NetScaler ingress controller

Contents

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 with in 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 NetScaler Ingress Controller

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

Using NetScaler Ingress Controller, you can configure NetScaler CPX, VPX, or MPX according to the Ingress rules and integrate your NetScalers with the Kubernetes environment.

Why NetScaler Ingress Controller

This topic provides information about some of the key benefits of integrating NetScalers with your Kubernetes cluster using NetScaler Ingress Controller.

Support for TCP and UDP traffic

Standard Kubernetes Ingress solutions provide load balancing only at layer 7 (HTTP or HTTPS traffic). Some times, you need to expose many legacy applications which rely on TCP or UDP applications and need a way to load balance those applications. NetScaler Ingress Controller solution using NetScaler 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

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

Flexible deployment topologies

NetScaler provides flexible and powerful topologies such as Single-Tier and Dual-Tier depending on how you want to manage your NetScalers 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 NetScaler 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. NetScaler provides a way to load balance such services using the Ingress controller and NetScaler. For more information, see Expose services of type LoadBalancer.

Deploy NetScaler Ingress Controller

You can deploy NetScaler Ingress Controller in the following deployment modes:

1. As a standalone pod: This mode is used when managing ADCs such as NetScaler MPX, or VPX that is outside the Kubernetes cluster.
2. As a sidecar in a pod along with the NetScaler CPX in the same pod: The controller is only responsible for the NetScaler CPX that resides in the same pod.

You can deploy the ingress controller provided by NetScaler using Kubernetes YAML or Helm charts. For more information, see Deploy NetScaler Ingress Controller using YAML or Deploy NetScaler Ingress Controller using Helm charts.

Getting started

December 31, 2023

This guide helps you to quickly evaluate NetScaler Ingress Controller for Kubernetes if you are new to NetScaler 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 NetScaler Ingress Controller

In this procedure you perform the following steps:

- Deploy NetScaler CPX (a containerized version of NetScaler) 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 NetScaler CPX with NetScaler Ingress Controller

Perform the following to deploy NetScaler CPX with NetScaler Ingress Controller.

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

```
kubectl create -f https://raw.githubusercontent.com/citrix/cloud-native-getting-started/master/beginners-guide/manifest/cpx.yaml
```

2. Verify the installation using the following command.

```
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.


```
kubectl create -f https://raw.githubusercontent.com/citrix/cloud-native-getting-started/master/beginners-guide/manifest/guestbook-app.yaml
```

2. Verify the installation using the following:

```
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).

```
kubectl create -f https://raw.githubusercontent.com/citrix/cloud-native-getting-started/master/beginners-guide/manifest/guestbook-ingress.yaml
```

2. Verify the Ingress deployment using the following command.

```
kubectl get ingress
```

3. Display information about the service using the following command.

```
kubectl get svc cpx-service

# kubectl get service cpx-service

NAME           TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                    AGE

cpx-service    NodePort    10.106.123.144   <none>        80:30592/TCP,443:31338/TCP   43m
```

You can get the NodePort information from this example, 30592 and 31338 are NodePorts.

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

```
curl -s -H \"Host: www.guestbook.com\" http://<MiniKube-IP-address>:<NodePort> | grep Guestbook
```

Expected output:

```
<title>Guestbook</title>
  <h2>Guestbook</h2>
```


> **Note:**
>
> You can get the Minikube IP address using the `minikube ip` command. You can also use the `minikube service cpx-service --url` command to directly get the URL used in the cURL command.

What is next

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

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

Deployment topologies

December 31, 2023

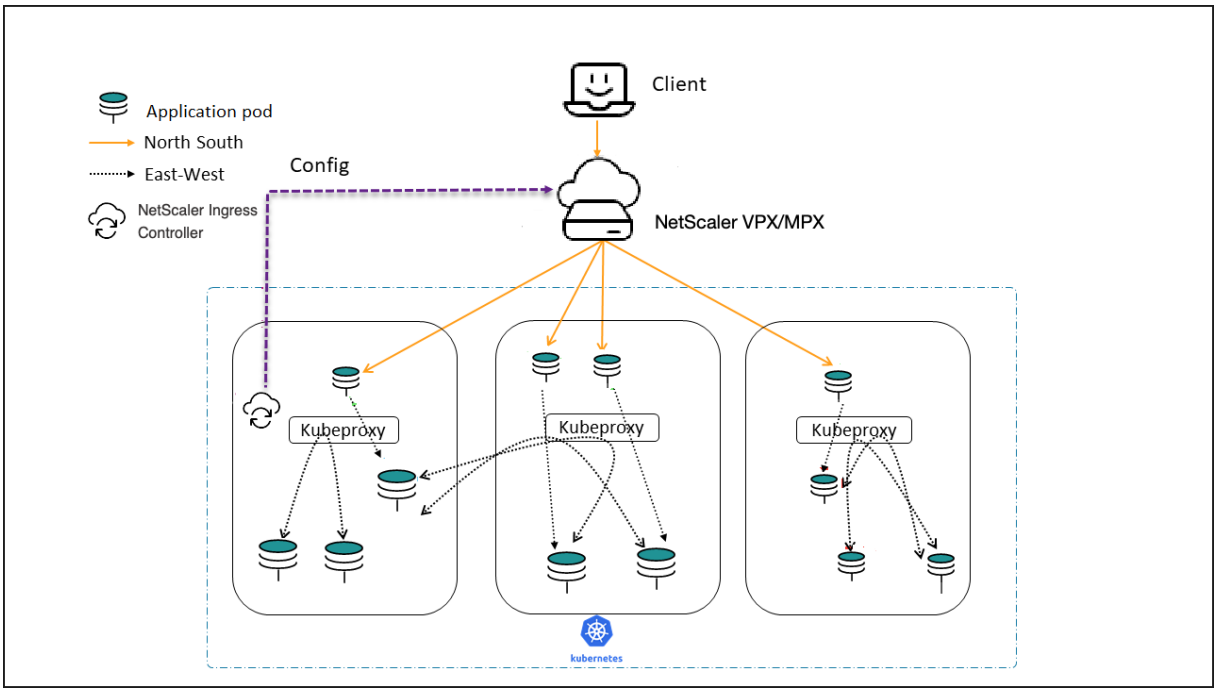
NetScalers can be combined in powerful and flexible topologies that complement organizational boundaries. Dual-tier deployments employ high-capacity hardware or virtualized NetScalers (NetScaler MPX and VPX) in the first tier to offload security functions and implement relatively static organizational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

In Dual-tier deployments, the second tier is within the Kubernetes Cluster (using the NetScaler 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

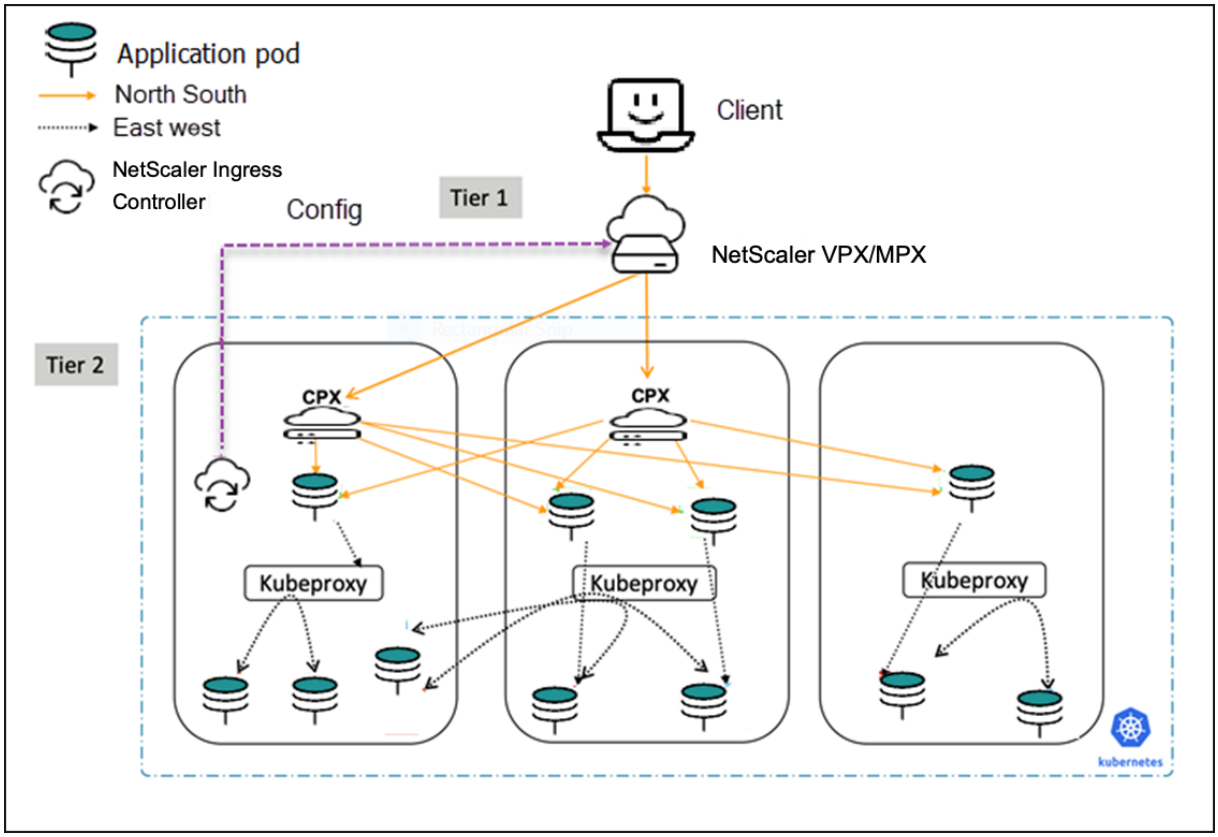
In a Single-Tier topology, NetScaler MPX or VPX devices proxy the (North-South) traffic from the clients to microservices inside the cluster. The NetScaler Ingress Controller is deployed as a standalone pod

in the Kubernetes cluster. The controller automates the configuration of NetScalers (MPX or VPX) based on the changes to the microservices or the Ingress resources.



Dual-Tier topology

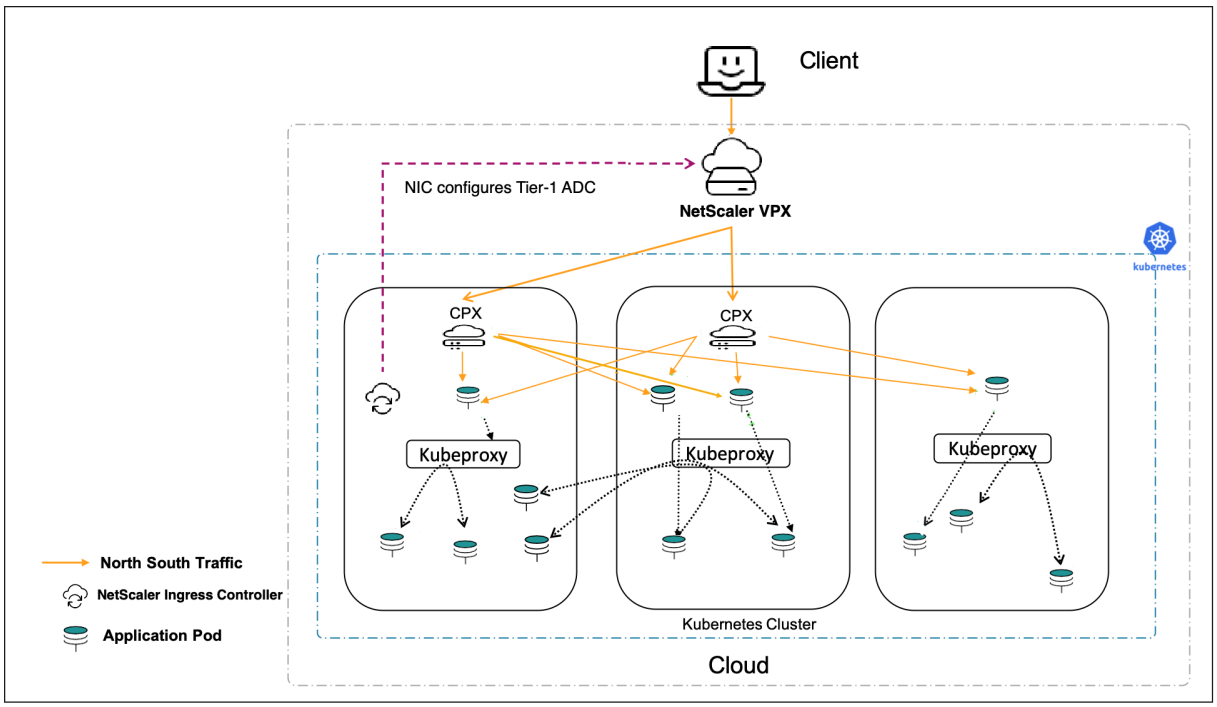
In Dual-Tier topology, NetScaler MPX or VPX devices in Tier-1 proxy the traffic (North-South) from the client to NetScaler CPXs in Tier-2. The Tier-2 NetScaler CPX then routes the traffic to the microservices in the Kubernetes cluster. The NetScaler Ingress Controller deployed as a standalone pod configures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



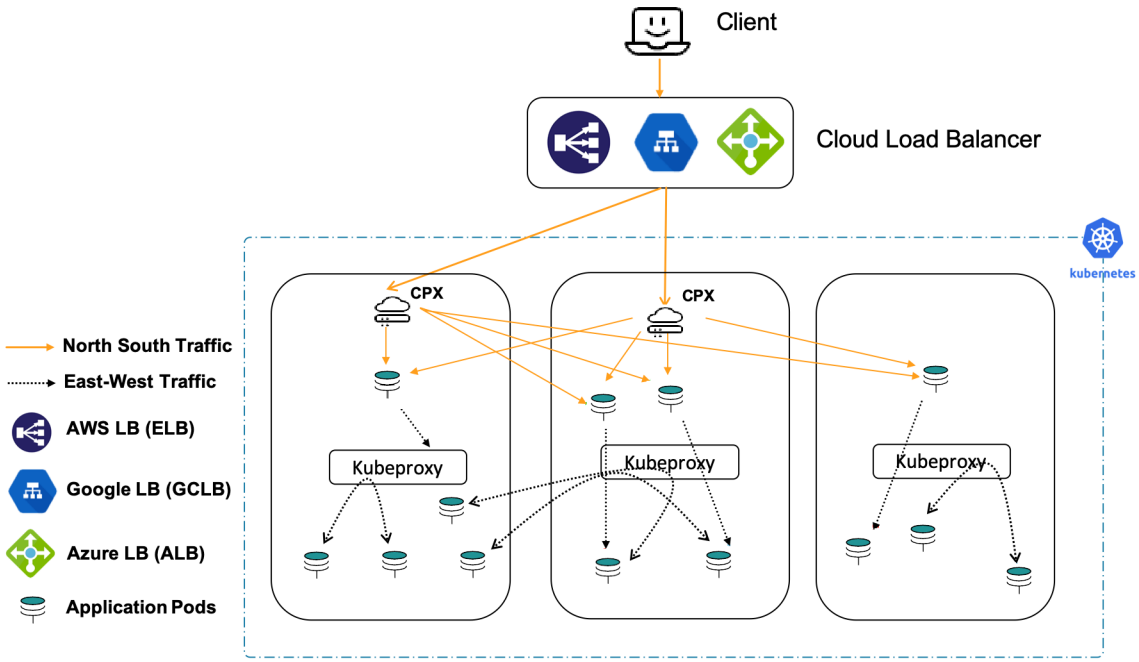
Cloud topology

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 NetScaler CPX. NetScaler 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 NetScaler CPX as the Ingress. If the cloud-based Kubernetes cluster is self-hosted or self-managed, the NetScaler VPX can be used as the first tier in a Dual-tier topology.

Cloud deployment with NetScaler (VPX) in tier-1:



Cloud deployment with Cloud LB in tier-1:



Service mesh lite

An Ingress solution typically performs layer 7 proxy functions for traffic from client to microservices inside the Kubernetes cluster (north-south traffic). The Service mesh lite architecture uses the same

Ingress solution to manage the traffic across services with in the Kubernetes cluster (east-west traffic) as well. Typically, a service mesh solution is used for managing east-west traffic, but it is heavier and complex to manage. Service mesh lite solution is a simplified version of the service mesh architecture and ideal when there is a need to manage both north-south and east-west traffic management. In a service mesh, there are as many sidecar proxies as the number of applications. But, in the service mesh lite architecture, a proxy is deployed as a standalone proxy managing multiple east-west connections. Hence, the Service mesh lite solution is lighter compared to a service mesh because the number of proxies required are less.

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

NetScaler (MPX, VPX, or CPX) can provide such benefits for east-west 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).NetScaler ADM Service Graph is an observability solution to monitor and debug microservices.

For more information, see Service mesh lite.

Following are some of the scenarios when service mesh lite topology is recommended:

- When you need both the north-south and the east-west traffic management for microservices.
- When you need the east-west traffic management through a proxy deployed as a standalone proxy and not as sidecar proxies to microservices.
- When you need the proxy inside the Kubernetes cluster to perform both north-south and east-west traffic management.
- When you need the benefits of service mesh, but wants a lighter and simpler solution.

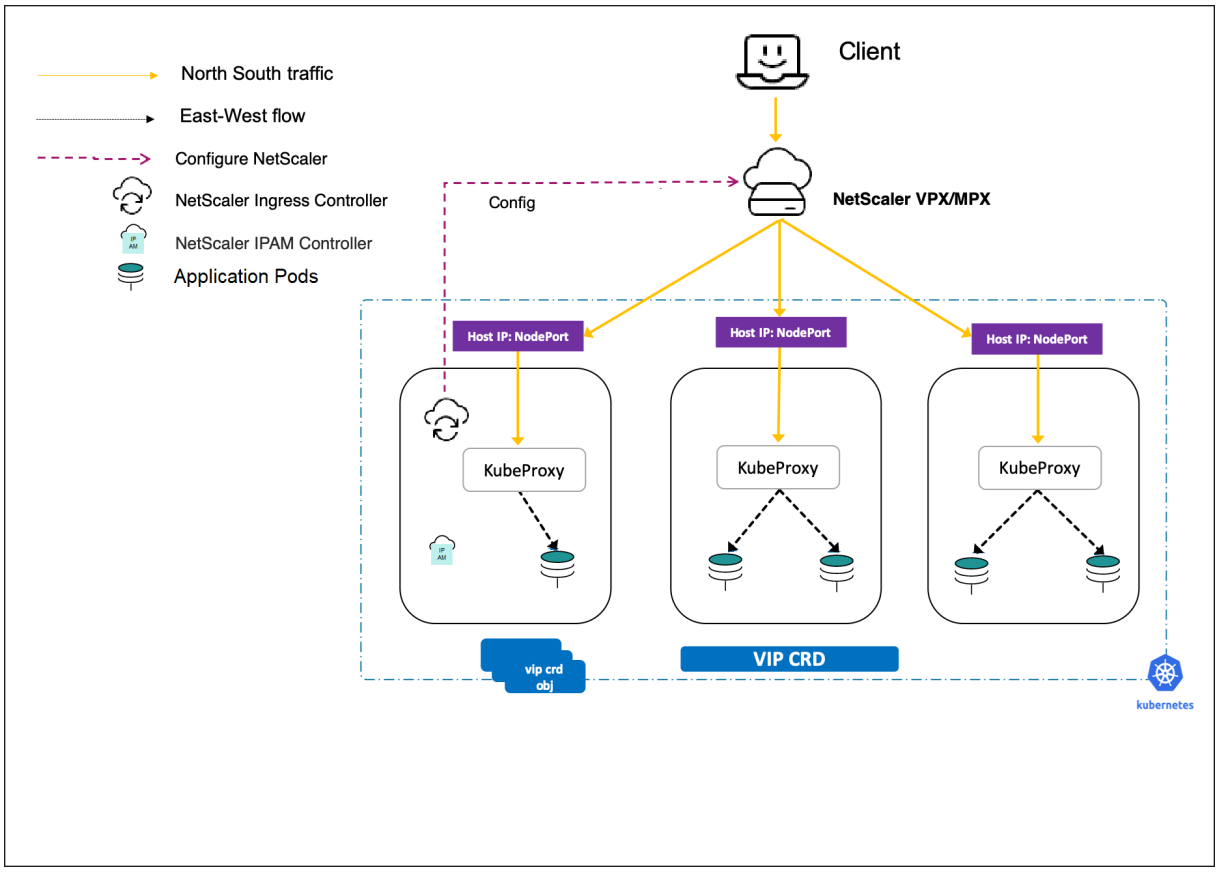
Services of type LoadBalancer

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

For more information, see Expose services of type LoadBalancer.

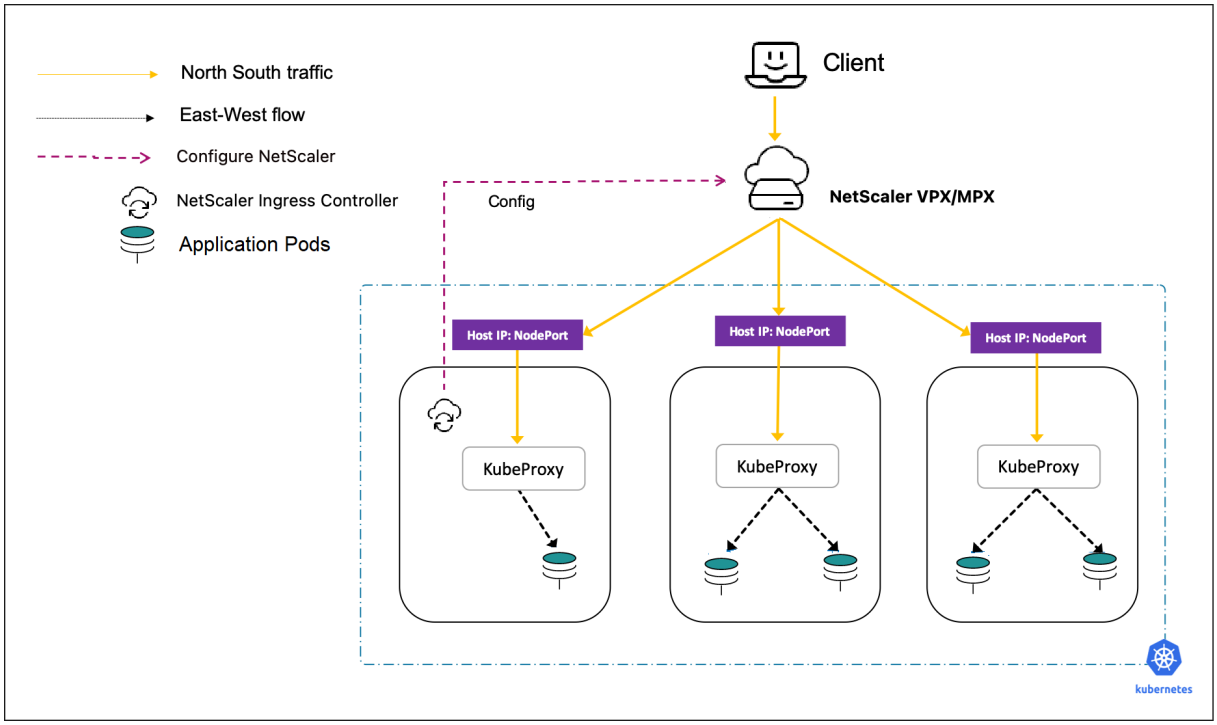


Services of type NodePort

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 NetScaler Ingress Controller supports services of type `NodePort`. Using the Ingress NetScaler and NetScaler Ingress Controller, you can expose the service of type `NodePort` to the outside world.

For more information, see Expose services of type NodePort.



Guidelines for choosing the topology

The following information helps you to choose the right deployment among the topologies Single-Tier and Dual-tier based on your needs.

Single-Tier (Unified Ingress)

Following are some of the scenarios when the Single-Tier (unified ingress) topology is recommended and the benefits:

- Easy to start and adopt because you can use the existing NetScaler as the ingress proxy in front of the Kubernetes cluster.
- When the Network team manages both NetScaler and the Kubernetes deployment.
- Your workload running as microservices is less and a Kubernetes proxy inside the Kubernetes cluster is not required.
- More suitable for north-south traffic deployments.

Dual-Tier

Following are some of the scenarios when the Dual-Tier ingress topology is preferred and the benefits:

- When you have significant workload running as microservices there is a need for a proxy inside the Kubernetes cluster.
- When the external proxy (managed by the network team) and Kubernetes proxies (managed by the platform team) are managed by two different teams.
- You need segregation of functions for proxies external to Kubernetes and for proxies inside Kubernetes. For example, WAF, and SSL offload on external NetScaler and policy enforcement and rate limiting on the Kubernetes proxy.
- The proxy inside the Kubernetes cluster performs north-south traffic management only.

Deployment using Helm charts and the NetScaler deployment builder

For deploying NetScaler 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 NetScaler cloud native deployments using the NetScaler deployment builder, which is a GUI.

The following topologies are supported by the NetScaler deployment builder:

- Single-Tier
  - Ingress
  - Service type LoadBalancer
- Dual-Tier
  - NetScaler CPX as NodePort
  - NetScaler CPX as service of type LoadBalancer
- GSLB Ingress
- Service mesh

For detailed information on how to use the NetScaler deployment builder, see the NetScaler deployment builder blog.

Deploy NetScaler Ingress Controller using YAML

December 31, 2023

You can deploy NetScaler 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 NetScalers (NetScaler MPX or NetScaler VPX) outside the cluster. For example, with dual-tier topologies, or single-tier topology where the single tier is a NetScaler MPX or VPX.
- As a sidecar (in the same pod) with NetScaler CPX in the Kubernetes cluster. The sidecar controller is only responsible for the associated NetScaler CPX within the same pod. This mode is used in dual-tier or cloud topologies.

Deploy NetScaler Ingress Controller as a standalone pod in the Kubernetes cluster for NetScaler MPX or VPX appliances

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

> **Note:**
>
> The NetScaler 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 NetScaler deployment:
  - (Standalone appliances) NSIP - The management IP address of a standalone NetScaler appliance. For more information, see IP Addressing in NetScaler
  - (Appliances in High Availability mode) SNIP - The subnet IP address. For more information, see IP Addressing in NetScaler
  - (Appliances in Clustered mode) CLIP - The cluster management IP (CLIP) address for a clustered NetScaler deployment. For more information, see IP addressing for a cluster
- The user name and password of the NetScaler VPX or MPX appliance used as the Ingress device. The NetScaler appliance must have a system user account (non-default) with certain privileges so that NetScaler Ingress Controller can configure the NetScaler VPX or MPX appliance. For instructions to create the system user account on NetScaler, see Create System User Account for NetScaler Ingress Controller in NetScaler

  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=<
    username> --from-literal=password=<password>
```

Create System User Account for NetScaler Ingress Controller in NetScaler NetScaler Ingress Controller configures the NetScaler appliance (MPX or VPX) using a system user account of the NetScaler. The system user account should have certain privileges so that the NetScaler Ingress Controller has permission to configure the following on the NetScaler:

- Add, Delete, or View Content Switching (CS) virtual server
- Configure CS policies and actions
- Configure Load Balancing (LB) virtual server
- Configure Service groups
- Cofigure 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 NetScaler appliance

To create the system user account, perform the following:

1. Log on to the NetScaler appliance. Perform the following:

   a) Use an SSH client, such as PuTTy, to open an SSH connection to the NetScaler 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 '^\(\?!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+\s+system\s+file)^\(\?!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+\s+system\s+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 NetScaler command line.

For configuring the command policy from the NetScaler configuration wizard (GUI), use the following command policy specification.

```
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)
    .\*|\(^install\s\*\(wi|wf))|\(^\S+\s+system\s+file)^\(?!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+\s+system\s+file)
```

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

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

Deploy NetScaler Ingress Controller as a pod

Perform the following:

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

```
1 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:

Environment Variable	Mandatory or Optional	Description
NS_IP	Mandatory	The IP address of the NetScaler appliance. For more details, see Prerequisites.
NS_USER and NS_PASSWORD	Mandatory	The user name and password of the NetScaler VPX or MPX appliance used as the Ingress device. For more details, see Prerequisites.
EULA	Mandatory	The End User License Agreement. Specify the value as `Yes`.
Kubernetes_url	Optional	The kube-apiserver url that NetScaler Ingress Controller uses to register the events. If the value is not specified, NetScaler Ingress Controller uses the internal kube-apiserver IP address.
LOGLEVEL	Optional	The log levels to control the logs generated by NetScaler 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
NS_PROTOCOL and NS_PORT	Optional	Defines the protocol and port that is used by the NetScaler Ingress Controller to communicate with NetScaler. By default, NetScaler Ingress Controller uses HTTP on port 80. You can also use HTTPS on port 443.

Environment Variable	Mandatory or Optional	Description
ingress-classes	Optional	If multiple ingress load balancers are used to load balance different ingress resources. You can use this environment variable to specify the NetScaler Ingress Controller to configure NetScaler associated with specific ingress class. For information on Ingress classes, see Ingress class support
NS_VIP	Optional	NetScaler Ingress Controller uses the IP address provided in this environment variable to configure a virtual IP address to the NetScaler that receives Ingress traffic. **Note:** NS_VIP acts as a fallback when the frontend-ip annotation is not provided in Ingress yaml. Not supported for Type Loadbalancer service.
NS_APPS_NAME_PREFIX	Optional	By default, the NetScaler Ingress Controller adds "**k8s**" as prefix to the NetScaler entities such as, content switching (CS) virtual server, load balancing (LB) virtual server and so on. You can now customize the prefix using the `NS_APPS_NAME_PREFIX` environment variable in the NetScaler Ingress Controller deployment YAML file. You can use alphanumeric characters for the prefix and the prefix length should not exceed 8 characters.

Environment Variable	Mandatory or Optional	Description
NAMESPACE	Optional	While running a NetScaler 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 NetScaler Ingress Controller can listen on a namespace. You can use the `NAMESPACE` environment variable to specify the namespace. For more information, see [Deploy the NetScaler Ingress Controller for a namespace(#deploy-the-citrix-ingress-controller-for-a-namespace).
POD_IPS_FOR_SERVICEGROUP_MEMBERS	Optional	By default, while configuring services of type LoadBalancer and NodePort on an external tier-1 NetScaler the NetScaler 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.

Environment Variable	Mandatory or Optional	Description
IGNORE_NODE_EXTERNAL_IP	Optional	While adding NodeIP for services of type LoadBalancer or NodePort on an external tier-1 NetScaler, the NetScaler 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`.
NS_DNS_NAMESERVER	Optional	Enables adding DNS nameservers on NetScaler VPX.
NS_CONFIG_DNS_REC	Optional	Enables adding DNS records on NetScaler 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 <abc.com 1.1.1.1>` is executed on NetScaler 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.

Environment Variable	Mandatory or Optional	Description
NS_SVC_LB_DNS_REC	Optional	Enables adding DNS records on NetScaler 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.
SCOPE	Optional	Enables configuring the scope of NetScaler Ingress Controller as `Role` or `ClusterRole` binding. You can set the value of the `SCOPE` environment variable as `local` or `cluster`. When you set this variable as `local`, NetScaler Ingress Controller is deployed with `Role` binding that has limited privileges. You can use this option when you want to deploy NetScaler Ingress Controller with minimal privileges for a particular namespace with `Role` binding. By default, the value of `SCOPE` is set as `cluster` and NetScaler Ingress Controller is deployed with `ClusterRole` binding.

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 NetScaler Ingress Controller is deployed successfully using the following command:


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

Deploy NetScaler Ingress Controller as a sidecar with NetScaler CPX

Use the citrix-k8s-cpx-ingress.yaml file to deploy a NetScaler CPX with NetScaler Ingress Controller as a sidecar. The YAML file deploys a NetScaler 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:

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

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

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

3. Verify if NetScaler Ingress Controller is deployed successfully using the following command:

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

Deploy NetScaler CPX with the NetScaler Ingress Controller as sidecar without the default credentials

Earlier, when you deploy NetScaler CPX with the NetScaler Ingress Controller as a sidecar without specifying the login credentials, the NetScaler Ingress Controller considers `nsroot/nsroot` as the default credentials.

