliency, high availability, and proximity. A multi-cloud approach also allows you to take advantage of the best of each cloud provider by reducing the risks such as vendor lock-in and cloud outages.

Citrix ADC with the help of the Citrix ingress controller can perform multi-cloud load balancing. Citrix ADC can direct traffic to clusters hosted on different cloud provider sites. The solution performs load balancing by distributing the traffic intelligently between the workloads running on Amazon EKS (Elastic Kubernetes Service) and Microsoft AKS (Azure Kubernetes Service) clusters.

You can deploy the multi-cloud and multi-cluster ingress and load balancing solution with Amazon EKS and Microsoft AKS.

**Deployment topology**

The following diagram explains a deployment topology of the multi-cloud ingress and load balancing solution for Kubernetes service provided by Amazon EKS and Microsoft AKS.
**Prerequisites**

- You should be familiar with AWS and Azure.
- You should be familiar with Citrix ADC and Citrix ADC networking.
- Instances of the same application must be deployed in Kubernetes clusters on Amazon EKS and Microsoft AKS.

To deploy the multi-cloud multi-cluster ingress and load balancing solution, you must perform the following tasks.

1. Deploy Citrix ADC VPX in AWS.
2. Deploy Citrix ADC VPX in Azure.
3. Configure ADNS service on Citrix ADC VPX deployed in AWS and AKS.
4. Configure GSLB service on Citrix ADC VPX deployed in AWS and AKS.
5. Apply GTP and GSE CRDs on AWS and Azure Kubernetes clusters.
6. Deploy the multi-cluster controller.

**Deploying Citrix ADC VPX in AWS**

You must ensure that the Citrix ADC VPX instances are installed in the same virtual private cloud (VPC) on the EKS cluster. It enables Citrix ADC VPX to communicate with EKS workloads. You can use an
existing EKS subnet or create a subnet to install the Citrix ADC VPX instances.

Also, you can install the Citrix ADC VPX instances in a different VPC. In that case, you must ensure that the VPC for EKS can communicate using VPC peering. For more information about VPC peering, see VPC peering documentation.

For high availability (HA), you can install two instances of Citrix ADC VPX in HA mode.

1. Install Citrix ADC VPX in AWS. For information on installing Citrix ADC VPX in AWS, see Deploy Citrix ADC VPX instance on AWS.

   Citrix ADC VPX requires a secondary public IP address other than the NSIP to run GSLB service sync and ADNS service.

2. Open the AWS console and choose EC2 > Network Interfaces > VPX primary ENI ID > Manage IP addresses. Click Assign new IP Address.

   After the secondary public IP address has been assigned to the VPX ENI, associate an elastic IP address to it.

3. Choose EC2 > Network Interfaces > VPX ENI ID - Actions, click Associate IP Address. Select an elastic IP address for the secondary IP address and click Associate.
4. Log in to the Citrix ADC VPX instance and add the secondary IP address as SNIP and enable the management access using the following command:

```
add ip 192.168.211.73 255.255.224.0 -mgmtAccess ENABLED -type SNIP
```

**Note:**

- To log in to Citrix ADC VPX using SSH, you must enable the SSH port in the security group. Route tables must have an internet gateway configured for the default traffic and the NACL must allow the SSH port.
- If you are running the Citrix ADC VPX in High Availability (HA) mode, you must perform this configuration in both of the Citrix ADC VPX instances.

5. Enable Content Switching (CS), Load Balancing (LB), Global Server Load Balancing (GSLB), and SSL features in Citrix ADC VPX using the following command:

```
enable feature *feature*
```
Note:
To enable GSLB, you must have an additional license.

6. Enable port 53 for UDP and TCP in the VPX security group for Citrix ADC VPX to receive DNS traffic. Also enable the TCP port 22 for SSH and the TCP port range 3008–3011 for GSLB metric exchange.

For information on adding rules to the security group, see Adding rules to a security group.

7. Add a nameserver to Citrix ADC VPX using the following command:

   ```bash
   add nameserver *nameserver IP*
   ```

### Deploying Citrix ADC VPX in Azure

You can run a standalone Citrix ADC VPX instance on an AKS cluster or run two Citrix ADC VPX instances in High Availability mode on the AKS cluster.

While installing, ensure that the AKS cluster must have connectivity with the VPX instances. To ensure the connectivity, you can install the Citrix ADC VPX in the same virtual network (VNet) on the AKS cluster in a different resource group.

While installing the Citrix ADC VPX, select the VNet where the AKS cluster is installed. Alternatively, you can use VNet peering to ensure the connectivity between AKS and Citrix ADC VPX if the VPX is deployed in a different VNet other than the AKS cluster.

1. Install Citrix ADC VPX in AWS. For information on installing Citrix ADC VPX in AKS, see Deploy a Citrix ADC VPX instance on Microsoft Azure.

   You must have a SNIP with public IP for GSLB sync and ADNS service. If SNIP already exists, associate a public IP address with it.

2. To associate, choose Home > Resource group > VPX instance > VPX NIC instance. Associate a public IP address as shown in the following image. Click Save to save the changes.
3. Log in to the Azure Citrix ADC VPX instance and add the secondary IP as SNIP with the management access enabled using the following command:

```
add ip 10.240.0.11 255.255.0.0 -type SNIP -mgmtAccess ENABLED
```

If the resource exists, you can use the following command to set the management access enabled on the existing resource.

```
set ip 10.240.0.11 -mgmtAccess ENABLED
```

4. Enable CS, LB, SSL, and GSLB features in the Citrix ADC VPX using the following command:

```
enable feature *feature*
```

To access the Citrix ADC VPX instance through SSH, you must enable the inbound port rule for the SSH port in the Azure network security group that is attached to the Citrix ADC VPX primary interface.
5. Enable the inbound rule for the following ports in the network security group on the Azure portal.
   - TCP: 3008–3011 for GSLB metric exchange
   - TCP: 22 for SSH
   - TCP and UDP: 53 for DNS

6. Add a nameserver to Citrix ADC VPX using the following command:

```
add nameserver *nameserver IP*
```

**Configure ADNS service in Citrix ADC VPX deployed in AWS and Azure**

The ADNS service in Citrix ADC VPX acts as an authoritative DNS for your domain. For more information on the ADNS service, see [Authoritative DNS service](#).

1. Log in to AWS Citrix ADC VPX and configure the ADNS service on the secondary IP address and port 53 using the following command:

```
add service Service-ADNS-1 192.168.211.73 ADNS 53
```

Verify the configuration using the following command:

```
show service Service-ADNS-1
```

2. Log in to Azure Citrix ADC VPX and configure the ADNS service on the secondary IP address and port 53 using the following command:

```
add service Service-ADNS-1 10.240.0.8 ADNS 53
```

Verify the configuration using the following command:

```
show service Service-ADNS-1
```

3. After creating two ADNS service for the domain, update the NS record of the domain to point to the ADNS services in the domain registrar.

For example, create an 'A' record `ns1.domain.com` pointing to the ADNS service public IP address. NS record for the domain must point to ns1.domain.com.

© 1999–2022 Citrix Systems, Inc. All rights reserved.
Configure GSLB service in Citrix ADC VPX deployed in AWS and Azure

You must create GSLB sites on Citrix ADC VPX deployed on AWS and Azure.

1. Log in to AWS Citrix ADC VPX and configure GSLB sites on the secondary IP address using the following command. Also, specify the public IP address using the \(-publicIP\) argument. For example:

```
1 add gslb site aws_site 192.168.197.18 -publicIP 3.139.156.175
2 add gslb site azure_site 10.240.0.11 -publicIP 23.100.28.121
```

2. Log in to Azure Citrix ADC VPX and configure GSLB sites. For example:

```
1 add gslb site aws_site 192.168.197.18 -publicIP 3.139.156.175
2 add gslb site azure_site 10.240.0.11 -publicIP 23.100.28.121
```

3. Verify that the GSLB sync is successful by initiating a sync from any of the sites using the following command:

```
1 sync gslb config -debug
```

**Note:**
If the initial sync fails, review the security groups on both AWS and Azure to allow the required ports.

Apply GTP and GSE CRDs on AWS and Azure Kubernetes clusters

The global traffic policy (GTP) and global service entry (GSE) CRDs help to configure Citrix ADC for performing GSLB in Kubernetes applications. These CRDs are designed for configuring multi-cluster ingress and load balancing solution for Kubernetes clusters.

**GTP CRD**

The GTP CRD accepts the parameters for configuring GSLB on the Citrix ADC including deployment type (canary, failover, and local-first), GSLB domain, health monitor for the ingress, and service type. For GTP CRD definition, see the [GTP CRD](#). Apply the GTP CRD definition on AWS and Azure Kubernetes clusters using the following command:
GSE CRD

The GSE CRD specifies the endpoint information (information about any Kubernetes object that routes traffic into the cluster) in each cluster. The global service entry automatically picks the external IP address of the application, which routes traffic into the cluster. If the external IP address of the routes change, the global service entry picks a newly assigned IP address and configure the multi-cluster endpoints of Citrix ADCs accordingly.

For the GSE CRD definition, see the GSE CRD. Apply the GSE CRD definition on AWS and Azure Kubernetes clusters using the following command:

```
1 kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/multicluster/Manifest/gtp-crd.yaml
```

Deploy multi-cluster controller

Multi-cluster controller helps you to ensure the high availability of the applications across clusters in a multi-cloud environment.

You can install the multi-cluster controller on the AWS and Azure clusters. Multi-cluster controller listens to GTP and GSE CRDs and configures the Citrix ADC for GSLB that provides high availability across multiple regions in a multi-cloud environment.

To deploy the multi-cluster controller, perform the following steps:


```
1 kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/multicluster/Manifest/gslb-rbac.yaml
```

2. Create the secrets on the AWS and Azure clusters using the following command:
Note:
Secrets enable the GSLB controller to connect and push the configuration to the GSLB devices.

```
kubectl create secret generic secret-1 --from-literal=username=<username> --from-literal=password=<password>
```

Note:
You can add a user to Citrix ADC using the `add system user` command.

3. Download the GSLB controller YAML file from `gslb-controller.yaml`.

4. Apply the `gslb-controller.yaml` in an AWS cluster using the following command:

```
kubectl apply -f gslb-controller.yaml
```

For the AWS environment, edit the `gslb-controller.yaml` to define the LOCAL_REGION, LOCAL_CLUSTER, and SITENAMES environment variables.

The following example defines the environment variable LOCAL_REGION as `us-east-2` and LOCAL_CLUSTER as `eks-cluster` and the SITENAMES environment variable as `aws_site,azure_site`.

```yaml
name: "LOCAL_REGION"
value: "us-east-2"
name: "LOCAL_CLUSTER"
value: "eks-cluster"
name: "SITENAMES"
value: "aws_site,azure_site"
name: "aws_site_ip"
value: "NSIP of aws VPX(internal IP)"
name: "aws_site_region"
value: "us-east-2"
name: "azure_site_ip"
value: "NSIP of azure_VPX(public IP)"
name: "azure_site_region"
value: "central-india"
name: "azure_site_username"
valueFrom:
secretKeyRef:
```
Apply the `gslb-controller.yaml` in the Azure cluster using the following command:

```
kubectl apply -f gslb-controller.yaml
```

5. For the Azure site, edit the `gslb-controller.yaml` to define `LOCAL_REGION`, `LOCAL_CLUSTER`, and `SITENAMES` environment variables.

The following example defines the environment variable `LOCAL_REGION` as `central-india`, `LOCAL_CLUSTER` as `azure-cluster`, and `SITENAMES` as `aws_site, azure_site`.

```
name: "LOCAL_REGION"
value: "central-india"
name: "LOCAL_CLUSTER"
value: "aks-cluster"
name: "SITENAMES"
value: "aws_site,azure_site"
name: "aws_site_ip"
value: "NSIP of AWS VPX(public IP)"
name: "aws_site_region"
value: "us-east-2"
name: "azure_site_ip"
value: "NSIP of azure VPX(internal IP)"
name: "azure_site_region"
value: "central-india"
```
Note:
The order of the GSLB site information should be the same in all clusters. The first site in the order is considered as the master site for pushing the configuration. Whenever the master site goes down, the next site in the list becomes the new master. Hence, the order of the sites should be the same in all Kubernetes clusters.

Deploy a sample application

In this example application deployment scenario, an https image of apache is used. However, you can choose the sample application of your choice.

The application is exposed as type LoadBalancer in both AWS and Azure clusters. You must run the commands in both AWS and Azure Kubernetes clusters.

1. Create a deployment of a sample apache application using the following command:

```
kubectl create deploy apache --image=httpd:latest port=80
```

2. Expose the apache application as service of type LoadBalancer using the following command:

```
kubectl expose deploy apache --type=LoadBalancer --port=80
```
3. Verify that an external IP address is allocated for the service of type LoadBalancer using the following command:

```
kubectl get svc apache
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER-IP</th>
<th>EXTERNAL-IP</th>
<th>PORT(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>LoadBalancer</td>
<td>10.0.16.231</td>
<td>20.62.235.193</td>
<td>80:32666/TCP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3m2s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After deploying the application on AWS and Azure clusters, you must configure the GTE custom resource to configure high availability in the multi-cloud clusters.

Create a GTP YAML resource `gtp_instance.yaml` as shown in the following example.

```
apiVersion: "citrix.com/v1beta1"
kind: globaltrafficpolicy
metadata:
  name: gtp-sample-app
  namespace: default
spec:
  serviceType: 'HTTP'
  hosts:
    - host: <domain name>
      policy:
        trafficPolicy: 'FAILOVER'
        secLbMethod: 'ROUNDROBIN'
        targets:
          - destination: 'apache.default.us-east-2.eks-cluster'
            weight: 1
          - destination: 'apache.default.central-india.aks-cluster'
            primary: false
            weight: 1
      monitor:
        - monType: http
          uri: '
            respCode: 200
      status:
        { }
```
Citrix ADC ingress controller

In this example, traffic policy is configured as **FAILOVER**. However, the multi-cluster controller supports multiple traffic policies. For more information, see the documentation for the traffic policies.

Apply the GTP resource in both the clusters using the following command:

```
1  kubectl apply -f gtp_instance.yaml
```

You can verify that the GSE resource is automatically created in both of the clusters with the required endpoint information derived from the service status. Verify using the following command:

```
1  kubectl get gse
2  kubectl get gse *name* -o yaml
```

Also, log in to Citrix ADC VPX and verify that the GSLB configuration is successfully created using the following command:

```
1  show gslb runningconfig
```

As the GTP CRD is configured for the traffic policy as **FAILOVER**, Citrix ADC VPX instances serve the traffic from the primary cluster (EKS cluster in this example).

```
1  curl -v http://*domain_name*
```

However, if an endpoint is not available in the EKS cluster, applications are automatically served from the Azure cluster. You can ensure it by setting the replica count to 0 in the primary cluster.

**Citrix ADC VPX as ingress and GSLB device for Amazon EKS and Microsoft AKS clusters**

You can deploy the multi-cloud and multi-cluster ingress and load balancing solution with Amazon EKS and Microsoft AKS with Citrix ADC VPX as GSLB and the same Citrix ADC VPX as ingress device too.

To deploy the multi-cloud multi-cluster ingress and load balancing with Citrix ADC VPX as the ingress device, you must complete the following tasks described in the previous sections:

1. Deploy Citrix ADC VPX in AWS
2. Deploy Citrix ADC VPX in Azure
3. Configure ADNS service on Citrix ADC VPX deployed in AWS and AKS
4. Configure GSLB service on Citrix ADC VPX deployed in AWS and AKS
5. Apply GTP and GSE CRDs on AWS and Azure Kubernetes clusters
6. Deploy the multi-cluster controller

After completing the preceding tasks, perform the following tasks:

1. Configure Citrix ADC VPX as Ingress Device for AWS
2. Configure Citrix ADC VPX as Ingress Device for Azure

**Configure Citrix ADC VPX as Ingress device for AWS**

Perform the following steps:

1. Create Citrix ADC VPX login credentials using Kubernetes secret

   ```sh
   kubectl create secret generic nslogin --from-literal=username='nsroot' --from-literal=password='<instance-id-of-vpx>'
   ```

   The Citrix ADC VPX password is usually the instance-id of the VPX if you have not changed it.

2. Configure SNIP in the Citrix ADC VPX by connecting to the Citrix ADC VPX using SSH. SNIP is the secondary IP address of Citrix a VPX to which the elastic IP address is not assigned.

   ```sh
   add ns ip 192.168.84.93 255.255.224.0
   ```

   This step is required for Citrix ADC to interact with the pods inside the Kubernetes cluster.

3. Update the Citrix ADC VPX management IP address and VIP in the Citrix ingress controller manifest.

   ```sh
   ```

   **Note:**
   If you do not have `wget` installed, you can use `fetch` or `curl`.

4. Update the primary IP address of Citrix ADC VPX in the `cic.yaml` in the following field.

   ```sh
   # Set NetScaler NSIP/SNIP, SNIP in case of HA (mgmt has to be enabled)
   ```
Citrix ADC ingress controller

```yaml
2  - name: "NS_IP"
3   value: "X.X.X.X"
```

5. Update the Citrix ADC VPX VIP in the `cic.yaml` in the following field. This is the private IP address to which you have assigned an elastic IP address.

```yaml
1  # Set NetScaler VIP for the data traffic
2  - name: "NS_VIP"
3  value: "X.X.X.X"
```

6. Once you have edited the YAML file with the required values deploy Citrix ingress controller.

```
kubectl create -f cic.yaml
```

**Configure Citrix ADC VPX as Ingress device for Azure**

Perform the following steps:

1. Create Citrix ADC VPX login credentials using Kubernetes secrets.

   ```bash
   kubectl create secret generic nslogin --from-literal=username='<azure-vpx-instance-username>' --from-literal=password='<azure-vpx-instance-password>'
   ```

   **Note:**

   The Citrix ADC VPX user name and password should be the same as the credentials set while creating Citrix ADC VPX on Azure.

2. Using SSH, configure a SNIP in the Citrix ADC VPX, which is the secondary IP address of the Citrix ADC VPX. This step is required for the Citrix ADC to interact with pods inside the Kubernetes cluster.

   ```bash
   add ns ip <snip-vpx-instance-private-ip> <vpx-instance-primary-ip-subnet>
   ```

   - `<snip-vpx-instance-private-ip>` is the dynamic private IP address assigned while adding a SNIP during the Citrix ADC VPX instance creation.
Citrix ADC ingress controller

- `vpx-instance-primary-ip-subnet` is the subnet of the primary private IP address of the Citrix ADC VPX instance.

To verify the subnet of the private IP address, SSH into the Citrix ADC VPX instance and use the following command.

```
show ip <primary-private-ip-address>
```

3. Update the Citrix ADC VPX image URL, management IP address, and VIP in the Citrix ingress controller YAML file.
   a) Download the Citrix ingress controller YAML file.

```
wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/azure/manifest/azurecic/cic.yaml
```

**Note:**
If you do not have `wget` installed, you can use the `fetch` or `curl` command.

b) Update the Citrix ingress controller image with the Azure image URL in the `cic.yaml` file.

```
- name: cic-k8s-ingress-controller
  # CIC Image from Azure
  image: "<azure-cic-image-url>"
```

c) Update the primary IP address of the Citrix ADC VPX in the `cic.yaml` with the primary private IP address of the Azure VPX instance.

```
# Set NetScaler NSIP/SNIP, SNIP in case of HA (mgmt has to be enabled)
- name: "NS_IP"
  value: "X.X.X.X"
```

d) Update the Citrix ADC VPX VIP in the `cic.yaml` with the private IP address of the VIP assigned during VPX Azure instance creation.
## Citrix ADC ingress controller

1. # Set NetScaler VIP for the data traffic
2. - name: "NS_VIP"
3. value: "X.X.X.X"

4. Once you have configured Citrix ingress controller with the required values, deploy the Citrix ingress controller using the following command.

```
kubectl create -f cic.yaml
```

### Annotations

**July 21, 2022**

**Ingress annotations**

The following are the Ingress annotations supported by Citrix:

<table>
<thead>
<tr>
<th>Annotations</th>
<th>Possible value</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>ingress.citrix.com/frontend-ip</td>
<td>IP address</td>
<td>Use this annotation to customize the virtual IP address (VIP). This IP address is configured in Citrix ADC as VIP. The annotation is mandatory if you are using Citrix ADC VPX or MPX.</td>
<td>Citrix ADC IP address is used as VIP.</td>
</tr>
</tbody>
</table>

**Note:** Do not use the annotation if you want to use the Citrix ADC IP address as VIP.
### Annotations

<table>
<thead>
<tr>
<th>Annotations</th>
<th>Possible value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ingress.citrix.com/frontend-ipset-name</td>
<td>Name of the IPSET</td>
<td>Use this annotation to specify the IPSET name for the front-end configuration. The IPSET is bound to the content switching virtual server. Use this annotation with the ingress.citrix.com/frontend-ip.</td>
</tr>
<tr>
<td>ingress.citrix.com/secure-port</td>
<td>Port number</td>
<td>Use this annotation to configure the port for HTTPS traffic. This port is configured in Citrix ADC as a port value for the corresponding CS virtual server.</td>
</tr>
<tr>
<td>ingress.citrix.com/insecure-port</td>
<td>Port number</td>
<td>Use this annotation to configure the port for HTTP, TCP, or UDP traffic. This port is configured in Citrix ADC as a port value for the corresponding CS virtual server.</td>
</tr>
</tbody>
</table>

**Note:** The IPSET name that you specify in the annotation should already be configured in Citrix ADC.
### Annotations

<table>
<thead>
<tr>
<th>Annotations</th>
<th>Possible value</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>ingress.citrix.com/insecure-termination</td>
<td>allow, redirect, or disallow</td>
<td>Use <strong>allow</strong> to allow HTTP traffic, Use <strong>redirect</strong> to redirect the HTTP request to HTTPS, or Use <strong>disallow</strong> if you want to drop the HTTP traffic.</td>
<td>disallow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For example: ingress.citrix.com/insecure-termination: &quot;redirect&quot;</td>
<td></td>
</tr>
<tr>
<td>ingress.citrix.com/secure-backend</td>
<td>In JSON form, list of services for secure-backend</td>
<td>Use <strong>True</strong>, if you want to establish secure HTTPS between Citrix ADC and the application, Use <strong>False</strong>, if you want to establish insecure HTTP connection Citrix ADC to the application.</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For example: ingress.citrix.com/secure-backend: { &quot;app1&quot; : &quot;True&quot;, &quot;app2&quot; : &quot;False&quot;, &quot;app3&quot; : &quot;True&quot;}</td>
<td></td>
</tr>
<tr>
<td>kubernetes.io/ingress.class</td>
<td>ingress class name</td>
<td>It is a way to associate a particular ingress resource with an ingress controller.</td>
<td>Configures all ingresses</td>
</tr>
</tbody>
</table>

© 1999–2022 Citrix Systems, Inc. All rights reserved.
<table>
<thead>
<tr>
<th>Annotations</th>
<th>Possible value</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>ingress.citrix.com/security-service-type</td>
<td>ssl or ssl_tcp</td>
<td>The annotation allows L4 load balancing with SSL over TCP as protocol. Use ssl_tcp if you want to use SSL over TCP.</td>
<td>ssl</td>
</tr>
<tr>
<td>ingress.citrix.com/insecure-service-type</td>
<td>http, tcp, udp, sip_udp, or any</td>
<td>The annotation allows L4 load balancing with tcp/udp/sip_udp any as protocol. Use tcp, if you want TCP as the protocol. Use udp, if you want UDP as the protocol.</td>
<td>http</td>
</tr>
<tr>
<td>ingress.citrix.com/path-match-method</td>
<td>prefix or exact</td>
<td>Use this annotation for ingress path matching. Use prefix for Citrix ingress controller to consider any path string as a prefix expression. Use exact for the Citrix ingress controller to consider the path as an exact match.</td>
<td>prefix</td>
</tr>
</tbody>
</table>

For example:

```
kubernetes.io/ingress.class:"Citrix"
```
### Annotations

<table>
<thead>
<tr>
<th>Annotations</th>
<th>Possible value</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ingress.citrix.com/path-match</strong></td>
<td></td>
<td>For example, the <code>ingress.citrix.com/path-match-method: &quot;prefix&quot;</code> annotation defines the Citrix ingress controller to consider any path string as a prefix expression.</td>
<td></td>
</tr>
<tr>
<td><strong>ingress.citrix.com/deployment</strong></td>
<td>dsr</td>
<td>Use this annotation to create Direct Server Return (DSR) configuration on Citrix ADC.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For example, the <code>ingress.citrix.com/deployment: &quot;dsr&quot;</code> annotation creates DSR configuration on the Citrix ADC.</td>
<td></td>
</tr>
<tr>
<td><strong>ingress.citrix.com/preconfigured-certkey</strong></td>
<td>Pre-configured certificate keys and type. Certificate type can be default, SNI, or CA.</td>
<td>Use this annotation to specify the preconfigured certificate key or keys in the Citrix ADC that you want to reuse and bind to applications.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the type parameter is not provided with the name of a certificate, then the certificate is considered as the default (non-SNI) type.</td>
<td></td>
</tr>
</tbody>
</table>
**Citrix ADC ingress controller**

<table>
<thead>
<tr>
<th>Annotations</th>
<th>Possible value</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For example in the</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ingress.citrix.com/</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>preconfigured-certkey:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'{ &quot;certs&quot;: [ { &quot;name&quot;: &quot;certkey1&quot;, &quot;type&quot;: &quot;default&quot; }, { &quot;name&quot;: &quot;certkey2&quot;, &quot;type&quot;: &quot;sni&quot; } ] } annotation, certkey1 is used as a non-SNI certificate and certkey2 is used as a SNI certificate.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Smart annotations for Ingress**

Smart annotation is an option provided by the Citrix ingress controller to efficiently enable Citrix ADC features using the Citrix ADC entity name. The Citrix ingress controller converts the Ingress in Kubernetes to a set of Citrix ADC objects. You can efficiently control these objects using smart annotations.

**Note:** To use smart annotations, you must have good understanding of Citrix ADC features and their respective entity names. For more information on Citrix ADC features and entity names, see [Citrix ADC Documentation](#).

Smart annotation takes JSON format as input. The key and value that you pass in the JSON format must match the Citrix ADC NITRO format. For more information on the Citrix ADC NITRO API, see [Citrix ADC 12.1 REST APIs - NITRO Documentation](#).

For example, if you want to enable the SRCIPDESTIPHASH based lb method, you must use the corresponding NITRO key and value format `lbmethod, SRCIPDESTIPHASH` respectively.

The following table details the smart annotations provided by the Citrix ingress controller:
**Citrix ADC ingress controller**

<table>
<thead>
<tr>
<th>Citrix ADC Entity Name</th>
<th>Smart Annotation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbvserver</td>
<td>ingress.citrix.com/\n  lbvserver</td>
<td>ingress.citrix.com/\n  lbvserver: '{ &quot;citrix-svc&quot;:{ &quot;lbmethod&quot;:&quot;SRCIPDESTIPHASH&quot; } } '</td>
</tr>
<tr>
<td>servicegroup</td>
<td>ingress.citrix.com/\n  servicegroup</td>
<td>ingress.citrix.com/\n  servicegroup: '{ &quot;appname&quot;:{ &quot;cip&quot;: &quot;Enabled&quot;,&quot;cipHeader&quot;:&quot;X-Forwarded-For&quot; } } '</td>
</tr>
<tr>
<td>monitor</td>
<td>ingress.citrix.com/\n  monitor</td>
<td>ingress.citrix.com/\n  monitor: '{ &quot;appname&quot;:{ &quot;type&quot;:&quot;http&quot; } }'</td>
</tr>
<tr>
<td>csvserver</td>
<td>ingress.citrix.com/\n  csvserver</td>
<td>ingress.citrix.com/\n  csvserver: '{ &quot;stateupdate&quot;:&quot;ENABLED&quot; }'</td>
</tr>
</tbody>
</table>

For information on smart annotations for HTTP, TCP, and SSL profiles, see [Configure HTTP, TCP, or SSL profiles on Citrix ADC](#).

**Sample ingress YAML with smart annotations**

The following is a sample Ingress YAML. It includes smart annotations to enable Citrix ADC features using the entities such as, lbvserver, servicegroup, and monitor:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    ingress.citrix.com/frontend-ip: 192.168.1.1
    ingress.citrix.com/insecure-port: "80"
    ingress.citrix.com/lbvserver: '{
      "citrix-svc":{
        "lbmethod":"LEASTCONNECTION", "persistenceType":"SOURCEIP" }
    }
```
The sample Ingress YAML includes use cases related to the service, *citrix-svc*, and the following table explains the smart annotations used in the sample:

<table>
<thead>
<tr>
<th>Smart Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ingress.citrix.com/lbvserver: '{ &quot;citrix-svc&quot;:{ &quot;lbmethod&quot;:&quot;LEASTCONNECTION&quot;, &quot;persistenceType&quot;:&quot;SOURCEIP&quot; } } '</td>
<td>Sets the load balancing method as <strong>Least Connection</strong> and also configures <strong>Source IP address persistence</strong>.</td>
</tr>
<tr>
<td>ingress.citrix.com/servicegroup: '{ &quot;citrix-svc&quot;:{ &quot;usip&quot;:&quot;yes } } '</td>
<td>Enables <strong>Use Source IP Mode (USIP)</strong> on the Ingress Citrix ADC device. When you enable USIP on the Citrix ADC, it uses the client’s IP address for communication with the back-end pods.</td>
</tr>
<tr>
<td>ingress.citrix.com/monitor: '{ &quot;citrix-svc&quot;:{ &quot;type&quot;:&quot;http&quot; } } '</td>
<td>Creates a <strong>custom HTTP monitor</strong> for the servicegroup.</td>
</tr>
</tbody>
</table>
When multiple ingresses are sharing the same front-end IP address and port, you cannot have conflicting configurations provided through multiple ingress configurations.

By default, the content switching virtual server does not depend on the state of the target load balancing virtual servers bound to it. The annotation `ingress.citrix.com/csvserver: '{ "stateupdate": "ENABLED" }'` sets the content switching virtual server to consider its state based on the state of the load balancing virtual server bound to it via the content switching policies.

**Smart annotations for routes**

Similar to Ingress, you can also use smart annotations with OpenShift routes. The Citrix ingress controller converts the routes in OpenShift to a set of Citrix ADC objects.

The following table details the smart annotations provided by the Citrix ingress controller:

<table>
<thead>
<tr>
<th>Citrix ADC entity name</th>
<th>Smart annotation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbvserver</td>
<td>route.citrix.com/lbvserver</td>
<td>route.citrix.com/lbvserver: '{ &quot;citrix-svc&quot; : &quot;lbmethod&quot;: SRCIPDESTIPHASH }'</td>
</tr>
<tr>
<td>monitor</td>
<td>route.citrix.com/monitor</td>
<td>route.citrix.com/monitor: '{ &quot;appname&quot; : { &quot;type&quot; : &quot;http&quot; } }'</td>
</tr>
</tbody>
</table>

**Sample route manifest with smart annotations**

The following is a sample route YAML file.

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
```
The sample route manifest includes use cases related to the service `citrix-svc` and the following table explains the smart annotations used in the sample Route:

<table>
<thead>
<tr>
<th>Smart annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>route.citrix.com/lbvserver: '{ &quot;citrix-svc&quot;:{ &quot;lbmethod&quot;:&quot;LEASTCONNECTION&quot;, &quot;persistenceType&quot;:&quot;SOURCEIP&quot; } }'</code></td>
<td>Sets the load balancing method as Least Connection and also configures Source IP address persistence.</td>
</tr>
<tr>
<td><code>route.citrix.com/servicegroup: '{ &quot;citrix-svc&quot;:{ &quot;usip&quot;:&quot;yes&quot; } }'</code></td>
<td>Enables Use Source IP Mode (USIP) on the Citrix ADC device. When you enable USIP on the Citrix ADC, it uses the IP address of the client for communication with the back-end pods.</td>
</tr>
</tbody>
</table>
Citrix ADC ingress controller

Smart annotation Description

```
route.citrix.com/monitor: '{ "citrix-svc":{ "type":"http" } }'
```

Creates a custom HTTP monitor for the servicegroup.

Service annotations

The following are the service annotations supported by Citrix.

**Note:** In service annotations, index is the ordered index of the ports in a service specification file. For example, if there are two ports in the service specification, then the index for the first port is zero and the second one is one.

<table>
<thead>
<tr>
<th>Annotations</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>service.citrix.com/service-type-&lt;index&gt;</code></td>
<td>Use this annotation to specify the service type for the Citrix ADC entities created. The acceptable values are TCP, HTTP, SSL, UDP, ANY, SSL_TCP, and SIP_UDP.</td>
<td><code>service.citrix.com/service-type-0: 'SSL'</code></td>
</tr>
<tr>
<td><code>service.citrix.com/ssl-certificate-data-&lt;index&gt;</code></td>
<td>Use this annotation to specify the server certificate value in the PEM format.</td>
<td>`service.citrix.com/ssl-certificate-data-0:</td>
</tr>
<tr>
<td><code>service.citrix.com/ssl-key-data-&lt;index&gt;</code></td>
<td>Use this annotation to specify the server key value in the PEM format.</td>
<td>`service.citrix.com/ssl-key-data-0:</td>
</tr>
<tr>
<td><code>service.citrix.com/ssl-ca-certificate-data-&lt;index&gt;</code></td>
<td>Use this annotation to specify the server CA certificate value to verify the client certificate in PEM format.</td>
<td>`service.citrix.com/ssl-ca-certificate-data-0:</td>
</tr>
<tr>
<td><code>service.citrix.com/ssl-backend-ca-certificate-data-&lt;index&gt;</code></td>
<td>Use this annotation to specify the CA certificate value to verify the server certificate of the back-end in PEM format.</td>
<td>`service.citrix.com/ssl-backend-ca-certificate-data-0:</td>
</tr>
<tr>
<td><code>service.citrix.com/ssl-termination-&lt;index&gt;</code></td>
<td>Use this annotation to specify the SSL termination. The accepted values are EDGE and REENCRIPT.</td>
<td><code>service.citrix.com/ssl-termination-0: 'EDGE'</code></td>
</tr>
</tbody>
</table>
## Annotations

<table>
<thead>
<tr>
<th>Annotations</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>service.citrix.com/insecure-redirect</td>
<td>Use this annotation to redirect insecure traffic to a secure port. You can either specify the secure port using <code>{secure-portname: port-number}</code> or <code>{secure-portnumber-secure-port-protocol: insecure-portnumber}</code> to redirect traffic from an insecure port.</td>
<td>service.citrix.com/insecure-redirect: <code>{&quot;port-443&quot;: 80 }</code> or service.citrix.com/insecure-redirect: <code>{&quot;443-tcp&quot;: 80 }</code></td>
</tr>
<tr>
<td>service.citrix.com/frontend-ip</td>
<td>Use this annotation to pass the VIP for services of type LoadBalancer.</td>
<td>service.citrix.com/frontend-ip: <code>'192.168.1.1'</code></td>
</tr>
<tr>
<td>service.citrix.com/ipam-range</td>
<td>Use this annotation to select a particular IP address range from a set of ranges specified to the Citrix IPAM controller. This annotation is used for services of type LoadBalancer.</td>
<td>service.citrix.com/ipam-range: <code>'Dev'</code></td>
</tr>
<tr>
<td>service.citrix.com/secret</td>
<td>Use this annotation to specify the name of the secret resource for the front-end server certificate. For more information and example, see SSL certificate for services of type LoadBalancer.</td>
<td>service.citrix.com/secret: <code>'hotdrink-secret'</code></td>
</tr>
</tbody>
</table>
### Annotations

<table>
<thead>
<tr>
<th>Annotations</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>service.citrix.com/ca-secret</td>
<td>Use this annotation to provide a CA certificate for client certificate authentication. This certificate is bound to the front-end SSL virtual server in Citrix ADC. For more information and example, see SSL certificate for services of type LoadBalancer.</td>
<td>service.citrix.com/ca-secret: 'hotdrink-ca-secret'</td>
</tr>
<tr>
<td>service.citrix.com/backend-secret</td>
<td>Use this annotation if the back-end communication between Citrix ADC and your workload is on an encrypted channel, and you need the client authentication in your workload. This certificate is sent to the server during the SSL handshake and it is bound to the back end SSL service group. For more information and example, see SSL certificate for services of type LoadBalancer.</td>
<td>service.citrix.com/backend-secret: 'hotdrink-secret'</td>
</tr>
<tr>
<td>service.citrix.com/backend-ca-secret</td>
<td>Use this annotation to enable server authentication which authenticates the back-end server certificate. This configuration binds the CA certificate of the server to the SSL service on the Citrix ADC. For more information and example, see SSL certificate for services of type LoadBalancer.</td>
<td>service.citrix.com/backend-ca-secret: 'hotdrink-ca-secret'</td>
</tr>
</tbody>
</table>
## Annotations

<table>
<thead>
<tr>
<th>Annotations</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>service.citrix.com/preconfigured-certkey</td>
<td>Use this annotation to specify the name of the preconfigured certificate key in the Citrix ADC to be used as a front-end server certificate.</td>
<td>service.citrix.com/preconfigured-certkey: 'coffee-cert'</td>
</tr>
<tr>
<td>service.citrix.com/preconfigured-ca-certkey</td>
<td>Use this annotation to specify the name of the preconfigured certificate key in the Citrix ADC to be used as a CA certificate for client certificate authentication. This certificate is bound to the front-end SSL virtual server in Citrix ADC.</td>
<td>service.citrix.com/preconfigured-backend-certkey: 'coffee-cert'</td>
</tr>
<tr>
<td>service.citrix.com/preconfigured-backend-ca-certkey</td>
<td>Use this annotation to specify the name of the preconfigured certificate key in the Citrix ADC to be bound to the back-end SSL service group. This certificate is sent to the server during the SSL handshake for server authentication.</td>
<td>service.citrix.com/preconfigured-backend-ca-certkey: 'coffee-ca-cert'</td>
</tr>
<tr>
<td>service.citrix.com/preconfigured-backend-ca-certkey</td>
<td>Use this annotation to specify the name of the preconfigured CA certificate key in the Citrix ADC to bound to the back-end SSL service group for server authentication.</td>
<td>service.citrix.com/preconfigured-backend-ca-certkey: 'coffee-ca-cert'</td>
</tr>
</tbody>
</table>

### Sample YAML with the service annotation to redirect insecure traffic

This example shows how to redirect traffic from clients making requests on an insecure port 80 to the secure port 443.

The following annotation is specified in the service YAML file to redirect traffic:
Following is a sample service definition:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
  annotations:
    service.citrix.com/service-type-0: SSL
    service.citrix.com/frontend-ip: '192.2.170.26'
    service.citrix.com/secret: '{
      "port-443": "web-ingress-secret" }
    service.citrix.com/ssl-termination-0: 'EDGE'
    service.citrix.com/insecure-redirect: '{
      "port-443": 80 }
  
spec:
  type: LoadBalancer
  selector:
    app: frontend
  ports:
    - port: 443
      targetPort: 80
      name: port-443

<!--NeedCopy-->
```

### Smart annotations for services

Smart annotations for services are used to configure the Citrix ADC with custom values for Citrix ADC configuration parameters. The annotations are used for services of type `LoadBalancer` and for the services in Citrix ADC CPX used for East-West traffic.

**Note:**

If you have configured a service with NodePort or ClusterIP for the North-South traffic, then the Citrix ADC is configured using the applicable ingress smart annotations rather than then service
Smart annotations for services take JSON format as input. The key and value that you pass in the JSON format must match the Citrix ADC NITRO format. For more information on the Citrix ADC NITRO API, see Citrix ADC 12.1 REST APIs - NITRO Documentation.

The following is a sample smart annotation for services:

```json
1  service.citrix.com/lbvserver: '{
2    "80-tcp":{
3      "lbmethod":"SRCIPDESTIPHASH" }
4  }
5 '
```

This annotation sets the load balancing method as SRCIPDESTIPHASH in the load balancing virtual server for the 80-tcp port of the given service.

The following table details the smart annotations for services:

<table>
<thead>
<tr>
<th>Citrix ADC Entity Name</th>
<th>Smart Annotation for Service</th>
<th>Example</th>
</tr>
</thead>
</table>
| lbvserver              | service.citrix.com/lbvserver | service.citrix.com/lbvserver: '{ "80-tcp":{
|                        |                             |         |
|                        |                             |   "lbmethod":"SRCIPDESTIPHASH" } } ' |
| csvserver              | service.citrix.com/csvserver | service.citrix.com/csvserver: '{ "l2conn":"on" } ' |
| servicegroup           | service.citrix.com/servicegroup | service.citrix.com/servicegroup: '{ "80-tcp":{
|                        |                             |         |
|                        |                             |   "usip":"yes" } } ' |
| monitor                | service.citrix.com/monitor   | service.citrix.com/monitor: '{ "80-tcp":{
|                        |                             |         |
|                        |                             |   "type":"http" } } ' |
You can use the smart annotations for services as follows:

- By providing the port-protocol value in the annotation: In the service definition, if you provide the port-protocol value in the annotation then the annotation is restricted to the particular port of that service.
- By not providing the port-protocol value in the annotation: If you do not provide the port-protocol value in the annotation, then the annotation is applicable to all the ports used by the service.

### Sample ingress YAML with smart annotations for services

The following is a sample deployment and service definition for a basic apache web server based application. It includes smart annotations for services to enable Citrix ADC features using the entities such as, lbvserver, csvserver, servicegroup, monitor, and analyticsprofile:

```yaml
# If using this on GKE, ensure sure you have cluster-admin role for your account
# The sample is a basic apache web server as application for illustration
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: apache
labels:
  name: apache
spec:
  selector:
    matchLabels:
      app: apache
  replicas: 8
template:
```
metadata:
  labels:
    app: apache
spec:
  containers:
    - name: apache
      image: httpd:latest
      ports:
        - name: http
          containerPort: 80
          imagePullPolicy: IfNotPresent

---
#Expose the apache web server as a service
apiVersion: v1
kind: Service
metadata:
  name: apache
  annotations:
    service.citrix.com/csvserver: '{
      "l2conn":"on" }
    
    service.citrix.com/lbvserver: '{
      "80-tcp":{
        "lbmethod":"SRCIPDESTIPHASH" }
    }
    
    service.citrix.com/servicegroup: '{
      "80-tcp":{
        "usip":"yes" }
    }
    
    service.citrix.com/monitor: '{
      "80-tcp":{
        "type":"http" }
    }
    
    service.citrix.com/frontend-ip: '10.217.212.16'
    service.citrix.com/analyticsprofile: '{
      "80-tcp":{
        "webinsight": {
          "httpurl":"ENABLED", "httpuseragent":"ENABLED" }
      }
    }
    

Examples

Sample Ingress YAML for SIP_UDP support in insecure service type annotation

The following is a sample Ingress YAML which includes the configuration for enabling SIP over UDP support using the `ingress.citrix.com/insecure-service-type` annotation.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    ingress.citrix.com/frontend-ip: 1.1.1.1
    ingress.citrix.com/insecure-port: "5060"
    ingress.citrix.com/insecure-service-type: sip_udp
    ingress.citrix.com/lbvserver: '{
      "asterisk17":{
        "lbmethod":"CALLIDHASH","persistenceType":"CALLID" }
    }'
  kubernetes.io/ingress.class: cic-vpx
spec:
defaultBackend:
  service:
    name: asterisk17
    port:
```
ConfigMap support for the Citrix ingress controller

July 5, 2022

The ConfigMap API resource holds key-value pairs of configuration data that can be consumed in pods or to store configuration data for system components such as controllers.

ConfigMaps allow you to separate your configurations from your pods and make your workloads portable. Using ConfigMaps, you can easily change and manage your workload configurations and reduce the need to hardcode configuration data to pod specifications.

The Citrix ingress controller supports the configuration command line arguments, and environment variables mentioned in deploying the Citrix ingress controller. But, you cannot update these configurations at runtime without rebooting the Citrix ingress controller pod. With ConfigMap support, you can update the configuration automatically while keeping the Citrix ingress controller pod running. You do not need to restart the pod after the update.

Supported environment variables in the Citrix ingress controller

The values for the following environment variables in the Citrix ingress controller can be specified in a ConfigMap.

- **LOGLEVEL**: Specifies the log levels to control the logs generated by the Citrix ingress controller (debug, info, critical, and so on). The default value is `debug`.
- **NS_HTTP2_SERVER_SIDE**: Enables HTTP2 for Citrix ADC service group configurations with possible values as ON or OFF.
- **NS_PROTOCOL**: Specifies the protocol to establish the ADC session (HTTP/HTTPS). The default value is `http`.
- **NS_PORT**: Specifies the port to establish a session. The default value is `80`.
- **NS_COOKIE_VERSION**: Specifies the persistence cookie version (0 or 1). The default value is `0`.
- **NS_DNS_NAMESERVER**: Enables adding DNS nameservers on Citrix ADC VPX.
- **POD_IPS_FOR_SERVICEGROUP_MEMBERS**: Specifies to add the IP address of the pod and port as service group members instead of **NodeIP** and **NodePort** while configuring services of type `LoadBalancer` or **NodePort** on an external tier-1 Citrix ADC.
Citrix ADC ingress controller

- **IGNORE_NODE_EXTERNAL_IP**: Specifies to ignore an external IP address and add an internal IP address for NodeIP while configuring NodeIP for services of type LoadBalancer or NodePort on an external tier-1 Citrix ADC.

- **FRONTEND_HTTP_PROFILE**: Sets the HTTP options for the front-end virtual server (client plane), unless overridden by the `ingress.citrix.com/frontend-httpprofile` smart annotation in the ingress definition.

- **FRONTEND_TCP_PROFILE**: Sets the TCP options for the front-end virtual server (client side), unless overridden by the `ingress.citrix.com/frontend-tcpprofile` smart annotation in the ingress definition.

- **FRONTEND_SSL_PROFILE**: Sets the SSL options for the front-end virtual server (client side) unless overridden by the `ingress.citrix.com/frontend-sslprofile` smart annotation in the ingress definition.

- **JSONLOG**: Set this argument to true if log messages are required in JSON format.

For more information about profile environment variables (FRONTEND_HTTP_PROFILE, FRONTEND_TCP_PROFILE, and FRONTEND_SSL_PROFILE), see Configure HTTP, TCP, or SSL profiles on Citrix ADC.

**Note:**
This is an initial version of the ConfigMap support and currently supports only a few parameters. Earlier, these parameters were configurable through environment variables except the `NS_HTTP2_SERVER_SIDE` parameter.

### Configuring ConfigMap support for the Citrix ingress controller

This example shows how to create a ConfigMap and apply the ConfigMap to the Citrix ingress controller. It also shows how to reapply the ConfigMap after you make changes. You can also optionally delete the changes.

Perform the following to configure ConfigMap support for the Citrix ingress controller.

1. Create a YAML file `cic-configmap.yaml` with the required key-value pairs in the ConfigMap.

```yaml
apiVersion: v1
type: ConfigMap
metadata:
  name: cic-configmap
  labels:
    app: citrix-ingress-controller
data:
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
2. Deploy the `cic-configmap.yaml` using the following command.

   ```bash
   kubectl create -f cic-configmap.yaml
   ```

3. Edit the `cic.yaml` file for deploying the Citrix ingress controller as a stand-alone pod and specify the following:

   ```yaml
   Args:
   - --configmap
     default/cic-configmap
   
   Note:
   It is mandatory to specify the namespace. If the namespace is not specified, ConfigMap is not considered.

   Following is a sample YAML file for deploying the Citrix ingress controller with the ConfigMap configuration. For the complete YAML file, see `citrix-k8s-ingress-controller.yaml`.

   ```yaml
   apiVersion: apps/v1
   kind: Deployment
   metadata:
   name: cic-k8s-ingress-controller
   spec:
   selector:
   matchLabels:
   app: cic-k8s-ingress-controller
   replicas: 1
   template:
   metadata:
   name: cic-k8s-ingress-controller
   labels:
   app: cic-k8s-ingress-controller
   annotations:
   spec:
   ```
4. Deploy the Citrix ingress controller as a stand-alone pod by applying the YAML:

```bash
kubectl apply -f cic.yaml
```

5. If you want to change the value of an environment variable, edit the values in the ConfigMap. In this example, the value of NS_HTTP2_SERVER_SIDE is changed to 'OFF':

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cic-configmap
  labels:
    app: citrix-ingress-controller
data:
  LOGLEVEL: 'info'
  NS_PROTOCOL: 'http'
  NS_PORT: '80'
  NS_COOKIE_VERSION: '0'
  NS_HTTP2_SERVER_SIDE: 'OFF'
```

6. Reapply the ConfigMap using the following command.
7. (Optional) If you need to delete the ConfigMap, use the following command.

```bash
kubectl delete -f cic-configmap.yaml
```

When you delete the ConfigMap, the environment variable configuration falls back as per the following order of precedence:
ConfigMap configuration > environment variable configuration > default

(Optional) In case, you want to define all keys in a ConfigMap as environment variables in the Citrix ingress controller, use the following in the Citrix ingress controller deployment YAML file.

```yaml
envFrom:
  - configMapRef:
      name: cic-configmap
```

**Ingress configurations**

February 3, 2022

Kubernetes Ingress provides you a way to route requests to services based on the request host or path, centralizing a number of services into a single entry point.

Citrix ingress controller is built around the Kubernetes Ingress and automatically configures one or more Citrix ADC based on the Ingress resource configuration.

**Host name based routing**

The following sample Ingress definition demonstrates how to set up an Ingress to route the traffic based on the host name:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: virtual-host-ingress
```
After the sample Ingress definition is deployed, all the HTTP request with a host header is load balanced by Citrix ADC to `service1`. And, the HTTP request with a host header is load balancer by Citrix ADC to `service2`.

**Path based routing**

The following sample Ingress definition demonstrates how to set up an Ingress to route the traffic based on URL path:
After the sample Ingress definition is deployed, any HTTP requests with host test.example.com and URL path with prefix /foo, Citrix ADC routes the request to service1 and all other requests are routed to service2.

Citrix ingress controller follows first match policy to evaluate paths. For effective matching, Citrix ingress controller orders the paths based on descending order of the path's length. It also orders the paths that belong to same hosts across multiple ingress resources.

**Wildcard host routing**

The following sample Ingress definition demonstrates how to set up an ingress with wildcard host.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: wildcard-ingress
  namespace: default
spec:
rules:
  - host: '*.example.com'
    http:
      paths:
        - backend:
            service:
              name: service1
              port:
                number: 80
```
After the sample Ingress definition is deployed, HTTP requests to all the subdomains of example.com is routed to service1 by Citrix ADC.

Note:
Rules with non-wildcard hosts are given higher priority than wildcard hosts. Among different wildcard hosts, rules are ordered on the descending order of length of hosts.

**Exact path matching**

Ingresses belonging to networking.k8s.io/v1 API version can make use of PathType: Exact to consider the path for the exact match.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: Path-exact-Ingress
  namespace: default
spec:
rules:
- host: test.example.com
  http:
    paths:
    - backend:
      service:
        name: service1
        port:
          name: 80
      path: /exact
      pathType: Exact
<!--NeedCopy-->
```

(Deprecated as of Kubernetes 1.22+) By default for Ingresses belonging to extension/v1beta1, paths are treated as Prefix expressions. Using the annotation ingress.citrix.com/path-match-method: "exact" in the ingress definition defines the Citrix ingress controller to consider the path for the exact match.

The following sample Ingress definition demonstrates how to set up Ingress for exact path matching:
After the sample Ingress definition is deployed, HTTP requests with path `/exact` is routed by Citrix ADC to `service1` but not to `/exact/somepath`.

**Non-Hostname routing**

Following example shows path based routing for the default traffic that does not match any host based routes. This ingress rule applies to all inbound HTTP traffic through the specified IP address.
All incoming traffic that does not match the ingress rules with host name is matched here for the paths for routing.

**Default back end**

Default back end is a service that handles all traffic that is not matched against any of the Ingress rules.

| apiVersion: networking.k8s.io/v1 |
| kind: Ingress |
| metadata: |
|   name: default-ingress |
|   namespace: default |
| spec: |
|   defaultBackend: |
|     service: |
|       name: testsvc |
|       port: |
|         number: 80 |

**Note:**

A global default back end can be specified if Citrix ADC CPX is load balancing the traffic. You can create a default back end per `frontend-ip:port` combination in case of Citrix ADC VPX or MPX is the ingress device.

**Ingress class support**

February 8, 2022
Citrix ADC ingress controller

**What is Ingress class?**

In a Kubernetes cluster, there might be multiple ingress controllers and you need to have a way to associate a particular ingress resource with an ingress controller.

You can specify the ingress controller that should handle the ingress resource by using the `kubernetes.io/ingress.class` annotation in your ingress resource definition.

**Citrix ingress controller and Ingress classes**

The Citrix ingress controller supports accepting multiple ingress resources, which have `kubernetes.io/ingress.class` annotation. Each ingress resource can be associated with only one `ingress.class`. However, the Ingress Controller might need to handle various ingress resources from different classes.

You can associate the Ingress Controller with multiple ingress classes using the `--ingress-classes` argument under the `spec` section of the YAML file.

If `ingress-classes` is not specified for the Ingress Controller, then it accepts all ingress resources irrespective of the presence of the `kubernetes.io/ingress.class` annotation in the ingress object.

If `ingress-classes` is specified, then the Ingress Controller accepts only those ingress resources that match the `kubernetes.io/ingress.class` annotation. The Ingress controller does not process an Ingress resource without the `ingress.class` annotation in such a case.

**Note:** Ingress class names are case-insensitive.

**Sample YAML configurations with Ingress classes**

Following is the snippet from a sample YAML file to associate `ingress-classes` with the Ingress Controller. This configuration works in both cases where the Ingress Controller runs as a standalone pod or runs as a sidecar with Citrix ADC CPX. In the given YAML snippet, the following ingress classes are associated with the Ingress Controller.

- `my-custom-class`
- `Citrix`

```yaml
spec:
  serviceAccountName: cic-k8s-role
  containers:
  - name: cic-k8s-ingress-controller
    image: "quay.io/citrix/citrix-k8s-ingress-controller:latest"
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
### Citrix ADC ingress controller

```yaml
# specify the ingress classes names to be supported by Ingress Controller in args section.
# First line should be --ingress-classes, and every subsequent line should be
# the name of allowed ingress class. In the given example, two classes named
# "citrix" and "my-custom-class" are accepted. This will be case-insensitive.
args:
- --ingress-classes
  Citrix
  my-custom-class
<!--NeedCopy-->```

Following is the snippet from an Ingress YAML file where the Ingress class association is depicted. In the given example, an Ingress resource named `web-ingress` is associated with the ingress class `my-custom-class`. If the Citrix ingress controller is configured to accept `my-custom-class`, it processes this Ingress resource.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: my-custom-class
name: web-ingress
<!--NeedCopy-->```

### Ingress V1 and IngressClass support

With the Kubernetes version 1.19, the Ingress resource is generally available. As a part of this change, a new resource named as `IngressClass` is added to the ingress API. Using this resource, you can associate specific Ingress controllers to Ingresses. For more information on the `IngressClass` resource, see the Kubernetes documentation.

The following is a sample `IngressClass` resource.

```yaml
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  name: citrix
spec:
```
An **IngressClass** resource must refer to the ingress class associated with the controller that should implement the Ingress rules as shown as follows:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
spec:
  ingressClassName: citrix
  rules:
    - host: abc.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: test
                port:
                  number: 80
```

The Citrix ingress controller uses the following rules to match the Ingresses.

- If the Citrix ingress controller is started without specifying the `--ingress-classes` argument:
  - If the Kubernetes version is lesser than 1.19 (IngressClass V1 resource is supported)
    * Matches any ingress object
  - If the Kubernetes version is greater than or equal to 1.19 (IngressClass V1 resource is supported)
    * Matches any ingress object in which the `spec.ingressClassName` field is not set.
    * Matches any ingress if the `spec.ingressClassName` field of the Ingress object is set and a `v1.IngressClass` resource exists with the same name and the `spec.controller` field of the resource is `citrix.com/ingress-controller`.

- If the Citrix ingress controller is started with one or more ingress classes set using the `--ingress-classes` argument.
Citrix ADC ingress controller

- If the Kubernetes version is lesser than 1.19 (IngressClass V1 resource is supported)
  - Matches any ingress with the ingress class annotation `kubernetes.io/ingress.class` matching to that of the configured ingress classes.

- If the Kubernetes version is greater than or equal to 1.19 (IngressClass V1 resource is supported).
  - Matches any ingress in which the ingress class annotation `kubernetes.io/ingress.class` matches with the configured ingress classes. This annotation is deprecated but it has higher precedence over the `spec.IngressClassName` field to support backward compatibility.
  - Matches any ingress object, if a `v1.IngressClass` resource exists with the following attributes:
    - The name of the resource matches the `--ingress-classes` argument value.
    - The `spec.controller` field of the resource is set as the `citrix.com/ingress-controller`.
    - The name of the resource matches with the `spec.ingressClassName` field of the Ingress object.
  - Matches any ingress object where the `spec.ingressClassName` field is not set and if a `v1.IngressClass` resource exists with the following attributes:
    - The name of the resources matches the `--ingress-classes` argument value.
    - The `spec.controller` field of the resource is set as `citrix.com/ingress-controller`.
    - The resource is configured as the default class using the `ingressclass.kubernetes.io/is-default-class` annotation. For more information, see the Kubernetes documentation.

Note:

- If both the annotation and `spec.ingressClassName` is defined, the annotation is matched before the `spec.ingressClassName`. If the annotation does not match, the matching operation for the `spec.ingressClassName` field is not performed.
- When you are using Helm charts to install the Citrix ingress controller, if the `IngressClass` resource is supported and the Citrix ingress controller is deployed with the `--ingress-classes` argument, the `v1.IngressClass` resource is created by default.
Citrix ADC ingress controller

**Updating the Ingress status for the Ingress resources with the specified IP address**

To update the `Status.LoadBalancer.Ingress` field of the Ingress resources managed by the Citrix ingress controller with the allocated IP addresses, specify the command line argument `--update-ingress-status yes` when you start the Citrix ingress controller. This feature is only supported for the Citrix ingress controller deployed as a stand-alone pod for managing Citrix ADC VPX or MPX. For Citrix ADC CPXs deployed as sidecars, this feature is not supported.

Following is an example YAML with the `--update-ingress-status yes` command line argument enabled.

```
args:
- --feature-node-watch false
- --ipam citrix-ipam-controller
- --update-ingress-status yes
imagePullPolicy: Always
```

**Ingress status update for sidecar deployments**

In Kubernetes, Ingress can be used as a single entry point for exposing multiple applications to the outside world. The Ingress would have an `Address (Status.LoadBalancer.IP)` field which is updated after the successful ingress creation. This field is updated with a public IP address or host name through which the Kubernetes application can be reached. In cloud deployments, this field can also be the IP address or host name of a cloud load-balancer.

In cloud deployments, Citrix ADC CPX along with the ingress controller is exposed using a service of type `LoadBalancer` which in turn creates a cloud load-balancer. The cloud load balancer then exposes the Citrix ADC CPX along with the ingress controller. So, the Ingress resources exposed with the Citrix ADC CPX should be updated using the public IP address or host name of the cloud load balancer.

This is applicable even on on-prem deployments. In dual-tier ingress deployments, in which the Citrix ADC CPX is exposed as service type `LoadBalancer` to the tier-1 Citrix ADC VPX ingress, the ingress resources operated by the Citrix ADC CPX is updated with the VIP address.

This topic provides information about how to enable the ingress status update for Citrix ADC CPX with the Citrix ingress controller as sidecar deployments.

**Note:** The ingress status update for the sidecar feature is supported only on services of type `LoadBalancer`. 
Citrix ADC ingress controller

Sample ingress output after an ingress status update

The following is a sample ingress output after the ingress status update:

```
$ kubectl get ingress
NAME     HOSTS              ADDRESS          PORTS   AGE
sample-ingress-80     sample.citrix.com  sample.abc.somexampledomain.com  80 1d
```

Enable ingress status update for the sidecar deployments

You can enable the ingress status update feature for side car deployments by specifying the following argument in the Citrix ADC CPX YAML file. You must add the argument to the `args` section of Citrix ADC CPX in the deployment YAML file for Citrix ADC CPX with the Citrix ingress controller.

```
args:
- --cpx-service <namespace>/<name-of-the-type-load-balancer-service-exposing-cpx>
```

The following table describes the argument for the ingress update in detail

<table>
<thead>
<tr>
<th>Keyword/variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--cpx-service</code></td>
<td>Specifies the argument for enabling this feature.</td>
</tr>
<tr>
<td><code>&lt;namespace&gt;/</code>&lt;br&gt;<code>&lt;name-of-the-type-load-balancer-service-exposing-cpx&gt;</code></td>
<td>Specifies the format in which the argument value to be provided.</td>
</tr>
<tr>
<td><code>&lt;namespace&gt;</code></td>
<td>Specifies the namespace in which the service is created.</td>
</tr>
<tr>
<td><code>&lt;name-of-the-type-load-balancer-service-exposing-cpx&gt;</code></td>
<td>Specifies the name of the service that exposes Citrix ADC CPX.</td>
</tr>
</tbody>
</table>

**Note:**

The ingress status update for the sidecar feature is supported only on services of type `LoadBalancer`. The service defined in the argument `--cpx-service default/some-cpx-
**Service class for services of type LoadBalancer**

January 19, 2022

When services of type LoadBalancer are deployed, all such services are processed by the Citrix ingress controller and configured on Citrix ADCs. However, there may be situations where you want to associate only specific services to a Citrix ingress controller if multiple Ingress controllers are deployed. For Ingress resources this functionality is already available using the Ingress class feature. Similar to the Ingress class functionality for Ingress resources, service class functionality is now added for services of type LoadBalancer.

You can associate a Citrix ingress controller with multiple service classes using the `--service-classes` argument under the `spec` section of the YAML file. If a service class is not specified for the ingress controller, then it accepts all services of the type LoadBalancer irrespective of the presence of the `service.citrix.com/class` annotation in the service.

If the service class is specified to the Citrix ingress controller, then it accepts only those services of the type LoadBalancer that match the `service.citrix.com/class` annotation. In this case, the Citrix ingress controller does not process a type LoadBalancer service if it is not associated with the `service.citrix.com/class` annotation.

**Sample YAML configurations with service classes**

Following is a snippet from a sample YAML file to associate `service-classes` with the Ingress Controller. In this snippet, the following service classes are associated with the Ingress Controller.

- `svc-class1`
- `svc-class2`

```yaml
spec:
serviceAccountName: cic-k8s-role
containers:
  - name: cic-k8s-ingress-controller
    # specify the service classes to be supported by Citrix Ingress Controller in args section.
    # First line should be --service-classes, and every subsequent line should be
    # the name of allowed service class. In the given example two classes named
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
# "svc-class1" and "svc-class2" are accepted. This will be case-insensitive.