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

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

- For NetScaler 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 NetScaler Ingress Controller and NetScaler CPX:

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

Following is a snippet of the shared volume configuration common for the NetScaler CPX and the NetScaler Ingress Controller.

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

- For earlier NetScaler 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 NetScaler Ingress Controller:

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

Deploy the NetScaler 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 NetScaler Ingress Controller monitors Ingress resources across all namespaces in the Kubernetes cluster. If multiple teams want to manage the same NetScaler, they can deploy a `Role` based NetScaler 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 NetScaler Ingress Controller. In this case, the NetScaler Ingress Controller listens only for events from the specified namespace and then configure the NetScaler accordingly.

You can use the `SCOPE` environment variable to configure the scope of NetScaler Ingress Controller as `Role` or `ClusterRole` binding. You can set the value of the `SCOPE` environment variable as `local` or `cluster`. When you set this variable as `local`, NetScaler Ingress Controller is deployed with minimal privileges for a particular namespace with `Role` binding. By default, the value of `SCOPE` is set as `cluster` and NetScaler Ingress Controller is deployed with the `ClusterRole` binding.

The following example shows a sample YAML file which defines a Role and RoleBinding for deploying a NetScaler Ingress Controller for a specific namespace.

```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: citrix
rules:
  - apiGroups: [""]
    resources:
      [
        "endpoints",
        "pods",
        "secrets",
        "nodes",
        "routes",
        "namespaces",
        "configmaps",
        "services",
      ]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["services/status"]
    verbs: ["patch"]
  - apiGroups: [""]
    resources: ["events"]
    verbs: ["create"]
  - apiGroups: ["extensions"]
    resources: ["ingresses", "ingresses/status"]
    verbs: ["get", "list", "watch", "patch"]
  - apiGroups: ["networking.k8s.io"]
    resources: ["ingresses", "ingresses/status", "ingressclasses"]
    verbs: ["get", "list", "watch", "patch"]
  - apiGroups: ["apiextensions.k8s.io"]
    resources: ["customresourcedefinitions"]
    verbs: ["get", "list", "watch"]
```



```
34    - apiGroups: ["apps"]
35      resources: ["deployments"]
36      verbs: ["get", "list", "watch"]
37    - apiGroups: ["citrix.com"]
38      resources:
39        [
40          "rewritepolicies",
41          "authpolicies",
42          "ratelimits",
43          "listeners",
44          "httproutes",
45          "continuousdeployments",
46          "apigatewaypolicies",
47          "wafs",
48          "bots",
49          "corspolicies",
50          "appqoepolicies",
51        ]
52      verbs: ["get", "list", "watch", "create", "delete", "patch"]
53    - apiGroups: ["citrix.com"]
54      resources:
55        [
56          "rewritepolicies/status",
57          "continuousdeployments/status",
58          "authpolicies/status",
59          "ratelimits/status",
60          "listeners/status",
61          "httproutes/status",
62          "wafs/status",
63          "apigatewaypolicies/status",
64          "bots/status",
65          "corspolicies/status",
66          "appqoepolicies/status",
67        ]
68      verbs: ["patch"]
69    - apiGroups: ["citrix.com"]
70      resources: ["vips"]
71      verbs: ["get", "list", "watch", "create", "delete"]
72    - apiGroups: ["route.openshift.io"]
73      resources: ["routes"]
74      verbs: ["get", "list", "watch"]
75    - apiGroups: ["crd.projectcalico.org"]
76      resources: ["ipamblocks"]
77      verbs: ["get", "list", "watch"]
78  ---
79  kind: RoleBinding
80  apiVersion: rbac.authorization.k8s.io/v1
81  metadata:
82    name: citrix
83  roleRef:
84    apiGroup: rbac.authorization.k8s.io
85    kind: Role
86    name: citrix
```



```
subjects:
  - kind: ServiceAccount
    name: citrix
    namespace: test
<!--NeedCopy-->
```

Restrictions

When the NetScaler 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 NetScaler Ingress Controller using Helm charts

December 31, 2023

You can deploy the NetScaler 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 NetScalers (NetScaler MPX or NetScaler VPX) outside the cluster. For example, with dual-tier topologies, or single-tier topology where the single tier is a NetScaler MPX or VPX.
- As a sidecar (in the same pod) with NetScaler CPX in the Kubernetes cluster. The sidecar controller is only responsible for the associated NetScaler CPX within the same pod. This mode is used in dual-tier or cloud topologies.

The helm charts for the NetScaler 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, NetScaler provides the NetScaler deployment builder which is a GUI for generating the `values.yaml` file for NetScaler cloud native deployments.

Deploy the NetScaler Ingress Controller as a standalone pod in the Kubernetes cluster

Use the netscaler-ingress-controller chart to run the NetScaler Ingress Controller as a pod in your Kubernetes cluster. The chart deploys the NetScaler Ingress Controller as a pod in your Kubernetes cluster and configures the NetScaler 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 anyone of the following depending on the type of NetScaler deployment:
  - (Standalone appliances) NSIP - The management IP address of a standalone NetScaler appliance. For more information, see IP Addressing in NetScaler.
  - (Appliances in High Availability mode) SNIP - The subnet IP address. For more information, see IP Addressing in NetScaler.
  - (Appliances in Clustered mode) CLIP - The cluster management IP (CLIP) address for a clustered NetScaler deployment. For more information, see IP addressing for a cluster.
- The user name and password of the NetScaler VPX or MPX appliance used as the Ingress device. The NetScaler appliance needs to have a system user account (non-default) with certain privileges so that the NetScaler Ingress Controller can configure the NetScaler VPX or MPX appliance. For instructions to create the system user account on NetScaler, see Create System User Account for NetScaler Ingress Controller in NetScaler.

  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=<
    username> --from-literal=password=<password>
```

Create a system user account for the NetScaler Ingress Controller in NetScaler The NetScaler Ingress Controller configures the NetScaler using a system user account of the NetScaler. The system user account should have certain privileges so that the NetScaler Ingress Controller has permission to configure the following on the NetScaler:

- Add, delete, or view content switching (CS) virtual server
- Configure CS policies and actions
- Configure Load Balancing (LB) virtual server
- Configure service groups
- Cofigure 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 NetScaler appliance

To create the system user account, perform the following:

1. Log on to the NetScaler appliance. Perform the following:

   a) Use an SSH client, such as PuTTy, to open an SSH connection to the NetScaler 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 '^\(\?!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+\s+system\s+file)^\(\?!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+\s+system\s+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 NetScaler command line.

   For configuring the command policy from NetScaler configuration wizard (GUI), use the following command policy specification.

```
^\(?!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+\s+system\s+file)^\(?!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+\s+system\s+file)
```

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

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

To deploy the NetScaler Ingress Controller as a standalone pod:

To deploy the NetScaler Ingress Controller as standalone pod, follow the instructions provided in the NetScaler Ingress Controller Artifact Hub.

Deploy the NetScaler Ingress Controller as a sidecar with NetScaler CPX in the Kubernetes cluster

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

To deploy NetScaler CPX with the NetScaler Ingress Controller as a sidecar, follow the instruction provided in the NetScaler Ingress Controller Helm Hub.

Deploy NetScaler Ingress Controller using kops

December 31, 2023

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. NetScaler provides a kops add-on for deploying NetScaler Ingress Controller.

Deploy NetScaler Ingress Controller using kops during cluster creation

Perform the following steps to deploy NetScaler 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 NetScaler 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 add-don.

Deploy NetScaler Ingress Controller using kops after cluster creation

You can use the `kubectl` command to deploy the NetScaler Ingress Controller add-on with kops after creating the cluster.

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

Deploy the NetScaler Ingress Controller on a Rancher managed Kubernetes cluster

December 31, 2023

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 NetScaler Ingress Controller is built around the Kubernetes Ingress and it can automatically configure one or more NetScalers based on the Ingress resource configuration. You can deploy the NetScaler Ingress Controller in a Rancher managed Kubernetes cluster to extend the advanced load balancing and traffic management capabilities of NetScaler to your cluster.

Prerequisites

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

Deployment options

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

Based on how you want to use NetScaler, there are two ways to deploy the NetScaler Ingress Controller in a Kubernetes cluster on the Rancher platform:

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

Deploy the NetScaler Ingress Controller as a sidecar with NetScaler CPX

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

Perform the following steps to deploy the NetScaler Ingress Controller as a sidecar with NetScaler CPX on the Rancher platform.

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

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

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 NetScaler Ingress Controller is deployed successfully using the following command.

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


Deploy the NetScaler Ingress Controller as a standalone pod

In this deployment, NetScaler Ingress Controller which runs as a stand-alone pod allows you to control the NetScaler 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 NetScaler Ingress Controller.

To deploy the NetScaler 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 NetScaler Ingress Controller as a pod.

   Note: To update the `Status.LoadBalancer.Ingress` field of the Ingress resources managed by the NetScaler Ingress Controller with the allocated IP addresses, you must specify the command line argument `--update-ingress-status yes` when you start the NetScaler 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.
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 NetScaler Ingress Controller is deployed successfully using the following command.

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


Deploy the NetScaler Ingress Controller on a PKS managed Kubernetes cluster

December 31, 2023

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

The NetScaler Ingress Controller is built around the Kubernetes Ingress and it can automatically configure one or more NetScalers based on the Ingress resource configuration. You can deploy the NetScaler Ingress Controller in a PKS managed Kubernetes cluster to extend the advanced load balancing and traffic management capabilities of NetScaler 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 NetScaler CPXs as pods inside the cluster or deploy a NetScaler MPX or VPX appliance outside the Kubernetes cluster.

Based on how you want to use NetScaler, there are two ways to deploy the NetScaler Ingress Controller in a Kubernetes cluster on the PKS:

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

Deploy NetScaler Ingress Controller as a pod

Follow the instruction provided in topic: Deploy NetScaler Ingress Controller as a standalone pod in the Kubernetes cluster for NetScaler MPX or VPX appliances.

Deploy NetScaler Ingress Controller as a sidecar with NetScaler CPX

Follow the instruction provided in topic: Deploy NetScaler Ingress Controller as a sidecar with NetScaler CPX.

Network Configuration

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

Deploy NetScaler-Integrated Canary Deployment Solution

December 31, 2023

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.

NetScaler provides the following options for canary deployment using the NetScaler 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.

NetScaler-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.

NetScaler comes with a rich application-centric configuration module and provides complete visibility to application traffic and health of application instances. The capabilities of NetScaler to generate accurate performance statistics can be leveraged for Canary analysis to take better decisions about the Canary deployment. In this solution, NetScaler is integrated with the Spinnaker platform and acts as a source for providing accurate metrics for analyzing Canary deployment using Kayenta.

NetScaler Metrics Exporter 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 NetScaler 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 NetScaler 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.

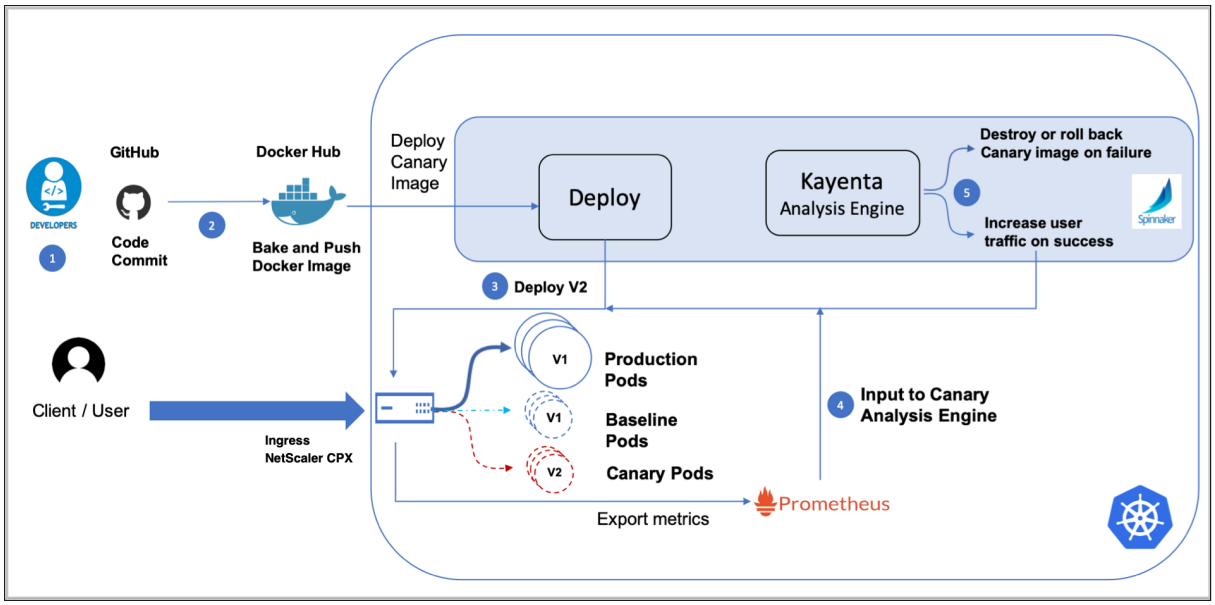
- Prometheus: Prometheus is an open-source systems monitoring and alerting toolkit. Prometheus is a monitoring system which can maintain a huge amount of data in a time series database. NetScaler Metrics exposes the performance metrics to Spinnaker through Prometheus.
- Jenkins: Jenkins is an open source 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.
- NetScaler Ingress Controller NetScaler provides an Ingress Controller for NetScaler MPX (hardware), NetScaler VPX (virtualized), and NetScaler CPX (containerized) for bare metal and cloud deployments. The NetScaler Ingress Controller is built around Kubernetes Ingress and automatically configures one or more NetScalers based on the Ingress resource configuration.

Following NetScaler software versions are required for Citrix-Integrated Canary Deployment Solution:

- NetScaler Ingress Controller build/version: `quay.io/citrix/citrix-k8s-ingress-controller:1.29.5`.
- NetScaler CPX version: `quay.io/citrix/citrix-k8s-cpx-ingress:13.0-83.27`.
- NetScaler Metrics Exporter version: `quay.io/citrix/netscaler-metrics-exporter:1.4.0`.

Workflow of a Spinnaker pipeline for NetScaler-Integrated Canary Deployment Solution

The following diagram explains the workflow of a Spinnaker pipeline for NetScaler-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. NetScaler collects the performance statistics and exports the statistics to Prometheus with the help of NetScaler 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 NetScaler-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.

```
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.

```
kubectl apply --as=admin --as-group=system:masters -f \
https://storage.googleapis.com/stackdriver-prometheus-
    documentation/rbac-setup.yml
```



```
curl -sS \"https://storage.googleapis.com/stackdriver-prometheus-documentation/prometheus-service.yml\" |
\sed \"s/\_stackdriver\_project\_id:.\*/\_stackdriver\_project\_id: \$GOOGLE\_CLOUD\_PROJECT/\" |
\sed \"s/\_kubernetes\_cluster\_name:.\*/\_kubernetes\_cluster\_name: kayenta-tutorial/\" |
\sed \"s/\_kubernetes\_location:.\*/\_kubernetes\_location: us-central1-f/\" |
\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.

```
dockerRegistry:
            enabled: true
            accounts:
            - name: my-docker-registry
            requiredGroupMembership: []
            providerVersion: V1
            permissions: {
  }

            address: https://index.docker.io
            username: <username>
            password: <password>
            email: <mail-id>
            cacheIntervalSeconds: 30
            clientTimeoutMillis: 60000
            cacheThreads: 1
            paginateSize: 100
            sortTagsByDate: false
            trackDigests: false
            insecureRegistry: false
            repositories:- <repository-name>
            primaryAccount: my-docker-registry
```

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

```
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-software-properties apt-transport-https
sudo add-apt-repository https://dl.bintray.com/spinnaker-releases/debians
```

> **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.

```
    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 NetScaler Metrics Exporter and perform the steps.

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

```
  data:
  config: |
    deploymentConfigurations:
    canary:
    enabled: true
serviceIntegrations:
      - name: prometheus
  enabled: true
accounts:
  - name: my-prometheus
endpoint:
      baseUrl: prometheus-endpoint
    supportedTypes:
- METRICS_STORE
  data:
  config: |
    deploymentConfigurations:
    metricStores:
      prometheus:
      enabled: true
      add_source_metalabels: true
      stackdriver:
```



```
  enabled: true
    period: 30
    enabled: true
```

g) To integrate Slack for notification with Spinnaker, update the following values in the `quick-install.yml` file.

```
data:
 config: |
  deploymentConfigurations:
   notifications:
    slack:
     enabled: true
      botName: <BotName>
      token: <token>
```

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:

```
DECK_POD=$(kubectl -n spinnaker get pods -l \
cluster=spin-deck,app=spin \
-o=jsonpath='{
 .items[0].metadata.name }
 ')
kubectl -n spinnaker port-forward $DECK_POD 8080:9000 >/dev/
    null &
```

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 NetScaler-Integrated Canary Deployment Solution. In this example, NetScaler CPX, MPX, or VPX is deployed as an Ingress device for a GKE cluster. NetScaler 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 NetScaler by deploying the rbac.yaml file.

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

2. You can either deploy the NetScaler Ingress Controller as a sidecar with NetScaler CPX or as a standalone pod which controls NetScaler VPX or MPX.

   Use the cpx-with-cic-sidecar.yml file to deploy the NetScaler Ingress Controller as a sidecar with NetScaler CPX. It also deploys NetScaler Metrics Exporter on the same pod.

```
kubectl apply -f cpx-with-cic-sidecar.yml
```

   To deploy the NetScaler Ingress Controller as a stand-alone pod for NetScaler VPX or MPX use the cic-vpx.yaml file. In this deployment, you should use the exporter.yaml file to deploy NetScaler Metrics Exporter.

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


> **Note:**
>
> Depending on how you are deploying the NetScaler Ingress Controller, you must edit the YAML file for NetScaler Ingress Controller deployment and modify values for the environmental variables as provided in deploying NetScaler Ingress Controller.

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

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

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

4. Once Spinnaker is exposed using NetScaler, 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.

Field	Description
serviceNames	List of services on which this CRD has to be applied
deployment	Specifies the deployment strategy as Kayenta.
percentage	Specifies the percentage of traffic to be diverted from production to baseline and canary.
matchExpression (optional)	Any NetScaler 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.
Spinnaker	Specifies the Spinnaker pipeline configurations you want to apply for your services.
domain	IP address or domain name of the Spinnaker gate.
port	Port number of the Spinnaker gate.
applicationName	The name of the application in Spinnaker.
pipelineName	The name of the pipeline under the Spinnaker application.
serviceName	Specifies the name of the service to which you want to apply the Spinnaker configuration.

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.

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


Troubleshooting

For troubleshooting the deployment, perform the following steps.

1. Check the pod logs for the respective components like Spinnaker, Prometheus, Kayenta, NetScaler CPX, NetScaler Metrics Exporter, NetScaler Ingress Controller.
2. Check the pod logs of the NetScaler Ingress Controller for any configuration-related errors while configuring the NetScaler proxy.
3. Search for the `exception`/`Exception` keyword in the NetScaler 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

```
{

  "appConfig": {
  }
 ,
  "description": "This pipeline deploys a canary version of the
      application, and a baseline (identical to production) version.\nIt
       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"
       }
```



```
23  ,
24        "useDefaultArtifact": false,
25        "usePriorExecution": false
26      }
27
28   ],
29   "keepWaitingPipelines": false,
30   "lastModifiedBy": "anonymous",
31   "limitConcurrent": true,
32   "parallel": true,
33   "parameterConfig": [],
34   "stages": [
35     {
36
37       "account": "my-kubernetes-account",
38       "cloudProvider": "kubernetes",
39       "kinds": [
40         "Deployment",
41         "ConfigMap"
42       ],
43       "labelSelectors": {
44
45         "selectors": [
46           {
47
48             "key": "version",
49             "kind": "EQUALS",
50             "values": [
51               "canary"
52             ]
53            }
54
55         ]
56        }
57  ,
58       "location": "default",
59       "name": "Delete Canary",
60       "options": {
61
62         "cascading": true
63        }
64  ,
65       "refId": "12",
66       "requisiteStageRefIds": [
67         "19",
68         "26"
69       ],
70       "type": "deleteManifest"
71      }
72  ,
73     {
74
75       "account": "my-kubernetes-account",
```



```
76        "cloudProvider": "kubernetes",
77        "kinds": [
78          "Deployment"
79        ],
80        "labelSelectors": {
81
82          "selectors": [
83            {
84
85              "key": "version",
86              "kind": "EQUALS",
87              "values": [
88                "baseline"
89              ]
90            }
91
92          ]
93        }
94    ,
95        "location": "default",
96        "name": "Delete Baseline",
97        "options": {
98
99          "cascading": true
100       }
101   ,
102       "refId": "13",
103       "requisiteStageRefIds": [
104         "19",
105         "26"
106       ],
107       "type": "deleteManifest"
108     }
109   ,
110     {
111
112       "name": "Successful deployment",
113       "preconditions": [],
114       "refId": "14",
115       "requisiteStageRefIds": [
116         "12",
117         "13"
118       ],
119       "type": "checkPreconditions"
120     }
121   ,
122     {
123
124       "application": "sampleapplicaion",
125       "expectedArtifacts": [
126         {
127
128           "defaultArtifact": {
```



```
129
130             "kind": "custom"
131           }
132 ,
133         "id": "9185c756-c6cd-49bc-beee-e3f7118f3412",
134         "matchArtifact": {
135
136             "kind": "docker",
137             "name": "index.docker.io/sample/demo",
138             "type": "docker/image"
139           }
140 ,
141         "useDefaultArtifact": false,
142         "usePriorExecution": false
143       }
144
145     ],
146     "failPipeline": true,
147     "name": "Deploy to Production",
148     "pipeline": "7048e5ac-2464-4557-a05a-bec8bdf868fc",
149     "refId": "19",
150     "requisiteStageRefIds": [
151       "25"
152     ],
153     "stageEnabled": {
154
155       "expression": "\"${
156  #stage('Canary Analysis')['status'].toString() == 'SUCCEEDED' }
157 \"",
158       "type": "expression"
159      }
160 ,
161     "type": "pipeline",
162     "waitForCompletion": true
163    }
164 ,
165   {
166
167     "account": "my-kubernetes-account",
168     "cloudProvider": "kubernetes",
169     "manifestArtifactAccount": "embedded-artifact",
170     "manifests": [
171       {
172
173         "apiVersion": "apps/v1",
174         "kind": "Deployment",
175         "metadata": {
176
177           "labels": {
178
179             "name": "sampleapplicaion-prod",
180             "version": "baseline"
181            }
```



```
,
            "name": "sampleapplicaion-baseline-deployment",
            "namespace": "default"
          }
,
          "spec": {

            "replicas": 4,
            "strategy": {

              "rollingUpdate": {

                "maxSurge": 10,
                "maxUnavailable": 10
              }
,
              "type": "RollingUpdate"
            }
,
            "template": {

              "metadata": {

                "labels": {

                  "name": "sampleapplicaion-prod"
                }

              }
,
              "spec": {

                "containers": [
                  {

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

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

                    ]
                  }

                ]
              }

            }

```



```
235             }
236
237           }
238
239       ],
240       "moniker": {
241
242         "app": "sampleapplicaion"
243        }
244   ,
245       "name": "Deploy Baseline",
246       "refId": "20",
247       "relationships": {
248
249         "loadBalancers": [],
250         "securityGroups": []
251        }
252   ,
253       "requisiteStageRefIds": [],
254       "source": "text",
255       "type": "deployManifest"
256      }
257   ,
258     {
259
260       "account": "my-kubernetes-account",
261       "cloudProvider": "kubernetes",
262       "manifestArtifactAccount": "embedded-artifact",
263       "manifests": [
264         {
265
266           "apiVersion": "apps/v1",
267           "kind": "Deployment",
268           "metadata": {
269
270             "labels": {
271
272               "name": "sampleapplicaion-prod",
273               "version": "canary"
274              }
275   ,
276             "name": "sampleapplicaion-canary-deployment",
277             "namespace": "default"
278            }
279   ,
280           "spec": {
281
282             "replicas": 4,
283             "strategy": {
284
285               "rollingUpdate": {
286
287                 "maxSurge": 10,
```



```
288                 "maxUnavailable": 10
289               }
290 ,
291               "type": "RollingUpdate"
292             }
293 ,
294           "template": {
295
296             "metadata": {
297
298               "labels": {
299
300                 "name": "sampleapplicaion-prod"
301                }
302
303               }
304 ,
305             "spec": {
306
307               "containers": [
308                 {
309
310                   "image": "index.docker.io/sample/demo",
311                   "imagePullPolicy": "Always",
312                   "name": "sampleapplicaion-prod",
313                   "ports": [
314                     {
315
316                       "containerPort": 8080,
317                       "name": "port-8080"
318                      }
319
320                   ]
321                  }
322
323               ]
324              }
325
326             }
327
328           }
329
330         }
331
332       ],
333       "moniker": {
334
335         "app": "sampleapplicaion"
336        }
337 ,
338       "name": "Deploy Canary",
339       "refId": "21",
340       "relationships": {
```



```
341
342         "loadBalancers": [],
343         "securityGroups": []
344        }
345 ,
346      "requiredArtifactIds": [
347        "ac842617-988f-48dc-a7a4-7f020d93cc42"
348      ],
349      "requisiteStageRefIds": [],
350      "source": "text",
351      "type": "deployManifest"
352     }
353 ,
354    {
355
356      "analysisType": "realTime",
357      "canaryConfig": {
358
359        "beginCanaryAnalysisAfterMins": "2",
360        "canaryAnalysisIntervalMins": "",
361        "canaryConfigId": "7bdb4ab4-f933-4a41-865f-6d3e9c786351",
362        "combinedCanaryResultStrategy": "LOWEST",
363        "lifetimeDuration": "PT0H5M",
364        "metricsAccountName": "my-prometheus",
365        "scopes": [
366          {
367
368            "controlLocation": "default",
369            "controlScope": "k8s-sampleapplicaion.default.80.k8s-
                 sampleapplicaion.default.8080.svc-baseline",
370            "experimentLocation": "default",
371            "experimentScope": "k8s-sampleapplicaion.default.80.k8s-
                 sampleapplicaion.default.8080.svc-canary",
372            "extendedScopeParams": {
373  }
374 ,
375            "scopeName": "default"
376           }
377
378        ],
379        "scoreThresholds": {
380
381          "marginal": "0",
382          "pass": "70"
383         }
384 ,
385        "storageAccountName": "kayenta-minio"
386       }
387 ,
388      "name": "Canary Analysis",
389      "refId": "25",
390      "requisiteStageRefIds": [
391        "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"
      }

   ],
   "triggers": [
     {

        "account": "my-docker-registry",
        "enabled": true,
        "expectedArtifactIds": [
          "ac842617-988f-48dc-a7a4-7f020d93cc42"
        ],
        "organization": "sample",
        "payloadConstraints": {
   }
  ,
        "registry": "index.docker.io",
        "repository": "sample/demo",
        "source": "dockerhub",
        "type": "webhook"
      }

   ],
   "updateTs": "1553144362000"
  }

<!--NeedCopy-->
```


A sample JSON file for automated canary configuration

Following is a sample JSON file for automated canary configuration.

```
{

  "applications": [
    "sampleapplicaion"
  ],
  "classifier": {

    "groupWeights": {

      "Group 1": 70,
      "Group 2": 30
     }
 ,
    "scoreThresholds": {

      "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 NetScaler 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 NetScaler 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 NetScaler Ingress Controller using the steps in deploy the NetScaler Ingress Controller. You can either deploy the NetScaler Ingress Controller as a sidecar with NetScaler CPX or as a standalone pod which controls NetScaler 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.

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

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

```
kubectl apply -f canary-ingress.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 NetScaler Ingress Controller to configure the NetScaler 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 NetScaler Ingress Controller, replace the canary annotation `ingress.citrix.com/canary-weight:` 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 NetScaler 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 `canary-ingress.yaml` file.

> **Note:**
>
> You can see the Canary example YAMLs for achieving canary based on header and canary based on header value.

Deploy NetScaler IPAM controller

April 2, 2024

NetScaler provides an IPAM controller for IP address management. NetScaler IPAM controller runs in parallel to NetScaler Ingress Controller in the Kubernetes cluster. NetScaler IPAM controller allocates IP addresses to services of type LoadBalancer and ingress resources from a specified IP address range.

NetScaler IPAM controller requires NetScaler's VIP custom resource definition (CRD). The VIP CRD is used for internal communication between NetScaler Ingress Controller and NetScaler IPAM controller.

Prerequisites

- Kubernetes cluster and a kubectl command-line tool to communicate with the cluster.
- Create a namespace called `netscaler` to isolate resources. Run the following command to create a namespace:

  ```
  kubectl create namespace netscaler
  ```

- Install NetScaler Ingress Controller for your NetScaler VPX or NetScaler MPX using the following Helm commands.

> **Note:**
>
> Ensure to create a secret using NetScaler VPX or NetScaler MPX credentials before running the following commands.

```
helm repo add netscaler https://netscaler.github.io/netscaler-helm-charts/

helm install NetScaler-ingress-controller netscaler/NetScaler-ingress-controller --set nsIP=<NSIP of MPX/VPX>,license.accept=yes,adcCredentialSecret=<Secret-for-ADC-credentials>,ingressClass[0]=netscaler,serviceClass[0]=netscaler,ipam=true,crds.install=true -n netscaler
<!--NeedCopy-->
```


For detailed information about deploying and configuring NetScaler Ingress Controller using Helm charts, see the Helm chart repository.

Deploy IPAM controller

1. Add the NetScaler Helm chart repository to your local registry using the following command.

```
helm repo add netscaler https://netscaler.github.io/netscaler-helm-charts/
<!--NeedCopy-->
```

2. Install NetScaler IPAM controller using the following command.

```
helm install netscaler-ipam-controller netscaler/netscaler-ipam-controller --set vipRange='[{
"<VIP-range-key>": ["<ip-range>"] }
]' -n netscaler
<!--NeedCopy-->
```

For information about all the configurable parameters while installing the IPAM controller using Helm charts, see the Helm chart repository.

IP address allocations

- For services of type LoadBalancer, a unique IP address is allocated to each service from the VIP range.
- For an ingress resource, an IP address in the specified IP range is allocated. When more ingress resources refer to the same VIP range, the IP address allocated to the first ingress resource is allocated to all the other ingress resources.
- Both services of type LoadBalancer and ingress resources can use NetScaler IPAM controller for IP address allocations at the same time. If an IP address is allocated to any one resource type, it is not available for another resource type. But, the same IP address can be allocated to multiple ingress resources.

Environment variables in IPAM controller

This section provides information about the environment variables in NetScaler 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.

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.

Reference

- For information about exposing services of type LoadBalancer with IP addresses assigned by the IPAM controller, see this section.

Deploying NetScaler API Gateway using Rancher

December 31, 2023

NetScaler API Gateway provides a single entry point for APIs by ensuring secure and reliable access to APIs and microservices on your system. NetScaler provides an enterprise-grade API gateway for North-South API traffic for Kubernetes clusters.
NetScaler API Gateway integrates with Kubernetes through the NetScaler Ingress Controller and the NetScaler (NetScaler MPX, VPX, or CPX) deployed as the Ingress Gateway for on-premises and cloud deployments.

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

Prerequisites

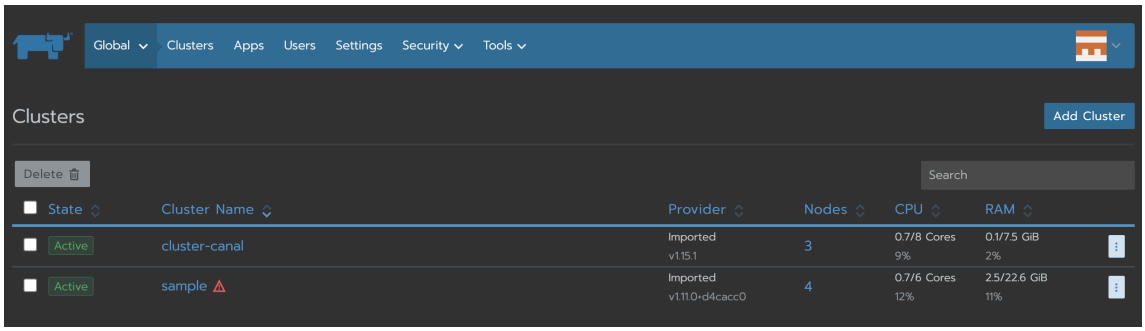
You must import the cluster, in which you want to deploy the API gateway, to the Rancher platform.

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**.

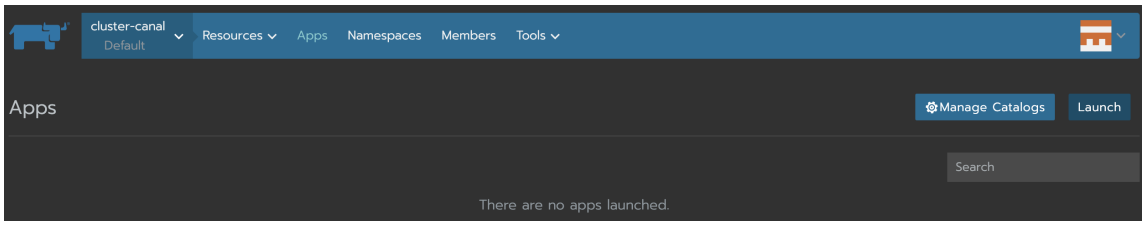


3. In the Add Cluster - Select Cluster Type page, choose the **Import an existing cluster** option.
4. Specify the **Cluster Name**.
5. Specify **Member Roles**, **Labels**, and **Annotations**.
6. Click **Create**.

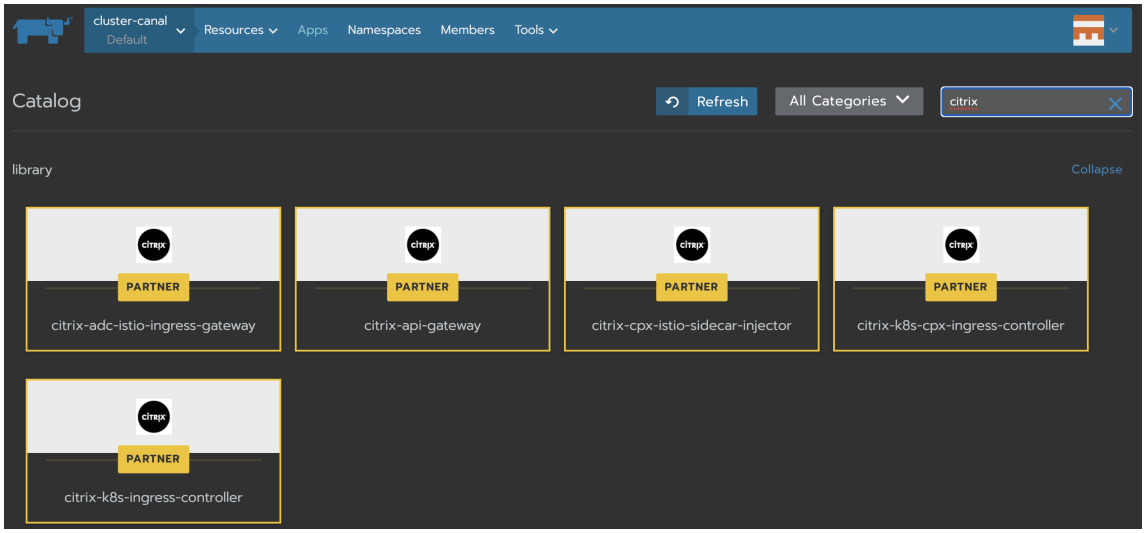
Deploy NetScaler API Gateway using the Rancher platform

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 NetScaler Ingress Controller image settings, and exporter settings).

   The mandatory fields include:

   - **Namespace:** Specify the namespace where you want to create the NetScaler 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 NetScaler license.
   - **Login File Name:** Specify the name of the Kubernetes secret. The secret file is used for the NetScaler login.
   - **NetScaler IP:** It is the NSIP or SNIP of the NetScaler 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

December 31, 2023

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 Gateway. 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.


```
swagger: "2.0"
info:
  version: "1.0.0"
host: "petstore.swagger.io"
basePath: "/v2"
tags:
- name: "pet"
  description: "Everything about
schemes:
- "https"
paths:
  /pet:
    post:
      tags:
      - "pet"
      parameters:
      - in: "body"
        name: "body"
      responses:
        "405":
          description: "Invalid
      security:
      - petstore_auth:
        - "write:pets"
        - "read:pets"
```

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.


```
1 apiVersion: citrix.com/v1
2 kind: rewritepolicy
3 metadata:
4   name: prefixurl
5 spec:
6   rewrite-policies:
7     - servicenames: []
8       rewrite-policy:
9         operation: replace
10        target: http.req.url
11        modify: '"newprefixurl/"+http.req.url'
12        comment: 'Some Prefix before a URL'
13        direction: REQUEST
14        criteria: 'http.req.url.equals_any("ur
15
16  patset:
17    - name: urls
18      values: []
```

API Gateway deployment CRD

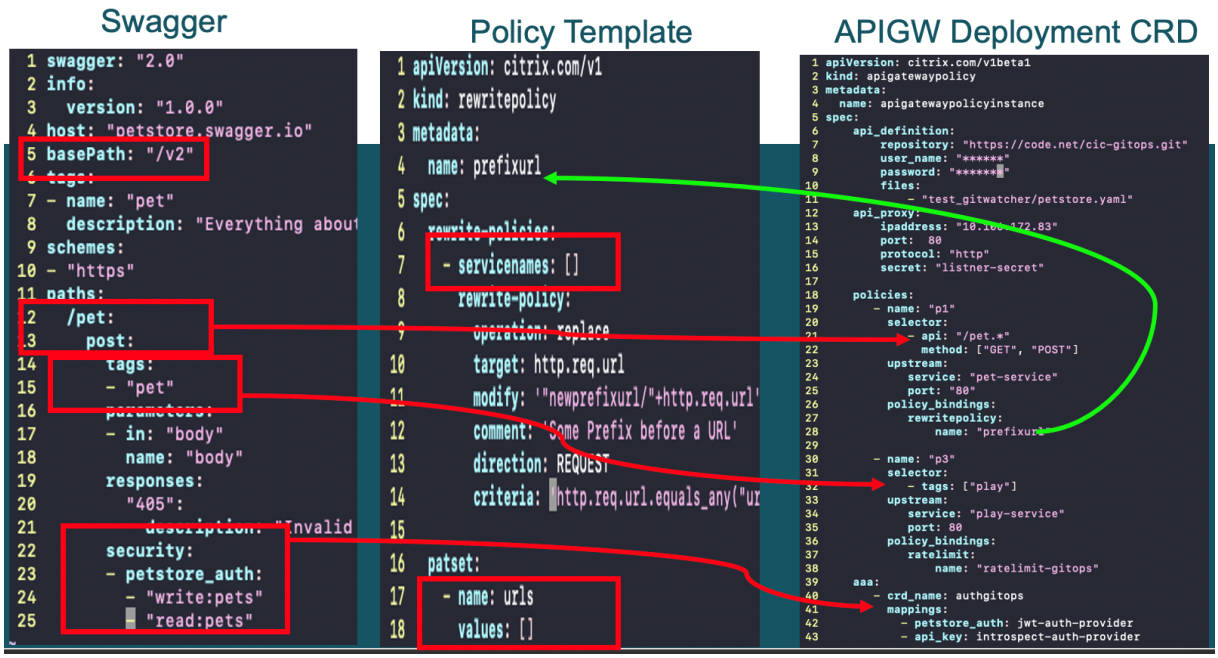
API Gateway deployment CRD binds the API specification document with policy templates. This CRD enables mapping of API resources with upstream services and API gateway policies related to routing and security. The API Gateway deployment CRD is maintained by the operations team with the data received 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) documents. 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
- 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 defenition: Git repository access details

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

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

Field	Description
`ip_address`	Specifies the IP address of the end point (VIP).
`port`	Specifies the endpoint port.
`protocol`	Specifies the protocol (HTTP/HTTPs).
`secret`	Specifies the SSL certificate secret for the endpoint configuration.

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.

Section	Sub section	Field	Sub field	Description
Policies				Specifies the policy and upstream mapping.
		Name		Specifies the name of the policy. It is unique in a CRD instance.
	Selector			A list of filters for selecting the API resources.
		API		Specifies the *Regexp* pattern for the API selection. All the APIs that match with this pattern are selected for applying policies from this block.

Section	Sub section	Field	Sub field	Description
		method		A list of HTTP verbs, if the API resource verb matches with ANY in the list, it is selected.
		Tags		A list of tags to match with an API. These tags are matched with *tags* in the API specification document. You can use either *ragexp* based path patterns or tags to match a policy.
	Upstream.			Specifies the upstream for the selected policy.
		Service		Specifies the back-end service name.
		Port		Specifies the back-end service port.
	Policy-binding			Specifies the policy list to be applied on the selected API.
			Type of the policy template	Specifies the exact type of policy. Supported types are WAF, rewrite policy, and rate limit.

Section	Sub section	Field	Sub field	Description
			Name	Specifies the name of the policy template.

AAA mappings

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

Section	Sub section	Field	Sub field	Description
aaa				Authentication, authorization, and auditing policy section mappings.
		Crd_name		Specifies the name of the authentication CRD template.
	Mappings			Mapping API specification security policy references with the appropriate sections in the authentication CRD template. **Note**: If the API specification refer to string matches with the policy section name in the CRD template, explicit mapping is not required.

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:

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

The following is an example API Gateway CRD configuration:

```
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: "listner-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
        mappings:
          - petstore_auth: jwt-auth-provider
          - api_key: introspect-auth-provider
<!--NeedCopy-->
```

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`

GSLB overview and deployment topologies

April 18, 2024

Overview

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

NetScaler GSLB controller configures NetScaler (GSLB device) to load balance services among geographically distributed locations. GSLB solution ensures better performance and reliability for your Kubernetes services that are exposed using ingress or service type LoadBalancer. In the GSLB topology, a GSLB device is deployed in each region; one of the GSLB devices acts as the primary ADC and

others act as the secondary ADCs. The GSLB primary ADC is configured by the GSLB controller deployed in each cluster deployed across sites. This GSLB device load balances services deployed in multiple clusters across sites.

For more information about GSLB, see Global Server Load Balancing.

> **Note:**
>
> The NetScaler GSLB controller image is the same as that of NetScaler Ingress Controller.

Deployment topologies

The components of GSLB deployment topology are described here:

- **GSLB device**: NetScaler MPX or NetScaler VPX is used as a global server load balancing (GSLB) device. A GSLB device is configured for each data center. In each GSLB device, one site is configured as a local site representing the local data center. The other sites are configured as remote sites. NetScaler MPX or NetScaler VPX used as the GSLB device can also be used as the ingress device with NetScaler Ingress Controller.
- **Ingress load balancer**: NetScaler CPX or any third-party application is deployed as the ingress load balancer in each Kubernetes cluster.
- **Ingress controller**: The ingress controller can be a NetScaler Ingress Controller or any third-party ingress controller.
- **GSLB controller**: Each cluster in the deployment runs a GSLB controller instance. Each GSLB controller configures the GSLB primary ADC for the applications deployed in its respective cluster. The global server load balancing (GSLB) configuration synchronization option is used to copy the GSLB configuration on the primary site to all the GSLB sites in the GSLB setup. The NetScaler on which you configure GSLB synchronization is referred as the primary site and the sites to which the configuration is copied are referred as the secondary sites.

The following deployment diagrams show sample topologies for NetScaler GSLB controller. Each sample topology contains two data centers (sites) in different regions and each data center contains a Kubernetes cluster.

Let's consider two sample deployment topologies based on the type of ingress controller used in the GSLB sites:

- NetScaler Ingress Controller in both GSLB sites
- Any third-party ingress controller in both GSLB sites

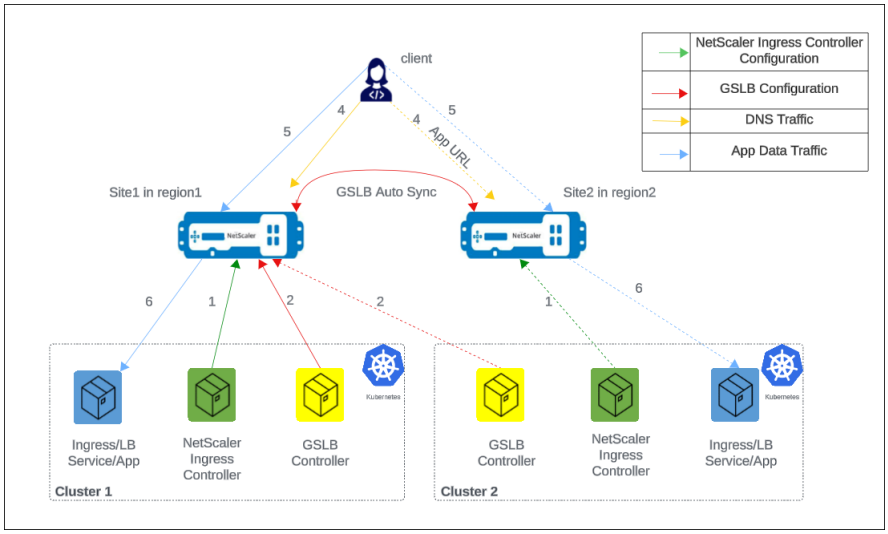
> **Note:**

> GSLB controller deployment is also supported with NetScaler Ingress Controller in one GSLB site and a third-party ingress controller in another GSLB site.

NetScaler Ingress Controller in both GSLB sites

> **Note:**
>
> NetScaler MPX or NetScaler VPX used for GSLB and ingress or service type LB can be the same or different. In the following example, the same NetScaler is used for GSLB and ingress or service type LB.

The following diagram explains the deployment topology for NetScaler GSLB controller in presence of NetScaler Ingress Controller.

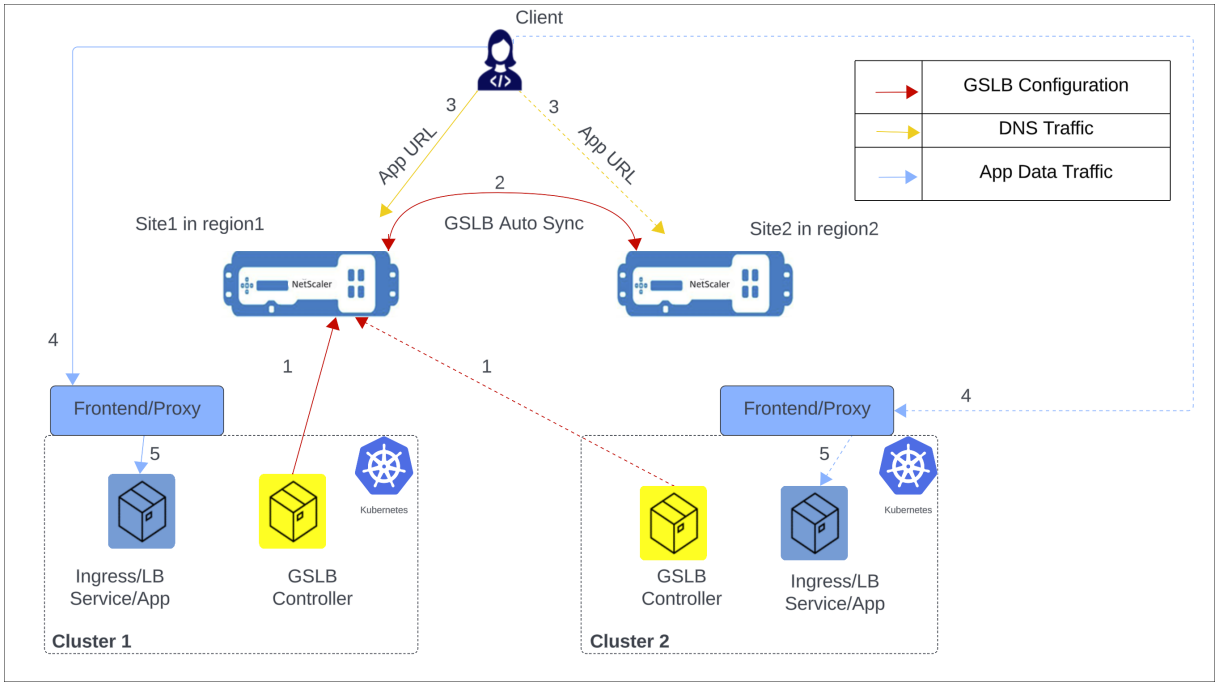


The numbers in the following steps map to the numbers in the earlier diagram.

1. In each cluster, NetScaler Ingress Controller configures NetScaler either using `ingress` or using the `service of type LoadBalancer` configuration.
2. In each cluster, NetScaler GSLB controller configures the GSLB device in the primary site with the GSLB configuration.
3. GSLB configuration synchronizes automatically between the GSLB devices in different GSLB sites.
4. A DNS query for application hostname or FQDN is sent to the GSLB virtual server configured on NetScaler. The DNS resolution on the GSLB virtual server resolves to an IP address on any one of the clusters based on the configured global traffic policy (GTP).
5. Based on the DNS resolution, data traffic lands on either the Ingress front-end IP address or service type LB IP address of one of the clusters.
6. The required application is accessed through the GSLB device.

Any third-party ingress controller in both GSLB sites

The following diagram explains the deployment topology for NetScaler GSLB controller in the presence of any third-party ingress controller.



The following items explain the previous diagram:

1. NetScaler GSLB controller configures the GSLB primary ADC in the primary site with the GSLB configuration.
2. GSLB configuration synchronizes automatically between the GSLB devices in different GSLB sites.
3. A DNS query for application hostname or FQDN is sent to the GSLB virtual server configured on NetScaler. The DNS resolution on the GSLB virtual server resolves to an IP address on any one of the clusters based on the configured global traffic policy (GTP).
4. Based on the DNS resolution, data traffic lands on either the ingress front-end IP address or service type LB front-end IP address of one of the clusters.
5. The required application is accessed through proxy.

GSLB methods and supported deployment types

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 NetScaler GSLB controller for applications deployed in distributed Kubernetes clusters

The following CRDs are introduced to support NetScaler 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 NetScaler including deployment type (canary, failover), GSLB domain, health monitor for the ingress, and service type.

The GTP CRD spec is available here.

> **Note:**
>
> GTP CRD is the same across all the clusters for a given domain.

The following table explains the GTP CRD attributes.

Field	Description
`ipType`	Specifies the DNS record type as A or AAAA. Currently, only `A` record type is supported
`serviceType`	Specifies the protocol to which GSLB support is applied.
`host`	Specifies the domain for which GSLB support is applied.
`trafficPolicy`	Specifies the traffic distribution policy supported in a GSLB deployment.
`sourceIpPersistenceId`	Specifies the unique source IP persistence ID. This attribute enables persistence based on the source IP address for the inbound packets. The `sourceIpPersistenceId` attribute should be a multiple of 100 and should be unique.
`secLbMethod`	Specifies the traffic distribution policy supported among clusters under a group in local-first, canary, or failover.
`destination`	Specifies the Ingress or LoadBalancer service endpoint in each cluster. The destination name should match with the name of GSE.
`weight`	Specifies the proportion of traffic to be distributed across clusters. For canary deployment, the proportion is specified as percentage.
`CIDR`	Specifies the CIDR to be used in local-first to determine the scope of the locality.
`primary`	Specifies whether the destination is a primary cluster or a backup cluster in the failover deployment.

Field	Description
`monType`	Specifies the type of probe to determine the health of the GSLB endpoint. When the monitor type is HTTPS, SNI is enabled by default during the TLS handshake.
`uri`	Specifies the path to be probed for the health of the GSLB endpoint for HTTP and HTTPS protocols.
`respCode`	Specifies the response code expected to mark the GSLB endpoint as healthy for HTTP and HTTPS protocols.
`customHeader`	Specifies the custom header that you want to add to the GSLB-endpoint monitoring traffic.

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 NetScaler Ingress Controller GitHub repo at: gse-crd.yaml.

The following table explains the **GSE CRD** attributes.

Field	Description
ipv4address	Local cluster ingress ipv4 address or service of type `LoadBalancer` endpoint ipv4 address
domainName	Local cluster ingress domain name or service of type `LoadBalancer` domain name
monitorPort	Listening port of local cluster ingress or Listening port of service of type `LoadBalancer`

> **Notes:**
>
> - GSE CRD is different for each cluster.
> - For GSE CRD auto generation with ingress, the host name must match 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.
> - For a service of type `LoadBalancer`, the GSE CRD is auto generated if the service is re-

> ferred in the GTP CRD instance and the `status-loadbalancer-ip/hostname` field is already populated.

Deploy NetScaler GSLB controller

April 25, 2024

The following steps describe how to deploy a GSLB controller in a cluster.

> **Note:**
>
> Repeat the steps 1 through 5 to deploy a GSLB controller in other clusters.

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

```
kubectl create secret generic secret-1  --from-literal=username=<
    username for gslb device1>  --from-literal=password=<password
    for gslb device1>
<!--NeedCopy-->
```

```
kubectl create secret generic secret-2  --from-literal=username=<
    username for gslb device2>  --from-literal=password=<password
    for gslb device2>
<!--NeedCopy-->
```

> **Note:**
>
> These secrets are provided as parameters while installing GSLB controller using `helm install` command for the respective sites. The `username` and `password` in the command specifies the credentials of a NetScaler GSLB device user. For information about creating a system user account on NetScaler, see Create system user account for NetScaler Ingress Controller in NetScaler
>
> .

2. You need to manually provision the GSLB sites, configure the ADNS service, and enable management access on each GSLB device using the following commands:

   - `add ip <site-ip-address> 255.255.255.0 -mgmtAccess ENABLED`
   - `add gslbsite site1 <sitedata[0].siteIp> -publicIP`
   - `add gslbsite site2 <sitedata[1].siteIp> -publicIP`
   - `add service adns_svc <site-ip-address> ADNS 53`

   For information about `sitedata[0].siteIp` and `sitedata[1].siteIp`, see the table in step 4.

`site-ip-address` is the GSLB device IP address in the site where the GSLB controller is deployed.

3. Add the NetScaler Helm chart repository to your local Helm registry using the following command:

```
helm repo add netscaler https://netscaler.github.io/netscaler-helm-charts/
<!--NeedCopy-->
```

4. Install the GSLB controller on a cluster using the Helm chart by running the following command: `helm install gslb-release netscaler/netscaler-gslb-controller -f values.yaml --set crds.install=true`

> **Notes:**
>
> - If CRDs are already installed, omit `--set crds.install=true` in the above installation command.
> - The chart installs the recommended RBAC roles and role bindings by default.

Example `values.yaml` file:

```
      license:
        accept: yes

      localRegion: "east"
      localCluster: "cluster1"

      entityPrefix: "k8s"

      sitedata:
      - siteName: "site1"
        siteIp: "x.x.x.x"
        siteMask: "y.y.y.y"
        sitePublicIp: "z.z.z.z"
        secretName: "secret-1"
        siteRegion: "east"
      - siteName: "site2"
        siteIp: "x.x.x.x"
        siteMask: "y.y.y.y"
        sitePublicIp: "z.z.z.z"
        secretName: "secret-2"
        siteRegion: "west"
<!--NeedCopy-->
```

Specify the following parameters in the values.yml file.

Parameter	Description
LocalRegion	Local region where the GSLB controller is deployed.
LocalCluster	The name of the cluster in which the GSLB controller is deployed. This value is unique for each kubernetes cluster.
sitedata[0].siteName	The name of the first GSLB site configured in the GSLB device.
sitedata[0].siteIp	IP address for the first GSLB site. Add the IP address of the NetScaler in site1 as *sitedata[0].siteIp*.
sitedata[0].siteMask	The netmask of the first GSLB site IP address.
sitedata[0].sitePublicIp	The public IP address of the first GSLB Site.
sitedata[0].secretName	The name of the secret that contains the login credentials of the first GSLB site.
sitedata[0].siteRegion	The region of the first site.
sitedata[1].siteName	The name of the second GSLB site configured in the GSLB device.
sitedata[1].siteIp	IP address for the second GSLB site. Add the IP address of the NetScaler in site2 as *sitedata[0].siteIp*
sitedata[1].siteMask	The netmask of the second GSLB site IP address.
sitedata[1].sitePublicIp	The public IP address of the second GSLB site.
sitedata[1].secretName	The secret containing the login credentials of the second site.
sitedata[1].siteRegion	The region of the second site.

> **Note:**
>
> The order of the GSLB site information should be the same in all the clusters. The first site in the order is considered as the primary site for pushing the configuration. When that primary site goes down, the next site in the list becomes the new primary.
> For example, if the order of sites is `site1` followed by `site2` in cluster1, all other clusters should have the same order.

5. Verify the installation using the following command: `kubectl get pods -l app=gslb-release-netscaler-gslb-controller`.

After the successful installation of the GSLB controller on each cluster, the ADNS service will be configured and the management access will be enabled on both the GSLB devices.

Synchronize GSLB configuration

Run the following commands in the same order on the primary NetScaler GSLB device to enable automatic synchronization of the GSLB configuration between the primary and secondary GSLB devices.

```
set gslb parameter -automaticconfigsync enable
sync gslb config -debug
<!--NeedCopy-->
```

Examples for global traffic policy (GTP) deployments

The GTP configuration should be the same across all the clusters.

In the following examples, an application app1 is deployed in the default namespace of the cluster1 of the east region and default namespace of cluster2 of the west region.

> **Note:**
>
> The destination information in the GTP yaml should be in the format `servicename.namespace.region.cluster`, where the servicename and namespace corresponds to the Kubernetes object of kind Service and its namespace.
>
> You can specify the load balancing method for canary and failover deployments.

Example 1: Round robin deployment

Use this deployment to distribute the traffic evenly across the clusters. The following example configures a GTP for round robin deployment.

Use the `weight` field to direct more client requests to a specific cluster within a group. Specify a custom header that you want to add to the GSLB-endpoint monitoring traffic by adding the `customHeader` argument under the `monitor` parameter.

```
kubectl apply -f - <<EOF
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'
          weight: 2
        - destination: 'app1.default.west.cluster2'
          weight: 5
        monitor:
        - monType: http
          uri: ''
          customHeader: "Host: <custom hostname>\r\n x-b3-traceid:
              afc38bae00096a96\r\n\r\n"
          respCode: 200
  EOF
<!--NeedCopy-->
```

Example 2: Failover deployment

Use this policy to configure the application in active-passive mode. In a failover deployment, the application is deployed in multiple clusters. Failover is achieved between the application instances (target destinations) in different clusters based on the weight assigned to those target destinations in the GTP policy.

The following example shows a sample GTP configuration for failover. Using the primary field, you can specify which target destination is active and which target destination is passive. The default value for the primary field is `True` indicating that the target destination is active. Bind a monitor to the endpoints in each cluster to probe their health.

```
  kubectl apply -f - <<EOF
  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:
```



```
21          - monType: http
22            uri: ''
23            respCode: 200
24    EOF
25  <!--NeedCopy-->
```

Example 3: RTT deployment

Use this policy to monitor the real-time status of the network and dynamically direct the client request to the target destination with the lowest RTT value.

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

```
1    kubectl apply -f - <<EOF
2    apiVersion: "citrix.com/v1beta1"
3    kind: globaltrafficpolicy
4    metadata:
5      name: gtp1
6      namespace: default
7    spec:
8      serviceType: 'HTTP'
9      hosts:
10     - host: 'app1.com'
11       policy:
12         trafficPolicy: 'RTT'
13         targets:
14         - destination: 'app1.default.east.cluster1'
15         - destination: 'app1.default.west.cluster2'
16         monitor:
17         - monType: tcp
18   EOF
19 <!--NeedCopy-->
```

Example 4: Canary deployment

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 describes a sample global traffic policy with Canary deployment, where a new version of an application needs to be rolled out before deploying in production.

In this example, 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.

```
kubectl apply -f - <<EOF
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
EOF
<!--NeedCopy-->
```

Example 5: Static proximity

Use this policy to select the service that best matches the proximity criteria.

Following GTP is an example for static proximity deployment.

> **Note:**
>
> For static proximity, you need to apply the location database manually on all the GSLB devices: `add locationfile /var/netscaler/inbuilt_db/Citrix_Netscaler_InBuilt_GeoIP_DB`.

```
kubectl apply -f - <<EOF
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'
      monitor:
      - monType: http
        uri: ''
        respCode: 200
EOF
  <!--NeedCopy-->
```

Example 6: source IP persistence

The following traffic policy is an example to enable source IP persistence by providing the parameter `sourceIpPersistenceId`.

```
kubectl apply -f - <<EOF
apiVersion: "citrix.com/v1beta1"
kind: globaltrafficpolicy
metadata
  name: gtp1
  namespace: default
spec:
  serviceType: 'HTTP'
  hosts:
  - host: 'app1.com'
    policy:
      trafficPolicy: 'ROUNDROBIN'
      sourceIpPersistenceId: 300
      targets:
      - destination: 'app1.default.east.cluster1'
        weight: 2
      - destination: 'app1.default.west.cluster2'
        weight: 5
      monitor:
      - monType: tcp
        uri: ''
        respCode: 200
EOF
<!--NeedCopy-->
```

Example for global service entry (GSE)

GSE configuration is applied in a specific cluster based on the cluster endpoint information. The GSE name must be the same as the target destination name in the global traffic policy.

> **Note:**
>
> Creating GSE is optional. If GSE is not created, NetScaler Ingress Controller looks for matching ingress with host matching `<svcname>.<namespace>.<region>.<cluster>` format.