```sh
args:
  - --service-classes
    svc-class1
    svc-class2
</!--NeedCopy-->
```

Following is a snippet from a type LoadBalancer service definition YAML file where the service class association is depicted. In this example, an Apache service is associated with the service class `svc-class1`. If the Citrix ingress controller is configured to accept `svc-class1`, it configures the service on the Citrix ADC.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: apache
  annotations:
    service.citrix.com/class: 'svc-class1'
  labels:
    name: apache
spec:
  type: LoadBalancer
  selector:
    name: apache
  ports:
    - name: http
      port: 80
      targetPort: http
      selector:
        app: apache
</!--NeedCopy-->

## Configure HTTP, TCP, or SSL profiles on Citrix ADC

February 7, 2022

Configurations such as, HTTP, TCP, or SSL for a Citrix ADC appliance can be specified using individual entities such as HTTP profile, TCP profile, or SSL profile respectively. The profile is a collection of settings pertaining to the individual protocols, for example, HTTP profile is a collection of HTTP settings.
Citrix ADC ingress controller

It offers ease of configuration and flexibility. Instead of configuring the settings on each entity you can configure them in a profile and bind the profile to all the entities that the settings apply to.

Citrix ingress controller enables you to configure HTTP, TCP, or SSL related configuration on the Ingress Citrix ADC using profiles.

Understand Citrix ADC configuration in Kubernetes environment

In a Kubernetes environment, the Ingress Citrix ADC uses Content Switching (CS) virtual server as the front end for external traffic. That is, it is the entity that receives the requests from the client. After processing the request, the CS virtual server passes the request data to a load balancing (LB) entity. The LB virtual server and the associated service group processes the request data and then forwards it to the appropriate app (microservice).

You need to have a separate front end configuration for the entities that receive the traffic from the client (highlighted as Client Plane in the diagram) and a back end configuration for the entities that forward the traffic from the Citrix ADC to the microservices in Kubernetes (highlighted as Server Plane in the diagram).

The Citrix ingress controller provides individual smart annotations for the front end and back-end configurations that you can use based on your requirement.
HTTP profile

An HTTP profile is a collection of HTTP settings. A default HTTP profile (nshttp_default_profile) is configured to set the HTTP configurations that are applied by default, globally to all services and virtual servers.

The Citrix ingress controller provides the following two smart annotations for HTTP profile. You can use these annotations to define the HTTP settings for the Citrix ADC. When you deploy an ingress that includes these annotations, the Citrix ingress controller creates an HTTP profile derived from the default HTTP profile (nshttp_default_profile) configured on the Citrix ADC. Then, it applies the parameters that you have provided in the annotations to the new HTTP profile and applies the profile to the Citrix ADC.

<table>
<thead>
<tr>
<th>Smart annotation</th>
<th>Description</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>ingress.citrix.com/</td>
<td>Use this annotation to create the front-end HTTP profile (Client Plane)</td>
<td>ingress.citrix.com/frontend-httpprofile: '{ &quot;dropinvalreqs&quot;:&quot;enabled&quot;, &quot;websocket&quot; : &quot;enabled&quot; }'</td>
</tr>
<tr>
<td>frontend-httpprofile</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ingress.citrix.com/</td>
<td>Use this annotation to create the back-end HTTP profile (Server Plane).</td>
<td>ingress.citrix.com/backend-httpprofile: '{ &quot;app-1&quot;: { &quot;dropinvalreqs&quot;:&quot;enabled&quot;, &quot;websocket&quot; : &quot;enabled&quot; } }'</td>
</tr>
<tr>
<td>backend-httpprofile</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Ensure that you manually enable the HTTP related global parameters on the Citrix ADC. For example, to use HTTP2 at the back end (Server Plane), ensure that you can enable HTTP2Server side global parameter in the Citrix ADC. For more information, see Configurating HTTP2.
**TCP profile**

A TCP profile is a collection of TCP settings. A default TCP profile (nstcp_default_profile) is configured to set the TCP configurations that is applied by default, globally to all services and virtual servers.

The Citrix ingress controller provides the following two smart annotations for TCP profile. You can use these annotations to define the TCP settings for the Citrix ADC. When you deploy an ingress that includes these annotations, the Citrix ingress controller creates a TCP profile derived from the default TCP profile (nstcp_default_profile) configured on the Citrix ADC. Then, it applies the parameters that you have provided in the annotations to the new TCP profile and applies the profile to the Citrix ADC.

<table>
<thead>
<tr>
<th>Smart annotation</th>
<th>Description</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>ingress.citrix.com/frontend-tcpprofile</td>
<td>Use this annotation to create the front-end TCP profile (Client Plane)</td>
<td>ingress.citrix.com/frontend-tcpprofile: <code>{ &quot;ws&quot;: &quot;enabled&quot;, &quot;sack&quot;: &quot;enabled&quot; }</code></td>
</tr>
<tr>
<td>ingress.citrix.com/backend-tcpprofile</td>
<td>Use this annotation to create the back-end TCP profile (Server Plane)</td>
<td>ingress.citrix.com/backend-tcpprofile: <code>{ &quot;citrix-svc&quot;: { &quot;ws&quot;: &quot;enabled&quot;, &quot;sack&quot;: &quot;enabled&quot; } }</code></td>
</tr>
</tbody>
</table>

**SSL profile**

An SSL profile is a collection of settings for SSL entities. It offers ease of configuration and flexibility. Instead of configuring the settings on each entity, you can configure them in a profile and bind the profile to all the entities that the settings apply to.

**Prerequisites**

On the Citrix ADC, by default, SSL profile is not enable on the Ingress Citrix ADC. Ensure that you manually enable SSL profile on the Citrix ADC. Enabling the SSL profile overrides all the existing SSL related setting on the Citrix ADC, for detailed information on SSL profiles, see SSL profiles.

SSL profiles are classified into two categories:

- Front end profiles: containing parameters applicable to the front-end entity. That is, they apply to the entity that receives requests from a client.
Citrix ADC ingress controller

- Back-end profiles: containing parameters applicable to the back-end entity. That is, they apply to the entity that sends client requests to a server.

Once you enable SSL profiles on the Citrix ADC, a default front end profile (ns_default_ssl_profile_frontend) is applied to the SSL virtual server and a default back-end profile (ns_default_ssl_profile_backend) is applied to the service or service group on the Citrix ADC.

The Citrix ingress controller provides the following two smart annotations for SSL profile. You can use these annotations to customize the default front end profile (ns_default_ssl_profile_frontend) and back-end profile (ns_default_ssl_profile_backend) based on your requirement:

<table>
<thead>
<tr>
<th>Smart annotation</th>
<th>Description</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>ingress.citrix.com/frontend-sslprofile</td>
<td>Use this annotation to create the front end SSL profile (Client Plane). The front end SSL profile is required only if you have enabled TLS on the Client Plane.</td>
<td>ingress.citrix.com/frontend-sslprofile: '{ &quot;hsts&quot;:&quot;enabled&quot;, &quot;tls12&quot;: &quot;enabled&quot; }'</td>
</tr>
<tr>
<td>ingress.citrix.com/backend-sslprofile</td>
<td>Use this annotation to create the back-end SSL profile (Server Plane). The SSL back end profile is required only if you use the ingress.citrix.com/secure-backend annotation for the back-end.</td>
<td>ingress.citrix.com/backend-sslprofile: '{ &quot;citrix-svc&quot;:{ &quot;hsts&quot;:&quot;enabled&quot;, &quot;tls1&quot;: &quot;enabled&quot; } }'</td>
</tr>
</tbody>
</table>

**Important:** SSL profile does not enable you to configure SSL certificate.

**Front-end profile configuration using annotations**

HTTP, TCP, and SSL front-end profiles are attached to the client-side content switching virtual server or SSL virtual server. Since there can be multiple ingresses that use the same frontend-ip and also use the same content switching virtual server in the front-end, there can be possible conflicts that can arise from the front-end profiles annotation specified in multiple ingresses that share the front-end IP address.

The following are the guidelines for front-end profiles annotations for HTTP, TCP, and SSL.

- For all ingresses with the same front-end IP address, it is recommended to have the same value for the front-end profile is specified in all ingresses.
If there are multiple ingresses that share front-end IP address, one can also create a separate ingress for each front-end IP address with empty rules (referred as the front-end ingress) where one can specify the front-end IP annotation as shown in the following example. You do not need to specify the front-end profile annotation in each ingress definition.

- To create a front-end ingress for an HTTP type virtual server, see the following example:

```yaml
#Sample ingress manifest for the front-end configuration for an HTTP virtual server
#The values for the parameters are for demonstration purpose only.

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: frontend-ingress
  annotations:
    # /* The CS virtual server is derived from the combination of insecure-port/secure-port, frontend-ip, and secure-service-type/insecure-service-type annotations. */
    ingress.citrix.com/insecure-port: "80"
    ingress.citrix.com/frontend-ip: "x.x.x.x"
    ingress.citrix.com/frontend-httpprofile:'{
      "dropinvalreqs":"enabled", "markconnreqInval" : "enabled" }
    '
    ingress.citrix.com/frontend-tcpprofile: '{
      "ws":"enabled", "sack" :
      "enabled" }
    '
  spec:
    rules:
      # Empty rule
      - host:
```

- To create a front-end ingress for SSL type service, see the following example:

```yaml
#Sample ingress manifest for the front-end configuration for an SSL virtual server
#The values for the parameters are for demonstration purpose only.
```
```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: frontend-ingress
  annotations:
    # The CS virtual server is derived from the combination of
    # insecure-port/secure-port, frontend-ip, and
    # secure-service-type/insecure-service-type annotations.
    ingress.citrix.com/insecure-port: "80"
    ingress.citrix.com/secure-port: "443"
    ingress.citrix.com/frontend-ip: "x.x.x.x"
    ingress.citrix.com/frontend-sslprofile:
      ':
        "tls13":"enabled", "hsts" : "enabled" }
    ingress.citrix.com/frontend-tcpprofile: '{
      "ws":"enabled", "sack" :
      "enabled" }
  spec:
    rules:
      - host:
        # Presence of tls is considered as a secure service
        tls:
      - hosts:
```

- If there are different values for the same front-end profile annotations in multiple ingresses, the following order is used to bind the profiles to the virtual server.
  - If any ingress definition has a front-end annotation with pre-configured profiles, that is bound to the virtual server.
  - Merge all the (key, values) from different ingresses of the same front-end IP address and use the resultant (key, value) for the front-end profiles smart annotation.
  - If there is a conflict for the same key due to different values from different ingresses, a value is randomly chosen and other values are ignored. You must avoid having conflicting values.

- If there is no front-end profiles annotation specified in any of the ingresses which share the front-end IP address, then the global values from the ConfigMap that is `FRONTEND_HTTP_PROFILE`, `FRONTEND_TCP_PROFILE`, or `FRONTEND_SSL_PROFILE` is used for the HTTP, TCP, and SSL front-end profiles respectively.
**Global front-end profile configuration using ConfigMap variables**

The ConfigMap variable is used for the front-end profile if it is not overridden by front-end profiles smart annotation in one or more ingresses that shares a front-end IP address. If you need to enable or disable a feature using any front-end profile for all ingresses, you can use the variables `FRONTEND_HTTP_PROFILE`, `FRONTEND_TCP_PROFILE`, or `FRONTEND_SSL_PROFILE` for HTTP, TCP, and SSL profiles respectively. For example, if you want to enable TLS 1.3 for all SSL ingresses, you can use `FRONTEND_SSL_PROFILE` to set this value instead of using the smart annotation in each ingress definition. See the ConfigMap documentation to know how to use ConfigMap with Citrix ingress controller.

**Configuration using FRONTEND_HTTP_PROFILE**

The `FRONTEND_HTTP_PROFILE` variable is used for setting the HTTP options for the front-end virtual server (client plane), unless overridden by the ingress.citrix.com/frontend-htpprofile smart annotation in the ingress definition.

To use an existing profile on Citrix ADC or use a built-in HTTP profile.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cic-configmap
labels:
  app: citrix-ingress-controller
data:
  FRONTEND_HTTP_PROFILE: |
  preconfigured: my_http_profile
<!-- NeedCopy-->
```

In this example, `my_http_profile` is a pre-existing HTTP profile in Citrix ADC.

Alternatively, you can set the profile parameters as specified as follows. See the HTTP profile NITRO documentation for all possible key-values.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cic-configmap
labels:
  app: citrix-ingress-controller
data:
```
Citrix ADC ingress controller

```yaml
FRONTEND_HTTP_PROFILE: |
  config:
  dropinvalreqs: 'ENABLED'
  websocket: 'ENABLED'
<!--NeedCopy-->
```

Configuration using FRONTEND_TCP_PROFILE

The `FRONTEND_TCP_PROFILE` variable is used for setting the TCP options for the front-end virtual server (client side), unless overridden by the `ingress.citrix.com/frontend-tcpprofile` smart annotation in the ingress definition.

To use an existing profile on Citrix ADC or use a built-in TCP profile:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cic-configmap
  labels:
    app: citrix-ingress-controller
data:
FRONTEND_TCP_PROFILE: |
  preconfigured: my_tcp_profile
<!--NeedCopy-->
```

In this example, `my_tcp_profile` is a pre-existing TCP profile in Citrix ADC.

Alternatively, you can set the profile parameters as follows. See the Citrix ADC TCP profile NITRO documentation for all possible key values.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cic-configmap
  labels:
    app: citrix-ingress-controller
data:
FRONTEND_TCP_PROFILE: |
  config:
    sack: 'ENABLED'
    nagle: 'ENABLED'
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
Configuration using FRONTEND_SSL_PROFILE

The `FRONTEND_SSL_PROFILE` variable is used for setting the SSL options for the front-end virtual server (client side) unless overridden by the `ingress.citrix.com/frontend-sslprofile` smart annotation in the ingress definition.

**Note:**
For the SSL profile to work correctly, you must enable the default profile in Citrix ADC using the `set ssl parameter -defaultProfile ENABLED` command. Make sure that Citrix ingress controller is restarted after enabling the default profile. The default profile is automatically enabled when Citrix ADC CPX is used as an ingress device. For more information about the SSL default profile, see the SSL profile documentation.

To use an existing profile on Citrix ADC or use a built-in SSL profile,

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cic-configmap
  labels:
    app: citrix-ingress-controller
data:
  FRONTEND_SSL_PROFILE: |
    preconfigured: my_ssl_profile
<!--NeedCopy-->'
```

In this example, `my_ssl_profile` is the pre-existing SSL profile in Citrix ADC.

**Note:**
Default front end profile (`ns_default_ssl_profile_frontend`) is not supported using the `FRONTEND_SSL_PROFILE.preconfigured` variable.

Alternatively, you can set the profile parameters as shown in the following example. See the SSL profile NITRO documentation for information on all possible key-values.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cic-configmap
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
The following example shows binding SSL cipher groups to the SSL profile. The order is as specified in the list with the higher priority is provided to the first in the list and so on. You can use any SSL ciphers available in Citrix ADC or user-created cipher groups in this field. For information about the list of ciphers available in the Citrix ADC, see Ciphers in Citrix ADC.

Back-end configuration

Any ingress definition that includes service details, spec: rules: host, spec: backend entry, and so on are considered as back-end configuration.

Sample backend ingress manifest without TLS configuration
Sample backend ingress manifest with TLS configuration

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    # /* The CS virtual server is derived from the combination of insecure-port/secure-port, frontend-ip, and secure-service-type/insecure-service-type annotations. */
    ingress.citrix.com/backend-htpprofile: '{}
    "hotdrink":{

```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
Using built-in or existing user-defined profiles on the Ingress Citrix ADC

You can use the individual smart annotations to configure the built-in profiles or existing user-defined profiles on the Ingress Citrix ADC for the front end and back-end configurations based on your requirement. For more information on built-in profiles, see Built-in TCP Profiles and Built-in HTTP profiles.

For the front end configuration, you can provide the name of the built-in or existing user-defined profiles on the Ingress Citrix ADC. The following is a sample ingress annotation:

```

Using built-in or existing user-defined profiles on the Ingress Citrix ADC

You can use the individual smart annotations to configure the built-in profiles or existing user-defined profiles on the Ingress Citrix ADC for the front end and back-end configurations based on your requirement. For more information on built-in profiles, see Built-in TCP Profiles and Built-in HTTP profiles.

For the front end configuration, you can provide the name of the built-in or existing user-defined profiles on the Ingress Citrix ADC. The following is a sample ingress annotation:

```

Using built-in or existing user-defined profiles on the Ingress Citrix ADC

You can use the individual smart annotations to configure the built-in profiles or existing user-defined profiles on the Ingress Citrix ADC for the front end and back-end configurations based on your requirement. For more information on built-in profiles, see Built-in TCP Profiles and Built-in HTTP profiles.

For the front end configuration, you can provide the name of the built-in or existing user-defined profiles on the Ingress Citrix ADC. The following is a sample ingress annotation:

```

Using built-in or existing user-defined profiles on the Ingress Citrix ADC

You can use the individual smart annotations to configure the built-in profiles or existing user-defined profiles on the Ingress Citrix ADC for the front end and back-end configurations based on your requirement. For more information on built-in profiles, see Built-in TCP Profiles and Built-in HTTP profiles.

For the front end configuration, you can provide the name of the built-in or existing user-defined profiles on the Ingress Citrix ADC. The following is a sample ingress annotation:

```

Using built-in or existing user-defined profiles on the Ingress Citrix ADC

You can use the individual smart annotations to configure the built-in profiles or existing user-defined profiles on the Ingress Citrix ADC for the front end and back-end configurations based on your requirement. For more information on built-in profiles, see Built-in TCP Profiles and Built-in HTTP profiles.

For the front end configuration, you can provide the name of the built-in or existing user-defined profiles on the Ingress Citrix ADC. The following is a sample ingress annotation:

```

Using built-in or existing user-defined profiles on the Ingress Citrix ADC

You can use the individual smart annotations to configure the built-in profiles or existing user-defined profiles on the Ingress Citrix ADC for the front end and back-end configurations based on your requirement. For more information on built-in profiles, see Built-in TCP Profiles and Built-in HTTP profiles.

For the front end configuration, you can provide the name of the built-in or existing user-defined profiles on the Ingress Citrix ADC. The following is a sample ingress annotation:

```

Using built-in or existing user-defined profiles on the Ingress Citrix ADC

You can use the individual smart annotations to configure the built-in profiles or existing user-defined profiles on the Ingress Citrix ADC for the front end and back-end configurations based on your requirement. For more information on built-in profiles, see Built-in TCP Profiles and Built-in HTTP profiles.

For the front end configuration, you can provide the name of the built-in or existing user-defined profiles on the Ingress Citrix ADC. The following is a sample ingress annotation:

```

Using built-in or existing user-defined profiles on the Ingress Citrix ADC

You can use the individual smart annotations to configure the built-in profiles or existing user-defined profiles on the Ingress Citrix ADC for the front end and back-end configurations based on your requirement. For more information on built-in profiles, see Built-in TCP Profiles and Built-in HTTP profiles.

For the front end configuration, you can provide the name of the built-in or existing user-defined profiles on the Ingress Citrix ADC. The following is a sample ingress annotation:

```

Using built-in or existing user-defined profiles on the Ingress Citrix ADC

You can use the individual smart annotations to configure the built-in profiles or existing user-defined profiles on the Ingress Citrix ADC for the front end and back-end configurations based on your requirement. For more information on built-in profiles, see Built-in TCP Profiles and Built-in HTTP profiles.

For the front end configuration, you can provide the name of the built-in or existing user-defined profiles on the Ingress Citrix ADC. The following is a sample ingress annotation:

```

Using built-in or existing user-defined profiles on the Ingress Citrix ADC

You can use the individual smart annotations to configure the built-in profiles or existing user-defined profiles on the Ingress Citrix ADC for the front end and back-end configurations based on your requirement. For more information on built-in profiles, see Built-in TCP Profiles and Built-in HTTP profiles.

For the front end configuration, you can provide the name of the built-in or existing user-defined profiles on the Ingress Citrix ADC. The following is a sample ingress annotation:

```

Using built-in or existing user-defined profiles on the Ingress Citrix ADC

You can use the individual smart annotations to configure the built-in profiles or existing user-defined profiles on the Ingress Citrix ADC for the front end and back-end configurations based on your requirement. For more information on built-in profiles, see Built-in TCP Profiles and Built-in HTTP profiles.

For the front end configuration, you can provide the name of the built-in or existing user-defined profiles on the Ingress Citrix ADC. The following is a sample ingress annotation:

```

Using built-in or existing user-defined profiles on the Ingress Citrix ADC

You can use the individual smart annotations to configure the built-in profiles or existing user-defined profiles on the Ingress Citrix ADC for the front end and back-end configurations based on your requirement. For more information on built-in profiles, see Built-in TCP Profiles and Built-in HTTP profiles.

For the front end configuration, you can provide the name of the built-in or existing user-defined profiles on the Ingress Citrix ADC. The following is a sample ingress annotation:

```

Using built-in or existing user-defined profiles on the Ingress Citrix ADC

You can use the individual smart annotations to configure the built-in profiles or existing user-defined profiles on the Ingress Citrix ADC for the front end and back-end configurations based on your requirement. For more information on built-in profiles, see Built-in TCP Profiles and Built-in HTTP profiles.

For the front end configuration, you can provide the name of the built-in or existing user-defined profiles on the Ingress Citrix ADC. The following is a sample ingress annotation:

```

Using built-in or existing user-defined profiles on the Ingress Citrix ADC

You can use the individual smart annotations to configure the built-in profiles or existing user-defined profiles on the Ingress Citrix ADC for the front end and back-end configurations based on your requirement. For more information on built-in profiles, see Built-in TCP Profiles and Built-in HTTP profiles.

For the front end configuration, you can provide the name of the built-in or existing user-defined profiles on the Ingress Citrix ADC. The following is a sample ingress annotation:

```

Using built-in or existing user-defined profiles on the Ingress Citrix ADC

You can use the individual smart annotations to configure the built-in profiles or existing user-defined profiles on the Ingress Citrix ADC for the front end and back-end configurations based on your requirement. For more information on built-in profiles, see Built-in TCP Profiles and Built-in HTTP profiles.

For the front end configuration, you can provide the name of the built-in or existing user-defined profiles on the Ingress Citrix ADC. The following is a sample ingress annotation:

```

Using built-in or existing user-defined profiles on the Ingress Citrix ADC

You can use the individual smart annotations to configure the built-in profiles or existing user-defined profiles on the Ingress Citrix ADC for the front end and back-end configurations based on your requirement. For more information on built-in profiles, see Built-in TCP Profiles and Built-in HTTP profiles.

For the front end configuration, you can provide the name of the built-in or existing user-defined profiles on the Ingress Citrix ADC. The following is a sample ingress annotation:

```
Citrix ADC ingress controller

```
1  ingress.citrix.com/frontend-httpprofile: "http_preconf_profile1"
```

Where, ‘http_preconf_profile1’ is the profile that exists on the Ingress Citrix ADC.

For the back-end configuration, you must provide the name of the built-in or existing profile on the Ingress Citrix ADC and the back-end service name. The following is a sample ingress annotation:

```
1  ingress.citrix.com/backend-httpprofile: '{
2    "citrix-svc": "http_preconf_profile1" }
3  '
```

Where, ‘http_preconf_profile1’ is the profile that exists on the Ingress Citrix ADC and `citrix-svc` is the back-end service name.

**Sample HTTP profile**

```
1  ingress.citrix.com/frontend-httpprofile: "http_preconf_profile"
2  ingress.citrix.com/backend-httpprofile: '{
3    "citrix-svc": "http_preconf_profile" }
4  '
```

**Sample TCP profile**

```
1  ingress.citrix.com/frontend-tcpprofile: "tcp_preconf_profile"
2  ingress.citrix.com/backend-tcpprofile: '{
3    "citrix-svc": "tcp_preconf_profile" }
4  '
```

**Sample SSL profile**

```
1  ingress.citrix.com/frontend-sslprofile: "ssl_preconf_profile"
2  ingress.citrix.com/backend-sslprofile: '{
3    "citrix-svc": "ssl_preconf_profile" }
4  '
```
Example for applying HTTP, SSL, and TCP profiles

This example shows how to apply HTTP, SSL, or TCP profiles.

To create SSL, TCP, and HTTP profiles and bind them to the defined Ingress resource, perform the following steps:

1. Define the front-end ingress resource with the required profiles. In this Ingress resource, backend and TLS is not defined.

   A sample YAML (ingress1.yaml) is provided as follows:

   ```yaml
   apiVersion: networking.k8s.io/v1
   kind: Ingress
   metadata:
     name: ingress-vpx1
     annotations:
       kubernetes.io/ingress.class: "vpx"
       ingress.citrix.com/insecure-termination: "allow"
       ingress.citrix.com/frontend-ip: "10.221.36.190"
       ingress.citrix.com/frontend-tcpprofile: '{
         "ws": "disabled",
         "sack": "disabled"
       }
       ingress.citrix.com/frontend-httpprofile: '{
         "dropinvalreqs": "enabled",
         "markconnreqInval": "enabled"
       }
       ingress.citrix.com/frontend-sslprofile: '{
         "hsts": "enabled",
         "tls13": "enabled"
       }
   spec:
     tls:
     - hosts:
     rules:
     - host:
   ```

2. Deploy the front-end ingress resource.

   `kubectl create -f ingress1.yaml`

3. Define the secondary ingress resource with the same front-end IP address and TLS and the backend defined which creates the load balancing resource definition.

   A sample YAML (ingress2.yaml) is provided as follows:
4. Deploy the back-end ingress resource.

```bash
kubectl create -f ingress2.yaml
```

5. Once the YAMLs are applied the corresponding entities, profiles, and ingress resources are created and they were bound to the ingress resource.

```bash
# show cs vserver <k8s150-10.221.36.190_443_ssl>

k8s150-10.221.36.190_443_ssl (10.221.36.190:443) - SSL Type: CONTENT
State: UP
Last state change was at Thu Apr 22 20:14:44 2021
Time since last state change: 0 days, 00:10:56.850
Client Idle Timeout: 180 sec
Down state flush: ENABLED
Disable Primary Vserver On Down : DISABLED
Comment: uid=
  QEYQI2LDW5WR46P3NSZ37XICKOJKV4HPEM2H4PSK4HWA3QWCLQ====
TCP profile name: k8s150-10.221.36.190_443_ssl
```
Example: Adding SNI certificate to an SSL virtual server

This example shows how to add a single SNI certificate.

Note:

For the SSL profile to work correctly, you must enable the default profile in Citrix ADC using the `set ssl parameter -defaultProfile ENABLED` command. Make sure that Citrix ingress controller is restarted after enabling default profile. For more information about the SSL default profile, see documentation.

1. Define the front-end ingress resource with the required profiles. In this Ingress resource, back-end and TLS is not defined.

A sample YAML (ingress1.yaml) is provided as follows:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-vpx1
  annotations:
    kubernetes.io/ingress.class: "vpx"
    ingress.citrix.com/insecure-termination: "allow"
```
2. Deploy the front-end ingress resource.

```bash
kubectl create -f ingress1.yaml
```

3. Define the secondary ingress resource with the same front-end IP address defining back-end as well as SNI certificates. If hosts are specified then the certkey specified as the secret name is added as the SNI certificate.

A sample YAML (ingress2.yaml) is provided as follows:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-vpx2
  annotations:
    kubernetes.io/ingress.class: "vpx"
    ingress.citrix.com/insecure-termination: "allow"
    ingress.citRIX.com/frontend-ip: "10.221.36.190"
spec:
  tls:
    - hosts:
      - hotdrink.beverages.com
      secretName: hotdrink-secret
  rules:
```
Citrix ADC ingress controller

```
- host: hotdrink.beverages.com
  http:
    paths:
    - path: /
      backend:
        serviceName: web
        servicePort: 80

<!-- NeedCopy -->
```

4. Deploy the secondary ingress resource.

```
kubectl create -f ingress2.yaml
```

If multiple SNI certificates need to be bound to the front-end VIP, following is a sample YAML file.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-vpx-frontend
  annotations:
    kubernetes.io/ingress.class: "vpx"
    ingress.citrix.com/insecure-termination: "allow"
    ingress.citrix.com/frontend-ip: "10.221.36.190"
spec:
  tls:
  - hosts:
      - hotdrink.beverages.com
        secretName: hotdrink-secret
  - hosts:
      - frontend.agiledevelopers.com
        secretName: <frontend-secret>
  rules:
  - host: hotdrink.beverages.com
    http:
      paths:
      - path: /
        backend:
          serviceName: web
          servicePort: 80
  - host: frontend.agiledevelopers.com
    http:
      paths:
```
Citrix ADC ingress controller

```yaml
- path: /
  backend:
    serviceName: frontend-developers
    servicePort: 80
```

**Example: Binding SSL cipher group**

This example shows how to bind SSL cipher group.

**Note:**

For the SSL profile to work correctly, you must enable the default profile in Citrix ADC using the `set ssl parameter -defaultProfile ENABLED` command. Make sure that Citrix ingress controller is restarted after enabling default profile.

Set default SSL profile on Citrix ADC using the command `set ssl parameter -defaultProfile ENABLED` before deploying Citrix ingress controller. If you have already deployed Citrix ingress controller, then redeploy it. For more information about the SSL default profile, see [documentation](#).

For information on supported Ciphers on the Citrix ADC appliances, see [Ciphers available on the Citrix ADC appliances](#).

For information about securing cipher, see [securing cipher](#).

A sample YAML (cat frontend_ingress.yaml) is provided as follows:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-vpx
  annotations:
    kubernetes.io/ingress.class: "citrix"
    ingress.citrix.com/insecure-termination: "allow"
    ingress.citrix.com/frontend-ip: "10.221.36.190"
    ingress.citrix.com/frontend-tcpprofile: '{
      "ws":"disabled", "sack" : "disabled"
    }
    ingress.citrix.com/frontend-httpprofile: '{
      "dropinvalreqs":"enabled", "markconnreqInval" : "enabled"
    }
    ingress.citrix.com/frontend-sslprofile: '{
      "snienable": "enabled", "hsts":"enabled", "tls13" : "enabled", "ciphers" : [
        "ciphersname": "test", "cipherpriority" : "1"
      ]
    }
```
Log levels

July 5, 2022

The logs generated by Citrix ingress controller are available as part of kubernetes logs. You can specify Citrix ingress controller to log in the following log levels:

- CRITICAL
- ERROR
- WARNING
- INFO
- DEBUG

By default, Citrix ingress controller is set to log in INFO log level. If you want to specify Citrix ingress controller to log in a particular log level then you need to specify the log level in the Citrix ingress controller deployment YAML file before deploying the Citrix ingress controller. You can specify the log level in the spec section of the YAML file as follows:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: citrixingresscontroller
  labels:
    app: citrixingresscontroller
spec:
  serviceAccountName: cpx
  containers:
  - name: citrixingresscontroller
    image: "quay.io/citrix/citrix-k8s-ingress-controller:1.26.7"
    env:
      - name: kubernetes_url
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
Modify the log levels

To modify the log level configured on the Citrix ingress controller instance, you need to delete the instance and update the log level value in the following section and redeploy the Citrix ingress controller instance:

```
# Set log level
- name: "LOGLEVEL"
  value: "XXXX"
```

Once you update the log level, save the YAML file and deploy it using the following command:

```
kubectl create -f citrix-k8s-ingress-controller.yaml
```

TCP profile support for services of type LoadBalancer

February 4, 2022

This topic contains information on how to apply TCP profiles for services of type LoadBalancer. TCP profile support for service of type LoadBalancer is similar to TCP profile support on Ingress. For information on TCP profile support on Ingress, see [TCP profile support on Ingress](#).
A TCP profile is a collection of TCP settings. Instead of configuring the settings on each entity, you can configure TCP settings in a profile and bind the profile to all the required entities.

The Citrix ingress controller provides the following service annotations for TCP profile for services of type `LoadBalancer`. You can use these annotations to define the TCP settings for the Citrix ADC.

<table>
<thead>
<tr>
<th>Service annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>service.citrix.com/frontend-tcpprofile</code></td>
<td>Use this annotation to create the front-end TCP profile (Client Plane).</td>
</tr>
<tr>
<td><code>service.citrix.com/backend-tcpprofile</code></td>
<td>Use this annotation to create the back-end TCP profile (Server Plane).</td>
</tr>
</tbody>
</table>

**User-defined TCP profiles**

Using service annotations for TCP, you can create custom profiles with name same as cs virtual server or service group and bind to the corresponding virtual server(`frontend-tcpprofile`) and service group (`backend-tcpprofile`).

<table>
<thead>
<tr>
<th>Service annotation</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>service.citrix.com/frontend-tcpprofile</code></td>
<td><code>service.citrix.com/frontend-tcpprofile: '{ &quot;ws&quot;:&quot;enabled&quot;, &quot;sack&quot;: &quot;enabled&quot; }'</code></td>
</tr>
<tr>
<td><code>service.citrix.com/backend-tcpprofile</code></td>
<td><code>service.citrix.com/backend-tcpprofile: '{ &quot;ws&quot;:&quot;enabled&quot;, &quot;sack&quot;: &quot;enabled&quot; }'</code></td>
</tr>
</tbody>
</table>

**Built-in TCP profiles**

Built-in TCP profiles do not create any profile and bind a given profile name in annotation to the corresponding virtual server(`frontend-tcpprofile`) and service group(`backend-tcpprofile`).

Following are examples for built-in TCP profiles:

```bash
1  service.citrix.com/frontend-tcpprofile: "tcp_preconf_profile"
2  service.citrix.com/backend-tcpprofile: '{
3    "citrix-svc":"tcp_preconf_profile"
}
```
Example: Service of Type load balancer with the TCP profile configuration

In this example, TCP profiles are configured for a sample application tea-beverage. This application is deployed and exposed using a service of type LoadBalancer using the tea-profile-example.yaml file.

For step by step instruction for exposing services of type LoadBalancer, see service of type LoadBalancer.

Following is a snippet of the service configuration with TCP profile.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: tea-beverage
annotations:
  service.citrix.com/secure_backend: '{
    "443-tcp": "True"
  }'
  service.citrix.com/service_type: 'SSL'
  service.citrix.com/backend-tcpprofile: '{
    "ws":"ENABLED", "sack" : "enabled" }
  ' service.citrix.com/frontend-tcpprofile: '{
    "ws":"ENABLED", "sack" : "enabled" }
  '
spec:
  type: LoadBalancer
  loadBalancerIP: 10.105.158.194
  ports:
  - name: tea-443
    port: 443
    targetPort: 443
  selector:
  name: tea-beverage
```

Note:
The TCP profile is supported for single port services.
Citrix ADC ingress controller

SSL certificate for services of type LoadBalancer through the Kubernetes secret resource

January 19, 2022

This section provides information on how to use the SSL certificate stored as a Kubernetes secret with services of type LoadBalancer. The certificate is applied if the annotation `service.citrix.com/service-type` is SSL or SSL_TCP.

Using the Citrix ingress controller default certificate

If the SSL certificate is not provided, you can use the default Citrix ingress controller certificate.

You must provide the secret name you want to use and the namespace from which it should be taken as arguments in the Citrix ingress controller YAML file.

Default Citrix ingress controller

```
--default-ssl-certificate <NAMESPACE>/<SECRET_NAME>
```

Service annotations for SSL certificate as Kubernetes secrets

Citrix ingress controller provides the following service annotations to use SSL certificates stored as Kubernetes secrets for services of type `LoadBalancer`.

<table>
<thead>
<tr>
<th>Service annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>service.citrix.com/secret</code></td>
<td>Use this annotation to specify the name of the secret resource for the front-end server certificate. It must contain a certificate and key. You can also provide a list of intermediate CA certificates in the certificate section followed by the server certificate. These intermediate CAs are automatically linked and sent to the client during the SSL handshake.</td>
</tr>
<tr>
<td><code>service.citrix.com/ca-secret</code></td>
<td>Use this annotation to provide a CA certificate for client certificate authentication. This certificate is bound to the front-end SSL virtual server in Citrix ADC.</td>
</tr>
</tbody>
</table>
## Service annotation

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>service.citrix.com/backend-secret</td>
<td>Use this annotation if the back-end communication between Citrix ADC and your workload is on an encrypted channel, and you need the client authentication in your workload. This certificate is sent to the server during the SSL handshake and it is bound to the back end SSL service group.</td>
</tr>
<tr>
<td>service.citrix.com/backend-ca-secret</td>
<td>Use this annotation to enable server authentication which authenticates the back-end server certificate. This configuration binds the CA certificate of the server to the SSL service on the Citrix ADC.</td>
</tr>
<tr>
<td>service.citrix.com/preconfigured-certkey</td>
<td>Use this annotation to specify the name of the preconfigured cert key in the Citrix ADC to be used as a front-end server certificate.</td>
</tr>
<tr>
<td>service.citrix.com/preconfigured-ca-certkey</td>
<td>Use this annotation to specify the name of the preconfigured cert key in the Citrix ADC to be used as a CA certificate for client certificate authentication. This certificate is bound to the front-end SSL virtual server in Citrix ADC.</td>
</tr>
<tr>
<td>service.citrix.com/preconfigured-backend-certkey</td>
<td>Use this annotation to specify the name of the preconfigured cert key in the Citrix ADC to be bound to the back-end SSL service group. This certificate is sent to the server during the SSL handshake for server authentication.</td>
</tr>
<tr>
<td>service.citrix.com/preconfigured-backend-ca-certkey</td>
<td>Use this annotation to specify the name of the preconfigured CA cert key in the Citrix ADC to bound to back-end SSL service group for server authentication.</td>
</tr>
</tbody>
</table>

### Examples: Front-end secret and Front-end CA secret

Following are some examples for the `service.citrix.com/secret` annotation:

The following annotation is applicable to all ports in the service.
You can use the following notation to specify the certificate applicable to specific ports by giving either `portname` or `port-protocol` as key.

```
# port-protocol : secret

service.citrix.com/secret: '{
  "443-tcp": "hotdrink-secret", "8443-tcp": "hotdrink-secret" }

# portname: secret

service.citrix.com/secret: '{
  "https": "hotdrink-secret" }
```

Following are some examples for the `service.citrix.com/ca-secret` annotation.

You need to specify the following annotation to attach the generated CA secret which is used for client certificate authentication for a service deployed in Kubernetes.

The following annotation is applicable to all ports in the service.

```
service.citrix.com/ca-secret: hotdrink-ca-secret
```

You can use the following notation to specify the certificate applicable to specific ports by giving either `portname` or `port-protocol` as key.

```
# port-protocol: secret

service.citrix.com/ca-secret: '{
  "443-tcp": "hotdrink-ca-secret", "8443-tcp": "hotdrink-ca-secret" }

# portname: secret

service.citrix.com/ca-secret: '{
  "https": "hotdrink-ca-secret" }
```
Examples: back-end secret and back-end CA secret

Following are some examples for the service.citrix.com/backend-secret annotation.

```
# port-protocol: secret
service.citrix.com/backend-secret: '{
  "443-tcp": "hotdrink-secret", "8443-tcp": "hotdrink-secret" }
'

# portname: secret

service.citrix.com/backend-secret: '{
  "tea-443": "hotdrink-secret", "tea-8443": "hotdrink-secret" }
'

# applicable to all ports

service.citrix.com/backend-secret: "hotdrink-secret"
```

Following are some examples for the service.citrix.com/backend-ca-secret annotation.

```
# portproto: secret
service.proto.citrix.com/backend-ca-secret: '{
  "443-tcp": "coffee-ca", "8443-tcp": "tea-ca" }
'

# portname: secret

service.proto.citrix.com/backend-ca-secret: '{
  "coffee-443": "coffee-ca", "tea-8443": "tea-ca" }
'

# applicable to all ports

service.proto.citrix.com/backend-ca-secret: "hotdrink-ca-secret"
```
BGP advertisement of external IP addresses for type LoadBalancer services and Ingresses using Citrix ADC CPX

February 15, 2022

Kubernetes service of type `LoadBalancer` support is provided by cloud load balancers in a cloud environment. Cloud service providers enable this support by automatically creating a load balancer and assigning an IP address which is displayed as part of the service status. Any traffic destined to the external IP address is load balanced on NodeIP and NodePort by the cloud load balancer. Once the traffic reaches the Kubernetes cluster, kube-proxy performs the routing to the actual application pods using iptables or IP virtual server rules. However, for on-prem environments the cloud load balancer auto configuration is not available.

You can expose the services of type `LoadBalancer` using the Citrix ingress controller and Tier-1 Citrix ADC devices such as Citrix ADC VPX or MPX. The Citrix ADC VPX or MPX residing outside the Kubernetes cluster load balances the incoming traffic to the Kubernetes services. For more information on such a deployment, see Expose services of type `LoadBalancer`.

However, it may not always be feasible to use an external ADC device to expose the service of type `LoadBalancer` in an on-prem environment. Sometimes, it is desirable to manage all related resources from the Kubernetes cluster itself without any external component. The Citrix ingress controller provides a way to expose the service of type `LoadBalancer` using Citrix ADC CPX that runs within the Kubernetes cluster. The existing BGP fabric to route the traffic to the Kubernetes nodes is leveraged to implement this solution.

In this deployment, Citrix ADC CPX is deployed as a daemonset on the Kubernetes nodes in host mode. Citrix ADC CPX establishes a BGP peering session with your network routers, and uses that peering session to advertise the IP addresses of external cluster services. If your routers have ECMP capability, the traffic is load-balanced to multiple CPX instances by the upstream router, which in turn load-balances to actual application pods. When you deploy the Citrix ADC CPX with this mode, Citrix ADC CPX adds iptables rules for each service of type `LoadBalancer` on Kubernetes nodes. The traffic destined to the external IP address is routed to Citrix ADC CPX pods.

The following diagram explains a deployment where Citrix ADC CPX is exposing a service of type `LoadBalancer`: 

© 1999‒2022 Citrix Systems, Inc. All rights reserved.
As shown in the diagram, Citrix ADC CPX runs as a daemon set and runs a BGP session over port 179 on the node IP address pointed by the Kubernetes node resource. For every service of type LoadBalancer added to the Kubernetes API server, the Citrix ingress controller configures the Citrix ADC CPX to advertise the external IP address to the BGP router configured. A /32 prefix is used to advertise the routes to the external router and the node IP address is used as a gateway to reach the external IP address. Once the traffic reaches to the Kubernetes node, the iptables rule steers the traffic to Citrix ADC CPX which in turn load balance to the actual service pods.

With this deployment, you can also use Kubernetes ingress resources and advertise the Ingress virtual IP (VIP) address to the router. You can specify the `NS_VIP` environment variable while deploying the Citrix ingress controller which acts as the VIP for all ingress resources. When an Ingress resource is added, Citrix ADC CPX advertises the `NS_VIP` to external routers through BGP to attract the traffic. Once traffic comes to the `NS_VIP`, Citrix ADC CPX performs the content switching and load balancing as specified in the ingress resource.

**Note:**

For this solution to work, the Citrix ingress controller must run as a root user and must have the `NET_ADMIN` capability.

**Deploy Citrix ADC CPX solution for services of type LoadBalancer**

This procedure explains how to deploy Citrix ADC CPX as a daemonset in the host network to expose services of type LoadBalancer.

This configuration includes the following tasks:

- Deploy Citrix ADC CPX with the Citrix ingress controller as sidecar
Citrix ADC ingress controller

- BGP configuration
- Service configuration

Prerequisites

- You must configure the upstream router for BGP routing with ECMP support and add Kubernetes nodes as neighbors.
- If the router supports load balancing, it is better to use a stable ECMP hashing algorithm for load-balancing with a higher entropy for even load-balancing.

Perform the following:

1. Download the `rbac.yaml` file and deploy the RBAC rules for Citrix ADC CPX and the Citrix ingress controller.

   ```bash
   kubectl apply -f rbac.yaml
   ``

2. Download the `citrix-k8s-cpx-ingress.yml` using the following command.

   ```bash
   ```

3. Edit the `citrix-k8s-cpx-ingress.yaml` file and specify the required values.
   - The argument `--configmap` specifies the ConfigMap location for the Citrix ingress controller in the form of `namespace/name`.
   - The argument `--ipam citrix-ipam-controller` can be specified if you are running the Citrix IPAM controller for automatic IP address allocation.
   - (Optional) `nodeSelector` to select the nodes where you need to run the Citrix ADC CPX daemonset. By default, it is run on all worker nodes.

4. Apply the `citrix-k8s-cpx-ingress.yaml` file to create a daemonset which starts Citrix ADC CPX and the Citrix ingress controller.

   ```bash
   kubectl apply -f citrix-k8s-cpx-ingress.yaml
   ```

5. Create a ConfigMap (configmap.yaml) with the BGP configuration which is passed as an argument to the Citrix ingress controller. For detailed information on BGP configuration, see BGP configuration.
You must have the following information to configure BGP routing:

- The router IP address for Citrix ADC CPX to connect
- The autonomous system (AS number) of the router
- The AS number for Citrix ADC CPX

Following is a sample ConfigMap with the BGP configuration.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config
  labels:
    app: cic
data:
  NS_BGP_CONFIG: |
  bgpConfig:
    bgpRouter:
      localAS: 100
      neighbor:
        address: 10.102.33.33
        remoteAS: 100
        advertisementInterval: 10
        ASOriginationInterval: 10
```

6. Apply the ConfigMap created in step 5 to apply the BGP configuration.

```
kubectl apply -f configmap.yaml
```

7. Create a YAML file with the required configuration for service of type LoadBalancer.

```
# This uses IPAM to allocate an IP from range 'Dev'
```

Note:

For detailed information, see service configuration. The service configuration section explains different ways to get an external IP address for the service and also how to use the service annotation provided by Citrix to configure different Citrix ADC functionalities.
# service.citrix.com/ipam-range: 'Dev'
service.citrix.com/frontend-ip: 172.217.163.17
service.citrix.com/service-type-0: 'HTTP'
service.citrix.com/service-type-1: 'SSL'
service.citrix.com/lbvserver: '{
  "80-tcp": {
    "lbmethod": "ROUNDROBIN"
  }
}
service.citrix.com/servicegroup: '{
  "80-tcp": {
    "usip": "yes"
  }
}
service.citrix.com/ssl-termination: edge
service.citrix.com/monitor: '{
  "80-tcp": {
    "type": "http"
  }
}

  service.citrix.com/frontend-httpprofile: '{
    "dropinvalreqs": "enabled",
    "websocket": "enabled"
  }

  service.citrix.com/backend-httpprofile: '{
    "dropinvalreqs": "enabled",
    "websocket": "enabled"
  }

  service.citrix.com/frontend-tcpprofile: '{
    "ws": "enabled",
    "sack": "enabled"
  }

  service.citrix.com/backend-tcpprofile: '{
    "ws": "enabled",
    "sack": "enabled"
  }

  service.citrix.com/frontend-sslprofile: '{
    "hsts": "enabled",
    "tls12": "enabled"
  }

  service.citrix.com/backend-sslprofile: '{
    "tls12": "enabled"
  }

  service.citrix.com/ssl-certificate-data-1: |
  -----BEGIN-----
  [...]
  -----END-----

  service.citrix.com/ssl-key-data-1: |

spec:
  type: LoadBalancer
8. Apply the service of type LoadBalancer.

```bash
kubectl apply -f service-example.yaml
```

Once the service is applied, the Citrix ingress controller creates a load balancing virtual server with BGP route health injection enabled. If the load balancing virtual server state is **UP**, the route for the external IP address is advertised to the neighbor router with a /32 prefix with the node IP address as the gateway.

**BGP configuration**

BGP configuration is performed using the ConfigMap which is passed as an argument to the Citrix ingress controller.

You must have the following information to configure BGP routing:

- The router IP address so that Citrix ADC CPX can connect to it
- The autonomous system (AS number) of the router
- The AS number for Citrix ADC CPX

In the following ConfigMap for the BGP configuration, the `bgpConfig` field represents the BGP configuration.

```yaml
apiVersion: v1
category: ConfigMap
metadata:
  name: config
  labels:
    app: cic
data:
  NS_BGP_CONFIG: |
  bgpConfig:
```
The following table explains the various fields of the `bgpConfig` field.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Type</th>
<th>Default value</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodeSelector</td>
<td>If the <code>nodeSelector</code> field is present, then the BGP router configuration is applicable for nodes which matches the <code>nodeSelector</code> field. <code>nodeSelector</code> accepts comma separated <code>key=value</code> pairs where each key represents a label name and the value is the label value. For example: <code>nodeSelector: datacenter=ds1,rack=rack1</code></td>
<td>string</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>
### Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Type</th>
<th>Default value</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>bgpRouter</td>
<td>Specifies the BGP configuration. For information on different fields of the <code>bgpRouter</code>, see the following table.</td>
<td>bgpRouter</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

The following table explains the fields for the `bgpRouter` field.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Type</th>
<th>Default value</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>localAS</td>
<td>AS number for the Citrix ADC CPX</td>
<td>integer</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>neighbor</td>
<td>Neighbor router BGP configuration.</td>
<td>neighbor</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

The following table explains the `neighbor` field.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Type</th>
<th>Default value</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>IP address for the neighbor router.</td>
<td>string</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>remoteAS</td>
<td>AS number of the neighbor router.</td>
<td>integer</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Type</th>
<th>Default value</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>advertisement</td>
<td>This field sets a minimum interval between the sending of BGP routing updates (in seconds).</td>
<td>integer</td>
<td>10 seconds</td>
<td>Yes</td>
</tr>
<tr>
<td>ASOrigination</td>
<td>This field sets the interval of sending AS origination routing updates (in seconds).</td>
<td>integer</td>
<td>10 seconds</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Different neighbors for different nodes

By default, every node in the cluster connects to all the neighbors listed in the configuration. But, if the Kubernetes cluster is spread across different data centers or different networks, different neighbor configurations for different nodes may be required. You can use the `nodeSelector` field to select the nodes required for the BGP routing configurations.
An example ConfigMap with the nodeSelector configuration is given as follows:

```yaml
apiVersion: v1
type: ConfigMap
metadata:
  name: config
  labels:
    app: cic
data:
  NS_BGP_CONFIG: |
    bgpConfig:
    - nodeSelector: datacenter=ds1
      bgpRouter:
        localAS: 100
        neighbor:
          - address: 10.102.33.44
            remoteAS: 100
            advertisementInterval: 10
            ASOriginationInterval: 10
    - nodeSelector: datacenter=ds2
      bgpRouter:
        localAS: 100
        neighbor:
          - address: 10.102.28.12
            remoteAS: 100
            advertisementInterval: 10
            ASOriginationInterval: 10

<!-- NeedCopy -->
```

In this example, the router with the IP address 10.102.33.44 is used as a neighbor by nodes with the label datacenter=ds1. The router with the IP address 10.102.28.12 is used by the nodes with the label datacenter=ds2.

**Service configuration**

**External IP address configuration**

An external IP address for the service of type LoadBalancer can be obtained by using one of the following methods.

- Specifying the `service.citrix.com/frontend-ip` annotation in the service specification as follows.
Citrix ADC ingress controller

```yaml
metadata:
  annotations:
    service.citrix.com/前端-ip: 172.217.163.17
```

- Specifying an IP address in the `spec.loadBalancerIP` field of the service specification as follows.

```yaml
spec:
  loadBalancerIP: 172.217.163.17
```

- By automatically assigning a virtual IP address to the service using the IPAM controller provided by Citrix. If one of the other two methods is specified, then that method takes precedence over the IPAM controller. The IPAM solution is designed in such a way that you can easily integrate the solution with ExternalDNS providers such as Infoblox. For more information, see Interoperability with ExternalDNS. For deploying and using the Citrix IPAM controller, see the documentation.

**Service annotation configuration**

The Citrix ingress controller provides many service annotations to leverage the various functionalities of the Citrix ADC. For example, the default service type for the load balancing virtual server is TCP, but you can override this configuration by the `service.citrix.com/service-type` annotation.

```yaml
metadata:
  annotations:
    service.citrix.com/service-type-0: 'HTTP'
    service.citrix.com/service-type-1: 'SSL'
```

With the help of various annotations provided by the Citrix Ingress Controller, you can leverage various ADC functionalities like SSL offloading, HTTP rewrite and responder policies, and other custom resource definitions (CRDs).

For more information on all annotations for service of type LoadBalancer, see service annotations.

For using secret resources for SSL certificates for Type LoadBalancer services, see SSL certificate for services of type LoadBalancer.
External traffic policy configuration

By default, the Citrix ingress controller adds all the service pods as a back-end for the load balancing virtual service in Citrix ADC CPX. This step ensures better high availability and equal distribution to the service pod instances. All nodes running Citrix ADC CPX advertises the routes to the upstream server and attracts the traffic from the router. This behavior can be changed by setting the `spec.externalTrafficPolicy` of the service to `Local`. When the external traffic policy is set to `Local`, only the pods running in the same node is added as a back-end for the load balancing virtual server as shown in the following diagram. In this mode, only those nodes which have the service pods advertise the external IP address to the router and CPX sends the traffic only to the local pods. If you do not want the traffic hopping across the nodes for performance reasons, you can use this feature.

![Diagram showing traffic flow]

Using Ingress resources

The Citrix ingress controller provides an nt variable `NS_VIP`, which is the external IP Address for all ingress resources. Whenever an ingress resource is added, Citrix ADC CPX advertises the ingress IP address to the external routers.

The Citrix ingress controller provides various annotations for ingress. For more information, see the Ingress annotation documentation.

Perform the following steps for the Ingress Configuration:

1. Download the `rbac.yaml` file and deploy the RBAC rules for Citrix ADC CPX and the Citrix ingress controller.

```bash
kubectl apply -f rbac.yaml
```
2. Download the `citrix-k8s-cpx-ingress.yml` using the following command.

   ```bash
   ```

3. Edit the `citrix-k8s-cpx-ingress.yml` file and specify the required values.
   - The argument `-configmap` specifies the ConfigMap location for the Citrix ingress controller in the form of namespace or name.
   - The environment variable `NS_VIP` to specify the external IP to be used for all Ingress resources. (This is a required parameter).

4. Apply the `citrix-k8s-cpx-ingress.yml` file to create a daemonset which starts Citrix ADC CPX and the Citrix ingress controller.

   ```bash
   kubectl apply -f citrix-k8s-cpx-ingress.yml
   ```

5. Configure BGP using ConfigMap as shown in the previous section.

6. Deploy a sample ingress resource as follows. This step advertises the IP address specified in the `NS_VIP` environment variable to the external router configured in ConfigMap.

   ```bash
   kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/docs/configure/cpx-bgp-router/ingress-example.yaml
   ```

7. Access the application using `NS_VIP:<port>`. By default, Ingress uses port 80 for insecure communication and port 443 for secure communication (If TLS section is provided).

   **Note:** Currently, the `ingress.citrix.com/frontend-ip` annotation is not supported for BGP advertisements.

**Helm Installation**

You can use Helm charts to install the Citrix ADC CPX as BGP router. For more information, see the Citrix Helm chart documentation.
Troubleshooting

- By default, Citrix ADC CPX uses the IP address range range 192.168.1.0/24 for internal communication, the IP address 192.168.1.1 as internal gateway to the host, and the IP address IP address 192.168.1.2 as NSIP. The ports 9080 and 9443 are used as management ports between the Citrix ingress controller and Citrix ADC CPX for HTTP and HTTPS. If the 192.168.1.0/24 network falls within the range of PodCIDR, you can allocate a different set of IP addresses for internal communication. The NS_IP and NS_GATEWAY environment variables control which IP address is used by Citrix ADC CPX for NSIP and gateway respectively. The same IP address must also be specified as part of the Citrix ingress controller environment variable NS_IP to establish the communication between the Citrix ingress controller and Citrix ADC CPX.

- By default, BGP on Citrix ADC CPX runs on port 179 and all the BGP traffic coming to the TCP port 179 is handled by Citrix ADC CPX. If there is a conflict, for example if you are using Calico's external BGP peering capability to advertise your cluster prefixes over BGP, you can change the BGP port with the environment variable to the Citrix ingress controller BGP_PORT.

- Use source IP (USIP) mode of Citrix ADC does not work due to the constraints in Kubernetes. If the source IP address is required by the service, you can enable the CIP (client IP header) feature on the HTTP/SSL service-type services by using the following annotations.

  service.citrix.com/servicegroup: ‘{"cip":"ENABLED" , "cipheader":"x-forwarded-for"}’

Citrix ADC CPX integration with MetalLB in layer 2 mode for on-premises Kubernetes clusters

February 3, 2022

Kubernetes service of type LoadBalancer support is provided by cloud load balancers in a cloud environment. Cloud service providers enable this support by automatically creates a load balancer and assign an IP address which is displayed as part of the service status. Any traffic destined to the external IP address is load balanced on NodeIP and NodePort by the cloud load balancer.

Citrix provides different options to support the type LoadBalancer services in an on-premises environment including:

- Using an external Citrix ADC VPX or Citrix ADC MPX as a tier-1 load balancer to load balance the incoming traffic to Kubernetes services.

For more information on such a deployment, see Expose services of type LoadBalancer.

- Expose applications running in a Kubernetes cluster using the Citrix ADC CPX daemonset running inside the Kubernetes cluster along with a router supporting ECMP over BGP. ECMP router
load balances the traffic to multiple Citrix ADC CPX instances. Citrix ADC CPX instances load balances the actual application pods. For more information on such a deployment, see BGP advertisement of external IP addresses for type LoadBalancer services and Ingresses using Citrix ADC CPX.

- Expose the Citrix ADC CPX services as an external IP service with a node external IP address. You can use this option if an external ADC as tier-1 is not feasible, and a BGP router does not exist. In this deployment, Kubernetes routes the traffic coming to the spec.externalIP of the Citrix ADC CPX service on service ports to Citrix ADC CPX pods. Ingress resources can be configured using the Citrix ingress controller to perform SSL (Secure Sockets Layer) offloading and load balancing applications. However, this deployment has the major drawback of not being reliable if there is a node failure.

- Use MetalLB which is a load-balancer implementation for bare metal Kubernetes clusters in the layer 2 mode with Citrix ADC CPX to achieve ingress capability.

This documentation shows how you can leverage MetalLB along with Citrix ADC CPX to achieve ingress capability in bare-metal clusters when the other solutions are not feasible. MetalLB in layer 2 mode configures one node to send all the traffic to the Citrix ADC CPX service. MetalLB automatically moves the IP address to a different node if there is a node failure. Thus providing better reliability than the ExternalIP service.

**Note:** MetalLB is still in the beta version. See the official documentation to know about the project maturity and any limitations.

Perform the following steps to deploy Citrix ADC CPX integration with MetalLB in layer 2 mode for on-premises Kubernetes clusters.

1. Install and configure MetalLB
2. Configure MetalLB configuration for layer 2
3. Install Citrix ADC CPX service

**Install and configure MetalLB**

First, you should install MetalLB in layer 2 mode. For more information on different types of installations for MetalLB, see the MetalLB documentation.

Perform the following steps to install MetalLB:

1. Create a namespace for deploying MetalLB.

```bash
kubectl apply -f https://raw.githubusercontent.com/metallb/metallb/v0.9.5/manifests/namespace.yaml
```
2. Deploy MetalLB using the following command.

```
kubectl apply -f https://raw.githubusercontent.com/metallb/metallb/v0.9.5/manifests/metallb.yaml
```

3. Perform the following step if you are performing the installation for the first time.

```
kubectl create secret generic -n metallb-system memberlist --from-literal=secretkey="$(openssl rand -base64 128)"
```

4. Verify the MetalLB installation and ensure that the speaker and controller is in the running state using the following command:

```
kubectl get pods -n metallb-system
```

These steps deploy MetalLB to your cluster, under the `metallb-system` namespace.

The MetalLB deployment YAML file contains the following components:

- The `metallb-system/controller` deployment: This component is the cluster-wide controller that handles IP address assignments.
- The `metallb-system/speaker` daemonset. This component communicates using protocols of your choice to make the services reachable.
- Service accounts for the controller and speaker, along with the RBAC permissions that the components need to function.

**MetalLB configuration for Layer 2**

Once MetalLB is installed, you should configure the MetalLB for layer 2 mode. MetalLB takes a range of IP addresses to be allocated to the type LoadBalancer services as external IP. In this deployment, a Citrix ADC CPX service acts as a front-end for all other applications. Hence, a single IP address is sufficient.

Create a ConfigMap for MetalLB using the following command where `metallb-config.yaml` is the YAML file with the MetalLB configuration.

```
kubectl create -f metallb-config.yaml
```
Citrix ADC ingress controller

Following is a sample MetalLB configuration for layer2 mode. In this example, 192.168.1.240-192.168.1.240 is specified as the IP address range.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    address-pools:
      - name: default
        protocol: layer2
        addresses:
          - 192.168.1.240-192.168.1.240
<!--NeedCopy-->
```

Citrix ADC CPX service installation

Once the metal LB is successfully installed, you can install the Citrix ADC CPX deployment and a service of type `LoadBalancer`.

To install Citrix ADC CPX, you can either use the YAML file or Helm charts.

To install Citrix ADC CPX using the YAML file, perform the following steps:

1. Download the Citrix ADC CPX deployment manifests.

   ```bash
   ```

2. Edit the Citrix ADC CPX deployment YAML:
   
   - Set the replica count as needed. It is better to have more than one replica for high availability.
   - Change the service type to `LoadBalancer`.

3. Apply the edited YAML file using the Kubectl command.

   ```bash
   kubectl apply -f citrix-k8s-cpx-ingress.yaml
   ```
4. View the service using the following command:

```bash
kubectl get svc cpx-service -output yaml
```

You can see that MetalLB allocates an external IP address to the Citrix ADC CPX service as follows:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: cpx-service
  namespace: default
spec:
  clusterIP: 10.107.136.241
  externalTrafficPolicy: Cluster
  healthCheckNodePort: 31916
  ports:
    - name: http
      nodePort: 31528
      port: 80
      protocol: TCP
      targetPort: 80
    - name: https
      nodePort: 31137
      port: 443
      protocol: TCP
      targetPort: 443
  selector:
    app: cpx-ingress
  sessionAffinity: None
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
    - ip: 192.168.1.240
<!–-NeedCopy-->"

**Deploy a sample application**

Perform the following steps to deploy a sample application and verify the deployment.

1. Create a sample deployment using the `sample-deployment.yaml` file.
1. `kubectl create -f sample-deployment.yaml`

2. Expose the application with a service using the `sample-service.yaml` file.

```
kubectl create -f sample-service.yaml
```

3. Once the service is created, you can add an ingress resource using the `sample-ingress.yaml`.

```
kubectl create -f sample-ingress.yaml
```

You can test the Ingress by accessing the application using a `cpx-service` external IP address as follows:

```
```

**Additional references**

For more information on configuration and troubleshooting for MetalLB see the following links:

- Metal LB troubleshooting
- Configuring routing for metal LB in layer 2 mode

**Advanced content routing for Kubernetes Ingress using the HTTPRoute CRD**

February 8, 2022

Kubernetes native Ingress offers basic host and path-based routing which is supported by the Citrix ingress controller.

Citrix also provides an alternative approach using content routing CRDs for supporting advanced routing capabilities. Content Routing CRDs include Listener CRD and HTTPRoute CRD. These CRDs provide advanced content routing features such as regex based expression and content switching based on query parameters, cookies, HTTP headers, and other Citrix ADC custom expressions.
With the Ingress version `networking.k8s.io/v1`, Kubernetes introduces support for resource backends. A resource backend is an `ObjectRef` to another Kubernetes resource within the same namespace as an Ingress object.

Now, Citrix supports configuring the HTTP route CRD resource as a resource backend in Ingress. By default, Ingress supports only limited content routing capabilities like path and host-based routing. With this feature, you can extend advanced content routing capabilities to Ingress and configure various content switching options. For a given domain, you can use the `HTTPRoute` custom resource to configure content switching without losing the third party compatibility support of the Kubernetes Ingress API.

**Note:**
- This feature supports the Kubernetes Ingress version `networking.k8s.io/v1` that is available on Kubernetes 1.19 and later versions.
- If the Ingress path routing and `HTTPRoute` are used for the same domain, all the content routing policies from the `HTTPRoute` resource get lower priority than the Ingress based content routing policies. So, it is recommended to configure all the content switching policies of the `HTTPRoute` resource for a given domain if advanced content routing is required.

**Configure advanced content routing for Kubernetes Ingress using the HTTPRoute CRD**

This procedure shows how to deploy an `HTTPRoute` resource as a resource backend to support advanced content routing.

**Prerequisites**
- Ensure that the ingress API version `networking.k8s.io/v1` is available in the Kubernetes cluster.
- Ensure that the `HTTPRoute` CRD is deployed.
Deploy the Ingress resource

Define the Ingress resource with the resource back-end pointing to a HTTPRoute custom resource in a YAML file. Specify all the front-end configurations such as certificates, front-end profiles, front-end IP address, and ingress class as part of the Ingress resource.

Following is a sample Ingress resource named as sample-ingress.yaml.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: kuard-ingress
  annotations:
    ingress.citrix.com/frontend-ip: "x.x.x.x"
    kubernetes.io/ingress.class: citrix
    ingress.citrix.com/insecure-termination: "redirect"

spec:
  tls:
    - secretName: web-ingress-secret
  rules:
    - host: kuard.example.com
      http:
        paths:
          - pathType: ImplementationSpecific
            backend:
              resource:
                apiGroup: citrix.com
                kind: HTTPRoute
                name: kuard-example-route
<!-- NeedCopy-->
```

After defining the Ingress resource in a YAML file, deploy the YAML file using the following command. Here, sample-ingress.yaml is the YAML file definition.

```bash
kubectl apply -f sample-ingress.yaml
```

In this example, content switching policies for the domain kuard.example.com are defined as part of the HTTPRoute custom resource called kuard-example-route. Certificates, frontend-ip, and ingress class are specified as part of the Ingress resource. Back-end annotations such as load balancing method and service group configurations are specified as part of the HTTPRoute custom resource.
Deploy the HTTPRoute resource

Define the HTTP route configuration in a YAML file. In the YAML file, use HTTPRoute in the `kind` field and in the `spec` section add the HTTPRoute CRD attributes based on your requirement for the HTTP route configuration.

For more information about API description and examples, see the HTTPRoute documentation.

Following is a sample `HTTPRoute` resource configuration. This example shows how to use query parameters based content switching for the various Kubernetes back-end microservices.

```yaml
apiVersion: citrix.com/v1
kind: HTTPRoute
metadata:
  name: kuard-example-route
spec:
  hostname: kuard.example.com
  rules:
  - name: kuard-blue
    match:
      queryParams:
        - name: version
          contains: v2
    action:
      backend:
        kube:
          service: kuard-blue
          port: 80
  - name: kuard-green
    match:
      queryParams:
        - name: version
          contains: v3
    action:
      backend:
        kube:
          service: kuard-green
          port: 80
  - name: kuard-default
    match:
      path:
        prefix: /
    action:
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
After you have defined the HTTP routes in the YAML file, deploy the YAML file. In this example, `httproute` is the YAML definition.

```bash
kubectl apply -f httproute.yaml
```

Profile support for the Listener CRD

February 4, 2022

You can use individual entities such as HTTP profile, TCP profile, and SSL profile to configure HTTP, TCP, and SSL respectively for the Listener CRD. Profile support for the Listener CRD helps you to customize the default protocol behavior. You can also select the SSL ciphers for the SSL virtual server.

**HTTP profile**

An HTTP profile is a collection of HTTP settings. A default HTTP profile called `nshttp_default_profile` is configured to set the HTTP configurations. These configurations are applied, by default, globally to all services and virtual servers. You can customize the HTTP configurations for a Listener resource by specifying `spec.policies.httpprofile`. If specified, Citrix ingress controller creates a new HTTP profile with the default values derived from the default HTTP profile and configures the values specified.

It helps to derive the default values from the default HTTP profile and configures the values specified.

The following example YAML shows how to enable websocket for a given front-end virtual server.

```yaml
apiVersion: citrix.com/v1
kind: Listener
metadata:
  name: test-listener
  namespace: default
spec:
  vip: x.x.x.x
```
For information about all the possible key-value pairs for the HTTP profile see, HTTP profile.

**Note:**
The ‘name’ is auto-generated.

You can also specify a built-in HTTP profile or a pre-configured HTTP profile and bind it to the front-end virtual server as shown in the following example.

```
apiVersion: citrix.com/v1
kind: Listener
metadata:
  name: test-listener
  namespace: default
spec:
  vip: x.x.x.x
  port: 80
  protocol: http
  policies:
    httpprofile:
      preconfigured: 'nshttp_default_strict_validation'

<!--NeedCopy-->
```

**TCP profile**

A TCP profile is a collection of TCP settings. A default TCP profile called nstcp_default_profile is configured to set the TCP configurations. These configurations are applied, by default, globally to all services and virtual servers. You can customize the TCP settings by specifying spec.policies.tcpprofile. When you specify spec.policies.tcpprofile, Citrix ingress controller creates a TCP profile that is derived from the default TCP profile and applies the values provided in the specification, and binds it to the front-end virtual server.