For a global traffic policy mentioned in the earlier section, here is the global service entry for cluster1. In this example, the global service entry name `app1.default.east.cluster1` is one of the target destination names in the global traffic policy created.

```
kubectl apply -f - <<EOF
apiVersion: "citrix.com/v1beta1"
kind: globalserviceentry
metadata:
  name: 'app1.default.east.cluster1'
  namespace: default
spec:
  endpoint:
    ipv4address: 10.102.217.70
    monitorPort: 33036
EOF
  <!--NeedCopy-->
```

Example: Ingress service or Service type LB

Example for Ingress service

The following sample YAML deploys Ingress service for GSE defined above.

```
kubectl apply -f - <<EOF
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app1-ingress
  namespace: default
  annotations:
    kubernetes.io/ingress.class: citrix
    ingress.citrix.com/frontend-ip: 10.102.217.70
spec:
  rules:
    - host: app1.com
      http:
        paths:
          - backend:
              service:
                name: app1
                port:
                  number: 80
            path: /
```



```
22            pathType: Prefix
23
24  ---
25  apiVersion: apps/v1
26  kind: Deployment
27  metadata:
28    name: app1
29    namespace: default
30    labels:
31      name: app1
32      app: app1
33      appHostname: app1.com
34  spec:
35    selector:
36      matchLabels:
37        app: app1
38    replicas: 2
39    template:
40      metadata:
41        labels:
42          name: app1
43          app: app1
44      spec:
45        containers:
46          - name: app1
47            image: <application image>
48            ports:
49              - name: http-80
50                containerPort: 80
51              - name: https-443
52                containerPort: 443
53
54  ---
55  apiVersion: v1
56  kind: Service
57  metadata:
58    name: app1
59    namespace: default
60    labels:
61      app: app1
62    annotations:
63      service.citrix.com/class: citrix
64  spec:
65    ports:
66      - name: http-80
67        port: 80
68        targetPort: 80
69      - name: https-443
70        port: 443
71        targetPort: 443
72    selector:
73      name: app1
74  EOF
```



```
<!--NeedCopy-->
```

Example for service type LB

The following sample YAML deploys LB service for GSE defined above.

```
kubectl apply -f - <<EOF
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app1
  namespace: default
  labels:
    name: app1
    app: app1
    appHostname: app1.com
spec:
  selector:
    matchLabels:
      app: app1
  replicas: 2
  template:
    metadata:
      labels:
        name: app1
        app: app1
    spec:
      containers:
        - name: app1
          image: <application image>
          ports:
            - name: http-80
              containerPort: 80
            - name: https-443
              containerPort: 443

---
---
apiVersion: v1
kind: Service
metadata:
  name: app1
  namespace: default
  annotations:
    service.citrix.com/class: citrix
    service.citrix.com/frontend-ip: 10.102.217.70
spec:
  type: LoadBalancer
  ports:
    - name: port-8080
      port: 443
```



```
      targetPort: 80
  selector:
    app: app1
status:
  loadBalancer:
    ingress:
      - ip: 10.102.217.70
EOF
<!--NeedCopy-->
```

NetScaler GSLB controller for single site

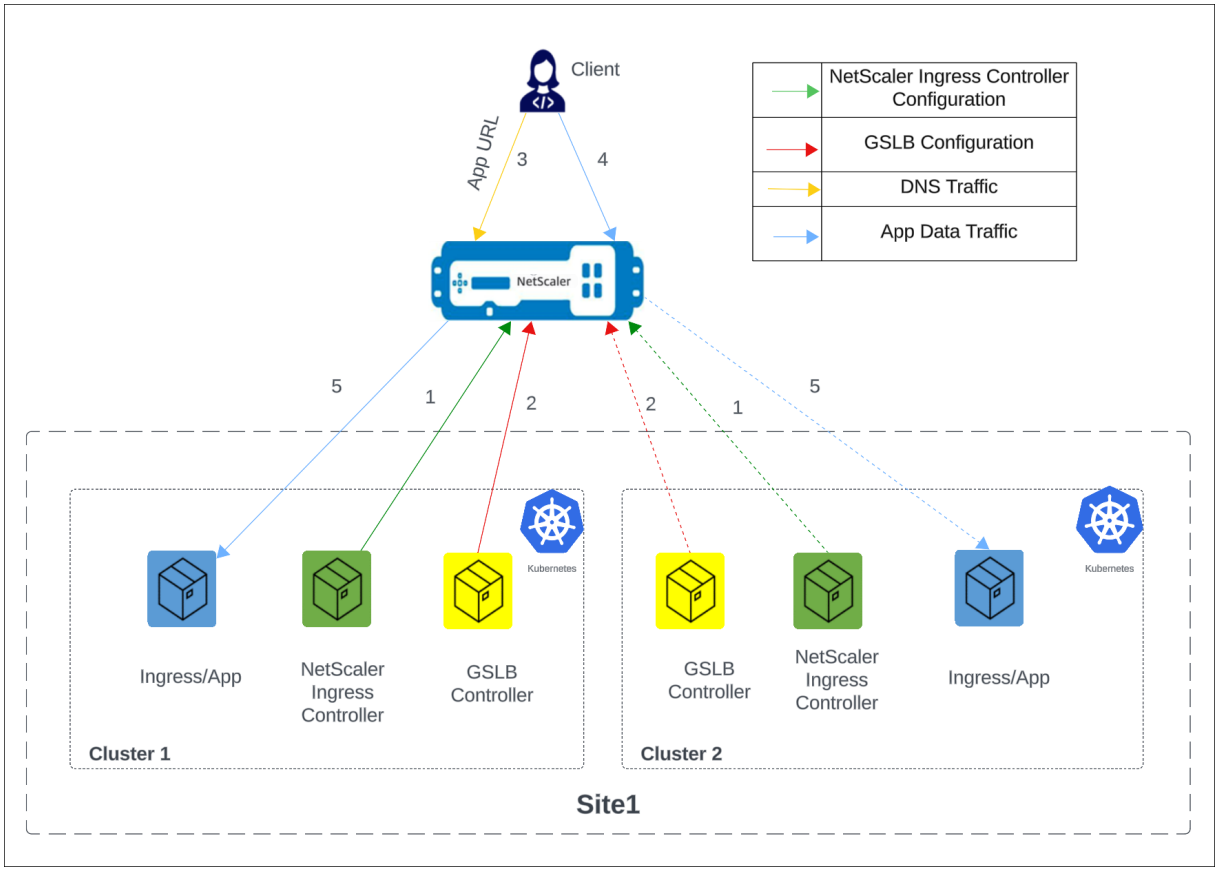
March 21, 2024

Overview

For ensuring high availability, proximity-based load balancing, and scalability, you need to deploy an application in multiple Kubernetes clusters. GSLB solution ensures better performance and reliability for your Kubernetes services that are exposed using Ingress. NetScaler GSLB controller configures NetScaler (GSLB device) to load balance services among geographically distributed locations. In a single-site GSLB solution, a GSLB device in a data center is configured by the GSLB controller deployed in each Kubernetes cluster of a data center. This GSLB device load balances services deployed in multiple clusters of the data center.

The following diagram describes the deployment topology for NetScaler GSLB controller in a data center with two Kubernetes clusters and a single GSLB site.

> **Note:**
>
> NetScaler (MPX or VPX) used for GSLB and Ingress can be the same or different. In the following diagram, the same NetScaler is used for GSLB and Ingress.



The numbers in the following steps map to the numbers in the earlier diagram.

1. In each cluster, the NetScaler Ingress Controller configures NetScaler using `Ingress`.
2. In each cluster, the NetScaler GSLB controller configures the GSLB device with the GSLB configuration.
3. A DNS query for the application URL is sent to the GSLB virtual server configured on NetScaler. The DNS resolution on the GSLB virtual server resolves to an IP address on any one of the clusters based on the configured global traffic policy (GTP).
4. Based on the DNS resolution, data traffic lands on either the Ingress front-end IP address or the content switching virtual server IP address of one of the clusters.
5. The required application is accessed through the GSLB device.

Deploy NetScaler GSLB controller

The following steps describe how to deploy a GSLB controller in a cluster.

> **Note:**
>
> Repeat the steps to deploy a GSLB controller in other clusters.

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

```
kubectl create secret generic secret--from-literal=username=<
    username for gslb device>--from-literal=password=<password for
    gslb device>
<!--NeedCopy-->
```

> **Note:**
>
> This secret is provided as a parameter in the GSLB controller `helm install` command for the respective sites. The `username` and `password` in the command specify the credentials of a NetScaler (GSLB device) user.

2. Add the NetScaler Helm chart repository to your local Helm registry using the following command:

```
helm repo add netscaler https://netscaler.github.io/netscaler-helm
    -charts/
<!--NeedCopy-->
```

3. Install GSLB controller using the Helm chart by running the following command: `helm install my-release netscaler/netscaler-gslb-controller -f values.yaml`

> **Note:**
>
> The chart installs the recommended RBAC roles and role bindings by default.

Example `values.yaml` file:

```yaml
license:
accept: yes

localRegion: "east"
localCluster: "cluster 1"

entityPrefix: "k8s"

sitedata:
- siteName: "site 1"
  siteIp: "x.x.x.x"
  siteMask:
  sitePublicIp:
  secretName: "secret"
  siteRegion: "east"
```



```
    nsIP: "x.x.x.x"
    crds.install: true
    adcCredentialSecret: <Secret-for-NetScaler-credentials>
  <!--NeedCopy-->
```

Specify the following parameters in the YAML file.

Parameter	Description
LocalRegion	Local region where the GSLB controller is deployed. This value is the same for GSLB controller deployment across all the clusters.
LocalCluster	The name of the cluster in which the GSLB controller is deployed. This value is unique for each Kubernetes cluster.
sitedata[0].siteName	The name of the GSLB site.
sitedata[0].siteIp	IP address for the GSLB site. Add the IP address of the NetScaler in site 1 as *sitedata[0].siteIp*.
sitedata[0].siteMask	The netmask of the GSLB site IP address.
sitedata[0].sitePublicIp	The site public IP address of the GSLB site.
sitedata[0].secretName	The name of the secret that contains the login credentials of the GSLB site.
sitedata[0].siteRegion	The region of the GSLB site.
NSIP	The SNIP (subnet IP address) of the GSLB device. Add the *sitedata[0].siteIp* as SNIP on NetScaler.
crds.install: true	This parameter installs the required GTP and GSE CRDs on the GSLB device.
adcCredentialSecret	The Kubernetes secret containing the login credentials for the NetScaler VPX or MPX.

If you don't specify `nsIP`, `adcCredentialSecret` parameters in the YAML file, you need to manually provision the GSLB sites, configure the ADNS service, and enable management access on each GSLB device using the following commands:

- `add ip <site-ip-address> 255.255.255.0 -mgmtAccess ENABLED`
- `add gslbsite site 1 <site1-ip-address> -publicIP`
- `add service adns_svc <site-ip-address> ADNS 53`

After the successful installation of a GSLB controller on each cluster, GSLB site and ADNS service are configured and management access is enabled on the GSLB site IP address.

Global traffic policy examples

In the following examples, a stable application app1 is deployed in the default namespace of cluster 1 and cluster 2 in site 1.

> **Notes:**
>
> - Ensure that the GTP configuration is the same across all the clusters. For information on GTP CRD and allowed values, see GTP CRD.
> - The destination information in the GTP YAML should be in the format `servicename.namespace.region.cluster`, where the service name and namespace correspond to the Kubernetes object of type Service and its namespace, respectively.

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

Example 1: Round robin deployment

Use this deployment to distribute the traffic evenly across the clusters. The following example configures a GTP for round robin deployment.
You can use the `weight` field to direct more client requests to a specific cluster within a group.

```
kubectl apply -f - <<EOF
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'
        weight: 2
      - destination: 'app1.default.east.cluster2'
        weight: 5
      monitor:
      - monType: tcp
        uri: ''
        respCode: 200
EOF
<!--NeedCopy-->
```


Example 2: Failover deployment

Use this policy to configure the application in active-passive mode. In a failover deployment, the application is deployed in multiple clusters. Failover is achieved between the instances in target destinations based on the weight assigned to those target destinations in the GTP policy.

The following example shows a sample GTP configuration for failover. Using the `primary` field, you can specify which target destination is active and which target destination is passive. The default value for the `primary` field is `True` indicating that the target destination is active. Bind a monitor to the endpoints in each cluster to probe their health.

```
kubectl apply -f - <<EOF
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.east.cluster2'
        primary: false
        weight: 1
      monitor:
      - monType: http
        uri: ''
        respCode: 200
EOF
<!--NeedCopy-->
```

Example 3: RTT deployment

Use this policy to monitor the real-time status of the network and dynamically direct the client request to the target destination with the lowest RTT value.

The following example configures a GTP for RTT deployment.

```
kubectl apply -f - <<EOF
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.east.cluster2'
      monitor:
      - monType: tcp
EOF
<!--NeedCopy-->
```

Example 4: Canary deployment

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 describes 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 `cluster2`. A new version of the application is deployed in `cluster1`. Using the `weight` field, 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 stable, the new version can be rolled out to the other clusters.

```
kubectl apply -f - <<EOF
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.east.cluster2'
      monitor:
      - monType: http
```



```
        uri: ''
        respCode: 200
EOF
<!--NeedCopy-->
```

Example 5: Static proximity

Use this policy to select the service that best matches the proximity criteria. Following traffic policy is an example for static proximity deployment.

> **Note:**
>
> For static proximity, you must apply the location database manually: `add locationfile /var/netscaler/inbuilt_db/Citrix_Netscaler_InBuilt_GeoIP_DB_IPv4`.

```
kubectl apply -f - <<EOF
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.east.cluster2'
      monitor:
      - monType: http
        uri: ''
        respCode: 200
EOF
  <!--NeedCopy-->
```

Example 6: Source IP persistence

The following traffic policy is an example to enable source IP persistence by providing the parameter `sourceIpPersistenceId`.

```
kubectl apply -f - <<EOF
apiVersion: "citrix.com/v1beta1"
kind: globaltrafficpolicy
metadata
  name: gtp1
  namespace: default
```



```
spec:
  serviceType: 'HTTP'
  hosts:
  - host: 'app1.com'
    policy:
      trafficPolicy: 'ROUNDROBIN'
      sourceIpPersistenceId: 300
      targets:
      - destination: 'app1.default.east.cluster1'
        weight: 2
      - destination: 'app1.default.east.cluster2'
        weight: 5
      monitor:
      - monType: tcp
        uri: ''
        respCode: 200
EOF
<!--NeedCopy-->
```

Global service entry (GSE) examples

GSE configuration is applied in a specific cluster based on the cluster endpoint information. The GSE name must be the same as the target destination name in the global traffic policy.

> **Note:**
>
> Creating GSE is optional. If GSE is not created, NetScaler Ingress Controller looks for matching ingress with host matching `<svcname>.<namespace>.<region>.<cluster>` format.

For a global traffic policy mentioned in the earlier examples, the following YAML is the global service entry for cluster1. In this example, the global service entry name `app1.default.east.cluster1` is one of the target destination names in the global traffic policy.

```
kubectl apply -f - <<EOF
apiVersion: "citrix.com/v1beta1"
kind: globalserviceentry
metadata:
  name: 'app1.default.east.cluster1'
  namespace: default
spec:
  endpoint:
  ipv4address: 10.102.217.70
  monitorPort: 33036
EOF
  <!--NeedCopy-->
```

For a global traffic policy mentioned in the earlier examples, the following YAML is the global service entry for cluster2. In this example, the global service entry name `app1.default.east.cluster2` is one of the target destination names in the global traffic policy.


```
kubectl apply -f - <<EOF
apiVersion: "citrix.com/v1beta1"
kind: globalserviceentry
metadata:
  name: 'app1.default.east.cluster2'
  namespace: default
spec:
  endpoint:
  ipv4address: 10.102.217.70
  monitorPort: 33036
EOF
  <!--NeedCopy-->
```

Service Mesh lite

December 31, 2023

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.

NetScaler (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). NetScaler 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.

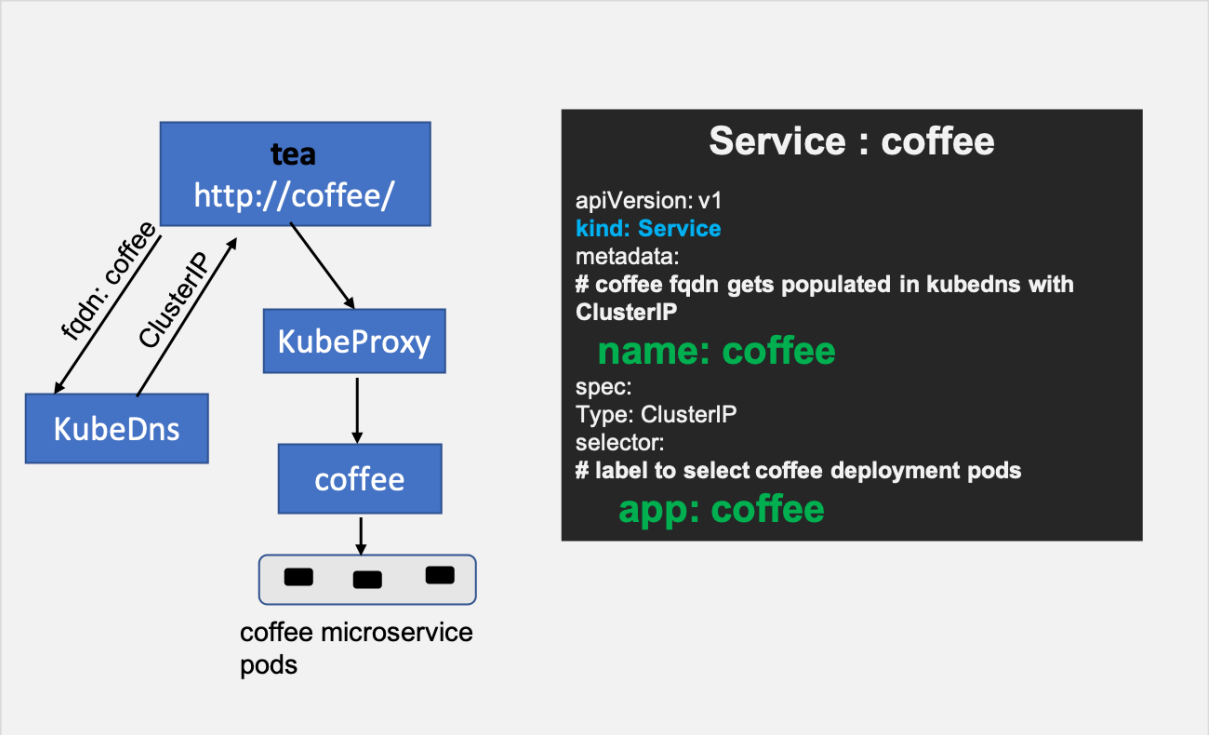
To configure east-west communication with NetScaler CPX in a Service mesh lite architecture, you must first understand how the kube-proxy is configured to manage east-west traffic.

East-west communication with kube-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 ab-

straction 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.



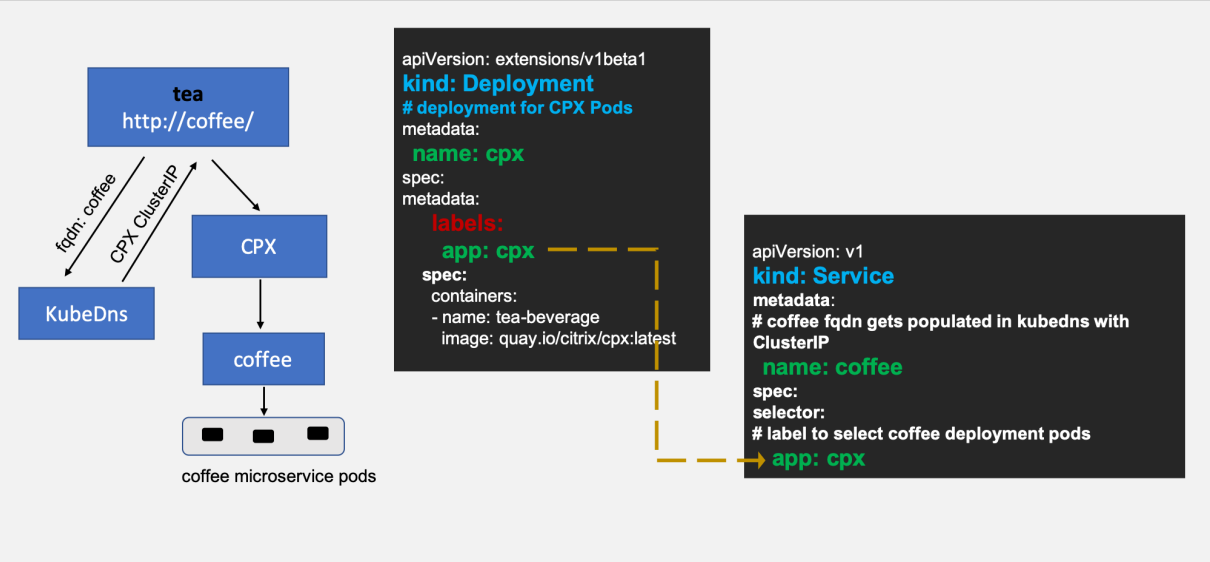
East-west communication with NetScaler CPX in Service Mesh Lite architecture

The goal is to insert the NetScaler 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 NetScaler CPX.

Step 1: Modify the coffee service definition to point to NetScaler CPX

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



> **Note:**
>
> If you are deploying service mesh lite to bring up the service graph in NetScaler ADM for observability, then you should add the label `citrix-adc: cpx` in all the services of your application which are pointing to the NetScaler 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 NetScaler CPX, you need to create one more service that represents coffee microservice deployment.

The following is a sample headless service resource:

```
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
<!--NeedCopy-->
```

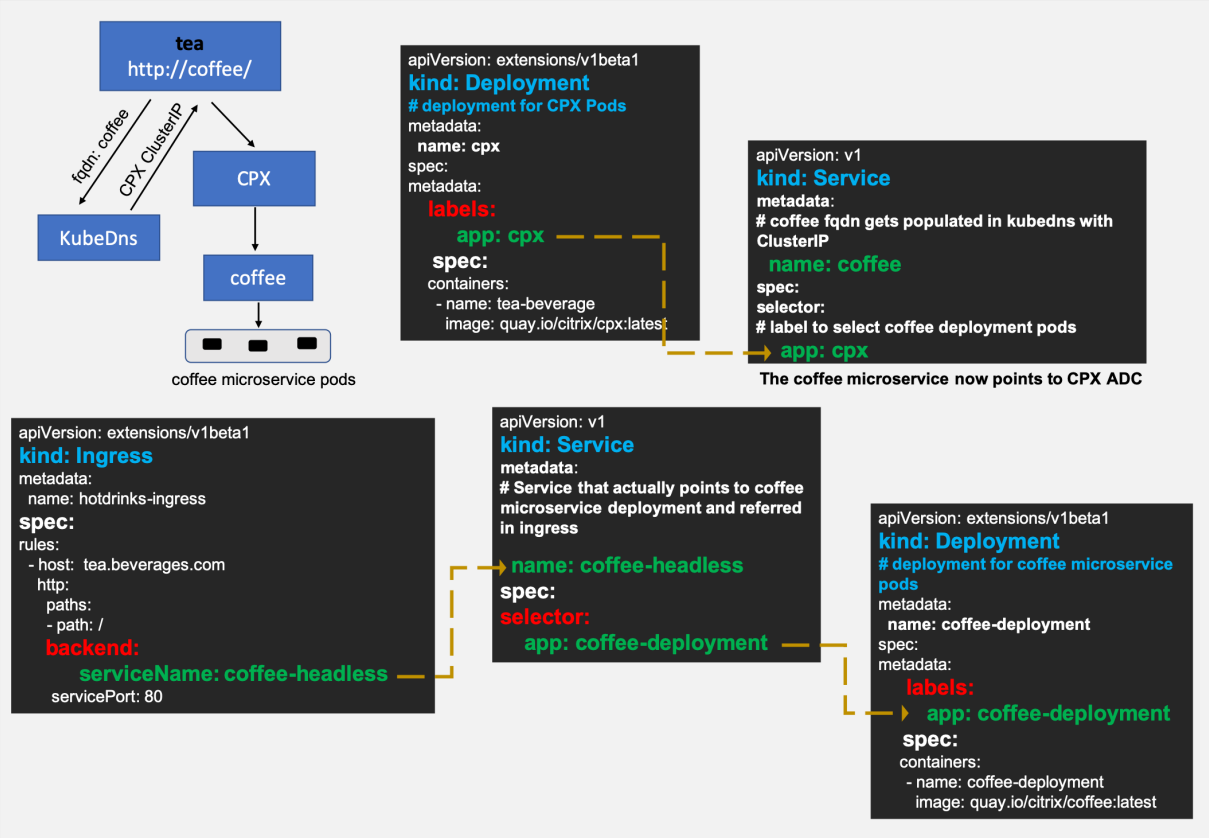

Step 3: Create an Ingress resource with rules for the `coffee-headless` service

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

The following is a sample Ingress resource:

```
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, NetScaler CPX can now load balance the east-west traffic. The following diagrams show how the NetScaler 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 NetScaler MPX or VPX in Service Mesh lite architecture

NetScaler 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 NetScaler 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 NetScaler 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.

```
kind: Service
apiVersion: v1
metadata:
name: coffee
spec:
type: ExternalName
externalName: myadc – instance1.us-east-1.mydomain.com
<!--NeedCopy-->
```

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 NetScaler MPX or VPX, you can create the following:

```
---
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
        name: "coffee"
<!--NeedCopy-->
```

Step 2: Create a headless service for microservice pods

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

Step 3: Create an Ingress resource

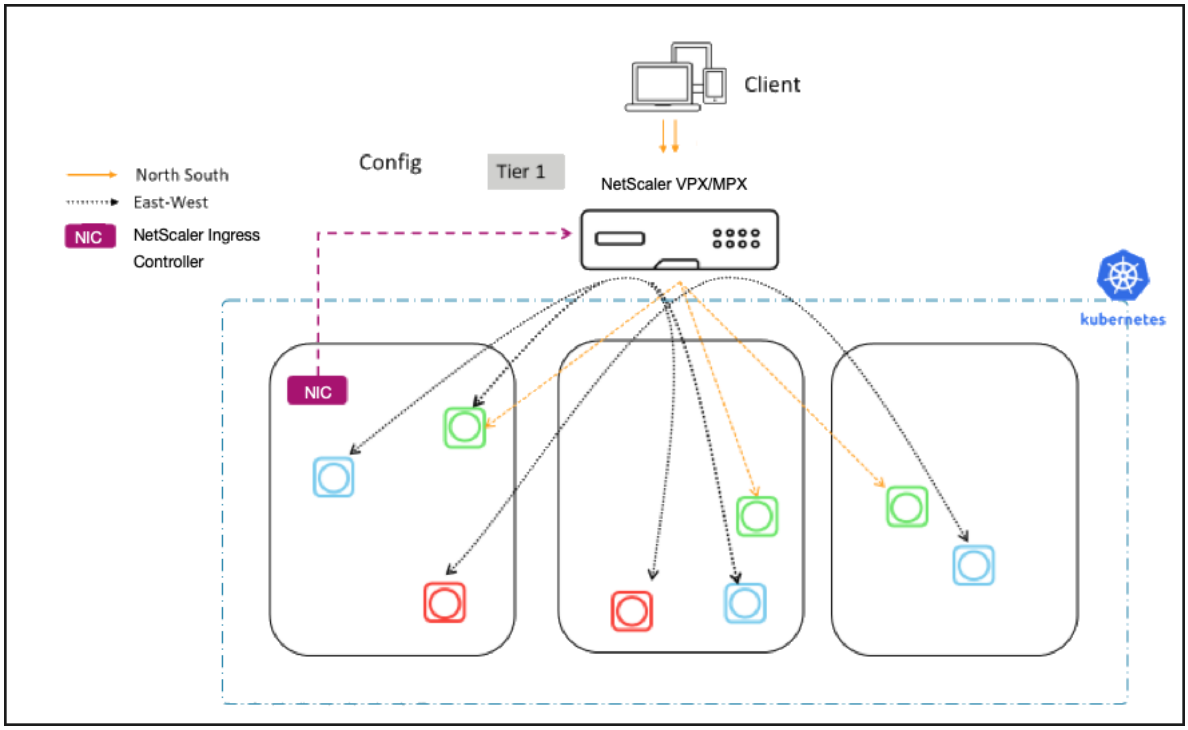
Create an Ingress resource using the `ingress.citrix.com/frontend-ip` annotation where the value matches the Ingress endpoint IP address in NetScaler MPX or VPX.

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

The following is a sample ingress resource:

```
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 NetScaler MPX can now load balance east-west traffic. The following diagram shows a NetScaler 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. NetScaler 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 NetScaler provided script.

Deploy the NetScaler Ingress Controller as an OpenShift router plug-in

December 31, 2023

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 NetScaler Ingress Controller can be deployed as a router plug-in in the OpenShift cluster to integrate with NetScalers deployed in your environment. The NetScaler Ingress Controller enables you to use the advanced load balancing and traffic management capabilities of NetScaler with your OpenShift cluster.

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

The NetScaler 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 NetScaler MPX or VPX appliance outside the OpenShift cluster or deploy NetScaler CPXs as pods inside the cluster. The NetScaler Ingress Controller integrates NetScalers with the OpenShift cluster and automatically configures NetScalers based on rules specified in routes.

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

- As a sidecar container alongside NetScaler CPX in the same pod: In this mode, the NetScaler Ingress Controller configures the NetScaler CPX.
- As a standalone pod in the OpenShift cluster: In this mode, you can control the NetScaler MPX or VPX appliance deployed outside the cluster.

For information on deploying the NetScaler Ingress Controller to control the OpenShift ingress, see the NetScaler Ingress Controller for Kubernetes.

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

Alternate Backend Support

OpenShift Alternate backends is now supported by NetScaler Ingress Controller.

NetScaler 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:

```
kind: Route
apiVersion: route.openshift.io/v1
metadata:
  name: r1
  labels:
    name: apache
  annotations:
    ingress.citrix.com/frontend-ip: "<Frontend-ip>"
spec:
  host: some.alternate-backends.com
  to:
    kind: Service
    name: apache-1
    weight: 30
  alternateBackends:
    - kind: Service
      name: apache-2
      weight: 20
    - kind: Service
      name: apache-3
      weight: 50
  port:
    targetPort: 80
  wildcardPolicy: None
<!--NeedCopy-->
```

For this route, 30 percent of the traffic is sent to the service apache-1 and 20 percent is sent to the service apache2 and 50 percent to the service apache-3 based on weights provided in the route manifest

Supported Citrix components on OpenShift

Citrix components	Versions
NetScaler Ingress Controller	Latest
NetScaler VPX	12.1 50.x and later
NetScaler CPX	13.0–36.28

> **Note:**
>
> CRDs provided for the NetScaler Ingress Controller is not supported for OpenShift routes. You

> can use OpenShift ingress to use CRDs.

Deploy NetScaler CPX as a router within the OpenShift cluster

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

Before you begin: When you deploy NetScaler CPX as a router, port conflicts can arise with the default router in OpenShift. You should remove the default router in OpenShift before deploying NetScaler 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.

```
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.

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

Perform the following steps to deploy NetScaler CPX as a router with the NetScaler Ingress Controller as a sidecar.

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

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

3. Deploy the NetScaler Ingress Controller using the following command:

```
oc create -f cpx_cic_side_car.yaml
```

4. Verify if the NetScaler Ingress Controller is deployed successfully using the following command:

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

Deploy NetScaler MPX/VPX as a router outside the OpenShift cluster

In this deployment, the NetScaler Ingress Controller which runs as a stand-alone pod allows you to control the NetScaler MPX, or VPX appliance from the OpenShift cluster.

You can use the cic.yaml file for this deployment.

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

Prerequisites

- Determine the IP address needed by the NetScaler Ingress Controller to communicate with the NetScaler appliance. The IP address might be any one of the following depending on the type of NetScaler deployment:
  - NSIP (for standalone appliances): The management IP address of a standalone NetScaler appliance. For more information, see IP Addressing in NetScaler.
  - SNIP (for appliances in High Availability mode): The subnet IP address. For more information, see IP Addressing in NetScaler.
  - CLIP (for appliances in clustered mode): The cluster management IP (CLIP) address for a clustered NetScaler deployment. For more information, see IP addressing for a cluster.
- The user name and password of the NetScaler VPX or MPX appliance used as the Ingress device. If you are not using the default credentials, the NetScaler appliance must have a system user account with certain privileges so that the NetScaler Ingress Controller can configure the NetScaler MPX, or VPX appliance. To create a system user account on NetScaler, see Create a system user account for the NetScaler Ingress Controller in NetScaler.

  You can directly pass the user name and password as environment variables to the NetScaler 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=<
     username> --from-literal=password=<password>
  ```

Create a system user account for the NetScaler Ingress Controller in NetScaler The NetScaler Ingress Controller configures a NetScaler appliance (MPX or VPX) using a system user account of the NetScaler appliance. The system user account must have the permissions to configure the following tasks on the NetScaler:

- Add, Delete, or View Content Switching (CS) virtual server
- Configure CS policies and actions
- Configure Load Balancing (LB) virtual server
- Configure Service groups
- Cofigure 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 NetScaler 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 NetScaler appliance using the following steps:

   a) Use an SSH client, such as PuTTy, to open an SSH connection to the NetScaler 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 '^\(\?!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+\s+system\s+file)^\(\?!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+\s+system\s+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 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 NetScaler command line.

For configuring the command policy from the NetScaler configuration wizard (GUI), use the following command policy specification.

```
^\(?!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+\s+system\s+file)^\(?!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+\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 the NetScaler Ingress Controller as a pod in an OpenShift cluster

Perform the following steps to deploy the NetScaler Ingress Controller as a pod:

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:

Environment Variable	Mandatory or Optional	Description
NS_IP	Mandatory	The IP address of the NetScaler appliance. For more details, see Prerequisites.
NS_USER and NS_PASSWORD	Mandatory	The user name and password of the NetScaler VPX or MPX appliance used as the Ingress device. For more details, see Prerequisites.
EULA	Mandatory	The End User License Agreement. Specify the value as `Yes`.

Environment Variable	Mandatory or Optional	Description
NS_VIP	Optional	NetScaler Ingress Controller uses the IP address provided in this environment variable to configure a virtual IP address to the NetScaler 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.

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:

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

5. Verify if the NetScaler Ingress Controller is deployed successfully using the following command:

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

6. Configure static routes on NetScaler 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 NetScaler instance.

   c) Add the route on the NetScaler instance using the following command.

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

```
<B>Example:</b>

      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 NetScaler is as follows:

      add route 10.1.1.0 255.255.255.0 192.168.122.46
```

Example: Deploy the NetScaler Ingress Controller as a router plug-in in an OpenShift cluster

In this example, the NetScaler 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.

```
oc create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/openshift/manifest/apache.yaml
```

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:

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

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

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    name: apache-only-http
  name: apache-only-http
spec:
  replicas: 4
  selector:
    matchLabels:
      app: apache-only-http
  template:
```



```
14      metadata:
15        labels:
16          app: apache-only-http
17      spec:
18        containers:
19        - image: raghulc/apache-multiport-http:1.0.0
20          imagePullPolicy: IfNotPresent
21          name: apache-only-http
22          ports:
23          - containerPort: 80
24            protocol: TCP
25          - containerPort: 5080
26            protocol: TCP
27          - containerPort: 5081
28            protocol: TCP
29          - containerPort: 5082
30            protocol: TCP
31  ---
32  apiVersion: apps/v1
33  kind: Deployment
34  metadata:
35    labels:
36      name: apache-only-ssl
37    name: apache-only-ssl
38  spec:
39    replicas: 4
40    selector:
41      matchLabels:
42        app: apache-only-ssl
43    template:
44      metadata:
45        labels:
46          app: apache-only-ssl
47      spec:
48        containers:
49        - image: raghulc/apache-multiport-ssl:1.0.0
50          imagePullPolicy: IfNotPresent
51          name: apache-only-ssl
52          ports:
53          - containerPort: 443
54            protocol: TCP
55          - containerPort: 5443
56            protocol: TCP
57          - containerPort: 5444
58            protocol: TCP
59          - containerPort: 5445
60            protocol: TCP
61  ---
62  apiVersion: v1
63  kind: Service
64  metadata:
65    name: svc-apache-multi-http
66  spec:
```



```
  ports:
  - name: apache-http-6080
    port: 6080
    targetPort: 5080
  - name: apache-http-6081
    port: 6081
    targetPort: 5081
  - name: apache-http-6082
    port: 6082
    targetPort: 5082
  selector:
    app: apache-only-http
---
apiVersion: v1
kind: Service
metadata:
  name: svc-apache-multi-ssl
spec:
  ports:
  - name: apache-ssl-6443
    port: 6443
    targetPort: 5443
  - name: apache-ssl-6444
    port: 6444
    targetPort: 5444
  - name: apache-ssl-6445
    port: 6445
    targetPort: 5445
  selector:
    app: apache-only-ssl
---
```

2. Deploy the NetScaler Ingress Controller for NetScaler VPX as a stand-alone pod in the OpenShift cluster using the steps in Deploy the NetScaler Ingress Controller as a pod.

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

Note:
To deploy the NetScaler Ingress Controller with NetScaler CPX in the OpenShift cluster, perform the steps in Deploy the NetScaler Ingress Controller as a sidecar with NetScaler CPX.

3. Create an OpenShift route for exposing the application.

   - For creating an unsecured OpenShift route (unsecured-route.yaml), use the following command:

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

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

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


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

```
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.

```
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.

```
curl http://<VIP of NetScaler VPX>/ -H 'Host: < host-name-as-per-the-host-configuration-in-route >'
<!--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:

```
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:

```
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 NetScaler Ingress Controller with OpenShift router sharding support

December 31, 2023

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.

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

- NetScaler CPX deployed as an OpenShift router along with NetScaler Ingress Controller inside the cluster
- NetScaler Ingress Controller as a router plug-in for NetScaler MPX or VPX deployed outside the cluster

To configure router sharding for a NetScaler deployment on OpenShift, a NetScaler Ingress Controller instance is required per shard. The NetScaler Ingress Controller instance is deployed with route or namespace labels or both as environment variables depending on the criteria required for sharding. When the NetScaler 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 NetScaler, 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.

Label	Values
sla	high, medium, low
geo	east, west
hw	modest, strong
dept	finance, dev, ops
type	static, dynamic
frequency	high, weekly

The following table shows selectors for route labels or namespace labels and a few sample selection expressions based on labels in the example. Route selection criteria is configured on the NetScaler Ingress Controller by using environment variables ROUTE_LABELS and NAMESPACE_LABLES.

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

> **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 NetScaler Ingress Controller updates the configuration on NetScaler.

Deploy NetScaler 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.

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

3. Deploy the NetScaler Ingress Controller using the following command.

```
oc create -f cpx_cic_side_car.yaml
```

Deploy the NetScaler Ingress Controller router plug-in with OpenShift router sharding support

To deploy a NetScaler 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 NetScaler Ingress Controller using the following command.

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

Example: Create an OpenShift route and verify the route configuration on NetScaler 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 NetScaler 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`.

```
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
```

2. Use the following command to deploy the route.

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

3. Add labels to the namespace where you create the route.

```
oc label namespace default 'frequency=high'
```

Verify route configuration

You can verify the OpenShift route configuration on a NetScaler VPX by performing the following steps:

1. Log on to NetScaler VPX by performing the following:
   - Use an SSH client such as PuTTy, to open an SSH connection to NetScaler VPX.
   - Log on to NetScaler 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 NetScaler 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 NetScaler Ingress Controller in OpenShift using NetScaler Operator

April 25, 2024

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

The NetScaler Operator enables you to deploy NetScaler Ingress Controller in an OpenShift cluster.

Deployment options

Based on your requirement of NetScalers, there are two ways to deploy NetScaler Ingress Controller in an OpenShift cluster using the NetScaler Operator:

- As a standalone pod in the OpenShift cluster: In this mode, NetScaler Ingress Controller configures NetScaler MPX NetScaler VPX residing outside the OpenShift cluster.
- As a sidecar container alongside NetScaler CPX in the same pod: In this mode, NetScaler Ingress Controller configures NetScaler CPX deployed in the OpenShift cluster.

Deploy NetScaler Operator

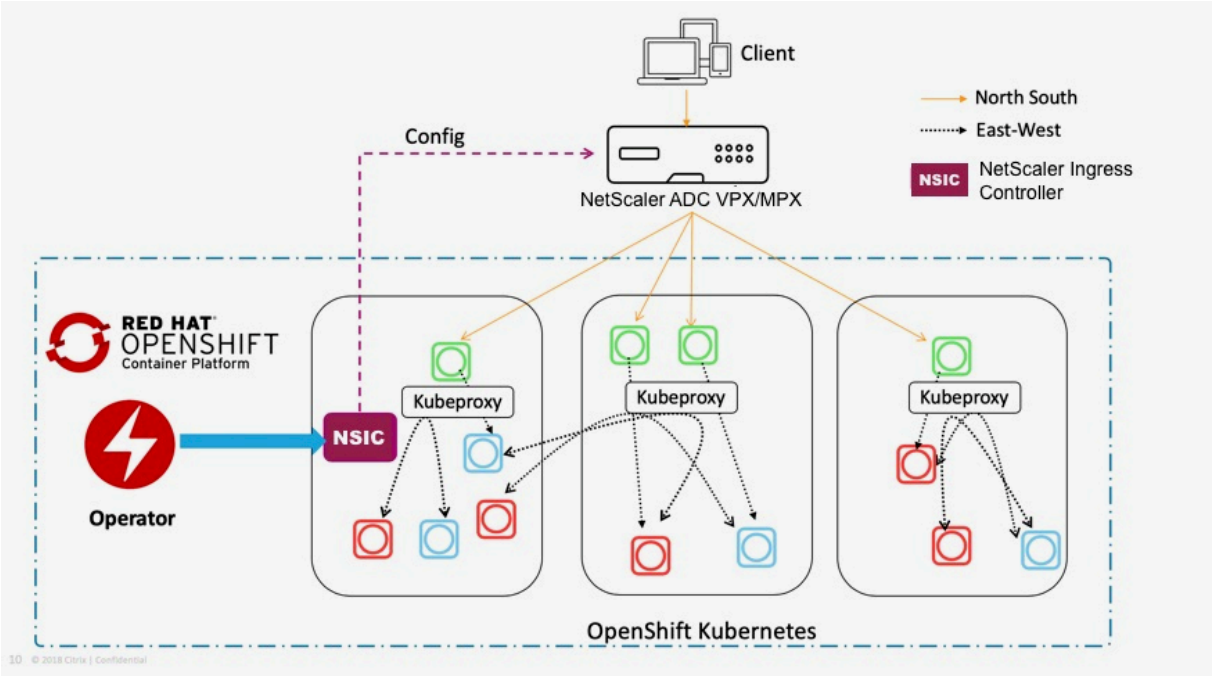
Perform the following steps:

1. Log in to the OpenShift cluster console.
2. Navigate to **Operators > OperatorHub**, select **Certified** source in the left panel, select **NetScaler Operator**, and then click **Install**.
3. To subscribe to NetScaler Operator, select one of the following options:
   - **All namespaces on the cluster (default)**: NetScaler Operator is available in all the namespaces on the OpenShift cluster. Hence, this option enables you to initiate the NetScaler instance from any namespace on the cluster.
   - **A specific namespace on the cluster**: NetScaler Operator is available in the selected namespace on the OpenShift cluster. Hence, this option enables you to initiate the NetScaler Operator instance on the selected namespace only.
4. In this case, let's select **A specific namespace on the cluster**.
5. Click **Install**.

   Wait until the NetScaler Operator is subscribed successfully.
6. Navigate to **Workloads > Pods** section and verify that the **netscaler-Operator-controller-manager** pod is up and running.

Deploy NetScaler Ingress Controller as a standalone pod using NetScaler Operator

Using the NetScaler Operator you can deploy NetScaler Ingress Controller as a standalone pod in an OpenShift cluster. NetScaler Ingress Controller configures the NetScaler VPX or MPX which is deployed as an Ingress device or router for an application running in the OpenShift cluster. The following diagram explains the topology:



Prerequisites

- Red Hat OpenShift Cluster (version 4.1 or later).
- Identify the IP address that NetScaler Ingress Controller needs to communicate with NetScaler. This IP address might be any one of the following IP addresses depending on the type of NetScaler deployment:
  - NSIP (for standalone appliances) - The management IP address of a standalone NetScaler appliance. For more information, see IP Addressing in NetScaler.
  - SNIP (for appliances in High Availability mode) - The subnet IP address. For more information, see IP Addressing in NetScaler.
  - CLIP (for appliances in Cluster mode) - The cluster management IP (CLIP) address for a clustered NetScaler deployment. For more information, see IP addressing for a cluster.
- The user name and password of NetScaler VPX or NetScaler MPX used as the Ingress device. NetScaler must have a system user account (non-default) with certain privileges so that the NetScaler Ingress Controller can configure NetScaler VPX or NetScaler MPX. For instructions to create a system user account on NetScaler, see Create a NetScaler user account to configure NetScaler using NetScaler Ingress Controller.

  You can directly pass the user name and password as environment variables to the controller, or use OpenShift secrets (recommended). To create a secret for the user name and password using the following command, modify the `<username>` and `<password>` to required values:


```
    oc create secret  generic nslogin --from-literal=username=<
       username> --from-literal=password=<password>
<!--NeedCopy-->
```

Deploy NetScaler Ingress Controller as a standalone pod using NetScaler Operator

Perform the following steps:

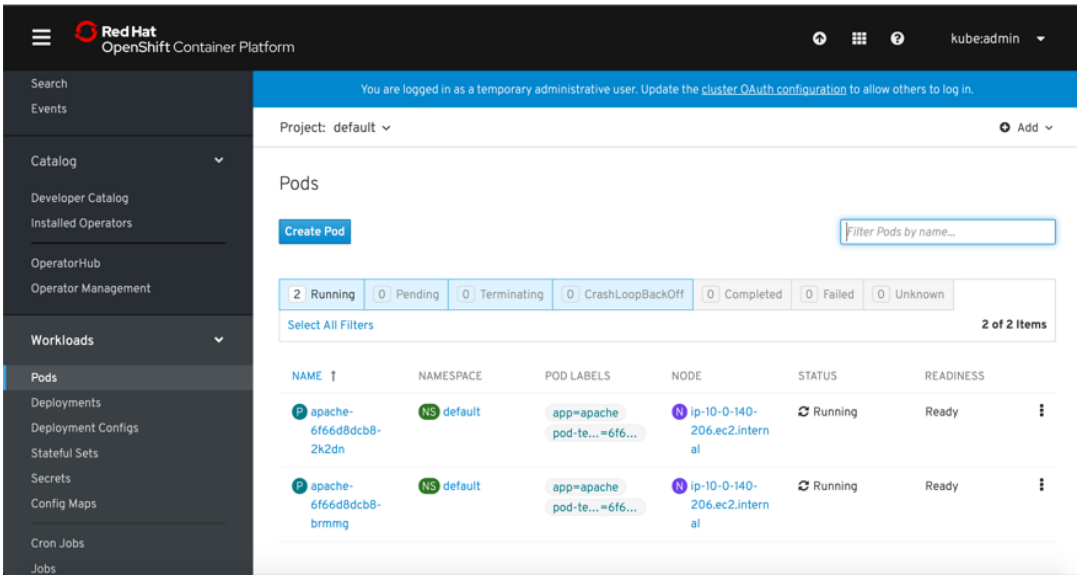
1. Log in to OpenShift 4.x Cluster console.
2. Deploy an Apache application using the console.

   a) Navigate to **Workloads > Deployments > Create Deployment** and use the following YAML file to create the deployment.

   > **NOTE:**
   >
   > The Apache application is for the demonstration purpose only. You can modify the YAML file based on your requirement.

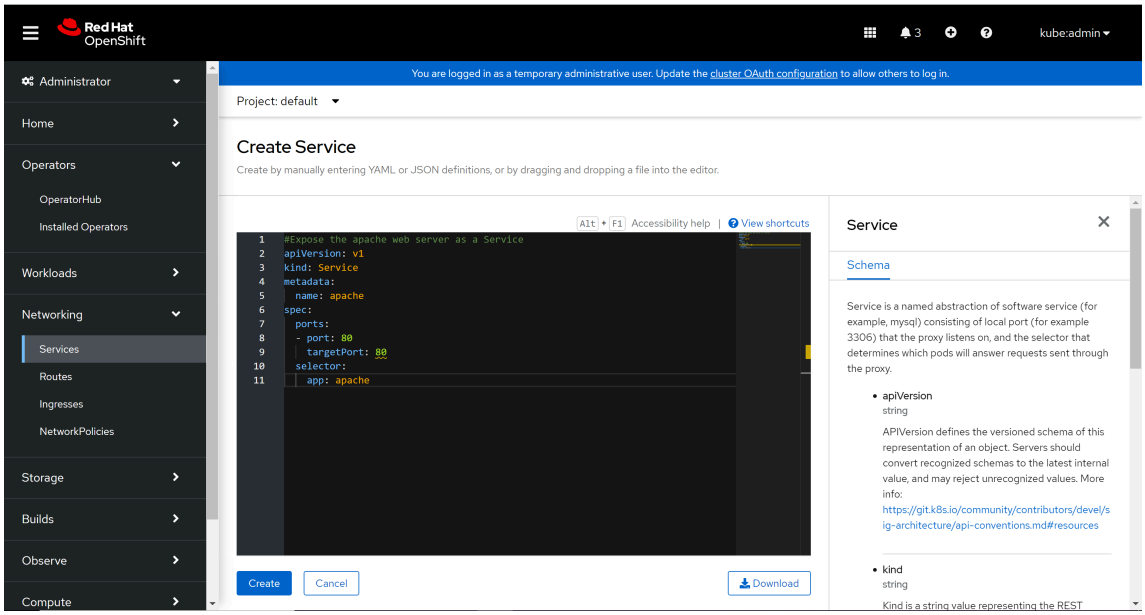
```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache
  labels:
      name: apache
spec:
  selector:
    matchLabels:
      app: apache
  replicas: 2
  template:
    metadata:
      labels:
        app: apache
    spec:
      containers:
      - name: apache
        image: httpd:latest
        ports:
        - containerPort: 80
    ---

<!--NeedCopy-->
```

   b) Navigate to **Workloads > Pods** section and ensure that the Apache application pods are up and running.



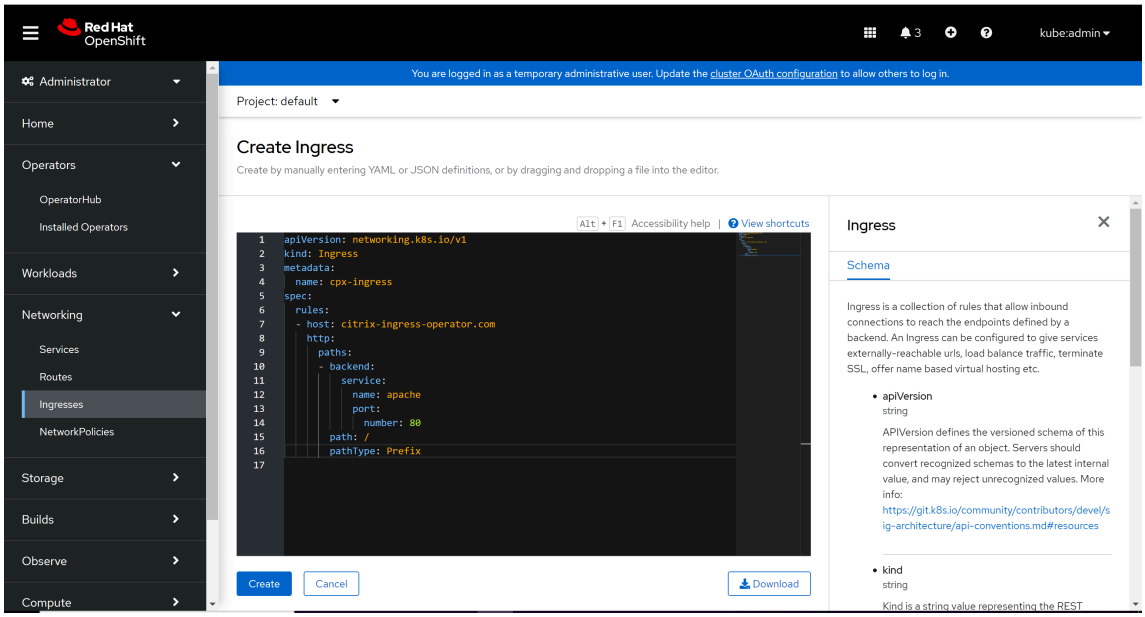
3. Create a service for the Apache application. Navigate to **Networking > Services > Create Service** and use the following YAML file.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: apache
spec:
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: apache

<!--NeedCopy-->
```

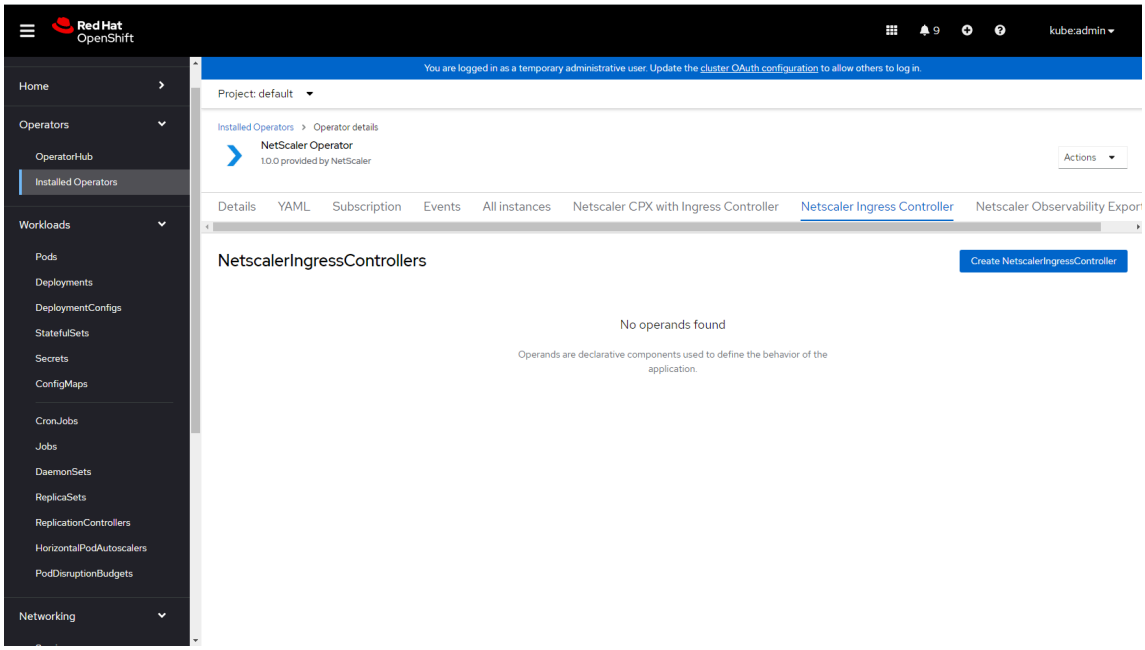



4. Create an ingress for the Apache application. Navigate to **Networking > Ingress > Create Ingress** and use the following YAML to create the ingress. Ensure to update VIP of the NetScaler VPX in the ingress YAML before applying it in the cluster.

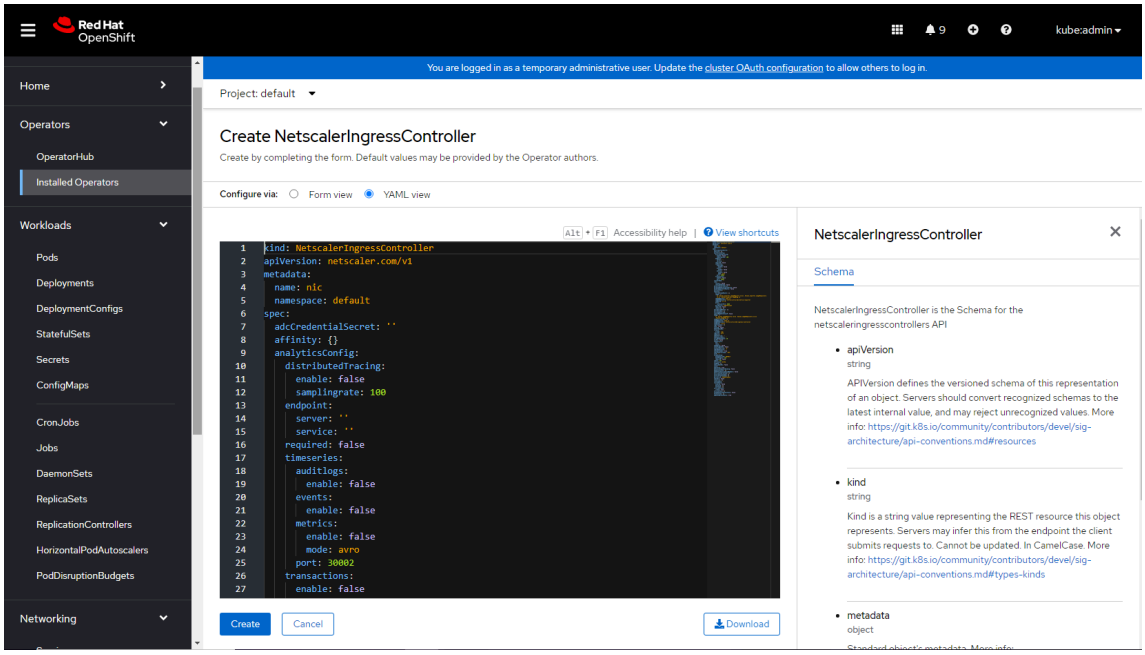
```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    ingress.citrix.com/frontend-ip: <NSVIP>
  name: vpx-ingress
  spec:
    rules:
    - host: citrix-ingress-operator.com
      http:
        paths:
        - backend:
            service:
              name: apache
              port:
                number: 80
        path: /
        pathType: Prefix

<!--NeedCopy-->
```

5. Navigate to **Operators > Installed Operators** and select **NetScaler Operator**.
6. Click **NetScaler Ingress Controller** tab and select **Create NetScalerIngressController**.



The NetScaler Ingress Controller YAML definition is displayed.



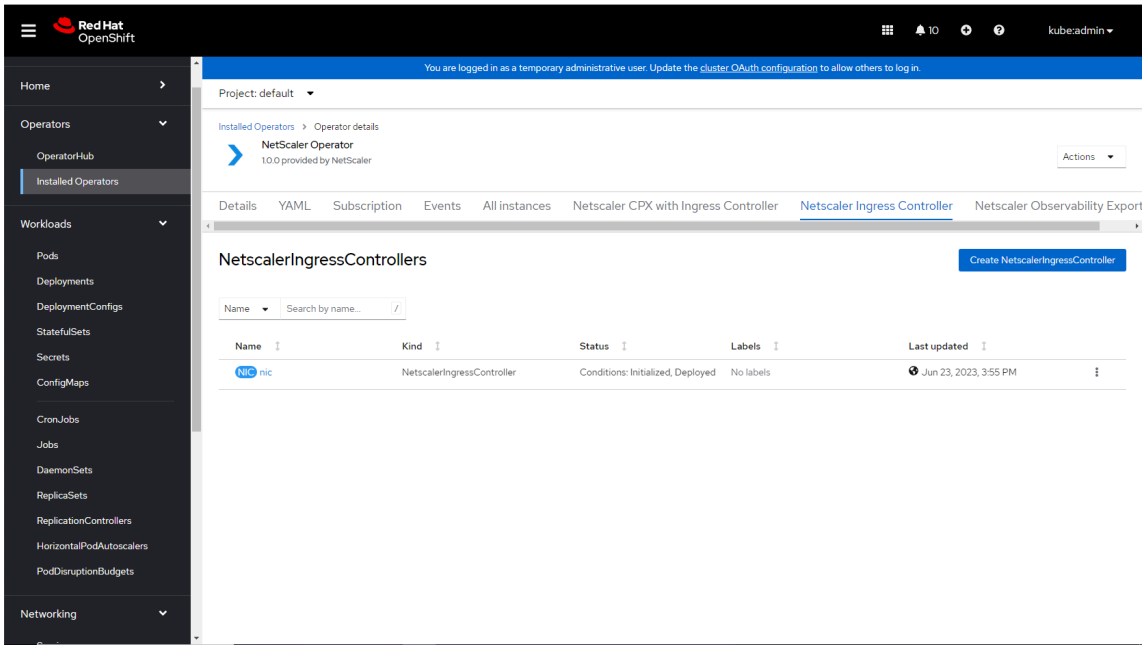
7. Refer this table that lists the mandatory and optional parameters that you can configure during installation.

> **Note:**
>
> - Ensure that the `license.accept` parameter is set to `yes`.
> - Provide the IP address of NetScaler VPX instance for `nsIP` parameter and Kubernetes secret created using NetScaler VPX credentials in `adcCredentialSecret` parameter respectively.

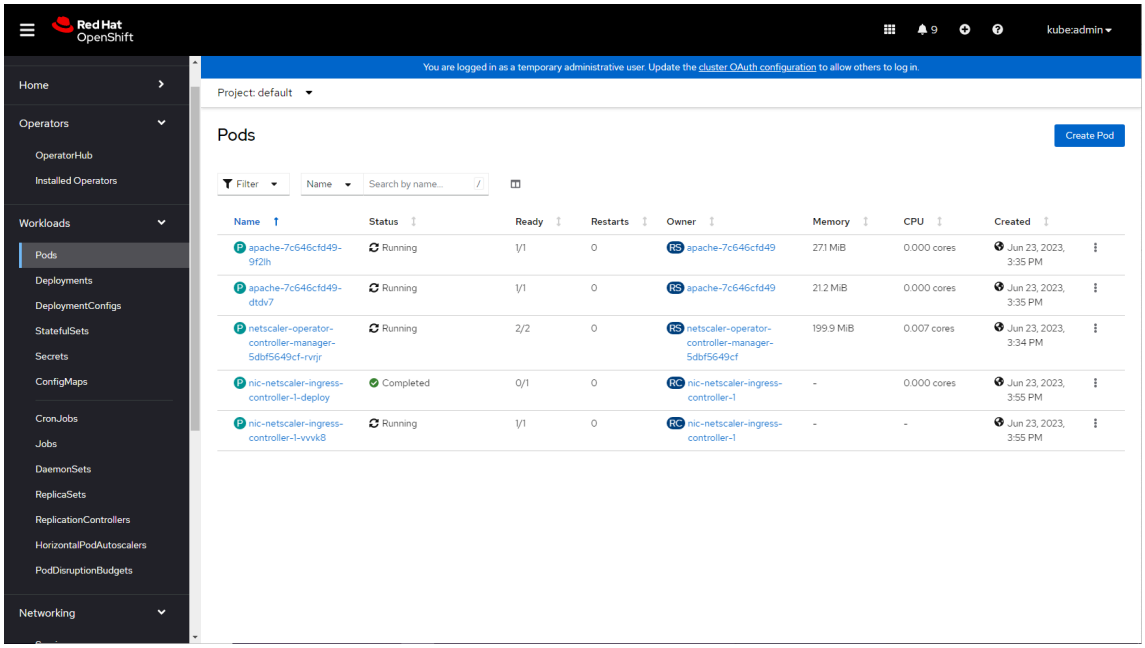
You can configure other available parameters depending upon your use case.

8. After updating the values for the required parameters, click **Create**.

   Ensure that NetScaler Ingress Controller is successfully deployed and initialized.



9. Navigate to **Workloads > Pods** section and verify whether the NetScaler Ingress Controller pod is up and running.



10. Verify the deployment by sending traffic:

```
curl http://citrix-ingress-Operator.com --resolve citrix-ingress-Operator.com:80:<VIP>
<!--NeedCopy-->
```

The previous curl command should return the following:

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



```
<!--NeedCopy-->
```

> **Note:**
>
> Ensure that the pod network in OpenShift cluster is reachable from NetScaler VPX or NetScaler MPX if you are using service of type ClusterIP for your application. To configure static route automatically using NSIC, see Configure static route.

Deploy NetScaler Ingress Controller as a sidecar with NetScaler CPX

Using the NetScaler Operator, you can deploy NetScaler CPX with the NetScaler Ingress Controller as a sidecar. The NetScaler Ingress Controller configures the NetScaler CPX which is deployed as an Ingress or router for an application running in the OpenShift cluster. The following diagram explains the topology.

Prerequisites

- Red Hat Openshift Cluster (version 4.1 or later).
- Install Prometheus Operator if you want to view the metrics of the NetScaler CPX collected through the direct Prometheus export.