For information about all the possible key-value pairs for a TCP profile, see TCP profile.
The following example shows how to enable tcpfastopen and HyStart for the front-end virtual server.

```yaml
apiVersion: citrix.com/v1
kind: Listener
metadata:
  name: test-listener
  namespace: default
spec:
  vip: x.x.x.x
  port: 80
  protocol: http
  policies:
    tcpprofile:
      config:
        tcpfastopen: "ENABLED"
        hystart: "ENABLED"
<!--NeedCopy--> 
```

You can also specify a built-in TCP profile or a pre-configured TCP profile name as shown in the following example:

```yaml
apiVersion: citrix.com/v1
kind: Listener
metadata:
  name: test-listener
  namespace: default
spec:
  vip: x.x.x.x
  port: 80
  protocol: http
  policies:
    tcpprofile:
      preconfigured: 'nstcp_default_Mobile_profile'
<!--NeedCopy--> 
```
SSL profile

An SSL profile is a collection of settings for SSL entities. SSL profile makes configuration easier and flexible. You can configure the settings in a profile and bind that profile to a virtual server instead of configuring the settings on each entity. An SSL profile allows you to customize many SSL parameters such as TLS protocol and ciphers. For more information about SSL profile, see SSL profile infrastructure.

Note:

By default, Citrix ADC creates a legacy SSL profile. The legacy SSL profile has many drawbacks including non-support for advanced protocols such as SSLv3. Hence, it is recommended to enable the default SSL profiles in Citrix ADC before Citrix ingress controller is launched.

To enable the advanced SSL profile, use the following command in the Citrix ADC command line:

```
set ssl parameter -defaultProfile ENABLED
```

The command enables the default SSL profile for all the existing SSL virtual servers and the SSL service groups.

You can specify `spec.policies.sslprofile` to customize the SSL profile. When specified, Citrix ingress controller creates an SSL profile derived from the default SSL front-end profile: `ns_default_ssl_profile_frontend`.

For information about key-value pairs supported in the SSL profile, see SSL profile.

Note:

The `name` is auto-generated.

The following example shows how to enable TLS1.3 and HSTS for the front-end virtual server.

```
apiVersion: citrix.com/v1
kind: Listener
metadata:
  name: test-listener
  namespace: default
spec:
  vip: x.x.x.x
  port: 443
  certificates:
    - secret:
        name: my-cert
  protocol: https
  policies:
  sslprofile:
```
**Citrix ADC ingress controller**

```yaml
config:
  tls13: "ENABLED"
  hsts: "ENABLED"

<!--NeedCopy-->
```

You can specify a built-in or pre-configured SSL profile name as shown in the following example:

```yaml
apiVersion: citrix.com/v1
kind: Listener
metadata:
  name: test-listener
  namespace: default
spec:
  vip: x.x.x.x
  port: 443
  certificates:
    - secret:
        name: my-cert
  protocol: https
  policies:
    sslprofile:
      preconfigured: 'ns_default_ssl_profile_secure_frontend'
<!--NeedCopy-->
```

**SSL ciphers**

The Ingress Citrix ADC has built-in cipher groups. By default, virtual servers use a DEFAULT cipher group for an SSL transaction. To use ciphers which are not part of the DEFAULT cipher group, you must explicitly bind them to an SSL profile. You can use `spec.policies.sslciphers` to provide a list of ciphers, list of built-in cipher groups, or the list of user-defined cipher groups.

**Note:**

The order of priority of ciphers is the same order defined in the list. The first one in the list gets the first priority and likewise.

The following example shows how to provide a list of built-in cipher suites.
For information about the list of cipher suites available in Citrix ADC, see SSL profile infrastructure.

Ensure that Citrix ADC has a user-defined cipher group for using a user-defined cipher group. Perform the following steps to configure a user-defined cipher group:

1. Create a user-defined cipher group. For example, **MY-CUSTOM-GROUP**.
2. Bind all the required ciphers to the user-defined cipher group.
3. Note down the user-defined cipher group name.

For detailed instructions, see Configure a user-defined cipher group.

**Note:** The order of priority of ciphers is the same order defined in the list. The first one in the list gets the first priority and likewise.

The following example shows how to provide a list of built-in cipher groups and/or user defined cipher group. The user-defined cipher groups must be present in Citrix ADC before you apply it to Listener.

```yaml
apiVersion: citrix.com/v1
kind: Listener
metadata:
  name: test-listener
  namespace: default
spec:
  vip: x.x.x.x
  port: 443
  certificates:
    secret:
      name: my-cert
    protocol: https
    policies:
      sslciphers:
        - 'TLS1.2-ECDHE-RSA-AES128-GCM-SHA256'
        - 'TLS1.2-ECDHE-RSA-AES256-GCM-SHA384'
        - 'TLS1.2-ECDHE-RSA-AES-128-SHA256'
        - 'TLS1.2-ECDHE-RSA-AES-256-SHA384'
</!--NeedCopy-->
In the preceding example, **SECURE** and **HIGH** are built-in cipher groups in Citrix ADC. **MY-CUSTOM-CIPHERS** is the pre-configured user-defined cipher groups.

**Note:** If you have specified the pre-configured SSL profile, you must bind the ciphers manually through Citrix ADC and `spec.policies.sslciphers` is not applied on the pre-configured SSL profile.

**Note:** The built-in cipher groups can be used in Tier-1 and Tier-2 Citrix ADC. The user-defined cipher group can be used only in a Tier-1 Citrix ADC.

### Analytics profile

Analytics profile enables Citrix ADC to export the type of transactions or data to an external platform. If you are using Citrix ADC Observability Exporter to collect metrics and transactions data and export it to endpoints such as Elasticsearch or Prometheus, you can configure the analytics profile to select the type of data that needs to be exported.

**Note:**

For the Analytics profile to be functional, you must configure the Citrix ADC Observability Exporter. Analytics configuration support using ConfigMap.

The following example shows how to enable **webinsight** and **tcpinsight** in the analytics profile.
The following example shows how to select the additional parameters for the type of `webinsight` which you want to be exported to Citrix ADC Observability Exporter. For information about the valid key-value pair, see Analytics Profile.

```yaml
apiVersion: citrix.com/v1
kind: Listener
metadata:
  name: test-listener
namespace: default
spec:
  vip: x.x.x.x
  port: 443
  certificates:
    - secret:
      name: my-cert
      protocol: https
  policies:
    analyticsprofile:
      config:
        - type: webinsight
          parameters:
            httpdomainname: "ENABLED"
            httplocation: "ENABLED"
<!--NeedCopy-->
```

The following example shows how to use pre-configured analytics profiles.

```yaml
apiVersion: citrix.com/v1
kind: Listener
metadata:
  name: test-listener
namespace: default
spec:
  vip: x.x.x.x
  port: 443
<!--NeedCopy-->
```
IP address management using the Citrix IPAM controller for Ingress resources

July 5, 2022

IPAM controller is an application provided by Citrix for IP address management and it runs in parallel to the Citrix ingress controller in the Kubernetes cluster. Automatically allocating IP addresses to services of type LoadBalancer from a specified IP address range using the IPAM controller is already supported. Now, you can also assign IP addresses to Ingress resources from a specified range using the IPAM controller.

You can specify IP address ranges in the YAML file while deploying the IPAM controller using YAML. The Citrix ingress controller configures the IP address allocated to the Ingress resource as a virtual IP address (VIP) in Citrix ADC MPX or VPX.

The IPAM controller requires the VIP CustomResourceDefinition (CRD) provided by Citrix. The VIP CRD is used for internal communication between the Citrix ingress controller and the IPAM controller.

Assign IP address for Ingress resource using the IPAM controller

This topic provides information on how to use the IPAM controller to assign IP addresses for Ingress resources.

To configure an Ingress resource with an IP address from the IPAM controller, perform the following steps:

1. Deploy the VIP CRD
2. Deploy the Citrix ingress controller
3. Deploy the IPAM controller
4. Deploy the application and Ingress resource
**Step 1: Deploy the VIP CRD**

Perform the following step to deploy the Citrix VIP CRD which enables communication between the Citrix ingress controller and the IPAM controller.

```bash
kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/crd/vip/vip.yaml
```

For more information on VIP CRD, see the VIP CustomResourceDefinition.

**Step 2: Deploy the Citrix ingress controller**

Perform the following steps to deploy the Citrix ingress controller with the IPAM controller argument.

1. Download the `citrix-k8s-ingress-controller.yaml` file using the following command:

```bash
wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/baremetal/citrix-k8s-ingress-controller.yaml
```

2. Edit the Citrix ingress controller YAML file:
   
   - Specify the values of the environment variables as per your requirements. For more information on specifying the environment variables, see the Deploy Citrix ingress controller. Here, you don’t need to specify `NS_VIP`.
   
   - Specify the IPAM controller as an argument using the following:

     ```yaml
     args:
     - --ipam
citrix-ipam-controller
     ```

     Here is a snippet of a sample Citrix ingress controller YAML file with the IPAM controller argument:

     ```yaml
     apiVersion: v1
     kind: Pod
     ```

   **Note:**
   This YAML is for demonstration purpose only and not the full version. Always, use the latest version of the YAML and edit as per your requirements. For the latest version see the `citrix-k8s-ingress-controller.yaml` file.
3. Deploy the Citrix ingress controller using the edited YAML file with the following command:

```
kubectl create -f citrix-k8s-ingress-controller.yaml
```

For more information on how to deploy the Citrix ingress controller, see the Deploy Citrix ingress controller.
Step 3: Deploy the IPAM controller

Perform the following steps to deploy the IPAM controller.

1. Create a file named `citrix-ipam-controller.yaml` with the following configuration:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: citrix-ipam-controller
  namespace: kube-system
spec:
  replicas: 1
  selector:
    matchLabels:
      app: citrix-ipam-controller
  template:
    metadata:
      labels:
        app: citrix-ipam-controller
    spec:
      serviceAccountName: citrix-ipam-controller
      containers:
      - name: citrix-ipam-controller
        image: quay.io/citrix/citrix-ipam-controller:1.0.3
        env:
          # This IPAM controller takes environment variable VIP_RANGE. IPs in this range are used to assign values for IP range
          - name: "VIP_RANGE"
            value: '[["10.217.6.115-10.217.6.117"], {
              "one-ip": ["5.5.5.5"]
            }, {
              "two-ip": ["6.6.6.6", "7.7.7.7"]
            }]
            # The IPAM controller can also be configured with name spaces for which it would work through the environment variable
            # VIP_NAMESPACES, This expects a set of namespaces passed as space separated string
      imagePullPolicy: Always
```

The manifest contains two environment variables, `VIP_RANGE` and `VIP_NAMESPACES`. You can specify the appropriate routable IP range with a valid CIDR under the `VIP_RANGE`. If necessary,
you can also specify a set of namespaces under VIP_NAMESPACES so that the IPAM controller allocates addresses only for services or Ingress resources from specific namespaces.

2. Deploy the IPAM controller using the following command:

```
kubectl create -f citrix-ipam-controller.yaml
```

**Step 4: Deploy Ingress resources**

Perform the following steps to deploy a sample application and Ingress resource.

1. Deploy the Guestbook application using the following command:

```
kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/example/guestbook/guestbook-all-in-one.yaml
```

2. Create the guestbook-ingress YAML file with Ingress resource definition to send traffic to the front-end of the guestbook application.

The following is a sample YAML:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: guestbook-ingress
  annotations:
  ingress.citrix.com/ipam-range: "two-ip"
  #ingress.citrix.com/frontend-ip: "5.5.5.5"
  kubernetes.io/ingress.class: "cic-vpx"
spec:
  rules:
  - host: www.guestbook.com
    http:
      paths:
      - path: /
        backend:
          serviceName: frontend
          servicePort: 80
```

3. Deploy the Ingress resource.
Multiple IP address allocations

For Ingress resources, an IP address can be allocated multiple times since multiple ingress resources may be handled by a single csvserver. If the specified IP range has only a single IP address, it is allocated multiple times. But, if the named IP range consists of multiple IP addresses, only one of them is constantly allocated.

To facilitate multiple allocations, the IPAM controller keeps track of allocated IP addresses. The IPAM controller places an IP address into the free pool only when all allocations of that IP address by Ingress resources are released.

Allocations by different resources

Both services of type LoadBalancer and Ingress resources can use the IPAM controller for IP allocations at the same time. If an IP address is allocated by one type of resource, it is not available for a resource of another type. But, the same IP address may be used by multiple ingress resources.

Apply CRDs through annotations

February 7, 2022

You can now apply CRDs such as Rewrite and Responder, Ratelimit, Auth, WAF, and Bot for ingress resources and services of type load balancer by referring them using annotations. Using this feature, when there are multiple services in an Ingress resource, you can apply the rewrite and responder policy for a specific service or all the services based on your requirements.

The following are the two benefits of this feature:

- You can apply a CRD at a per-ingress, per-service level. For example, the same service referred through an internal VIP may have different set of rewrite-responder policies compared to the one exposed outside.
- Operations team can create CRD instances without specifying the service names. The application developers can choose the right policies based on their requirements.

Note:

CRD instances should be created without service names.
Ingress annotation for referring CRDs

An Ingress resource can refer a Rewrite and Responder CRD directly using the `ingress.citrix.com/rewrite-responder` annotation.

The following are different ways of referring the rewrite-responder CRD using annotations.

- You can apply the Rewrite and Responder CRD for all the services referred in the given ingress using the following format:

```
1 ingress.citrix.com/rewrite-responder_crd: <Rewritepolicy Custom-resource-instance-name>
```

Example:

```
1 ingress.citrix.com/rewrite-responder_crd: "blockurlpolicy"
```

In this example, the Rewrite and Responder policy is applied for all the services referred in the given ingress.

- You can apply the Rewrite and Responder CRD to a specified Kubernetes service in an Ingress resource using the following format:

```
1 ingress.citrix.com/rewrite-responder_crd: '{
2   <Kubernetes-service-name>: <Rewritepolicy Custom-resource-instance-name -name> }
3 '}
```

Example:

```
1 ingress.citrix.com/rewrite-responder_crd: '{
2   "frontendsvc": "blockurlpolicy", "backendsvc": "addresponseheaders" }
3 '}
```

In this example, the rewrite policy `blockurlpolicy` is applied on the traffic coming to the `frontendsvc` service and the `addresponseheaders` policy is applied to the `backendsvc` service coming through the current ingress.

You can also apply the Auth, Bot, WAF, and Ratelimit CRDs using ingress annotations:
The following table explains the annotations and examples for Auth, Bot, WAF, and Ratelimit CRDs.
Service of type LoadBalancer annotation for referring Rewrite and Responder CRD

A service of type LoadBalancer can refer a Rewrite and Responder CRD using annotations. The following is the format for the annotation:

```plaintext
1 service.citrix.com/rewrite-responder: <Rewritepolicy Custom-resource-instance-name>
```

Listener CRD support for Ingress through annotation

March 17, 2022

Ingress is a standard Kubernetes resource that specifies HTTP routing capability to back-end Kubernetes services. Citrix ingress controller provides various annotations to fine-tune the Ingress parameters for both front-end and back-end configurations. For example, using the `ingress.citrix.com/frontend-ip` annotation you can specify the front-end listener IP address configured in Citrix ADC by Citrix ingress controller. Similarly, there are other front-end annotations to fine-tune HTTP and SSL parameters. When there are multiple Ingress resources and if they share front-end IP and port, specifying these annotations in each Ingress resource is difficult.
Citrix ADC ingress controller

Sometimes, there is a separation of responsibility between network operations professionals (NetOps) and developers. NetOps are responsible for coming up with front-end configurations like front-end IP, certificates, and SSL parameters. Developers are responsible for HTTP routing and back-end configurations. Citrix ingress controller already provides content routing CRDs such as listener CRD for front-end configurations and HTTPRoute for back-end routing logic.

Now, Listener CRD can be applied for Ingress resources using an annotation provided by Citrix.

Through this feature, you can use the Listener CRD for your Ingress resource and separate the creation of the front-end configuration from the Ingress definition. Hence, NetOps can separately define the Listener resource to configure front-end IP, certificates, and other front-end parameters (TCP, HTTP, and SSL). Any configuration changes can be applied to the listener resources without changing each Ingress resource. In Citrix ADC, a listener resource corresponds to content switching virtual servers, SSL virtual servers, certkeys and front-end HTTP, SSL, and TCP profiles.

**Note:**

While using this feature, you must ensure that all ingresses with the same front-end IP and port refer to the same Listener resource. For Ingresses that use the same front-end IP and port combinations, one Ingress referring to a listener resource and another Ingress referring to the ingress .citrix.com/frontend-ip annotation is not supported.

**Restrictions**

When Listener is used for the front-end configurations, the following annotations are ignored and there may not be any effect:

- ingress.citrix.com/frontend-ip
- Ingress.citrix.com/frontend-ipset-name
- ingress.citrix.com/secure-port
- ingress.citrix.com/insecure-port
- ingress.citrix.com/insecure-termination
- ingress.citrix.com/secure-service-type
- ingress.citrix.com/insecure-service-type
- ingress.citrix.com/csvserver
- ingress.citrix.com/frontend-tcpprofile
- ingress.citrix.com/frontend-sslprofile
- ingress.citrix.com/frontend-httpprofile

**Deploying a Listener CRD resource for Ingress**

Using the ingress.citrix.com/listener annotation, you can specify the name and namespace of the Listener resource for the ingress in the form of namespace/name. The namespace is not re-
Citrix ADC ingress controller

Required if the Listener resource is in the same namespace as that of Ingress.

Following is an example for the annotation:

```plaintext
-ingress.citrix.com/listener: default/listener1
```

Here, **default** is the namespace of the Listener resource and **listener1** is the name of the Listener resource which specifies the front-end parameters.

Perform the following steps to deploy a Listener resource for the Ingress:

1. Create a Listener resource (**listener.yaml**) as follows:

```yaml
apiVersion: citrix.com/v1
kind: Listener
metadata:
  name: my-listener
  namespace: default
spec:
  ingressClass: citrix
  vip: '192.168.0.1' # Virtual IP address to be used, not required when CPX is used as ingress device
  port: 443
  protocol: https
  redirectPort: 80
  secondaryVips:
    - "10.0.0.1"
    - "1.1.1.1"
  policies:
    httpprofile:
      config:
        websocket: "ENABLED"
    tcpprofile:
      config:
        sack: "ENABLED"
    sslprofile:
      config:
        ssl3: "ENABLED"
        sslciphers:
          - "SECURE"
          - "MEDIUM"
    analyticsprofile:
      config:
```

© 1999–2022 Citrix Systems, Inc. All rights reserved. 342
Here, the Listener resource my-listener in the default namespace specifies the front-end configuration such as VIP, secondary VIPs, HTTP profile, TCP profile, SSL profile, and SSL ciphers. It creates a content switching virtual server in Citrix ADC on port 443 for HTTPS traffic, and all HTTP traffic on port 80 is redirected to HTTPS.

**Note:**
The `vip` field in the Listener resource is not required when Citrix ADC CPX is used as an ingress device. For Citrix ADC VPX, VIP is the same as the pod IP address which is automatically configured by Citrix ingress controller.

2. Apply the Listener resource.

```bash
kubectl apply -f listener.yaml
```

3. Create an Ingress resource (ingress.yaml) by referring to the Listener resource.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  namespace: default
  annotations:
    ingress.citrix.com/listener: my-listener
    kubernetes.io/ingress.class: "citrix"
spec:
tls:
  - secretName: my-secret
    hosts:
    - example.com
    rules:
    - host: example.com
      http:
        paths:
        - path: /
          pathType: Prefix
        backend:
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
Citrix ADC ingress controller

```
21  service:
22   name: kuard
23  port:
24   number: 80
```

Here, the ingress resource `my-ingress` refers to the Listener resource `my-listener` in the default namespace for front-end configurations.

4. Apply the ingress resource.

```
kubectl apply -f ingress.yaml
```

**Certificate management**

There are two ways in which you can specify the certificates for Ingress resources. You can specify the certificates as part of the Ingress resource or provide the certificates as part of the Listener resource.

**Certificate management through Ingress resource**

In this approach, all certificates are specified as part of the regular ingress resource as follows. Listener resource does not specify certificates. In this mode, you need to specify certificates as part of the Ingress resource.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  namespace: default
  annotations:
    ingress.citrix.com/listener: my-listener
    kubernetes.io/ingress.class: "citrix"
spec:
  tls:
    - secretName: my-secret
      hosts:
        - example.com
      rules:
        - host: example.com
          http:
            paths:
```
Certificate management through Listener resource

In this approach, certificates are provided as part of the Listener resource. You do not have to specify certificates as part of the Ingress resource.

The following Listener resource example shows certificates.

```yaml
apiVersion: citrix.com/v1
kind: Listener
metadata:
  name: my-listener
  namespace: default
spec:
  ingressClass: citrix
certificates:
  - secret:
      name: my-secret
      # Secret named 'my-secret' in current namespace bound as default certificate
      default: true
  - secret:
      # Secret 'other-secret' in demo namespace bound as SNI certificate
      name: other-secret
      namespace: demo
      vip: '192.168.0.1'
      # Virtual IP address to be used, not required when CPX is used as ingress device
      port: 443
      protocol: https
      redirectPort: 80
```
Configuring consistent hashing algorithm using Citrix ingress controller

May 25, 2022

Load balancing algorithms define the criteria that the Citrix ADC appliance uses to select the service to which to redirect each client request. Different load balancing algorithms use different criteria and consistent hashing is one the load balancing algorithms supported by Citrix ADC. Consistent hashing algorithms are often used to load balance when the back-end is a caching server to achieve stateless persistency. Consistent hashing can ensure that when a cache server is removed, only the requests cached in that specific server is rehashed and the rest of the requests are not affected. For more information on the consistent hashing algorithm, see the Citrix ADC documentation.

You can now configure the consistent hashing algorithm on Citrix ADC using Citrix ingress controller. This configuration is enabled with in the Citrix ingress controller using a ConfigMap.
**Configure hashing algorithm**

A new parameter `NS_LB_HASH_ALGO` is introduced in the Citrix ingress controller ConfigMap for hashing algorithm support.

Supported environment variables for consistent hashing algorithm using ConfigMap under the `NS_LB_HASH_ALGO` parameter:

- `hashFingers`: Specifies the number of fingers to be used for the hashing algorithm. Possible values are from 1 to 1024. Increasing the number of fingers provides better distribution of traffic at the expense of extra memory.
- `hashAlgorithm`: Specifies the supported algorithm. Supported algorithms are `default`, `jarh`, `prac`.

The following example shows a sample ConfigMap for configuring consistent hashing algorithm using Citrix ingress controller. In this example, the hashing algorithm is used as Prime Re-Shuffled Assisted CARP (PRAC) and the number of fingers to be used in PRAC is set as 50.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cic-configmap
labels:
  app: citrix-ingress-controller
data:
  NS_LB_HASH_ALGO: |
  hashFingers: 50
  hashAlgorithm: 'prac'
```

---

**Add DNS records using Citrix ADC ingress controller**

July 6, 2022

A DNS address record is a mapping of the domain name to the IP address.

When you want to use Citrix ADC as a DNS resolver, you can add the DNS records on Citrix ADC using Citrix ingress controller.

For more information on creating DNS records on Citrix ADC, see the Citrix ADC documentation.
Adding DNS records for Ingress resources

You need to enable the following environment variable during the Citrix ingress controller deployment to add DNS records for an Ingress resource.

**NS_CONFIG_DNS_REC**: This variable is configured at the boot time and cannot be changed at runtime. Possible values are **true** or **false**. The default value is false and you need to set it as true to enable the DNS server configuration. When you set the value as **true**, an address record is created on Citrix ADC.

Adding DNS records for services of type LoadBalancer

You need to perform the following tasks to add DNS records for services of type LoadBalancer:

- Enable the **NS_SVC_LB_DNS_REC** environment variable by setting the value as **True** for adding DNS records for a service of type LoadBalancer.
- Specify the DNS host name using the **service.citrix.com/dns-hostname** annotation.

When you create a service of type LoadBalancer with the **service.citrix.com/dns-hostname** annotation, Citrix ingress controller adds the DNS record on Citrix ADC. The DNS record is configured using the domain name specified in the annotation and the external IP address assigned to the service.

When you delete a service of type LoadBalancer with the **service.citrix.com/dns-hostname** annotation, Citrix ADC ingress controller removes the DNS records from the Citrix ADC. Citrix ingress controller also removes the stale entries of DNS records during boot up if the service is not available.

The following example shows a sample service of type LoadBalancer with the annotation configuration to add DNS records to Citrix ADC:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: guestbook
  annotations:
    service.citrix.com/dns-hostname: "guestbook.com"
spec:
  loadBalancerIP: "192.2.212.16"
type: LoadBalancer
  ports:
    - port: 9006
targetPort: 80
  protocol: TCP
```
Configure static route on Ingress Citrix ADC VPX or MPX

July 5, 2022

In a Kubernetes cluster, pods run on an overlay network. The overlay network can be Flannel, Calico, Weave, and so on. The pods in the cluster are assigned with an IP address from the overlay network which is different from the host network.

The Ingress Citrix ADC VPX or MPX outside the Kubernetes cluster receives all the Ingress traffic to the microservices deployed in the Kubernetes cluster. You need to establish network connectivity between the Ingress Citrix ADC instance and the pods for the ingress traffic to reach the microservices.

One of the ways to achieve network connectivity between pods and Citrix ADC VPX or MPX instance outside the Kubernetes cluster is to configure routes on the Citrix ADC instance to the overlay network.

You can either do this manually or Citrix ingress controller provides an option to automatically configure the network.

**Note:**
Ensure that the Citrix ADC instance (MPX or VPX) has SNIP configured on the host network. The host network is the network on which the Kubernetes nodes communicate with each other.

Manually configure route on the Citrix ADC instance

Perform the following:

1. On the master node in the Kubernetes cluster, get the podCIDR using the following command:
If you are using **Calico** CNI then use the following command to get the podCIDR:

```sh
# kubectl get nodes -o jsonpath="{range .items[*]} { 'podNetwork: ' } { .metadata.annotations.projectcalico\./org/IPv4IPIPTunnelAddr } { '\tgateway: ' } { .metadata.annotations.projectcalico\./org/IPv4Address } { '\n' } "
```

2. Log on to the Citrix ADC instance.
3. Add route on the Citrix ADC instance using the podCIDR information. Use the following command:

```
add route <pod_network> <podCIDR_netmask> <gateway>
```

For example,
Automatically configure route on the Citrix ADC instance

In the `citrix-k8s-ingress-controller.yaml` file, you can use an argument, `feature-node-watch` to automatically configure route on the associated Citrix ADC instance.

Set the `feature-node-watch` argument to `true` to enable automatic route configuration.

You can specify this argument in the `citrix-k8s-ingress-controller.yaml` file as follows:

```yaml
spec:
  serviceAccountName: cic-k8s-role
  containers:
    - name: cic-k8s-ingress-controller
      image: "quay.io/citrix/citrix-k8s-ingress-controller:1.26.7"
      # feature-node-watch argument configures route(s) on the Ingress Citrix ADC
      # to provide connectivity to the pod network. By default, this feature is disabled.
      args:
        - --feature-node-watch
        true

By default, the `feature-node-watch` argument is set to `false`. Set the argument to `true` to enable the automatic route configuration.

For automatic route configuration, you must provide permissions to listen to the events of nodes resource type. You can provide the required permissions in the `citrix-k8s-ingress-controller.yaml` file as follows:

```yaml
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: cic-k8s-role
rules:
  - apiGroups: [""]
    resources: ["services", "endpoints", "ingresses", "pods", "secrets", "nodes"]
    verbs: ["*"]
<!-- NeedCopy -->
```
Establish network between Kubernetes nodes and Ingress Citrix ADC using Citrix node controller

February 15, 2022

In Kubernetes environments, when you expose the services for external access through the Ingress device you need to appropriately configure the network between the Kubernetes nodes and the Ingress device.

Configuring the network is challenging as the pods use private IP addresses based on the CNI framework. Without proper network configuration, the Ingress device cannot access these private IP addresses. Also, manually configuring the network to ensure such reachability is cumbersome in Kubernetes environments.

Also, if the Kubernetes cluster and the Ingress Citrix ADC are in different subnets, you cannot establish a route between them using Static routing. This scenario requires an overlay mechanism to establish a route between the Kubernetes cluster and the Ingress Citrix ADC.

Citrix provides a node controller that you can use to create a VXLAN based overlay network between the Kubernetes nodes and the Ingress Citrix ADC as shown in the following diagram:

To establish network connectivity using Citrix node controller:

1. Deploy the Citrix ingress controller. Perform the following steps:
a) Download the citrix-k8s-ingress-controller.yaml using the following command:

```
wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/baremetal/citrix-k8s-ingress-controller.yaml
```

b) Edit the citrix-k8s-ingress-controller.yaml file and enter the values for the environmental variables. For more information, see Deploy the Citrix ingress controller.

c) Once you update the environment variables, save the YAML file and deploy it using the following command:

```
kubectl create -f citrix-k8s-ingress-controller.yaml
```

d) Verify if the Citrix ingress controller is deployed successfully using the following command:

```
kubectl get pods --all-namespaces
```

2. Deploy the Citrix node controller. For information on how to deploy the Citrix node controller, see Deploy the Citrix k8s node controller.

### Expose Service of type NodePort using Ingress

January 26, 2022

In a single-tier deployment, the Ingress Citrix ADC (VPX or MPX) outside the Kubernetes cluster receives all the Ingress traffic to the microservices deployed in the Kubernetes cluster. For the Ingress traffic to reach the microservices, you need to establish network connectivity between the Ingress Citrix ADC instance and pods.

As pods run on overlay network, the pod IP addresses are private IP addresses and the Ingress Citrix ADC instance cannot reach the microservices running within the pods. To make the service accessible from outside of the cluster, you can create the service of type NodePort. The Citrix ADC instance load balances the Ingress traffic to the nodes that contain the pods.

To create the service of type NodePort, in your service definition file, specify `spec.type: NodePort` and optionally specify a port in the range 30000–32767.
Sample deployment

Consider a scenario wherein you are using a NodePort based service, for example, an apache app and want to expose the app to North-South traffic using an Ingress. In this case, you need to create the apache app deployment, define the service of type NodePort, and create an Ingress definition to configure Ingress Citrix ADC to send the North-South traffic to the nodeport of the apache app.

In this example, you create a deployment named apache, and deploy it in your Kubernetes cluster.

1. Create a manifest for the deployment named apache-deployment.yaml.

```
# If using this on GKE
# Make sure you have cluster-admin role for your account
# kubectl create clusterrolebinding citrix-cluster-admin --clusterrole=cluster-admin --user=<username of your google account>
#

# For illustration a basic apache web server is used as a application
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache
  labels: 
    name: apache
spec:
  selector:
    matchLabels:
      app: apache
  replicas: 4
  template:
    metadata:
      labels: 
        app: apache
    spec:
      containers: 
        - name: apache
          image: httpd:latest
          ports: 
            - name: http
              containerPort: 80
          imagePullPolicy: IfNotPresent
<!--NeedCopy-->
Citrix ADC ingress controller

Containers in this deployment listen on port 80.

2. Create the deployment using the following command:

```
kubectl create -f apache-deployment.yaml
```

3. Verify that four pods are running using the following:

```
kubectl get pods
```

4. Once you verify that pods are up and running, create a service of type `NodePort`. The following is a manifest for the service:

```yaml
# Expose the apache web server as a Service
apiVersion: v1
kind: Service
metadata:
  name: apache
labels:
  name: apache
spec:
  type: NodePort
  ports:
  - name: http
    port: 80
    targetPort: http
  selector:
    app: apache
<!--NeedCopy-->
```

5. Copy the manifest to a file named `apache-service.yaml` and create the service using the following command:

```
kubectl create -f apache-service.yaml
```

The sample deploys and exposes the Apache web server as a service. You can access the service using the `<NodeIP>:<NodePort>` address.
6. After you have deployed the service, create an Ingress resource to configure the Ingress Citrix ADC to send the North-South traffic to the nodeport of the apache app. The following is a manifest for the Ingress definition named as `vpx-ingress.yaml`.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    ingress.citrix.com/frontend-ip: xx.xxx.xxx.xx
  name: vpx-ingress
spec:
  defaultBackend:
    service:
      name: apache
      port:
        number: 80
```

7. Deploy the Ingress object.

```bash
kubectl create -f vpx-ingress.yaml
```

**Configure pod to pod communication using Calico**

February 3, 2022

Configuring a network in Kubernetes is a challenge. It requires you to deal with many nodes and pods in a cluster system. There are four problems you need to address while configuring the network:

- Container to container (which collectively provides a service) communication
- Pod to pod communication
- Pod to service communication
- External to service communication

**Pod to pod communication**

By default, docker creates a virtual bridge called `docker0` on the host machine and it assigns a private network range to it. For each container that is created, a virtual Ethernet device is attached to this bridge. The virtual Ethernet device is then mapped to `eth0` inside the container, with an IP from the
network range. This process happens for each host that is running docker. There is no coordination between these hosts therefore the network ranges might collide.

Because of this, containers can only communicate with containers that are connected to the same virtual bridge. To communicate with other containers on other hosts, they must rely on port mapping. That means, you need to assign a port on the host machine to each container and then forward all the traffic on that port to that container.

Since the local IP address of the application is translated to the host IP address and port on the host machine, Kubernetes assumes that all nodes can communicate with each other without NAT. It also assumes that the IP address that a container sees for itself is the same IP address that the other containers see for the container. This approach also enables you to port applications easily from virtual machines to containers.

Calico is one of the many different networking options that offer these capabilities for Kubernetes.

**Calico**

Calico is designed to simplify, scale, and secure cloud networks. The open source framework enables Kubernetes networking and network policy for clusters across the cloud. Within the Kubernetes ecosystem, Calico is starting to emerge as one of the most popularly used network frameworks or plug-ins, with many enterprises using it at scale.

Calico uses a pure IP networking fabric to deliver high performance Kubernetes networking, and its policy engine enforces developer intent for high-level network policy management. Calico provides Layer 3 networking capabilities and associates a virtual router with each node. It enables host to host and pod to pod networking. Calico allows establishment of zone boundaries through BGP or encapsulation through IP on IP or VXLAN methods.

**Integration between Kubernetes and Calico**

Calico integrates with Kubernetes through a CNI plug-in built on a fully distributed, layer 3 architecture. Hence, it scales smoothly from a single laptop to large enterprise. It relies on an IP layer and it is relatively easy to debug with existing tools.

**Configure the network with Calico**

First, bring up a Kubernetes cluster with Calico using the following commands:

```
1 > kubeadm init --pod-network-cidr=192.168.0.0/16
2 > export KUBECONFIG=/etc/kubernetes/admin.conf
3 > kubectl apply -f calico.yaml
```
A master node is created with Calico as the CNI. After the master node is up and running, you can join the other nodes to the master using the `join` command.

Calico processes that are part of the Kubernetes master node are:

- Calico etcd
  
  `kube-system calico-etcd-j4rwc 1/1 Running`

- Calico controller
  
  `kube-system calico-kube-controllers-679568f47c-vz69g 1/1 Running`

- Calico nodes
  
  `kube-system calico-node-ct6c9 2/2 Running`

**Note:**
When you join a node to the Kubernetes cluster, a new Calico node is initiated on the Kubernetes node.

**Configure BGP peer with Ingress Citrix ADC**

Whenever you deploy an application after establishing the Calico network in the cluster, Kubernetes assigns an IP address from the IP address pool of Calico to the service associated with the application. Border Gateway Protocol (BGP) uses autonomous system number (AS number) to identify the remote nodes. The AS number is a special number assigned by IANA used primarily with BGP to identify a network under a single network administration that uses unique routing policy.

**Configure BGP on Kubernetes using Ingress Citrix ADC**

Using a YAML file, you can apply BGP configuration of a remote node using the `kubectl create` command. In the YAML file, you need to add the peer IP address and the AS number. The peer IP address is the Ingress Citrix ADC IP address and the AS number is the AS number that is used in the Ingress Citrix ADC.

**Obtain the AS Number of the cluster**

Using the `calicoctl` command, you can obtain the AS number that is used by Calico BGP in the Kubernetes cluster as shown in the following image:

```
root@ubuntu194:/kubeCluster# ETCO_ENDPOINTS=http://10.102.13.194:6666 ./calicoctl get bgpConfiguration
NAME  LOCALSEVERITY  MESHENABLED  ASNUMBER
default  Info  false  64912
```
Configure global BGP peer

Using the `calicoctl` utility, you can peer Calico nodes with global BGP speakers. This kind of peers is called global peers.

Create a YAML definition file called `bgp.yml` with the following definition:

```yaml
apiVersion: projectcalico.org/v3  # This is the version of Calico
kind: BGPPeer  # BGPPeer specifies that its Global peering.
metadata:
  name: bgppeer-global-3040  # The name of the configuration
spec:
  peerIP: 10.102.33.208  # IP address of the Ingress Citrix ADC
  asNumber: 500  # AS number configured on the Ingress Citrix ADC
```

Deploy the definition file using the following command:

```bash
> kubectl create -f bgp.yml
```

Add the BGP configurations on the Ingress Citrix ADC

Perform the following:

1. Log on to the Citrix ADC command-line interface.
2. Enable the BGP feature using the following command:

```bash
> en feature bgp
Done
```
3. Type `vtysh` and press Enter.

```bash
> vtysh
ns#
```
4. Change to config terminal using the `conf t` command:
Enter configuration commands, one per line. End with CNTL/Z.

5. Add the BGP route with the AS number as 500 for demonstration purpose. You can use any number as the AS number.

```
ns(config)# router bgp 500
ns(config-router)#
```

6. Add neighbors using the following command:

```
ns(config-router)# Neighbor 10.102.33.198 remote-as 64512
ns(config-router)# Neighbor 10.102.22.202 remote-as 64512
```

7. Review the running configuration using the following command:

```
ns(config-router)#show running-config
! log syslog
! log record-priority
! ns route-install bgp
! interface lo0
  ip address 127.0.0.1/8
  ipv6 address fe80::1/64
  ipv6 address ::1/128
! interface vlan0
  ip address 10.102.33.208/24
  ipv6 address fe80::2cf6:beff:fe94:9f63/64
! router bgp 500
  max-paths ebgp 8
  max-paths ibgp 8
  neighbor 10.102.33.198 remote-as 64512
  neighbor 10.102.33.202 remote-as 64512
! end
```
In the sample, the AS number of Calico is 64512, you can change this number as per your requirement.

8. Install the BGP routes to Citrix ADC routing table using the following command:

```
ns(config)# ns route-install bgp
ns(config)#
exit
ns#exit
Done
```

9. Verify the route and add to the routing table using the following command:

```
sh route
+------------------+-----------------+----------------+-----------+----------------+
| Network | Netmask | Gateway/OwnerIP | State | Traffic Domain |
+------------------+-----------------+----------------+-----------+----------------+
| 0.0.0.0          | 255.0.0.0       | 10.102.33.1    | UP       | 0              |
| 127.0.0.0        | 255.0.0.0       | 127.0.0.1      | UP       | 0              |
| 192.168.1.0      | 255.255.255.0   | 192.168.1.1    | UP       | 0              |
| 192.168.1.128    | 255.255.255.0   | 192.168.1.128  | UP       | 0              |
| 192.168.71.64    | 255.255.255.0   | 192.168.71.64  | UP       | 0              |
Done
```

Once the route is installed, the Citrix ADC is able to communicate with services that are present in the Kubernetes cluster:

Troubleshooting

You can verify BGP configurations on the master node in the Kubernetes cluster using the `calicoctl` script.
View the peer IP address and AS number configurations

You can view the peer IP address and AS number configurations using the following command:

```
> ./calicoctl get bgpPeer
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>PEERIP</th>
<th>NODE</th>
<th>ASN</th>
</tr>
</thead>
<tbody>
<tr>
<td>bgppeer-global</td>
<td>10.102.33.208</td>
<td>(global)</td>
<td>500</td>
</tr>
</tbody>
</table>

View the BGP node status

You can view the status of a BGP node using the following command:

```
> calicoctl node status
```

---

Enhancements for Kubernetes service of type LoadBalancer support in the Citrix ingress controller

July 5, 2022

Kubernetes service of type LoadBalancer support in the Citrix ingress controller is enhanced with the following features:

- BGP route health injection (RHI) support
- Advertise or recall load balancer IP addresses (VIPs) based on the availability of service’s pods in a set of nodes (zones) defined by node’s labels

Support for automatic configuration of BGP RHI on Citrix ADC

Route health injection (RHI) allows the Citrix ADC to advertise the availability of a VIP as a host route throughout the network using BGP. However, you had to manually perform the configuration on Citrix ADC to support RHI. Using Citrix ingress controllers deployed in a Kubernetes environment, you can automate the configuration on Citrix ADCs to advertise VIPs.
When a service of type LoadBalancer is created, the Citrix ingress controller configures a VIP on the Citrix ADC for the service. If BGP RHI support is enabled for the Citrix ingress controller, it automatically configures Citrix ADC to advertise the VIP to the BGP network.

**Advertise and recall VIPs based on the availability of pods**

In the topology as shown in the following diagram, nodes in a Kubernetes cluster are physically distributed across three different racks. They are logically grouped into three zones. Each zone has a Citrix ADC MPX as the Tier-1 ADC and a Citrix ingress controller on the same in the Kubernetes cluster. Citrix ingress controllers in all zones listen to the same Kubernetes API server. So, whenever a service of type LoadBalancer is created, all Citrix ADCs in the cluster advertises the same IP address to the BGP fabric. Even, if there is no workload on a zone, the Citrix ADC in that zone still advertises the IP address.

Citrix provides a solution to advertise or recall the VIP based on the availability of pods in a zone. You need to label the nodes on each zone so that the Citrix ingress controller can identify nodes belonging to the same zone. The Citrix ingress controller on each zone performs a check to see if there are pods on nodes in the zone. If there are pods on nodes in the zone, it advertises the VIP. Otherwise, it revokes
the advertisement of VIP from the Citrix ADC on the zone.

**Configuring BGP RHI on Citrix ADCs using the Citrix ingress controller**

This topic provides information on how to configure BGP RHI on Citrix ADCs using the Citrix ingress controller based on a sample topology. In this topology, nodes in a Kubernetes cluster are deployed across two zones. Each zone has a Citrix ADC VPX or MPX as the Tier-1 ADC and a Citrix ingress controller for configuring ADC in the Kubernetes cluster. The ADCs are peered using BGP with the upstream router.

### Prerequisites

- You must configure Citrix ADC MPX or VPX as a BGP peer with the upstream routers.

Perform the following steps to configure BGP RHI support based on the sample topology.

1. Label nodes in each zone using the following command:

   For zone 1:

   ```
   kubectl label nodes node1 rack=rack-1
   kubectl label nodes node2 rack=rack-1
   ```

   For zone 2:

   ```
   kubectl label nodes node3 rack=rack-2
   kubectl label nodes node4 rack=rack-2
   ```
2. Configure the following environmental variables in the Citrix ingress controller configuration YAML files as follows:

For zone 1:

```yaml
- name: "NODE_LABELS"
  value: "rack-1"
- name: "BGP_ADVERTISEMENT"
  value: "True"
```

For zone 2:

```yaml
- name: "NODE_LABELS"
  value: "rack-2"
- name: "BGP_ADVERTISEMENT"
  value: "True"
```

A sample cic.yaml file for deploying the Citrix ingress controller on zone 1 is provided as follows:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: cic-k8s-ingress-controller-1
  labels:
    app: cic-k8s-ingress-controller-1
spec:
  serviceAccountName: cic-k8s-role
  containers:
  - name: cic-k8s-ingress-controller
    image: "quay.io/citrix/citrix-k8s-ingress-controller:1.26.7"
  env:
    # Set NetScaler NSIP/SNIP, SNIP in case of HA (mgmt has to be enabled)
    - name: "NS_IP"
      value: "10.217.212.24"
```
3. Deploy the Citrix ingress controller using the following command.

   Note:
   You need to deploy the Citrix ingress controller on both racks (per zone).

   ```bash
   Kubectl create -f cic.yaml
   ```


   ```bash
   Kubectl create -f web-frontend-lb.yaml
   ```

   The content of the `web-frontend-lb.yaml` is as follows:

   ```yaml
   apiVersion: v1
   kind: Deployment
   metadata:
     name: web-frontend
   spec:
     selector:
       matchLabels:
   ```
5. Create a service of type `LoadBalancer` for exposing the application.

```
Kubectl create -f web-frontend-lb-service.yaml
```

The content of the `web-frontend-lb-service.yaml` is as follows:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web-frontend
labels:
  app: web-frontend
spec:
  type: LoadBalancer
  ports:
  - port: 80
    protocol: TCP
    name: http
  selector:
    app: web-frontend
```

6. Verify the service group creation on Citrix ADCs using the following command.

```
show servicegroup <service-group-name>
```

Following is a sample output for the command.
```
# show servicegroup k8s-web-frontend_default_80_svc_k8s-web-frontend_default_80_svc

k8s-web-frontend_default_80_svc_k8s-web-frontend_default_80_svc - TCP
State: ENABLED Effective State: UP Monitor Threshold : 0
Max Conn: 0 Max Req: 0 Max Bandwidth: 0 kbits
Use Source IP: NO
Client Keepalive(CKA): NO
TCP Buffering(TCPB): NO
HTTP Compression(CMP): NO
Idle timeout: Client: 9000 sec Server: 9000 sec
Client IP: DISABLED
Cacheable: NO
SC: OFF
SP: OFF
Down state flush: ENABLED
Monitor Connection Close : NONE
Appflow logging: ENABLED
ContentInspection profile name: ???
Process Local: DISABLED
Traffic Domain: 0

1) 10.217.212.23:30126 State: UP Server Name: 10.217.212.23
   Server ID: None Weight: 1
   Last state change was at Wed Jan 22 23:35:11 2020
   Time since last state change: 5 days, 00:45:09.760
   Monitor Name: tcp-default State: UP Passive: 0
   Probes: 86941 Failed [Total: 0 Current: 0]
   Last response: Success - TCP syn+ack received.
   Response Time: 0 millisec

   Server ID: None Weight: 1
   Last state change was at Wed Jan 22 23:35:11 2020
   Time since last state change: 5 days, 00:45:09.790
   Monitor Name: tcp-default State: UP Passive: 0
   Probes: 86941 Failed [Total: 0 Current: 0]
   Last response: Success - TCP syn+ack received.
```
Citrix ADC ingress controller

7. Verify the VIP advertisement on the BGP router using the following command.

```
> VTYSH
# show ip route bgp
B 172.29.46.78/32 [200/0] via 2.2.2.100, vlan20, 1
d00h35m
[200/0] via 2.2.2.101, vlan20, 1
d00h35m
Gateway of last resort is not set
```

TLS certificates handling in Citrix ingress controller

February 4, 2022

Citrix ingress controller provides option to configure TLS certificates for Citrix ADC SSL-based virtual servers. The SSL virtual server intercepts SSL traffic, decrypts it and processes it before sending it to services that are bound to the virtual server.

By default, SSL virtual server can bind to one default certificate and the application receives the traffic based on the policy bound to the certificate. However, you have the Server Name Indication (SNI) option to bind multiple certificates to a single virtual server. Citrix ADC determines which certificate to present to the client based on the domain name in the TLS handshake.

Citrix ingress controller handles the certificates in the following three ways:

- Citrix ingress controller default Certificate
- Preconfigured certificates
- TLS Section in the Ingress YAML

**Prerequisite**

For handling TLS certificates using Citrix ingress controller, you need to enable TLS support in Citrix ADC for the application and also if you are using certificates in your Kubernetes deployment then you need to generate Kubernetes secret using the certificate.

**Enable TLS support in Citrix ADC for the application**

Citrix Ingress Controller uses the **TLS** section in the ingress definition as an enabler for TLS support with Citrix ADC.
Note:
In case of Default certificate or Preconfigured certificates, you need to add an empty secret in the `spec.tls.secretName` field in your ingress definition to enable TLS.

The following sample snippet of the ingress definition:

```
spec:
  tls:
    - secretName: <!--NeedCopy-->
```

Generate Kubernetes secret

To generate Kubernetes secret for an existing certificate, use the following `kubectl` command:

```
kubectl create secret tls k8s-secret --cert=path/to/tls.cert --key=path/to/tls.key --namespace=default
```

secret "k8s-secret" created

The command creates a Kubernetes secret with a PEM formatted certificate under `tls.crt` key and a PEM formatted private key under `tls.key` key.

Alternatively, you can also generate the Kubernetes secret using the following YAML definition:

```
apiVersion: v1
kind: Secret
metadata:
  name: k8s-secret
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
<!--NeedCopy-->
```

Deploy the YAML using the `kubectl -create <file-name>` command. It creates a Kubernetes secret with a PEM formatted certificate under `tls.crt` key and a PEM formatted private key under `tls.key` key.
Citrix ADC ingress controller

**Citrix ingress controller default certificate**

The Citrix ingress controller default certificate is used to provide a secret on Kubernetes that needs to be used as a non-SNI certificate. You must provide the secret name to be used and namespace from which it should be taken as arguments in the .yaml file of the Citrix ingress controller:

```
--default-ssl-certificate <NAMESPACE>/<SECRET_NAME>
```

The following is a sample Citrix ingress controller YAML definition file that contains a TLS secret (hotdrink.secret) picked from the ssl namespace and provided as the Citrix ingress controller default certificate.

**Note:**

NAMESPACE is mandatory along with a valid SECRET_NAME.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: cic
  labels:
    app: cic
spec:
  serviceAccountName: cpx
  containers:
  - name: cic
    image: "xxxx"
    imagePullPolicy: Always
    args:
      - --default-ssl-certificate
        ssl/hotdrink.secret
    env:
      # Set Citrix ADM Management IP
      - name: "NS_IP"
        value: "xx.xx.xx.xx"
      # Set port for Nitro
      - name: "NS_PORT"
        value: "xx"
      # Set Protocol for Nitro
      - name: "NS_PROTOCOL"
        value: "HTTP"
      # Set username for Nitro
      - name: "NS_USER"
```
Citrix ADC ingress controller

```yaml
28 value: \"nsroot\"
29 # Set user password for Nitro
30 - name: \"NS_PASSWORD\"
31 value: \"nsroot\"
32 <!--NeedCopy--> 
```

**Preconfigured certificates**

Citrix ingress controller allows you to use the certkeys that are already configured on the Citrix ADC. You must provide the details about the certificate using the following annotation in your ingress definition:

```yaml
1 ingress.citrix.com/preconfigured-certkey : '{
2   "certs": [ {
3     "name": "<name>", "type": "default|sni|ca" }
4   ] }
5 '
```

You can provide details about multiple certificates as a list within the annotation. Also, you can define the way the certificate is treated. In the following sample annotation, certkey1 is used as a non-SNI certificate and certkey2 is used as an SNI certificate:

```yaml
1 ingress.citrix.com/preconfigured-certkey : '{
2   "certs": [ {
3     "name": "certkey1", "type": "default" }
4   , {
5     "name": "certkey2", "type": "sni" }
6   ] }
7 '
```

If the type parameter is not provided with the name of a certificate, then it is considered as the default (non-SNI) type.

**Note:**

Ensure that you use this feature in cases where you want to reuse the certificates that are present on the Citrix ADC and bind them to the applications that are managed by Citrix ingress controller. Citrix ingress controller does not manage the life cycle of the certificates. That is, it does not create or delete the certificates, but only binds them to the necessary applications.
TLS Section in the Ingress YAML

Kubernetes allows you to provide the TLS secrets in the `spec:` section of an ingress definition. This section describes how the Citrix ingress controller uses these secrets.

With the host section

If the secret name is provided with the host section, Citrix ingress controller binds the secret as an SNI certificate.

```yaml
spec:
  tls:
    - secretName: fruitjuice.secret
      hosts:
        - items.fruit.juice

<!-- NeedCopy -->
```

Without the host section

If the secret name is provided without the host section, Citrix ingress controller binds the secret as a default certificate.

```yaml
spec:
  tls:
    - secretName: colddrink.secret

<!-- NeedCopy -->
```

Note:

If there are more than one secret given then Citrix ingress controller binds all the certificates as SNI enabled certificates.

Points to note

1. When, multiple secrets are provided to the Citrix ingress controller, the following precedence is followed:
   a) preconfigured-default-certkey or non-host tls secret
   b) default-ssl-certificate
2. If there is a conflict in precedence among the same grade certificates (for example, two ingress files configure a non-host TLS secret each, as default/non-SNI type), then the Citrix ingress controller binds the Citrix ingress controller default certificate as the non-SNI certificate and uses all other certificates with SNI.

3. Certificate used for a secret given under the TLS section must have a CN name. Otherwise, it does not bind to Citrix ADC.

4. If SNI enabled for SSL virtual server then:
   - Non-SNI (Default) certificate is used for the following HTTPS requests:

```
1  curl -1 -v -k https://1.1.1.1/
2  curl -1 -v -k -H 'HOST:* colddrink.beverages' https://1.1.1.1/
```

   - SNI enabled certificate is used for a request with full domain name:

```
1  curl -1 -v -k https://items.colddrink.beverages/
```

If any request is received that does not match with certificates, CN name fails.

**TLS client authentication support in Citrix ADC**

February 3, 2022

In TLS client authentication, a server requests a valid certificate from the client for authentication and ensures that it is only accessible by authorized machines and users.

You can enable TLS client authentication using Citrix ADC SSL-based virtual servers. With client authentication enabled on a Citrix ADC SSL virtual server, the Citrix ADC asks for the client certificate during the SSL handshake. The appliance checks the certificate presented by the client for normal constraints, such as the issuer signature and expiration date.

The following diagram explains the TLS client authentication feature on Citrix ADC.
TLS client authentication can be set to mandatory, or optional. If the SSL client authentication is set as mandatory and the SSL Client does not provide a valid client certificate, then the connection is dropped. A valid client certificate means that it is signed or issued by a specific Certificate Authority, and not expired or revoked. If it is marked as optional, then the Citrix ADC requests the client certificate, but the connection is not dropped. The Citrix ADC proceeds with the SSL transaction even if the client does not present a certificate or the certificate is invalid. The optional configuration is useful for authentication scenarios like two-factor authentication.

### Configuring TLS client authentication

Perform the following steps to configure TLS client authentication.

1. **Enable the TLS support in Citrix ADC.**
   
   The Citrix ingress controller uses the **TLS** section in the Ingress definition as an enabler for TLS support with Citrix ADC. The following is a sample snippet of the Ingress definition:

   ```yaml
   spec:
   tls:
     - secretName:
   ```

2. **Apply a CA certificate to the Kubernetes environment.**
   
   To generate a Kubernetes secret for an existing certificate, use the following kubectl command:
3. Configure Ingress to enable client authentication.

You need to specify the following annotation to attach the generated CA secret which is used for client certificate authentication for a service deployed in Kubernetes.

```yaml
ingress.citrix.com/ca-secret: '{
  "frontend-hotdrinks": "hotdrink-ca-secret"
}
```

By default, client certificate authentication is set to mandatory but you can configure it to optional using the `frontend_sslprofile` annotation in the front end configuration.

```yaml
ingress.citrix.com/frontend_sslprofile: '{
  "clientauth": "ENABLED",
  "clientcert": "optional"
}
```

TLS server authentication support in Citrix ADC using the Citrix ingress controller

February 3, 2022

Server authentication allows a client to verify the authenticity of the web server that it is accessing. Usually, the Citrix ADC device performs SSL offload and acceleration on behalf of a web server and does not authenticate the certificate of the Web server. However, you can authenticate the server in deployments that require end-to-end SSL encryption.

In such a situation, the Citrix ADC device becomes the SSL client and performs the following:
Citrix ADC ingress controller

- carries out a secure transaction with the SSL server
- verifies that a CA whose certificate is bound to the SSL service has signed the server certificate
- checks the validity of the server certificate.

To authenticate the server, you must first enable server authentication and then bind the certificate of the CA that signed the certificate of the server to the SSL service on the Citrix ADC appliance. When you bind the certificate, you must specify the bind as a CA option.

**Configuring TLS server authentication**

Perform the following steps to configure TLS server authentication.

1. Enable the TLS support in Citrix ADC.

   The Citrix ingress controller uses the **TLS** section in the Ingress definition as an enabler for TLS support with Citrix ADC.

   The following is a sample snippet of the Ingress definition:

   ```
   spec:
   tls:
   - secretName:
   ```

2. To generate a Kubernetes secret for an existing certificate, perform the following.

   a) Generate a client certificate to be used with the service.

   ```
   kubectl create secret tls tea-beverage --cert=path/to/tls.cert --key=path/to/tls.key --namespace=default
   ```

   b) Generate a secret for an existing CA certificate. This certificate is required to sign the back end server certificate.

   ```
   kubectl create secret generic tea-ca --from-file=tls.crt=cacerts.pem
   ```

   **Note:**

   You must specify **tls.crt** while creating a secret. This file is used by the Citrix ingress controller while parsing a CA secret.
3. Enable secure back end communication to the service using the following annotation in the Ingress configuration.

   ```
   ingress.citrix.com/secure-backend: "True"
   ```

4. Use the following annotation to bind the certificate to SSL service. This certificate is used when the Citrix ADC acts as a client to send the request to the back end server.

   ```
   ingress.citrix.com/backend-secret: '{
   "tea-beverage": "tea-beverage", "coffee-beverage": "coffee-beverage"
   }
   ```

5. To enable server authentication which authenticates the back end server certificate, you can use the following annotation. This configuration binds the CA certificate of the server to the SSL service on the Citrix ADC.

   ```
   ingress.citrix.com/backend-ca-secret: '{
   "coffee-beverage":"coffee-ca", "tea-beverage":"tea-ca"
   }
   ```

Install, link, and update certificates on a Citrix ADC using the Citrix ingress controller

January 19, 2022

On the Ingress Citrix ADC, you can install, link, and update certificates. Many server certificates are signed by multiple hierarchical certificate authorities (CAs). This means that certificates form a chain.

A certificate chain is an ordered list of certificates containing an SSL certificate and certificate authority (CA) certificates. It enables the receiver to verify that the sender and all CAs are trustworthy. The chain or path begins with the SSL certificate, and each certificate in the chain is signed by the entity identified by the next certificate in the chain.

Any certificate that sits between the SSL certificate and the root certificate is called a chain or intermediate certificate. The intermediate certificate is the signer or issuer of the SSL certificate. The root CA certificate is the signer or issuer of the intermediate certificate.
If the intermediate certificate is not installed on the server (where the SSL certificate is installed) it may prevent some browsers, mobile devices, and applications from trusting the SSL certificate. To make the SSL certificate compatible with all clients, it is necessary that the intermediate certificate is installed.

Certificates linking in Kubernetes

The Citrix ingress controller supports automatic provisioning and renewal of TLS certificates using the Kubernetes cert-manager. The cert-manager issues certificates from different sources, such as Let's Encrypt and HashiCorp Vault and converts them to Kubernetes secrets.

The following diagram explains how the cert-manager performs certificate management.
When you create a Kubernetes secret from a PEM certificate embedded with multiple CA certificates, you need to link the server certificates with the associated CAs. While applying the Kubernetes secret, you can link the server certificates with all the associated CAs using the Ingress Citrix ADC. Linking the server certificates and CAs enable the receiver to verify if the sender and CAs are trustworthy.

The following is a sample Ingress definition:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: frontendssl
spec:
rules:
- host: frontend.com
  http:
    paths:
    - backend:
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
Citrix ADC ingress controller

```yaml
  service:
    name: frontend
    port:
      number: 443
      path: /web-frontend/frontend.php
      pathType: Prefix
    tls:
      secretName: certchain1

<!-- NeedCopy -->
```

On the Citrix ADC, you can verify if certificates are added to the Citrix ADC. Perform the following:

1. Log on to the Citrix ADC command-line interface.
2. Verify if certificates are added to the Citrix ADC using the following command:

   ```
   > show certkey
   ```

   For sample outputs, see the Citrix ADC documentation.
3. Verify that the server certificate and CAs are linked using the following command:

   ```
   > show certlink
   ```

   **Output:**

   ```
   1) Cert Name: k8s-3KC24EQYHG6ZKEDAY5Y3SG26MT2  CA Cert Name: k8s-3KC24EQYHG6ZKEDAY5Y3SG2_ic1
   2) Cert Name: k8s-3KC24EQYHG6ZKEDAY5Y3SG2_ic1  CA Cert Name: k8s-3KC24EQYHG6ZKEDAY5Y3SG2_ic2
   ```

**Configure SSL passthrough using Kubernetes Ingress**

February 3, 2022

SSL passthrough feature allows you to pass incoming security sockets layer (SSL) requests directly to a server for decryption rather than decrypting the request using a load balancer. SSL passthrough is widely used for web application security and it uses the TCP mode to pass encrypted data to servers.
The proxy SSL passthrough configuration does not require the installation of an SSL certificate on the load balancer. SSL certificates are installed on the back end server as they handle the SSL connection instead of the load balancer.

The following diagram explains the SSL passthrough feature.

As shown in this diagram, SSL traffic is not terminated at the Citrix ADC and SSL traffic is passed through the Citrix ADC to the back end server. SSL certificate at the back end server is used for the SSL handshake.

The Citrix ingress controller provides the following Ingress annotation that you can use to enable SSL passthrough on the Ingress Citrix ADC:

```
-ingress.citrix.com/ssl-passthrough: 'True|False'
```

The default value of the annotation is `False`.

SSL passthrough is enabled for all services or host names provided in the Ingress definition. SSL
Citrix ADC ingress controller

passthrough uses host name (wildcard host name is also supported) and ignores paths given in Ingress.

**Note:**
The Citrix ingress controller does not support SSL passthrough for non-hostname based Ingress. Also, SSL passthrough is not valid for default back end Ingress.

To configure SSL passthrough on the Ingress Citrix ADC, you must define the `ingress.citrix.com/ssl-passthrough:` as shown in the following sample Ingress definition. You must also enable TLS for the host as shown in the example.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    ingress.citrix.com/frontend-ip: x.x.x.x
    ingress.citrix.com/insecure-termination: redirect
    ingress.citrix.com/secure-backend: "True"
    ingress.citrix.com/ssl-passthrough: "True"
    kubernetes.io/ingress.class: citrix
name: hotdrinks-ingress
spec:
rules:
- host: hotdrinks.beverages.com
  http:
    paths:
    - backend:
      service:
        name: frontend-hotdrinks
        port:
          number: 443
      path: /
      pathType: Prefix
    tls:
    - secretName: beverages
<!--NeedCopy-->

Automated certificate management with cert-manager

February 3, 2022
Citrix ADC ingress controller

Citrix ingress controller supports automatic provisioning and renewal of TLS certificates using cert-manager. The cert-manager is a native Kubernetes certificate management controller. It issues certificates from different sources, such as Let's Encrypt and HashiCorp Vault.

As shown in the following diagram, cert-manager interacts with the external Certificate Authorities (CA) to sign the certificates and converts it to Kubernetes secrets. These secrets are used by Citrix ingress controller to configure SSL virtual server on the Citrix ADC.

For detailed configurations, refer:

- Deploying HTTPS web applications on Kubernetes with Citrix ingress controller and Let’s Encrypt using cert-manager
- Deploying HTTPS web application on Kubernetes with Citrix ingress controller and HashiCorp Vault using cert-manager
Deploy HTTPS web application on Kubernetes with the Citrix ingress controller and Let’s Encrypt using cert-manager

February 15, 2022

Let’s Encrypt and the ACME (Automatic Certificate Management Environment) protocol enables you to set up an HTTPS server and automatically obtain a browser-trusted certificate. To get a certificate for your website’s domain from Let’s Encrypt, you have to demonstrate control over the domain by accomplishing certain challenges. A challenge is one among a list of specified tasks that only someone who controls the domain can accomplish.

Currently there are two types of challenges:

- HTTP-01 challenge: HTTP-01 challenges are completed by posting a specified file in a specified location on a website. Let’s Encrypt CA verifies the file by making an HTTP request on the HTTP URI to satisfy the challenge.

- DNS-01 challenge: DNS-01 challenges are completed by providing a computed key that is present at a DNS TXT record. Once this TXT record has been propagated across the internet, the ACME server can successfully retrieve this key via a DNS lookup. The ACME server can validate that the client owns the domain for the requested certificate. With the correct permissions, cert-manager automatically presents this TXT record for your specified DNS provider.

On successful validation of the challenge, a certificate is granted for the domain.

This topic provides information on how to securely deploy an HTTPS web application on a Kubernetes cluster, using:

- The Citrix ingress controller
- JetStack’s cert-manager to provision TLS certificates from the Let’s Encrypt project.

Prerequisites

Ensure that you have:

- The domain for which the certificate is requested is publicly accessible.
- Enabled RBAC on your Kubernetes cluster.
- Deployed Citrix ADC MPX, VPX, or CPX deployed in Tier 1 or Tier 2 deployment model.

In the Tier 1 deployment model, Citrix ADC MPX or VPX is used as an Application Delivery Controller (ADC). The Citrix ingress controller running in Kubernetes cluster configures the virtual services for services running on Kubernetes cluster. Citrix ADC runs the virtual service on the publicly routable IP address and offloads SSL for client traffic with the help of the Let’s Encrypt generated certificate.
In the Tier 2 deployment model, a TCP service is configured on the Citrix ADC (VPX/MPX) running outside the Kubernetes cluster. This service is created to forward the traffic to Citrix ADC CPX instances running in the Kubernetes cluster. Citrix ADC CPX ends the SSL session and load-balances the traffic to actual service pods.

• Deployed the Citrix ingress controller. Click [here](#) for various deployment scenarios.

• Opened port 80 for the virtual IP address on the firewall for the Let's Encrypt CA to validate the domain for HTTP01 challenge.

• A DNS domain that you control, where you host your web application for the ACME DNS01 challenge.

• Administrator permissions for all deployment steps. If you encounter failures due to permissions, make sure you have administrator permissions.

**Install cert-manager**

To install cert-manager, see the [cert-manager installation documentation](#).

You can install cert-manager either using manifest files or Helm chart.

Once you install the cert-manager, verify that cert-manager is up and running as explained [verifying the installation](#).

**Deploy a sample web application**

Perform the following to deploy a sample web application:

**Note:**

Kuard, a Kubernetes demo application is used for reference in this topic.

1. Create a deployment YAML file (`kuard-deployment.yaml`) for Kuard with the following configuration:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: kuard
    name: kuard
spec:
  replicas: 1
  selector:
    matchLabels:
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
2. Deploy the Kuard deployment file (kuard-deployment.yaml) to your cluster, using the following commands:

% kubectl create -f kuard-deployment.yaml

deployment.extensions/kuard created

% kubectl get pod -l app=kuard

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>kuard-fc4d89bfb-djljt</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>24s</td>
</tr>
</tbody>
</table>

3. Create a service for the deployment. Create a file called service.yaml with the following configuration:

```yaml
apiVersion: v1
deployment.extensions/
kind: Service
metadata:
  name: kuard
spec:
  ports:
    - port: 80
targetPort: 8080
  protocol: TCP
selector:
```
Citrix ADC ingress controller

```yaml
app: kuard
<--NeedCopy-->
```

4. Deploy and verify the service using the following commands:

```
% kubectl create -f service.yaml

service/kuard created

% kubectl get svc kuard
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER-IP</th>
<th>EXTERNAL-IP</th>
<th>PORT(S)</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>kuard</td>
<td>ClusterIP</td>
<td>10.103.49.171</td>
<td>&lt;none&gt;</td>
<td>80/TCP</td>
<td>13s</td>
</tr>
</tbody>
</table>

5. Expose this service to outside world by creating an Ingress that is deployed on Citrix ADC CPX or VPX as Content switching virtual server.