Deploy NetScaler Ingress Controller as a sidecar with NetScaler CPX using NetScaler Operator

Perform the following steps:

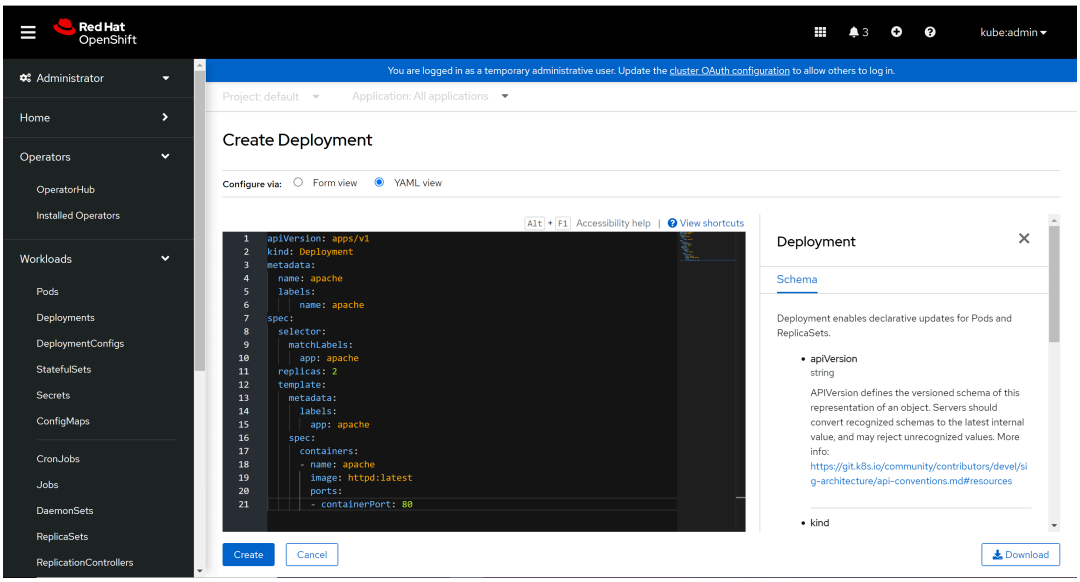
1. Log in to OpenShift 4.x Cluster console.
2. Deploy an Apache application using the console.

   a) Navigate to **Workloads > Deployments > Create Deployment** and use the following YAML to create the deployment.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache
  labels:
     name: apache
spec:
  selector:
    matchLabels:
      app: apache
  replicas: 2
  template:
```

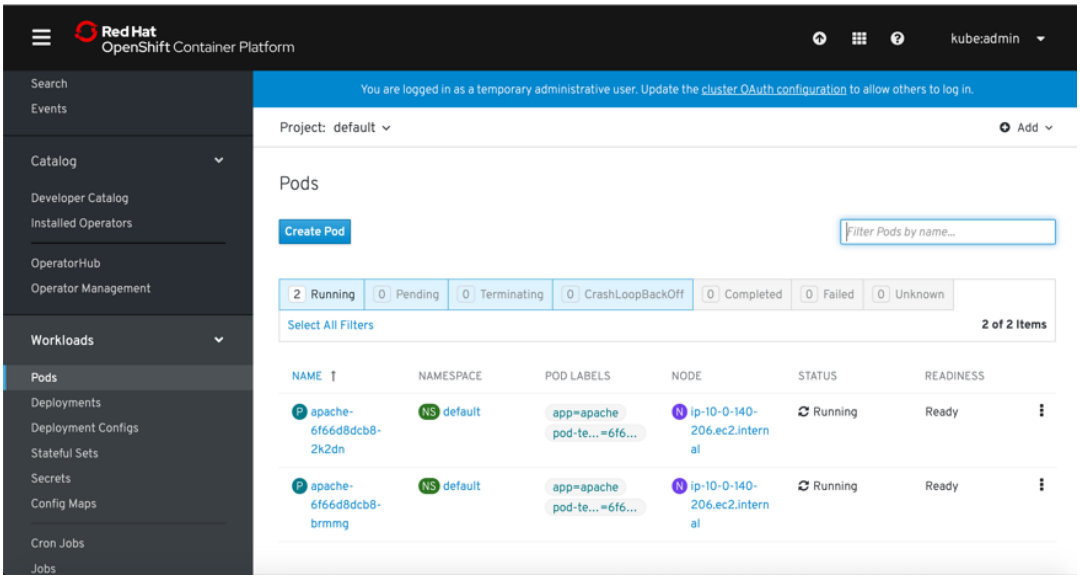


```
    metadata:
      labels:
        app: apache
    spec:
      containers:
      - name: apache
        image: httpd:latest
        ports:
        - containerPort: 80
    ---

<!--NeedCopy-->
```



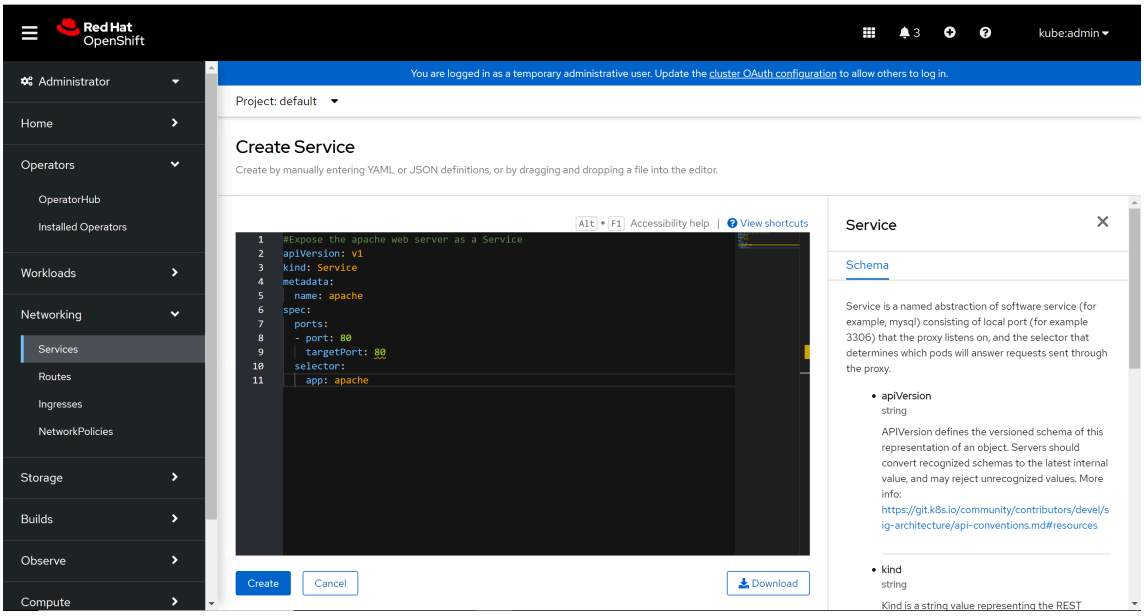
> **Note:**
>
> The Apache application is for the demonstration purpose only, you can modify the YAML file based on your requirement.

b) Navigate to **Workloads > Pods** section and ensure that the Apache application pods are up and running.



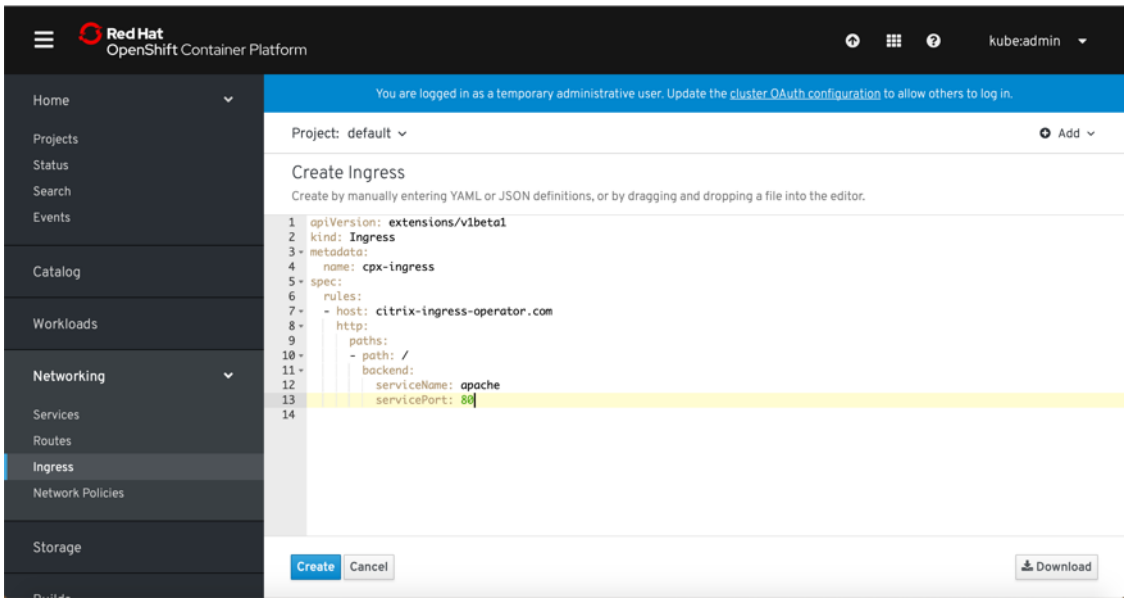
3. Create a service for the Apache application. Navigate to **Networking > Services > Create Service** and use the following YAML to create the service.

```
    apiVersion: v1
    kind: Service
    metadata:
      name: apache
    spec:
      ports:
      - port: 80
        targetPort: 80
      selector:
        app: apache
<!--NeedCopy-->
```



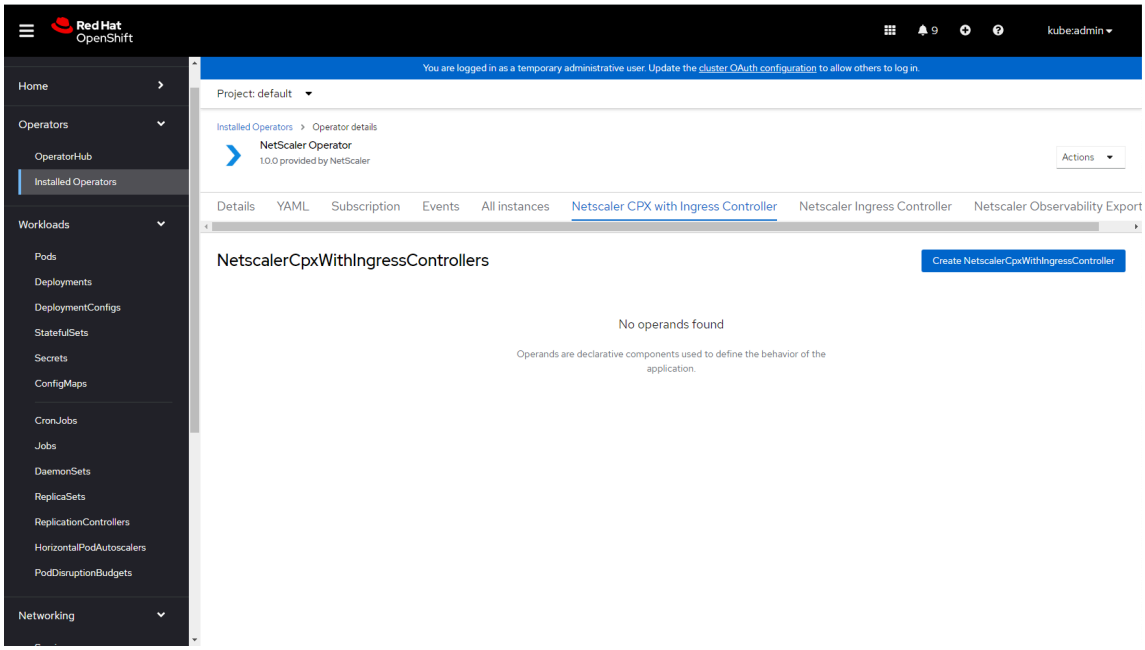
4. Create an Ingress for the Apache application. Navigate to **Networking > Ingress > Create Ingress** and use the following YAML to create the ingress.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: cpx-ingress
spec:
  rules:
  - host: citrix-ingress-operator.com
    http:
      paths:
      - backend:
          service:
            name: apache
            port:
              number: 80
        path: /
        pathType: Prefix
<!--NeedCopy-->
```

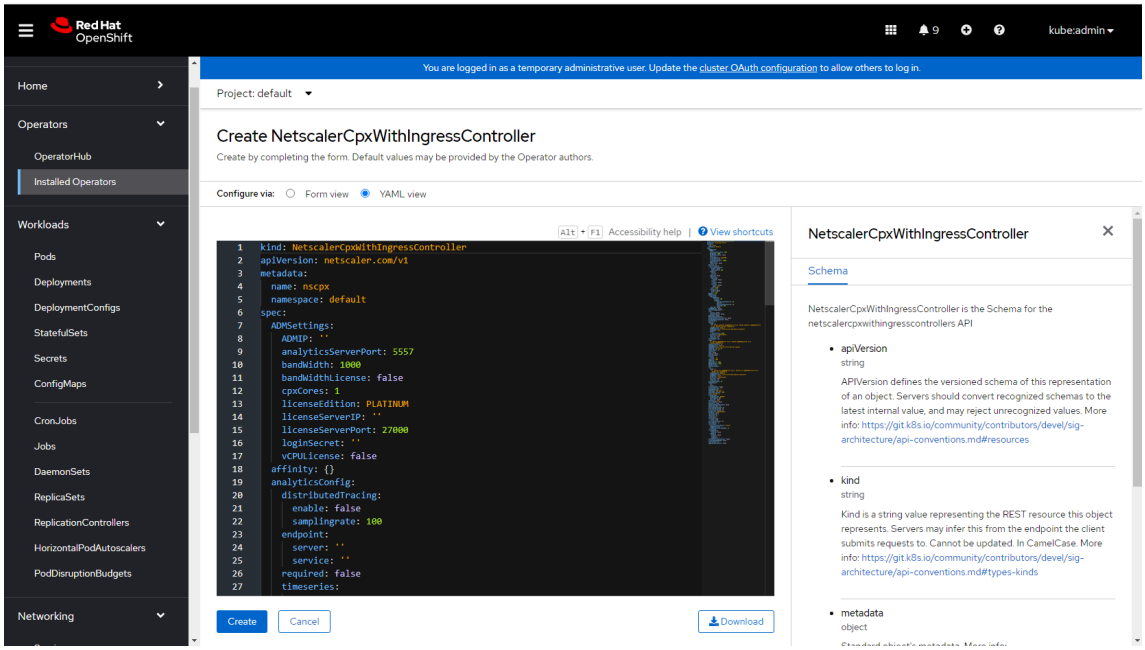


5. Navigate to **Operators > Installed Operators** and select **NetScaler Operator**.

6. Click **NetScaler CPX with Ingress Controller** tab and click **Create NetScalerCpxWithIngressController** .



The NetScaler CPX with ingress controller YAML definition is displayed.



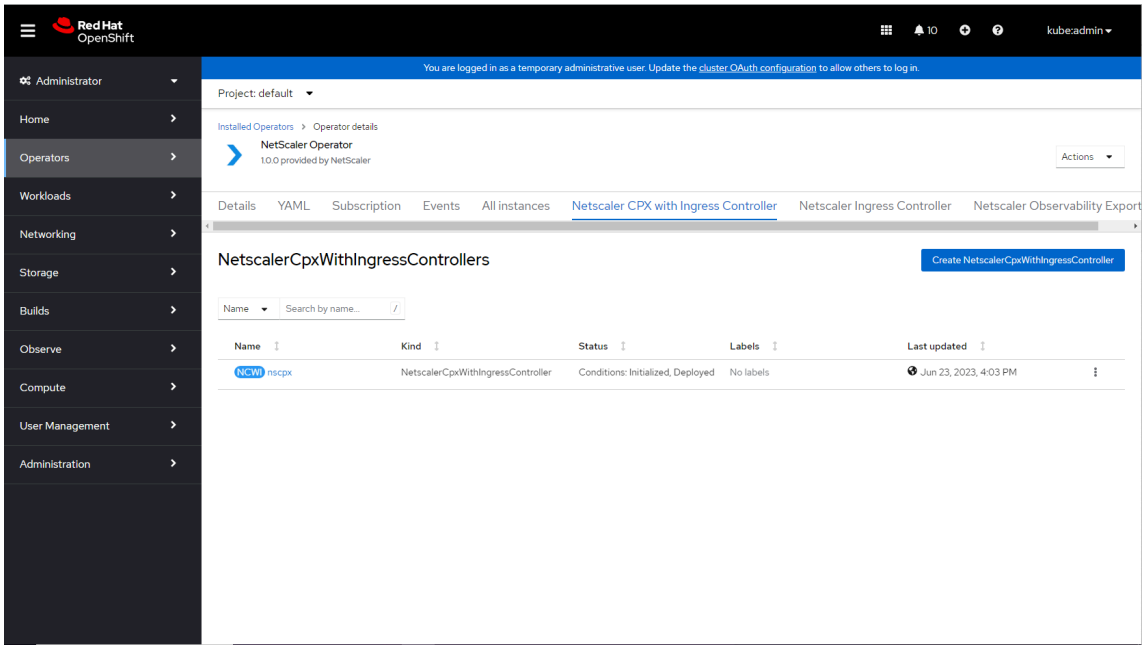
See this table that lists the mandatory and optional parameters that you can configure during installation.

> **Note:**
>
> - Ensure that the `license.accept` parameter is set to `yes`.
> - To expose NetScaler CPX service using type `nodePort` to access the Apache application, set `serviceType.nodePort.enabled` to true.

> - If CRDs are already installed, specify `crds.install=false`.

You can configure other available parameters depending upon your use case.

7. After updating the values for required parameters, click **Create**. Ensure that the NetScaler CPX with Ingress Controller is succesfully deployed and initialised.

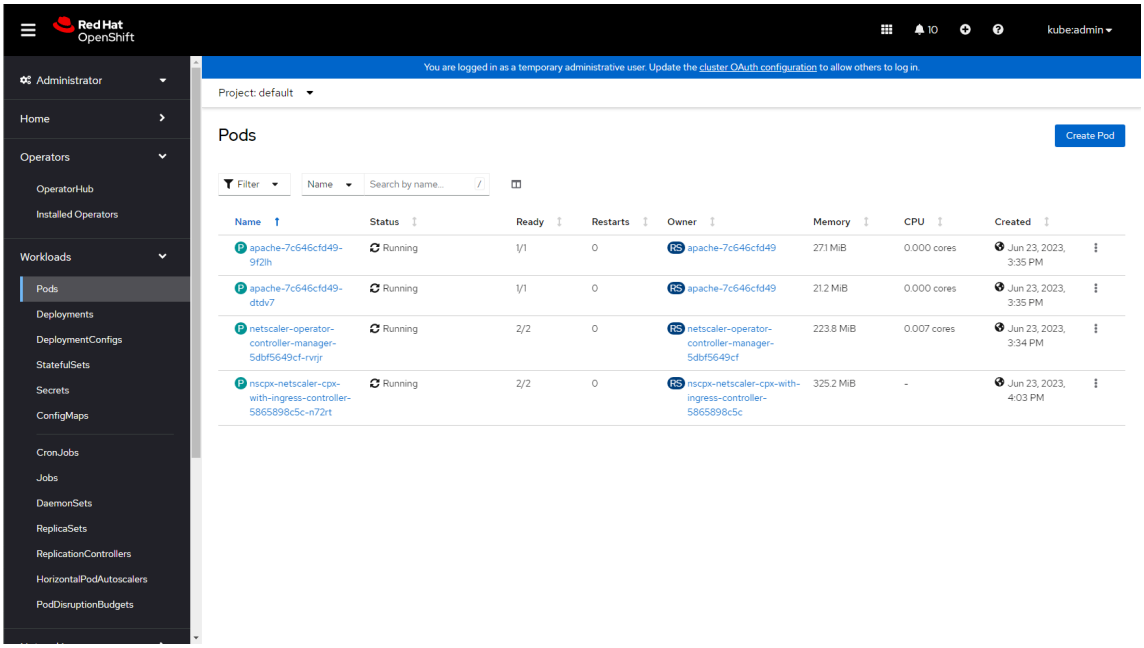


8. Attach privileged security context constraints to the service account of NetScaler CPX (as it runs as a privileged pod) by using the following commands:

   - Get the service account name used by NetScaler CPX using the following command in the namespace where NetScaler CPX has been deployed: `oc get sa`
   - Attach privileged SCC to the service account of the NetScaler CPX:

```
oc adm policy add-scc-to-user privileged -z <CPX-ServiceAccount-Name retrieved in the previous step>
<!--NeedCopy-->
```

9. Navigate to **Workloads > Pods** section and verify that the `netscaler-cpx-with-ingress-controller` pod is up and running.



10. Verify the deployment by sending traffic.

    a) Obtain the NodePort details using the following command:

```
oc get svc
<!--NeedCopy-->
```

    b) Use `cpx-service` NodePort and send the traffic as shown in the following command:

```
curl http://citrix-ingress-Operator.com:<NodePort> --resolve
    citrix-ingress-Operator.com:<NodePort>:<Master-Node-IP>
<!--NeedCopy-->
```

    The above curl command should return the following output:

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

References

- For information about how to deploy NetScaler Observability Exporter using NetScaler Operator, see Deploy NetScaler Observability Exporter using NetScaler Operator.
- For information about how to deploy NetScaler ADM Agent using NetScaler Agent Operator, see Install a NetScaler agent operator using the OpenShift console.
- Alternatively, you can deploy NetScaler Ingress Controller using Helm charts. See Deploy the NetScaler Ingress Controller using Helm charts.

Deploy NetScaler Observability Exporter using NetScaler Operator

April 23, 2024

NetScaler Observability Exporter is a container that collects metrics and transactions from NetScaler and transforms them to suitable formats such as JSON and AVRO for supported endpoints. You can export the data collected by NetScaler Observability Exporter to the desired endpoint for analysis and get valuable insights at the microservices level for applications proxied by NetScalers.

Prerequisites

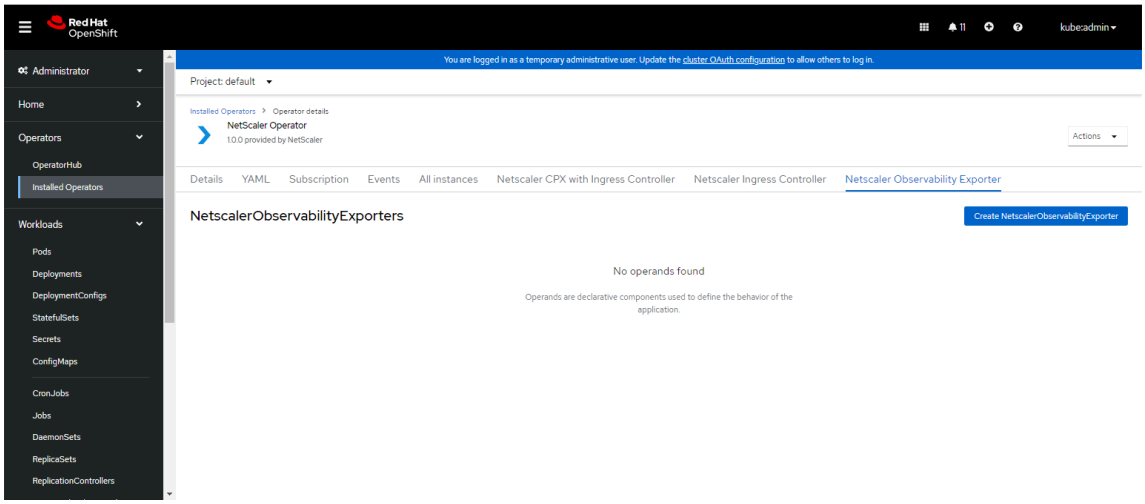
- Red Hat OpenShift Cluster (version 4.1 or later).
- Deploy NetScaler Operator. See Deploy NetScaler Operator.
- Because NSOE operates via any User ID (uid), deploy the following security context constraints (SCC) for the namespace in which NSOE is deployed.

```
oc adm policy add-scc-to-user anyuid system:serviceaccount:<namespace>:default
<!--NeedCopy-->
```

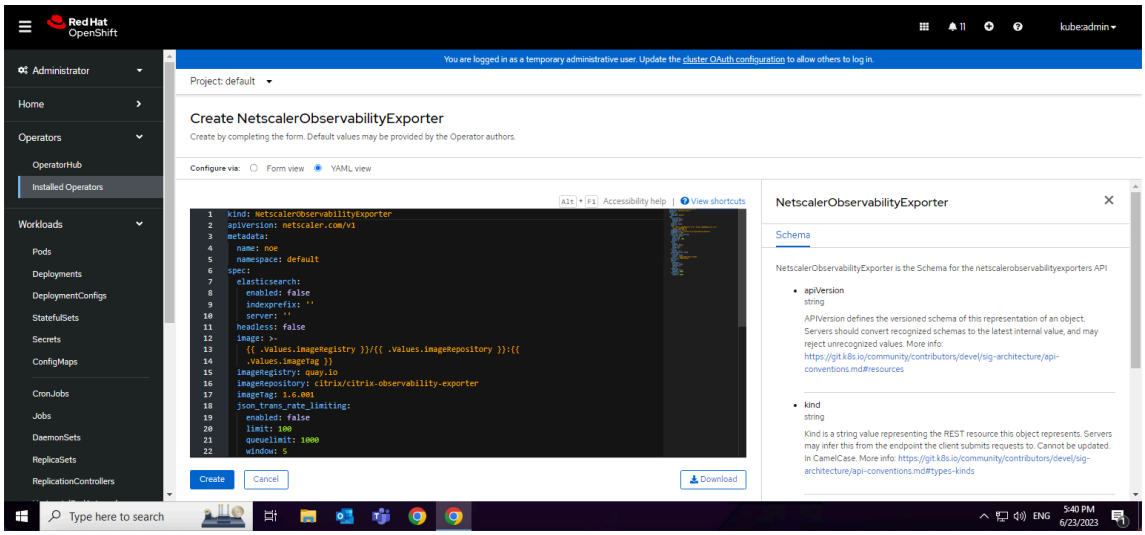
Deploy NetScaler Observability Exporter using NetScaler Operator

Perform the following steps:

1. Log in to OpenShift 4.x Cluster console.
2. Navigate to **Operators > Installed Operators** and select the **NetScaler Operator**.
3. Click **NetScaler Observability Exporter** tab and select **Create NetScalerObservabilityExporter** option.



The NetScaler Observability Exporter YAML definition is displayed.



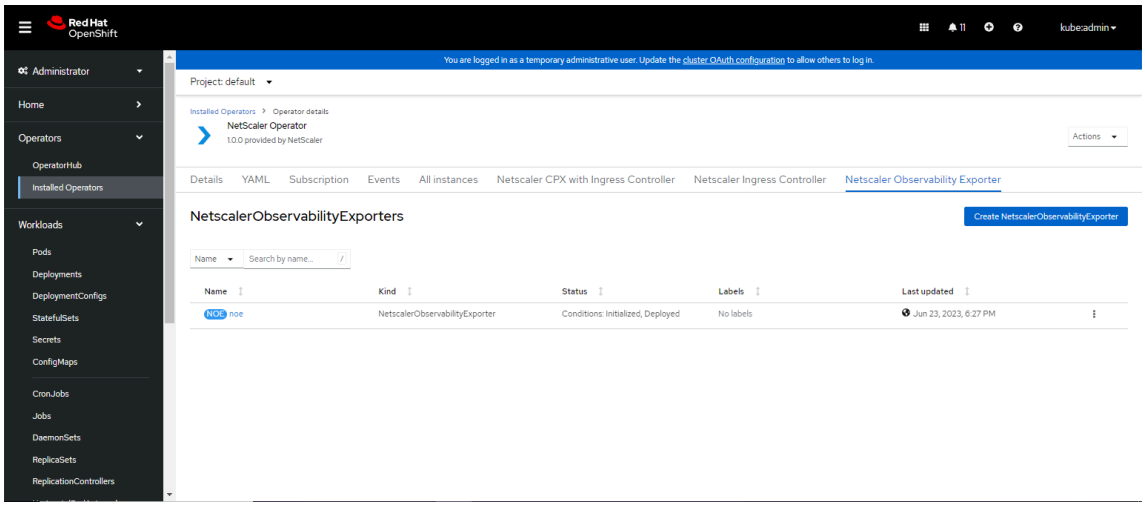
4. Refer this table that lists the mandatory and optional parameters and their default values that you can configure during installation.

> **Notes:**
>
> - To enable tracing, set `ns_tracing.enabled` to true and `ns_tracing.server` to the tracer endpoint such as **zipkin.default.cluster.svc.local:9411/api/v1/spans**. Default value for Zipkin server is zipkin:9411/api/v1/spans.
> - To enable Elasticsearch endpoint for transactions, set `elasticsearch.enabled` to true and `elasticsearch.server` to the elasticsearch endpoint such as **elasticsearch.default.svc.cluster.local:9200**. Default value for Elasticsearch endpoint is elasticsearch:9200.
> - To enable Kafka endpoint for transactions, set `kafka.enabled` to true. Set `kafka.broker`, `kafka.topic`, and `kafka.dataFormat` to required values. Default

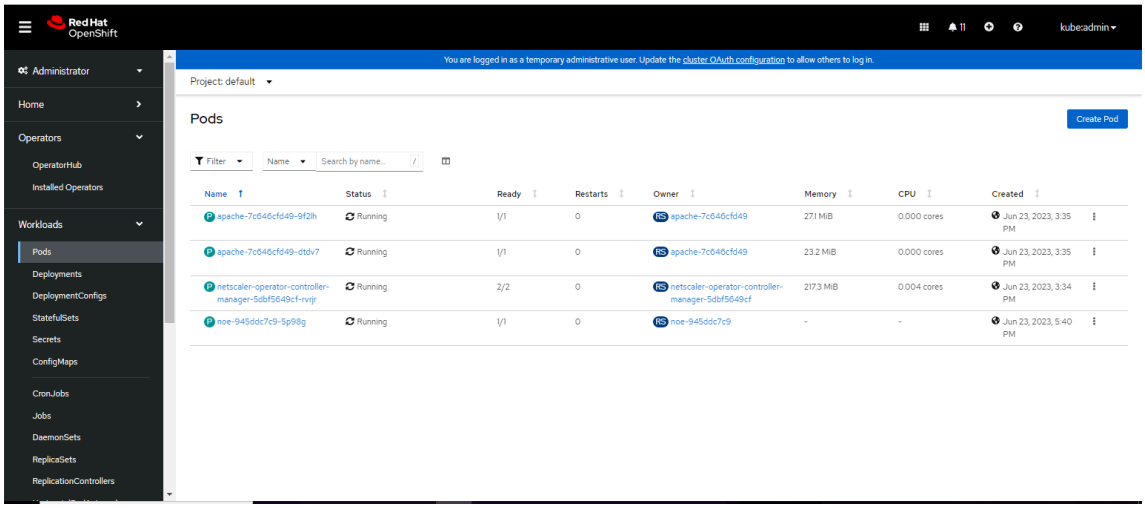
> value for `kafka.topic` is HTTP. Default value for `kafka.dataFormat` is AVRO.
> - To enable Timeseries data upload in prometheus format, set `timeseries.enabled` to true. Currently, Prometheus is the only Timeseries endpoint supported.
> - To enable Splunk endpoint for transactions, set `splunk.enabled` to true, `splunk.server` to Splunk server with port, `splunk.authtoken` to Splunk authentication token and `splunk.indexprefix` to index prefix to upload the transactions. Default value for `splunk.indexprefix` is adc_noe.

5. After updating the values for the required parameters, click **Create**.

   Ensure that the NetScaler Observability Exporter is succesfully deployed.



6. Navigate to **Workloads > Pods** section and verify that the NetScaler Observability Exporter pod is up and running.



Deploy NetScaler CPX as an Ingress device in an Azure Kubernetes Service cluster

December 31, 2023

This topic explains how to deploy NetScaler CPX as an ingress device in an Azure Kubernetes Service (AKS) cluster. NetScaler 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 NetScaler CPX or the NetScaler Ingress Controller instead of the default quay.io images, then see Deploy NetScaler CPX as an Ingress device in an AKS cluster using Azure repository images.

Deploy NetScaler CPX as an ingress device in an AKS cluster

Perform the following steps to deploy NetScaler 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.

```
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 NetScaler 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/azure/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/azure/manifest/cpx_ingress.yaml
```


4. Create a service of type LoadBalancer for accessing the NetScaler CPX by using the following command.

```
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.

```
kubectl  get svc

|NAME|TYPE|CLUSTER-IP|EXTERNAL-IP|PORT\(S)| AGE|
|----|----|-----|-----|----|----|
|apache |ClusterIP|10.0.103.3|none|   80/TCP | 2m|
|cpx-ingress |LoadBalancer |10.0.37.255 | pending |80:32258/TCP,443:32084/TCP |2m|
|Kubernetes |ClusterIP | 10.0.0.1 |none |  443/TCP | 22h |
```

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.

```
kubectl  get svc

|NAME|TYPE|CLUSTER-IP|EXTERNAL-IP|PORT\(S)|  AGE|
|---|---|----|----|----|----|
|apache|ClusterIP|10.0.103.3 |none|80/TCP|  3m|
|cpx-ingress |LoadBalancer|10.0.37.255|  EXTERNAL-IP CREATED|80:32258/TCP,443:32084/TCP |  3m|
|Kubernetes|    ClusterIP|10.0.0.1 |none| 443/TCP| 22h|
```

> **Note:**
>
> The health check for the cloud load-balancer is obtained from the readinessProbe configured in the NetScaler CPX deployment yaml file. If the health check fails, you should check the readinessProbe configured for NetScaler CPX. For more information, see readinessProbe and external Load balancer.

7. Access the application using the following command.

```
curl http://<External-ip-of-loadbalancer>/ -H 'Host: citrix-ingress.com
```

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 NetScaler CPX ingress with in built NetScaler Ingress Controller in your Kubernetes cluster using the all-in-one.yaml.

```
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.

```
curl http://<External-ip-of-loadbalancer>/ -H 'Host: citrix-ingress.com'
```

> **Note:**
>
> To delete the deployment, use the `kubectl delete -f all-in-one.yaml` command.

Deploy NetScaler Ingress Controller in an Azure Kubernetes Service cluster with NetScaler VPX

December 31, 2023

This topic explains how to deploy the NetScaler Ingress Controller with NetScaler VPX in an Azure Kubernetes Service (AKS) cluster. You can also configure the Kubernetes cluster on Azure VMs and then deploy the NetScaler Ingress Controller with NetScaler 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

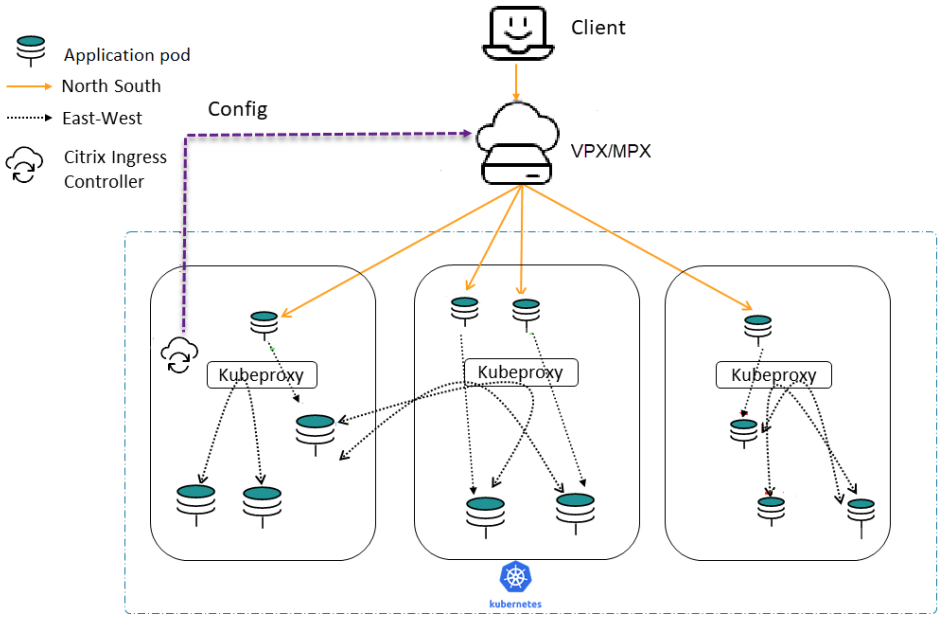
You should complete the following tasks before performing the steps in the procedure.

- 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 NetScaler VPX instance from Azure Marketplace

You can create NetScaler VPX from the Azure Marketplace.
For more information on how to create a NetScaler VPX instance from Azure Marketplace, see Get NetScaler VPX from Azure Marketplace.

Get the NetScaler Ingress Controller from Azure Marketplace

To deploy the NetScaler Ingress Controller, an image registry should be created on Azure and the corresponding image URL should be used to fetch the NetScaler Ingress Controller image.

For more information on how to create a registry and get the image URL, see Get NetScaler Ingress Controller from Azure Marketplace.

Once a registry is created, the NetScaler 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 NetScaler Ingress Controller

Perform the following steps to deploy the NetScaler Ingress Controller.

1. Create NetScaler 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 NetScaler VPX user name and password should be the same as the credentials set while creating NetScaler VPX on Azure.

2. Using SSH, configure a SNIP in the NetScaler VPX, which is the secondary IP address of the NetScaler VPX. This step is required for the NetScaler 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 NetScaler VPX instance creation.
   - `vpx-instance-primary-ip-subnet` is the subnet of the primary private IP address of the NetScaler VPX instance.

   To verify the subnet of the private IP address, SSH into the NetScaler VPX instance and use the following command.

```
 show ip <primary-private-ip-addess>
```

3. Update the NetScaler VPX image URL, management IP, and VIP in the NetScaler Ingress Controller YAML file.

   a) Download the NetScaler 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 NetScaler 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 NetScaler VPX in the `cic.yaml` in the following field 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 NetScaler VPX VIP in the `cic.yaml` in the following field with the private IP address of the VIP assigned during VPX Azure instance creation.

```
# Set NetScaler VIP for the data traffic
- name: "NS_VIP"
  value: "X.X.X.X"
```

4. Once you have configured the NetScaler Ingress Controller with the required values, deploy the NetScaler Ingress Controller using the following command.

```
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.

```
kubectl create -f  https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/azure/manifest/azurecic/apache.yaml
```

2. Create the Ingress resource using the following command.

```
kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/azure/manifest/azurecic/ingress.yaml
```

3. To validate your deployment, use the following command.

```
$ curl --resolve citrix-ingress.com:80:<Public-ip-address-of-VIP> http://citrix-ingress.com
<html><body><h1>It works!</h1></body></html>
```

The response is received from the sample microservice (Apache) which is inside the Kubernetes cluster. NetScaler VPX has load-balanced the request.

Deploy NetScaler CPX as an Ingress device in Google Cloud Platform

December 31, 2023

This topic explains how to deploy NetScaler CPX as an ingress device in Google Kubernetes Engine (GKE)

Prerequisites

You should complete the following tasks before performing the steps in the procedure.

- Ensure that you have a Kubernetes Cluster up and running.
- If you are running your cluster in GKE, ensure that you have configured a cluster-admin role binding.

You can use the following command to configure cluster-admin role binding.

```
kubectl create clusterrolebinding citrix-cluster-admin --clusterrole=
    cluster-admin --user=<email-id of your google account>
```

You can get your Google account details using the following command.

```
gcloud info | grep Account
```

Deploy NetScaler 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 NetScaler 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 NetScaler 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

|NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT\(S) | AGE |
| --- | ---| ----| ----| ----| ----|
|apache | ClusterIP |10.7.248.216 |none |  80/TCP | 2m |
|cpx-ingress |LoadBalancer | 10.7.241.6 |  pending | 80:32258/TCP,443:32084/TCP | 2m|
|kubernetes |ClusterIP |10.7.240.1 |none | 443/TCP | 22h|
```

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

|Name | Type | Cluster-IP | External IP| Port\(s) | Age |
|-----| -----| -------| -----| -----| ----|
|apache| ClusterIP|10.7.248.216|none|80/TCP |3m|
|cpx-ingress|LoadBalancer|10.7.241.6|EXTERNAL-IP CREATED|80:32258/TCP,443:32084/TCP|3m|
|kubernetes| ClusterIP| 10.7.240.1|none|443/TCP|22h|`
```

Note:
The health check for the cloud load-balancer is obtained from the readinessProbe configured in the NetScaler CPX service YAML file. If the health check fails, you should check the readinessProbe configured for NetScaler CPX.
For more information, see readinessProbe and external Load balancer.

3. Access the application using the following command.

```
curl http://<External-ip-of-loadbalancer>/ -H 'Host: citrix-ingress.com'
```


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 NetScaler CPX ingress with in built NetScaler Ingress Controller in your Kubernetes cluster using the all-in-one.yaml.

```
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.

```
curl http://<External-ip-of-loadbalancer>/ -H 'Host: citrix-ingress.com'
```

> **Note:**
>
> To delete the deployment, use the `kubectl delete -f all-in-one.yaml` command.

Deploy the NetScaler Ingress Controller in Anthos

December 31, 2023

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 NetScaler Ingress Controller can be deployed in Anthos GKE on-premises using the following deployment modes:

- Exposing NetScaler CPX with the sidecar ingress controller as a service of type `LoadBalancer`.
- Dual-tier Ingress deployment

Expose NetScaler CPX as a service of type `LoadBalancer` in Anthos GKE on-prem

In this deployment, NetScaler VPX or MPX is deployed outside the cluster at Tier-1 and NetScaler CPX at Tier-2 inside the Anthos cluster similar to a dual-tier deployment. However instead of using Ingress, the NetScaler CPX is exposed using the Kubernetes service of type `LoadBalancer`.
The NetScaler 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 NetScaler 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 NetScaler and Anthos GKE on-prem cluster nodes should be reachable using the IP address.
- To use a NetScaler VPX or MPX from a different network, use node controller to enable communication between the NetScaler 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 NetScaler CPX as service of type `LoadBalancer` in Anthos GKE on-premises

Perform the following steps to deploy NetScaler 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 NetScaler CPX with the sidecar NetScaler 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.

```
openssl req -subj '/CN=anthos-citrix-ingress.com/O=Citrix Systems Inc/C=IN' -new -newkey rsa:2048 -days 5794 -nodes -x509 -keyout $PWD/anthos-citrix-certificate.key -out $PWD/anthos-citrix-certificate.crt;openssl rsa -in $PWD/anthos-citrix-certificate.key -out $PWD/anthos-citrix-certificate.key
```


> **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.

```
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.

```
kubectl --kubeconfig user-cluster-1-kubeconfig create -f https://
    raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/
    master/deployment/anthos/manifest/service-type-lb/tier-2-
    ingress.yaml
```

6. Create a Kubernetes secret for the Tier-1 NetScaler.

```
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 NetScaler Ingress Controller as a Tier-1 ingress controller.

   a) Download the cic.yaml file.

   b) Enter the management IP address of NetScaler. Update the Tier-1 NetScaler'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
```

8. Expose NetScaler 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-service-type-lb.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.

```
curl -k --resolve anthos-citrix-ingress.com:443:<VIP-for-accessing-microservices> https://anthos-citrix-ingress.com/ <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 in step 9 for demonstration purpose.

Clean up the installation: Expose NetScaler CPX as service of type `LoadBalancer`

To clean up the installation, use the `kubectl --kubeconfig delete` command to delete each deployment.

To delete the NetScaler CPX service deployment (CPX+CIC service) use the following command:

```
kubectl --kubeconfig user-cluster-1-kubeconfig delete -f cpx-service-type-lb.yaml
```

To delete the Tier-2 Ingress object, use the following command.

```
kubectl --kubeconfig user-cluster-1-kubeconfig delete -f tier-2-ingress.yaml
```

To delete the NetScaler CPX deployment along with the sidecar NetScaler Ingress Controller, use the following command.

```
kubectl --kubeconfig user-cluster-1-kubeconfig delete -f cpx-cic.yaml
```

To delete the stand-alone NetScaler 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
```

Dual tier Ingress deployment

In a dual-tier Ingress deployment, NetScaler VPX or MPX is deployed outside the Kubernetes cluster (Tier-1) and NetScaler CPXs are deployed inside the Kubernetes cluster (Tier-2).

NetScaler MPX or VPX devices in Tier-1 proxy the traffic (North-South) from the client to NetScaler CPXs in Tier-2. The Tier-2 NetScaler CPX then routes the traffic to the microservices in the Kubernetes cluster. The NetScaler Ingress Controller deployed as a standalone pod configures the Tier-1 NetScaler. The sidecar NetScaler Ingress Controller in one or more NetScaler CPX pods configures the associated NetScaler CPX in the same pod.

Prerequisites

- You must deploy a Tier-1 NetScaler 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 NetScaler and Anthos GKE on-prem cluster nodes should be reachable using the IP address.
- To use a NetScaler VPX or MPX from a different network, use the node controller to enable communication between the NetScaler 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 NetScaler 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.


```
kubectl --kubeconfig user-cluster-1-kubeconfig create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/anthos/manifest/dual-tiered-ingress/apache.yaml
```

> **Note:**
>
> In this example, `apache.yaml` is used. You should use the specific YAML file for your application.

2. Deploy NetScaler CPX with the NetScaler Ingress Controller as Tier-2 Ingress 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/dual-tiered-ingress/cpx-cic.yaml
```

3. Expose NetScaler CPX as a Kubernetes service using the `cpx-service.yaml` file.

```
kubectl --kubeconfig user-cluster-1-kubeconfig create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/anthos/manifest/dual-tiered-ingress/cpx-service.yaml
```

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.

```
openssl req -subj '/CN=anthos-citrix-ingress.com/O=Citrix Systems Inc/C=IN' -new -newkey rsa:2048 -days 5794 -nodes -x509 -keyout $PWD/anthos-citrix-certificate.key -out $PWD/anthos-citrix-certificate.crt;openssl rsa -in $PWD/anthos-citrix-certificate.key -out $PWD/anthos-citrix-certificate.key
```

> **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.

```
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.


```
kubectl --kubeconfig user-cluster-1-kubeconfig create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/anthos/manifest/dual-tiered-ingress/tier-2-ingress.yaml
```

7. Create a Kubernetes secret for the Tier-1 NetScaler.

```
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 NetScaler Ingress Controller as a Tier-1 ingress controller.

   a) Download the cic.yaml file.

   b) Enter the management IP address of NetScaler. Update the Tier-1 NetScaler'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.

```
curl -k --resolve anthos-citrix-ingress.com:443:<VIP-Citrix-ADC> https://anthos-citrix-ingress.com/
<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.

```
kubectl --kubeconfig user-cluster-1-kubeconfig delete -f tier-1-ingress.yaml
```

To delete the Tier-2 Ingress object, use the following command.

```
kubectl --kubeconfig user-cluster-1-kubeconfig delete -f tier-2-ingress.yaml`
```

To delete the NetScaler CPX deployment along with the sidecar NetScaler Ingress Controller, use the following command.

```
kubectl --kubeconfig user-cluster-1-kubeconfig delete -f cpx-cic.yaml
```

To delete the NetScaler CPX service deployment, use the following command:

```
kubectl --kubeconfig user-cluster-1-kubeconfig delete -f cpx-service.yaml
```

To delete the stand-alone NetScaler 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`
```

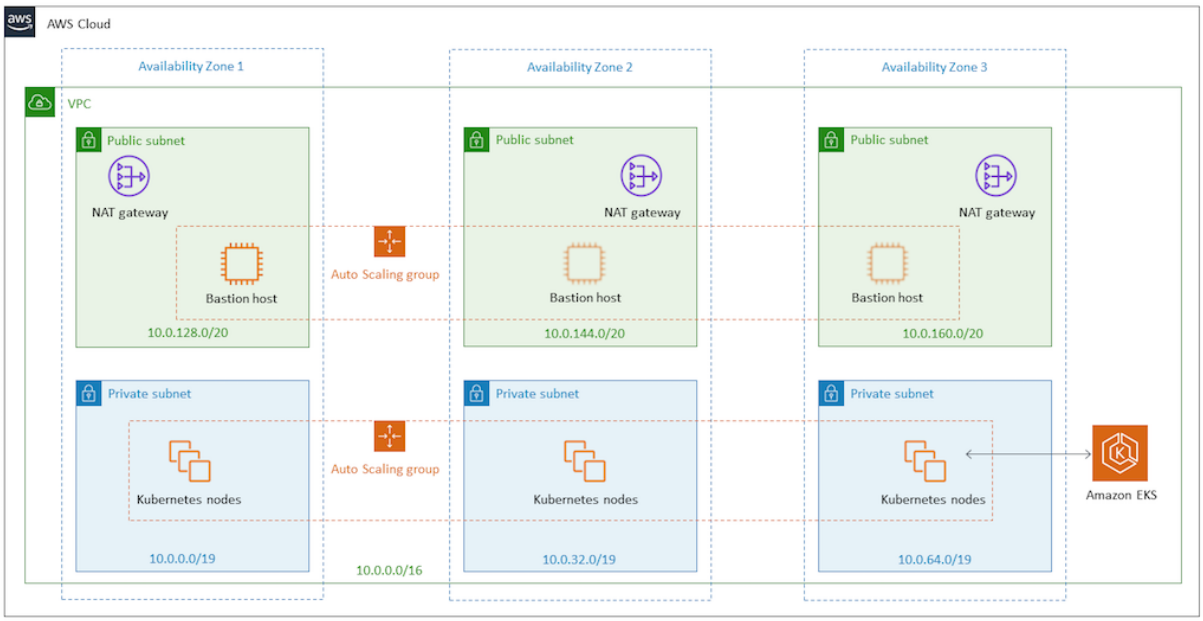

Deploy NetScaler VPX in active-active high availability in EKS environment using Amazon ELB and NetScaler Ingress Controller

December 31, 2023

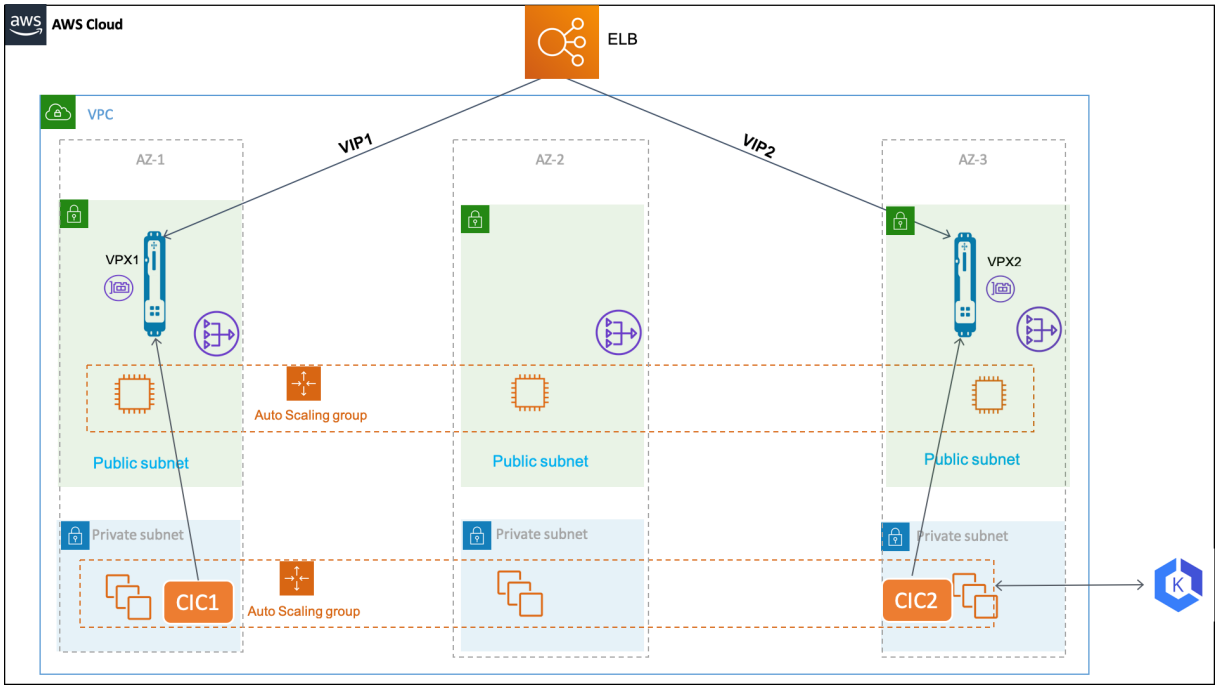
The topic covers a solution to deploy NetScaler VPX in active-active high availability mode on multiple availability zones in AWS Elastic Container Service (EKS) platform. The solution combines AWS Elastic load balancing (ELB) and NetScaler VPX to load balance the Ingress traffic to the microservices deployed in EKS cluster. AWS ELB handles the Layer 4 traffic and the NetScaler 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 NetScaler VPX instances. The NetScaler 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 NetScaler VPX.

A NetScaler Ingress Controller is deployed in the EKS cluster for each NetScaler VPX instance. The NetScaler Ingress Controllers are configured with the same ingress class. And, it configures the Ingress objects in the EKS cluster on the respective NetScaler 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 NetScaler VPX Instances.
2. Deploy NetScaler 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 NetScaler VPX instances

NetScaler VPX is available as CloudFormation Template. The CloudFormation template deploys an instance of NetScaler VPX with single ENI on a given subnet. It also configures the NSIP, VIP, and SNIP for the NetScaler VPX instance.

For this solution you need to deploy two instances of NetScaler VPX. Deploy the NetScaler VPX instances on two availability zones by specifying the same NetScaler VPX and different public subnet.

After you deploy the NetScaler 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 NetScaler VPX instances:

Overview | Outputs | Resources | Events | Template | Parameters | Tags | Stack Policy | Change Sets | Rollback Triggers

Key	Value	Description	Export Name
PrivateNSIP	10.0.165.108	Private IP (NS IP) used for management	
SNIP	10.0.160.143	Private IP address of the VPX instance associate...	
PrivateVIP	10.0.161.33	Private IP address of the VPX instance associate...	
PublicNSIP	54.68.242.236	Public IP (NS IP) used for management	
ManagementURL	https://54.68.242.236	HTTPS URL to the Management GUI (uses self-si...	
InstanceIdNS	i-08687cfee6f5c3433	Instance Id of newly created VPX instance	
SecurityGroup	sg-0bd10f17e64d6f194	Security group id that the VPX belongs to	
PublicIpVIP	54.202.202.189	Elastic IP address of the VPX instance associated...	
ManagementURL2	http://54.68.242.236	HTTP URL to the Management GUI	

> **Note:**
>
> The CloudFormation template deploys the NetScaler VPX instance with primary IP address of the NetScaler VPX EC2 instance as VIP and secondary IP address as management IP address.

After the NetScaler 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 NetScaler Ingress Controller

Deploy separate instance of NetScaler Ingress Controller for each NetScaler VPX instance. Follow the deployment instructions to deploy NetScaler Ingress Controller.

After the NetScaler VPX instance is up, you must set up a system user account on the NetScaler VPX instances. The system user account is used by NetScaler Ingress Controller to log into the NetScaler VPX instances. For instruction to set up the system user account, see Create System User Account for CIC in NetScaler.

1. Edit the NetScaler Ingress Controller deployment YAML (citrix-ingress-controller.yaml).

   Replace `NS_IP` with the `Private NSIP` address of the respective NetScaler VPX instance. Also, provide the system user account user name and password that you have created on the NetScaler VPX instance. Once you edited the `citrix-ingress-controller.yaml` file, deploy the updated YAML file using the following command:

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

2. Perform Step 1 on the second NetScaler Ingress Controller instance.
3. Ensure that both the pods are UP and running. Also, verify if NetScaler Ingress Controller is able to connect to the respective NetScaler VPX instance using the logs:

   ```
   kubectl logs <cic_pod_name>
   ```

After the NetScaler 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 NetScaler VPX instances.

Setup elastic load balancing

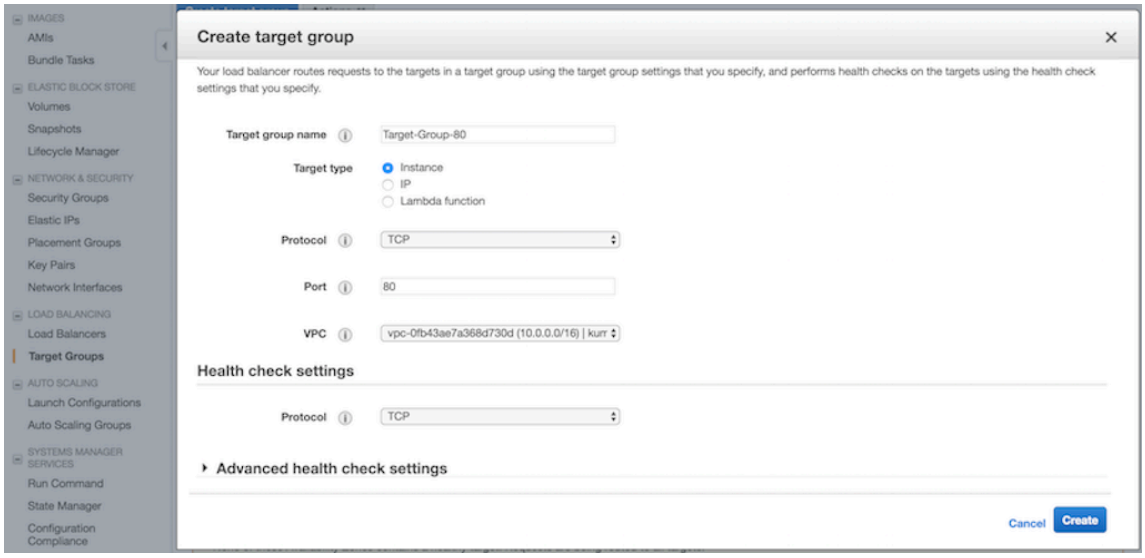
Depending upon your requirement you can configure any of the following load balancers:

- Network Load Balancers
- Classic Load Balancers

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 NetScaler 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.
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.

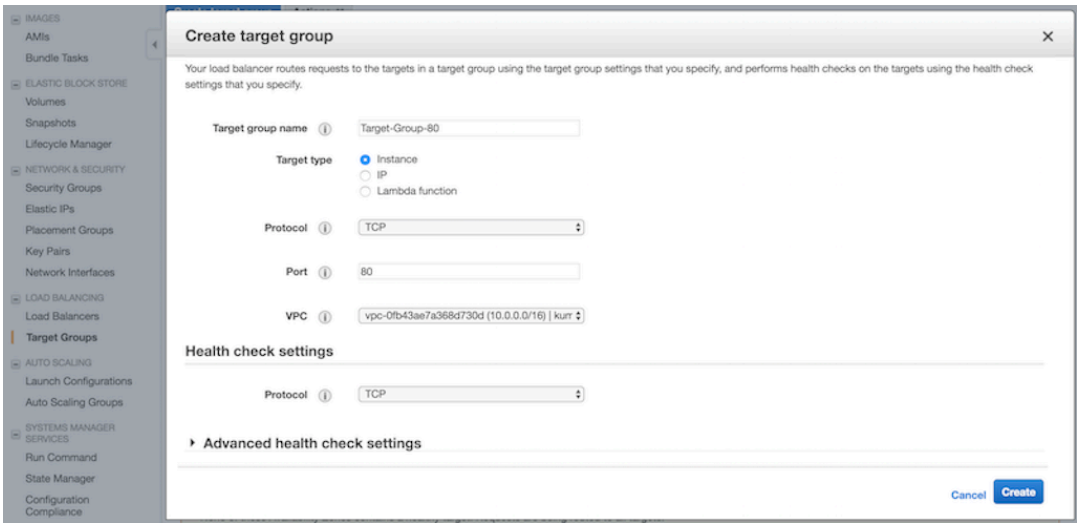
   c) From the **Protocol** list, select **TCP**.

   d) In the **Port** field, enter **80**.

   e) From the **VCP** 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.

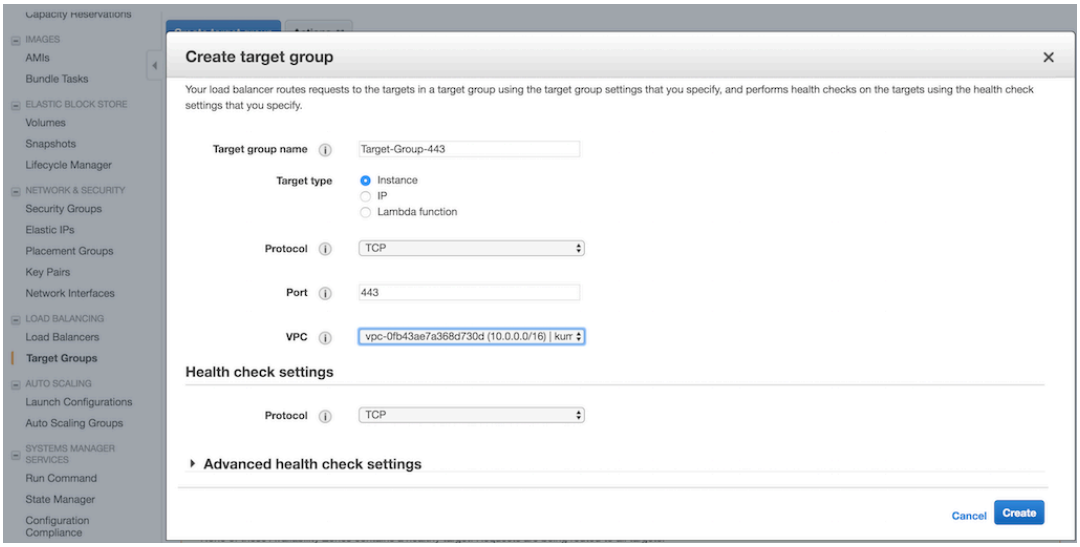
c) From the **Protocol** list, select **TCP**.

d) In the **Port** field, enter **443**.

e) From the **VCP** list, select 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.



5. Once you have created the target groups, you must register the target instances.

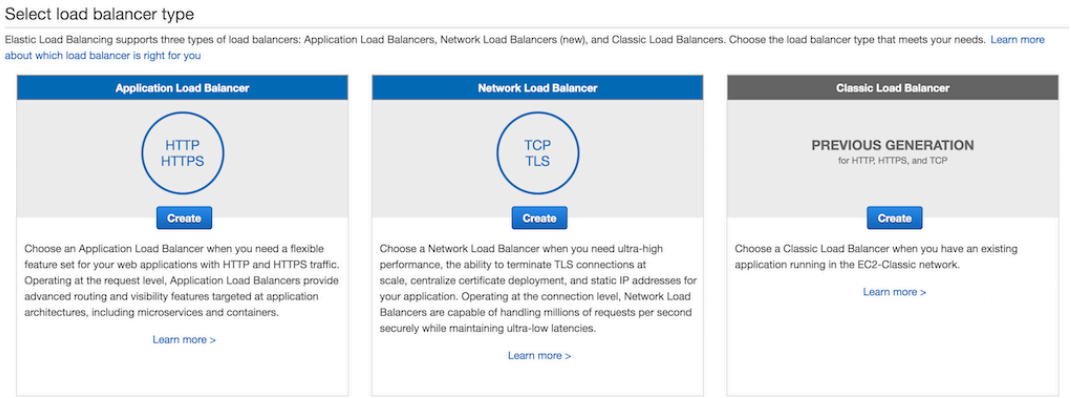
   a) Select the created target group in the list page, click the **Target** tab, and select **edit**.
   b) In the **Instances** tab, select the two NetScaler VPX instances and click **Add to registered**.

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.

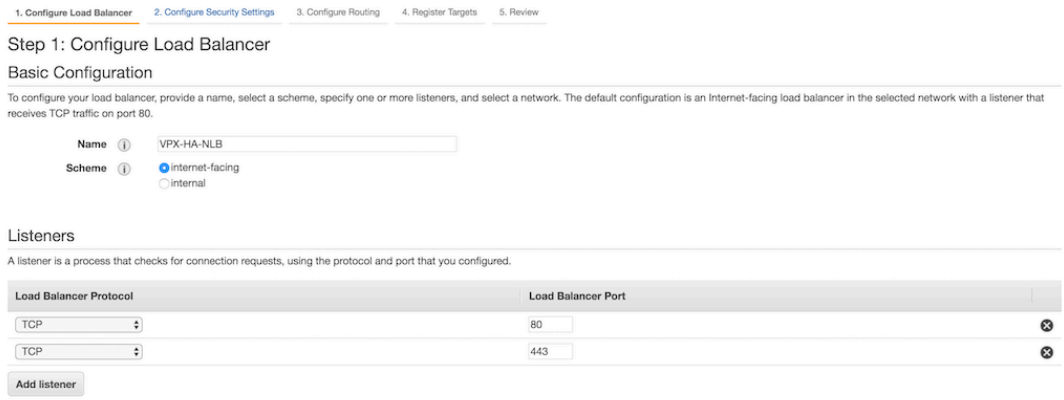


8. In the **Configure Load Balancer** page, do the following:

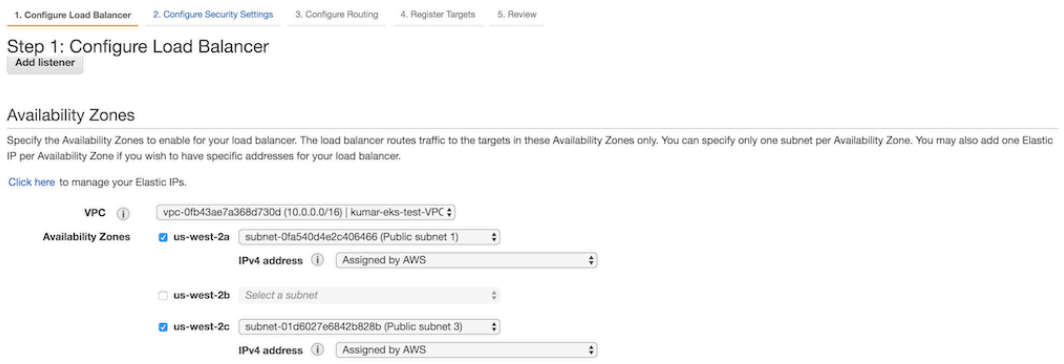
   a) In the **Name** field, enter a name for the load balancer.