**Note:**
Ensures that you change the value of `kubernetes.io/ingress.class` to your ingress class on which the Citrix ingress controller is started.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: citrix
  name: kuard
spec:
  rules:
    - host: kuard.example.com
      http:
        paths:
          - backend:
              service:
                name: kuard
                port:
                  number: 80
              path: /
              pathType: Prefix
```

**Note:**
You must change the value of `spec.rules.host` to the domain that you control. Ensure that a
Citrix ADC ingress controller

DNS entry exists to route the traffic to Citrix ADC CPX or VPX.

1. Deploy the Ingress using the following command:

   ```bash
   % kubectl apply -f ingress.yml
   ingress.extensions/kuard created
   ``

   ```bash
   root@ubuntu-:~/cert-manager# kubectl get ingress
   NAME      HOSTS          ADDRESS        PORTS     AGE
   kuard     kuard.example.com  80           7s
   ```

2. Verify that the Ingress is configured on Citrix ADC CPX or VPX by using the following command:

   ```bash
   $ kubectl exec -it cpx-ingress-5b85d7c69d-ngd72 /bin/bash
   root@cpx-ingress-55c88788fd-qd4rg:/# cli_script.sh 'show cs vserver'
   exec: show cs vserver
   1) k8s-192.168.8.178_80_http (192.168.8.178:80) - HTTP Type: CONTENT
   State: UP
   Last state change was at Sat Jan 4 13:36:14 2020
   Time since last state change: 0 days, 00:18:01.950
   Client Idle Timeout: 180 sec
   Down state flush: ENABLED
   Disable Primary Vserver On Down : DISABLED
   Comment: uid=MPPL57E3AFY6NMNGDKN2VT57HEZV0V53Z7DWKH44X25GLIH4ZWQ===
   Appflow logging: ENABLED
   Port Rewrite : DISABLED
   State Update: DISABLED
   Default:  Content Precedence: RULE
   Vserver IP and Port insertion: OFF
   L2Conn:  OFF Case Sensitivity: ON
   Authentication: OFF
   401 Based Authentication: OFF
   Push:  DISABLED  Push VServer: none
   Push Label Rule: none
   Persistence: NONE
   Listen Policy: NONE
   IcmpResponse: PASSIVE
   RHIsate: PASSIVE
   Traffic Domain: 0
3. Verify that the webpage is correctly being served when requested using the `curl` command.

```bash
% curl -sS -D - kuard.example.com -o /dev/null
HTTP/1.1 200 OK
Content-Length: 1458
Content-Type: text/html
Date: Thu, 21 Feb 2019 09:09:05 GMT
```

**Configure issuing ACME certificate using the HTTP challenge**

This section describes a way to issue the ACME certificate using the HTTP validation. If you want to use the DNS validation, skip this section and proceed to the next section.

The HTTP validation using cert-manager is a simple way of getting a certificate from Let's Encrypt for your domain. In this method, you prove ownership of a domain by ensuring that a particular file is present at the domain. It is assumed that you control the domain if you are able to publish the given file under a given path.

**Deploy the Let's Encrypt ClusterIssuer with the HTTP01 challenge provider**

The cert-manager supports two different CRDs for configuration, anIssuer, scoped to a single namespace, and aClusterIssuer, with cluster-wide scope.

For the Citrix ingress controller to use the Ingress from any namespace, useClusterIssuer. Alternatively, you can also create anIssuer for each namespace on which you are creating an Ingress resource.

For more information, see cert-manager documentation for HTTP validation.

1. Create a file calledissuer-letsencrypt-staging.yamlwith the following configuration:

```yaml
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: letsencrypt-staging
spec:
```
```yaml
acme:
  # You must replace this email address with your own.
  # Let’s Encrypt will use this to contact you about expiring certificates, and issues related to your account.
  email: user@example.com
  server: https://acme-staging-v02.api.letsencrypt.org/directory
  privateKeySecretRef:
    # Secret resource used to store the account’s private key.
    name: example-issuer-account-key
  # Add a single challenge solver, HTTP01 using citrix
  solvers:
    - http01:
      ingress:
        class: citrix
```

**spec.acme.solvers[].http01.ingress.class** refers to the Ingress class of Citrix ingress controller. If the Citrix ingress controller has no ingress class, you do not need to specify this field.

**Note:**
This is a sample *Clusterissuer* of cert-manager.io/v1alpha2 resource. For more information, see cert-manager http01 documentation.

The staging Let’s Encrypt server issues fake certificate, but it is not bound by the API rate limits of the production server. This approach lets you set up and test your environment without worrying about rate limits. You can repeat the same step for the Let’s Encrypt production server.

2. After you edit and save the file, deploy the file using the following command:

```
% kubectl apply -f issuer-letsencrypt-staging.yaml
clusterissuer "letsencrypt-staging" created
```

3. Verify that the issuer is created and registered to the ACME server.

```
% kubectl get issuer
NAME          AGE
letsencrypt-staging  8d
```

4. Verify that the *ClusterIssuer* is properly registered using the command `kubectl describe issuer letsencrypt-staging:`
Issue certificate for the Ingress object

Once ClusterIssuer is successfully registered, you can get a certificate for the Ingress domain ‘kuard.example.com’.

You can request a certificate for the specified Ingress resource using the following methods:

- Adding Ingress-shim annotations to the ingress object.
- Creating a certificate CRD object.

The first method is quick and simple, but if you need more customization and granularity in terms of certificate renewal, you can choose the second method. You can choose the method according to your needs.

Adding Ingress-shim annotations to the Ingress object

In this approach, you add the following two annotations to the Ingress object for which you request a certificate from the ACME server.

```
certmanager.io/cluster-issuer: "letsencrypt-staging"
```

**Note:**

You can find all supported annotations from cert-manager for Ingress-shim, at supported-annotations.

Also, modify the ingress.yaml to use TLS by specifying a secret.
The `cert-manager.io/cluster-issuer: "letsencrypt-staging"` annotation tells cert-manager to use the `letsencrypt-staging` cluster-wide issuer to request a certificate from Let's Encrypt's staging servers. Cert-manager creates a certificate object that is used to manage the lifecycle of the certificate for `kuard.example.com`. The value for the domain name and challenge method for the certificate object is derived from the ingress object. Cert-manager manages the contents of the secret as long as the Ingress is present in your cluster.

Deploy the `ingress.yaml` file using the following command:

```
% kubectl apply -f ingress.yaml

ingress.extensions/kuard configured
```

```
% kubectl get ingress kuard

NAME   HOSTS             ADDRESS   PORTS  AGE
kuard  kuard.example.com 80, 443   4h39m
```
Alternatively, you can deploy a certificate CRD object independent of the Ingress object. Documentation of “certificate” CRD can be found at [HTTP validation](http://example.com).

1. Create the `certificate.yaml` file with the following configuration:

```yaml
apiVersion: cert-manager.io/v1alpha2
kind: Certificate
metadata:
  name: example-com
  namespace: default
spec:
  secretName: kuard-example-tls
  issuerRef:
    name: letsencrypt-staging
  commonName: kuard.example.com
  dnsNames:
  - www.kuard.example.com
<!--NeedCopy-->
```

The `spec.secretName` key is the name of the secret where the certificate is stored on successfully issuing the certificate.

1. Deploy the `certificate.yaml` file on the Kubernetes cluster:

```
kubectl create -f certificate.yaml
certificate.cert-manager.io/example-com created
```

2. Verify that certificate custom resource is created by the cert-manager which represents the certificate specified in the Ingress. After few minutes, if ACME validation goes well, certificate ‘READY’ status is set to true.

```
kubectl get certificates.cert-manager.io kuard-example-tls
NAME     READY     SECRET     AGE
kuard-example-tls True kuard-example-tls 3m44s

% kubectl get certificates.cert-manager.io kuard-example-tls
Name: kuard-example-tls
Namespace: default
Labels: <none>
Annotations: <none>
```
3. Verify that the secret resource is created.
Issuing an ACME certificate using the DNS challenge

This section describes a way to use the DNS validation to get the ACME certificate from Let’sEncrypt CA. With a DNS-01 challenge, you prove the ownership of a domain by proving that you control its DNS records. This is done by creating a TXT record with specific content that proves you have control of the domain’s DNS records. For detailed explanation of DNS challenge and best security practices in deploying DNS challenge, see [A Technical Deep Dive: Securing the Automation of ACME DNS Challenge Validation](#).

**Note:**
In this procedure, route53 is used as the DNS provider. For other providers, see cert-manager documentation of DNS validation.

Deploy the Let’s Encrypt ClusterIssuer with the DNS01 challenge provider

Perform the following to deploy the Let’s Encrypt ClusterIssuer with the DNS01 challenge provider:

1. Create an AWS IAM user account and download the secret access key ID and secret access key.
2. Grant the following IAM policy to your user:

   ```
   Route53 access policy
   ```

3. Create a Kubernetes secret `acme-route53` in `kube-system` namespace.

   ```
   kubectl create secret generic acme-route53 --from-literal secret-access-key=<secret_access_key>
   ```

4. Create an [Issuer](#) or [ClusterIssuer](#) with the DNS01 challenge provider.

   You can provide multiple providers under DNS01, and specify which provider to be used at the time of certificate creation.

   You must have access to the DNS provider for cert-manager to create a TXT record. Credentials are stored in the Kubernetes secret specified in `spec.dns01.secretAccessKeySecretRef`. For detailed instructions on how to obtain credentials, see the DNS provider documentation.
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: letsencrypt-staging
spec:
  acme:
    # You must replace this email address with your own.
    # Let's Encrypt will use this to contact you about expiring certificates, and issues related to your account.
    email: user@example.com
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: example-issuer-account-key
    solvers:
      - dns01:
          route53:
            region: us-west-2
            accessKeyId: <IAMKEY>
            secretAccessKeySecretRef:
              name: acme-route53
              key: secret-access-key

Note:
Replace user@example.com with your email address. For each domain mentioned in a DNS01 stanza, cert-manager uses the provider's credentials from the referenced Issuer to create a TXT record called _acme-challenge. This record is then verified by the ACME server to issue the certificate. For more information about the DNS provider configuration, and the list of supported providers, see DNS01 reference doc.

5. After you edit and save the file, deploy the file using the following command:

   % kubectl apply -f acme_clusterissuer_dns.yaml
   clusterissuer "letsencrypt-staging" created

6. Verify if the issuer is created and registered to the ACME server using the following command:

   % kubectl get issuer
   NAME   AGE
7. Verify if the ClusterIssuer is properly registered using the command `kubectl describe issuer letsencrypt-staging`:

```bash
Status:
  Acme:
    Uri: https://acme-staging-v02.api.letsencrypt.org/acme/acct/8200869
Conditions:
  Last Transition Time: 2019-02-11T12:06:31Z
  Message: The ACME account was registered with the ACME server
  Reason: ACMEAccountRegistered
  Status: True
  Type: Ready
```

**Issue certificate for the Ingress resource**

Once the issuer is successfully registered, you can get a certificate for the ingress domain `kuard.example.com`. Similar to HTTP01 challenge, there are two ways you can request the certificate for a specified Ingress resource:

- Adding `Ingress-shim` annotations to the Ingress object.
- Creating a `certificate` CRD object. For detailed instructions, see Create a Certificate CRD resource.

**Adding Ingress-shim annotations to the ingress object**

Add the following annotation to the Ingress object along with the `spec.tls` section:

```yaml
certmanager.io/cluster-issuer: "letsencrypt-staging"
```

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
```
The cert-manager creates a Certificate CRD resource with the DNS01 challenge. It uses credentials specified in the ClusterIssuer to create a TXT record in the DNS server for the domain you own. Then, Let’s Encrypt CA validates the content of the TXT record to complete the challenge.

**Adding a Certificate CRD resource**

Alternatively, you can explicitly create a certificate custom resource definition resource to trigger automatic generation of certificates.

1. Create the certificate.yaml file with the following configuration:

```yaml
apiVersion: cert-manager.io/v1alpha2
kind: Certificate
metadata:
  name: example-com
  namespace: default
spec:
  secretName: kuard-example-tls
  issuerRef:
    name: letsencrypt-staging
  commonName: kuard.example.com
  dnsNames:
```
After successful validation of the domain name, certificate READY status is set to True.

2. Verify that the certificate is issued.

```bash
% kubectl get certificate kuard-example-tls
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>SECRET</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>example-tls</td>
<td>True</td>
<td>kuard-example-tls</td>
<td>10m</td>
</tr>
</tbody>
</table>

You can watch the progress of the certificate as it is issued, using the following command:

```bash
% kubectl describe certificates kuard-example-tls | tail -n 6
```

```
Not After: 2020-04-04T13:34:23Z
Events:

## Normal
- Requested 11m cert-manager Created new CertificateRequest resource "kuard-example-tls-3030465986"
- Issued 7m21s cert-manager Certificate issued successfully
```

### Verify certificate in Citrix ADC

Let's encrypt CA successfully validated the domain and issued a new certificate for the domain. A kubernetes.io/tls secret is created with the secretName specified in the tls: field of the Ingress. Also, cert-manager automatically initiates a renewal, 30 days before the expiry.

For HTTP challenge, cert-manager creates a temporary Ingress resource to route the Let's Encrypt CA generated traffic to cert-manager pods. On successful validations of the domain, this temporary Ingress is deleted.

1. Verify that the secret is created using the following command:

```bash
% kubectl get secret kuard-example-tls
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>DATA</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>kuard-example-tls</td>
<td>kubernetes.io/tls</td>
<td>3</td>
<td>30m</td>
</tr>
</tbody>
</table>
The Citrix ingress controller picks up the secret and binds the certificate to the content switching virtual server on the Citrix ADC CPX. If there are any intermediate CA certificates, it is automatically linked to the server certificate and presented to the client during SSL negotiation.

2. Log on to Citrix ADC CPX and verify if the certificate is bound to the SSL virtual server.

```
% kubectl exec -it cpx-ingress-55c88788fd-n2x9r bash -c cpx-ingress
Defaulting container name to cpx-ingress.

% cli_script.sh 'sh ssl vs k8s-192.168.8.178_443_ssl'
exec: sh ssl vs k8s-192.168.8.178_443_ssl

Advanced SSL configuration for VServer k8s-192.168.8.178_443_ssl:

DH: DISABLED
DH Private-Key Exponent Size Limit: DISABLED Ephemeral RSA: ENABLED  Refresh Count: 0
Session Reuse: ENABLED  Timeout: 120 seconds
Cipher Redirect: DISABLED
ClearText Port: 0
Client Auth: DISABLED
SSL Redirect: DISABLED
Non FIPS Ciphers: DISABLED
SNI: ENABLED
OCSP Stapling: DISABLED
HSTS: DISABLED
HSTS IncludeSubDomains: NO
HSTS Max-Age: 0
HSTS Preload: NO
SSLv3: ENABLED  TLSv1.0: ENABLED  TLSv1.1: ENABLED  TLSv1.2: ENABLED  TLSv1.3: DISABLED
Push Encryption Trigger: Always
Send Close-Notify: YES
Strict Sig-Digest Check: DISABLED
Zero RTT Early Data: DISABLED
DHE Key Exchange With PSK: NO
Tickets Per Authentication Context: 1
, P_256, P_384, P_224, P_5216) CertKey Name: k8s-GVWNYGVZKRRHKF7MZVTOAEZYBS Server Certificate for SNI
7) Cipher Name: DEFAULT
```
<table>
<thead>
<tr>
<th>Description: Default cipher list with encryption strength &gt;= 128 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>cli_script.sh 'sh certkey'</td>
</tr>
<tr>
<td>1) Name: k8s-GVWNYGVZKKRHKF7M2ZTLOAEZYBS</td>
</tr>
<tr>
<td>Cert Path: k8s-GVWNYGVZKKRHKF7M2ZTLOAEZYBS.crt</td>
</tr>
<tr>
<td>Key Path: k8s-GVWNYGVZKKRHKF7M2ZTLOAEZYBS.key</td>
</tr>
<tr>
<td>Form: PEM</td>
</tr>
<tr>
<td>Status: Valid, Days to expiration:89</td>
</tr>
<tr>
<td>Certificate Expiry Monitor: ENABLED</td>
</tr>
<tr>
<td>Expiry Notification period: 30 days</td>
</tr>
<tr>
<td>Certificate Type: &quot;Client Certificate&quot; &quot;Server Certificate&quot;</td>
</tr>
<tr>
<td>Version: 3</td>
</tr>
<tr>
<td>Serial Number: 03B2B57EA9E61A93F1D05EA3272FA95203C2</td>
</tr>
<tr>
<td>Signature Algorithm: sha256WithRSAEncryption</td>
</tr>
<tr>
<td>Issuer: O=Digital Signature Trust Co.,CN=DST Root CA X3</td>
</tr>
<tr>
<td>Validity</td>
</tr>
<tr>
<td>Not Before: Jan 5 13:34:23 2020 GMT</td>
</tr>
<tr>
<td>Not After : Apr 4 13:34:23 2020 GMT</td>
</tr>
<tr>
<td>Subject: CN=acme.cloudpst.net</td>
</tr>
<tr>
<td>Public Key Algorithm: rsaEncryption</td>
</tr>
<tr>
<td>Public Key size: 2048</td>
</tr>
<tr>
<td>Ocsp Response Status: NONE</td>
</tr>
<tr>
<td>2) Name: k8s-GVWNYGVZKKRHKF7M2ZTLOAEZYBS_ic1</td>
</tr>
<tr>
<td>Cert Path: k8s-GVWNYGVZKKRHKF7M2ZTLOAEZYBS.crt_ic1</td>
</tr>
<tr>
<td>Form: PEM</td>
</tr>
<tr>
<td>Status: Valid, Days to expiration:437</td>
</tr>
<tr>
<td>Certificate Expiry Monitor: ENABLED</td>
</tr>
<tr>
<td>Expiry Notification period: 30 days</td>
</tr>
<tr>
<td>Certificate Type: &quot;Intermediate CA&quot;</td>
</tr>
<tr>
<td>Version: 3</td>
</tr>
<tr>
<td>Serial Number: 0A014142000015385736A0B85ECA708</td>
</tr>
<tr>
<td>Signature Algorithm: sha256WithRSAEncryption</td>
</tr>
<tr>
<td>Issuer: O=Digital Signature Trust Co.,CN=DST Root CA X3</td>
</tr>
<tr>
<td>Validity</td>
</tr>
<tr>
<td>Not Before: Mar 17 16:40:46 2016 GMT</td>
</tr>
<tr>
<td>Not After : Mar 17 16:40:46 2021 GMT</td>
</tr>
<tr>
<td>Subject: C=US,O=Let's Encrypt,CN=Let's Encrypt Authority X3</td>
</tr>
<tr>
<td>Public Key Algorithm: rsaEncryption</td>
</tr>
<tr>
<td>Public Key size: 2048</td>
</tr>
<tr>
<td>Ocsp Response Status: NONE</td>
</tr>
</tbody>
</table>

The HTTPS webserver is now UP with a fake LE signed certificate. Next step is to move to production.
Move to production

After successfully testing with Let's Encrypt-staging, you can get the actual Let's Encrypt certificate.

You need to change Let's Encrypt endpoint from `https://acme-staging-v02.api.letsencrypt.org/directory` to `https://acme-v02.api.letsencrypt.org/directory`

Then, change the name of the ClusterIssuer from `letsencrypt-staging` to `letsencrypt-production`

```yaml
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    # You must replace this email address with your own.
    # Let's Encrypt will use this to contact you about expiring certificates, and issues related to your account.
    email: user@example.com
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      # Secret resource used to store the account's private key.
      name: example-issuer-account-key
    solvers:
      - http01:
        ingress:
          class: citrix
<!--NeedCopy-->`

Note:

Replace `user@example.com` with your email address.

Deploy the file using the following command:

```bash
% kubectl apply -f letsencrypt-prod.yaml
clusterissuer "letsencrypt-prod" created
```
Now, repeat the procedure of modifying the annotation in Ingress or creating a CRD certificate which triggers the generation of new certificate.

Note

Ensure that you delete the old secret so that cert-manager starts a fresh challenge with the production CA.

```
% kubectl delete secret kuard-example-tls
secret "kuard-example-tls" deleted
```

Once the HTTP website is up, you can redirect the traffic from HTTP to HTTPS using the annotation `ingress.citrix.com/insecure-termination: redirect` in the ingress object.

**Troubleshooting**

Since the certificate generation involves multiple components, this section summarizes the troubleshooting techniques that you can use if there was failures.

**Verify the status of certificate generation**

The certificate CRD object defines the life cycle management of generation and renewal of certificates. You can view the status of the certificate using the `kubectl describe` command as follows.

```
% kubectl get certificate
NAME    READY   SECRET   AGE
kuard-example-tls False kuard-example-tls 9s

% kubectl describe certificate kuard-example-tls
Status: Conditions:
    Last Transition Time: 2019-03-05T09:50:29Z
    Message: Certificate does not exist
    Reason: NotFound
    Status: False
    Type: Ready
Events:
```
Also you can view the major certificate events using the `kubectl events` command:

```
1  kubectl get events
2
3  LAST SEEN  TYPE    REASON          KIND          MESSAGE
4 36s    Normal  Started  Challenge    Challenge scheduled for processing
5 36s    Normal  Created  Order        Created Challenge resource "kuard-example-tls-1754626579-0" for domain "acme.cloudpsts.net"
6 38s    Normal  OrderCreated  Certificate  Created Order resource "kuard-example-tls-1754626579"
7 38s    Normal  CreateCertificate  Ingress  Successfully created Certificate "kuard-example-tls"
```

### Analyze logs from cert-manager

If there is a failure, first step is to analyze logs from the cert-manager component. Identify the cert-manager pod using the following command:

```
1  % kubectl get po -n cert-manager
2
3  NAME                       READY STATUS RESTARTS AGE
4  cert-manager-76d48d47bf-5w4vx 1/1  Running 0 23h
5  cert-manager-webhook-67cfb86d56-6qtxr 1/1  Running 0 23h
6  cert-manager-webhook-ca-sync-x4q6f 0/1  Completed 4 23h
```

Here `cert-manager-76d48d47bf-5w4vx` is the main cert-manager pod, and other two pods are cert-manager webhook pods.

Get the logs of the cert-manager using the following command:

```
1  % kubectl logs -f cert-manager-76d48d47bf-5w4vx -n cert-manager
```
Citrix ADC ingress controller

If there is any failure to get the certificate, the ERROR logs give details about the failure.

**Check the Kubernetes secret**

Use the `kubectl describe` command to verify if both certificates and key are populated in Kubernetes secret.

```
% kubectl describe secret kuard-example-tls

Name: kuard-example-tls
Namespace: default
Labels: certmanager.k8s.io/certificate-name=kuard-example-tls
Annotations: certmanager.k8s.io/alt-names: acme.cloudpiston.net
certmanager.k8s.io/common-name: acme.cloudpiston.net
certmanager.k8s.io/issuer-kind: ClusterIssuer
certmanager.k8s.io/issuer-name: letsencrypt-staging

Type: kubernetes.io/tls

Data ====
tls.crt: 3553 bytes
tls.key: 1679 bytes
cacrt: 0 bytes
```

If both `tls.crt` and `tls.key` are populated in the Kubernetes secret, certificate generation is complete. If only `tls.key` is present, certificate generation is incomplete. Analyze the cert-manager logs for more details about the issue.

**Analyze logs from the Citrix ingress controller**

If a Kubernetes secret is generated and complete, but it is not uploaded to the Citrix ADC, you can analyze the logs from the Citrix ingress controller using the following command.

```
% kubectl logs -f cpx-ingress-685c8bc976-zgz8q
```
Deploy an HTTPS web application on Kubernetes with Citrix ingress controller and HashiCorp Vault using cert-manager

February 15, 2022

For ingress resources deployed with the Citrix ingress controller, you can automate TLS certificate provisioning, revocation, and renewal using cert-manager and HashiCorp Vault. This topic provides a sample workflow that uses HashiCorp Vault as a self-signed certificate authority for certificate signing requests from cert-manager.

Specifically, the workflow uses the Vault PKI Secrets Engine to create a certificate authority (CA). This tutorial assumes that you have a Vault server installed and reachable from the Kubernetes cluster. The PKI secrets engine of Vault is suitable for internal applications. For external facing applications that require public trust, see automating TLS certificates using Let's Encrypt CA.

The workflow uses a Vault secret engine and authentication methods. For the full list of Vault features, see the following Vault documentation:

- Vault Secrets Engines
- Vault Authentication Methods

This topic provides you information on how to deploy an HTTPS web application on a Kubernetes cluster, using:

- Citrix ingress controller
- JetStack’s cert-manager to provision TLS certificates from HashiCorp Vault
- HashiCorp Vault

Prerequisites

Ensure that you have:

- The Vault server is installed, unsealed, and is reachable from the Kubernetes cluster. For information on installing the Vault server, see the Vault installation documentation.
- Enabled RBAC on your Kubernetes cluster.
- Deployed Citrix ADC MPX, VPX, or CPX in Tier 1 or Tier 2 deployment model.

In the Tier 1 deployment model, Citrix ADC MPX or VPX is used as an Application Delivery Controller (ADC). The Citrix ingress controller running in the Kubernetes cluster configures the virtual services for the services running on the Kubernetes cluster. Citrix ADC runs the virtual service on the publicly routable IP address and offloads SSL for client traffic with the help of the Let’s Encrypt generated certificate.
In the Tier 2 deployment, a TCP service is configured on the Citrix ADC (VPX/MPX) running outside the Kubernetes cluster to forward the traffic to Citrix ADC CPX instances running in the Kubernetes cluster. Citrix ADC CPX ends the SSL session and load-balances the traffic to actual service pods.

- Deployed Citrix ingress controller. See Deployment Topologies for various deployment scenarios.
- Administrator permissions for all the deployment steps. If you encounter failures due to permissions, make sure that you have the administrator permission.

**Note:**
The following procedure shows steps to configure Vault as a certificate authority with Citrix ADC CPX used as the ingress device. When a Citrix ADC VPX or MPX is used as the ingress device, the steps are the same except the steps to verify the ingress configuration in the Citrix ADC.

**Deploy cert-manager using the manifest file**
Perform the following steps to deploy cert-manager using the supplied YAML manifest file.

1. **Install cert-manager.** For information on installing cert-manager, see the cert-manager documentation.

   ```bash
   kubectl apply -f https://github.com/jetstack/cert-manager/releases/download/vx.x.x/cert-manager.yaml
   ```

   You can also install cert-manager with Helm. For more information, see the cert-manager documentation.

2. **Verify that cert-manager is up and running using the following command.**

   ```bash
   kubectl -n cert-manager get all
   NAME             READY   STATUS    RESTARTS   AGE
   pod/cert-manager-77fd74fb64-d68v7         1/1       Running   0          4m41s
   pod/cert-manager-webhook-67bf86d45-k77jj   1/1       Running   0          4m41s
   ```
<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>deployment.apps/cert-manager</td>
<td>1/1</td>
<td>13d</td>
</tr>
<tr>
<td>deployment.apps/cert-manager-webhook</td>
<td>1/1</td>
<td>13d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESIRED</th>
<th>CURRENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>replicaset.apps/cert-manager-77fd74fb64</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>replicaset.apps/cert-manager-webhook-67bf86d45</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>DURATION</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>job.batch/cert-manager-webhook-ca-sync</td>
<td>22s</td>
<td>13d</td>
</tr>
<tr>
<td>job.batch/cert-manager-webhook-ca-sync-1549756800</td>
<td>21s</td>
<td>10d</td>
</tr>
<tr>
<td>job.batch/cert-manager-webhook-ca-sync-1550361600</td>
<td>19s</td>
<td>3d8h</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>SCHEDULE</th>
<th>SUSPEND</th>
</tr>
</thead>
<tbody>
<tr>
<td>cronjob.batch/cert-manager-webhook-ca-sync</td>
<td>@weekly</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>0 3d8h 13d</td>
<td></td>
</tr>
</tbody>
</table>

### Deploy a sample web application

Perform the following steps to deploy a sample web application.

**Note:**

*Kuard*, a Kubernetes demo application is used for reference in this topic.

1. Create a deployment YAML file (`kuard-deployment.yaml`) for Kuard with the following configuration.

```yaml
apiVersion: apps/v1
kind: Deployment
```
2. Deploy the Kuard deployment file (kuard-deployment.yaml) to your cluster, using the following commands.

```bash
% kubectl create -f kuard-deployment.yaml
deployment.extensions/kuard created
% kubectl get pod -l app=kuard

NAME         READY STATUS    RESTARTS AGE
kuard-6fc4d89bfb-djljt 1/1   Running 0 24s
```

3. Create a service for the deployment. Create a file called service.yaml with the following configuration.

```yaml
apiVersion: v1
type: ClusterIP
categories:
  kuard:
kind: Service
metadata:
  name: kuard
categories:
  kuard:
spec:
  selector:
    app: kuard
  ports:
    - name: kuard
      port: 8080
      targetPort: 8080
      protocol: TCP
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
4. Deploy and verify the service using the following command.

```
% kubectl create -f service.yaml
service/kuard created
% kubectl get svc kuard
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER-IP</th>
<th>EXTERNAL-IP</th>
<th>PORT(S)</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>kuard</td>
<td>ClusterIP</td>
<td>10.103.49.171</td>
<td>&lt;none&gt;</td>
<td>80/TCP</td>
<td>13s</td>
</tr>
</tbody>
</table>

5. Expose this service to the outside world by creating an Ingress that is deployed on Citrix ADC CPX or VPX as Content switching virtual server.

**Note:**
Ensure that you change `kubernetes.io/ingress.class` to your ingress class on which Citrix ingress controller is started.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: citrix
name: kuard
spec:
  rules:
  - host: kuard.example.com
    http:
      paths:
      - backend:
        service:
          name: kuard
          port:
            number: 80
        path: /
        pathType: Prefix
```

**Note:**
Change the value of `spec.rules.host` to the domain that you control. Ensure that a DNS
Citrix ADC ingress controller

entry exists to route the traffic to Citrix ADC CPX or VPX.

6. Deploy the Ingress using the following command.

```
% kubectl apply -f ingress.yml
ingress.extensions/kuard created
```

```
root@ubuntu-vivek-225:/cert-manager# kubectl get ingress
NAME   HOSTS            ADDRESS       PORTS   AGE
kuard  kuard.example.com  80          7s
```

7. Verify if the ingress is configured on Citrix ADC CPX or VPX using the following command.

```
kubectl exec -it cpx-ingress-5b85d7c69d-ngd72 /bin/bash
```

```
root@cpx-ingress-5b85d7c69d-ngd72:/# cli_script.sh 'sh cs vs'
exec: sh cs vs
1) k8s-10.244.1.50:80:http (10.244.1.50:80) - HTTP Type: CONTENT
State: UP
Last state change was at Thu Feb 21 09:02:14 2019
Time since last state change: 0 days, 00:00:41.140
Client Idle Timeout: 180 sec
Down state flush: ENABLED
Disable Primary Vserver On Down: DISABLED
Comment: uid=75
     VBGF07NZXV7SCI4LSDJML2Q5X6FSN6NXQPWGMDYGBW2IM0GQ====
Appflow logging: ENABLED
Port Rewrite: DISABLED
State Update: DISABLED
Default: Content Precedence: RULE
Vserver IP and Port insertion: OFF
L2Conn: OFF Case Sensitivity: ON
Authentication: OFF
401 Based Authentication: OFF
Push: DISABLED Push VServer: none
Push Label Rule: none
Listen Policy: NONE
IcmpResponse: PASSIVE
RHIstate: PASSIVE
Traffic Domain: 0
Done
```

8. Verify if the page is correctly being served when requested using the `curl` command.
Once you have deployed the sample HTTP application, you can proceed to make the application available over HTTPS. Here the Vault server signs the CSR generated by the cert-manager and a server certificate is automatically generated for the application.

In the following procedure, you use the configured Vault as a certificate authority and configure the cert-manager to use the Vault as signing authority for the CSR.

**Configure HashiCorp Vault as Certificate Authority**

In this procedure, you set up an intermediate CA certificate signing request using HashiCorp Vault. This Vault endpoint is used by the cert-manager to sign the certificate for the ingress resources.

**Note:**

Ensure that you have installed the `jq` utility before performing these steps.

**Create a root CA**

For the sample workflow you can generate your own Root Certificate Authority within the Vault. In a production environment, you should use an external Root CA to sign the intermediate CA that Vault uses to generate certificates. If you have a root CA generated elsewhere, skip this step.

**Note:**

`PKI_ROOT` is a path where you mount the root CA, typically it is `pki`. `$({DOMAIN})` in this procedure is `example.com`

```
% export DOMAIN=example.com
% export PKI_ROOT=pki
% vault secrets enable -path="${PKI_ROOT}"
# Set the max TTL for the root CA to 10 years
% vault secrets tune -max-leasettl=87600h "$({"pki"})"
```
Generate an intermediate CA

After creating the root CA, perform the following steps to create an intermediate CSR using the root CA.

1. Enable pki from a different path $PKI\_INT$ from root CA, typically $pki/\_int$. Use the following command:

```bash
% export PKI\_INT=pki\_int
% vault secrets enable -path=${PKI\_INT} pki
```
# Set the max TTL to 3 year

% vault secrets tune -max-lease-ttl=26280h ${PKI_INT }

<!--NeedCopy-->
5. Set the CA and CRL location using the following command.

```sh
vault write "${PKI_INT}"
  /config/urls issuing_certificates="${VAULT_ADDR}"
  /v1/{}
  ${PKI_INT}]
  /ca" crl_distribution_points="${VAULT_ADDR}"
  /v1/{}
  ${PKI_INT}]
  /crl"
  <!--NeedCopy-->"
```

An intermediate CA is set up and can be used to sign certificates for ingress resources.

**Configure a role**

A role is a logical name which maps to policies. An administrator can control the certificate generation through the roles.

Create a role for the intermediate CA that provides a set of policies for issuing or signing the certificates using this CA.

There are many configurations that can be configured when creating roles. For more information, see the Vault role documentation.

For the workflow, create a role `kube-ingress` that allows you to sign certificates of `${DOMAIN}` and its subdomains with a TTL of 90 days.

```sh
# with a Max TTL of 90 days
vault write ${PKI_INT}
  /roles/kube-ingress
    allowed_domains=${DOMAIN}
    allow_subdomains=true
    max_ttl="2160h"
```
Create Approle based authentication

After configuring an intermediate CA to sign the certificates, you need to provide an authentication mechanism for the cert-manager to use the Vault for signing the certificates. Cert-manager supports Approle authentication method which provides a way for the applications to access the Vault defined roles.

An AppRole represents a set of Vault policies and login constraints that must be met to receive a token with those policies. For more information on this authentication method, see the Approle documentation.

Create an Approle

Create an Approle named Kube-role. The secret_id for the cert-manager should not be expired to use this Approle for authentication. Hence, do not set a TTL or set it to 0.

```
% vault auth enable approle
% vault write auth/approle/role/kube-role token_ttl=0
```

Associate a policy with the Approle

Perform the following steps to associate a policy with an Approle.

1. Create a file pki_int.hcl with the following configuration to allow the signing endpoints of the intermediate CA.

```
  path "${PKI_INT} /sign/*" {
    capabilities = ["create","update"]
  }
  <!--NeedCopy-->
```
2. Add the file to a new policy called `kube_allow_sign` using the following command.

```
vault policy write kube-allow-sign pki_int.hcl
```

3. Update this policy to the Approle using the following command.

```
vault write auth/approle/role/kube-role policies=kube-allow-sign
```

The `kube-role` approle allows you to sign the CSR with intermediate CA.

**Generate the role ID and secret ID**

The role ID and secret ID are used by the cert-manager to authenticate with the Vault.

Generate the role ID and secret ID and encode the secret ID with Base64. Perform the following:

```
% vault read auth/approle/role/kube-role/role-id
role_id
db02de05-fa39-4855-059b-67221c5c2f63

% vault write -f auth/approle/role/kube-role/secret-id
secret_id
6a174c20-f6de-a53c-74d2-6018fcceff64
secret_id_accessor
c454f7e5-996e-7230-6074-6ef26b7bcf86

# encode secret_id with base64
% echo 6a174c20-f6de-a53c-74d2-6018fcceff64 | base64
NmExNzRjMjAtZjZkZS1hNTNjLTc0ZDItNjAxOGZjY2VmZjY0Cg==
```

**Configure issuing certificates in Kubernetes**

After you have configured Vault as the intermediate CA, and the Approle authentication method for the cert-manager to access Vault, you need to configure the certificate for the ingress.

**Create a secret with the Approle secret ID**

Perform the following to create a secret with the Approle secret ID.

1. Create a secret file called `secretid.yaml` with the following configuration.
Deploy the secret file (`secretid.yaml`) using the following command.

```
% kubectl create -f secretid.yaml
```

### Deploy the Vault cluster issuer

The cert-manager supports two different CRDs for configuration, an **Issuer**, which is scoped to a single namespace, and a **ClusterIssuer**, which is cluster-wide. For the workflow, you need to use **ClusterIssuer**.

Perform the following steps to deploy the Vault cluster issuer.

1. Create a file called `issuer-vault.yaml` with the following configuration.

```yaml
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: vault-issuer
spec:
  vault:
    path: pki_int/sign/kube-ingress
    server: <vault-server-url>
    # caBundle: <base64 encoded caBundle PEM file>
    auth:
      appRole:
```

Note:
The secret ID `data.secretId` is the base64 encoded secret ID generated in Generate the role id and secret id. If you are using an Issuer resource in the next step, the secret must be in the same namespace as the **Issuer**. For **ClusterIssuer**, the secret must be in the `cert-manager` namespace.
SecretRef is the Kubernetes secret name created in the previous step. Replace roleId with the role_id retrieved from the Vault.
An optional base64 encoded caBundle in the PEM format can be provided to validate the TLS connection to the Vault Server. When caBundle is set it replaces the CA bundle inside the container running the cert-manager. This parameter has no effect if the connection used is in plain HTTP.

2. Deploy the file (issuer-vault.yaml) using the following command.

```bash
% kubectl create -f issuer-vault.yaml
```

3. Using the following command verify if the Vault cluster issuer is successfully authenticated with the Vault.

```
% kubectl describe clusterIssuer vault-issuer | tail -n 7
Conditions:
  Last Transition Time: 2019-02-26T06:18:40Z
  Message: Vault verified
  Reason: VaultVerified
  Status: True
  Type: Ready
  Events: <none>
```

Now, you have successfully setup the cert-manager for Vault as the CA. The next step is securing the ingress by generating the server certificate. There are two different options for securing your ingress. You can proceed with one of the approaches to secure your ingresses.

- Ingress Shim approach
- Manually creating the certificate CRD object for the certificate.

**Ingress-shim approach**

In this approach, you modify the ingress annotation for the cert-manager to automatically generate the certificate for the given host name and store it in the specified secret.
1. Modify the ingress with the **tls** section specifying a host name and secret. Also, specify the cert-manager annotation `cert-manager.io/cluster-issuer` as follows.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    cert-manager.io/cluster-issuer: vault-issuer
    kubernetes.io/ingress.class: citrix
  name: kuard
spec:
rules:
  - host: kuard.example.com
    http:
      paths:
        - backend:
            service:
              name: kuard-service
              port:
                number: 80
      path: /
      pathType: Prefix
tls:
  - hosts:
    - kuard.example.com
    secretName: kuard-example-tls
<!-- NeedCopy-->
```

2. Deploy the modified ingress as follows.

```bash
% kubectl apply -f ingress.yml
ingress.extensions/kuard created
% kubectl get ingress kuard
NAME      HOSTS                  ADDRESS     PORTS   AGE
kuard     kuard.example.com     80, 443    12s
```

This step triggers a certificate object by the cert-manager which creates a certificate signing request (CSR) for the domain `kuard.example.com`. On successful signing of CSR, the certificate is stored in the secret name `kuard-example-tls` specified in the ingress.

1. Verify that the certificate is successfully issued using the following command.
% kubectl describe certificates kuard-example-tls | grep -A5

Events:

Type  Reason     Age  From          Message
------ ------ ------- ---- ------------
Normal CertIssued 48s  cert-manager Certificate issued successfully

Create a certificate CRD object for the certificate

Once the issuer is successfully registered, you need to get the certificate for the ingress domain kuard.example.com.

You need to create a certificate resource with the commonName and dnsNames. For more information, see cert-manager documentation. You can specify multiple dnsNames which are used for the SAN field in the certificate.

To create a "certificate" CRD object for the certificate, perform the following:

1. Create a file called certificate.yaml with the following configuration.

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: kuard-example-tls
  namespace: default
spec:
  secretName: kuard-example-tls
  issuerRef:
    kind: ClusterIssuer
    name: vault-issuer
  commonName: kuard.example.com
  duration: 720h
  renewBefore: 168h
  dnsNames:
    - www.kuard.example.com
```

The certificate has CN=kuard.example.com and SAN=Kuard.example.com, www.kuard.example.com.
**Citrix ADC ingress controller**

`spec.secretName` is the name of the secret where the certificate is stored after the certificate is issued successfully.

2. Deploy the file (`certificate.yaml`) on the Kubernetes cluster using the following command.

   ```bash
   kubectl create -f certificate.yaml
   certificate.certmanager.k8s.io/kuard-example-tls created
   ```

**Verify if the certificate is issued**

You can watch the progress of the certificate as it is issued using the following command:

```
% kubectl describe certificates kuard-example-tls | grep -A5 Events
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Reason</th>
<th>Age</th>
<th>From</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>CertIssued</td>
<td>48s</td>
<td>cert-manager</td>
<td>Certificate issued successfully &gt; <strong>Note</strong> &gt; &gt; You may encounter some errors due to the Vault policies. If you encounter any such errors, return to the Vault and fix it.</td>
</tr>
</tbody>
</table>

After successful signing, a `kubernetes.io/tls` secret is created with the `secretName` specified in the `Certificate` resource.

```
% kubectl get secret kuard-example-tls
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>DATA</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>kuard-exmaple-tls</td>
<td>kubernetes.io/tls</td>
<td>3</td>
<td>4m20s</td>
</tr>
</tbody>
</table>

**Modify the ingress to use the generated secret**

Perform the following steps to modify the ingress to use the generated secret.

1. Edit the original ingress and add a `spec.tls` section specifying the secret `kuard-example-tls` as follows.

   ```yaml
   apiVersion: networking.k8s.io/v1
   kind: Ingress
   metadata:
   annotations:
   kubernetes.io/ingress.class: citrix
   ```
Citrix ADC ingress controller

```yaml
name: kuard
spec:
rules:
  - host: kuard.example.com
    http:
      paths:
        - backend:
            service:
              name: kuard
              port:
                number: 80
                pathType: Prefix
              path: /
            tls:
              - hosts:
                - kuard.example.com
      secretName: kuard-example-tls
```

2. Deploy the ingress using the following command.

```bash
% kubectl apply -f ingress.yml
ingress.extensions/kuard created

% kubectl get ingress kuard
NAME     HOSTS            ADDRESS      PORTS     AGE
kuard    kuard.example.com 80, 443 12s
```

**Verify the Ingress configuration in Citrix ADC**

Once the certificate is successfully generated, Citrix ingress controller uses this certificate for configuring the front-end SSL virtual server. You can verify it with the following steps.

1. Log on to Citrix ADC CPX and verify if the Certificate is bound to the SSL virtual server.

```bash
% kubectl exec -it cpx- ingress-668bf6695f-4fwh8 bash
cli_script.sh 'shsslvs'
exec: shsslvs
1) Vserver Name: k8s-10.244.3.148:443:ssl
   DH: DISABLED
   DH Private-Key Exponent Size Limit: DISABLED Ephemeral RSA: ENABLED Refresh Count: 0
```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Session Reuse:</strong></td>
<td><strong>ENABLED</strong></td>
</tr>
<tr>
<td><strong>Timeout:</strong></td>
<td><strong>120 seconds</strong></td>
</tr>
<tr>
<td><strong>Cipher Redirect:</strong></td>
<td><strong>DISABLED</strong></td>
</tr>
<tr>
<td><strong>SSLv2 Redirect:</strong></td>
<td><strong>DISABLED</strong></td>
</tr>
<tr>
<td><strong>ClearText Port:</strong></td>
<td><strong>0</strong></td>
</tr>
<tr>
<td><strong>Client Auth:</strong></td>
<td><strong>DISABLED</strong></td>
</tr>
<tr>
<td><strong>SSL Redirect:</strong></td>
<td><strong>DISABLED</strong></td>
</tr>
<tr>
<td><strong>Non FIPS Ciphers:</strong></td>
<td><strong>DISABLED</strong></td>
</tr>
<tr>
<td><strong>SNI:</strong></td>
<td><strong>ENABLED</strong></td>
</tr>
<tr>
<td><strong>OCSP Stapling:</strong></td>
<td><strong>DISABLED</strong></td>
</tr>
<tr>
<td><strong>HSTS:</strong></td>
<td><strong>DISABLED</strong></td>
</tr>
<tr>
<td><strong>HSTS IncludeSubDomains:</strong></td>
<td><strong>NO</strong></td>
</tr>
<tr>
<td><strong>HSTS Max-Age:</strong></td>
<td><strong>0</strong></td>
</tr>
<tr>
<td><strong>SSLv2:</strong></td>
<td><strong>DISABLED</strong></td>
</tr>
<tr>
<td><strong>SSLv3:</strong></td>
<td><strong>ENABLED</strong></td>
</tr>
<tr>
<td><strong>TLSv1.0:</strong></td>
<td><strong>ENABLED</strong></td>
</tr>
<tr>
<td><strong>TLSv1.1:</strong></td>
<td><strong>DISABLED</strong></td>
</tr>
<tr>
<td><strong>TLSv1.2:</strong></td>
<td><strong>ENABLED</strong></td>
</tr>
<tr>
<td><strong>TLSv1.3:</strong></td>
<td><strong>DISABLED</strong></td>
</tr>
<tr>
<td><strong>Push Encryption Trigger:</strong></td>
<td><strong>Always</strong></td>
</tr>
<tr>
<td><strong>Send Close-Notify:</strong></td>
<td><strong>YES</strong></td>
</tr>
<tr>
<td><strong>Strict Sig-Digest Check:</strong></td>
<td><strong>DISABLED</strong></td>
</tr>
<tr>
<td><strong>Zero RTT Early Data:</strong></td>
<td><strong>DISABLED</strong></td>
</tr>
<tr>
<td><strong>DHE Key Exchange With PSK:</strong></td>
<td><strong>NO</strong></td>
</tr>
<tr>
<td><strong>Tickets Per Authentication Context:</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td><strong>Done</strong></td>
<td></td>
</tr>
</tbody>
</table>

```
root@cpx-ingress-668bf6695f-4fwh8:/# cli_script.sh 'shsslvs k8s -10.244.3.148:443:ssl'
```

```
exec: shsslvs k8s-10.244.3.148:443:ssl
```

```
Advanced SSL configuration for VServer k8s-10.244.3.148:443:ssl:
DH: DISABLED
DH Private-Key Exponent Size Limit: DISABLED Ephemeral RSA: ENABLED Refresh Count: 0
Session Reuse: **ENABLED** Timeout: **120 seconds**
Cipher Redirect: **DISABLED**
SSLv2 Redirect: **DISABLED**
ClearText Port: **0**
Client Auth: **DISABLED**
SSL Redirect: **DISABLED**
Non FIPS Ciphers: **DISABLED**
SNI: **ENABLED**
OCSP Stapling: **DISABLED**
HSTS: **DISABLED**
HSTS IncludeSubDomains: **NO**
HSTS Max-Age: **0**
SSLv2: **DISABLED** SSLv3: **ENABLED** TLSv1.0: **ENABLED** TLSv1.1: **ENABLED** TLSv1.2: **ENABLED** TLSv1.3: **DISABLED**
Push Encryption Trigger: **Always**
The HTTPS webserver is up with the vault signed certificate. Cert-manager automatically renews the certificate as specified in the RenewBefore parameter in the certificate, before expiry of the certificate.

**Note:**

The Vault signing of the certificate fails if the expiry of a certificate is beyond the expiry of the root CA or intermediate CA. You should ensure that the CA certificates are renewed in advance before the expiry.

2. Verify that the application is accessible using the HTTPS protocol.
Enable Citrix ADC certificate validation in the Citrix ingress controller

January 19, 2022

The Citrix ingress controller provides an option to ensure secure communication between the Citrix ingress controller and Citrix ADC by using the HTTPS protocol. You can achieve this by using pre-loaded certificates in the Citrix ADC. As an extra measure to avoid any possible man-in-the-middle (MITM) attack, the Citrix ingress controller also allows you to validate the SSL server certificate provided by the Citrix ADC.

To enable certificate signature and common name validation of the ADC server certificate by the Citrix ingress controller, security administrators can optionally install signed (or self-signed) certificates in the Citrix ADC and configure the Citrix ingress controller with the corresponding CA certificate bundle. Once the validation is enabled and CA certificate bundles are configured, the Citrix ingress controller starts validating the certificate (including certificate name validation). If the validation fails, the Citrix ingress controller logs the same and none of the configurations are used on an unsecure channel.

This validation is turned off by default and an administrator can chose to enable the validation in the Citrix ingress controller as follows.

Prerequisites

- For enabling certificate validation, you must configure a Citrix ADC with proper SSL server certificates (with proper server name or IP address in certificate subject). For more information, see Citrix ADC documentation.
- The CA certificate for the installed server certificate-key pair is used to configure the Citrix ingress controller to enable validation of these certificates.

Configure the Citrix ingress controller for certificate validation

To make a CA certificate available for configuration, you need to configure the CA certificate as a Kubernetes secret so that the Citrix ingress controller can access it on a mounted storage volume.
To generate a Kubernetes secret for an existing certificate, use the following `kubectl` command:

```bash
$ kubectl create secret generic ciccacert --from-file=path/myCA.pem
    - namespace default
secret "ciccacert" created
```

Alternatively, you can also generate the Kubernetes secret using the following YAML definition:

```yaml
apiVersion: v1
custom: Secret
metadata:
  name: ciccacert
data:
  myCA.pem: <base64 encoded cert>
```

The following is a sample YAML file with the Citrix ingress controller configuration for enabling certificate validation.

```yaml
kind: Pod
metadata:
  name: cic
labels:
  app: cic
spec:
  serviceAccountName: cpx
  # Make secret available as a volume
  volumes:
  - name: certs
    secret:
      secretName: ciccacert
  containers:
  - name: cic
    image: "xxxx"
    imagePullPolicy: Always
    args: []
    # Mounting certs in a volume path
    volumeMounts:
    - name: certs
      mountPath: <Path to mount the certificate>
      readOnly: true
```
As specified in the example YAML file, following are the specific changes required for enabling certificate validation in the Citrix ingress controller.

**Configure Kubernetes secret as a volume**

- Configure a volume section declared with `secret` as the source. Here, `secretName` should match the Kubernetes secret name created for the CA certificate.

**Configure a volume mount location for the CA certificate**

- Configure a `volumeMounts` section with the same name as that of `secretName` in the volume section
- Declare a `mountPath` directory to mount the CA certificate
- Set the volume as `ReadOnly`

**Configure secure communication**

- Set the environment variable `NS_PROTOCOL` as HTTPS
- Set the environment variable `NS_PORT` as ADC HTTPS port
Enable and configure CA validation and certificate path

- Set the environment variable NS_VALIDATE_CERT to yes (no for disabling)
- Set the environment variable NS_CACERT_PATH as the mount path (volumeMounts->mountPath)/ PEM file name (used while creating the secret).

Disable API server certificate verification

January 25, 2022

While communicating with the API server from Citrix ingress controller or multicluster ingress, you have the option to disable the API server certificate verification on Citrix ingress controller.

Disable API server certificate verification on Citrix ingress controller or Multi-cluster ingress

When you deploy Citrix ingress controller using YAML, you can disable the API server certificate verification by providing the following argument in the Citrix ingress controller deployment YAML file.

```
args:
- --disable-apiserver-cert-verify
  true
```

When you deploy Citrix ingress controller using Helm charts, the parameter disableAPIServerCertVerify can be mentioned as True in the Helm values file as follows:

```
disableAPIServerCertVerify: True
```

View metrics of Citrix ADCs using Prometheus and Grafana

February 3, 2022

You can use the Citrix ADC metrics exporter and Prometheus-Operator to monitor Citrix ADC VPX or CPX ingress devices and Citrix ADC CPX (east-west) devices.
Citrix ADC ingress controller

Citrix ADC metrics exporter

Citrix ADC metrics exporter is a simple server that collects Citrix ADC stats and exports them to Prometheus using HTTP. You can then add Prometheus as a data source to Grafana and graphically view the Citrix ADC stats. For more information see, Citrix ADC metrics exporter.

Launch prometheus operator

The Prometheus Operator has an expansive method of monitoring services on Kubernetes. To get started, this topic uses kube-prometheus and its manifest files. The manifest files help you to deploy a basic working model. Deploy the Prometheus Operator in your Kubernetes environment using the following commands:

```
1 git clone https://github.com/coreos/kube-prometheus.git
2 kubectl create -f kube-prometheus/manifests/setup/
3 kubectl create -f kube-prometheus/manifests/
```

Once you deploy Prometheus-Operator, several pods and services are deployed. From the deployed pods, the prometheus-k8s-xx pods are for metrics aggregation and timestamping, and the grafana pods are for visualization. If you view all the container images running in the cluster, you can see the following output:

```
$ kubectl get pods -n monitoring
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>alertmanager-main-0</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>2h</td>
</tr>
<tr>
<td>alertmanager-main-1</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>2h</td>
</tr>
<tr>
<td>alertmanager-main-2</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>2h</td>
</tr>
<tr>
<td>grafana-5b68464b84-5fvxq</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>2h</td>
</tr>
<tr>
<td>kube-state-metrics-6588b6b755-d6ftg</td>
<td>4/4</td>
<td>Running</td>
<td>0</td>
<td>2h</td>
</tr>
<tr>
<td>node-exporter-4hbcph</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>2h</td>
</tr>
<tr>
<td>node-exporter-kn9dg</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>2h</td>
</tr>
<tr>
<td>node-exporter-tpxhp</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>2h</td>
</tr>
</tbody>
</table>
Note:
The files in the manifests folder are interdependent and hence the order in which they are created is important. In certain scenarios the manifest files might be created out of order and this leads to an error messages from Kubernetes.
To resolve this scenario, re-execute the `kubectl create -f kube-prometheus/manifests` command. Any YAML files that were not created the first time due to unmet dependencies, are created now.

It is recommended to expose the Prometheus and Grafana pods through NodePorts. To do so, you need to modify the `prometheus-service.yaml` and `grafana-service.yaml` files as follows:

Modify Prometheus service

```yaml
apiVersion: v1
type: Service
spec:
  ports:
  - name: web
    port: 9090
    targetPort: web
  selector:
    app: prometheus
    prometheus: k8s
<!-- NeedCopy-->
```

After you modify the `prometheus-service.yaml` file, apply the changes to the Kubernetes cluster using the following command:

```bash
kubectl apply -f prometheus-service.yaml
```
Citrix ADC ingress controller

Modify Grafana service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: grafana
  namespace: monitoring
spec:
  type: NodePort
  ports:
  - name: http
    port: 3000
    targetPort: http
  selector:
    app: grafana
<!-- NeedCopy -->
```

After you modify the `grafana-service.yaml` file, apply the changes to the Kubernetes cluster using the following command:

```
kubectl apply -f grafana-service.yaml
```

Configure Citrix ADC metrics exporter

This topic describes how to integrate the Citrix ADC metrics exporter with Citrix ADC VPX or CPX ingress or Citrix ADC CPX (east-west) devices.

Configure Citrix ADC metrics exporter for Citrix ADC VPX Ingress device

To monitor an ingress Citrix ADC VPX device, the Citrix ADC metrics exporter is run as a pod within the Kubernetes cluster. The IP address of the Citrix ADC VPX ingress device is provided as an argument to the Citrix ADC metrics exporter. To provide the login credentials to access ADC, create a secret and mount the volume at mountpath “/mnt/nslogin”.

```
kubectl create secret generic nslogin --from-literal=username=<citrix-adc-user> --from-literal=password=<citrix-adc-password> -n <namespace>
<!-- NeedCopy -->
```
The following is a sample YAML file to deploy the exporter:

```
apiVersion: v1
kind: Pod
metadata:
  name: exporter-vpx-ingress
  labels:
    app: exporter-vpx-ingress
spec:
  containers:
    - name: exporter
      image: "quay.io/citrix/citrix-adc-metrics-exporter:1.4.8"
      imagePullPolicy: IfNotPresent
      args:
        - "--target-nsip=<IP_of_VPX>"
        - "--port=8888"
      volumeMounts:
        - name: nslogin
          mountPath: "/mnt/nslogin"
      readOnly: true
      securityContext:
        readOnlyRootFilesystem: true
  volumes:
    - name: nslogin
      secret:
        secretName: nslogin
      ---
kind: Service
apiVersion: v1
metadata:
  name: exporter-vpx-ingress
  labels:
    service-type: citrix-adc-monitor
spec:
  selector:
    name: exporter-vpx-ingress
  ports:
    - name: exporter-port
      port: 8888
      targetPort: 8888
```

The IP address and the port of the Citrix ADC VPX device needs to be provided in the `--target-nsip` option.
Configure Citrix ADC metrics exporter for Citrix ADC CPX Ingress device

To monitor a Citrix ADC CPX ingress device, the Citrix ADC metrics exporter is added as a sidecar to the Citrix ADC CPX. The following is a sample YAML file of a Citrix ADC CPX ingress device with the exporter as a side car:

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: cpx-ingress
  name: cpx-ingress
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cpx-ingress
  template:
    metadata:
      annotations:
        NETSCALER_AS_APP: "True"
    labels:
      app: cpx-ingress
  spec:
    containers:
    - env:
      - name: EULA
        value: "YES"
      - name: NS_PROTOCOL
        value: HTTP
      - name: NS_PORT
        value: "9080"
    # Define the NITRO port here
    image: quay.io/citrix/citrix-k8s-cpx-ingress:13.0-52.24
    imagePullPolicy: IfNotPresent
  name: cpx-ingress
  ports:
  - containerPort: 80
    name: http
    protocol: TCP
```
Here, the exporter uses the local IP address (192.0.0.2) to fetch metrics from the Citrix ADC CPX.
Citrix ADC ingress controller

Configure Citrix ADC metrics exporter for Citrix ADC CPX (east-west) device

To monitor a Citrix ADC CPX (east-west) device, the Citrix ADC metrics exporter is added as a sidecar to the Citrix ADCCPX. The following is a sample YAML file of a Citrix ADC CPX (east-west) device with the exporter as a side car:

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
    annotations:
      deprecated.daemonset.template.generation: "0"
    labels:
      app: cpx-ew
      name: cpx-ew
spec:
    selector:
      matchLabels:
        app: cpx-ew
    template:
      metadata:
        annotations:
          NETSCALER_AS_APP: "True"
        labels:
          app: cpx-ew
          name: cpx-ew
    spec:
      containers:
        - name: EULA
          value: "yes"
        - name: NS_NETMODE
          value: HOST
        # name: "kubernetes_url"
        # value: "https://10..xx.xx:6443"
        image: quay.io/citrix/citrix-k8s-cpx-ingress:13.0-52.24
        imagePullPolicy: IfNotPresent
        name: cpx
        securityContext:
          privileged: true
        # Add exporter as a sidecar
        - args:
          - --target-nsip=192.168.0.2
          - --port=8888
          - --secure=no
```
Here, the exporter uses the local IP (192.168.0.2) to fetch metrics from the Citrix ADC CPX (east-west) device.

**ServiceMonitors to detect Citrix ADC**

The Citrix ADC metrics exporter helps collect data from the Citrix ADC VPX or CPX ingress and Citrix ADC CPX (east-west) devices. The Prometheus Operator needs to detect these exporters so that the metrics can be timestamped, stored, and exposed for visualization on Grafana. The Prometheus Operator uses the concept of ServiceMonitors to detect pods that belong to a service, using the labels attached to that service.

The following example YAML file detects all the exporter services (given in the sample YAML files) which have the label `service-type: citrix-adc-monitor` associated with them.
The `ServiceMonitor` directs Prometheus to detect Exporters in the **default** and **monitoring** namespaces only. To detect Exporters from other namespaces add the names of those namespaces under the `namespaceSelector` field.

**Note:**

If the Exporter that needs to be monitored exists in a namespace other than the **default** or **monitoring** namespace, then additional RBAC privileges must be provided to Prometheus to access those namespaces. The following is sample YAML (`prometheus-clusterRole.yaml`) file the provides Prometheus full access to resources across the namespaces:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus-k8s
rules:
- apiGroups: 
  - ""
  resources: 
  - nodes/metrics
  - namespaces
  - services
  - endpoints
  - pods
  verbs: ["*"]
- nonResourceURLs: 
  - /metrics
```
To provide additional privileges Prometheus, deploy the sample YAML using the following command:

```bash
kubectl apply -f prometheus-clusterRole.yaml
```

**View the metrics in grafana**

The Citrix ADC instances that are detected for monitoring appears in the **Targets** page of the prometheus container. You can access the **Targets** page using the following URL: `http://<k8s_cluster_ip>:<prometheus_nodeport>/targets`

To view the metrics graphically:

1. Log into grafana using `http://<k8s_cluster_ip>:<grafana_nodeport>` with default credentials `admin:admin`
2. On the left panel, select + and click **import** to import the sample grafana dashboard.
A dashboard containing the graphs similar to the following appears:

You can further enhance the dashboard using Grafana’s [documentation](#) or [demo videos](#).

**Analytics and observability**

February 3, 2022
Analytics from Citrix ADC instances provides you deep-level insights about application performance which helps you to quickly identify issues and take any necessary action.

Enabling analytics using annotations in the Citrix ingress controller YAML file

You can enable analytics using the analytics profile which is defined as a smart annotation in Ingress or service of type LoadBalancer configuration. You can define the specific parameters you need to monitor by specifying them in the Ingress or service configuration of the application.

The following is a sample Ingress annotation with analytics profile for HTTP records:

```yaml
ingress.citrix.com/analyticsprofile: '{ "webinsight": { "httpurl":"ENABLED", "httpuseragent":"ENABLED", "httpHost":"ENABLED", "httpMethod":"ENABLED", "httpContentType":"ENABLED" } }'
```

The following is a sample Ingress configuration with the analytics profile for a web application.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    ingress.citrix.com/analyticsprofile: '{
      "webinsight": {
        "httpurl":"ENABLED", "httpuseragent":"ENABLED", 
        "httpHost":"ENABLED", "httpmethod":"ENABLED", "httpcontenttype":"ENABLED"
      }
    }
  ingress.citrix.com/insecure-termination: allow
name: webserver-ingress
spec:
  rules:
  - http:
    paths:
      - backend:
        service:
          name: webserver
        port:
          number: 80
        path: /
        pathType: Prefix
tls:
  - secretName: name
<!-- NeedCopy -->
```
The following is a service annotation:

tenet.scaler.com/analyticsprofile: '{ "80-tcp": { "webinsight": { "httpurl": "ENABLED", "httpuseragent": "ENABLED" } } }'

The following is a sample service configuration with the analytics profile which exposes an Apache web application.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: apache
  annotations:
    service.citrix.com/csvserver: '{
      "l2conn":"on"
    }'
    service.citrix.com/lbvserver: '{
      "80-tcp":{
        "lbmethod":"SRCIPDESTIPHASH"
      }
    }'
    service.citrix.com/servicegroup: '{
      "80-tcp":{
        "usip":"yes"
      }
    }'
    service.citrix.com/monitor: '{
      "80-tcp":{
        "type":"http"
      }
    }'
    service.citrix.com/frontend-ip: "192.0.2.16"
    service.citrix.com/analyticsprofile: '{
      "80-tcp":{
        "webinsight": {
          "httpurl": "ENABLED", "httpuseragent": "ENABLED"
        }
      }
    }'
    NETSCALER_VPORT: "80"
  labels:
    name: apache
spec:
  externalTrafficPolicy: Local
```
For information about annotations, see the annotation documentation.

Analytics using Citrix ADM

Citrix ADM provides a comprehensive observability solution including analytics on various events happening in the system and a service graph for monitoring services in an easy to use user interface.

Citrix ADM analytics provide an easy and scalable way to get various insights out of the data from Citrix ADC instances to describe, predict, and improve the application performance. You can use one or more analytics features simultaneously on Citrix ADM. For more information on the service graph, see the service graph documentation.

To use the ADM analytics or service graph:

• You must install an ADM agent and ensure the communication between Citrix ADM and Kubernetes cluster or managed instances in your data center or cloud. It makes Citrix ADC instances discoverable by Citrix ADM.
• Ensure that an appropriate license is available and auto licensing is enabled on ADM.

Analytics with open source tools

Citrix ADC can be integrated with various open source tools for observability using Citrix observability exporter. Citrix observability exporter is a container which collects metrics and transactions from Citrix ADCs and transforms them to suitable formats (such as JSON, AVRO) for supported endpoints. You can export the collected data to the desired endpoint. By analyzing the data, you can get valuable insights at a microservice level for applications proxied by Citrix ADCs.
For more information on Citrix ADC observability exporter, see the Citrix ADC observability exporter documentation.
You can use Citrix Observability Exporter to export metrics and transactions from Citrix ADC CPX, MPX, or VPX and analyze the exported data to get meaningful insights. The Citrix Observability Exporter support is enabled with in the Citrix ingress controller configuration. You can now enable the Citrix Observability Exporter configuration with in the Citrix ingress controller using a ConfigMap.

**Supported environment variables for analytics configuration using ConfigMap**

You can configure the following parameters under `NS_ANALYTICS_CONFIG` using a ConfigMap:

- **distributed_tracing**: This variable enables or disables OpenTracing in Citrix ADC and has the following attributes:
  - **enable**: Set this value to `true` to enable OpenTracing. The default value is `false`.
  - **samplingrate**: Specifies the OpenTracing sampling rate in percentage. The default value is 100.

- **endpoint**: Specifies the IP address or DNS address of the analytics server.
  - **server**: Set this value as the IP address or DNS address of the server.
  - **service**: Specifies the IP address or service name of the Citrix ADC observability exporter service depending on whether the service is running on a virtual machine or as a Kubernetes service.
    - If the Citrix ADC observability exporter instance is running on a virtual machine this parameter specifies the IP address. If the Citrix ADC observability exporter instance is running as a service in the Kubernetes cluster, this parameter specifies the instance as namespace/service name.

- **timeseries**: Enables exporting time series data from Citrix ADC. You can specify the following attributes for time series configuration.
  - **port**: Specifies the port number of time series end point of the analytics server. The default value is 5563.
  - **metrics**: Enables exporting metrics from Citrix ADC.
    - **enable**: Set this value to `true` to enable sending metrics. The default value is `false`.
  - **mode**: Specifies the mode of metric endpoint. The default value is `avro`.
  - **auditlogs**: Enables exporting audit log data from Citrix ADC.
    - **enable**: Set this value to `true` to enable audit log data. The default value is `false`.
  - **events**: Enables exporting events from the Citrix ADC.
    - **enable**: Set this value to `true` to enable exporting events. The default value is `false`.
transactions: Enables exporting transactions from Citrix ADC.
- enable: Set this value to true to enable sending transactions. The default value is false.
- port: Specifies the port number of transactional endpoint of analytics server. The default value is 5557.

The following configurations cannot be changed while the Citrix ingress controller is running and you need to reboot the Citrix ingress controller to apply these settings.

- server configuration (endpoint)
- port configuration (time series)
- port configuration (transactions)

You can change other ConfigMap settings at runtime while the Citrix ingress controller is running.

Note:
When the user specifies value for a service as namespace/service name, Citrix ingress controller derives the endpoint associated to that service and dynamically bind them to the transactional service group in Citrix tier-1 ADC. If a user specifies the value for a service as IP address, the IP address is directly bound to the transactional service group. Citrix ingress controller is enhanced to create default web or TCP based analytics profiles and bind them to the logging virtual server. The default analytics profiles are bound to all load balancing virtual servers of applications if the Citrix ADC observability exporter is enabled in the cluster. If the user wants to change the analytics profile, they can use the analyticsprofile annotation.

The attributes of NS_ANALYTICS_CONFIG should follow a well-defined schema. If any value provided does not confirm with the schema, then the entire configuration is rejected. For reference, see the schema file ns_analytics_config_schema.yaml.

Creating a ConfigMap for analytics configuration

This topic provides information on how to create a ConfigMap for analytics configuration.

Create a YAML file cic-configmap.yaml with the required key-value pairs in the ConfigMap.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cic-configmap
  labels:
    app: citrix-ingress-controller
data:
  LOGLEVEL: 'info'
  NS_PROTOCOL: 'http'
```
For more information on how to configure ConfigMap support on the Citrix ingress controller, see configuring ConfigMap support for the Citrix ingress controller.

**Schema for NS_ANALYTICS_CONFIG**

Following is the schema for `NS_ANALYTICS_CONFIG`. The attributes should confirm with this schema.
required: no
type: map
mapping:
  enable:
    required: yes
type: str
enum:
  - 'true'
  - 'false'
samplingrate:
  required: no
type: int
range:
  max: 100
  min: 0
timeseries:
  required: no
type: map
mapping:
  port:
    required: no
type: int
metrics:
  required: no
type: map
mapping:
  enable:
    required: yes
type: str
enum:
  - 'true'
  - 'false'
mode:
  required: yes
type: str
enum:
  - prometheus
  - avro
  - influx
auditlogs:
  required: no
type: map
mapping:
  enable:
    required: yes
```
    type: str
    enum:
        - 'true'
        - 'false'
    events:
        required: no
        type: map
        mapping:
            enable:
                required: yes
                type: str
                enum:
                    - 'true'
                    - 'false'
    transactions:
        required: no
        type: map
        mapping:
            enable:
                required: yes
                type: str
                enum:
                    - 'true'
                    - 'false'
    port:
        required: no
        type: int
```

**Troubleshooting**

January 19, 2022

The following table describes some of the common issues and workarounds.
<table>
<thead>
<tr>
<th>Problem</th>
<th>Log</th>
<th>Workaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>Citrix ADC instance is not reachable</td>
<td><strong>2019-01-10 05:05:27,250 - ERROR - [nitrointerface.py:login_logout:94]</strong> <em>(MainThread) Exception: HTTPConnectionPool(host='10.106.76.200', port=80): Max retries exceeded with url: /nitro/v1/config/login (Caused by NewConnectionError('&lt;urllib3.connection.HTTPConnection object at 0x7f4d45bd63d0&gt;: Failed to establish a new connection: [Errno 113] No route to host'))</em>*</td>
<td>Ensure that the Citrix ADC is up and running, and you can ping the NSIP address.</td>
</tr>
<tr>
<td>Wrong user name password</td>
<td><strong>2019-01-10 05:03:05,958 - ERROR - [nitrointerface.py:login_logout:90]</strong> <em>(MainThread) Nitro Exception::login_logout::errorcode=354,message=Invalid username or password</em>*</td>
<td></td>
</tr>
<tr>
<td>SNIP is not enabled with management access</td>
<td><strong>2019-01-10 05:43:03,418 - ERROR - [nitrointerface.py:login_logout:94]</strong> <em>(MainThread) Exception: HTTPConnectionPool(host='10.106.76.242', port=80): Max retries exceeded with url: /nitro/v1/config/login (Caused by NewConnectionError('&quot;&lt;urllib3.connection.HTTPConnection object at 0x7f302a8cfad0&gt;: Failed to establish a new connection: [Errno 110] Connection timed out'))</em>*</td>
<td>Ensure that you have enabled the management access in Citrix ADC (for Citrix ADC VPX high availability) and set the IP address, <strong>NSIP</strong>, with management access enabled.</td>
</tr>
<tr>
<td>Problem</td>
<td>Log</td>
<td>Workaround</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Error while parsing annotations</td>
<td>2019-01-10 05:16:10,611 - ERROR - [kubernetes.py: set_annotations_to_csapp:1040] (MainThread) set_annotations_to_csapp: Error message=No JSON object could be decodedInvalid Annotation $service_weights please fix and apply ${“frontend”:, “catalog”:95}</td>
<td>Verify if the correct port is specified for NITRO access. By default, Citrix ingress controller uses the port 80 for communication.</td>
</tr>
<tr>
<td>Wrong port for NITRO access</td>
<td>2019-01-10 05:18:53,964 - ERROR - [nitrointerface.py: login_logout:94] (MainThread) Exception: HTTPConnectionPool(host='10.106.76.242', port=34438): Max retries exceeded with url: /nitro/v1/config/login (Caused by NewConnectionError('&lt;urllib3.connection.HTTP object at 0x7fc592cb8b10&gt;: Failed to establish a new connection: [Errno 111] Connection refused.')</td>
<td>Verify that the ingress file belongs to the ingress class at Citrix ingress controller monitors. See the following log for information about the ingress classes listened by Citrix ingress controller:</td>
</tr>
<tr>
<td>Ingress class is wrong</td>
<td>2019-01-10 05:27:27,149 - INFO - [kubernetes.py: get_all_ingresses:1329] (MainThread) Unsupported Ingress class for ingress object web-ingress.default</td>
<td></td>
</tr>
</tbody>
</table>
## Problem Log Workaround

<table>
<thead>
<tr>
<th>Problem</th>
<th>Log</th>
<th>Workaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kubernetes API is not reachable</td>
<td>2019-01-10 05:32:09,729 - ERROR - [kubernetes.py:<strong>get</strong>:222]</td>
<td>Check if the kubernetes_url is correct. Use the command, <code>kubectl cluster-info</code> to get the URL information. Ensure that the Kubernetes main node is running at <a href="https://kubernetes_master_address:6443">https://kubernetes_master_address:6443</a> and the Kubernetes API server pod is up and running.</td>
</tr>
<tr>
<td>Incorrect service port specified in the YAML file</td>
<td>2019-01-10 05:32:09,729 - ERROR - [kubernetes.py:<strong>get</strong>:222]</td>
<td>Provide the correct port details in the ingress YAML file and reapply to solve the issue.</td>
</tr>
<tr>
<td>Load balancing virtual server and service group are created but they are down</td>
<td>2019-01-10 05:32:09,729 - ERROR - [kubernetes.py:<strong>get</strong>:222]</td>
<td>Check for the service name and port used in the YAML file. For Citrix ADC VPX, ensure that <code>--feature-node-watch</code> is set to <code>true</code>, when bringing up the Citrix ingress controller.</td>
</tr>
</tbody>
</table>
## Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>Problem</th>
<th>Log</th>
<th>Workaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS virtual server is not getting created for Citrix ADC VPX.</td>
<td></td>
<td>Use the annotation, <code>ingress.citrix.com/ frontend-ip</code>, in the ingress YAML file for Citrix ADC VPX.</td>
</tr>
<tr>
<td>Incorrect secret provided in the TLS section in the ingress YAML file</td>
<td>2019-01-10 09:30:50,673 - INFO - [kubernetes.py: get:231] (MainThread) Resource not found: /secrets/default-secret12345 namespace default</td>
<td>Correct the values in the YAML file and reapply to solve the issue.</td>
</tr>
<tr>
<td></td>
<td>2019-01-10 09:30:50,673 - INFO - [kubernetes.py: get_secret:1712] (MainThread) Failed to get secret for the app default-secret12345.default</td>
<td></td>
</tr>
<tr>
<td>The <code>feature-node-watch</code> argument is specified, but static routes are not added in the Citrix ADC VPX</td>
<td>ERROR - [nitrointerface.py: add_ns_route:4495] (MainThread) Nitro Exception::add_ns_route::errorcode::gateway is not directly reachable</td>
<td>This error occurs when <code>feature-node-watch</code> is enabled and the Citrix ADC VPX and Kubernetes cluster are not in the same network. You must remove the <code>--feature-node-watch</code> argument from the Citrix ingress controller YAML file. Static routes do not work when the Citrix ADC VPX and Kubernetes cluster are in different network. Use Citrix node controller to create tunnels between Citrix ADC VPX and cluster nodes.</td>
</tr>
</tbody>
</table>

© 1999–2022 Citrix Systems, Inc. All rights reserved. 453
<table>
<thead>
<tr>
<th>Problem</th>
<th>Log</th>
<th>Workaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRD status not updated</td>
<td>ERROR - [crdinfrautils.py:update_crd_status:42]</td>
<td>Verify that permission to push CRD status is provided in the RBAC. The permission should be similar to the following:</td>
</tr>
<tr>
<td></td>
<td>(MainThread) Exception during CRD status update</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for negrwaddmuloccmod: 403 Client Error: Forbidden for url:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><a href="https://10.96.0.1:443//apis/citrix.com/v1/namespaces/default/">https://10.96.0.1:443//apis/citrix.com/v1/namespaces/default/</a></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rewritepolicies/negrwaddmuloccmod/status</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Citrix ingress controller event not updated</td>
<td>ERROR - [clienthelper.py:post:94]</td>
<td>Verify that the permission to update the Citrix ingress controller pod events is provided in the RBAC.</td>
</tr>
<tr>
<td></td>
<td>(MainThread) Request /events to api server is forbidden</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- apiGroups: ["citrix.com"]
- resources:
  - ["rewritepolicies/status", "canarycrds/status", "authpolicies/status", "ratelimits/status", "listeners/status", "httproutes/status", "wafs/status"]

- apiGroups: [""]
- resources:
  - ["events"] verbs: ["create"]
<table>
<thead>
<tr>
<th>Problem</th>
<th>Log</th>
<th>Workaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rewrite-responder policy not added</td>
<td>ERROR - [config_dispatcher.py:__dispatch_config_pack:324] (Dispatcher) Status: 104, ErrorCode: 3081, Reason: Nitro Exception: Expression syntax error [D(10, 20) ^.RE_SELECT(, Offset 15] &lt;</td>
<td>Such errors are due to incorrect expressions in rewrite-responder CRDs. Fix the expression and reapply the CRD.</td>
</tr>
<tr>
<td>Application of a CRD failed. The Citrix ingress controller converts a CRD into a set of configurations to configure the Citrix ADC to the desired state as per the specified CRD. If the configuration fails, then the CRD instance may not get applied on the Citrix ADC.</td>
<td>2020-07-13 08:49:07,620 - ERROR - [config_dispatcher.py:__dispatch_config_pack:256] (Dispatcher) Failed to execute config ADD_sslprofile_k8s_crd_k8service_kuard-service_default_80_tcp_backend {name:k8s_crd_k8service_kuard-service_default_80_tcp_backend sslprofiletype:BackEnd tls12:enabled } from ConfigPack ‘default.k8service.kuard-service.add_spec’</td>
<td>Log shows that the NITRO command has failed. The Citrix ADC ns . log and search for the command using the command name in Citrix ADC as well. Check the Citrix ADC command which failed during the application of CRD. Try to delete the CRD and add it again. If you see the issue again, report it on the cloud native slack channel.</td>
</tr>
</tbody>
</table>
Problem Log Workaround

2020-07-13 08:49:07,620 - ERROR - [config_dispatcher.py:__dispatch_config_pack:257] (Dispatcher) Status: 104, ErrorCode: 1074, Reason: Nitro Exception: Invalid value [sslProfileType, value differs from existing entity and it can't be updated.]


Troubleshooting - Prometheus and Grafana Integration

<table>
<thead>
<tr>
<th>Problem</th>
<th>Description</th>
<th>Workaround</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grafana dashboard has no plots</td>
<td>If the graphs on the Grafana dashboards do not have any values plotted, then Grafana is unable to obtain statistics from its datasource.</td>
<td>Check if the Prometheus datasource is saved and working properly. On saving the datasource after providing the Name and IP, a <strong>Data source is working</strong> message appears in green indicating the datasource is reachable and detected.</td>
</tr>
<tr>
<td><strong>Problem</strong></td>
<td><strong>Description</strong></td>
<td><strong>Workaround</strong></td>
</tr>
<tr>
<td>-------------</td>
<td>----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>If the dashboard is created using <code>sample_grafana_dashboard.json</code>, ensure that the name given to the Prometheus datasource begins with the word <code>prometheus</code> in the lowercase.</td>
<td></td>
<td>If the message appears against any of the exporter targets of Prometheus, then Prometheus is either unable to connect to the exporter or unable to fetch all the metrics within the given <code>scrape_timeout</code>.</td>
</tr>
<tr>
<td><strong>DOWN:</strong> Context deadline exceeded</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Troubleshooting - OpenShift feature node watch**

**Problem:** While using OpenShift-ovn CNI `feature-node-watch` is not adding correct routes.

**Description:** Citrix ingress controller looks for Node annotations for fetching the necessary details to add the static routes.

**Workaround:** Do the following steps as a workaround.
1. Make sure that following RBAC permission is provided to Citrix ingress controller along with `route.openshift.io` to run in the OpenShift environment with OVN CNI.

```yaml
- apiGroups: ["config.openshift.io"]
  resources: ["networks"]
  verbs: ["get", "list"]
```

2. Citrix ingress controller looks for the following two annotations added by OVN, make sure that it exists on the cluster nodes.

```yaml
"k8s.ovn.org/node-subnets": {
  "default": "10.128.0.0/23"
 },
"k8s.ovn.org/node-primary-ifaddr": {
  "ipv4": "x.x.x.x/24"
 }
```

3. If the annotation does not exist, `feature-node-watch` might not work for OVN CNI. In that case, you must manually configure the static routes on Citrix ADC VPX.

**Problem:** While using OpenShift-sdn CNI, feature-node-watch is not adding correct routes.

**Description:** Citrix ingress controller looks for the Host subnet CRD for fetching the necessary details to add the static routes.

**Workaround:** Do the following steps as a workaround.

1. Make sure that following RBAC permission is provided to Citrix ingress controller along with `route.openshift.io` to run in the OpenShift environment with SDN CNI.

```yaml
- apiGroups: ["network.openshift.io"]
  resources: ["hostsubnets"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["config.openshift.io"]
  resources: ["networks"]
  verbs: ["get", "list"]
```

2. Citrix ingress controller looks for the following CRD and specification.
3. If the CRD does not exist with the expected specification, feature-node-watch might not work for OpenShift-SDN CNI. In that case, you must manually configure the static routes on Citrix ADC VPX.

**Troubleshooting the Citrix ingress controller during runtime**

February 3, 2022

You can debug the Citrix ingress controller using the following methods:

- Event based debugging
- Log based debugging

**Event based debugging**

Events are Kubernetes entities which can provide information about the flow of execution on other Kubernetes entities.

Event based debugging for the Citrix ingress controller is enabled at the pod level. To enable event based debugging, the RBAC cluster role permissions for the pod should be the same as the cluster role permissions present in the `citrix-k8s-ingress-controller.yaml` file.

Use the following command to view the events for Citrix ingress controller.

```bash
oc get hostsubnets.network.openshift.io <cluster node-name> -ojson
```
You can view the events under the events section.

In this example, the Citrix ADC has been deliberately made unreachable and the same information can be seen under the events section.

```
kubectl describe pods cic-vpx-functionaltest -n functionaltest
```

You can use the events section to check the flow of events within the Citrix ingress controller. Events provide information on the flow of events. For further debugging, you should check the logs of the
Citrix ADC ingress controller

Citrix ingress controller pod.

**Log based debugging**

You can change the log level of the Citrix ingress controller at runtime using the ConfigMap feature. For changing the log level during runtime, see the ConfigMap documentation.

To check logs on the Citrix ingress controller, use the following command.

```
kubectl logs <citrix-k8s-ingress-controller> -n namespace
```

**Call Home enablement for the Citrix ingress controller in Citrix ADC**

November 3, 2021

Sometimes, Citrix needs to collect information about the performance of a product to diagnose issues and resolve them. The Call Home feature is designed to gather customer information and upload it to a Citrix server. Now, the Call Home feature available on Citrix ADC is enabled for the Citrix ingress controller.

The Call Home feature is enabled by default and requires no specific configuration by users. When the latest version of the Citrix ingress controller is deployed, a string map is configured on the Citrix ADC with the Citrix ingress controller specific information.

**Upgrade Citrix ingress controller**

July 5, 2022

This topic explains how to upgrade the Citrix ingress controller instance for Citrix ADC CPX with the Citrix ingress controller as sidecar and Citrix ingress controller standalone deployments.

**Upgrade Citrix ADC CPX with Citrix ingress controller as a sidecar**

To upgrade a Citrix ADC CPX with the Citrix ingress controller as a sidecar, you can either modify the associated YAML definition file (for example, citrix-k8s-cpx-ingress.yml) or use the Helm chart.

If you want to upgrade by modifying the YAML definition file, perform the following:

1. Change the version of the Citrix ingress controller and Citrix ADC CPX image under `containers` section to the following:
Citrix ADC ingress controller

- Citrix ADC CPX version: 13.0-83.27 (quay.io/citrix/citrix-k8s-cpx-ingress:13.0-83.27)
- Citrix ingress controller version: 1.26.7 (quay.io/citrix/citrix-k8s-ingress-controller:1.26.7)

2. Update the ClusterRole as follows:

```yaml
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: cic-k8s-role
rules:
- apiGroups: [""]
  resources: ["endpoints", "ingresses", "services", "pods", "secrets", "nodes", "routes", "namespaces"]
  verbs: ["get", "list", "watch"]

# service status is needed to update the loadbalancer IP in service status for integrating
# service of type LoadBalancer with external-dns
- apiGroups: [""]
  resources: ["services/status"]
  verbs: ["patch"]
- apiGroups: [""extensions"]
  resources: [""ingresses", ""ingresses/status"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["apiextensions.k8s.io"]
  resources: ["customresourcedefinitions"]
  verbs: ["get", "list", "watch"]
- apiGroups: [""apps""
  resources: [""deployments"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["citrix.com"
  resources: ["rewritepolicies", "canarycrds", "authpolicies", "ratelimits"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["citrix.com"
  resources: ["vips"]
  verbs: ["get", "list", "watch", "create", "delete"]
- apiGroups: ["route.openshift.io"
  resources: ["routes"]
  verbs: ["get", "list", "watch"]
```

3. Save the YAML definition file and reapply the file.
Upgrade a standalone Citrix ingress controller to version 1.21.9

To upgrade a standalone Citrix ingress controller instance, you can either modify the YAML definition file or use the Helm chart.

If you want to upgrade Citrix ingress controller to version 1.21.9 by modifying the YAML definition file, perform the following:

1. Change the version for the Citrix ingress controller image under `containers` section. For example, consider you have the following YAML file.

   ```yaml
   apiVersion: v1
   kind: Pod
   metadata:
     name: cic-k8s-ingress-controller
   labels:
     app: ...
   spec:
     serviceAccountName: ...
     containers:
       - name: cic-k8s-ingress-controller
         image: "citrix-k8s-ingress-controller:1.21.9"
         env: ...
         args: ...
   ``

   You should change the version of the image to version 1.21.9. For example, `quay.io/citrix/citrix-k8s-ingress-controller:1.21.9`.

2. Update the `ClusterRole` as follows:

   ```yaml
   kind: ClusterRole
   apiVersion: rbac.authorization.k8s.io/v1
   metadata:
     name: cic-k8s-role
   rules:
     - apiGroups: [""]
       resources: ["endpoints", "ingresses", "pods", "secrets", "nodes", "routes", "namespaces"]
       verbs: ["get", "list", "watch"]
   
   # services/status is needed to update the loadbalancer IP in service status for integrating
   
   # service of type LoadBalancer with external-dns
   ```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
3. Save the YAML definition file and reapply the file.

Securing Ingress

February 3, 2022

The topic covers the various ways to secure your Ingress using Citrix ADC and the annotations provided by the Citrix ingress controller.

The following table lists the TLS use cases with sample annotations that you can use to secure your Ingress using the Ingress Citrix ADC and the Citrix ingress controller:
<table>
<thead>
<tr>
<th>Use cases</th>
<th>Sample annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable TLSv1.3 protocol</td>
<td><code>ingress.citrix.com/frontend-sslprofile: '{ &quot;tls13&quot;:&quot;enabled&quot;, &quot;tls13sessionticketsperauthcontext&quot;:&quot;1&quot;, &quot;dhekeyexchangewithpsk&quot;:&quot;yes&quot; }'</code></td>
</tr>
<tr>
<td>HTTP strict transport security (HSTS)</td>
<td><code>ingress.citrix.com/frontend-sslprofile: '{ &quot;hsts&quot;:&quot;enabled&quot;, &quot;maxage&quot; : &quot;157680000&quot;, &quot;includesubdomain&quot;:&quot;yes&quot; }'</code></td>
</tr>
<tr>
<td>OCSP stapling</td>
<td><code>ingress.citrix.com/frontend-sslprofile: '{ &quot;ocspstapling&quot;:&quot;enabled&quot; }'</code></td>
</tr>
<tr>
<td>Set client authentication to mandatory</td>
<td><code>ingress.citrix.com/frontend-sslprofile: '{ &quot;clientauth&quot;:&quot;enabled&quot;, &quot;clientcert&quot; : &quot;mandatory&quot; }'</code></td>
</tr>
<tr>
<td>TLS session ticket extension</td>
<td><code>ingress.citrix.com/frontend-sslprofile: '{ &quot;sessionticket&quot; : &quot;enabled&quot;, &quot;sessionticketlifetime : &quot;300&quot; }'</code></td>
</tr>
<tr>
<td>SSL session reuse</td>
<td><code>ingress.citrix.com/frontend-sslprofile: '{ &quot;sessreuse&quot; : &quot;enabled&quot;, &quot;sesstimeout : &quot;120&quot; }'</code></td>
</tr>
<tr>
<td>Cipher groups</td>
<td><code>ingress.citrix.com/frontend-sslprofile:'{ &quot;snienable&quot;: &quot;enabled&quot;, &quot;ciphers&quot; : [{ &quot;ciphername&quot;: &quot;secure&quot;, &quot;cipherpriority&quot; :&quot;1&quot; }, { &quot;ciphername&quot;: &quot;secure&quot;, &quot;cipherpriority&quot; :&quot;21&quot; } ] }'</code></td>
</tr>
</tbody>
</table>
Enable TLS v1.3 protocol

Using the annotations for SSL profiles, you can enable TLS 1.3 protocol support on the SSL profile and set the `tls13SessionTicketsPerAuthContext` and `dheKeyExchangeWithPsk` parameters in the SSL profile for the Ingress Citrix ADC.

The `tls13SessionTicketsPerAuthContext` parameter enables you to set the number of tickets the Ingress Citrix ADC issues anytime TLS 1.3 is negotiated, ticket-based resumption is enabled, and either a handshake completes or post-handshake client authentication completes. The value can be increased to enable clients to open multiple parallel connections using a fresh ticket for each connection. The minimum value you can set is 1 and the maximum is 10. By default, the value is set to 1.

**Note:**
No tickets are sent if resumption is disabled.

The `dheKeyExchangeWithPsk` parameter allows you to specify whether the Ingress Citrix ADC requires a DHE key exchange to occur when a preshared key is accepted during a TLS 1.3 session resumption handshake. A DHE key exchange ensures forward secrecy, even if ticket keys are compromised, at the expense of extra resources required to carry out the DHE key exchange.

The following is a sample annotation for the HTTP profile to enable TLS 1.3 protocol support on SSL profile and set the `tls13SessionTicketsPerAuthContext` and `dheKeyExchangeWithPsk` parameters in the SSL profile.

```plaintext
ingress.citrix.com/frontend-sslprofile: {
  "tls13": "enabled",
  "tls13sessionticketsperauthcontext": "1",
  "dhekeyexchangewithpsk": "yes"
}
```
HTTP strict transport security (HSTS)

The Ingress Citrix ADC appliances support HTTP strict transport security (HSTS) as an inbuilt option in SSL profiles. Using HSTS, a server can enforce the use of an HTTPS connection for all communication with a client. That is, the site can be accessed only by using HTTPS. Support for HSTS is required for A+ certification from SSL Labs. For more information, see Citrix ADC support for HSTS.

Using the annotations for SSL profiles, you can enable HSTS in an SSL front-end profile on the Ingress Citrix ADC. The following is a sample ingress annotation:

```yaml
ingress.citrix.com/frontend-sslprofile: '{
  "hsts":"enabled", "maxage" : "157680000", "includesubdomain":"yes" }
'
```

Where:
- **HSTS** - The state of HTTP Strict Transport Security (HSTS) on the SSL profile. Using HSTS, a server can enforce the use of an HTTPS connection for all communication with a client. The supported values are ENABLED and DISABLED. By default, the value is set to DISABLED.
- **maxage** - Allows you to set the maximum time, in seconds, in the strict transport security (STS) header during which the client must send only HTTPS requests to the server. The minimum time you can set is 0 and the maximum is 4294967294. By default the value is to 0.
- **IncludeSubdomains** - Allows you to enable HSTS for subdomains. If set to Yes, a client must send only HTTPS requests for subdomains. By default the value is set to No.

OCSP stapling

The Ingress Citrix ADC can send the revocation status of a server certificate to a client, at the time of the SSL handshake, after validating the certificate status from an OCSP responder. The revocation status of a server certificate is “stapled” to the response the appliance sends to the client as part of the SSL handshake. For more information on Citrix ADC implementation of CRL and OCSP reports, see OCSP stapling.

To use the OCSP stapling feature, you can enable it using an SSL profile with the following ingress annotation:

```yaml
ingress.citrix.com/frontend-sslprofile: '{
  "ocspstapling":"enabled" }
'
```
**Citrix ADC ingress controller**

**Note:**
To use OCSP stapling, you must add an OCSP responder on the Citrix ADC appliance.

**Set Client authentication to mandatory**

Using the annotations for SSL profiles, you can enable client authentication, the Ingress Citrix ADC appliance asks for the client certificate during the SSL handshake.

The appliance checks the certificate presented by the client for normal constraints, such as the issuer signature and expiration date.

Here are some use cases:

- Require a valid client certificate before website content is displayed. This restricts website content to only authorized machines and users.
- Request a valid client certificate. If a valid client certificate is not provided, then prompt the user for multifactor authentication.

Client authentication can be set to mandatory, or optional.

- When it is set as mandatory, if the SSL Client does not transmit a valid Client Certificate, then the connection is dropped. Valid means: signed/issued by a specific Certificate Authority, and not expired or revoked.
- When it is optional, then the Citrix ADC requests the client certificate, but proceeds with the SSL transaction even if the client presents an invalid certificate or no certificate. This configuration is useful for authentication scenarios (for example require two-factor authentication if a valid Client Certificate is not provided).

Using the annotations for SSL profiles, you can enable client authentication on an SSL virtual server and set client authentication as **Mandatory**.

The following is a sample annotation of the SSL profile:

```plaintext
1 ingress.citrix.com/frontend-sslprofile: '{
2   "clientauth":"enabled", "clientcert": "mandatory" }
3 ' 
```

**Note:**
Make sure that you bind the client-certificate to the SSL virtual server on the Ingress Citrix ADC.
TLS session ticket extension

An SSL handshake is a CPU-intensive operation. If session reuse is enabled, the server or client key exchange operation is skipped for existing clients. They are allowed to resume their sessions. This improves the response time and increases the number of SSL transactions per second that a server can support. However, the server must store details of each session state, which consumes memory and is difficult to share among multiple servers if requests are load balanced across servers.

The Ingress Citrix ADC appliances support the SessionTicket TLS extension. Use of this extension indicates that the session details are stored on the client instead of on the server. The client must indicate that it supports this mechanism by including the session ticket TLS extension in the client Hello message. For new clients, this extension is empty. The server sends a new session ticket in the NewSessionTicket handshake message. The session ticket is encrypted by using a key-pair known only to the server. If a server cannot issue a new ticket currently, it completes a regular handshake.

Using the annotations for SSL profiles, you can enable the use of session tickets, as per the RFC 5077. Also, you can set the life time of the session tickets issued by the Ingress Citrix ADC, using the `sessionticketlifetime` parameter.

The following is the sample ingress annotation:

```plaintext
ingress.citrix.com/frontend-sslprofile: '{
   "sessionticket": "enabled",
   "sessionticketlifetime": "300"
}
'
```

SSL session reuse

You can reuse an existing SSL session on a Citrix ADC appliance. While the SSL renegotiation process consists of a full SSL handshake, the SSL reuse consists of a partial handshake because the client sends the SSL ID with the request.

Using the annotations for SSL profiles, you can enable session reuse and also set the session timeout value (in seconds) on the Ingress Citrix ADC.

The following is the sample ingress annotation:

```plaintext
ingress.citrix.com/frontend-sslprofile: '{
   "sessreuse": "enabled",
   "sesstimeout": "120"
}
'
```

By default, the session reuse option is enabled on the appliance and the timeout value for the same is set to 120 seconds. Therefore, if a client sends a request on another TCP connection and the earlier
SSL session ID within 120 seconds, then the appliance performs a partial handshake.

**Using cipher groups**

The Ingress Citrix ADC ships with built-in cipher groups. To use ciphers that are not part of the DEFAULT cipher group, you have to explicitly bind them to an SSL profile. You can also create a user-defined cipher group to bind to the SSL virtual server on the Ingress Citrix ADC.

The built-in cipher groups can be used in Tier-1 and Tier-2 Citrix ADC, and the user-defined cipher group can be used only in Tier-1 Citrix ADC.

To use a user-defined cipher group, ensure that the Citrix ADC has a user-defined cipher group. Perform the following:

1. Create a user-defined cipher group. For example, **testgroup**.
2. Bind all the required ciphers to the user-defined cipher group.
3. Note down the user-defined cipher group name.

For detailed instructions, see Configure a user-defined cipher group.

Using the annotations for SSL profiles, you can bind the built-in cipher groups, a user-defined cipher group or both to the SSL profile.

The following is the syntax of the ingress annotation that you can use to bind the built-in cipher groups and a user-defined cipher group to an SSL profile:

```json
ingress.citrix.com/frontend-sslprofile: {
  "snienable": "enabled",
  "ciphers": [ {
    "ciphername": "secure",
    "cipherpriority": "1"
  }, {
    "ciphername": "testgroup",
    "cipherpriority": "2"
  } ]
}
```

The ingress annotation binds the built-in cipher group, **SECURE**, and the user-defined cipher group, **testgroup**, to the SSL profile.

**Using cipher redirect**

During the SSL handshake, the SSL client (usually a web browser) announces the suite of ciphers that it supports, in the configured order of cipher preference. From that list, the SSL server then selects a cipher that matches its own list of configured ciphers.
If the ciphers announced by the client does not match those ciphers configured on the SSL server, the SSL handshake fails. The failure is announced by a cryptic error message displayed in the browser. These messages rarely mention the exact cause of the error.

With cipher redirection, you can configure an SSL virtual server to deliver accurate, meaningful error messages when an SSL handshake fails. When the SSL handshake fails, the Citrix ADC appliance redirects the user to a previously configured URL or, if no URL is configured, displays an internally generated error page.

The following is the syntax of the ingress annotation that you can use to bind cipher groups and enable cipher redirect to redirect the request to `redirecturl`.

```plaintext
ingress.citrix.com/frontend-sslprofile:'{
  "snienable": "enabled",
  "ciphers": [{
    "ciphername": "secure",
    "cipherpriority": "1"
  }],
  "cipherredirect": "enabled",
  "cipherurl": "https://redirecturl"
}
```

# TCP use cases

February 3, 2022

This topic covers various TCP use cases that you can configure on the Ingress Citrix ADC using the annotations in the Citrix ingress controller.

The following table lists the TCP use cases with sample annotations:

<table>
<thead>
<tr>
<th>Use case</th>
<th>Sample annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Silently drop idle TCP connections</td>
<td>ingress.citrix.com/frontend-tcpprofile: '{ &quot;apache&quot;:{ &quot;DropHalfClosedConnOnTimeout&quot;: &quot;ENABLE&quot;, &quot;DropEstConnOnTimeout&quot;: &quot;ENABLE&quot; } }'</td>
</tr>
<tr>
<td>Delayed TCP connection acknowledgments</td>
<td>ingress.citrix.com/frontend-tcpprofile: '{ &quot;apache&quot;:{ &quot;delayack&quot;: &quot;150&quot; } }'</td>
</tr>
</tbody>
</table>
Silently drop idle TCP connections

In a network, large number of TCP connections become idle, and the Ingress Citrix ADC sends RST packets to close them. The packets sent over the channels activate those channels unnecessarily, causing a flood of messages that in turn causes the Ingress Citrix ADC to generate a flood of service-reject messages.

Using the `drophalfclosedconnontimeout` and `dropestconnontimeout` parameters in TCP profiles, you can silently drop TCP half closed connections on idle timeout or drop TCP established connections on an idle timeout. By default, these parameters are disabled on the Ingress Citrix ADC. If you enable both of them, neither a half closed connection nor an established connection causes an RST packet to be sent to the client when the connection times out. The Citrix ADC just drops the connection.

Using the annotations for TCP profiles, you can enable or disable the `drophalfclosedconnontimeout` and `dropestconnontimeout` on the Ingress Citrix ADC. The following is a sample annotation of TCP profile to enable these parameters:

```plaintext
ingress.citrix.com/frontend-tcprofile: '{
  "apache":{
    "mptcp": "ENABLED",
    "mptcpSessionTimeout": "7200"
  }
}'
```
Citrix ADC ingress controller

Delayed TCP connection acknowledgments

To avoid sending several ACK packets, Ingress Citrix ADC supports TCP delayed acknowledgment mechanism. It sends delayed ACK with a default timeout of 100 ms. Ingress Citrix ADC accumulates data packets and sends ACK only if it receives two data packets in continuation or if the timer expires. The minimum delay you can set for the TCP deployed ACK is 10 ms and the maximum is 300 ms. By default the delay is set to 100 ms.

Using the annotations for TCP profiles, you can manage the delayed ACK parameter. The following is a sample annotation of TCP profile to enable these parameters:

```plaintext
ingress.citrix.com/frontend-tcpprofile: '{
  "apache":{
    "delayack": "150"
  }
}
'
```

Client side MPTCP session management

You perform TCP configuration on the Ingress Citrix ADC for MPTCP connections between the client and Ingress Citrix ADC. MPTCP connections are not supported between Citrix ADC and the back-end communication. Both the client and the Ingress Citrix ADC appliance must support the same MPTCP version.

You can enable MPTCP and set the MPTCP session timeout (mptcpsessiontimeout) in seconds using TCP profiles in the Ingress Citrix ADC. If the mptcpsessiontimeout value is not set then the MPTCP sessions are flushed after the client idle timeout. The minimum timeout value you can set is 0 and the maximum is 86400. By default, the timeout value is set to 0.

Using the annotations for TCP profiles, you can enable MPTCP and set the mptcpsessiontimeout parameter value on the Ingress Citrix ADC. The following is a sample annotation of TCP profile to enable MPTCP and set the mptcpsessiontimeout parameter value to 7200 on the Ingress Citrix ADC:

```plaintext
ingress.citrix.com/frontend-tcpprofile: '{
  "apache":{
    "mptcp": "ENABLED",
    "mptcpSessionTimeout": "7200"
  }
}
'
```
TCP Optimization

Most of the relevant TCP optimization capabilities of the Ingress Citrix ADC are exposed through a corresponding TCP profile. Using the annotations for TCP profiles, you can enable the following TCP optimization capabilities on the Ingress Citrix ADC:

- **Selective acknowledgment (SACK):** TCP SACK addresses the problem of multiple packet losses which reduces the overall throughput capacity. With selective acknowledgment the receiver can inform the sender about all the segments which are received successfully, enabling sender to only retransmit the segments which were lost. This technique helps T1 improve overall throughput and reduce the connection latency.

  The following is a sample annotation of TCP profile to enable SACK on the Ingress Citrix ADC:

  ```
  ingress.citrix.com/frontend_tcpprofile: '{
  "sack" : "enabled"
  }
  ```

- **Forward acknowledgment (FACK):** To avoid TCP congestion by explicitly measuring the total number of data bytes outstanding in the network, and helping the sender (either T1 or a client) control the amount of data injected into the network during retransmission timeouts.

  The following is a sample annotation of TCP profile to enable FACK on the Ingress Citrix ADC:

  ```
  ingress.citrix.com/frontend_tcpprofile: '{
  "fack" : "enabled"
  }
  ```

- **Window Scaling (WS):** TCP Window scaling allows increasing the TCP receive window size beyond 65535 bytes. It helps improving TCP performance overall and specially in high bandwidth and long delay networks. It helps with reducing latency and improving response time over TCP.

  The following is a sample annotation of TCP profile to enable WS on the Ingress Citrix ADC:

  ```
  ingress.citrix.com/frontend_tcpprofile: '{
  "ws" : "enabled", "wsval" : "9"
  }
  ```

Where **wsval** is the factor used to calculate the new window size. The argument is mandatory only when window scaling is enabled. The minimum value you can set is 0 and the maximum is 14. By default, the value is set to 4.
Citrix ADC ingress controller

- **Maximum Segment Size (MSS):** MSS of a single TCP segment. This value depends on the MTU setting on intermediate routers and end clients. A value of 1460 corresponds to an MTU of 1500.

  The following is a sample annotation of TCP profile to enable MSS on the Ingress Citrix ADC:

```plaintext
ingress.citrix.com/frontend_tcpprofile: '{
  "mss": "1460",
  "maxPktPerMss": "512"
}
```

Where:

- `mss` is the MSS to use for the TCP connection. The minimum value you can set is 0 and the maximum is 9176.
- `maxPktPerMss` is the maximum number of TCP packets allowed per maximum segment size (MSS). The minimum value you can set is 0 and the maximum is 1460.

- **Keep-Alive (KA):** Send periodic TCP keep-alive (KA) probes to check if the peer is still up.

  The following is a sample annotation of TCP profile to enable TCP keep-alive (KA) on the Ingress Citrix ADC:

```plaintext
ingress.citrix.com/frontend_tcpprofile: '{
  "ka": "enabled",
  "kaprobeupdatelastactivity": "enabled",
  "KAconnIdleTime": "900",
  "kamaxprobes": "3",
  "kaprobeinterval": "75"
}
```

Where:

- `ka` is used to enable sending periodic TCP keep-alive (KA) probes to check if the peer is still up. Possible values: ENABLED, DISABLED. Default value: DISABLED.
- `kaprobeupdatelastactivity` updates the last activity for the connection after receiving keep-alive (KA) probes. Possible values: ENABLED, DISABLED. Default value: ENABLED.
- `KAconnIdleTime` is the duration (in seconds) for the connection to be idle, before sending a keep-alive (KA) probe. The minimum value you can set is 1 and the maximum is 4095.
- `kaprobeinterval` is the time interval (in seconds) before the next keep-alive (KA) probe, if the peer does not respond. The minimum value you can set is 1 and the maximum is 4095.

- **bufferSize:** Specify the TCP buffer size, in bytes. The minimum value you can set is 8190 and the maximum is 20971520. By default the value is set to 8190.

  The following is a sample annotation of TCP profile to specify the TCP buffer size:
• **MPTCP**: Enable MPTCP and set the optional MPTCP configuration. The following is a sample annotation of TCP profile to enable MPTCP and set the optional MPTCP configurations:

```
-ingress.citrix.com/frontend_tcpprofile: '{
  "bufferSize" : "8190"
}
```

```
-ingress.citrix.com/frontend_tcpprofile: '{
  "mptcp" : "enabled",
  "mptcpDropDataOnPreEstSF" : "enabled",
  "mptcpFastOpen" : "enabled",
  "mptcpSessionTimeout" : "7200"
}
```

• **flavor**: Set the TCP congestion control algorithm. Valid values are Default, BIC, CUBIC, Westwood, and Nile. By default the value is set to Default. The following is a sample annotation of TCP profile to set the TCP congestion control algorithm:

```
-ingress.citrix.com/frontend_tcpprofile: '{
  "flavor" : "westwood"
}
```

• **Dynamic receive buffering**: Enable or disable dynamic receive buffering. When enabled, it allows the receive buffer to be adjusted dynamically based on memory and network conditions. Possible values: ENABLED, DISABLED, and the Default value: DISABLED.

**Note:**
The buffer size argument must be set for dynamic adjustments to take place.

```
-ingress.citrix.com/frontend_tcpprofile: '{
  "dynamicReceiveBuffering" : "enabled"
}
```

---

**Defend TCP against spoofing attacks**

You can enable the Ingress Citrix ADC to defend TCP against spoof attacks using the `rstWindowAttenuation` in TCP profiles. By default the `rstWindowAttenuation` parameter is disabled. This parameter is enabled to protect the Ingress Citrix ADC against spoofing. If you enable, it replies with corrective acknowledgment (ACK) for an invalid sequence number. Possible values are Enabled or Disabled.

The following is a sample annotation of TCP profile to enable `rstWindowAttenuation` on the Ingress Citrix ADC:
This topic covers various HTTP use cases that you can configure on the Ingress Citrix ADC using the annotations in the Citrix ingress controller.

The following table lists the HTTP use cases with sample annotations:

<table>
<thead>
<tr>
<th>Use case</th>
<th>Sample annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuring HTTP/2</td>
<td><code>ingress.citrix.com/frontend_tcpprofile: '{ &quot;http2&quot;:&quot;enabled&quot; }', </code></td>
</tr>
<tr>
<td></td>
<td><code>ingress.citrix.com/backend_httpprofile: '{ &quot;apache&quot;:{ &quot;http2direct&quot;: &quot;enabled&quot; }', </code></td>
</tr>
<tr>
<td>Handling HTTP session timeouts</td>
<td><code>ingress.citrix.com/frontend_httpprofile: '{ &quot;apache&quot;:{ &quot;reqtimeout&quot; : &quot;10&quot;, &quot;reqtimeoutaction&quot;:&quot;drop&quot; } }', </code></td>
</tr>
<tr>
<td></td>
<td><code>ingress.citrix.com/backend_httpprofile: '{ &quot;apache&quot;:{ &quot;reqtimeout&quot; : &quot;10&quot;, &quot;adptimeout&quot; : &quot;enable&quot; } }', </code></td>
</tr>
<tr>
<td></td>
<td>`ingress.citrix.com/backend_httpprofile: '{ &quot;apache&quot;:{ &quot;reusepooltimeout&quot; : &quot;20000&quot; } }'</td>
</tr>
</tbody>
</table>
Configuring HTTP/2

The Ingress Citrix ADC HTTP/2 on the client side as well on the server side. For more information, see HTTP/2 support on Citrix ADC. For an HTTP load balancing configuration on the Ingress Citrix ADC, it uses one of the following methods to start communicating with the client/server using HTTP/2.

The Ingress Citrix ADC provides configurable options in an HTTP profile for the HTTP/2 methods. These HTTP/2 options can be applied to the client side as well to the server side of an HTTPS or HTTP load balancing setup. The Citrix ingress controller provides annotations to configure HTTP profile on the Ingress Citrix ADC. You use these annotations to configure the various HTTP loads balancing configuration on the Ingress Citrix ADC to communicate with the client/server using HTTP/2.

Note:
Ensure that the HTTP/2 Service Side global parameter (HTTP2Serverside) is enabled on the Ingress Citrix ADC. For more information, see nshttpparam.

HTTP/2 upgrade

In this method, a client sends an HTTP/1.1 request to a server. The request includes an upgrade header, which asks the server for upgrading the connection to HTTP/2. If the server supports HTTP/2, the server accepts the upgrade request and notifies it in its response. The client and the server start communicating using HTTP/2 after the client receives the upgrade confirmation response.

Using the annotations for HTTP profiles, you can configure the HTTP/2 upgrade method on the Ingress Citrix ADC. The following is a sample annotation of HTTP profile to configure HTTP/2 upgrade method on the Ingress Citrix ADC:

```
1 ingress.citrix.com/frontend-httpprofile: ' {
2    "http2":"enabled" }
3 ',
```

Direct HTTP/2

In this method, a client directly starts communicating to a server in HTTP/2 instead of using the HTTP/2 upgrade method. If the server does not support HTTP/2 or is not configured to directly accept HTTP/2 requests, it drops the HTTP/2 packets from the client. This method is helpful if the admin of the client device already knows that the server supports HTTP/2.

Using the annotations for HTTP profiles, you can configure the direct HTTP/2 method on the Ingress Citrix ADC. The following is a sample annotation of HTTP profile to configure the direct HTTP/2 method on the Ingress Citrix ADC:
Direct HTTP/2 using Alternative Service (ALT-SVC)

In this method, a server advertises that it supports HTTP/2 to a client by including an Alternative Service (ALT-SVC) field in its HTTP/1.1 response. If the client is configured to understand the ALT-SVC field, the client and the server start directly communicating using HTTP/2 after the client receives the response.

The following is a sample annotation of HTTP profile to configure the direct HTTP/2 using alternative service (ALT-SVC) method on the Ingress Citrix ADC:

```plaintext
ingress.citrix.com/backend-httpprofile: '{
  "apache":{
    "http2direct": "enabled",
    "altsvc": "enabled"
  }
}
```

Handling HTTP session timeouts

To handle the different type of HTTP request and also to mitigate attacks such as, Slowloris DDoS attack, where in the clients initiate connections that you might want to restrict. On the Ingress Citrix ADC, you can configure the following timeouts for these scenarios:

- reqTimeout and reqTimeoutAction
- adptTimeout
- reusePoolTimeout

reqTimeout and reqTimeoutAction

In Citrix ADC, you can configure the HTTP request timeout value and the request timeout action using the `reqTimeout` and `reqTimeoutAction` parameter in the HTTP profile. The `reqTimeout` value is set in seconds and the HTTP request must complete within the specified time in the `reqTimeout` parameter. If the HTTP request does not complete within defined time, the specified request timeout action in the `reqTimeoutAction` is executed. The minimum timeout value you can set is 0 and the maximum is 86400. By default, the timeout value is set to 0.
Using the `reqTimeoutAction` parameter you can specify the type of action that must be taken in case the HTTP request timeout value (`reqTimeout`) elapses. You can specify the following actions:

- RESET
- DROP

Using the annotations for HTTP profiles, you can configure the HTTP request timeout and HTTP request timeout action. The following is a sample annotation of HTTP profile to configure the HTTP request timeout and HTTP request timeout action on the Ingress Citrix ADC:

```plaintext
ingress.citrix.com/frontend-httpprofile: '{
  "apache": {
    "reqtimeout": "10", "reqtimeoutaction":"drop" 
  }
}
'
```

**adptTimeout**

Instead of using a set timeout value for the requested sessions, you can also enable `adptTimeout`. The `adptTimeout` parameter adapts the request timeout as per the flow conditions. If enabled, then request timeout is increased or decreased internally and applied on the flow. By default, this parameter is set as DISABLED.

Using annotations for HTTP profiles, you can enable or disable the `adpttimeout` parameter as follows:

```plaintext
ingress.citrix.com/frontend-httpprofile: '{
  "apache": {
    "reqtimeout": "10", "adpttimeout": "enable" 
  }
}
'
```

**reusePoolTimeout**

You can configure a reuse pool timeout value to flush any idle server connections in from the reuse pool. If the server is idle for the configured amount of time, then the corresponding connections are flushed.

The minimum timeout value you can set is 0 and the maximum is 31536000. By default, the timeout value is set to 0.
Using annotations for HTTP profiles, you can configure the required timeout value as follows:

```python
ingress.citrix.com/backend-httpprofile: '{
  "apache": {
    "reuserequesttimeout": "20000"
  }
}
```

### HTTP callout with the rewrite and responder policy

February 3, 2022

An HTTP callout allows Citrix ADC to generate and send an HTTP or HTTPS request to an external server (callout agent) as part of the policy evaluation. The information that is retrieved from the server (callout agent) can be analyzed by advanced policy expressions and an appropriate action can be performed. For more information about the HTTP callout, see the [Citrix ADC documentation](#).

You can initiate the HTTP callout through the following expressions with the rewrite and responder CRD provided by Citrix:

- **sys.http_callout()**: This expression is used for blocking the call when the httpcallout agent response needs to be evaluated.

- **sys.nonBlocking_http_callout()**: This expression is used for non-blocking calls (for example: traffic mirroring)

These expressions accept the `httpcallout_policy` name defined in the CRD as a parameter, where the name needs to be specified in double quotes.

For example: `sys.http_callout("callout_name")`.

In this expression, `callout_name` refers to the appropriate `httpcallout_policy` defined in the rewrite and responder CRD YAML file.

The following table explains the attributes of the HTTP callout request in the rewrite and responder CRD.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Specifies the name of the callout, maximum is up to 32 characters.</td>
</tr>
<tr>
<td>server_ip</td>
<td>Specifies the IP Address of the server (callout agent) to which the callout is sent.</td>
</tr>
</tbody>
</table>
### Server Port

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server_port</td>
<td>Specifies the Port of the server (callout agent) to which the callout is sent.</td>
</tr>
</tbody>
</table>

### HTTP Method

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>http_method</td>
<td>Specifies the method used in the HTTP request that this callout sends. The default value is GET.</td>
</tr>
</tbody>
</table>

### Host Expression

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host_expr</td>
<td>Specifies the text expression to configure the host header. This expression can be a literal value (for example, 192.101.10.11) or it can be an advanced expression (for example, http.req.header(&quot;Host&quot;)) that derives the value. The literal value can be an IP address or a fully qualified domain name. Mutually exclusive with the full HTTP request expression.</td>
</tr>
</tbody>
</table>

### URL Stem Expression

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url_stem_expr</td>
<td>Specifies a string expression for generating the URL stem. The string expression can contain a literal string (for example, &quot;/mysite/index.html&quot;) or an expression that derives the value (for example, http.req.url).</td>
</tr>
</tbody>
</table>

### Headers

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>headers</td>
<td>Specifies one or more headers to insert into the HTTP request. Each header name and exp, where exp is an expression that is evaluated at runtime to provide the value for the named header.</td>
</tr>
</tbody>
</table>

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameters</td>
<td>Specifies one or more query parameters to insert into the HTTP request URL (for a GET request) or into the request body (for a POST request). Each parameter is represented by a name and an expr, where expr is an expression that is evaluated at run time to provide the value for the named parameter (name=value). The parameter values are URL encoded.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>body_expr</td>
<td>An advanced string expression for generating the body of the request. The expression can contain a literal string or an expression that derives the value (e.g., client.ip.src).</td>
</tr>
<tr>
<td>full_req_expr</td>
<td>Specifies the exact HTTP request, in the form of an expression, which the Citrix ADC sends to the callout agent. The request expression is constrained by the feature for which the callout is used. For example, an HTTP.RES expression cannot be used in a request-time policy bank or in a TCP content switching policy bank.</td>
</tr>
<tr>
<td>scheme</td>
<td>Specifies the type of scheme for the callout server. Example: HTTP, HTTPS</td>
</tr>
<tr>
<td>return_type</td>
<td>Specifies the type of data that the target callout agent returns in response to the callout. The available settings function as follows: TEXT - Treat the returned value as a text string. NUM - Treat the returned value as a number. BOOL - Treat the returned value as a boolean value.</td>
</tr>
<tr>
<td>cache_for_secs</td>
<td>Specifies the duration, in seconds, for which the callout response is cached. The cached responses are stored in an integrated caching content group named calloutContentGroup. If the duration is not configured, the callout responses are not cached unless a normal caching configuration is used to cache them. This parameter takes precedence over any normal caching configuration that would otherwise apply to these responses.</td>
</tr>
</tbody>
</table>
**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result_expr</td>
<td>Specifies the expression that extracts the callout results from the response sent by the HTTP callout agent. This expression must be a response based expression, that is, it must begin with <code>HTTP.RES</code>. The operations in this expression must match the return type. For example, if you configure a return type of <code>TEXT</code>, the result expression must be a text based expression. If the return type is <code>NUM</code>, the result expression (result_expr) must return a numeric value, as in the following example: <code>http.res.body(10000).length</code></td>
</tr>
<tr>
<td>comment</td>
<td>Specifies any comments to preserve the information about this HTTP callout.</td>
</tr>
</tbody>
</table>

**Using the rewrite and responder CRD to validate whether a client IP address is blocklisted**

This section shows how to initiate an HTTP callout using the rewrite and responder CRD to validate whether a client IP address is blocklisted or not and take appropriate action.

The following diagram explains the workflow of a request where each number in the diagram denotes a step in the workflow:

1. Client request
2. HTTP callout request to check if the client is blocklisted (The client IP address is sent as a query parameter with the name `Cip`)

3. Response from the HTTP callout server

4. Request is forwarded to the service if the response in step 3 indicates a safe IP address (the client IP address is not matching with the blocklisted IP addresses on the callout server).

5. Respond to the client as `Access denied`, if the response in step 3 indicates a bad IP address (the client IP address is matching with the blocklisted IP addresses on the callout server).

The following is a sample YAML file (`ip_validate_responder.yaml`) for validating a blocklisted IP address:

```
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
  name: validateip
spec:
  responder-policies:
    - servicenames:
      - frontend
        responder-policy:
          respondwith:
            http-payload-string: '"HTTP/1.1 401 Access denied\r\n\r\n"
          respond-criteria: 'sys.http_callout("blocklist_callout").CONTAINS("IP Matched")'  # Callout name needs to be given in double quotes to pick httpcallout_policy
          comment: 'Invalid access'

httpcallout_policy:
  - name: blocklist_callout
    server_ip: "192.2.156.160"
    server_port: 80
    http_method: GET
    host_expr: '"192.2.156.160"'
    url_stem_expr: '"/validateIP.pl"'
    headers:
      - name: X-Request
        expr: '"Callout Request"'
    parameters:
```

**Note:**
You must deploy the **rewrite and responder CRD** before deploying the `ip_validate_responder` YAML file.
Using the rewrite and responder CRD to update the URL with a valid path requested by the client

This section shows how to initiate an HTTP callout using the rewrite and responder CRD when a path exposed to the client is different from the actual path due to security reasons.

The work flow of a request is explained in the following diagram where each number in the diagram denotes a step in the workflow.

1. Client request
2. HTTP callout request to get the valid path (the path requested from the client is sent as a query parameter with the name path to the callout server)
3. Response from the HTTP callout server
4. The URL request is rewritten with a valid path and forwarded to the service (where the valid path is mentioned between the tags newpath in the callout response).

The following is a sample YAML (path_rewrite) file.

```
- name: Cip
  expr: 'CLIENT.IP.SRC'
  return_type: TEXT
  result_expr: 'HTTP.RES.BODY(100)'
```

Note:
You must deploy the rewrite and responder CRD before deploying the path_rewrite YAML file.
Configure session affinity or persistence on the Ingress Citrix ADC

January 20, 2022

Session affinity or persistence settings on the Ingress Citrix ADC allows you to direct client requests
to the same selected server regardless of which virtual server in the group receives the client request. When the configured time for persistence expires, any virtual server in the group is selected for the incoming client requests.

If persistence is configured, it overrides the load balancing methods once the server has been selected. It maintains the states of connections on the servers represented by that virtual server. The Citrix ADC then uses the configured load balancing method for the initial selection of a server, but forwards to that same server all subsequent requests from the same client.

The most commonly used persistence type is persistence based on cookies.

**Configure persistence based on cookies**

When you enable persistence based on cookies, the Citrix ADC adds an HTTP cookie into the `Set-Cookie` header field of the HTTP response. The cookie contains information about the service to which the HTTP requests must be sent. The client stores the cookie and includes it in all subsequent requests, and the ADC uses it to select the service for those requests.

The Citrix ADC inserts the cookie `<NSC_XXXX>=<ServiceIP> <ServicePort>`. Where:

- `<<NSC_XXXX>>` is the virtual server ID that is derived from the virtual server name.
- `<<ServiceIP>>` is the hexadecimal value of the IP address of the service.
- `<<ServicePort>>` is the hexadecimal value of the port of the service.

The Citrix ADC encrypts `ServiceIP` and `ServicePort` when it inserts a cookie, and decrypts them when it receives a cookie.

For example, `a.com=ffffffff02091f1045525d5f4f58455e445a4a423660; expires=Fri, 23-Aug-2019 07:01:45.`

You can configure persistence setting on the ingress Citrix ADC, using the following Ingress annotation provided by the Citrix ingress controller:

```yaml
ingress.citrix.com/lbvserver: '{
  "apache":{
    "persistenceType":"COOKIEINSERT", "timeout":"20", "cookiename":"
    k8s_cookie"
  }
}
'
```

**Where:**
Citrix ADC ingress controller

- **timeout** specifies the duration of persistence. If session cookies are used with a timeout value of 0, no expiry time is specified by Citrix ADC regardless of the HTTP cookie version used. The session cookie expires when the Web browser is closed.
- **cookiename** specifies the name of cookie with a maximum of 32 characters. If not specified, cookie name is internally generated.
- **persistenceType** here specifies the type of persistence to be used, COOKIENAME is used to cookie based persistence. Apart from cookie, other options can also be used along with appropriate arguments and other required parameters.

Possible values are SOURCEIP, SSLSESSION, DESTIP, SRCIPDESTIP, and so on.

**Source IP address persistence**

When source IP persistence is configured on the Ingress Citrix ADC, you can set persistence to an load balancing virtual server, that creating a stickiness for the subsequent requests from the same client.

The following is a sample Ingress annotation to configure source IP address persistence:

```plaintext
ingress.citrix.com/lbvserver: '{
"apache":{
"persistenceType":"SOURCEIP",
"timeout":"10"
}
}
'
```

**SSL session ID persistence**

When SSL session ID persistence is configured, the Citrix ADC appliance uses the SSL session ID, which is part of the SSL handshake process, to create a persistence session before the initial request is directed to a service. The load balancing virtual server directs subsequent requests that have the same SSL session ID to the same service. This type of persistence is used for SSL bridge services.

The following is a sample Ingress annotation to configure SSL session ID persistence:

```plaintext
ingress.citrix.com/lbvserver: '{
"apache":{
"persistenceType":"SSLSESSION"
}
}
'
```
Citrix ADC ingress controller

Destination IP address-based persistence

In this type of persistence, when the Ingress Citrix ADC receives a request from a new client, it creates a persistence session based on the IP address of the service selected by the virtual server (the destination IP address). Subsequently, it directs requests to the same destination IP to the same service. This type of persistence is used with link load balancing.

The following is a sample Ingress annotation to configure destination IP address-based persistence:

```json
ingress.citrix.com/lbvserver: '{
  "apache":{
    "persistenceType":"DESTIP" }
  }
'
```

Source and destination IP address-based persistence

In this type of persistence, when the Citrix ADC appliance receives a request, it creates a persistence session based on both the IP address of the client (the source IP address) and the IP address of the service selected by the virtual server (the destination IP address). Subsequently, it directs requests from the same source IP and to the same destination IP to the same service.

The following is a sample Ingress annotation to configure source and destination IP address-based persistence:

```json
ingress.citrix.com/lbvserver: '{
  "apache":{
    "persistenceType":"SRCIPDESTIP" }
  }
'
```

Allowlisting or blocklisting IP addresses

February 3, 2022

Allowlisting IP addresses allows you to create a list of trusted IP addresses or IP address ranges from which users can access your domains. It is a security feature that is often used to limit and control access only to trusted users.
Blocklisting IP addresses is a basic access control mechanism. It denies access to the users accessing your domain using the IP addresses that you have blocklisted.

The Rewrite and Responder CRD provided by Citrix enables you to define extensive rewrite and responder policies using datasets, patsets, and string maps and also enable audit logs for statistics on the Ingress Citrix ADC.

Using the rewrite or responder policies you can allowlist or blocklist the IP addresses/CIDR using which users can access your domain.

The following sections cover various ways you can allowlist or blocklist the IP addresses/CIDR using the rewrite or responder policies.

Allowlist IP addresses

Using a responder policy, you can allowlist IP addresses and silently drop the requests from the clients using IP addresses different from the allowlisted IP addresses.

Create a file named `allowlist-ip.yaml` with the following rewrite policy configuration:

```yaml
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
  name: allowlistip
spec:
  responder-policies:
    - servicenames:
        - frontend
      responder-policy:
        drop:
          respond-criteria: '!client.ip.src.TYPECAST_text_t.equals_any("allowlistip")'
          comment: 'Allowlist certain IP addresses'
    patset:
      - name: allowlistip
        values:
          - '10.xxx.170.xx'
          - '10.xxx.16.xx'
<!-- NeedCopy -->
```

You can also provide the IP addresses as a list:

```yaml
apiVersion: citrix.com/v1
```
Then, deploy the YAML file (allowlist-ip.yaml) using the following command:

```
kubectl create -f allowlist-ip.yaml
```

### Allowlist IP addresses and send 403 response to the request from clients not in the allowlist

Using a responder policy, you can allowlist a list of IP addresses and send the HTTP/1.1 403 Forbidden response to the requests from the clients using IP addresses different from the allowlisted IP addresses.

Create a file named `allowlist-ip-403.yaml` with the following rewrite policy configuration:
Then, deploy the YAML file (allowlist-ip-403.yaml) using the following command:

```
kubectl create -f allowlist-ip-403.yaml
```

**Allowlist a CIDR**

You can allowlist a CIDR using a responder policy. The following is a sample responder policy configuration to allowlist a CIDR:

```
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
  name: blocklistips1
spec:
  responder-policies:
    - servicenames:
    - frontend
      responder-policy:
        respondwith:
          http-payload-string: '"HTTP/1.1 403 Forbidden\r\n\n" + "Client: " + CLIENT.IP.SRC + "] is not authorized to access URL:" + HTTP.REQUEST.HTTP_URL_HTTP_URL_SAFE +"\n"
        respond-criteria: '"client.ip.src.IN_SUBNET(10.xxx.170.xx/24)"
        comment: 'Allowlist certain IPs'
<!--NeedCopy-->
Citrix ADC ingress controller

**Blocklist IP addresses**

Using a responder policy, you can blocklist IP addresses and silently drop the requests from the clients using the blocklisted IP addresses.

Create a file named `blocklist-ip.yaml` with the following responder policy configuration:

```yaml
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
  name: blocklistips
spec:
  responder-policies:
    - servicenames:
      - frontend
        responder-policy:
          respondwith: drop:
            respond-criteria: 'client.ip.src.TYPECAST_text_t.equals_any("blocklistips")'
            comment: 'Blocklist certain IPS'
  patset:
    - name: blocklistips
      values:
        - '10.xxx.170.xx'
        - '10.xxx.16.xx'
<!--NeedCopy-->`
```

Then, deploy the YAML file (`blocklist-ip.yaml`) using the following command:

```bash
kubectl create -f blocklist-ip.yaml
```

**Blocklist a CIDR**

You can blocklist a CIDR using a responder policy. The following is a sample responder policy configuration to blocklist a CIDR:

```yaml
apiVersion: citrix.com/v1
kind: rewritepolicy
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
Allowlist a CIDR and blocklist IP addresses

You can allowlist a CIDR and also blocklist IP addresses using a responder policy. The following is a sample responder policy configuration:

```yaml
metadata:
  name: blocklistips1
spec:
  responder-policies:
    - servicenames:
      responder-policy:
        respondwith:
          http-payload-string: "HTTP/1.1 403 Forbidden\n\nClient: " + CLIENT.IP.SRC + " is not authorized to access URL:" + HTTP.REQ.URL.HTTP_URL_SAFE +"\n"
        respond-criteria: 'client.ip.src.IN_SUBNET(10.xxx.170.xx/24)'
        comment: 'Blocklist certain IPs'
<!--NeedCopy-->
```

```yaml
apiVersion: citrix.com/v1
class: rewritepolicy
metadata:
  name: allowlistsub
spec:
  responder-policies:
    - servicenames:
      responder-policy:
        drop:
          respond-criteria: 'client.ip.src.TYPECAST_text_t.equals_any("blocklistips") || !client.ip.src.IN_SUBNET(10.xxx.170.xx/24)'
          comment: 'Allowlist a subnet and blocklist few IP's'

  patset:
    - name: blocklistips
      values:
        - '10.xxx.170.xx'
<!--NeedCopy-->
```
Blocklist a CIDR and allowlist IP addresses

You can blocklist a CIDR and also allowlist IP addresses using a responder policy. The following is a sample responder policy configuration:

```yaml
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
  name: blocklistips1
spec:
  responder-policies:
    - servicenames:
      - frontend
      responder-policy:
        drop:
        respond-criteria: 'client.ip.src.IN_SUBNET(10.xxx.170.xx/24) && !client.ip.src.TYPECAST_text_t.equals_any("allowlistips")'
        comment: 'Blocklist a subnet and allowlist few IP's'
    patset:
      - name: allowlistips
        values:
        - '10.xxx.170.xx'
        - '10.xxx.16.xx'

<!--NeedCopy-->
```

Interoperability with ExternalDNS

February 3, 2022

In a Kubernetes environment, you can expose your deployment using a service of type `LoadBalancer`. Also, an IP address can be assigned to the service using Citrix IPAM controller. The Citrix IPAM controller assigns IP address to the service from a defined pool of IP addresses. For more information, see [Expose services of type LoadBalancer with IP addresses assigned by the IPAM controller](#).

The service can be accessed using the IP address assigned by the IPAM controller and for service discovery you need to manually register the IP address to a DNS provider. If the IP address assigned to the service changes, the associated DNS record must be manually updated and the entire process becomes cumbersome. In such cases, you can use a `ExternalDNS` to keep the DNS records synchronized with your external entry points. Also, `ExternalDNS` allows you to control DNS records dynamically through Kubernetes resources in a DNS provider-agnostic way.
Citrix ADC ingress controller

For the ExternalDNS integration to work, the `external-dns.alpha.kubernetes.io/hostname` annotation must contain the host name.

**Note:**

For ExternalDNS to work, ensure that you add the annotation `external-dns.alpha.kubernetes.io/hostname` in the service specification and specify a host name for the service using the annotation.

To integrate with ExternalDNS:

1. Install the ExternalDNS with Infoblox provider.

   **Note:**
   The interoperability solution has been tested with Infoblox provider and the solution might work for other providers as well.

2. Specify the domain name in the ExternalDNS configuration.

3. In the service of type `LoadBalancer` specification, add the following annotation and specify a host name for the service using the annotation:

   ```
   external-dns.alpha.kubernetes.io/hostname
   ```

4. Deploy the service using the following command:

   ```
   kubectl create -f <service-name>.yml
   ```

**Using Citrix ADC credentials stored in a Vault server for the Citrix ingress controller**

April 4, 2022

In most organizations, tier 1 Citrix ADC Ingress devices and Kubernetes clusters are managed by separate teams. Usually, network administrators manage tier 1 Citrix ADC Ingress devices, while developers manage Kubernetes clusters. The Citrix ingress controller requires Citrix ADC credentials such as Citrix ADC user name and password to configure the Citrix ADC. You can specify Citrix ADC credentials as part of the Citrix ingress controller specification and store the ADC credentials as Kubernetes secrets. However, you can also store Citrix ADC credentials in a Vault server and pass credentials to the Citrix
Citrix ADC ingress controller

ingress controller to minimize any security risk. This topic provides information on how to use Citrix ADC credentials stored in a Vault server for the Citrix ingress controller.

The following diagram explains the steps for using Citrix ADC credentials which are stored in a Vault server with the Citrix ingress controller.

**Prerequisites**

Ensure that you have setup a Vault server and enabled key-value (KV) secret store. For more information, see Vault documentation.

**Using Citrix ADC credentials from a Vault server for the Citrix ingress controller**

Perform the following tasks to use Citrix ADC credentials from a Vault server for the Citrix ingress controller.

1. Create a service account for Kubernetes authentication.
2. Create a Key Vault secret and setup Kubernetes authentication on Vault server.
3. Leverage Vault Auto-Auth functionality to fetch Citrix ADC credentials for the Citrix ingress controller.

**Create a service account for Kubernetes authentication**

Create a service account for Kubernetes authentication by using the following steps:
1. Create a service account `cic-k8s-role` and provide the service account necessary permissions to access the Kubernetes TokenReview API by using the following command.

```
$ kubectl apply -f cic-k8s-role-service-account.yml
```

| 1 | serviceaccount/cic-k8s-role created |
| 2 | clusterrole.rbac.authorization.k8s.io/cic-k8s-role configured |
| 3 | clusterrolebinding.rbac.authorization.k8s.io/cic-k8s-role configured |
| 4 | clusterrolebinding.rbac.authorization.k8s.io/role-tokenreview-binding configured |

Following is a part of the sample `cic-k8s-role-service-account.yml` file.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: role-tokenreview-binding
namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:auth-delegator
subjects:
- kind: ServiceAccount
  name: cic-k8s-role
  namespace: default
```

2. Set the `VAULT_SA_NAME` environment variable to the name of the service account you have already created.

```
export VAULT_SA_NAME=$(kubectl get sa cic-k8s-role -o jsonpath="{.secrets["*"]['name']}")
```

3. Set the `SA_JWT_TOKEN` environment variable to the JWT of the service account that you used to access the TokenReview API.
export SA_JWT_TOKEN=$(kubectl get secret $VAULT_SA_NAME -o jsonpath="
  .data.token 
"
  | base64 --decode; echo)
<!--NeedCopy-->
2. Create a KV secret with Citrix ADC credentials at the secret/citrix-adc/ path.

   ```bash
   vault kv put secret/citrix-adc/credential username='<ADC username>' \ 
   password='<ADC password>' \ 
   ttl='30m'
   ```

3. Enable Kubernetes authentication at the default path (auth/kubernetes).

   ```bash
   # $ vault auth enable kubernetes
   ```

4. Specify how to communicate with the Kubernetes cluster.

   ```bash
   $ vault write auth/kubernetes/config \ 
   token_reviewer_jwt="$SA_JWT_TOKEN" \ 
   kubernetes_host="https://<K8S_CLUSTER_URL>:<API_SERVER_PORT>" \ 
   kubernetes_ca_cert="$SA_CA_CRT"
   ```

5. Create a role to map the Kubernetes service account to Vault policies and the default token TTL. This role authorizes the cic-k8s-role service account in the default namespace and maps the service account to the citrix-adc-kv-ro policy.

   ```bash
   $ vault write auth/kubernetes/role/cic-vault-example\ 
   bound_service_account_names=cic-k8s-role \ 
   bound_service_account_namespaces=default \ 
   policies=citrix-adc-kv-ro \ 
   ttl=24h
   ```

   **Note:**
   Authorization with Kubernetes authentication back-end is role based. Before a token is used for login, it must be configured as part of a role.

**Leverage Vault agent auto-authentication for the Citrix ingress controller**

Perform the following steps to leverage Vault auto-authentication.

© 1999–2022 Citrix Systems, Inc. All rights reserved.
1. Review the provided Vault Agent configuration file, `vault-agent-config.hcl`.

```hcl
exit_after_auth = true
pid_file = "/home/vault/pidfile"

auto_auth {
method "kubernetes" {
mount_path = "auth/kubernetes"
config = {
  role = "cic-vault-example"
}
}

sink "file" {
config = {
  path = "/home/vault/.vault-token"
}
}
}
```

**Note:**

The Vault agent **Auto-AUTH** is configured to use the Kubernetes authentication method enabled at the `auth/kubernetes` path on the Vault server. The Vault Agent uses the `cic-vault-example` role to authenticate.

The sink block specifies the location on disk where to write tokens. Vault Agent **Auto-AUTH** sink can be configured multiple times if you want Vault Agent to place the token into multiple locations. In this example, the sink is set to `/home/vault/.vault-token`.

2. Review the Consul template `consul-template-config.hcl` file.

```hcl
vault {
```
This template reads secrets at the `secret/citrix-adc/credential` path and sets the user name and password values.
If you are using KV store version 1, use the following template.

3. Create a Kubernetes config-map from `vault-agent-config.hcl` and `consul-template-config.hcl`.

```bash
kubectl create configmap example-vault-agent-config --from-file=./vault-agent-config.hcl --from-file=./consul-template-config.hcl
```
4. Create a Citrix ingress controller pod with Vault and consul template as init container `citrix-k8s-ingress-controller-vault.yaml`. Vault fetches the token using the Kubernetes authentication method and pass it on to a consul template which creates the `.env` file on shared volume. This token is used by the Citrix ingress controller for authentication with tier 1 Citrix ADC.

```bash
kubectl apply citrix-k8s-ingress-controller-vault.yaml
```

The `citrix-k8s-ingress-controller-vault.yaml` file is as follows:

```yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    name: cic-vault
  namespace: default
spec:
  containers:
    - name: cic-k8s-ingress-controller
      volumeMounts:
        - mountPath: /etc/citrix
          name: shared-data
      initContainers:
        - name: cic-vault
          image: vault
          imagePullPolicy: Always
          env:
            - name: VAULT_ADDR
              value: <VAULT URL>
	image: in-docker-reg.eng.citrite.net/cpx-dev/kumar-cic:latest
  initContainers:
    - name: cic-k8s-ingress-controller
      image: in-docker-reg.eng.citrite.net/cpx-dev/kumar-cic:latest
      initContainers:
        - agent
        - --config=/etc/vault/vault-agent-config.hcl
        - --log-level=debug
        - name: cilium
          value: <Tier 1 ADC IP-ADDRESS>
          env:
            - name: NS_IP
              value: <Tier 1 ADC IP-ADDRESS>
            - name: EULA
              value: "yes"
```
If the configuration is successful, the Vault server fetches a token and passes it on to a Consul template.
Citrix ADC ingress controller

container. The Consul template uses the token to read Citrix ADC credentials and write it as an environment variable in the path /etc/citrix/.env. The Citrix ingress controller uses these credentials for communicating with the tier 1 Citrix ADC.

Verify that the Citrix ingress controller is running successfully using credentials fetched from the Vault server.

How to use Kubernetes secrets for storing Citrix ADC credentials

February 3, 2022

In most organizations, Tier 1 Citrix ADC Ingress devices and Kubernetes clusters are managed by separate teams. The Citrix ingress controller requires Citrix ADC credentials such as Citrix ADC user name and password to configure the Citrix ADC. Usually, Citrix ADC credentials are specified as environment variables in the Citrix ingress Controller pod specification. But, another secure option is to use Kubernetes secrets to store the Citrix ADC credentials.

This topic describes how to use Kubernetes secrets to store the ADC credentials and various ways to provide the credentials stored as secret data for the Citrix ingress controller.

Create a Kubernetes secret

Perform the following steps to create a Kubernetes secret.

1. Create a file adc-credential-secret.yaml which defines a Kubernetes secret YAML with Citrix ADC user name and password in the data section as follows.

```
apiVersion: v1
kind: Secret
metadata:
  name: adc-credential
data:
  username: <ADC user name>
  password: <ADC password>
```

2. Apply the adc-credential-secret.yaml file to create a secret.

```
kubectl apply -f adc-credential-secret.yaml
```
Alternatively, you can also create the Kubernetes secret using `--from-literal` option of the `kubectl` command as shown as follows:

```bash
kubectl create secret generic adc-credentials --from-literal=
  username=<username> --from-literal=password=<password>
```

Once you have created a Kubernetes secret, you can use one of the following options to use the secret data in the Citrix ingress controller pod specification.

- Use secret data as environment variables in the Citrix ingress controller pod specification
- Use a secret volume mount to pass credentials to the Citrix ingress controller

**Use secret data as environment variables in the Citrix ingress controller pod specification**

You can use secret data from the Kubernetes secret as the values for the environment variables in the Citrix ingress controller deployment specification.

A snippet of the YAML file is shown as follows.

```yaml
- name: "NS_USER"
  valueFrom:
    secretKeyRef:
      name: adc-credentials
      key: username
  # Set user password for Nitro
- name: "NS_PASSWORD"
  valueFrom:
    secretKeyRef:
      name: adc-credentials
      key: password
```

Here is an example of the Citrix ingress controller deployment with value of environment variables sourced from the secret object.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cic-k8s-ingress-controller
spec:
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
use a secret volume mount to pass credentials to the Citrix ingress controller

Alternatively, you can also use a volume mount using the secret object as a source for the Citrix ADC credentials. The Citrix ingress controller expects the secret to be mounted at path /etc/citrix and it looks for the credentials in files username and password.
You can create a volume from the secret object and then mount the volume using `volumeMounts` at `/etc/citrix` as shown in the following deployment example.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cic-k8s-ingress-controller
spec:
  selector:
    matchLabels:
      app: cic-k8s-ingress-controller
  replicas: 1
  template:
    metadata:
      name: cic-k8s-ingress-controller
      labels:
        app: cic-k8s-ingress-controller
      annotations:
        spec:
          serviceAccountName: cic-k8s-role
          containers:
            - name: cic-k8s-ingress-controller
              image: <image location>
              env:
                - name: "NS_IP"
                  value: "x.x.x.x"
                - name: "EULA"
                  value: "yes"
              volumeMounts:
                - name: secret-volume
                  mountPath: /etc/citrix
                  imagePullPolicy: Always
              volumes:
                - name: secret-volume
                  secret:
                    secretName: adc-credentials
<!--NeedCopy-->
Use Citrix ADC credentials stored in a Hashicorp Vault server

You can also use the Citrix ADC credentials stored in a Hashicorp Vault server for the Citrix ingress controller and push the credentials through a sidecar container.

For more information, see Using Citrix ADC credentials stored in a Vault server.

How to load balance Ingress traffic to TCP or UDP based application

February 3, 2022

In a Kubernetes environment, an Ingress is an object that allows access to the Kubernetes services from outside the Kubernetes cluster. Standard Kubernetes Ingress resources assume that all the traffic is HTTP-based and do not cater to non-HTTP based protocols such as, TCP, TCP-SSL, and UDP. Hence, you cannot expose critical applications based on layer 7 protocols such as DNS, FTP, or LDAP using the standard Kubernetes Ingress.

Citrix provides a solution using Ingress annotations to load balance TCP or UDP based Ingress traffic. When you specify these annotations in the Ingress resource definition, the Citrix ingress controller configures the Citrix ADC to load balance TCP or UDP based Ingress traffic.

You can use the following annotations in your Kubernetes Ingress resource definition to load balance the TCP or UDP based Ingress traffic:

- **ingress.citrix.com/insecure-service-type**: The annotation enables L4 load balancing with TCP, UDP, or ANY as protocol for Citrix ADC.
- **ingress.citrix.com/insecure-port**: The annotation configures the TCP port. The annotation is helpful when micro service access is required on a non-standard port. By default, port 80 is configured.

For more information about annotations, see the annotations page.

You can also use the standard Kubernetes solution of creating a service of type LoadBalancer with Citrix ADC. You can find out more about Service Type LoadBalancer in Citrix ADC.

Sample: Ingress definition for TCP-based Ingress.

```plaintext
1 apiVersion: networking.k8s.io/v1
2 kind: Ingress
3 metadata:
4   annotations:
5     ingress.citrix.com/insecure-port: "6379"
6     ingress.citrix.com/insecure-service-type: "tcp"
7     kubernetes.io/ingress.class: "guestbook"
```
Sample: Ingress definition for UDP-based Ingress. The following is a sample for Citrix ingress controller version 1.1.1:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    ingress.citrix.com/insecure-port: "5084"
    ingress.citrix.com/insecure-service-type: "udp"
name: udp-ingress
spec:
defaultBackend:
  service:
    name: frontend
    port:
      name: udp-53  # Service port name defined in the service definition
<!--NeedCopy-->
Citrix ADC ingress controller

Sample: Ingress definition for UDP-based Ingress. The following is a sample for Citrix ingress controller version 1.5.25:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    ingress.citrix.com/insecure-port: "5084"
    ingress.citrix.com/insecure-service-type: "udp"
name: udp-ingress
spec:
defaultBackend:
  service:
    name: frontend
    port:
      number: 53
<!--NeedCopy-->
```

Load balance Ingress traffic based on TCP over SSL

Citrix ingress controller provides an 'ingress.citrix.com/secure-service-type: ssl_tcp' annotation that you can use to load balance Ingress traffic based on TCP over SSL.

Sample: Ingress definition for TCP over SSL based Ingress.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    ingress.citrix.com/secure-service-type: "ssl_tcp"
    ingress.citrix.com/secure-backend: '{
      "frontendcoldrinks":true 
    }
    kubernetes.io/ingress.class: "colddrink"
name: colddrinks-ingress
spec:
defaultBackend:
<!--NeedCopy-->
```
Monitor and improve the performance of your TCP or UDP based applications

Application developers can closely monitor the health of TCP or UDP based applications through rich monitors (such as TCP-ECV, UDP-ECV) in Citrix ADC. The ECV (extended content validation) monitors help in checking whether the application is returning expected content or not.

Also, the application performance can be improved by using persistence methods such as Source IP. You can use these Citrix ADC features through Smart Annotations in Kubernetes. The following is one such example:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    ingress.citrix.com/csvserver: '{
      "l2conn":"on"
    }'
    ingress.citrix.com/frontend-ip: "192.168.1.1"
    ingress.citrix.com/insecure-port: "80"
    ingress.citrix.com/lbvserver: '{
      "mongodb-svc":{ "lbmethod":"SRCIPDESTIPHASH" }
    }
    ingress.citrix.com/monitor: '{
      "mongodbsvc":{
        "type":"tcp-ecv"
      }
    }
  name: mongdb
spec:
  rules:
    - host: mongodb.beverages.com
```
For more information on the different deployment options supported by the Citrix ingress controller, see Deployment topologies.

For more information on deploying the Citrix ingress controller:

- Deploy the Citrix ingress controller using YAML
- Deploy the Citrix ingress controller using Helm charts

**How to set up dual-tier deployment**

February 15, 2022

In a dual-tier deployment, Citrix ADC VPX or MPX is deployed outside the Kubernetes cluster (Tier-1) and Citrix ADC CPXs are deployed inside the Kubernetes cluster (Tier-2).

Citrix ADC MPX or VPX devices in Tier-1 proxy the traffic (North-South) from the client to Citrix ADC CPXs in Tier-2. The Tier-2 Citrix ADC CPX then routes the traffic to the microservices in the Kubernetes cluster. The Citrix ingress controller deployed as a standalone pod configures the Tier-1 Citrix ADC. And, the sidecar Citrix ingress controller in one or more Citrix ADC CPX pods configures the associated Citrix ADC CPX in the same pod.

The Dual-Tier deployment can be set up on Kubernetes in bare metal environment or on public clouds such as, AWS, GCP, or Azure.

The following diagram shows a Dual-Tier deployment:
Citrix ADC ingress controller

Setup process

The Citrix ingress controller repo provides a sample Apache microservice and manifests for Citrix ADC CPX for Tier-2, ingress object for Tier-2 Citrix ADC CPX, Citrix ingress controller, and an ingress object for Tier-1 Citrix ADC for demonstration purpose. These samples are used in the setup process to deploy a dual-tier topology.

Perform the following:

1. Create a Kubernetes cluster in cloud or on-premises. The Kubernetes cluster in cloud can be a managed Kubernetes (for example: GKE, EKS, or AKS) or a custom created Kubernetes deployment.

2. Deploy Citrix ADC MPX or VPX on a multi-NIC deployment mode outside the Kubernetes cluster.
   - For instructions to deploy Citrix ADC MPX, see Citrix ADC documentation.
   - For instructions to deploy Citrix ADC VPX, see Deploy a Citrix ADC VPX instance.

Perform the following after you deploy Citrix ADC VPX or MPX:

   a) Configure an IP address from the subnet of the Kubernetes cluster as SNIP on the Citrix ADC. For information on configuring SNIPs in Citrix ADC, see Configuring Subnet IP Addresses (SNIPs).
Citrix ADC ingress controller

b) Enable management access for the SNIP that is the same subnet of the Kubernetes cluster. The SNIP should be used as **NS_IP** variable in the Citrix ingress controller YAML file to enable Citrix ingress controller to configure the Tier-1 Citrix ADC.

**Note:**

It is not mandatory to use SNIP as **NS_IP**. If the management IP address of the Citrix ADC is reachable from Citrix ingress controller then you can use the management IP address as **NS_IP**.

c) In cloud deployments, enable **MAC-Based Forwarding mode** on the Tier-1 Citrix ADC VPX. As Citrix ADC VPX is deployed in multi-NIC mode, it would not have the return route to reach the POD CNI network or the Client network. Hence, you need to enable MAC-Based Forwarding mode on the Tier-1 Citrix ADC VPX to handle this scenario.

d) Create a **Citrix ADC system user account** specific to Citrix ingress controller. Citrix ingress controller uses the system user account to automatically configure the Tier-1 Citrix ADC.

e) Configure your on-premises firewall or security groups on your cloud to allow inbound traffic to the ports required for Citrix ADC. The Setup process uses port 80 and port 443, you can modify these ports based on your requirement.

3. Deploy a sample microservice. Use the following command:

```bash
```

4. Deploy Citrix ADC CPX as Tier-2 ingress. Use the following command:

```bash
```

5. Create an ingress object for the Tier-2 Citrix ADC CPX. Use the following command:

```bash
```

6. Deploy the Citrix ingress controller for Tier-1 Citrix ADC. Perform the following:
a) Download the Citrix ingress controller manifest file. Use the following command:

```bash
```

b) Edit the Citrix ingress controller manifest file and enter the values for the following environmental variables:

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Mandatory or Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS_IP</td>
<td>Mandatory</td>
<td>The IP address of the Citrix ADC appliance. For more details, see Prerequisites.</td>
</tr>
<tr>
<td>NS_USER and NS_PASSWORD</td>
<td>Mandatory</td>
<td>The user name and password of the Citrix ADC VPX or MPX appliance used as the Ingress device. For more details, see Prerequisites.</td>
</tr>
<tr>
<td>EULA</td>
<td>Mandatory</td>
<td>The End User License Agreement. Specify the value as Yes.</td>
</tr>
<tr>
<td>LOGLEVEL</td>
<td>Optional</td>
<td>The log levels to control the logs generated by Citrix ingress controller. By default, the value is set to DEBUG. The supported values are: CRITICAL, ERROR, WARNING, INFO, and DEBUG. For more information, see Log Levels.</td>
</tr>
<tr>
<td>NS_PROTOCOL and NS_PORT</td>
<td>Optional</td>
<td>Defines the protocol and port that must be used by Citrix ingress controller to communicate with Citrix ADC. By default, Citrix ingress controller uses HTTPS on port 443. You can also use HTTP on port 80.</td>
</tr>
</tbody>
</table>
Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Mandatory or Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ingress-classes</td>
<td>Optional</td>
<td>If multiple ingress load balancers are used to load balance different ingress resources. You can use this environment variable to specify Citrix ingress controller to configure Citrix ADC associated with specific ingress class. For information on Ingress classes, see Ingress class support.</td>
</tr>
<tr>
<td>NS_VIP</td>
<td>Optional</td>
<td>Citrix ingress controller uses the IP address provided in this environment variable to configure a virtual IP address to the Citrix ADC that receives Ingress traffic. <strong>Note:</strong> NS_VIP takes precedence over the frontend-ip annotation.</td>
</tr>
</tbody>
</table>

c) Deploy the updated Citrix ingress controller manifest file. Use the following command:

```bash
kubectl create -f tier-1-vpx-cic.yaml
```

7. Create an ingress object for the Tier-1 Citrix ADC. Use the following command:

```bash
```

8. Update DNS server details in the cloud or on-premises to point your website to the VIP of the Tier-1 Citrix ADC.

   For example: **citrix-ingress.com 192.250.9.1**

   Where 192.250.9.1 is the VIP of the Tier-1 Citrix ADC and **citrix-ingress.com** is the microservice running in your Kubernetes cluster.
Set up dual-tier deployment using one step deployment manifest file

For easy deployment, the Citrix ingress controller repo includes an all-in-one deployment manifest. You can download the file and update it with values for the following environmental variables and deploy the manifest file.

**Note:**
Ensure that you have completed step 1–2 in the Setup process.

Perform the following:

1. Download the all-in-one deployment manifest file. Use the following command:

   ```
   ```

2. Edit the all-in-one deployment manifest file and enter the values for the following environmental variables:

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Mandatory or Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS_IP</td>
<td>Mandatory</td>
<td>The IP address of the Citrix ADC appliance. For more details, see Prerequisites.</td>
</tr>
<tr>
<td>NS_USER and NS_PASSWORD</td>
<td>Mandatory</td>
<td>The user name and password of the Citrix ADC VPX or MPX appliance used as the Ingress device. For more details, see Prerequisites.</td>
</tr>
<tr>
<td>EULA</td>
<td>Mandatory</td>
<td>The End User License Agreement. Specify the value as Yes.</td>
</tr>
<tr>
<td>Environment Variable</td>
<td>Mandatory or Optional</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>LOGLEVEL</td>
<td>Optional</td>
<td>The log levels to control the logs generated by Citrix ingress controller. By default, the value is set to DEBUG. The supported values are: CRITICAL, ERROR, WARNING, INFO, and DEBUG. For more information, see <a href="/en-us/citrix-k8s-ingress-controller/configure.log-level.html">Log Levels</a></td>
</tr>
<tr>
<td>NS_PROTOCOL and NS_PORT</td>
<td>Optional</td>
<td>Defines the protocol and port that must be used by Citrix ingress controller to communicate with Citrix ADC. By default, Citrix ingress controller uses HTTPS on port 443. You can also use HTTP on port 80.</td>
</tr>
<tr>
<td>ingress-classes</td>
<td>Optional</td>
<td>If multiple ingress load balancers are used to load balance different ingress resources. You can use this environment variable to specify Citrix ingress controller to configure Citrix ADC associated with specific ingress class. For information on Ingress classes, see <a href="/en-us/citrix-k8s-ingress-controller/configure/ingress-classes.html">Ingress class support</a></td>
</tr>
<tr>
<td>Environment Variable</td>
<td>Mandatory or Optional</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>NS_VIP</td>
<td>Optional</td>
<td>Citrix ingress controller uses the IP address provided in this environment variable to configure a virtual IP address to the Citrix ADC that receives Ingress traffic. <strong>Note:</strong> NS_VIP acts as a fallback when the <code>frontend-ip</code> annotation is not provided in Ingress yaml. Not supported for Type Loadbalancer service.</td>
</tr>
</tbody>
</table>

3. Deploy the updated all-in-one deployment manifest file. Use the following command:

```
kubectl create -f all-in-one-dual-tier-demo.yaml
```

**Horizontal pod autoscaler for Citrix ADC CPX with custom metrics**

February 3, 2022

While deploying workloads in a Kubernetes cluster for the first time, it is difficult to exactly predict the resource requirements and how those requirements might change in a production environment. Using Horizontal pod autoscaler (HPA), you can automatically scale the number of pods in your workload based on different metrics like actual resource usage. HPA is a resource provided by Kubernetes which scales Kubernetes based resources like deployments, replicasets, and replication controllers.

Traditionally, HPA gets the required metrics from a metrics server. It then periodically adjusts the number of replicas in a deployment to match the observed average metrics to the target you specify.
Citrix provides a custom-metric based HPA solution for Citrix ADC CPX. By default, the metrics server only gives CPU and memory metrics for a pod. Citrix ADC provides a rich set of in-built metrics for analyzing application performance and based on these metrics you can take a better autoscaling judgment. A custom metric based HPA is a better solution like autoscaling based on HTTP request rate, SSL transactions, or ADC bandwidth.

**Citrix ADC CPX HPA solution**

Citrix ADC CPX HPA solution consists of the following components:

- Citrix ADC VPX: Citrix ADC VPX or MPX is deployed at Tier-1 and load balances the client requests among the Citrix ADC CPX pods inside the cluster.

- Citrix ADC CPX: Citrix ADC CPX deployed inside the cluster acts as a Tier-2 load balancer for the endpoint application pods. The Citrix ADC CPX pod is running along with the Citrix ingress controller and Citrix ADC metric exporter as sidecars.

- Citrix ingress controller: The **Citrix ingress controller** is an ingress controller which is built around the Kubernetes Ingress and automatically configures Citrix ADC based on the Ingress resource configuration. The Citrix ingress controller deployed as a stand-alone pod configures the Citrix ADC VPX and other instances configures Citrix ADC CPXs.

- Citrix ADC metrics exporter: The **Citrix ADC metrics exporter** exports the application performance metrics to the open-source monitoring system Prometheus. The Citrix ADC Metrics Exporter collects metrics from Citrix ADC CPX and exposes it in a format that Prometheus can understand.
Citrix ADC ingress controller

- Promotheus: Prometheus is an open-source systems monitoring and alerting toolkit. Prometheus is used to collect metrics from Citrix ADC CPXs and expose them using a Prometheus adapter which is queried by the HPA controller to keep a check on metrics.

- Prometheus adapter: Prometheus adapter contains an implementation of the Kubernetes resource metrics API and custom metrics API. This adapter is suitable for use with the autoscaling/v2 HPA in Kubernetes version 1.6+. It can also replace the metrics server on clusters that already run Prometheus and collect the appropriate metrics.

The following diagram is a visual representation of how the Citrix ADC CPX HPA solution works.

![Diagram of the Citrix ADC CPX HPA solution](image)

The Tier-1 Citrix ADC VPX load balances the Citrix ADC CPXs at Tier-2. Citrix ADC CPXs load balance applications. Other components like Prometheus, Prometheus-adapter, and an HPA controller is also deployed.

The HPA controller keeps polling the Prometheus-adapter for custom metrics like HTTP requests rate or bandwidth. Whenever the limit defined by the user in the HPA is reached, the HPA controller scales the Citrix ADC CPX deployment and creates another Citrix ADC CPX pod to handle the load.

**Deploy Citrix ADC CPX HPA solution**

Perform the following steps to deploy the Citrix ADC CPX HPA solution.

1. Clone the citrix-k8s-ingress-controller repository from GitHub using the following command.

   ```bash
   ```
After cloning, change your directory to the HPA folder with the following command.

```bash
cd citrix-k8s-ingress-controller/blob/master/docs/how-to/hpa
```

2. From the HPA directory, open and edit the `values.sh` file and set the following values for Citrix ADC VPX.

- `VPX_IP`: IP address of the Citrix ADC VPX
- `VPX_PASSWORD`: The password of the nsroot user on the Citrix ADC VPX
- `VIRTUAL_IP_VPX`: The IP address on which the sample guestbook application is accessed.

3. Create all the required resources by running the `create_all.sh` file.

```bash
./create_all.sh
```

This step creates the following resources:

- Prometheus and Grafana for monitoring
- Citrix ADC CPX with the Citrix ingress controller and metrics exporter as sidecars
- Citrix ingress controller as a stand-alone pod to configure Citrix ADC VPX
- A sample guestbook application
- HPA controller for monitoring the Citrix ADC CPX autoscale deployment
- Prometheus adapter for exposing the custom metrics

4. Add an entry in the `hosts` file. The route must be added in the `hosts` file to route traffic for the guestbook application to the Citrix ADC VPX virtual IP address.
   For most Linux distros, the `hosts` file is present in the `/etc` folder.

5. Send some generated traffic and verify the Citrix ADC CPX autoscale deployment.

   The Citrix ADC CPX deployment HPA has been configured in such a way that when the average HTTP requests rate of the Citrix ADC CPX goes above 20 requests per second, it autoscales. You can use the following scripts provided in the HPA folder for sending traffic:

   - `16_curl.sh` - Send 16 HTTP requests per second (lesser than the threshold)
   - `30_curl.sh` - Send 30 HTTP requests per second (greater than the threshold)

   a. Run the `16_curl.sh` script to send 16 HTTP requests per second to the Citrix ADC CPX.

```bash
./16_curl.sh
```
Citrix ADC ingress controller

The following diagram a Grafana dashboard which displays HTTP requests per second.

![Grafana Dashboard](image)

The following output shows the HPA state with 16 HTTP RPS.

<table>
<thead>
<tr>
<th>NAME</th>
<th>REFERENCE</th>
<th>TARGETS</th>
<th>MINPODS</th>
<th>MAXPODS</th>
<th>REPLICAS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpx-builtin</td>
<td>Deployment/cpx-builtin</td>
<td>16/28</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>4m9s</td>
</tr>
</tbody>
</table>

b. Run the `30_curl.sh` script to send 30 HTTP requests per second to Citrix ADC CPX.

```
./30_curl.sh
```

When you run this script, the threshold of 20 requests that was set has been crossed and the Citrix ADC CPX deployment autoscales from one pod to two pods. The average value of the metric `HTTP request rate` also goes down from 30 to 15 as there are two Citrix ADC CPX pods.

The following output shows the state of HPA when the target is crossed.

<table>
<thead>
<tr>
<th>NAME</th>
<th>REFERENCE</th>
<th>TARGETS</th>
<th>MINPODS</th>
<th>MAXPODS</th>
<th>REPLICAS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpx-builtin</td>
<td>Deployment/cpx-builtin</td>
<td>30/28</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>8m4s</td>
</tr>
</tbody>
</table>

The following output shows that the number of replicas of Citrix ADC CPX have gone up to 2 and the average value of HTTP RPS comes down to 15.

<table>
<thead>
<tr>
<th>NAME</th>
<th>REFERENCE</th>
<th>TARGETS</th>
<th>MINPODS</th>
<th>MAXPODS</th>
<th>REPLICAS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpx-builtin</td>
<td>Deployment/cpx-builtin</td>
<td>15/28</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>9m24s</td>
</tr>
</tbody>
</table>

The following diagram shows a Grafana dashboard with two Citrix ADC CPXs load balancing the traffic.

![Grafana Dashboard](image)
Citrix ADC ingress controller

6. Clean up by executing the `delete_all.sh` script.

```
./delete_all.sh
```

**Note:**

If the Tier-1 Citrix ADC VPX is not present, use NodePort to expose the Citrix ADC CPX service.

**Deploy Direct Server Return**

February 8, 2022

In a typical load-balanced system, a load balancer acts as a mediator between web servers and clients. Incoming client requests are received by the load balancer and it passes the requests to the appropriate server with slight modifications to the data packets. The server responds to the load balancer with the required data and then the load balancer forwards the response to the client.

In a Direct Server Return (DSR) deployment, load balancer forwards the client request to the server, but the back-end server directly sends the response to the client. The use of different network paths for request and response helps to avoid extra hops and reduces the latency. Because the server directly responds to the client, DSR speeds up the response time between the client and the server and also removes some extra load from the load balancer. Using DSR is a transparent way to achieve increased network performance for your applications with little to no infrastructure changes. For more information on DSR using Citrix ADC, see the [Citrix ADC documentation](https://docs.citrix.com).

DSR solution is useful in the following situations:

- While handling applications that deliver video streaming where low latency (response time) matters.
- Where intelligent load balancing is not required
- When the output capacity of the load-balancer can be the bottleneck

However, when you use the DSR advanced layer 7 load balancing features are not supported.

**DSR network topology for Kubernetes using Citrix ADC**

In this topology, there is an external load-balancer (Tier-1 ADC) that distributes the traffic to the ingress ADC (Tier 2 ADC) deployed inside the Kubernetes cluster over an overlay (L3 DSR IPIP). Tier-2 ADC picks up the packet, decapsulate the packet, and performs load balancing among services. The Tier-2 ADC sends the return traffic from service to the client instead of sending it via Tier-1 ADC.
Deploying DSR for cloud native applications using Citrix ADC

Perform the steps in the following sections to deploy DSR for applications deployed on the Kubernetes cluster.

Deploy Citrix ADC CPX as Tier-2 ADC

This section contains steps to create configurations required on the ingress device for DSR topology.

1. Create a namespace for DSR using the following command:

```
kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/example/dsr/KubernetesConfig/dsr_namespace.yaml
```

2. Create a ConfigMap using the following command.

```
kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/example/dsr/KubernetesConfig/cpx_config.yaml -n dsr
```

**Note:**

In this example, the Citrix Node Controller network is configured as 192.168.1.0/24. Hence, the command to create IP tunnel is provided as `add iptunnel dsr 192.168.1.254`
Deploying a sample application on the Kubernetes cluster

Perform the steps in this section to deploy a sample application on Kubernetes cluster.

1. Deploy the guestbook application using the following command.

```bash
kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/example/dsr/KubernetesConfig/citrix-k8s-cpx-ingress.yml -n dsr
```

2. Expose the guestbook application using Ingress.

   a) Download the guestbook ingress YAML file using the following command.

```bash
wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/example/dsr/KubernetesConfig/guestbook-all-in-one.yaml
```

   b) Edit and provide the DSR IP or public IP address through which you access your application using the `ingress.citrix.com/frontend-ip:` annotations.

```bash
ingress.citrix.com/frontend-ip: "<ip-address>"
```

   c) Save the YAML file and deploy the Ingress resource using the following command.

```bash
kubectl apply -f guestbook-ingress.yaml -n dsr
```
Citrix ADC ingress controller

Establish network connectivity between Tier-1 and Tier-2 ADCs

Perform the steps in this section to establish network connectivity between Tier-1 and Tier-2 ADCs.

1. Download the YAML to deploy Citrix Node Controller using the following command.

   ```bash
   wget https://raw.githubusercontent.com/citrix/citrix-k8s-node-controller/master/deploy/citrix-k8s-node-controller.yaml
   ```

2. Edit the YAML file and provide the values for NS_IP, NS_USER, NS_PASSWORD, and REMOTE_VTEPIP arguments. For detailed information, see Citrix Node Controller.

3. Save the YAML file and deploy the Citrix Node Controller.

   ```bash
   kubectl create -f citrix-k8s-node-controller.yaml -n dsr
   ```

Deploy the Citrix ingress controller for Tier-1 ADC and expose Citrix ADC CPX as a service

Perform the following steps to deploy the Citrix ingress controller as a stand-alone pod and create an Ingress resource for Tier-2 Citrix ADC CPX.

1. Download the Citrix Ingress Controller YAML file using the following command.

   ```bash
   wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/example/dsr/KubernetesConfig/citrix-k8s-ingress-controller.yaml
   ```

2. Edit the YAML file and update the following values for Citrix Ingress Controller.

   - NS_IP
   - NS_USER
   - NS_PASSWORD

   For more information, see Deploy the Citrix ingress controller using YAML.

3. Save the YAML file and deploy the Citrix Ingress Controller.

   ```bash
   kubectl create -f citrix-k8s-ingress-controller.yaml -n dsr
   ```
4. Create DSR configuration on Tier-1 ADC by creating an ingress resource for the Tier-2 Citrix ADC CPX.

```bash
wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/example/dsr/KubernetesConfig/vpx-ingress.yaml
```

5. Edit the YAML file and provide the DSR or public IP address through which user access your application using the `ingress.citrix.com/frontend-ip:` annotation. This IP address must be same as the IP address you have specified in step 2.

```bash
kubectl apply -f vpx-ingress.yaml -n dsr
```

**Test the DSR deployment**

To test the DSR deployment, access the application from a browser using the IP address specified for the `ingress.citrix.com/frontend-ip:` annotation. A guestbook page is populated.

A sample output is given as follows:

**Guestbook**

<table>
<thead>
<tr>
<th>Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Submit**

**Troubleshooting**

When you test the application, it might not populate any pages even though all the required configurations are created. This is because of `rp_filter<--NeedCopy-->` rules on the host. If you experience such an issue, use the following commands on all the hosts to disable the rules.

```bash
1. sysctl -w net.ipv4.conf.all_rp_filter=0
2. sysctl -w net.ipv4.conf.cni0_rp_filter=0
3. sysctl -w net.ipv4.conf.eth0_rp_filter=0
4. sysctl -w net.ipv4.conf.cni0_rp_filter=0
5. sysctl -w net.ipv4.conf.default_rp_filter=0
```
Support for admission controller webhooks

February 3, 2022

Admission controllers are powerful tools for intercepting requests to the Kubernetes API server prior to the persistence of the object. Using Kubernetes admission controllers, you can define and customize what is allowed to run on your cluster. Hence, they are useful tools for cluster administrators to deploy preventive security controls on your cluster. But you need to compile the admission controllers into the kube-apiserver binary and they offer limited flexibility.

To overcome this limitation, Kubernetes supports dynamic admission controllers that can be developed as extensions and run as webhooks configured at runtime.

Using the Admission controller webhooks Kubernetes cluster administrators can create additional plug-ins to the admission chain of API server without recompiling them. Admission controller webhooks can be executed whenever a resource is created, updated, or deleted.

You can define two types of admission controller webhooks:

- validating admission webhook
- mutating admission webhook

Mutating admission webhooks are invoked first, and they can modify objects sent to the API server to enforce custom defaults. Once all the object modifications are complete, and the incoming object is validated by the API server, validating admission webhooks are invoked. Validating admission hooks process requests and accept or reject requests to enforce custom policies.

The following diagram explains how the admission controller webhook works:

Here are some of the scenarios where admission webhooks are useful:
Citrix ADC ingress controller

- To mandate a reasonable security baseline across an entire namespace or cluster mandating. For example, disallowing containers from running as root or making sure the container’s root filesystem is always mounted as read-only.

- To enforce the adherence to certain standard and practices for labels, annotations, or resource limits. For example, enforce label validation on different objects to ensure proper labels are being used for various objects.

- To validate the configuration of the objects running in the cluster and prevent any obvious misconfigurations from hitting your cluster. For example, to detect and fix images deployed without semantic tags.

How to apply admission controllers

Writing an admission controller for each specific use case is not scalable and it helps to have a system that supports multiple configurations covering different resource types and fields. You can use Open policy agent (OPA) and Gatekeeper to implement a customizable admission webhook for Kubernetes.

OPA is an open source, general-purpose policy engine that unifies policy enforcement across the stack. Gatekeeper is a customizable validating webhook that enforces CRD-based policies executed by OPA.

Gatekeeper introduces the following functionalities

- An extensible, parameterized policy library
- Native Kubernetes CRDs for instantiating the policy library (constraints)
- Native Kubernetes CRDs for extending the policy library (constraint templates)
Citrix ADC ingress controller

- Audit functionality

Writing and deploying an admission controller webhook

Prerequisites

- Kubernetes 1.14.0 or later with the admissionregistration.k8s.io/v1beta1 API enabled. You can verify whether the API is enabled by using the following command:

```bash
kubectl api-versions | grep admissionregistration.k8s.io/v1beta1
```

The following output indicates that the API is enabled:

```
admissionregistration.k8s.io/v1beta1
```

- The mutating admission webhook and validate admission webhook admission controllers should be added and listed in the correct order in the admission-control flag of `kube-apiserver`.

With Minikube, you can perform this task by starting Minikube with the following command:

```bash
minikube start --extra-config=apiserver.enable-admission-plugins=NamespaceLifecycle,LimitRanger,ServiceAccount,DefaultStorageClass,DefaultTolerationSeconds,NodeRestriction,MutatingAdmissionWebhook,ValidatingAdmissionWebhook`
```

- Ensure that you have cluster administrator permissions.

```bash
kubectl create clusterrolebinding cluster-admin-binding --clusterrole cluster-admin --user <YOUR USER NAME>
```

Mutating admission webhook configuration

For more information on mutating admission webhook configuration, see `ingress-admission-webhook`.

The following use cases are covered in the mutating admission webhook example:

- Update port in an Ingress based on the Ingress name
- Enable secure back-end forcefully based on a namespace
Validating admission webhook configuration using Gatekeeper

Gatekeeper uses a CRD that allows you to create constraints as Kubernetes resources. This CRD is called a ConstraintTemplate in Gatekeeper. The schema of the constraint allows an administrator to fine-tune the behavior of a constraint, similar to arguments to a function. Constraints are used to inform Gatekeeper that the administrator wants a constraint template to be enforced, and how.

You can apply various policies using constraint templates. Various examples are listed at the Gatekeeper library.

Deploying a sample policy

Perform the following steps to deploy HttspOnly as a sample policy using Gatekeeper. The HttspOnly policy allows only an Ingress configuration with HTTPS.

1. Install Gatekeeper using the following command.

   ```bash
   # kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/master/deploy/gatekeeper.yaml
   ```

   You can verify the installation using the following command.

   ```bash
   kubectl get crd | grep -i constraintsonstrainttemplates.templates.gatekeeper.sh
   ```

   You can check all the constraint templates using the following command:

   ```bash
   kubectl get constrainttemplates.templates.gatekeeper.sh
   ```

2. Apply the HttspOnly constraint template.

   ```bash
   kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/docs/how-to/webhook/httspOnly/template.yaml
   ```
3. Apply a constraint to enforce the `httpsonly` policy.

```
kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/docs/how-to/webhook/httpsonly/constraint.yaml
```

4. Deploy a sample Ingress which violates the policy to verify the policy. It should display an error while creating the Ingress.

```
kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/docs/how-to/webhook/httpsonly/bad-example-ingress.yaml
```

```
Error from server ([denied by ingress-httpsonly] Ingress must be https. tls configuration is required for test-ingress): error when creating "ingress.yaml": admission webhook "validation.gatekeeper.sh" denied the request: [denied by ingress-httpsonly] Ingress must be https. tls configuration is required for test-ingress
```

5. Now, deploy an Ingress which has the required TLS section in Ingress.

```
# kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/docs/how-to/webhook/httpsonly/good-example-ingress.yaml
```

```
ingress.networking.k8s.io/test-ingress created
```

6. Clean up the installation using the following commands once you have finished the verification of Gatekeeper policies.

```
Uninstall all packages and template installed.
kubectl delete -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/docs/how-to/webhook/httpsonly/good-example-ingress.yaml
kubectl delete -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/docs/how-to/webhook/httpsonly/constraint.yaml
```
More sample use cases

There are multiple use cases listed under the webhook directory. The steps are similar to what is specified in the example and can be summarized as follows:

1. Apply the template YAML file given in each use case directory.
2. Apply the constraint YAML file.
3. Verify by applying bad or good sample YAML files to validate the use case.

For further use cases, see the Gatekeeper library.

Enable gRPC support using the Citrix ingress controller

February 3, 2022

gRPC is a high performance, open-source universal RPC framework created by Google. In gRPC, a client application can directly call methods on a server application from a different server in the same way you call local methods.

You can easily create distributed applications and services using gRPC.

Enable gRPC support

Perform the following steps to enable GRPC support using HTTP2.

1. Create a YAML file `cic-configmap.yaml` and enable the global parameter for HTTP2 server side support using the following entry in the ConfigMap. For more information on using ConfigMap, see the ConfigMap documentation.

   ```
   NS_HTTP2_SERVER_SIDE: 'ON'
   ```

2. Apply the ConfigMap using the following command.

   ```
   kubectl apply -f cic-configmap.yaml
   ```
3. Edit the `cic.yaml` file for deploying the Citrix ingress controller to support ConfigMap.

```yaml
args:
- --ingress-classes
  citrix
- --configmap
  default/cic-configmap
```

4. Deploy the Citrix ingress controller as a stand-alone pod by applying the edited YAML file.

```bash
kubectl apply -f cic.yaml
```

5. To test the gRPC traffic, you may need to install `grpcurl`. Perform the following steps to install `grpcurl` on a Linux machine.

```bash
go get github.com/fullstorydev/grpcurl
go install github.com/fullstorydev/grpcurl/cmd/grpcurl
```

6. Apply the gRPC test service YAML file (`grpc-service.yaml`).

```bash
kubectl apply -f grpc-service.yaml
```

Following is a sample content for the `grpc-service.yaml` file.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: grpc-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: grpc-service
  template:
    metadata:
      labels:
        app: grpc-service
    spec:
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
7. Create a certificate for the gRPC Ingress configuration.

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout tls.key -out tls.crt -subj "/CN=grpc.example.com/O=grpc.example.com"
```

```
kubectl create secret tls grpc-secret --key tls.key --cert tls.crt
```

secret "grpc-secret" created

8. Enable HTTP2 using Ingress annotations. See HTTP/2 support for steps to enable HTTP2 using the Citrix ingress controller.

- Create a YAML file for the front-end Ingress configuration and apply it to enable HTTP/2 on the content switching virtual server.

  kubectl apply -f frontend-ingress.yaml

The content of the frontend-ingress.yaml file is provided as follows:
- Create a YAML file for the back-end Ingress configuration with the following content and apply it to enable HTTP2 on back-end (service group).

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    ingress.citrix.com/frontend-httpprofile: '{
      "http2": "enabled",
      "http2direct": "enabled"
    }
    ingress.citrix.com/frontend-ip: 192.0.2.1
    ingress.citrix.com/secure-port: "443"
  kubernetes.io/ingress.class: citrix
name: grpc-ingress
spec:
  rules:
    - host: grpc.example.com
      http:
```
Citrix ADC ingress controller

```yaml
paths:
  - backend:
      service:
        name: grpc-service
        port:
          number: 50051
      path: /
      pathType: Prefix
      tls:
        hosts:
          - grpc.example.com
        secretName: grpc-secret
```

9. Test the gRPC traffic using the `grpcurl` command.

```bash
grpcurl -v -insecure -d '{
  "name": "gRPC"
}' grpc.example.com:443 helloworld.Greeter.SayHello
```

The output of the command is shown as follows:

```plaintext
Resolved method descriptor:
rpc SayHello (.helloworld.HelloRequest) returns (.helloworld.HelloReply);

Request metadata to send:
(empty)

Response headers received:
content-type: application/grpc

Response contents:
{
  "message": "Hello gRPC"
}
```
Response trailers received: (empty)
Sent 1 request and received 1 response

Validate the rate limit CRD

Perform the following steps to validate the rate limit CRD.

1. Apply the rate limit CRD using the `ratelimit-crd.yaml` file.

   ```bash
   kubectl create -f ratelimit-crd.yaml
   ```

2. Create a YAML file (`ratelimit-crd-object.yaml`) with the following content for the rate limit policy.

   ```yaml
   apiVersion: citrix.com/v1beta1
   kind: ratelimit
   metadata:
     name: throttle-req-per-clientip
   spec:
     servicenames:
     - grpc-service
     selector_keys:
       basic:
       path: "/"
       per_client_ip: true
     req_threshold: 5
     timeslice: 60000
     throttle_action: "RESPOND"
   ```

3. Apply the YAML file using the following command.

   ```bash
   kubectl create -f ratelimit-crd-object.yaml
   ```

4. Test gRPC traffic using the `grpcurl` command.

   ```bash
   grpcurl -v -insecure -d '{
   "name": "gRPC"
   }' grpc.example.com:443 helloworld.Greeter.SayHello
   ```

   The command returns the following error in response after the rate limit is reached:
1 Error invoking method "helloworld.Greeter.SayHello": failed to query for service descriptor "helloworld.Greeter": rpc error: code = Unavailable desc =
2 Too Many Requests: HTTP status code 429; transport: missing content-type field

Validate the Rewrite and Responder CRD with gRPC

Perform the following steps to validate the Rewrite and Responder CRD.

1. Apply the Rewrite and Responder CRD using the rewrite-responder-policies-deployment.yaml file.

   kubectl create -f rewrite-responder-policies-deployment.yaml

2. Create a YAML file (rewrite-crd-object.yaml) with the following content for the rewrite policy.

   ```yaml
   apiVersion: citrix.com/v1
   kind: rewritepolicy
   metadata:
     name: addcustomheaders
   spec:
     rewrite-policies:
       - servicenames:
         - grpc-service
         rewrite-policy:
           operation: insert_http_header
           target: 'sessionID'
           modify-expression: "48592th42gl24456284536tgt2"
           comment: 'insert SessionID in header'
           direction: RESPONSE
           rewrite-criteria: 'http.res.is_valid'
   ```

3. Apply the YAML file using the following command.

   ```bash
   kubectl create -f rewrite-crd-object.yaml
   ```

4. Test the gRPC traffic using the grpcurl command.

© 1999-2022 Citrix Systems, Inc. All rights reserved.
This command adds a session id in the gRPC request response.

Resolved method descriptor:
```
rpc SayHello (.helloworld.HelloRequest ) returns ( .helloworld.HelloReply );
```

Request metadata to send:
```
(empty)
```

Response headers received:
```
content-type: application/grpc
sessionid: 48592th42gl24456284536tgt2
```

Response contents:
```
{
  "message": "Hello gRPC"
}
```

Response trailers received:
```
(empty)
```

Sent 1 request and received 1 response

Policy based routing support for multiple Kubernetes clusters

February 8, 2022

When you are using a single Citrix ADC to load balance multiple Kubernetes clusters, the Citrix ingress controller adds pod CIDR networks through static routes. These routes establish networking connectivity between Kubernetes pods and Citrix ADC. However, when the pod CIDRs overlap there may be route conflicts. Citrix ADC supports policy based routing (PBR) to address the networking conflicts in such scenarios. In PBR, decisions are taken based on the criteria that you specify. Typically, a next hop is specified where you send the selected packets. In a multi-cluster Kubernetes environment, PBR is implemented by reserving a subnet IP address (SNIP) for each Kubernetes cluster or the Citrix Ingress
Citrix ADC ingress controller

Controller. Using net profile, the SNIP is bound to all service groups created by the same Citrix ingress controller. For all the traffic generated from service groups belonging to the same cluster, the source IP address is the same SNIP.

Following is a sample topology where PBR is configured for two Kubernetes clusters which are load balanced using a Citrix ADC VPX or MPX.

Configure PBR using the Citrix ingress controller

To configure PBR, you need one SNIP or more per Kubernetes cluster. You can provide SNIP values either using the environment variable in the Citrix ingress controller deployment YAML file during bootup or using ConfigMap.

Perform the following steps to deploy the Citrix ingress controller and configure PBR using ConfigMap.

1. Download the `citrix-k8s-ingress-controller.yaml` using the following command:

   ```bash
   wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/baremetal/citrix-k8s-ingress-controller.yaml
   ```

2. Edit the Citrix ingress controller YAML file:
Citrix ADC ingress controller

1. Specify the values of the environment variables as per your requirements. For more information on specifying the environment variables, see the [Deploy Citrix ingress controller](/en-us/citrix-k8s-ingress-controller/cic-yaml.html) documentation.

3. Deploy the Citrix ingress controller using the edited YAML file with the following command on each cluster.

```
kubectl create -f citrix-k8s-ingress-controller.yaml
```

4. Create a YAML file `cic-configmap.yaml` with the required SNIP values in the ConfigMap. Following is an example for a ConfigMap with the SNIP values:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: pbr-test
  namespace: default
data:
  NS_SNIPS: ['"192.0.2.2", "192.0.2.1"']
```

5. Apply the ConfigMap.

```
kubectl create -f cic-configmap.yaml
```

You can also specify the SNIPs using the `NS_SNIPS` environment variable in the Citrix ingress controller deployment YAML file.

```yaml
- name: "NS_SNIPS"
  value: ['"192.0.2.2", "192.0.2.1"]
```

The following are the usage guidelines while using ConfigMap for configuring SNIP:

- Only SNIPs can be added or removed via ConfigMap. The `feature-node-watch` argument can only be enabled during bootup.
- When you add a ConfigMap:
Citrix ADC ingress controller

- If SNIPs are already provided using the environment variable during bootup and you want to retain them, those SNIPs should be specified in the ConfigMap along with the new SNIPs.

  - When you delete ConfigMap:
    - All PBRs generated by ConfigMap SNIPs are deleted. If SNIPs are provided via the environment variable, PBR for those IP addresses is added.
    - If SNIPs are not provided using the `NS_SNIPS` environment variable, static routes are added since `feature-node-watch` is enabled.

Validate PBR configuration on a Citrix ADC after deploying the Citrix ingress controller

This validation example uses a two node Kubernetes cluster with the Citrix ingress controller deployed along with the following ConfigMap with two SNIPs.

You can verify that the Citrix ingress controller adds the following configurations to the ADC:

1. An IPset of all NS_SNIPs provided by the ConfigMap is added.

2. A net profile is added with the SrcIP set to the IPset.
3. The service group added by the Citrix ingress controller contains the net profile set.

4. Finally, the Citrix ingress controller adds PBRs.
The number of PBRs is equivalent to (number of SNIPs) * (number of Kubernetes nodes). In this case, it adds four (2*2) PBRs.

The srcIP of the PBR is the NS_SNIPS provided to the Citrix ingress controller by ConfigMap. The destIP is the CNI overlay subnet range of the Kubernetes node.

NextHop is the IP address of the Kubernetes node.

5. You can also use the logs of the Citrix ingress controller to validate the configuration.
Configure PBR using the Citrix node controller

You can configure PBR using the Citrix node controller for multiple Kubernetes clusters. When you are using a single Citrix ADC to load balance multiple Kubernetes clusters with Citrix node controller for networking, the static routes added by it to forward packets to the IP address of the VXLAN tunnel interface may cause route conflicts. To support PBR, Citrix node controller needs to work in conjunction with the Citrix ingress controller to bind the net profile to the service group.

Perform the following steps to configure PBR using the Citrix node controller:

1. While starting the Citrix node controller, provide the CLUSTER_NAME as an environment variable. Specifying this variable indicates that it is a multi-cluster deployment and the Citrix node controller configures PBR instead of static routes.

   Example:

   ```
   name: CLUSTER_NAME
   value: "dev-cluster"
   ```

2. While deploying the Citrix ingress controller, provide the CLUSTER_NAME as an environment variable. This value should be the same as the value provided in Citrix node controller.

   Example:

   ```
   name: CLUSTER_NAME
   value: "dev-cluster"
   ```

3. Specify the argument --enable-cnc-pbr as True in the arguments section of the Citrix ingress controller deployment YAML file. When you specify this argument, Citrix ingress controller is aware that the Citrix node controller is configuring PBR on the Citrix ADC.

   Example:

   ```
   args:
   - --enable-cnc-pbr True
   ```
Citrix ADC ingress controller

**Note:**

- The value provided for `CLUSTER_NAME` in the Citrix node controller and Citrix ingress controller deployment files should match when they are deployed in the same Kubernetes cluster.
- The `CLUSTER_NAME` is used while creating the net profile entity and binding it to service groups on Citrix ADC VPX or MPX.

**Validate PBR configuration on a Citrix ADC after deploying the Citrix node controller**

This validation example uses a two node Kubernetes cluster with Citrix node controller and Citrix ingress controller deployed.

You can verify that the following configurations are added to the ADC by Citrix node controller:

1. A net profile is added, with the value of `srcIP` set to the SNIP added by Citrix node controller while creating the VXLAN tunnel network between the Citrix ADC and Kubernetes nodes.

```
> show netprofile
1) Name: cnc-cluster1_netprof
   SrcIP: 162.11.1254
   SrcIPPersistency: DISABLED
   OverrideLSN: DISABLED
   MBF: DISABLED
   Proxy Protocol: DISABLED
   Proxy Protocol version: V1
```

2. Citrix ingress controller binds the net profile to the service groups it creates.
3. Finally, Citrix node controller adds PBRs.

Here:

- The number of PBRs is equal to number of Kubernetes nodes. In this case, it adds two PBRs.
- The `srcIP` of the PBR is the `SNIP` added by Citrix node controller in tunnel network. The `destIP` is the CNI overlay subnet range of the Kubernetes node. The `NextHop` is the IP address of Kubernetes node’s VXLAN Tunnel interface.
Note:
Citrix node controller adds PBRs instead of static routes. The rest of the configuration of the VXLAN and bridge table remains the same. For more information, see the Citrix node controller configuration.

Single tier Citrix Ingress solution for MongoDB

February 15, 2022

MongoDB is one of the most popular NoSQL databases which is designed to process and store massive amounts of unstructured data. Cloud-native applications widely use MongoDB as a NoSQL database in the Kubernetes platform.

To identify and troubleshoot performance issues are a challenge in a Kubernetes environment due to the massive scale of application deployments. For database deployments like MongoDB, monitoring is a critical component of database administration to ensure that high availability and high performance requirements are met.

Citrix provides an ingress solution for load balancing and monitoring MongoDB databases on a Kubernetes platform using the advanced load balancing and performance monitoring capabilities of Citrix ADCs. Citrix Ingress solution for MongoDB provides you deeper visibility into MongoDB transactions and helps you to quickly identify and address performance issues whenever they occur. Using Citrix ADC observability exporter, you can export the MongoDB transactions to Elasticsearch and visualize them using Kibana dashboards to get deeper insights.

The following diagram explains Citrix Ingress solution for MongoDB using a single-tier deployment of Citrix ADC.
In this solution, a Citrix ADC VPX is deployed outside the Kubernetes cluster (Tier-1) and Citrix ADC observability exporter is deployed inside the Kubernetes cluster.

The Tier-1 Citrix ADC VPX routes the traffic (North-South) from MongoDB clients to Mongo DB query routers (Mongos) in the Kubernetes cluster. Citrix observability exporter is deployed inside the Kubernetes cluster.

As part of this deployment, an Ingress resource is created for Citrix ADC VPX (Tier-1 Ingress). The Tier-1 Ingress resource defines rules to enable load balancing for MongoDB traffic on Citrix ADC VPX and specifies the port for Mongo. Whenever MongoDB traffic arrives on the specified port on a Citrix ADC VPX, it routes this traffic to one of the Mongo service instances mentioned in the Ingress rule. Mongo service is exposed by the MongoDB administrator, and the same service instance is specified in the Ingress.

The Citrix observability exporter instance aggregates transactions from Citrix ADC VPX and uploads them to the Elasticsearch server. You can set up Kibana dashboards to visualize the required data (for example, query response time, most queried collection names) and analyze them to get meaningful insights. Only insert, update, delete, find, and reply operations are parsed and metrics are sent to the Citrix Observability Exporter.

**Prerequisites**

You must complete the following steps before deploying the Citrix Ingress solution for MongoDB.

- Set up a Kubernetes cluster in cloud or on-premises
- Deploy MongoDB in the Kubernetes cluster with deployment mode as **sharded replica set**. Other deployment modes for MongoDB are not supported.
Citrix ADC ingress controller

- Ensure that you have Elasticsearch installed and configured. Use the `elasticsearch.yaml` file for deploying Elasticsearch.
- Ensure that you have installed Kibana to visualize your transaction data. Use the `kibana.yaml` file for deploying Kibana.
- Deploy a Citrix ADC VPX instance outside the Kubernetes cluster. For instructions on how to deploy Citrix ADC VPX, see Deploy a Citrix ADC VPX instance.

Perform the following after you deploy the Citrix ADC VPX:

1. Configure an IP address from the subnet of the Kubernetes cluster as SNIP on the Citrix ADC. For information on configuring SNIPS in Citrix ADC, see Configuring Subnet IP Addresses (SNIPS).

2. Enable management access for the SNIP that is the same subnet of the Kubernetes cluster. The SNIP should be used as `NS_IP` variable in the Citrix ingress controller YAML file to enable the Citrix ingress controller to configure the Tier-1 Citrix ADC.

   **Note:**

   It is not mandatory to use SNIP as `NS_IP`. If the management IP address of the Citrix ADC is reachable from the Citrix ingress controller then you can use the management IP address as `NS_IP`.

3. Create a Citrix ADC system user account specific to the Citrix ingress controller. The Citrix ingress controller uses the system user account to automatically configure the Tier-1 Citrix ADC.

4. Configure Citrix ADC VPX to forward DNS queries to CoreDNS pod IP addresses in the Kubernetes cluster.

```bash
1 add dns nameServer <core-dns-pod-ip-address>
```

For example, if the pod IP addresses are 192.244.0.2 and 192.244.0.3, configure the name servers on Citrix ADC VPX as:

```bash
1 add dns nameServer 192.244.0.3
2 add dns nameServer 192.244.0.2
```

**Deploy the Citrix Ingress solution for MongoDB**

When you deploy the Citrix Ingress solution for MongoDB, you deploy the following components in the Kubernetes cluster:

- A stand-alone Citrix ingress controller for Citrix ADC VPX
Perform the following steps to deploy the Citrix Ingress solution for MongoDB.

1. Create a Kubernetes secret with the user name and password for Citrix ADC VPX.

   ```sh
   kubectl create secret generic nslogin --from-literal=username='username' --from-literal=password='mypassword'
   ```

2. Download the `cic-configmap.yaml` file and then deploy it using the following command.

   ```sh
   kubectl create -f cic-configmap.yaml
   ```

3. Deploy the Citrix ingress controller as a pod using the following steps.

   a) Download the Citrix ingress controller manifest file. Use the following command:

   ```sh
   ```

   b) Edit the Citrix ingress controller manifest file and enter the values for the following environmental variables:

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Mandatory or Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS_IP</td>
<td>Mandatory</td>
<td>The IP address of the Citrix ADC appliance. For more details, see Prerequisites.</td>
</tr>
<tr>
<td>NS_USER and NS_PASSWORD</td>
<td>Mandatory</td>
<td>The user name and password of the Citrix ADC VPX or MPX appliance used as the Ingress device.</td>
</tr>
<tr>
<td>EULA</td>
<td>Mandatory</td>
<td>The End User License Agreement. Specify the value as Yes.</td>
</tr>
<tr>
<td>Environment Variable</td>
<td>Mandatory or Optional</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LOGLEVEL</td>
<td>Optional</td>
<td>The log levels to control the logs generated by Citrix ingress controller. By default, the value is set to DEBUG. The supported values are: CRITICAL, ERROR, WARNING, INFO, and DEBUG.</td>
</tr>
<tr>
<td>NS_PROTOCOL and NS_PORT</td>
<td>Optional</td>
<td>Defines the protocol and port that must be used by the Citrix ingress controller to communicate with Citrix ADC. By default, the Citrix ingress controller uses HTTPS on port 443. You can also use HTTP on port 80.</td>
</tr>
<tr>
<td>ingress-classes</td>
<td>Optional</td>
<td>If multiple Ingress load balancers are used to load balance different Ingress resources. You can use this environment variable to specify the Citrix ingress controller to configure Citrix ADC associated with a specific Ingress class. For information on Ingress classes, see Ingress class support</td>
</tr>
<tr>
<td>NS_VIP</td>
<td>Optional</td>
<td>Citrix ingress controller uses the IP address provided in this environment variable to configure a virtual IP address to the Citrix ADC that receives Ingress traffic. <strong>Note:</strong> NS_VIP takes precedence over the frontend-ip annotation.</td>
</tr>
</tbody>
</table>

c) Specify or modify the following arguments in the Citrix ingress controller YAML file.
Citrix ADC ingress controller

d) Deploy the updated Citrix ingress controller manifest file using the following command:

```bash
kubectl create -f tier-1-vpx-cic.yaml
```

4. Create an Ingress object for the Tier-1 Citrix ADC using the `tier-1-vpx-ingress.yaml` file.

```bash
kubectl apply -f tier-1-vpx-ingress.yaml
```

Following is the content for the `tier-1-vpx-ingress.yaml` file. As per the rules specified in this Ingress resource, Citrix ingress controller configures the Citrix ADC VPX to listen for MongoDB traffic at port 27017. As shown in this example, you must specify the service that you have created for MongoDB query routers (for example: `serviceName: mongodb-mongos`) so that the Citrix ADC VPX can route traffic to it. Here, `mongodb-mongos` is the service for MongoDB query routers.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    ingress.citrix.com/analyticsprofile: '{
      "tcpinsight": {
        "tcpBurstReporting": "DISABLED"
      }
    }'
  ingress.citrix.com/insecure-port: "27017"
  ingress.citrix.com/insecure-service-type: mongo
  ingress.citrix.com/insecure-termination: allow
  kubernetes.io/ingress.class: tier-1-vpx
spec:
  defaultBackend:
    service:
```
5. Deploy Citrix ADC observability exporter with Elasticsearch as the endpoint using the `coe-es-mongo.yaml` file.

```bash
kubectl apply -f coe-es-mongo.yaml
```

**Note:**
You must set the Elasticsearch server details in the ELKServer environment variable either based on an IP address or the DNS name, along with the port information.

Following is a sample ConfigMap file.

```yaml
apiVersion: v1
code: ConfigMap
metadata:
  name: coe-config-es
data:
  lstreamd_default.conf: |
  |
    "Endpoints": {
    
      "ES": {
      
        "ServerUrl": "elasticsearch.default.svc.cluster.local:9200",
        "IndexPrefix": "adc_coe",
        "IndexInterval": "daily",
        "RecordType": {
        
          "HTTP": "all",
          "TCP": "all",
          "SWG": "all",
          "VPN": "all",
          "NGS": "all",
          "ICA": "all",
          "APPPFW": "none",
          "BOT": "none",
          "VIDEOOPT": "none",
        
```
Verify the deployment of Citrix Ingress solution

You can use the command as shown in the following example to verify that all the applications are deployed and list all services and ports.

```
# kubectl get ingress
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>HOSTS</th>
<th>ADDRESS</th>
<th>PORTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpx-ingress</td>
<td>*</td>
<td>80</td>
<td>22d</td>
<td></td>
</tr>
</tbody>
</table>

You can use the `kubectl get ingress` command as shown in the following example to get information about the Ingress objects deployed.
Verify observability for MongoDB traffic

This topic provides information on how to get visibility into MongoDB transactions using the Citrix Ingress solution and it uses Kibana dashboards to visualize the database performance statistics.

Before performing the steps in this topic ensure that:

- You have deployed MongoDB as a sharded replica set in the Kubernetes cluster
- Deployed the Citrix Ingress solution for MongoDB
- A client application for MongoDB is installed to send traffic to the MongoDB.
- Kibana is installed for visualization

Perform the following steps to verify observability for MongoDB traffic.

1. Configure your client application for MongoDB to point to the virtual IP address of the Tier-1 Citrix ADC VPX.
   For example:
   ```
   1. `mongodb://<vip-of-vpx>:27017/`
   ```

2. Send multiple requests (for example insert, update, delete) to the MongoDB database using your MongoDB client application. The transactions are uploaded to the Elasticsearch server.

3. Set up a Kibana dashboard to visualize the MongoDB transactions. You can use the following sample Kibana dashboard.
Citrix ADC ingress controller

In this dashboard, you can see performance statistics for your MongoDB deployment including the different type of queries and query response time. Analyzing this data helps you to find any anomalies like latency in a transaction and take immediate action.

Export telemetry data to Prometheus

For your Kubernetes deployment, if you have your Prometheus server deployed in the same Kubernetes cluster, you can configure annotations to enable Prometheus to automatically add Citrix ADC observability exporter as a scrape target.

Following is a snippet of Citrix ADC observability exporter YAML file (coe-es-mongodb.yaml) with these annotations.

```yaml
1 template:
2   metadata:
3     name: coe-es
4     labels:
5       app: coe-es
6     annotations:
```
Citrix ADC ingress controller

```
7  prometheus.io/scrape: "true"
8  prometheus.io/port: "5563"
```

Alternatively, you can manually add Citrix ADC observability exporter as the scrape target on your Prometheus server configuration file.

Also, ensure that metrics for Prometheus are enabled in the `cic-configmap.yaml` file as shown in the following YAML file.

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: cic-configmap
5    namespace: default
6  data:
7    NS_ANALYTICS_CONFIG: |
8      distributed_tracing:
9        enable: 'false'
10       samplingrate: 0
11      endpoint:
12        server: 'coe-es.default.svc.cluster.local'
13    timeseries:
14      port: 5563
15    metrics:
16      enable: 'true'
17      mode: 'prometheus'
18    auditlogs:
19      enable: 'false'
20    events:
21      enable: 'false'
22    transactions:
23      enable: 'true'
24      port: 5557
25  <!--NeedCopy-->  
```

In this YAML file, the following configuration enables metrics for Prometheus.

```
1  metrics:
2    enable: 'true'
3    mode: 'prometheus'
```
Canary and blue-green deployment using Citrix ADC VPX and Azure pipelines for Kubernetes based applications

February 3, 2022

This topic provides information on how to achieve canary and blue-green deployment for Kubernetes applications using Citrix ADC VPX and Azure pipelines.

Canary deployment using Citrix ADC VPX and Azure pipelines for Kubernetes based applications

Canary is a deployment strategy which involves deploying new versions of an application in small and phased incremental steps. The idea of canary is to first deploy the new changes to a small set of users to take a decision on whether to reject or promote the new deployments and then roll out the changes to the rest of the users. This strategy limits the risk involved in deploying a new version of the application in the production environment.

Azure pipelines are a cloud service provided by Azure DevOps which allows you to automatically run builds, perform tests, and deploy code to various development and production environments.

This section provides information on how to achieve canary deployment for Kubernetes based application using Citrix ADC VPX and Citrix ingress controller with Azure pipelines.

Benefits of Canary deployment

- Canary version of application acts as an early warning for potential problems that might be present in the new code and the deployment issues.
- You can use the canary version for smoke tests and A/B testing.
- Canary offers easy rollback and zero-downtime upgrades.
- You can run multiple versions of applications together at the same time.

In this solution, Citrix ADC VPX is deployed on the Azure platform to enable load balancing of an application and achieve canary deployment using Citrix ADC VPX. For more information on how to deploy Citrix ADC on Microsoft Azure, see the Citrix documentation link.

Canary deployment using Citrix ADC

You can achieve canary deployment using Citrix ADC with Ingress annotations which is a rule based canary deployment. In this approach, you need to define an additional Ingress object with specific annotations to indicate that the application request needs to be served based on the rule based canary deployment strategy. In the Citrix solution, Canary based traffic routing at the Ingress level can be achieved by defining various sets of rules as follows:
Citrix ADC ingress controller

- Applying the canary rules based on weight
- Applying the canary rules based on the HTTP request header
- Applying the canary rules based on the HTTP header value

For more information, see simplified canary deployment using Ingress annotations

Canary deployment using Citrix ADC VPX with Azure pipelines

Citrix proposes a solution for canary deployment using Citrix ADC VPX and Citrix ingress controller with Azure pipelines for Kubernetes based applications.

In this solution, there are three configuration directories:

- kubernetes_configs
- deployment_configs
- pipeline_configs

kubernetes_configs

This directory includes the version based application specific deployment YAML files and the Helm based configuration files to deploy Citrix ingress Controller which is responsible to push Citrix ADC configuration to achieve canary deployment.
Citrix ADC ingress controller

You can download the latest Helm charts from the Citrix ingress controller Helm charts repository and place it under the cic_helm directory.

**deployment_configs**

This directory includes the setup_config and teardown_config JSON files that specify the path of the YAML files available for the specific version of the application to be deployed or brought down during canary deployment.

**pipeline_configs**

This directory includes the Azure pipeline script and the python script which reads the user configurations and triggers the pipeline based on the user request to introduce a new version of the application or teardown a version of an application. The change in percentage of traffic weight in application ingress YAML would trigger the pipeline to switch the traffic between the available version of applications.
Citrix ADC ingress controller

With all the three configuration files in place, any update to the files under `deployment_configs` and `kubernetes_configs` directories in GitHub, would trigger the pipeline in Azure.

The traffic split percentage can be adjusted using the `ingress.citrix.com/canary-weight` annotation in the ingress YAML of the application.

**Deploy a sample application on Canary in Azure pipelines**

This topic explains how to deploy a sample application on Canary mode using Citrix ADC and Azure pipelines.

**Prerequisites**

Ensure that:

- Citrix ADC VPX is already deployed on the Azure platform and is ready to be used by our sample application.
- AKS cluster with Kubernetes service connection configured for the Azure pipeline.

Perform the following steps:

1. Clone the GitHub repository and go to the directory `cd/canary-azure-devops`.
2. Place the application deployment specific YAMLs (with the ingress file) under a versioned folder `v1` in the `kubernetes_configs` directory.
3. Create three Azure pipelines using the existing YAML files, `deploy_cic.yaml`, `deploy.yaml`, and `teardown.yaml`, for deploying Citrix ingress controller and deploying and tearing down the applications. See, Azure pipelines for creating a pipeline.
4. Update the subscription, agent pool, service connection and Citrix ADC details in the pipeline YAML.
5. Save the pipeline.
6. Update the path in `deploy_config.json` with the path specifying the directory where the application YAMLs are placed.

```json
{
  "K8S_CONFIG_PATH" : "cd/canary-azure-devops/kubernetes_configs/v1",
}
```

7. Commit the `deploy_config.json` file and v1 directory using Git to trigger the pipeline to deploy the v1 version of the application.

8. Access the application through Citrix ADC.

9. Introduce the v2 version of the application by creating the v2 directory under `kubernetes_configs`. Make sure that the ingress under this version has the canary annotation specified with the right weight to be set for traffic split.

10. Deploy the version v2 of the application by updating `deploy_config.json` with the path specifying the v2 directory. Now, the traffic is split between version v1 and v2 based on the canary weight set in the ingress annotation (for example, `ingress.citrix.com/canary-weight: "40"`)

11. Continue progressively increasing the traffic weight in the ingress annotation until the new version is ready to serve all the traffic.

### Blue-green deployment using Citrix ADC VPX and Azure pipelines for Kubernetes based applications

Blue-green deployment is a technique that reduces downtime and risk by running two identical production environments called blue and green. At any time, only one of the environments is live that serves all the production traffic. The basis of the blue-green method is side-by-side deployments of two separate but identical environments. Deploying an application in both the environments can be fully automated by using jobs and tasks. This approach enforces duplication of every resource of an application. However, there are many different ways blue-green deployments can be carried out in various continuous deployment tools.

Using Citrix ADC VPX with Azure pipelines the same canary based solution can be used to achieve blue-green deployment by adjusting the traffic weight to either zero or 100.
Traffic management for external services

February 3, 2022

Sometimes, all the available services of an application may not be deployed completely on a single Kubernetes cluster. You may have applications that rely on the services outside of one cluster as well. In this case, micro services need to define an ExternalName service to resolve the domain name. However, in this approach, you would not be able to get features such as traffic management, policy enforcement, fail over management and so on. As an alternative, you can configure Citrix ADC to resolve the domain names and leverage the features of Citrix ADC.

Configure Citrix ADC to reach external services

You can configure Citrix ADC as a domain name resolver using Citrix ingress controller. When you configure Citrix ADC as domain name resolver, you need to resolve:

- Reachability of Citrix ADC from microservices
- Domain name resolution at Citrix ADC to reach external services

Configure a service for reachability from Kubernetes cluster to Citrix ADC

To reach Citrix ADC from microservices, you have to define a headless service which would be resolved to a Citrix ADC service and thus the connectivity between microservices and Citrix ADC establishes.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: external-svc
spec:
  selector:
    app: cpx
  ports:
  - protocol: TCP
    port: 80
```

Configure Citrix ADC as a domain name resolver using Citrix ingress controller

You can configure Citrix ADC through Citrix ingress controller to create a domain based service group using the ingress annotation ingress.citrix.com/external-service. The value for ingress
.citrix.com/external-service is a list of external name services with their corresponding domain names. For Citrix ADC VPX, name servers are configured on Citrix ADC using the ConfigMap.

Note:
ConfigMaps are used to configure name servers on Citrix ADC only for Citrix ADC VPX. For Citrix ADC CPX, CoreDNS forwards the name resolution request to the upstream DNS server.

Traffic management using Citrix ADC CPX

The following diagram explains Citrix ADC CPX deployment to reach external services. An Ingress is deployed where the external service annotation is specified to configure DNS on Citrix ADC CPX.

Note: A ConfigMap is used to configure name servers on Citrix ADC VPX.

In this deployment:
1. A microservice sends the DNS query for www.externalsvc.com which would get resolved to the Citrix ADC CPX service.
2. Citrix ADC CPX resolves www.externalsvc.com and reaches external service.

Following are the steps to configure Citrix ADC CPX to load balance external services:
1. Define a headless service to reach Citrix ADC.
2. Define an ingress and specify the external-service annotation as specified in the `dbs-ingress.yaml` file. When you specify this annotation, Citrix ingress controller creates DNS servers on Citrix ADC and binds the servers to the corresponding service group.

   ```yaml
   apiVersion: v1
   kind: Service
   metadata:
     name: external-svc
   spec:
     selector:
       app: cpx
     ports:
       - protocol: TCP
         port: 80
   ```

   annotations:
   ingress.citrix.com/external-service: '{
     "external-svc": {
       "domain": "www.externalsvc.com"
     }
   }'

3. Add the IP address of the DNS server on Citrix ADC using ConfigMap.

   ```yaml
   apiVersion: v1
   kind: ConfigMap
   metadata:
     name: nameserver-cmap
   namespace: default
   data:
     NS_DNS_NAMESERVER: '[]'
   ```

   Note:
   This step is applicable only for Citrix ADC VPX.
Supported platforms and deployments

July 12, 2022

This topic provides details about various Kubernetes platforms, deployment topologies, features, and CNIs supported in Cloud-Native deployments that include Citrix ADC and Citrix ingress controller.

Kubernetes platforms

Citrix ingress controller is supported on the following platforms:

- Kubernetes v1.10 (and later) on bare metal or self-hosted on public clouds such as, AWS, GCP, or Azure.
- Google Kubernetes Engine (GKE)
- Elastic Kubernetes Service (EKS)
- Azure Kubernetes Service (AKS)
- Red Hat OpenShift version 3.11 and later
- Pivotal Container Service (PKS)
- Diamanti Enterprise Kubernetes Platform
- Mirantis Kubernetes Engine
- VMware Tanzu

Citrix ADC platforms

The following table lists the Citrix ADC platforms supported by the Citrix ingress controller:

<table>
<thead>
<tr>
<th>Citrix ADC Platform</th>
<th>Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Citrix ADC MPX</td>
<td>11.1–61.7 and later</td>
</tr>
<tr>
<td>Citrix ADC VPX</td>
<td>11.1–61.7 and later</td>
</tr>
<tr>
<td>Citrix ADC CPX</td>
<td>12.1–51.16 and later</td>
</tr>
</tbody>
</table>

Supported deployment topologies on platforms (on-premises)

The following table lists the various deployment topologies supported by the Citrix ingress controller on the supported Kubernetes (on-premises) platforms:
# Supported deployment topologies on cloud platforms

The following table lists the various deployment topologies supported by the Citrix ingress controller on the supported cloud platforms:

<table>
<thead>
<tr>
<th>Deployment Topologies</th>
<th>Kubernetes</th>
<th>Red Hat OpenShift</th>
<th>PKS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single-Tier</strong> (Citrix ADC MPX or VPX in tier-1)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Dual-Tier</strong> (Citrix ADC MPX or VPX in tier-1 and Citrix ADC CPXs in tier-2)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Service mesh lite</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Services of type LoadBalancer</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Services of type NodePort</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>Deployment Topologies</th>
<th>GKE</th>
<th>EKS</th>
<th>AKS (Basic mode - Kubenet)</th>
<th>AKS (Advanced mode - Azure CNI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual-Tier Cloud topology (Cloud LB in tier-1 and Citrix ADC CPXs in tier-2)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Supported Citrix ingress controller feature on platforms**

The following table lists the Citrix ingress controller features supported on various cloud-native platforms:

<table>
<thead>
<tr>
<th>Citrix ingress controller features</th>
<th>Kubernetes</th>
<th>Google Cloud</th>
<th>AWS</th>
<th>Azure</th>
<th>Red Hat OpenShift</th>
<th>PKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP Ingress</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>UDP Ingress</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SSL Ingress</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>TCP over SSL Ingress</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>HTTP, TCP, or SSL profiles</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>NodePort support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
The following table lists the Citrix ingress controller features supported on the respective Citrix Ingress Controller versions and Citrix ADC versions:

<table>
<thead>
<tr>
<th>Citrix ingress controller features</th>
<th>Citrix ingress controller versions</th>
<th>Citrix ADC MPX or VPX versions</th>
<th>Citrix ADC CPX versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP Ingress</td>
<td>1.1.1 and later</td>
<td>11.1–61.7 and later</td>
<td>12.1–51.16 and later</td>
</tr>
</tbody>
</table>
# Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>Citrix ingress controller features</th>
<th>Citrix ingress controller versions</th>
<th>Citrix ADC MPX or VPX versions</th>
<th>Citrix ADC CPX versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDP Ingress</td>
<td>1.1.1 and later</td>
<td>11.1–61.7 and later</td>
<td>12.1–51.16 and later</td>
</tr>
<tr>
<td>SSL Ingress</td>
<td>1.1.1 and later</td>
<td>11.1–61.7 and later</td>
<td>12.1–51.16 and later</td>
</tr>
<tr>
<td>TCP over SSL Ingress</td>
<td>1.1.1 and later</td>
<td>11.1–61.7 and later</td>
<td>12.1–51.16 and later</td>
</tr>
<tr>
<td>HTTP, TCP, or SSL profiles</td>
<td>1.4.392</td>
<td>11.1–61.7 and later</td>
<td>12.1–51.16 and later</td>
</tr>
<tr>
<td>NodePort support</td>
<td>1.1.1 and later</td>
<td>11.1–61.7 and later</td>
<td>12.1–51.16 and later</td>
</tr>
<tr>
<td>Type LoadBalancer support</td>
<td>1.2.0 and later</td>
<td>11.1–61.7 and later</td>
<td>12.1–51.16 and later</td>
</tr>
<tr>
<td>Rewrite and Responder CRD</td>
<td>1.1.1 and later</td>
<td>11.1–61.7 and later</td>
<td>12.1–51.16 and later</td>
</tr>
<tr>
<td>Rate limit CRD</td>
<td>1.4.392</td>
<td>11.1–61.7 and later</td>
<td>12.1–51.16 and later</td>
</tr>
<tr>
<td>Auth CRD</td>
<td>1.4.392</td>
<td>11.1–61.7 and later</td>
<td>12.1–51.16 and later</td>
</tr>
<tr>
<td>Advanced content routing</td>
<td>1.7.46</td>
<td>12.1–51.16 and later</td>
<td>12.1–51.16 and later</td>
</tr>
<tr>
<td>WAF CRD</td>
<td>1.9.2</td>
<td>13.0–65.4 and later</td>
<td>13.0–65.4 and later</td>
</tr>
<tr>
<td>Bot CRD</td>
<td>1.11.3</td>
<td>Citrix ADC VPX version 13.0.67.39 and later</td>
<td>Not supported</td>
</tr>
<tr>
<td>OpenShift Routes</td>
<td>1.1.3 and later</td>
<td>12.1–51.16 and later</td>
<td>13.0–36.28 and later</td>
</tr>
<tr>
<td>OpenShift router sharding</td>
<td>1.2.0 and later</td>
<td>12.1–51.16 and later</td>
<td>13.0–36.28 and later</td>
</tr>
<tr>
<td>Simplified canary using Ingress</td>
<td>Version 1.13.15 and later</td>
<td>11.1–61.7 and later</td>
<td>12.1–51.16 and later</td>
</tr>
</tbody>
</table>

## Container network interfaces (CNIs) for Citrix ADC CPX

The following table lists the Container network interfaces (CNIs) supported by Citrix ADC CPX:

<table>
<thead>
<tr>
<th>Container network interfaces (CNI)</th>
<th>Citrix ADC CPX versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flannel</td>
<td>12.1–51.16 and later</td>
</tr>
<tr>
<td>Kubenet</td>
<td>12.1–51.16 and later</td>
</tr>
<tr>
<td>Calico</td>
<td>13.0–36.28</td>
</tr>
</tbody>
</table>

© 1999–2022 Citrix Systems, Inc. All rights reserved.
Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>Container network interfaces (CNI)</th>
<th>Citrix ADC CPX versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canal</td>
<td>13.0–36.28</td>
</tr>
<tr>
<td>Calico on GKE</td>
<td>12.1–51.16 and later</td>
</tr>
<tr>
<td>OVS</td>
<td>13.0–36.28</td>
</tr>
<tr>
<td>Weave</td>
<td>12.1–51.16 and later</td>
</tr>
<tr>
<td>Cilium</td>
<td>13-0-71-40 and later</td>
</tr>
</tbody>
</table>

**Supported container runtime interfaces for Citrix ADC CPX**

The following table lists the container runtime interfaces (CRIs) supported by Citrix ADC CPX.

<table>
<thead>
<tr>
<th>CRI</th>
<th>Supported versions of Citrix ADC CPX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Docker</td>
<td>11.1 and later</td>
</tr>
<tr>
<td>CRI-O</td>
<td>13.0–47.103 and later</td>
</tr>
</tbody>
</table>

**Support matrix for cloud native solution components**

The following matrix provides information on compatibility between the different components of the cloud native solution offered by Citrix.

For example, the first row of this table explains the versions of Citrix ADC CPX/VPX/MPX which supports different components of the Citrix cloud native solution. In this table NA is marked if the components are not dependent on each other or when the components are the same.
<table>
<thead>
<tr>
<th>Product/component</th>
<th>Citrix ADC ingress controller</th>
<th>Citrix observability exporter (COE)</th>
<th>Citrix istio adapter (CIA)</th>
<th>Citrix node controller</th>
<th>ADM agent</th>
<th>ADM service on-prem</th>
<th>ADM metrics exporter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Citrix ADC CPX/VPX</td>
<td>NA</td>
<td>Citrix ingress controller version 1.0.001 onwards</td>
<td>1.0.0-alpha onwards</td>
<td>1.0.0-alpha onwards</td>
<td>13.0 onwards</td>
<td>47.22 onwards</td>
<td>12.1 onwards</td>
</tr>
<tr>
<td>CPX/VPX</td>
<td>NA</td>
<td>Citrix ingress controller version 1.1.1 onwards</td>
<td>1.1.1 onwards</td>
<td>1.0.01 onwards</td>
<td>13.0 onwards</td>
<td>47.22 onwards</td>
<td>12.1 onwards</td>
</tr>
<tr>
<td></td>
<td>NA</td>
<td>Citrix ingress controller version 12.1+ onwards</td>
<td>12.1+ onwards</td>
<td>12.1+ onwards</td>
<td>11.1 onwards</td>
<td>47.22 onwards</td>
<td>12.1 onwards</td>
</tr>
<tr>
<td></td>
<td>NA</td>
<td>Citrix ingress controller version 11.1 onwards</td>
<td>11.1 onwards</td>
<td>11.1 onwards</td>
<td>11.1 onwards</td>
<td>47.22 onwards</td>
<td>12.1 onwards</td>
</tr>
<tr>
<td></td>
<td>NA</td>
<td>Citrix ingress controller version 10.1 onwards</td>
<td>10.1 onwards</td>
<td>10.1 onwards</td>
<td>10.1 onwards</td>
<td>47.22 onwards</td>
<td>12.1 onwards</td>
</tr>
<tr>
<td></td>
<td>NA</td>
<td>Citrix ingress controller version 9.1 onwards</td>
<td>9.1 onwards</td>
<td>9.1 onwards</td>
<td>9.1 onwards</td>
<td>47.22 onwards</td>
<td>12.1 onwards</td>
</tr>
<tr>
<td></td>
<td>NA</td>
<td>Citrix ingress controller version 8.1 onwards</td>
<td>8.1 onwards</td>
<td>8.1 onwards</td>
<td>8.1 onwards</td>
<td>47.22 onwards</td>
<td>12.1 onwards</td>
</tr>
<tr>
<td></td>
<td>NA</td>
<td>Citrix ingress controller version 7.1 onwards</td>
<td>7.1 onwards</td>
<td>7.1 onwards</td>
<td>7.1 onwards</td>
<td>47.22 onwards</td>
<td>12.1 onwards</td>
</tr>
<tr>
<td></td>
<td>NA</td>
<td>Citrix ingress controller version 6.1 onwards</td>
<td>6.1 onwards</td>
<td>6.1 onwards</td>
<td>6.1 onwards</td>
<td>47.22 onwards</td>
<td>12.1 onwards</td>
</tr>
<tr>
<td></td>
<td>NA</td>
<td>Citrix ingress controller version 5.1 onwards</td>
<td>5.1 onwards</td>
<td>5.1 onwards</td>
<td>5.1 onwards</td>
<td>47.22 onwards</td>
<td>12.1 onwards</td>
</tr>
<tr>
<td></td>
<td>NA</td>
<td>Citrix ingress controller version 4.1 onwards</td>
<td>4.1 onwards</td>
<td>4.1 onwards</td>
<td>4.1 onwards</td>
<td>47.22 onwards</td>
<td>12.1 onwards</td>
</tr>
<tr>
<td></td>
<td>NA</td>
<td>Citrix ingress controller version 3.1 onwards</td>
<td>3.1 onwards</td>
<td>3.1 onwards</td>
<td>3.1 onwards</td>
<td>47.22 onwards</td>
<td>12.1 onwards</td>
</tr>
<tr>
<td></td>
<td>NA</td>
<td>Citrix ingress controller version 2.1 onwards</td>
<td>2.1 onwards</td>
<td>2.1 onwards</td>
<td>2.1 onwards</td>
<td>47.22 onwards</td>
<td>12.1 onwards</td>
</tr>
<tr>
<td></td>
<td>NA</td>
<td>Citrix ingress controller version 1.1 onwards</td>
<td>1.1 onwards</td>
<td>1.1 onwards</td>
<td>1.1 onwards</td>
<td>47.22 onwards</td>
<td>12.1 onwards</td>
</tr>
</tbody>
</table>

© 1999–2022 Citrix Systems, Inc. All rights reserved.
<table>
<thead>
<tr>
<th>Product/component</th>
<th>Citrix ob-</th>
<th>Citrix observability</th>
<th>Citrix istio exporter</th>
<th>Citrix node controller</th>
<th>ADM agent</th>
<th>ADM service</th>
<th>ADM on-prem</th>
<th>Citrix ADC metrics exporter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Citrix ADC ingress controller</td>
<td>CPX</td>
<td>NA</td>
<td>COE</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>12.1+ onwards</td>
<td></td>
<td></td>
<td>1.0.001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NA</td>
</tr>
<tr>
<td>and</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NA</td>
</tr>
<tr>
<td>VPX/MPX 11.1+ onwards</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NA</td>
</tr>
<tr>
<td>on-wards</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NA</td>
</tr>
<tr>
<td>supports</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NA</td>
</tr>
<tr>
<td>Citrix ingress controller</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NA</td>
</tr>
<tr>
<td>version 1.1.1 onwards</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NA</td>
</tr>
<tr>
<td>and</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NA</td>
</tr>
<tr>
<td>is supported with</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NA</td>
</tr>
<tr>
<td>Citrix ingress controller</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NA</td>
</tr>
<tr>
<td>version 1.5.6 onwards</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NA</td>
</tr>
<tr>
<td>on-wards</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NA</td>
</tr>
<tr>
<td>Product/component</td>
<td>Citrix ADC ingress controller</td>
<td>Citrix observability exporter</td>
<td>Citrix istio adapter (CIA)</td>
<td>Citrix node controller ADM on-prem</td>
<td>Citrix ADC metrics exporter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------------</td>
<td>-------------------------------</td>
<td>--------------------------</td>
<td>---------------------------------</td>
<td>----------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Citrix ADC CPX/VPX/MPX</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.0 onwards</td>
<td>NA</td>
<td>NA</td>
<td>1.2.0-beta</td>
<td>NA</td>
<td>NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>is supported with</td>
<td></td>
<td></td>
<td>is supported with COE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.5.6</td>
<td></td>
<td></td>
<td>version</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COE</td>
<td></td>
<td></td>
<td>1.0.001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>version is supported with</td>
<td></td>
<td></td>
<td>1.0.001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0.001 onwards</td>
<td></td>
<td></td>
<td>1.0.001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>onwards</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Citrix istio adapter (CIA)</td>
<td>CPX/VPX/MPX</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.1+</td>
<td>1.0.001</td>
<td>is supported with</td>
<td>is supported with CIA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>onwards</td>
<td></td>
<td>version</td>
<td>version</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>is supported with</td>
<td></td>
<td>1.2.0-beta</td>
<td>1.2.0-beta</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CIA</td>
<td></td>
<td>version</td>
<td>version</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>version</td>
<td></td>
<td>1.0.001</td>
<td>1.0.001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2.0-beta</td>
<td></td>
<td>is supported with</td>
<td>is supported with</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>onwards</td>
<td></td>
<td>version</td>
<td>CIA version</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.2.0-beta</td>
<td>1.2.0-beta</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>version</td>
<td>version</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.0.001</td>
<td>1.0.001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>Product/component</th>
<th>Citrix ADC ingress controller</th>
<th>Citrix observability exporter (COE)</th>
<th>Citrix istio adaptor (CIA)</th>
<th>Citrix node controller</th>
<th>ADM agent</th>
<th>ADM service</th>
<th>ADM on-prem</th>
<th>Citrix ADC metrics exporter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Citrix node controller</td>
<td>CPX/VPX 12.0 onwards</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>ADM agent</td>
<td>CPX/VPX/MPX 13.0–47.22 onwards</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>ADM service</td>
<td>CPX/VPX 13.0–47.22 onwards</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>ADM on-prem</td>
<td>CPX/VPX/MPX 11.1 onwards</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Citrix ADC metrics exporter</td>
<td>CPX/VPX 12.1 onwards</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Note:**
For better use case coverage, use the latest versions of the components provided in the compatibility table.
Authentication and authorization policies for Kubernetes with Citrix ADC

April 14, 2022

Authentication and authorization policies are used to enforce access restrictions to the resources hosted by an application or API server. While you can verify the identity using the authentication policies, authorization policies are used to verify whether a specified request has the necessary permissions to access a resource.

Citrix provides a Kubernetes CustomResourceDefinition (CRD) called the **Auth CRD** that you can use with the Citrix ingress controller to define authentication policies on the ingress Citrix ADC.

**Auth CRD definition**

The Auth CRD is available in the Citrix ingress controller GitHub repo at: auth-crd.yaml. The Auth CRD provides attributes for the various options that are required to define the authentication policies on the Ingress Citrix ADC.

**Auth CRD attributes**

The Auth CRD provides the following attributes that you use to define the authentication policies:

- servicenames
- authentication_mechanism
- authentication_providers
- authentication_policies
- authorization_policies

**Servicenames**

The name of the services for which the authentication and authorization policies need to be applied.

**Authentication mechanism**

The following authentication mechanisms are supported:

- Using request headers:
  Enables user authentication using the request header. You can use this mechanism when the credentials or API keys are passed in a header (typically Authorization header). For example, you can use authentication using request headers for basic, digest, bearer authentication, or API keys.
Citrix ADC ingress controller

- Using forms:
  You can use this mechanism with user or web authentication including the relying party configuration for OpenID connect and the service provider configuration for SAML.

When the authentication mechanism is not specified, the default is authentication using the request header.

The following are the attributes for forms based authentication.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>authentication_host</td>
<td>Specifies a fully qualified domain name (FQDN) to which the user must be redirected for ADC authentication service. This FQDN should be unique and should resolve to the front-end IP address of Citrix ADC with Ingress/service type LoadBalancer or the VIP address of the Listener CRD.</td>
</tr>
<tr>
<td>authentication_host_cert</td>
<td>Specifies the name of the SSL certificate to be used with the authentication_host. This certificate is mandatory while performing authentication using the form.</td>
</tr>
<tr>
<td>ingress_name</td>
<td>Specifies the Ingress name for which the authentication using forms is applicable.</td>
</tr>
<tr>
<td>lb_service_name</td>
<td>Specifies the name of the service of type LoadBalancer for which the authentication using forms is applicable.</td>
</tr>
<tr>
<td>listener_name</td>
<td>The name of the Listener CRD for which the authentication using forms is applicable.</td>
</tr>
<tr>
<td>vip</td>
<td>Specifies the front-end IP address of the Ingress for which the authentication using forms is applicable. This attribute refers to the frontend-ip address provided with the Ingress. If there is more than one Ingress resource which uses the same frontend-ip, it is recommended to use vip.</td>
</tr>
</tbody>
</table>

Note:
- While using forms, authentication can be enabled for all types of traffic. Currently, granular
authentication is not supported.

- Depending on the resource to which you need to apply form based authentication, you can use one of the `ingress_name`, `lb_service_name`, `listener_name`, or `vip` attributes to specify the resource.

**Authentication providers**

The **providers** define the authentication mechanism and parameters that are required for the authentication mechanism.

**Basic authentication**

Specifies that local authentication is used with the HTTP basic authentication scheme. To use basic authentication, you must create user accounts on the ingress Citrix ADC.

**OAuth authentication**

The OAuth authentication mechanism, requires an external identity provider to authenticate the client using OAuth2 and issue an Access token. When the client presents the Access token to a Citrix ADC as an access credential, the Citrix ADC validates the token using the configured values. If the token validation is successful then Citrix ADC grants access to the client.

**OAuth authentication attributes**

The following are the attributes for OAuth authentication:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issuer</td>
<td>The identity (usually a URL) of the server whose tokens need to be accepted for authentication.</td>
</tr>
<tr>
<td>jwks_uri</td>
<td>The URL of the endpoint that contains JWKs (JSON Web Key) for JWT (JSON Web Token) verification.</td>
</tr>
<tr>
<td>audience</td>
<td>The identity of the service or application for which the token is applicable.</td>
</tr>
<tr>
<td>token_in_hdr</td>
<td>The custom header name where the token is present. The default value is the Authorization header.</td>
</tr>
</tbody>
</table>

**Note:** You can specify more than one header.
### Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>token_in_param</strong></td>
<td>The query parameter where the token is present.</td>
</tr>
<tr>
<td><strong>signature_algorithms</strong></td>
<td>Specifies the list of signature algorithms which are allowed. By default HS256, RS256, and RS512 algorithms are allowed.</td>
</tr>
<tr>
<td><strong>introspect_url</strong></td>
<td>The URL of the introspection endpoint of the authentication server (IdP). If the access token presented is an opaque token, introspection is used for the token verification.</td>
</tr>
<tr>
<td><strong>client_credentials</strong></td>
<td>The name of the Kubernetes secrets object that contains the client id and client secret required to authenticate with the authentication server.</td>
</tr>
<tr>
<td><strong>claims_to_save</strong></td>
<td>The list of claims to be saved. Claims are used to create authorization policies.</td>
</tr>
</tbody>
</table>

OpenID Connect (OIDC) is a simple identity layer on top of the OAuth 2.0 protocol. OIDC allows clients to verify the identity of the end-user based on the authentication performed by an authorization server, as well as to obtain basic profile information about the end-user. In addition to the OAuth attributes, you can use the following attributes to configure OIDC.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>metadata_url</strong></td>
<td>Specifies the URL that is used to get OAUTH or OIDC provider metadata.</td>
</tr>
<tr>
<td><strong>user_field</strong></td>
<td>Specifies the attribute in the token from which the user name should be extracted. By default, Citrix ADC examines the email attribute for user ID.</td>
</tr>
<tr>
<td><strong>default_group</strong></td>
<td>Specifies the group assigned to the request if authentication succeeds. This group is in addition to any extracted groups from the token.</td>
</tr>
<tr>
<td><strong>grant_type</strong></td>
<td>Specifies the type of flow to the token end point. The default value is <strong>CODE</strong>.</td>
</tr>
</tbody>
</table>
Citrix ADC ingress controller

**Attribute Description**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pkce</td>
<td>Specifies whether to enable Proof Key for Code Exchange (PKCE). The default value is <strong>ENABLED</strong>.</td>
</tr>
<tr>
<td>token_ep_auth_method</td>
<td>Specifies the authentication method to be used with the token end point. The default value is <strong>client_secret_post</strong>.</td>
</tr>
</tbody>
</table>

**SAML authentication**

Security assertion markup language (SAML) is an XML-based open standard which enables authentication of users across products or organizations. The SAML authentication mechanism, requires an external identity provider to authenticate the client. SAML works by transferring the client identity from the identity provider to the Citrix ADC. On successful validation of the client identity, the Citrix ADC grants access to the client.

The following are the attributes for SAML authentication.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>metadata_url</td>
<td>Specifies the URL used for obtaining SAML metadata.</td>
</tr>
<tr>
<td>metadata_refresh_interval</td>
<td>Specifies the interval in minutes for fetching metadata from the specified metadata URL.</td>
</tr>
<tr>
<td>signing_cert</td>
<td>Specifies the SSL certificate to sign requests from the service provider (SP) to the identity provider (IdP).</td>
</tr>
<tr>
<td>audience</td>
<td>Specifies the identity of the service or application for which the token is applicable.</td>
</tr>
<tr>
<td>issuer_name</td>
<td>Specifies the name used in requests sent from SP to IdP to identify the Citrix ADC.</td>
</tr>
<tr>
<td>binding</td>
<td>Specifies the transport mechanism of the SAML message. The default value is <strong>POST</strong>.</td>
</tr>
<tr>
<td>artifact_resolution_service_url</td>
<td>Specifies the URL of the artifact resolution service on IdP.</td>
</tr>
<tr>
<td>logout_binding</td>
<td>Specifies the transport mechanism of the SAML logout. The default value is <strong>POST</strong>.</td>
</tr>
</tbody>
</table>
Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reject_unsigned_assertion</td>
<td>Rejects unsigned SAML assertions. If this value is <strong>ON</strong>, it rejects assertion without signature.</td>
</tr>
<tr>
<td>user_field</td>
<td>Specifies the SAML user ID specified in the SAML assertion</td>
</tr>
<tr>
<td>default_authentication_group</td>
<td>Specifies the default group that is chosen when the authentication succeeds in addition to extracted groups.</td>
</tr>
<tr>
<td>skewtime</td>
<td>Specifies the allowed clock skew time in minutes on an incoming SAML assertion.</td>
</tr>
<tr>
<td>attributes_to_save</td>
<td>Specifies the list of attribute names separated by commas which needs to be extracted and stored as key-value pairs for the session on Citrix ADC.</td>
</tr>
</tbody>
</table>

**LDAP authentication**

LDAP (Lightweight Directory Access Protocol) is an open, vendor-neutral, industry standard application protocol for accessing and maintaining distributed directory information services over an Internet Protocol (IP) network. A common use of LDAP is to provide a central place to store user names and passwords. LDAP allows many different applications and services to connect to the LDAP server to validate users.

**Note:**

LDAP authentication is supported through both the authentication mechanisms using the request header or using forms.

The following are the attributes for LDAP authentication.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server_ip</td>
<td>Specifies the IP address assigned to the LDAP server.</td>
</tr>
<tr>
<td>server_name</td>
<td>Specifies the LDAP server name as an FQDN.</td>
</tr>
<tr>
<td>server_port</td>
<td>Specifies the port on which the LDAP server accepts connections. The default value is 389.</td>
</tr>
</tbody>
</table>
### Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>Specifies the base node on which to start LDAP searches. If the LDAP server is running locally, the default value of base is <code>dc=netscaler, dc=com</code>.</td>
</tr>
<tr>
<td>server_login_credentials</td>
<td>Specifies the Kubernetes secret object providing credentials to log in to the LDAP server. The secret data should have user name and password.</td>
</tr>
<tr>
<td>login_name</td>
<td>Specifies the <strong>LDAP login name</strong> attribute. The Citrix ADC uses the LDAP login name to query external LDAP servers or Active Directories.</td>
</tr>
<tr>
<td>security_type</td>
<td>Specifies the type of security used for communications between the Citrix ADC and the LDAP server. The default is TLS.</td>
</tr>
<tr>
<td>validate_server_cert</td>
<td>Validates LDAP server certificates. The default value is NO.</td>
</tr>
<tr>
<td>hostname</td>
<td>Specifies the host name for the LDAP server. If <code>validate_server_cert</code> is ON, this value must be the host name on the certificate from the LDAP. A host name mismatch causes a connection failure.</td>
</tr>
<tr>
<td>sub_attribute_name</td>
<td>Specifies the LDAP group subattribute name. This attribute is used for group extraction from the LDAP server.</td>
</tr>
<tr>
<td>group_attribute_name</td>
<td>Specifies the LDAP group attribute name. This attribute is used for group extraction on the LDAP server.</td>
</tr>
</tbody>
</table>
Attribute | Description
---|---
**search_filter** | Specifies the string to be combined with the default LDAP user search string to form the search value. For example, if the search filter “vpnallowed=true” is combined with the LDAP login name “samaccount” and the user-supplied user name is “bob”, the result is the LDAP search string “(vpnallowed=true)(samaccount=bob)”. Enclose the search string in two sets of double quotation marks.

**auth_timeout** | Specifies the number of seconds the Citrix ADC waits for a response from the server. The default value is 3.

**password_change** | Allows password change requests. The default value is **DISABLED**.

**attributes_to_save** | List of attribute names separated by comma which needs to be fetched from the LDAP server and stored as key-value pairs for the session on Citrix ADC.

**Authentication policies**

The **authentication_policies** allow you to define the traffic selection criteria to apply the authentication mechanism and also to specify the provider that you want to use for the selected traffic.

Authentication policy supports two formats through which you can specify authentication rules:

- resource format
- expression format

The following are the attributes for policies with resource format:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
</table>
| **path** | An array of URL path prefixes that refer to a specific API endpoint. For example, /api/v1/products/.
### Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>method</strong></td>
<td>An array of HTTP methods. Allowed values are GET, PUT, POST, or DELETE. The traffic is selected if the incoming request URI matches with any of the paths and any of the listed methods. If the method is not specified then the path alone is used for the traffic selection criteria.</td>
</tr>
<tr>
<td><strong>provider</strong></td>
<td>Specifies the authentication mechanism that needs to be used. If the authentication mechanism is not provided, then authentication is not performed.</td>
</tr>
</tbody>
</table>

The following attributes are for authentication policies with expression format:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>expression</strong></td>
<td>Specifies Citrix ADC expression to be evaluated based on authentication</td>
</tr>
<tr>
<td><strong>provider</strong></td>
<td>Specifies the authentication mechanism that needs to be used. If the authentication mechanism is not provided, then authentication is not performed.</td>
</tr>
</tbody>
</table>

**Note:**

If you want to skip authentication for a specific end point, create a policy with the `provider` attribute set as empty list. Otherwise, the request is denied.

**Authorization policies**

Authorization policies allow you to define the traffic selection criteria to apply the authorization requirements for the selected traffic.

Authorization policy supports two formats through which the you can specify the authorization rules:

- resource format
- expression format

The following are the attributes for authorization policies with resource format:
## Citrix ADC ingress controller

### Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>path</strong></td>
<td>An array of URL path prefixes that refer to a specific API endpoint. For example, <code>/api/v1/products/</code>.</td>
</tr>
<tr>
<td><strong>method</strong></td>
<td>An array of HTTP methods. Allowed values are GET, PUT, POST, or DELETE.</td>
</tr>
<tr>
<td><strong>claims</strong></td>
<td>Specifies the claims required to access a specific API endpoint. <strong>name</strong> indicates the claim name and <strong>values</strong> indicate the required permissions. You can have more than one claim. If an empty list is specified, it implies that authorization is not required. <strong>Note:</strong> Any claim that needs to be used for authorization, should be saved as part of authentication.</td>
</tr>
</tbody>
</table>

The following are the attributes for authorization policies with expression format:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>expression</strong></td>
<td>Specifies an expression to be evaluated for authorization.</td>
</tr>
</tbody>
</table>

**Note:**

Citrix ADC requires both authentication and authorization policies for the API traffic. Therefore, you must configure an authorization policy with an authentication policy. Even if you do not have any authorization checks, you must create an authorization policy with empty claims. Otherwise, the request is denied with a 403 error.

**Note:**

Authorization would be successful if the incoming request matches a policy (path, method, and claims). All policies are tried until there is a match. If it is required to selectively bypass authorization for a specific endpoint, an explicit policy needs to be created.

### Deploy the Auth CRD

Perform the following to deploy the Auth CRD:
1. Download the CRD (`auth-crd.yaml`).

2. Deploy the Auth CRD using the following command:

```bash
kubectl create -f auth-crd.yaml
```

For example:

```bash
root@master:~# kubectl create -f auth-crd.yaml
customresourcedefinition.apiextensions.k8s.io/authpolicies.citrix.com created
```

How to write authentication and authorization policies

After you have deployed the CRD provided by Citrix in the Kubernetes cluster, you can define the authentication policy configuration in a `.yaml` file. In the `.yaml` file, use `authpolicy` in the `kind` field and in the `spec` section add the Auth CRD attributes based on your requirement for the policy configuration.

After you deploy the `.yaml` file, the Citrix ingress controller applies the authentication policy configuration on the Ingress Citrix ADC device.

Local auth provider

The following is a sample authentication and authorization policy definition for the local-auth-provider type (`local_auth.yaml`).

```yaml
apiVersion: citrix.com/v1beta1
kind: authpolicy
metadata:
  name: authexample
spec:
  servicenames:
    - frontend
  authentication_providers:
    - name: "local-auth-provider"
      basic_local_db:
        use_local_auth: 'YES'
```
The sample policy definition performs the following:

- Citrix ADC performs the local authentication on the requests to the following:
  - GET or POST operation on orders and shipping end points.
- Citrix ADC does not perform the authentication for GET operation on the products endpoint.
- Citrix ADC does not apply any authorization permissions.

OAuth JWT verification

The following is a sample authentication and authorization policy definition for OAuth JWT verification (oauth_jwt_auth.yaml).
authentication_providers:
  - name: "jwt-auth-provider"
    oauth:
      issuer: "https://sts.windows.net/tenant1/"
      jwks_uri: "https://login.microsoftonline.com/tenant1/
        discovery/v2.0/keys"
      audience: ["https://api.service.net"]
      claims_to_save: ["scope"]

authentication_policies:
  - resource:
      path:
        - '/orders/
        - '/shipping/
      method: [GET, POST]
      provider: ["jwt-auth-provider"]

    # skip authentication for this
    - resource:
      path:
        - '/products/
      method: [GET]
      provider: []

authorization_policies:
  - resource:
      path:
        - '/orders/
        - '/shipping/
      method: [POST]
      claims:
        - name: "scope"
          values: ["read", "write"]
      # skip authorization, no claims required
      - resource:
          path:
            - '/orders/
          method: [GET]
          claims:
            - name: "scope"
              values: ["read"]

© 1999–2022 Citrix Systems, Inc. All rights reserved.
The sample policy definition performs the following:

- Citrix ADC performs JWT verification on the requests to the following:
  - The GET or POST operation on orders and shipping endpoints.
- Citrix ADC skips authentication for the GET operation on the products endpoint.
- Citrix ADC requires the scope claim with read and write permissions for POST operation on orders and shipping endpoints.
- Citrix ADC requires the scope claim with the read permission for GET operation on the orders endpoint.
- Citrix ADC does not need any permissions for GET operation on the shipping end point.

For OAuth, if the token is present in a custom header, it can be specified using the token_in_hdr attribute as follows:

```
oauth:
  issuer: "https://sts.windows.net/tenant1/
  jwks_uri: "https://login.microsoftonline.com/tenant1/discovery/v2.0/keys"
  audience : ["https://vault.azure.net"]
  token_in_hdr : [ "custom-hdr1" ]
```

Similarly, if the token is present in a query parameter, it can be specified using the token_in_param attribute as follows:

```
oauth:
  issuer: "https://sts.windows.net/tenant1/
  jwks_uri: "https://login.microsoftonline.com/tenant1/discovery/v2.0/keys"
  audience : ["https://vault.azure.net"]
  token_in_param : [ "query-param1" ]
```

**oAuth Introspection**

The following is a sample authentication and authorization policy definition for oAuth JWT verification. (oauth_intro_auth.yaml)
```yaml
apiVersion: citrix.com/v1beta1
kind: authpolicy
metadata:
  name: authexample
spec:
  servicenames:
    - frontend
  authentication_providers:
    - name: "introspect-provider"
      oauth:
        issuer: "ns-idp"
        jwks_url: "https://idp.aaa/oauth/idp/certs"
        audience: ["https://api.service.net"]
        client_credentials: "oauthsecret"
        introspect_url: https://idp.aaa/oauth/idp/introspect
        claims_to_save: ["scope"]
  authentication_policies:
    - resource:
      path: []
      method: []
      provider: ["introspect-provider"]
  authorization_policies:
    - resource:
      path: []
      method: [POST]
      claims:
        - name: "scope"
          values: ["read", "write"]
    - resource:
      path: []
      method: [GET]
      claims:
        - name: "scope"
          values: ["read"]
```

The sample policy definition performs the following:

- Citrix ADC performs the OAuth introspection as specified in the provider `introspect-provider` for all requests.
- Citrix ADC requires the scope claim with `read` and `write` permissions for all `POST` requests.
Citrix ADC requires the scope claim with the read permission for all GET requests.

Creating a secrets object with client credentials for introspection

A Kubernetes secrets object is needed for configuring the OAuth introspection.

You can create a secret object in a similar way as shown in the following example:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: oauthsecret
type: Opaque
stringData:
  client_id: "nsintro"
  client_secret: "nssintro"
```

Note:
Keys of the opaque secret object must be client_id and client_secret. A user can set the values for them as desired.

SAML authentication using forms

The following is an example for SAML authentication using forms. In the example, authhost-tls-cert-secret and saml-tls-cert-secret are Kubernetes TLS secrets referring to certificate and key.

Note:
When certkey.cert and certkey.key are certificate and key respectively for the authentication host, then the authhost-tls-cert-secret can be formed using the following command:

```bash
kubectl create secret tls authhost-tls-cert-secret --key="certkey.key" --cert="certkey.cert"
```

Similarly, you can use this command to form saml-tls-cert-secret with the required certificate and key.

```yaml
apiVersion: citrix.com/v1beta1
kind: authpolicy
```
```yaml
metadata:
  name: samlexample
spec:
  servicenames:
    - frontend
  
  authentication_mechanism:
    using_forms:
      authentication_host: "fqdn_authenticate_host"
      authentication_host_cert:
        tls_secret: authhost-tls-cert-secret
      listener_name: "example-listener"

authentication_providers:
  - name: "saml-auth-provider"
    saml:
      metadata_url: "https://idp.aaa/metadata/samlidp/aaa"
      signing_cert:
        tls_secret: saml-tls-cert-secret

authentication_policies:
  - resource:
      path: []
      method: []
      provider: ["saml-auth-provider"]

authorization_policies:
  - resource:
      path: []
      method: []
      claims: []
```

The sample policy definition performs the following:

- Citrix ADC performs SAML authentication as specified in the provider `saml-auth-provider` for all requests.
  
  **Note**: Granular authentication is not supported for the forms mechanism.

- Citrix ADC requires the group claim with `admin` permission for all POST requests.

- Citrix ADC does not require any specific permission for GET requests.
OpenID Connect authentication using forms

The following is an example for creating OpenID Connect authentication to configure Citrix ADC in a Relaying Party (RP) role to authenticate users for an external identity provider. The `authentication_mechanism` must be set to `using_forms` to trigger the OpenID Connect procedures.

```yaml
apiVersion: citrix.com/v1beta1
kind: authpolicy
metadata:
  name: authoidc
spec:
servicenames:
  - frontend
    authentication_mechanism:
      using_forms:
        authentication_host: "10.221.35.213"
        authentication_host_cert:
          tls_secret: "oidc-tls-secret"
        ingress_name: "example-ingress"
authentication_providers:
  - name: "oidc-provider"
    oauth:
      audience: ["https://app1.citrix.com"]
      client_credentials: "oidcsecret"
      metadata_url: "https://10.221.35.214/oauth/idp/.well-known/openid-configuration"
      default_group: "groupA"
      user_field: "sub"
      pkce: "ENABLED"
      token_ep_auth_method: "client_secret_post"

authentication_policies:
  - resource:
      path: []
      method: []
      provider: ["oidc-provider"]
authorization_policies:
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
#default - no authorization requirements
  - resource:
    path: []
    method: []
    claims: []

The sample policy definition performs the following:

- Citrix ADC performs OIDC authentication (relying party) as specified in the provider `oidc-provider` for all requests.
  
  **Note:** Granular authentication is not supported for the forms mechanism.

- Citrix ADC does not require any authorization permissions.

**LDAP authentication using the request header**

The following is an example for LDAP authentication using the request header.

In this example, `ldapcredential` is the Kubernetes secret referring to the LDAP server credentials. See the `ldap_secret.yaml` file for information on how to create LDAP server credentials.
Note: With the request header based authentication mechanism, granular authentication based on traffic is supported.

LDAP authentication using forms

In the example authhost-tls-cert-secret is the Kubernetes TLS secret referring to certificate and key.

When certkey.cert and certkey.key are certificate and key respectively for the authentication host, then the authhost-tls-cert-secret can be formed using the following command:

```bash
kubectl create secret tls authhost-tls-cert-secret --key="certkey.key" --cert="certkey.cert"
```

In this example, ldapcredential is the Kubernetes secret referring to the LDAP server credentials. See the ldap_secret.yaml file for information on how to create LDAP server credentials.
The sample policy definition performs the following:

- Citrix ADC performs the LDAP authentication for entire traffic (all requests).
- Citrix ADC does not apply any authorization permission.

The following is an example for `LDAP_secret.yaml`.

```yaml
apiVersion: v1
type: Opaque
```
**Example for Citrix ADC expression support with Auth CRD**

This example shows how you can specify Citrix ADC expressions along with authentication and authorization policies:

```yaml
apiVersion: citrix.com/v1beta1
kind: authpolicy
metadata:
  name: authexample
spec:
  servicenames:
    - frontend
  authentication_mechanism:
    using_request_header: 'ON'
  authentication_providers:
    - name: "ldap-auth-provider"
      ldap:
        server_ip: "192.2.156.160"
        base: 'dc=aaa,dc=local'
        login_name: accountname
        sub_attribute_name: CN
        server_login_credentials: ldapcredential
        # "memberof" attribute details are extracted from LDAP server.
        attributes_to_save: memberof
  authentication_policies:
    # Perform LDAP authentication for the host hotdrink.beverages.com
    - expression: 'HTTP.REQ.HOSTNAME.SET_TEXT_MODE(IGNORECASE).EQ("hotdrink.beverages.com")'
      provider: ["ldap-auth-provider"]
```

---

© 1999–2022 Citrix Systems, Inc. All rights reserved.
Rate limiting in Kubernetes using Citrix ADC

February 3, 2022

In a Kubernetes deployment, you can rate limit the requests to the resources on the back end server or services using rate limiting feature provided by the ingress Citrix ADC.

Citrix provides a Kubernetes CustomResourceDefinitions (CRDs) called the Rate limit CRD that you can use with the Citrix ingress controller to configure the rate limiting configurations on the Citrix ADCs used as Ingress devices.

Apart from rate limiting the requests to the services in a Kubernetes environment, you can use the Rate limit CRD for API security as well. The Rate limit CRD allows you to limit the REST API request to API servers or specific API endpoints on the API servers. It monitors and keeps track of the requests to the API server or endpoints against the allowed limit per time slice and hence protects from attacks such as the DDoS attack.

You can enable logging for observability with the rate limit CRD. Logs are stored on Citrix ADC which can be viewed by checking the logs using the shell command. The file location is based on the syslog configuration. For example, /var/logs/ns.log.

Rate limit CRD definition

The Rate limit CRD spec is available in the Citrix ingress controller GitHub repo at: ratelimit-crd.yaml. The Rate limit CRD provides attributes for the various options that are required to define the rate limit policies on the Ingress Citrix ADC that acts as an API gateway.

Rate limit CRD attributes

The following table lists the various attributes provided in the Rate limit CRD:

<table>
<thead>
<tr>
<th>CRD attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servicename</td>
<td>The list of Kubernetes services to which you want to apply the rate limit policies.</td>
</tr>
<tr>
<td>CRD attribute</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>selector_keys</td>
<td>The traffic selector keys that filter the traffic to identify the API requests against which the throttling is applied and monitored.</td>
</tr>
</tbody>
</table>

**Note:** The `selector_keys` is an optional attribute. You can choose to configure zero, one or more of the selector keys. If more than one selector keys are configured then it is considered as a logical AND expression.

In this version of the Rate limit CRD, `selector_keys` provides the basic configuration section that you can use to configure the following commonly used traffic characteristics as keys against which the configured limits are monitored and throttled:

- **path:** An array of URL path prefixes that refer to a specific API endpoint. For example, `/api/v1/products/`.
- **method:** An array of HTTP methods. Allowed values are GET, PUT, POST, or DELETE.
- **header_name:** HTTP header that has the unique API client or user identifier. For example, `X-apikey` which comes with a unique API-key that identifies the API client sending the request.
- **per_client_ip:** Allows you to monitor and apply the configured threshold to each API request received per unique client IP address.

<table>
<thead>
<tr>
<th>req_threshold</th>
<th>The maximum number of requests that are allowed in the given time slice (request rate).</th>
</tr>
</thead>
<tbody>
<tr>
<td>timeslice</td>
<td>The time interval specified in microseconds (multiple of 10 s), during which the requests are monitored against the configured limits. If not specified it defaults to 1000 milliseconds.</td>
</tr>
<tr>
<td>CRD attribute</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>limittype</td>
<td>It allows you to configure the type of throttling algorithms that you want to use to apply the limit. Supported algorithms are burst and smooth. The default is the <strong>burst</strong> mode.</td>
</tr>
<tr>
<td>throttle_action</td>
<td>It allows you to define the throttle action that needs to be taken on the traffic that is throttled for crossing the configured threshold. The following are the throttle action that you can define:</td>
</tr>
<tr>
<td></td>
<td><strong>DROP:</strong> Drops the requests above the configured traffic limits.</td>
</tr>
<tr>
<td></td>
<td><strong>RESET:</strong> Resets the connection for the requests crossing the configured limit.</td>
</tr>
<tr>
<td></td>
<td><strong>REDIRECT:</strong> Redirects the traffic to the configured redirect_url.</td>
</tr>
<tr>
<td></td>
<td><strong>RESPOND:</strong> Responds with the standard “<strong>429 Too many requests</strong>” response.</td>
</tr>
<tr>
<td>redirect_url</td>
<td>This attribute is an optional attribute that is required only if throttle_action is configured with the value REDIRECT.</td>
</tr>
<tr>
<td>logpackets</td>
<td>Enables audit logs.</td>
</tr>
<tr>
<td>logexpression</td>
<td>Specifies the default-syntax expression that defines the format and content of the log message.</td>
</tr>
<tr>
<td>loglevel</td>
<td>Specifies the severity level of the log message that is generated.</td>
</tr>
</tbody>
</table>

**Deploy the Rate limit CRD**

Perform the following to deploy the Rate limit CRD:

1. Download the CRD (*ratelimit-crd.yaml*).
2. Deploy the Rate limit CRD using the following command:
How to write a rate-based policy configuration

After you have deployed the CRD provided by Citrix in the Kubernetes cluster, you can define the rate-based policy configuration in a .yaml file. In the .yaml file, use ratelimit in the kind field and in the spec section add the Rate limit CRD attributes based on your requirement for the policy configuration.

After you deploy the .yaml file, the Citrix ingress controller applies the rate-based policy configuration on the Ingress Citrix ADC device.

Following are some examples for rate limit policy configurations.

Limit API requests to configured API endpoint prefixes

Consider a scenario wherein you want to define a rate-based policy in Citrix ADC to limit the API requests to 15 requests per minute from each unique client IP address to the configured API endpoint prefixes. Create a .yaml file called ratelimit-example1.yaml and use the appropriate CRD attributes to define the rate-based policy as follows:
Limit API requests to calendar APIs

Consider a scenario wherein you want to define a rate-based policy in a Citrix ADC to limit the API requests (GET or POST) to five requests from each API client identified using the HTTP header X-API-Key to the calendar APIs. Create a .yaml file called `ratelimit-example2.yaml` and use the appropriate CRD attributes to define the rate-based policy as follows:

```yaml
apiVersion: citrix.com/v1beta1
kind: ratelimit
metadata:
  name: throttle-calendarapi-perapikey
spec:
servicenames:
  - frontend
selector_keys:
  basic:
  path:
```
After you have defined the policy configuration, deploy the .yaml file using the following command:

```
root@master:~# kubectl create -f ratelimit-example2.yaml
ratelimit.citrix.com/throttle-req-per-clientip created
```

The Citrix ingress controller applies the policy configuration on the Ingress Citrix ADC device.

### Use Rewrite and Responder policies in Kubernetes

February 15, 2022

In a Kubernetes environment, to deploy specific layer 7 policies to handle scenarios such as:

- Redirecting HTTP traffic to a specific URL
- Blocking a set of IP addresses to mitigate DDoS attacks
- Imposing HTTP to HTTPS

Requires you to add appropriate libraries within the microservices and manually configure the policies. Instead, you can use the Rewrite and Responder features provided by the Ingress Citrix ADC device to deploy these policies.

Citrix provides Kubernetes CustomResourceDefinitions (CRDs) that you can use with the Citrix ingress controller to automate the configurations and deployment of these policies on the Citrix ADCs used as Ingress devices.

The Rewrite and Responder CRD provided by Citrix is designed to expose a set of tools used in frontline Citrix ADCs. Using these functionalities you can rewrite the header and payload of ingress and egress HTTP traffic as well as respond to HTTP traffic on behalf of a microservice.
Once you deploy the Rewrite and Responder CRD in the Kubernetes cluster. You can define extensive rewrite and responder policies using datasets, pat sets, and string maps and also enable audit logs for statistics on the ingress device. For more information on the rewrite and responder policy feature provided by Citrix ADC, see Rewrite policy and Responder policy.

Note:
The Rewrite and Responder CRD is not supported for OpenShift routes. You can use OpenShift ingress to use Rewrite and Responder CRD.

**Deploy the Citrix Rewrite and Responder CRD**

The Citrix Rewrite and Responder CRD deployment YAML file: `rewrite-responder-policies-deployment.yaml`.

Note:
Ensure that you do not modify the deployment YAML file.

Deploy the CRD, using the following command:

```bash
kubectl create -f rewrite-responder-policies-deployment.yaml
```

For example,

```
root@master:~# kubectl create -f rewrite-responder-policies-deployment.yaml
```

**Rewrite and Responder CRD attributes**

The CRD provides attributes for the various options required to define the rewrite and responder policies. Also, it provides attributes for dataset, pat set, string map, and audit logs to use within the rewrite and responder policies. These CRD attributes correspond to Citrix ADC command and attribute respectively.

**Rewrite policy**

The following table lists the CRD attributes that you can use to define a rewrite policy. Also, the table lists the corresponding Citrix ADC command and attributes.
### Citrix ADC ingress controller

<table>
<thead>
<tr>
<th>CRD attribute</th>
<th>Citrix ADC command</th>
<th>Citrix ADC attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>rewrite-criteria</td>
<td>Add rewrite policy</td>
<td>rule</td>
</tr>
<tr>
<td>default-action</td>
<td>Add rewrite policy</td>
<td>undefAction</td>
</tr>
<tr>
<td>operation</td>
<td>Add rewrite action</td>
<td>type</td>
</tr>
<tr>
<td>target</td>
<td>Add rewrite action</td>
<td>target</td>
</tr>
<tr>
<td>modify-expression</td>
<td>Add rewrite action</td>
<td>stringBuilderExpr</td>
</tr>
<tr>
<td>multiple-occurrence-modify</td>
<td>Add rewrite action</td>
<td>Search</td>
</tr>
<tr>
<td>additional-multiple-occurrence-modify</td>
<td>Add rewrite action</td>
<td>RefineSearch</td>
</tr>
<tr>
<td>Direction</td>
<td>Bind lb vserver</td>
<td>Type</td>
</tr>
</tbody>
</table>

### Responder policy

The following table lists the CRD attributes that you can use to define a responder policy. Also, the table lists the corresponding Citrix ADC command and attributes.

<table>
<thead>
<tr>
<th>CRD attribute</th>
<th>Citrix ADC command</th>
<th>Citrix ADC attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redirect</td>
<td>Add responder action</td>
<td>Type (the value of type)</td>
</tr>
<tr>
<td>url</td>
<td>Add responder action</td>
<td>Target</td>
</tr>
<tr>
<td>redirect-status-code</td>
<td>Add responder action</td>
<td>responseStatusCode</td>
</tr>
<tr>
<td>redirect-reason</td>
<td>Add responder action</td>
<td>reasonPhrase</td>
</tr>
<tr>
<td>Respond-with</td>
<td>Add responder action</td>
<td>Type (the value of type)</td>
</tr>
<tr>
<td>http-payload-string</td>
<td>Add responder action</td>
<td>Target</td>
</tr>
<tr>
<td>Noop</td>
<td>Add responder policy</td>
<td>Action (the value of action)</td>
</tr>
<tr>
<td>Reset</td>
<td>Add responder policy</td>
<td>Action (the value of action)</td>
</tr>
<tr>
<td>Drop</td>
<td>Add responder policy</td>
<td>Action (the value of action)</td>
</tr>
<tr>
<td>Respond-criteria</td>
<td>Add responder policy</td>
<td>Rule</td>
</tr>
<tr>
<td>Default-action</td>
<td>Add responder policy</td>
<td>undefAction</td>
</tr>
</tbody>
</table>
### Audit log

The following table lists the CRD attributes provide to enable audit log within the rewrite or responder policies. Also, the table lists the corresponding Citrix ADC command and attributes.

<table>
<thead>
<tr>
<th>CRD attribute</th>
<th>Citrix ADC command</th>
<th>Citrix ADC attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logexpression</td>
<td>Add audit message action</td>
<td>stringBuilderExpr</td>
</tr>
<tr>
<td>Loglevel</td>
<td>Add audit message action</td>
<td>Loglevel</td>
</tr>
</tbody>
</table>

### Dataset

The following table lists the CRD attributes for dataset that you can use within the rewrite or responder policies. Also, the table lists the corresponding Citrix ADC command and attributes.

<table>
<thead>
<tr>
<th>CRD attribute</th>
<th>Citrix ADC command</th>
<th>Citrix ADC attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Add policy dataset</td>
<td>Name</td>
</tr>
<tr>
<td>Type</td>
<td>Add policy dataset</td>
<td>Type</td>
</tr>
<tr>
<td>Values</td>
<td>Bind policy dataset</td>
<td>Value</td>
</tr>
</tbody>
</table>

### Patset

<table>
<thead>
<tr>
<th>CRD attribute</th>
<th>Citrix ADC command</th>
<th>Citrix ADC attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Add policy patset</td>
<td>Name</td>
</tr>
<tr>
<td>Values</td>
<td>Bind policy patset</td>
<td>string</td>
</tr>
</tbody>
</table>

### String map

<table>
<thead>
<tr>
<th>CRD attribute</th>
<th>Citrix ADC command</th>
<th>Citrix ADC attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Add policy stringmap</td>
<td>Name</td>
</tr>
<tr>
<td>Key</td>
<td>Bind policy stringmap</td>
<td>Key</td>
</tr>
<tr>
<td>Value</td>
<td>Bind policy stringmap</td>
<td>Value</td>
</tr>
</tbody>
</table>
**Goto-priority-expression**

The following table provides information about the goto-priority-expression attribute, which is a CRD attribute for binding a group of multiple consecutive policies to services.

<table>
<thead>
<tr>
<th>CRD attribute</th>
<th>Citrix ADC command</th>
<th>Citrix ADC attribute</th>
<th>Supported values</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto-priority-express</td>
<td>Bind lb vserver</td>
<td>gotoPriorityExpress</td>
<td>NEXT and END</td>
<td>End</td>
</tr>
</tbody>
</table>

For more information on how to use the goto-priority-expression attribute, see the example Modify strings and host name in the requested URL

**How to write a policy configuration**

After you have deployed the CRD provided by Citrix in the Kubernetes cluster, you can define the policy configuration in a .yaml file. In the .yaml file, use rewritepolicy in the kind field and based on your requirement add any of the following individual sections in spec for policy configuration.

- rewrite-policy - To define rewrite policy configuration.
- responder-policy - To define responder policy configuration.
- logpackets - To enable audit logs.
- dataset - To use a data set for extensive policy configuration.
- patset - To use a pat set for extensive policy configuration.
- stringmaps - To use string maps for extensive policy configuration.

In these sections, you need to use the CRD attributes provided for the respective policy configuration (rewrite or responder) to define the policy.

Also, in the spec section, you need to include a rewrite-policies section to specify the service or services to which the policy must be applied. For more information, see Sample policy configurations.

After you deploy the .yaml file, the Citrix ingress controller applies the policy configuration on the Ingress Citrix ADC device.

**Guidelines for the policy configuration**

- If the CRD is associated with a namespace then, by default, the policy is applied to the services associated with the namespace. For example, if you have the same service name associated with multiple namespaces, then the policy is applied to the service that belongs to the namespace associated with the CRD.
Citrix ADC ingress controller

- If you have defined multiple policies in a single .yaml file then the first policy configuration defined in the file takes priority and the subsequent policy configurations is applied as per the sequence. If you have multiple policies defined in different files then the first policy configuration defined in the file that you deployed first takes priority.

Guidelines for the usage of Goto-priority-expression

- The rewrite and responder policies can be combined as multiple groups using the NEXT keyword within the goto-priority-expression field.

- When the goto-priority-expression field is NEXT within the current policy and if the current policy evaluates to True, the next policy in the group is executed and the flow moves to the next consecutive policies unless the goto-priority-expression field points to END.

- When the current policy evaluates to FALSE, the goto-priority-expression has no impact, as the policy execution stops at the current policy.

- The rewrite or responder policy group within the rewrite or responder policies begins with the policy assigned with goto-priority-expression as NEXT and includes all the consecutive policies until the goto-priority-expression field is assigned to END.

- When you group rewrite or responder policies using goto-priority-expression, the service names bound to the policies within the group should be the same.

- The last policy within the rewrite-policies or responder-policies should always have the goto-priority-expression as END.

- If the goto-priority-expression field is not specified for a policy, the default value of END is assigned to goto-priority-expression.

Note:

For more information on how to use the goto-priority-expression field, see the example Modify strings and host name in the requested URL.

Create and verify a rewrite and responder policy

Consider a scenario where you want to define a policy in Citrix ADC to rewrite all the incoming URLs to new-url-for-the-application and send it to the microservices. Create a .yaml file called target-url-rewrite.yaml and use the appropriate CRD attributes to define the rewrite policy as follows:

```yaml
1 apiVersion: citrix.com/v1
2 kind: rewritepolicy
```
After you have defined the policy configuration, deploy the `.yaml` file using the following command:

```bash
kubectl create -f target-url-rewrite.yaml
```

After you deploy the `.yaml` file, the Citrix ingress controller applies the policy configuration on the Ingress Citrix ADC device.

On the master node in the Kubernetes cluster, you can verify the status of the applied rewrite policy CRD using the following command:

```bash
Kubectl get rewritepolicies.citrix.com targeturlrewrite
```

You can view the status as follows:

<table>
<thead>
<tr>
<th>NAME</th>
<th>STATUS</th>
<th>MESSAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>targeturlrewrite</td>
<td>Success</td>
<td>CRD Activated</td>
</tr>
</tbody>
</table>

If there are issues while creating or applying the CRD, the same can be debugged using the citrix-k8s-ingress-controller logs.
Also, you can verify whether the configuration is applied on the Citrix ADC by using the following steps.

1. Log on to the Citrix ADC command-line.
2. Use the following command to verify if the configuration is applied to the Citrix ADC:

```
1 show run | grep `lb vserver`
2 add lb vserver k8s-citrix_default_80_k8s-citrix-svc Default_80_svc
   HTTP 0.0.0.0 0 -persistenceType NONE -cltTimeout 180
3 bind lb vserver k8s-citrix_default_80_k8s-citrix-
   svc_default_80_svc k8s-citrix_default_80_k8s-citrix-
   svc_default_80_svc
4 bind lb vserver k8s-citrix_default_80_k8s-citrix-
   svc_default_80_svc -policyName
   k8s_crd_rewritepolicy_rwpolicy_targeturlrewrite_0_default -
   priority 100300076 -gotoPriorityExpression END -type REQUEST
```

You can verify that the policy `k8s_crd_rewritepolicy_rwpolicy_targeturlrewrite_0_default` is bound to the load balancing virtual server.

**Sample policy configurations**

**Responder policy configuration**

Following is a sample responder policy configuration (`block-list-urls.yaml`):

```
1 apiVersion: citrix.com/v1
2 kind: rewritepolicy
3 metadata:
4   name: blocklisturls
5 spec:
6   responder-policies:
7     servicenames:
8       citrix-svc
9     responder-policy:
10    respondwith:
11       http-payload-string: '\"HTTP/1.1 401 Access denied\"'
12    respond-criteria: 'http.req.url.equals_any("blocklistUrls")'
13    comment: 'Blocklist certain Urls'
```
Citrix ADC ingress controller

```yaml
patset:
  - name: blocklistUrls
    values:
      - '/app1'
      - '/app2'
      - '/app3'
```

In this example, if Citrix ADC receives any URL that matches the /app1, /app2, or /app3 strings defined in the patset, Citrix ADC blocks the URL.

**Policy with audit logs enabled**

Following is a sample policy with audit logs enabled (block-list-urls-audit-log.yaml).

```yaml
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
  name: blocklisturls
spec:
  responder-policies:
    - servicenames:
      - citrix-svc
      logpackets:
        logexpression: "http.req.url"
        loglevel: INFORMATIONAL
  responder-policy:
    respondwith:
      http-payload-string: '"HTTP/1.1 401 Access denied"'
      respond-criteria: 'http.req.url.equals_any("blocklistUrls")'
    comment: 'Blocklist certain Urls'

patset:
  - name: blocklistUrls
    values:
      - '/app1'
      - '/app2'
      - '/app3'
```

© 1999‒2022 Citrix Systems, Inc. All rights reserved.
Multiple policy configurations

You can add multiple policy configurations in a single `.yaml` file and apply the policies to the Citrix ADC device. You need add separate sections for each policy configuration (multi-policy-config.yaml).

```yaml
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
  name: multipolicy
spec:
  responder-policies:
    - servicenames:
      - citrix-svc
        responder-policy:
          redirect:
            url: "www.citrix.com"
            respond-criteria: 'client.ip.src.TYPECAST_text_t.equals_any("redirectIPs")'
            comment: 'Redirect IPs to citrix.com'
    - servicenames:
      - citrix-svc
        responder-policy:
          redirect:
            url: 'HTTP.REQ.HOSTNAME+http.req.url.
                  MAP_STRING_DEFAULT_TO_KEY("modifyurls")'
            respond-criteria: 'http.req.is_valid'
            comment: 'modify specific URLs'
  rewrite-policies:
    - servicenames:
      - citrix-svc
        rewrite-policy:
          operation: insert_http_header
          target: 'sessionID'
          modify-expression: '"48592th42gl24456284536tgt2"
          comment: 'insert SessionID in header'
          direction: RESPONSE
          rewrite-criteria: 'http.res.is_valid'
  dataset:
    - name: redirectIPs
      type: ipv4
```
The example contains two responder policies and a rewrite policy, based on these policies the Citrix ADC device performs the following:

- Any client request to IP addresses defined in the `redirectIP` dataset, that is, 1.1.1.1 or 2.2.2.2 respectively, the request is redirected to `www.citrix.com`.
- Any incoming URL with strings provided in the `modifyurls` stringmap is modified to the value provided in the stringmap. For example, if the incoming URL has the string `/app1/` is modified to `/internal-app1/`
- Adds a session ID as a new header in the response to the client.

### Example use cases

#### Add response headers

When the requested URL from the client contains `/citrix-app/`, you can add the following headers in the HTTP response from the microservices to the client using a rewrite policy:

- Client source port to the header
- Server destination IP address
- random HTTP header

The following sample rewrite policy definition adds these headers to the HTTP response from the microservices to the client:

```yaml
apiVersion: citrix.com/v1
catkind: rewriterpolicy
catmetadata:
  name: addresponseheaders
```
Create a YAML file (add_response_headers.yaml) with the rewrite policy definition and deploy the YAML file using the following command:

```
kubectl create -f add_response_headers.yaml
```

You can verify the HTTP header added to the response as follows:

```
$ curl -vvv http://app.cic-citrix.org/citrix-app/
*   Trying 10.102.33.176...
*     TCP_NODELAY set
* Connected to app.cic-citrix.org (10.102.33.176) port 80 (#0)
> GET /citrix-app/ HTTP/1.1
> Host: app.cic-citrix.org
> User-Agent: curl/7.54.0
> Accept: */*
> 
< HTTP/1.1 200 OK
< Server: nginx/1.8.1
< Date: Fri, 29 Mar 2019 11:14:04 GMT
< Content-Type: text/html
< Transfer-Encoding: chunked
< Connection: keep-alive
< X-Powered-By: PHP/5.5.9-1ubuntu4.14
< x-port: 22481 ===========> NEW RESPONSE HEADER
< x-ip:10.102.33.176 ===========> NEW RESPONSE HEADER
< x-new-dummy-header: Sending_a_gift ===========> NEW RESPONSE HEADER
```
Add custom header to the HTTP response packet

Using a rewrite policy, you can add custom headers in the HTTP response from the microservices to the client.

The following sample rewrite policy definition adds a custom header to the HTTP response from the microservices to the client:

```yaml
apiVersion: citrix.com/v1
definition
kind: rewritepolicy
metadata:
  name: addcustomheaders
spec:
  rewrite-policies:
    - servicenames:
      - frontend
        rewrite-policy:
          operation: insert_before_all
          target: http.res.full_header
          modify-expression: '\r\nx-request-time:"+sys.time+"\r\nx-using
-\citrix-ingress-controller: true'"
          multiple-occurrence-modify: 'text("\r\n\r\n")'
          comment: 'Adding custom headers'
          direction: RESPONSE
          rewrite-criteria: 'http.req.is_valid'

<!-- NeedCopy -->
```

Create a YAML file (`add_custom_headers.yaml`) with the rewrite policy definition and deploy the YAML file using the following command:

```
kubectl create -f add_custom_headers.yaml
```
You can verify the custom HTTP header added to the response as follows:

```bash
$ curl -vvv http://app.cic-citrix.org/
* Trying 10.102.33.176...
* TCP_NODELAY set
* Connected to app.cic-citrix.org (10.102.33.176) port 80 (#0)
> GET / HTTP/1.1
> Host: app.cic-citrix.org
> User-Agent: curl/7.54.0
> Accept: */*
>
HTTP/1.1 200 OK
Server: nginx/1.8.1
Date: Fri, 29 Mar 2019 12:15:09 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: keep-alive
X-Powered-By: PHP/5.5.9-1ubuntu4.14

.NEW HEADER ADDED
.x-using-citrix-ingress-controller: true

.NEW HEADER ADDED
```

**Replace host name in the request**

You can define a rewrite policy as shown in the following example YAML (`http_request_modify_prefixasprefix.yaml`) to replace the host name in an HTTP request as per your requirement:

```yaml
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
  name: httpheadermodifyretainprefix
spec:
```
Citrix ADC ingress controller

```yaml
rewrite-policies:
  - servicenames:
    - frontend
  rewrite-policy:
    operation: replace_all
    target: 'http.req.header("host")'
    modify-expression: "citrix-service-app"
    multiple-occurence-modify: 'text("app.cic-citrix.org")'
    comment: 'HTTP header rewrite of hostname'
    direction: REQUEST
    rewrite-criteria: 'http.req.is_valid'
<!-- NeedCopy-->
```

Create a YAML file (http_request_modify_prefixasprefix.yaml) with the rewrite policy definition and deploy the YAML file using the following command:

```bash
kubectl create -f http_request_modify_prefixasprefix.yaml
```

You can verify the policy definition using the `curl` command. The host name in the request is replaced with the defined host name.

```bash
curl http://app.cic-citrix.org/prefix/foo/bar
```

Output:

```text
Hypertext Transfer Protocol
GET /prefix/foo/bar HTTP/1.1

[Expert Info (Chat/Sequence): GET /prefix/foo/bar HTTP/1.1]
Request Method: GET
Request URI: /prefix/foo/bar
Request Version: HTTP/1.1
Host: citrix-service-app
User-Agent: curl/7.54.0
Accept: */*

[Full request URI: http://citrix-service-app/prefix/foo/bar]
[HTTP request 1/1]
```

Modify the application root

You can define a rewrite policy to modify the application root if the existing application root is /.

The following sample rewrite policy modifies / to /citrix-approot/ in the request URL:
Create a YAML file (http_approot_request_modify.yaml) with the rewrite policy definition and deploy the YAML file using the following command:

```sh
cubectl create -f http_approot_request_modify.yaml
```

Using the `curl` command, you can verify if the application root is modified as per your requirement:

```sh
curl -vvv http://app.cic-citrix.org/
```

Output:

```plaintext
<--- NeedCopy

Modify the strings in the requested URL

You can define a rewrite policy to modify the strings in the requested URL as per your requirement. The following sample rewrite policy replaces the strings `something` to `simple` in the requested URL:
Create a YAML file (`http_url_replace_string.yaml`) with the rewrite policy definition and deploy the YAML using the following command:

```bash
kubectl create -f http_url_replace_string.yaml
```

You can verify the policy definition using a `curl` request with the string `something`. The string `something` is replaced with the string `simple` as shown in the following examples:

**Example 1:**

```bash
curl http://app.cic-citrix.org/something/simple/citrix
```

**Output:**

```
HTTP/1.1 200 OK
Date: Fri, 24 Dec 2022 12:00:00 GMT
Content-Type: text/html;charset=UTF-8
Content-Length: 36

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>Redirected</title>
</head>
<body>

<h1>Redirected</h1>
<p>The document has been redirected from http://app.cic-citrix.org/something/simple/citrix to http://app.cic-citrix.org/simple/citrix/citrix.</p>

</body>
</html>
```
Citrix ADC ingress controller

Example 2:

```yaml
curl http://app.cic-citrix.org/something
```

Or,

```yaml
curl http://app.cic-citrix.org/something/
```

Output:

```
GET / HTTP/1.1
Host: app.cic-citrix.org


[Full request URL: http://app.cic-citrix.org/]
[HTTP request 1/1]
[Response in frame: 32]
```

Add the X-Forwarded-For header within an HTTP request

You can define a rewrite policy as shown in the following example YAML (http_x_forwarded_for_insert.yaml) to add the X-Forwarded-For header within an HTTP request:

```yaml
apiVersion: citrix.com/v1
type: rewritepolicy
metadata:
  name: httpxforwardedforaddition
spec:
  rewrite-policies:
    - servicenames:
        - frontend
          rewrite-policy:
            operation: insert_http_header
            target: X-Forwarded-For
            modify-expression: client.ip.src
            comment: 'HTTP Initial X-Forwarded-For header add'
            direction: REQUEST
            rewrite-criteria: 'HTTP.REQ.HEADER("X-Forwarded-For").EXISTS. NOT'
```
Create a YAML file (`http_x_forwarded_for_insert.yaml`) with the rewrite policy definition and deploy the YAML file using the following command:

```bash
kubectl create -f http_x_forwarded_for_insert.yaml
```

Using the `curl` command you can verify the HTTP packet with and without the `X-Forwarded-For` header.

Example: Output of the HTTP request packet without `X-Forwarded-For` header:

```
curl http://app.cic-citrix.org/
```

Output:

```
HTTP/1.1 200 OK
Date: Thu, 03 Nov 2022 16:39:05 GMT
Content-Type: text/plain;charset=utf-8

```

Example: Output of the HTTP request packet with `X-Forwarded-For` header:

```
curl curl --header "X-Forwarded-For: 1.1.1.1" http://app.cic-citrix.org/
```

Output:
Citrix ADC ingress controller

Redirect HTTP request to HTTPS request using responder policy

You can define a responder policy definition as shown in the following example YAML (http_to_https_redirect.yaml) to redirect HTTP requests to HTTPS request:

```yaml
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
  name: httptohttps
spec:
  responder-policies:
    - servicenames:
      - frontend
    responder-policy:
      redirect:
        url: 'https://' +http.req.HOSTNAME.SERVER+"":"+"443"+http.req.url'
        respond-criteria: 'http.req.is_valid'
        comment: 'http to https'
```

Create a YAML file (http_to_https_redirect.yaml) with the responder policy definition and deploy the YAML file using the following command:

```
kubectl create -f http_to_https_redirect.yaml
```

You can verify if the HTTP request is redirected to HTTPS as follows:

Example 1:

```
$ curl -v http://app.cic-citrix.org
```
Modify strings and host name in the requested URL

This example shows the usage of the `goto-priority-expression` attribute. The guidelines for usage of the `goto-priority-expression` field can be found at [How to write a policy configuration.](#)
This example modifies the URL http://www.citrite.org/something/simple/citrix to http://app.cic-citrix.org/simple/citrix.

Two rewrite policies are written to modify the URL:

- **Rewrite policy 1:** This policy is used to modify the host name www.citrite.org to app.cic-citrix.org.
- **Rewrite Policy 2:** This policy is used to modify the url /something/simple/citrix to /simple/citrix

You can bind the two policies using the `goto-priority-expression` attribute as shown in the following YAML:

```yaml
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
  name: hostnameurlrewrite
spec:
  rewrite-policies:
    - servicenames:
      - citrix-svc
goto-priority-expression: NEXT
      rewrite-policy:
        operation: replace_all
        target: 'http.req.header("host")'
        modify-expression: '"app.cic-citrix.org"'
        multiple-occurence-modify: 'text("www.citrite.org")'
        comment: 'HTTP header rewrite of hostname'
        direction: REQUEST
        rewrite-criteria: 'http.req.is_valid.and(HTTP.REQ.HOSTNAME.EQ("www.citrite.org"))'
    - servicenames:
      - citrix-svc
goto-priority-expression: END
      rewrite-policy:
        operation: replace_all
        target: http.req.url
        modify-expression: '""'
        multiple-occurence-modify: 'regex(re-((^\//something\/)*)((^\/something$))~)'
        comment: 'HTTP url replace string'
        direction: REQUEST
        rewrite-criteria: 'http.req.is_valid.and(HTTP.REQ.HOSTNAME.EQ("www.citrite.org"))'
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
Citrix ADC ingress controller

Verification

You can verify whether the following curl request http://www.citruste.org/something/simple/citrix is modified to http://app.cic-citrix.org/simple/citrix.

Example: Modifying the requested URL

```bash
curl http://www.citrite.org/something/simple/citrix
```

Modified host name and URL for the requested URL is present in the image shown as follows:

HTTP callout

An HTTP callout allows the Citrix ADC to generate and send an HTTP or HTTPS request to an external server as part of the policy evaluation and take appropriate action based on the response obtained from the external server. You can use the rewrite and responder CRD to initiate HTTP callout requests from the Citrix ADC. For more information, see the HTTP callout documentation.

Related articles

- Feature Documentation
  - Citrix ADC Rewrite Feature Documentation
  - Citrix ADC Responder Feature Documentation
- Developer Documentation
  - Citrix ADC Rewrite Policy
  - Citrix ADC Rewrite Action
  - Citrix ADC Responder Policy

© 1999–2022 Citrix Systems, Inc. All rights reserved.
Citrix ADC ingress controller

- Citrix ADC Responder Action
- Citrix ADC Audit Message Action
- Citrix ADC Policy Dataset

VIP CustomResourceDefinitions

February 3, 2022

Citrix provides a CustomResourceDefinitions (CRD) called VIP for asynchronous communication between the IPAM controller and Citrix ingress controller.

The IPAM controller is provided by Citrix for IP address management. It allocates IP address to the service from a defined IP address range. The Citrix ingress controller configures the IP address allocated to the service as virtual IP (VIP) in Citrix ADX VPX. And, the service is exposed using the IP address.

When a new service is created, the Citrix ingress controller creates a CRD object for the service with an empty IP address field. The IPAM Controller listens to addition, deletion, or modification of the CRD and updates it with an IP address to the CRD. Once the CRD object is updated, the Citrix ingress controller automatically configures Citrix ADC-specific configuration in the tier-1 Citrix ADC VPX.

Note:
The VIP CRD is not supported for OpenShift routes. You can use OpenShift ingress to use the VIP CRD.

Deploy the VIP CRD

Deploy the VIP CRD using the following command:

```sh
kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/crd/vip/vip.yaml
```

Related Articles

- Expose services of type LoadBalancer using an IP address from the Citrix IPAM controller
Advanced content routing for Kubernetes with Citrix ADC

February 3, 2022

Kubernetes native Ingress offers basic host and path based routing. But, other advanced routing techniques like routing based on header values or query strings is not supported in the Ingress structure. You can expose these features on the Kubernetes Ingress through Ingress annotations, but annotations are complex to manage and validate.

You can expose the advanced content routing abilities provided by Citrix ADC as a custom resource definition (CRD) API for Kubernetes.

Using content routing CRDs, you can route the traffic based on the following parameters:

- Hostname
- URL path
- HTTP headers
- Cookie
- Query parameters
- HTTP method
- Citrix ADC policy expression

**Note:**

An Ingress resource and content routing CRDs cannot co-exist for the same endpoint (IP address and port). The usage of content routing CRDs with Ingress is not supported.

The advanced content routing feature is exposed in Kubernetes with the following CRDs:

- Listener
- HTTPRoute
Listener CRD

A Listener CRD object represents the end-point information like virtual IP address, port, certificates, and other front-end configurations. It also defines the default actions like sending the default traffic to a back end or redirecting the traffic. A Listener CRD object can refer to HTTPRoute CRD objects which represents HTTP routing logic for the incoming HTTP request.

For the full CRD definition, see the Listener CRD.
For complete information on all attributes of the Listener CRD, see Listener CRD documentation.

Listener CRD supports HTTP, SSL, and TCP profiles. Using these profiles, you can customize the default protocol behavior. Listener CRD also supports the analytics profile which enables Citrix ADC to export the type of transactions or data to different endpoints.
For more information about profile support for Listener CRD, see the Profile support for the Listener CRD.
Citrix ADC ingress controller

**Deploy the Listener CRD**

1. Download the Listener CRD.

2. Deploy the listener CRD with following command.

```
Kubectl create -f Listener.yaml
```

Example:

```
root@k8smaster:~# kubectl create -f Listener.yaml
customresourcedefinition.apiextensions.k8s.io/listeners.citrix.com created
```

**How to write Listener CRD objects**

After you have deployed the CRD provided by Citrix in the Kubernetes cluster, you can define the listener configuration in a YAML file. In the YAML file, use `Listener` in the kind field and in the spec section add the listener CRD attributes based on your requirement for the listener configuration.

After you deploy the YAML file, the Citrix ingress controller applies the listener configuration on the Ingress Citrix ADC device.

Following is a sample Listener CRD object definition named as `Listener-crd.yaml`.

```
apiVersion: citrix.com/v1
kind: Listener
metadata:
  name: my-listener
  namespace: default
spec:
certificates:
  - secret:
      name: my-secret
      # Secret named 'my-secret' in current namespace bound as default certificate
    default: true
  - secret:
      # Secret 'other-secret' in demo namespace bound as SNI certificate
    name: other-secret
```
namespace: demo
- preconfigured: second-secret
  # preconfigured certkey name in ADC
vip: '192.168.0.1' # Virtual IP address to be used, not required when CPX is used as ingress device
port: 443
protocol: https
redirectPort: 80
secondaryVips:
- "10.0.0.1"
- "1.1.1.1"
policies:
  httpprofile:
    config:
      websocket: "ENABLED"
tcpprofile:
    config:
      sack: "ENABLED"
sslprofile:
    config:
      ssl3: "ENABLED"
sslciphers:
  - SECURE
  - MEDIUM
analyticsprofile:
  config:
    - type: webinsight
      parameters:
        allhttpheaders: "ENABLED"
csvserverConfig:
  rhistate: 'ACTIVE'
routes:
  # Attach the policies from the below Routes
  - name: domain1-route
    namespace: default
  - name: domain2-route
    namespace: default
  - labelSelector:
    # Attach all HTTPRoutes with label route=my-route
    route: my-route
  # Default action when traffic matches none of the policies in the HTTPRoute
defaultAction:
  backend:
  kube:
In this example, a listener is exposing an HTTPS endpoint. Under certificates section, SSL certificates for the endpoint are configured using Kubernetes secrets named `my-secret` and `other-secret` and a default ADC preconfigured certificate with certkey named `second-secret`. The default action for the listener is configured as a Kubernetes service. Routes are attached with the listener using both label selectors and individual route references using name and namespace.

After you have defined the Listener CRD object in the YAML file, deploy the YAML file using the following command. In this example, `Listener-crd.yaml` is the YAML definition.

```
Kubectl create -f Listener-crd.yaml
```

**HTTPRoute CRD**

An HTTPRoute CRD object represents the HTTP routing logic for the incoming HTTP requests. You can use a combination of various HTTP parameters like host name, path, headers, query parameters, and cookies to route the incoming traffic to a back-end service. An HTTPRoute object can be attached to one or more Listener objects which represent the end point information. You can have one or more rules in an HTTPRoute object, with each rule specifying an action associated with it. Order of evaluation of the rules within an HTTPRoute object is same as the order mentioned in the object. For example, if there are two rules with the order rule1 and rule2, with rule1 is written before rule2, rule1 is evaluated first before rule2.

HTTPRoute CRD definition is available at `HTTPRoute.yaml`. For complete information on the attributes for HTTP Route CRD, see [HTTPRoute CRD documentation](https://httproute.docs.citrix.com/). Now, Citrix supports configuring the HTTP route CRD resource as a resource backend in the Ingress with Kubernetes Ingress version `networking.k8s.io/v1`. With this feature, you can extend advanced content routing capabilities to Ingress. For more information, see [Advanced content routing for Kubernetes Ingress using HTTPRoute CRD](https://httproute.docs.citrix.com/).
Deploy the HTTPRoute CRD

Perform the following to deploy the HTTPRoute CRD:

1. Download the HTTPRoute.yaml.
2. Apply the HTTPRoute CRD in your cluster using the following command.

   Kubectl apply -f HTTPRoute.yaml

   Example:

   ```
   root@k8smaster:~ # kubectl create -f HTTPRoute.yaml
   customresourcedefinition.apiextensions.k8s.io/httproutes.citrix.com configured
   ```

How to write HTTPRoute CRD objects

Once you have deployed the HTTPRoute CRD, you can define the HTTP route configuration in a YAML file. In the YAML file, use HTTPRoute in the kind field and in the spec section add the HTTPRoute CRD attributes based on your requirement for the HTTP route configuration.

Following is a sample HTTPRoute CRD object definition named as Route-crd.yaml.

```yaml
apiVersion: citrix.com/v1
customresourcedefinition.apiextensions.k8s.io/httproutes.citrix.com
kind: HTTPRoute
metadata:
  name: test-route
spec:
  hostname:
    - host1.com
  rules:
    - name: header-routing
      match:
        headers:
          - headerName:
              exact: my-header
      action:
        backend:
          kube:
            service: mobile-app
            port: 80
          backendConfig:
            secureBackend: true
```
In this example, any request with a header name matching `my-header` is routed to the mobile-app service and all other traffic is routed to the default-app service.

For detailed explanations and API specifications of HTTPRoute, see [HTTPRoute CRD](#).

After you have defined the HTTP routes in the YAML file, deploy the YAML file for HTTPRoute CRD object using the following command. In this example, `Route-crd.yaml` is the YAML definition.

```bash
kubectl create -f Route-crd.yaml
```

Once you deploy the YAML file, the Citrix ingress controller applies the HTTP route configuration on the Ingress Citrix ADC device.

### Attaching HTTPRoute CRD objects to a Listener CRD object

You can attach HTTPRoute CRD objects to a Listener CRD object in two ways:

- Using name and namespace
- Using labels and selector

### Attaching HTTPRoute CRD objects using name and namespace

In this approach, a Listener CRD object explicitly refer to one or more HTTPRoute objects by specifying the name and namespace in the `routes` section.

The order of evaluation of HTTPRoute objects is same as the order specified in the Listener CRD object with the first HTTPRoute object is evaluated first and so on.

For example, a snippet of the Listener CRD object is shown as follows.
In this example, the HTTPRoute CRD object named route1 is evaluated before the HTTPRoute named route2.

### Attaching an HTTPRoute CRD object using labels and selector

You can also attach HTTPRoute objects to a Listener object by using labels and selector. You can specify one or more labels in the Listener CRD object. Any HTTPRoute objects which match the labels are automatically linked to the Listener object and the rules are created in Citrix ADC. When you use this approach, there is no particular order of evaluation between multiple HTTPRoute objects. Only exception is an HTTPRoute object with a default route (a route with just a host name or a ‘/’ path) which is evaluated as the last object.

For example, snippet of a listener resource is as follows:

```yaml
routes:
- name: route-1
  namespace: default
- name: route-2
  namespace: default
```

### Configure web application firewall policies with the Citrix ingress controller

January 26, 2022

Citrix provides a Custom Resource Definition (CRD) called the WAF CRD for Kubernetes. You can use the WAF CRD to configure the web application firewall policies with the Citrix ingress controller on the Citrix ADC VPX, MPX, SDX, and CPX. The WAF CRD enables communication between the Citrix ingress controller and Citrix ADC for enforcing web application firewall policies.

In a Kubernetes deployment, you can enforce a web application firewall policy to protect the server using the WAF CRD. For more information about web application firewall, see [Web application security](#).

With the WAF CRD, you can configure the firewall security policy to enforce the following types of security checks for Kubernetes native applications.
Common protections

- Buffer overflow
- Content type
- Allow URL
- Block URL
- Cookie consistency
- Credit card

HTML protections

- CSRF (cross side request forgery) form tagging
- Field formats
- Form field consistency
- File upload types
- HTML cross-site scripting
- HTML SQL injection

JSON protections

- JSON denial of service
- JSON SQL injection
- JSON cross-site scripting

XML protections

- XML web services interoperability
- XML attachment
- XML cross-site scripting
- XML denial of service
- XML format
- XML message validation
- XML SOAP fault filtering
- XML SQL injection

Based on the type of security checks, you can specify the metadata and use the CRD attributes in the WAF CRD .yaml file to define the WAF policy.
Citrix ADC ingress controller

WAF CRD definition

The WAF CRD is available in the Citrix ingress controller GitHub repository at waf-crd.yaml. The WAF CRD provides attributes for the various options that are required to define the web application firewall policies on Citrix ADC.

WAF CRD attributes

The following table lists the various attributes provided in the WAF CRD:

<table>
<thead>
<tr>
<th>CRD attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>commonchecks</td>
<td>Specifies a list of common security checks, which are applied irrespective of the content type.</td>
</tr>
<tr>
<td>block_urls</td>
<td>Protects URLs.</td>
</tr>
<tr>
<td>buffer_overflow</td>
<td>Protects buffer overflow.</td>
</tr>
<tr>
<td>content_type</td>
<td>Protects content type.</td>
</tr>
<tr>
<td>htmlchecks</td>
<td>Specifies a list of security checks to be applied for HTML content types.</td>
</tr>
<tr>
<td>cross_site_scripting</td>
<td>Prevents cross site scripting attacks.</td>
</tr>
<tr>
<td>sql_injection</td>
<td>Prevents SQL injection attacks.</td>
</tr>
<tr>
<td>form_field_consistency</td>
<td>Prevents form tampering.</td>
</tr>
<tr>
<td>csrf</td>
<td>Prevents cross side request forgery (CSRF) attacks.</td>
</tr>
<tr>
<td>cookie_consistency</td>
<td>Prevents cookie tampering or session takeover.</td>
</tr>
<tr>
<td>field_format</td>
<td>Validates the form submission.</td>
</tr>
<tr>
<td>fileupload_type</td>
<td>Prevents malicious file uploads.</td>
</tr>
<tr>
<td>jsonchecks</td>
<td>Specifies security checks for JSON content types.</td>
</tr>
<tr>
<td>xmlchecks</td>
<td>Specifies security checks for XML content types.</td>
</tr>
<tr>
<td>wsi</td>
<td>Protects web services interoperability.</td>
</tr>
<tr>
<td>redirect_url</td>
<td>Redirects URL when block is enabled on protection.</td>
</tr>
</tbody>
</table>
## CRD attribute

<table>
<thead>
<tr>
<th>CRD attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servicenames</td>
<td>Specifies the services to which the WAF policies are applied.</td>
</tr>
<tr>
<td>application_type</td>
<td>Protects application types.</td>
</tr>
<tr>
<td>signatures</td>
<td>Specifies the location of the external signature file.</td>
</tr>
<tr>
<td>html_error_object</td>
<td>Specifies the location of the customized error page to respond when HTML or common violations are attempted.</td>
</tr>
<tr>
<td>xml_error_object</td>
<td>Specifies the location of the customized error page to respond when XML violations are attempted.</td>
</tr>
<tr>
<td>json_error_object</td>
<td>Specifies the location of the customized error page to respond when JSON violations are attempted.</td>
</tr>
<tr>
<td>ip_reputation</td>
<td>Enables the IP reputation feature.</td>
</tr>
<tr>
<td>target</td>
<td>Determines the traffic to be inspected by the WAF. If you do not specify the traffic targeted, all traffic is inspected by default.</td>
</tr>
<tr>
<td>paths</td>
<td>Specifies the list of HTTP URLs to be inspected.</td>
</tr>
<tr>
<td>method</td>
<td>Specifies the list of HTTP methods to be inspected.</td>
</tr>
<tr>
<td>header</td>
<td>Specifies the list of HTTP headers to be inspected.</td>
</tr>
</tbody>
</table>

## Deploy WAF CRD

Perform the following steps to deploy the WAF CRD:

1. Download the CRD (`waf-crd.yaml`).
2. Deploy the WAF CRD using the following command:

   ```shell
   kubectl create -f waf-crd.yaml
   ```

   For example,
Citrix ADC ingress controller

```
1 root@master:~# kubectl create -f waf-crd.yaml
2 customresourcedefinition.apiextensions.k8s.io/wafpolicies.citrix.com created
3 <!--NeedCopy--> 
```

How to write a WAF configuration

After you have deployed the WAF CRD provided by Citrix in the Kubernetes cluster, you can define the web application firewall policy configuration in a .yaml file. In the .yaml file, use waf in the kind field. In the spec section add the WAF CRD attributes based on your requirements for the policy configuration.

After you deploy the .yaml file, the Citrix ingress controller applies the WAF configuration on the Ingress Citrix ADC device.

The following are some examples for writing web application firewall policies.

Enable protection for cross-site scripting and SQL injection attacks

Consider a scenario in which you want to define and specify a web application firewall policy in the Citrix ADC to enable protection for the cross-site scripting and SQL injection attacks. You can create a .yaml file called `wafhtmlxsssql.yaml` and use the appropriate CRD attributes to define the WAF policy as follows:

```
1 apiVersion: citrix.com/v1
2 kind: waf
3 metadata:
4   name: wafhtmlxsssql
5 spec:
6   servicenames:
7     - frontend
8   application_type: HTML
9   html_page_url: "http://x.x.x.x/crd/error_page.html"
10  security_checks:
11     html:
12       cross_site_scripting: "on"
13       sql_injection: "on"
14 <!--NeedCopy--> 
```
Apply rules to allow only known content types

Consider a scenario in which you want to define a web application firewall policy that specifies rules to allow only known content types and block unknown content types. Create a .yaml file called `waf-contenttype.yaml` and use the appropriate CRD attributes to define the WAF policy as follows:

```yaml
apiVersion: citrix.com/v1
kind: waf
metadata:
  name: wafcontenttype
spec:
servicenames:
  - frontend
  application_type: HTML
  html_error_object: "http://x.x.x.x/crd/error_page.html"
security_checks:
  common:
    content_type: "on"
relaxations:
  common:
    content_type:
types:
  - custom_cnt_type
  - image/crd

<!--NeedCopy-->
```

Protect against known attacks

The following is an example of a WAF CRD configuration for applying external signatures. You can copy the latest WAF signatures from Signature Location to the local web server and provide the location of the copied file as `signature_url`.

```yaml
apiVersion: citrix.com/v1
kind: waf
metadata:
  name: wafhtmlsigxsssql
spec:
servicenames:
  - frontend
  application_type: HTML
```
The following is an example of a WAF CRD configuration for protecting buffer overflow.

```yaml
apiVersion: citrix.com/v1
dependent: waf
metadata:
  name: wafhdrbufferoverflow
spec:
servicenames:
  - frontend
  application_type: HTML
  html_error_object: "http://x.x.x.x/crd/error_page.html"
security_checks:
  common:
    buffer_overflow: "on"
    multiple_headers:
      action: ["block", "log"]
  settings:
    common:
      buffer_overflow:
        max_cookie_len: 409
        max_header_len: 4096
        max_url_len: 1024
<!--NeedCopy-->
```

**Protect from header buffer overflow attacks and block multiple headers**

The following is an example of a WAF CRD configuration for protecting buffer overflow.

**Prevent repeated attempts to access random URLs on a web site**

The following is an example of a WAF CRD configuration for providing URL filter rules. You can add URLs to permit under `allow_url` and URLs to deny under `block_url`. The URL can be a regular expression also.
Prevent leakage of sensitive data

Data breaches involve leakage of sensitive data such as credit card and social security number (SSN). You can add custom regexes for the sensitive data in the Enforcements safe objects section.

The following is an example of a WAF CRD configuration for preventing leakage of sensitive data.
Citrix ADC ingress controller

```yaml
name: wafdataleak
spec:
servicenames:
  - frontend
application_type: HTML
html_error_object: "http://x.x.x.x/crd/error_page.html"
security_checks:
  common:
    credit_card: "on"
settings:
  common:
    credit_card:
      card_type: ["visa","amex"]
    max_allowed: 1
    card_xout: "on"
    secureLogging: "on"
enforcements:
  common:
    safe_object:
      - rule:
        name: aadhar
        expression: "[1-9]{4,4}\s[1-9]{4,4}\s[1-9]{4,4}"
        max_match_len: 19
        action: ["log","block"]
<!-- NeedCopy -->
```

### Protect HTML forms from CSRF and form attacks

The following is an example of a WAF CRD configuration for protecting HTML forms from CSRF and form attacks.

```yaml
apiVersion: citrix.com/v1
kind: waf
metadata:
  name: wafforms
spec:
```
Protect forms and headers

The following is an example of a WAF CRD configuration for protecting both forms and headers.

```yaml
apiVersion: citrix.com/v1
kind: waf
metadata:
  name: wafhdrforms
spec:
  servicenames:
    - frontend
  application_type: HTML
  html_page_url: "http://x.x.x.x/crd/error_page.html"
  security_checks:
    html:
      cross_site_scripting: "on"
      sql_injection: "on"
      form_field_consistency:
        action: ["log","block"]
      csrf: "on"
    common:
      buffer_overflow: "on"
      multiple_headers:
        action: ["block","log"]
    html:
      cross_site_scripting: "on"
      sql_injection: "on"
      form_field_consistency:
        action: ["log","block"]
      csrf: "on"
  settings:
    common:
      buffer_overflow:
        max_cookie_len: 409
        max_header_len: 4096
```
Enable basic WAF security checks

The basic security checks are required to protect any application with minimal effect on performance. It does not require any sessionization. The following is an example of a WAF CRD configuration for enabling basic WAF security checks.

```yaml
apiVersion: citrix.com/v1
kind: waf
metadata:
  name: wafbasic
spec:
  servicenames:
    - frontend
  security_checks:
    common:
      allow_url: "on"
      block_url: "on"
      buffer_overflow: "on"
      multiple_headers:
        action: ["block", "log"]
      html:
        cross_site_scripting: "on"
        field_format: "on"
        sql_injection: "on"
        fileupload_type: "on"
    json:
      dos: "on"
      sql_injection: "on"
      cross_site_scripting: "on"
    xml:
      dos: "on"
      wsi: "on"
      attachment: "on"
      format: "on"
    relaxations:
      common:
```
Enable advanced WAF security check

Advanced security checks such as cookie consistency, allow URL closure, field consistency, and CSRF are resource-intensive (CPU and memory) as they require WAF sessionization. For example, when a form is protected by the WAF, form field information in the response is retained in the system memory. When the client submits the form in the next request, it is checked for inconsistencies before the information is sent to the web server. This process is known as sessionization. The following is an example of a WAF CRD configuration for enabling WAF advanced security checks.

```yaml
apiVersion: citrix.com/v1
kind: waf
metadata:
  name: wafadvanced
spec:
  servicenames:
    - frontend
  security_checks:
    common:
      allow_url: "on"
      block_url: "on"
      buffer_overflow: "on"
      content_type: "on"
      cookie_consistency: "on"
      multiple_headers: 
        action: ["log"]
      html:
        cross_site_scripting: "on"
        field_format: "on"
        sql_injection: "on"
        form_field_consistency: "on"
        csrf: "on"
        fileupload_type: "on"
      json:
      dos: "on"
```
Enable IP reputation

The following is an example of a WAF CRD configuration for enabling IP reputation to reject requests that come from IP addresses with bad reputation.

```yaml
apiVersion: citrix.com/v1
kind: waf
metadata:
  name: wafiprep
spec:
  application_type: html
  servicenames:
    - frontend
  ip_reputation: "on"
```

Enable IP reputation to reject requests of a particular category

The following is an example of a WAF CRD configuration for enabling IP reputation to reject requests from particular threat categories.

```yaml
apiVersion: citrix.com/v1
kind: waf
metadata:
  name: wafiprepcategory
```
Protect JSON applications from denial of service attacks

The following is an example of a WAF CRD configuration for protecting the JSON applications from denial of service attacks.

```yaml
metadata:
  name: wafjsondos
spec:
  servicenames:
    - frontend
  application_type: JSON
  json_error_object: "http://x.x.x.x/crd/error_page.json"
  security_checks:
    json:
      dos: "on"
  settings:
    json:
      dos:
        container:
          max_depth: 2
```

---

© 1999–2022 Citrix Systems, Inc. All rights reserved.
Protect RESTful APIs

The following is an example of a WAF CRD configuration for protecting RESTful APIs from SQL injection, cross-site scripting, and denial of service attacks.

Here, the back-end application or service is purely based on RESTful APIs.

```yaml
apiVersion: citrix.com/v1
kind: waf
metadata:
  name: wafjson
spec:
  servicenames:
    - frontend
  application_type: JSON
  json_error_object: "http://x.x.x.x/crd/error_page.json"
  security_checks:
    json:
      dos: "on"
      sql_injection:
        action: ["block"]
      cross_site_scripting: "on"
  settings:
    json:
      dos:
        container:
          max_depth: 5
      document:
        max_len: 20000000
```
Protect XML applications from denial of service attacks

The following is an example of a WAF CRD configuration for protecting the XML applications from denial of service attacks.

```yaml
apiVersion: citrix.com/v1
kind: waf
metadata:
  name: wafxmlldos
spec:
  servicenames:
    - frontend
  application_type: XML
  xml_error_object: "http://x.x.x/crd/error_page.xml"
  security_checks:
    xml:
      dos: "on"
    settings:
      xml:
        dos:
          attribute:
            max_attributes: 1024
            max_name_len: 128
            max_value_len: 128
          element:
            max_elements: 1024
            max_children: 128
            max_depth: 128
          file:
            max_size: 2123
            min_size: 9
          entity:
            max_expansions: 512
```
Produce XML applications from security attacks

This example provides a WAF CRD configuration for protecting XML applications from the following security attacks:

- SQL injection
- Cross-site scripting
- Validation (schema or message)
- Format
- Denial of service
- Web service interoperability (WSI)

```yaml
apiVersion: citrix.com/v1
kind: waf
metadata:
  name: wafxml
spec:
  servicenames:
    - frontend
  application_type: XML
  xml_error_object: "http://x.x.x.x/crd/error_page.json"
  security_checks:
    xml:
      dos: "on"
      sql_injection: "on"
      cross_site_scripting: "off"
  wsi:
    action: ["block"]
    validation: "on"
    attachment: "on"
  format:
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
Configure bot management policies with the Citrix ingress controller

January 26, 2022
A bot is a software application that automates manual tasks. Using Bot management policies you can allow useful bots to access your cloud native environment and block the malicious bots.

Custom Resource Definitions (CRDs) are the primary way of configuring policies in cloud native deployments. Using the Bot CRD provided by Citrix, you can configure the bot management policies with the Citrix ingress controller on the Citrix ADC VPX. The Bot CRD enables communication between the Citrix ingress controller and Citrix ADC for enforcing bot management policies.

In a Kubernetes deployment, you can enforce bot management policy on the requests and responses from and to the server using the Bot CRD. For more information on security vulnerabilities, see Bot Detection.

With the Bot CRD, you can configure the bot management security policy for the following types of security vulnerabilities for the Kubernetes-native applications:

- Allow list
- Block list
- Device Fingerprint (DFP)
- Bot TPS
- Trap insertion
- IP reputation
- Rate limit

Based on the type of protections required, you can specify the metadata and use the CRD attributes in the Bot CRD .yam l file to define the bot policy.

**Bot CRD definition**

The Bot CRD is available in the Citrix ingress controller GitHub repo at bot-crd.yaml. The Bot CRD provides attributes for the various options that are required to define the bot management policies on Citrix ADC.

**Bot CRD attributes**

The following table lists the various attributes provided in the Bot CRD:

<table>
<thead>
<tr>
<th>CRD attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>security_checks</td>
<td>List of security checks to be applied for incoming traffic.</td>
</tr>
<tr>
<td>allow_list</td>
<td>List of allowed IP, subnet, and policy expressions.</td>
</tr>
</tbody>
</table>

© 1999–2022 Citrix Systems, Inc. All rights reserved.
## Deploy the Bot CRD

Perform the following steps to deploy the Bot CRD:

1. Download the `bot-crd.yaml`
2. Deploy the Bot CRD using the following command:

```
kubectl create -f bot-crd.yaml
```

For example,

```
root@master:~# kubectl create -f bot-crd.yaml
```
How to write a Bot configuration

After you have deployed the Bot CRD provided by Citrix in the Kubernetes cluster, you can define the bot management policy configuration in a YAML file. In the YAML file, specify bot in the kind field. In the spec section, add the Bot CRD attributes based on your requirements for the policy configuration.

After you deploy the YAML file, the Citrix ingress controller applies the bot configuration on the Ingress Citrix ADC device.

Following are some examples for bot policy configurations:

Block malicious traffic using known IP, subnet, or ADC policy expressions

When you want to define and employ a web bot management policy in Citrix ADC to enable bot for blocking malicious traffic, you can create a YAML file called botblocklist.yml and use the appropriate CRD attributes to define the bot policy as follows:

```yaml
apiVersion: citrix.com/v1
kind: bot
metadata:
  name: botblocklist
spec:
  servicenames:
    - frontend
  security_checks:
    block_list: "ON"
  bindings:
    block_list:
      - subnet:
          value:
            - 172.16.1.0/12
            - 172.16.2.0/12
            - 172.16.3.0/12
            - 172.16.4.0/12
          action:
            - "drop"
        ip:
          value: 10.102.30.40
        expression:
```
Allow known traffic without bot security checks**

When you want to avoid security checks for certain traffic such as staging or trusted traffic, you can avoid such traffic from security checks. You can create a YAML file called `botallowlist.yaml` and use the appropriate CRD attributes to define the bot policy as follows:

```yaml
apiVersion: citrix.com/v1
kind: bot
metadata:
  name: botallowlist
spec:
  servicenames:
  - frontend
  security_checks:
    allow_list: "ON"
  bindings:
    allow_list:
      - subnet:
        value:
          - 172.16.1.0/12
          - 172.16.2.0/12
          - 172.16.3.0/12
          - 172.16.4.0/12
        action:
          - "log"
      - ip:
        value: 10.102.30.40
        expression:
        value: http.req.url.contains("/index.html")
        action:
          - "log"
<!--NeedCopy-->
Enable bot signatures to detect bots

Citrix provides thousands of inbuilt signatures to detect bots based on user agents. Citrix threat intelligence team keeps on updating and releasing new bot signatures in every two weeks. The latest bot signature file is available at: Bot signatures. You can create a YAML file called *botsignatures.yaml* and use the appropriate CRD attributes to define the bot policy as follows:

```yaml
apiVersion: citrix.com/v1
kind: bot
metadata:
  name: botsignatures
spec:
  servicenames:
    - frontend
  redirect_url: "/error_page.html"
signatures: "http://10.106.102.242/ganeshka/bot_sig.json"
<!--NeedCopy-->
```

Enable the bot device fingerprint and customize the action

Device fingerprinting involves inserting a JavaScript snippet in the HTML response to the client. This JavaScript snippet, when invoked by the browser on the client, collects the attributes of the browser and client. And sends a POST request to Citrix ADC with that information. These attributes are examined to determine whether the connection is requested from a bot or a human being. You can create a YAML file called *botdfp.yaml* and use the appropriate CRD attributes to define the bot policy as follows:

```yaml
apiVersion: citrix.com/v1
kind: bot
metadata:
  name: botdfp
spec:
  servicenames:
    - frontend
  redirect_url: "/error_page.html"
security_checks:
  device_fingerprint:
    action:
    - "log"
    - "drop"
<!--NeedCopy-->
```
Enable the bot TPS and customize the action

If the bot TPS is configured, it detects incoming traffic as bots if the maximum number of requests or increase in requests exceeds the configured time interval. You can configure the TPS limits as per geolocation, host, source IP, and URL in the bindings section. You can create a YAML file called `bottps.yaml` and use the appropriate CRD attributes to define the bot policy as follows:

```yaml
apiVersion: citrix.com/v1
kind: bot
metadata:
  name: bottps
spec:
servicenames:
  - frontend
redirect_url: "/error_page.html"
security_checks:
  tps: "ON"
bindings:
  tps:
    geolocation:
      threshold: 101
      percentage: 100
    host:
      threshold: 10
      percentage: 100
    action:
      - "log"
      - "mitigation"
<!--NeedCopy-->
```

Enable the trap insertion protection and customize the action

Detects and blocks automated bots by advertising a trap URL in the client response. The URL is invisible and not accessible to the client, if it is human. The detection method is effective in blocking attacks from automated bots. Insertion of the trap URL in the URL responses is random. You can enforce the trap URL insertion to a particular URL response by configuring the trap bindings. You can create a YAML file called `trapinsertion.yaml` and use the appropriate CRD attributes to define the bot policy as follows:
---

Enable IP reputation to reject requests of a particular category

The following is an example of a Bot CRD configuration for enabling only specific threat categories of IP reputation that are suitable for the user environment. You can create a YAML file called `- NeedCopy-->

```yaml
apiVersion: citrix.com/v1
kind: bot
metadata:
  name: trapinsertion
spec:
  servicenames:
    - frontend
  redirect_url: "/error_page.html"
  security_checks:
    trap:
      action:
      - "log"
      - "drop"
    bindings:
      trapinsertion:
        urls:
          - "/index.html"
          - "/submit.php"
          - "/login.html"
<!--NeedCopy-->
**Citrix ADC ingress controller**

```yaml
13   categories:
14       - SPAM_SOURCES:
15           action:
16               - "log"
17               - "redirect"
18       - MOBILE_THREATS
19       - SPAM_SOURCES
20 !--NeedCopy-->
```

**Enable rate limit to control request rate**

The following is an example of a Bot CRD configuration for enforcing the request rate limit using the parameters: URL, cookies, and IP. You can create a YAML file called `botratelimit.yaml` and use the appropriate CRD attributes to define the bot policy as follows:

```yaml
apiVersion: citrix.com/v1
kind: bot
metadata:
  name: botratelimit
spec:
  servicenames:
    - frontend
  redirect_url: "/error_page.html"
  security_checks:
    ratelimit: "ON"
  bindings:
    ratelimit:
      - url:
            value: index.html
            rate: 2000
            timeslice: 1000
      - cookie:
            value: citrix_bot_id
            rate: 2000
            timeslice: 1000
      - ip:
            rate: 2000
            timeslice: 1000
            action:
                - "log"
                - "reset"
      !--NeedCopy-->
```
Configure cross-origin resource sharing policies with Citrix ingress controller

January 19, 2022

Citrix provides a Custom Resource Definition (CRD) called the CORS CRD for Kubernetes. You can use the CORS CRD to configure the cross-origin resource sharing (CORS) policies with Citrix ingress controller on the Citrix ADC.

What is CORS

Cross-Origin resource sharing is a mechanism that allows the browser to determine whether a specific web application can share resources with another web application from a different origin. It allows users request resources (For example, images, fonts, and videos) from domains outside the original domain.

CORS pre-flight

Before a web browser allowing Javascript to issue a POST to a URL, it performs a pre-flight request. A pre-flight request is a simple request to the server with the same URL using the method OPTIONS rather than POST. The web browser checks the HTTP headers for CORS related headers to determine if POST operation on behalf of the user is allowed.
**CORS CRD definition**

The CORS CRD is available in the Citrix ingress controller GitHub repo at: cors-crd.yaml. The CORS CRD provides attributes for the various options that are required to define the CORS policy on the Ingress Citrix ADC that acts as an API gateway. The required attributes include: servicenames, allow_origin, allow_methods, and allow_headers.

The following are the attributes provided in the CORS CRD:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servicenames</td>
<td>Specifies the list of Kubernetes services to which you want to apply the CORS policies.</td>
</tr>
<tr>
<td>allow_origin</td>
<td>Specifies the list of allowed origins. Incoming origin is screened against this list.</td>
</tr>
<tr>
<td>allow_methods</td>
<td>Specifies the list of allowed methods as part of the CORS protocol.</td>
</tr>
<tr>
<td>allow_headers</td>
<td>Specifies the list of allowed headers as part of the CORS protocol.</td>
</tr>
<tr>
<td>max_age</td>
<td>Specifies the number of seconds the information provided by the Access-Control-Allow-Methods and Access-Control-Allow-Headers headers can be cached. The default value is 86400.</td>
</tr>
<tr>
<td>allow_credentials</td>
<td>Specifies whether the response can be shared when the credentials mode of the request is “include”. The default value is ‘true’.</td>
</tr>
</tbody>
</table>

**Deploy the CORS CRD**

Perform the following to deploy the CORS CRD:

1. Download the CORS CRD.
2. Deploy the CORS CRD using the following command:

   ```bash
   kubectl create -f cors-crd.yaml
   ```

   For example:
How to write a CORS policy configuration

After you have deployed the CORS CRD provided by Citrix in the Kubernetes cluster, you can define the CORS policy configuration in a .yaml file. In the .yaml file, use corspolicy in the kind field and in the spec section add the CORS CRD attributes based on your requirement for the policy configuration.

The following YAML file applies the configured policy to the services listed in the servicenames field. Citrix ADC responds with a 200 OK response code for the pre-flight request if the origin is one of the allow_origins [“random1234.com”, “hotdrink.beverages.com”]. The response includes configured allow_methods, allow_headers, and max_age.

```yaml
apiVersion: citrix.com/v1beta1
kind: corspolicy
metadata:
  name: corspolicy-example
spec:
servicenames:
- "cors-service"
allow_origin:
- "random1234.com"
- "hotdrink.beverages.com"
allow_methods:
- "POST"
- "GET"
- "OPTIONS"
allow_headers:
- "Origin"
- "X-Requested-With"
- "Content-Type"
- "Accept"
- "X-PINGOTHER"
max_age: 86400
allow_credentials: true
```
Citrix ADC ingress controller

After you have defined the policy configuration, deploy the .yaml file using the following commands:

```
1 user@master:~/cors$ kubectl create -f corspolicy-example.yaml
2 corspolicy.citrix.com/corspolicy-example created
```

The Citrix ingress controller applies the policy configuration on the Ingress Citrix ADC device.

Enable request retry feature using AppQoE for Citrix ingress controller

May 25, 2022

When a Citrix ADC appliance receives an HTTP request and forwards it to a back-end server, sometimes there may be connection failures with the back-end server. You can configure the request-retry feature on Citrix ADC to forward the request to the next available server, instead of sending the reset to the client. Hence, the client saves round trip time when Citrix ADC initiates the same request to the next available service. For more information request retry feature, see the Citrix ADC documentation

Now, you can configure request retry on Citrix ADC with Citrix ingress controller. Custom Resource Definitions (CRDs) are the primary way of configuring policies in cloud native deployments. Using the AppQoE CRD provided by Citrix, you can configure request-retry policies on Citrix ADC with the Citrix ingress controller. The AppQoE CRD enables communication between the Citrix ingress controller and Citrix ADC for enforcing AppQoE policies.

AppQoE CRD definition

The AppQoE CRD is available in the Citrix ingress controller GitHub repo at: appqoe-crd.yaml. The AppQoE CRD provides attributes for the various options that are required to define the AppQoE policy on Citrix ADC.

The following are the attributes provided in the AppQoE CRD:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servicenames</td>
<td>Specifies the list of Kubernetes services to which you want to apply the AppQoE policies.</td>
</tr>
<tr>
<td>on-reset</td>
<td>Specifies whether to set retry on connection Reset or Not</td>
</tr>
<tr>
<td>on-timeout</td>
<td>Specifies the time in milliseconds for retry</td>
</tr>
<tr>
<td>number-of-retries</td>
<td>Specifies the number of retries</td>
</tr>
</tbody>
</table>
### Deploy the AppQoE CRD

Perform the following to deploy the AppQoE CRD:

1. Download the AppQoE CRD.
2. Deploy the AppQoE CRD using the following command:

   ```bash
   kubectl create -f appqoe-crd.yaml
   ```

### How to write a AppQoE policy configuration

After you have deployed the AppQoE CRD provided by Citrix in the Kubernetes cluster, you can define the AppQoE policy configuration in a `.yaml` file. In the `.yaml` file, use `appqoepolicy` in the `kind` field and in the `spec` section add the AppQoE CRD attributes based on your requirement for the policy configuration.

The following YAML file applies the AppQoE policy to the services listed in the `servicenames` field. You must configure the AppQoE action to retry on timeout and define the number of retry attempts.

```yaml
apiVersion: citrix.com/v1
kind: appqoepolicy
metadata:
  name: targeturlappqoe
spec:
  appqoe-policies:
    - servicenames:
      - apache
    appqoe-policy:
      operation-retry:
        onReset: 'YES'
        onTimeout: 33
      number-of-retries: 2
```
After you have defined the policy configuration, deploy the .yaml file using the following commands:

$ kubectl create -f appqoe-example.yaml

**Entity name change**

December 16, 2021

While adding the Citrix ADC entities, the Citrix ingress controller maintains unique names per Ingress, service or namespace. Sometimes, it results in Citrix ADC entities with large names even exceeding the name limits in Citrix ADC.

Now, the naming format in the Citrix ingress controller is updated to shorten the entity names. In the updated naming format, a part of the entity name is hashed and all the necessary information is provided as part of the entity comments.

After this update, the comments available on lbvserver and servicegroup entity names provides all the necessary details like the ingress name, ingress port, service name, service port, and the namespace of the application.

**Format for comments**

Ingress: ing:<ingress-name>, ingport:<ingress-port>, ns:<k8s-namespace>, svc:<k8s-servicename>, svcport:<k8s-serviceport>

Service of type LoadBalancer: lbsvc:<k8s-servicename>, svcport:<k8s-serviceport>, ns:<k8s-namespace>

The following table explains the entity name changes introduced with the Citrix ingress controller version 1.12.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Old naming format</th>
<th>New naming format</th>
<th>Description/Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>csvserver (ingress)</td>
<td>k8s-192.2.170.67_80_http</td>
<td>k8s-192.2.170.67_80_http</td>
<td>no changes</td>
</tr>
<tr>
<td>csvserver (type LoadBalancer)</td>
<td>k8s-apache_default_80_http</td>
<td>k8s-apache_default_80_http</td>
<td>Now, the port is followed by a namespace</td>
</tr>
<tr>
<td>Entity</td>
<td>Old naming format</td>
<td>New naming format</td>
<td>Description/Comments</td>
</tr>
<tr>
<td>--------</td>
<td>------------------</td>
<td>------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>lbvserver (type LoadBalancer)</td>
<td>k8s-apache_default_80_ - apache_default_80_svc</td>
<td>k8s-apache_default_80_svc</td>
<td>The comment for type LoadBalancer is now different</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Comment: lbsvc:apache, svcport:80, ns: default</td>
</tr>
<tr>
<td>servicegroup (type LoadBalancer)</td>
<td>k8s-apache_default_80_sgp</td>
<td>k8s-apache_default_80_svc</td>
<td>The suffix sgp is added.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cspolicy or csaction or responder policy</td>
<td>k8s-web-ingress_default_44 frontend_default_</td>
<td>k8s-web-ingress_default_44 frontend_default_</td>
<td>Moved service-name, service-port to the beginning, added suffix of cs, hashed ingress-name, ingress-port, and namespace</td>
</tr>
<tr>
<td></td>
<td>80_csp_2631ng</td>
<td>80_csp_2631ng</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lbvserver (ingress)</td>
<td>k8s-web-ingress_default_44frontend_default_80_lbv_267pneiak5rw6hoygvrqrzpm4k6thz2p</td>
<td>k8s-web-ingress_default_44frontend_default_80_svc</td>
<td>Suffix lbv and comment added to the entity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Comment:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>servicegroup (ingress)</td>
<td>k8s-web-ingress_default_44frontend_default_80_sgp_2631ng</td>
<td>k8s-web-ingress_default_44frontend_default_80_svc</td>
<td>Suffix sgp is added.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lbvserver (UDP)</td>
<td>k8s-web-ingress_default_90_udp_lbv_uyomblblagjt2xzkwpf7w_k8s-bind_default_53-udp_svc</td>
<td>k8s-bind_default_53-udp_svc</td>
<td>UDP is still appended to the port as earlier.</td>
</tr>
</tbody>
</table>
Citrix ADC ingress controller

When you upgrade from an older version of the Citrix ingress controller to the latest version, the Citrix ingress controller renames all the entities with the new naming format. However, the Citrix ingress controller does not handle the downgrade from the latest version to an older version.

Licensing

January 19, 2022

For licensing the Citrix ADC CPX, you need to provide the following information in the YAML for the Citrix Application Delivery Management (ADM) to automatically pick the licensing information:

- LS_IP (License server IP) – Specify the Citrix ADM IP address.
- LS_PORT (License server Port) – This is not a mandatory field. You must specify the ADM port only if you have changed it. The default port is 27000.
- PLATFORM – Specify the Platform License. Platform is CP1000.

The following is a sample yaml file:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    name: cpx-ingress
    name: cpx-ingress
spec:
  replicas: 1
  selector:
    matchLabels:
      name: cpx-ingress
  template:
    metadata:
      annotations:
      NETSCALER_AS_APP: "True"
      labels:
        name: cpx-ingress
    spec:
      serviceAccountName: cpx
      containers:
      - args:
        - --ingress-classes citrix-ingress
        env:
```

© 1999–2022 Citrix Systems, Inc. All rights reserved.
For deploying Citrix cloud native topologies, there are various options available using YAML and Helm charts. Helm charts are one of the easiest ways for deployment in a Kubernetes environment. When you deploy using the Helm charts, you can use a values.yaml file to specify the values of the configurable parameters instead of providing each parameter as an argument.

You can generate the values.yaml file for Citrix cloud native deployments using Citrix deployment builder.
Citrix ADC ingress controller

builder, which is a GUI.

The following topologies are supported by the Citrix deployment builder:

- Single-Tier
  - Ingress
  - Service type LoadBalancer
- Dual-Tier
  - Citrix ADC CPX as NodePort
  - Citrix ADC CPX as service of type LoadBalancer
- Multi-cluster Ingress
- Service mesh

For detailed information on how to use the Citrix deployment builder, see the Citrix deployment builder blog.