   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:



   d) In the Availability Zones section, select the VPC, availability zones, and subnets where the NetScaler VPX instances are deployed.



9. In the **Configure routing** page. do the following:

   a) In the **Target group** list, click **Existing target group**.

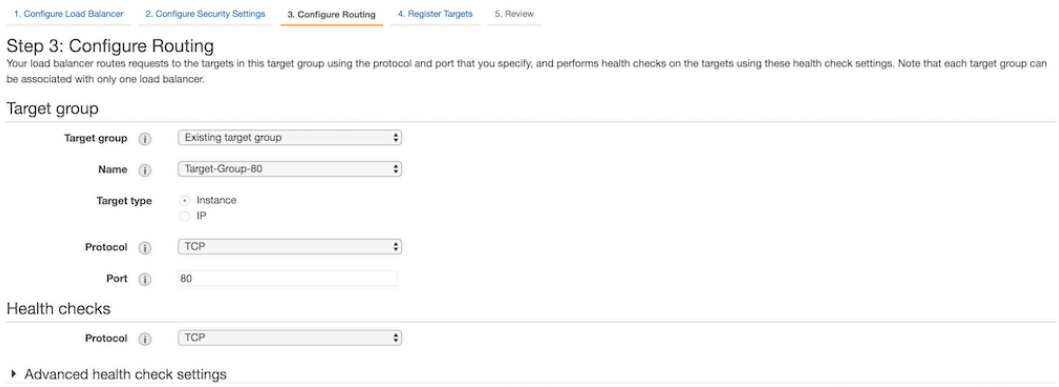
   b) In the **Name** field, enter **Target-Group-80**.

   c) In the **Target type** field, select **Instance**.

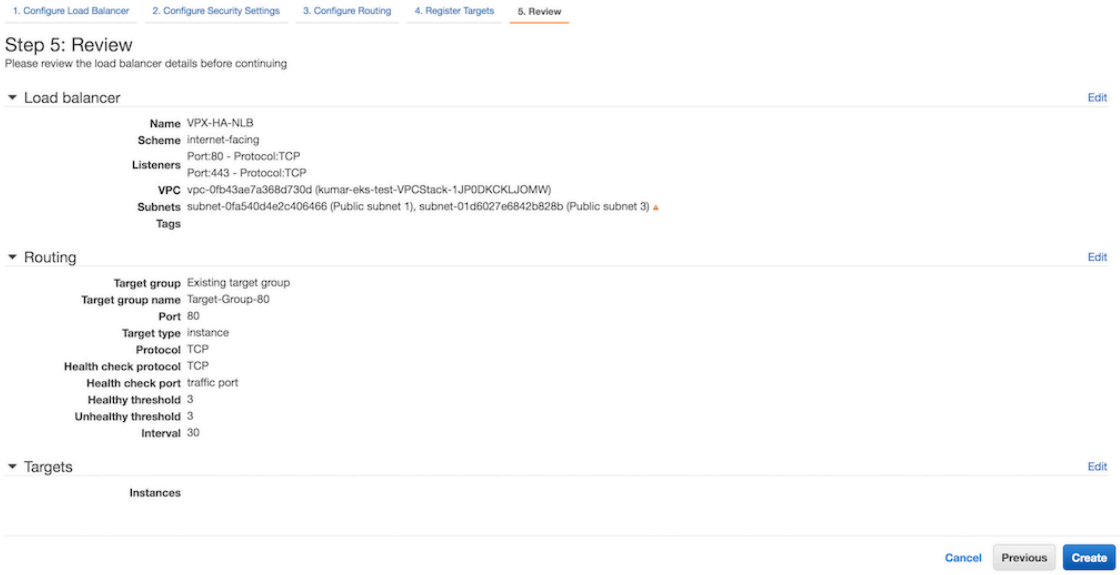
   d) In the **Protocol** list, select **TCP**.

   e) In the **Port** field, enter **80**.

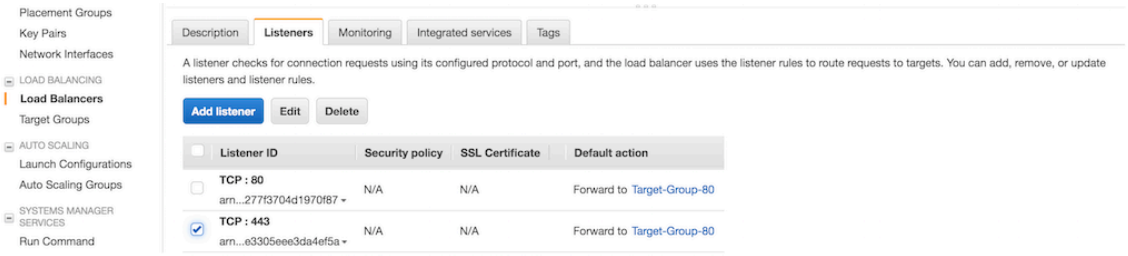
   f) Select **TCP** from the **Protocol** list in the **Health checks** section as shown in the following image:



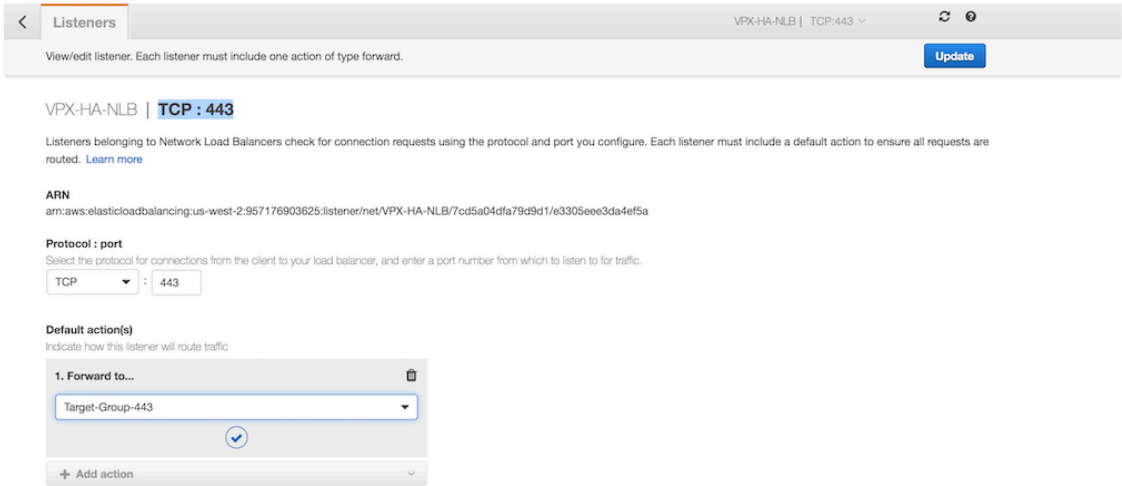
10. In the **Review** page, review your configuration and click **Create**.



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**.



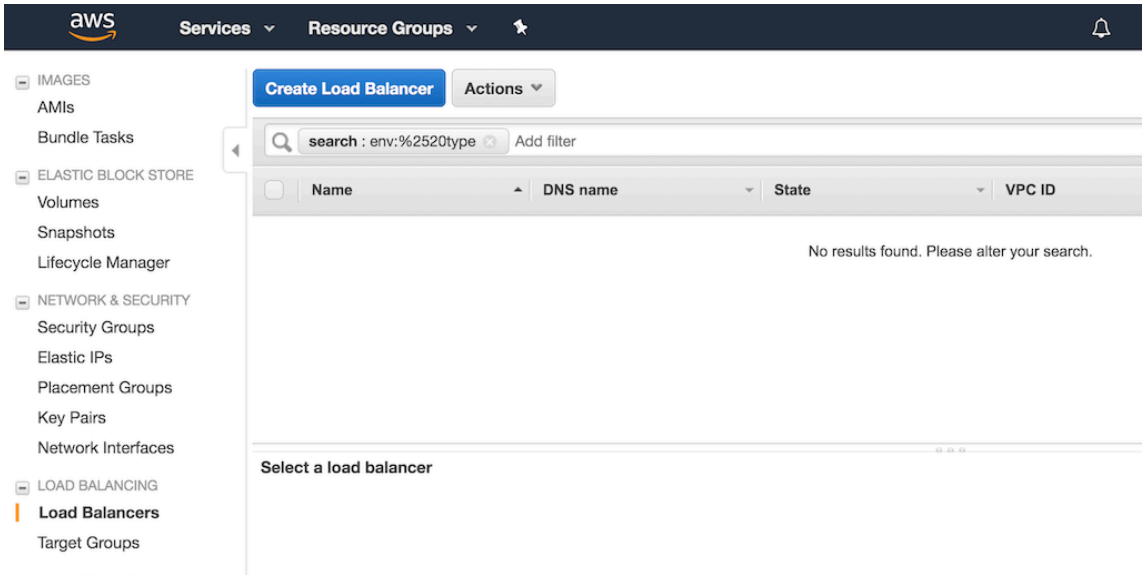
12. In the **Listeners** page, delete the default action and then select **Target-Group-443** in the **Forward to** list.



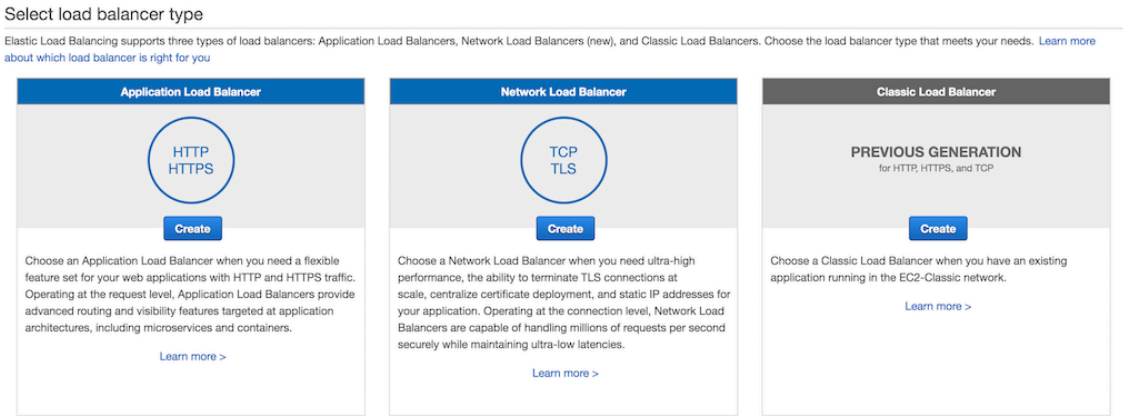
13. Click **Update**.

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.

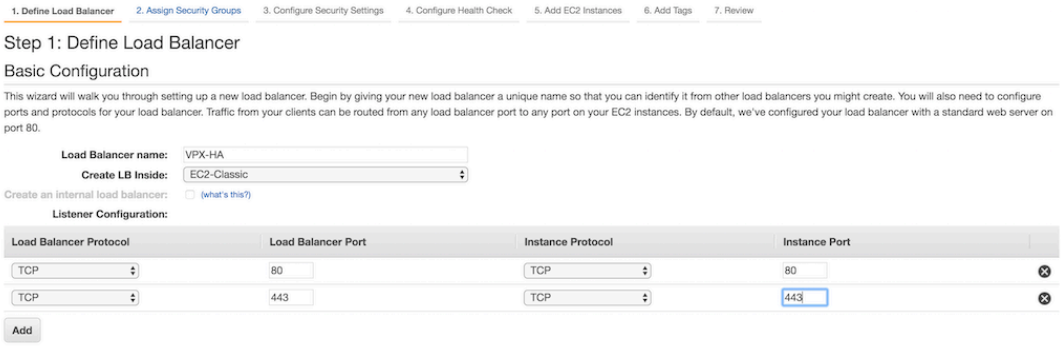


4. In the **Define Load Balancer** page, do the following:

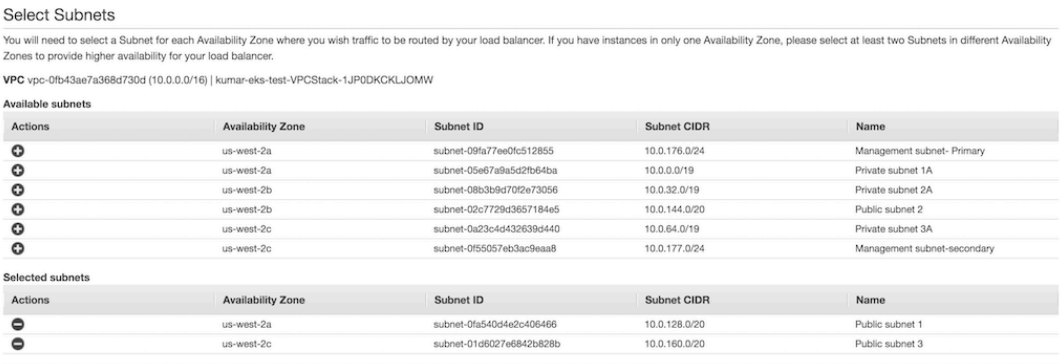
   a) In the **Load Balancer name** field, enter a name for the load balancer.

   b) In the **Create LB Inside** list, select your NetScaler VPX.

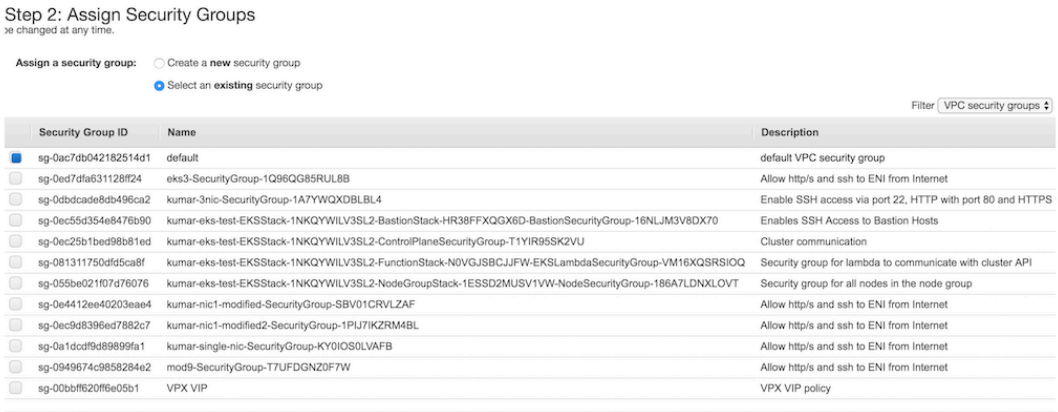
   c) In the **Listener Configuration** section, click **Add** and add two entries with TCP as the load balancer protocol and `80` and `443` as the load balancer port respectively. Also, select TCP as instance protocol and `80` and `443` as the instance port respectively as shown in the following image:



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 NetScaler VPX instances.



e) In the **Assign Security Groups** page, select a security group for the ELB instance. The security group can be same as the security group attached to NetScaler 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 NetScaler VPX security group from the ELB security group and conversely.



f) In the **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.

g) In the **Add EC2 Instances** page, select two NetScaler VPX instances that were deployed earlier.

```
![Classic ADD EC2 Instances](/en-us/netscaler-k8s-ingress-controller/media/classic-add-ec2.png)
```

h) In the Add Tags page, add tags as per your requirement.

i) In the Review page, review your configurations.

j) Click **Create**.

Verify the solution

After you have successfully deployed NetScaler VPX, AWS ELB, and NetScaler Ingress Controller, you can verify the solution using a sample service.

Perform the following:

1. Deploy a sample service and ingress using app.yaml.

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

2. Log on to the NetScaler VPX instance and verify if the Content Switching vserver are successfully configured on both the NetScaler VPX instance. Do the following:

   a) Log on to the NetScaler VPX instance. Perform the following:

      i. Use an SSH client, such as PuTTy, to open an SSH connection to the NetScaler 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:

```
    sh cs vserver
```

   Output:


```
1     1) k8s-10.0.139.87:80:http (10.0.139.87:80) - HTTP Type:
         CONTENT
2             State: UP
3             Last state change was at Fri Apr 12 14:24:13 2019
4             Time since last state change: 3 days, 03:09:18.920
5             Client Idle Timeout: 180 sec
6             Down state flush: ENABLED
7             Disable Primary Vserver On Down : DISABLED
8             Comment: uid=
                 NNJRYQ54VM2KWCXOERK6HRJHR4VEQYRI7U3W4BNFQLTIAENMTHWA
                 ====
9             Appflow logging: ENABLED
10            Port Rewrite : DISABLED
11            State Update: DISABLED
12            Default:  Content Precedence: RULE
13            Vserver IP and Port insertion: OFF
14            L2Conn: OFF Case Sensitivity: ON
15            Authentication: OFF
16            401 Based Authentication: OFF
17            Push: DISABLED Push VServer:
18            Push Label Rule: none
19            Listen Policy: NONE
20            IcmpResponse: PASSIVE
21            RHIstate:  PASSIVE
22            Traffic Domain: 0
```

c) Access the application `test.example.com` using the DNS name of the ELB instance.

```
1     # curl -H 'Host: test.example.com' <DNS name of the ELB>
```

Example:

```
1     % curl -H 'Host: test.example.com' http://VPX-HA
         -829787521.us-west-2.elb.amazonaws.com
```

d) To delete the deployment, use the following command:

```
1     kubectl delete -f app.yaml
```

Troubleshooting

Problem	Resolution
CloudFormation stack failure	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.

Problem	Resolution
NetScaler Ingress Controller unable to communicate with the NetScaler VPX instances.	Ensure that user name and password is correct in `citrix-ingress-controller.yaml` file. Ensure that the NetScaler VPX security group allows the traffic on port `80` and `443` from the EKS node group security group.
The services are DOWN in the NetScaler VPX instances.	Ensure that the NetScaler VPX traffic can reach the EKS cluster. Modify the security group of EKS node group to allow traffic from NetScaler VPX security group.
Traffic not routing to NetScaler VPX instance from ELB.	Ensure that security group of NetScaler VPX allows traffic from the ELB security group.

Deploy the NetScaler Ingress Controller for NetScaler with admin partitions

December 31, 2023

NetScaler Ingress Controller is used to automatically configure one or more NetScaler based on the Ingress resource configuration. The ingress NetScaler appliance (MPX or VPX) can be partitioned into logical entities called admin partitions, where each partition can be configured and used as a separate NetScaler appliance. For more information, see Admin Partition. NetScaler Ingress Controller can also be deployed to configure NetScaler with admin partitions.

For NetScaler with admin partitions, you must deploy a single instance of NetScaler Ingress Controller for each partition. And, the partition must be associated with a partition user specific to the NetScaler Ingress Controller instance.

> **Note:**
>
> NetScaler Metrics Exporter supports exporting metrics from the admin partitions of NetScaler.

Prerequisites

Ensure that:

- Admin partitions are configured on the NetScaler appliance. For instructions see, Configure admin partitions.

- Create a partition user specifically for the NetScaler Ingress Controller. NetScaler Ingress Controller configures the NetScaler using this partition user account. Ensure that you do not associate this partition user to other partitions in the NetScaler appliance.

> **Note:**
>
> For SSL-related use cases in the admin partition, ensure that you use NetScaler version 12.0–56.8 and above.

To deploy the NetScaler Ingress Controller for NetScaler with admin partitions

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:

Environment Variable	Mandatory or Optional	Description
NS_IP	Mandatory	The IP address of the NetScaler appliance. For more details, see Prerequisites.
NS_USER and NS_PASSWORD	Mandatory	The user name and password of the partition user that you have created for the NetScaler Ingress Controller. For more details, see Prerequisites.
NS_VIP	Mandatory	NetScaler Ingress Controller uses the IP address provided in this environment variable to configure a virtual IP address to the NetScaler that receives the Ingress traffic. **Note:** NS_VIP acts as a fallback when the frontend-ip annotation is not provided in Ingress YAML. Only Supported for Ingress.

Environment Variable	Mandatory or Optional	Description
NS_SNIPS	Optional	Specifies the SNIP addresses on the NetScaler appliance or the SNIP addresses on a specific admin partition on the NetScaler appliance.
NS_ENABLE_MONITORING	Mandatory	Set the value `Yes` to monitor NetScaler. **Note:** Ensure that you disable NetScaler monitoring for NetScaler with admin partitions. Set the value to `No`.
EULA	Mandatory	The End User License Agreement. Specify the value as `Yes`.
Kubernetes_url	Optional	The kube-apiserver url that NetScaler Ingress Controller uses to register the events. If the value is not specified, NetScaler Ingress Controller uses the internal kube-apiserver IP address.
LOGLEVEL	Optional	The log levels to control the logs generated by NetScaler 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
NS_PROTOCOL and NS_PORT	Optional	Defines the protocol and port that must be used by the NetScaler Ingress Controller to communicate with NetScaler. By default, the NetScaler Ingress Controller uses HTTPS on port 443. You can also use HTTP on port 80.

Environment Variable	Mandatory or Optional	Description
ingress-classes	Optional	If multiple ingress load balancers are used to load balance different ingress resources. You can use this environment variable to specify the NetScaler Ingress Controller to configure NetScaler associated with a specific ingress class. For information on Ingress classes, see Ingress class support

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 the NetScaler Ingress Controller is deployed successfully using the following command:

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

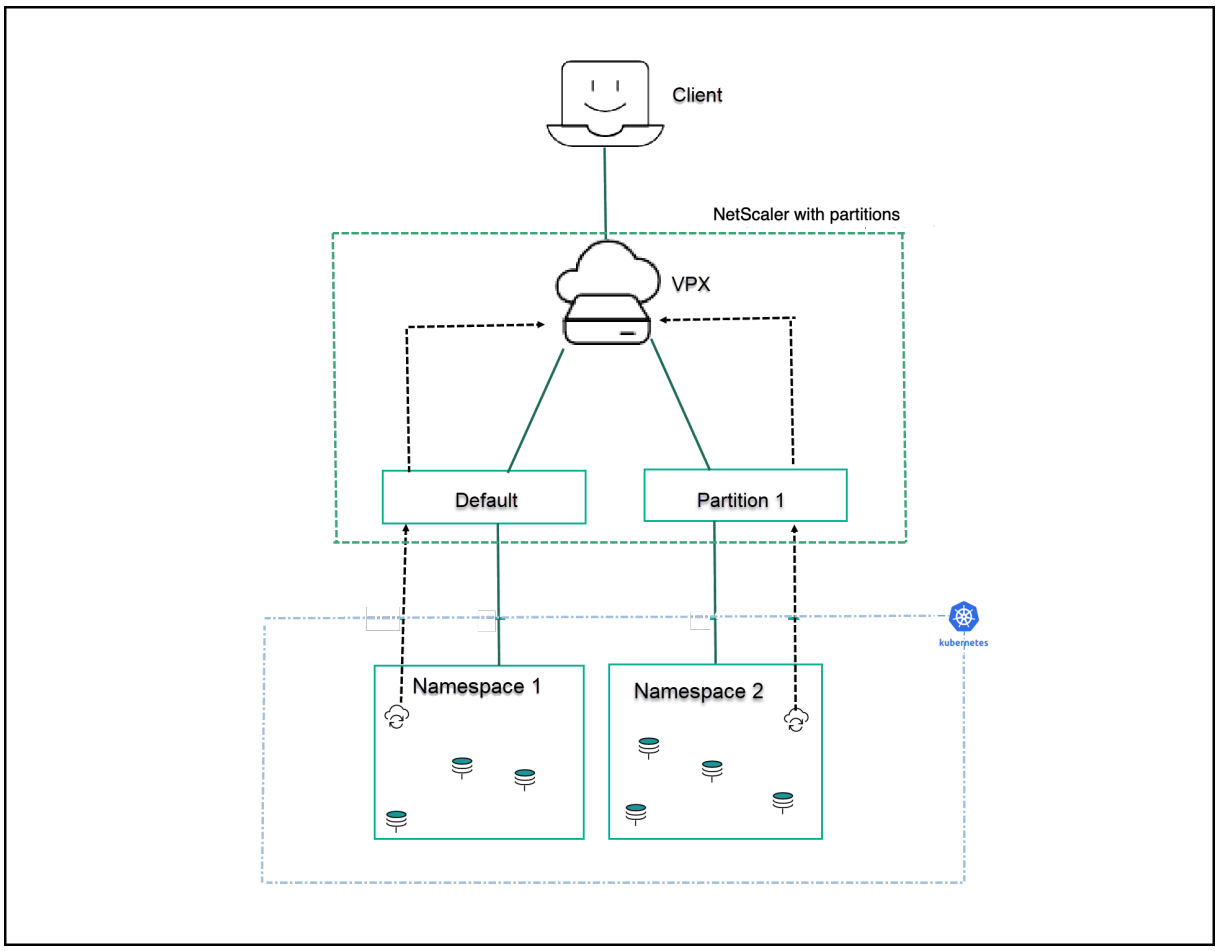
Use case: How to securely deliver multitenant microservice-based applications using NetScaler admin partitions

You can isolate ingress traffic between different microservice based applications with the NetScaler admin partition using NetScaler Ingress Controller. NetScaler admin partition enables multitenancy at the software level in a single NetScaler instance. Each partition has its own control plane and network plane.

You can deploy one instance of NetScaler 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 NetScaler Ingress Controller separately in both of these namespaces.

NetScaler Ingress Controller instances provide configuration instructions to the respective NetScaler 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 teams want to manage their workload independently and do not want to share resources. NetScaler 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 NetScaler appliance. For instructions see, Configure admin partitions.
- Ensure that you create a partition user account specifically for the NetScaler Ingress Controller. NetScaler Ingress Controller configures the NetScaler using this partition user account. Ensure that you do not associate this partition user to other partitions in the NetScaler 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 parti-

tion.

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 NetScaler 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:

```
kubectl create namespace ns1
kubectl create namespace ns2
```

Configurations in namespace ns1

1. Deploy the `apache` application in `ns1`.

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: ns1

---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: apache-ns1
  name: apache-ns1
  namespace: ns1
spec:
  replicas: 2
  selector:
    matchLabels:
      app: apache-ns1
  template:
    metadata:
      labels:
        app: apache-ns1
    spec:
      containers:
      - image: httpd
        name: httpd
---

apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
```



```
  labels:
    app: apache-ns1
  name: apache-ns1
  namespace: ns1
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: apache-ns1
```

2. Deploy NetScaler Ingress Controller in `ns1`.

   You can use the YAML file to deploy NetScaler Ingress Controller or use the Helm chart.

   Ensure that you use the user credentials that are bound to the default partition.

```
helm 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.

```
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:
      - backend:
          service:
            name: apache-ns1
            port:
              number: 80
        pathType: Prefix
        path: /index.html
```

4. NetScaler Ingress Controller in `ns1` configures the ADC entities in the default partition.

Configurations in namespace ns2

1. Deploy `guestbook` application in `ns2`.

```
apiVersion: v1
```



```yaml
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
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:
  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, then to
          # instead access an environment variable to find the master
          # service's host, comment out the 'value: dns' line above, and
          # uncomment the line below:
```



```
103             # value: env
104           ports:
105           - containerPort: 6379
106 ---
107 apiVersion: v1
108 kind: Service
109 metadata:
110   name: frontend
111   namespace: ns2
112   labels:
113     app: guestbook
114     tier: frontend
115 spec:
116   # if your cluster supports it, uncomment the following to
        automatically create
117   # an external load-balanced IP for the frontend service.
118   # type: LoadBalancer
119   ports:
120   - port: 80
121   selector:
122     app: guestbook
123     tier: frontend
124 ---
125 apiVersion: apps/v1 #  for k8s versions before 1.9.0 use apps/
      v1beta2  and before 1.8.0 use extensions/v1beta1
126 kind: Deployment
127 metadata:
128   name: frontend
129   namespace: ns2
130 spec:
131   selector:
132     matchLabels:
133       app: guestbook
134       tier: frontend
135   replicas: 3
136   template:
137     metadata:
138       labels:
139         app: guestbook
140         tier: frontend
141     spec:
142       containers:
143       - name: php-redis
144         image: gcr.io/google-samples/gb-frontend:v4
145         resources:
146           requests:
147             cpu: 100m
148             memory: 100Mi
149         env:
150         - name: GET_HOSTS_FROM
151           value: dns
152           # If your cluster config does not include a dns service,
                then to
```



```
153            # instead access environment variables to find service
                 host
154            # info, comment out the 'value: dns' line above, and
                 uncomment the
155            # line below:
156            # value: env
157          ports:
158          - containerPort: 80
```

2. Deploy NetScaler Ingress Controller in namespace `ns2`.

   Ensure that you use the user credentials that are bound to the partition `p1`.

```
1 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.

```
1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    annotations:
5    kubernetes.io/ingress.class: citrix-adm-part-ns2
6    ingress.citrix.com/frontend-ip: "<VIP in partition 1>"
7    name: guestbook-ingress
8    namespace: ns2
9  spec:
10   rules:
11   - host: www.guestbook.com
12     http:
13       paths:
14       - backend:
15           service:
16             name: frontend
17             port:
18               number: 80
19         path: /
20         pathType: Prefix
```

4. NetScaler Ingress Controller in `ns2` configures the ADC entities in partition `p1`.

Deploy Citrix solution for service of type LoadBalancer in AWS

December 31, 2023

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 LoadBal-

ancer, 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 NetScaler to directly load balance and expose a service instead of the cloud managed load balancer. NetScaler 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, allocates the IP address and that IP address is the VIP of NetScaler VPX. NetScaler Ingress Controller, deployed in a Kubernetes cluster, configures a NetScaler deployed outside the cluster to load balance the incoming traffic. So, the service is accessed through NetScaler 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.

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 NetScaler VPX and EKS are deployed and running in the same VPC. For information about creating NetScaler VPX in AWS, see Create a NetScaler 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.

1. Download the unified-lb-values.yaml file and edit the YAML file for specifying the following details:

    - NetScaler VPX NSIP. For more information, see NetScaler Ingress Controller Helm chart.
    - Secret created using the NetScaler VPX credentials. For more information, see NetScaler Ingress Controller Helm chart.

- List of VIPs to be used in IPAM controller. For more information, see IPAM Helm chart.

2. Deploy and NetScaler Ingress Controller on your Amazon EKS cluster using the edited YAML file. Use the following commands:

```
helm repo add citrix https://citrix.github.io/citrix-helm-charts/

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:

```
beta.kubernetes.io/aws-load-balancer-type: "external"
```

   b) Deploy the application and service with the modified annotation using the following command:

```
kubectl create -f https://github.com/citrix/citrix-k8s-ingress
    -controller/blob/master/docs/how-to/typeLB/aws/guestbook-
    all-in-one-lb.yaml
```

   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 NetScaler 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.

   - NetScaler VPX NSIP
   - Secret created using the NetScaler VPX credentials. For information about creating the secret, see Create a secret.
   - Specify the argument for :

```
 args:
   - --ipam
     citrix-ipam-controller
```


2. Deploy the NetScaler Ingress Controller using the modified YAML.

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

3. Deploy the NetScaler VIP CRD which enables communication between the NetScaler 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 NetScaler 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 NetScaler 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.

```
kubectl create -f https://github.com/citrix/citrix-k8s-ingress-controller/blob/master/docs/how-to/typeLB/aws/guestbook-all-in-one-lb.yaml
```

> **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 NetScaler VPX.

   d) Access the application using a browser. For example, `http://EIP-associated-with-vip`.

Multi-cloud and GSLB solution with Amazon EKS and Microsoft AKS clusters

December 31, 2023

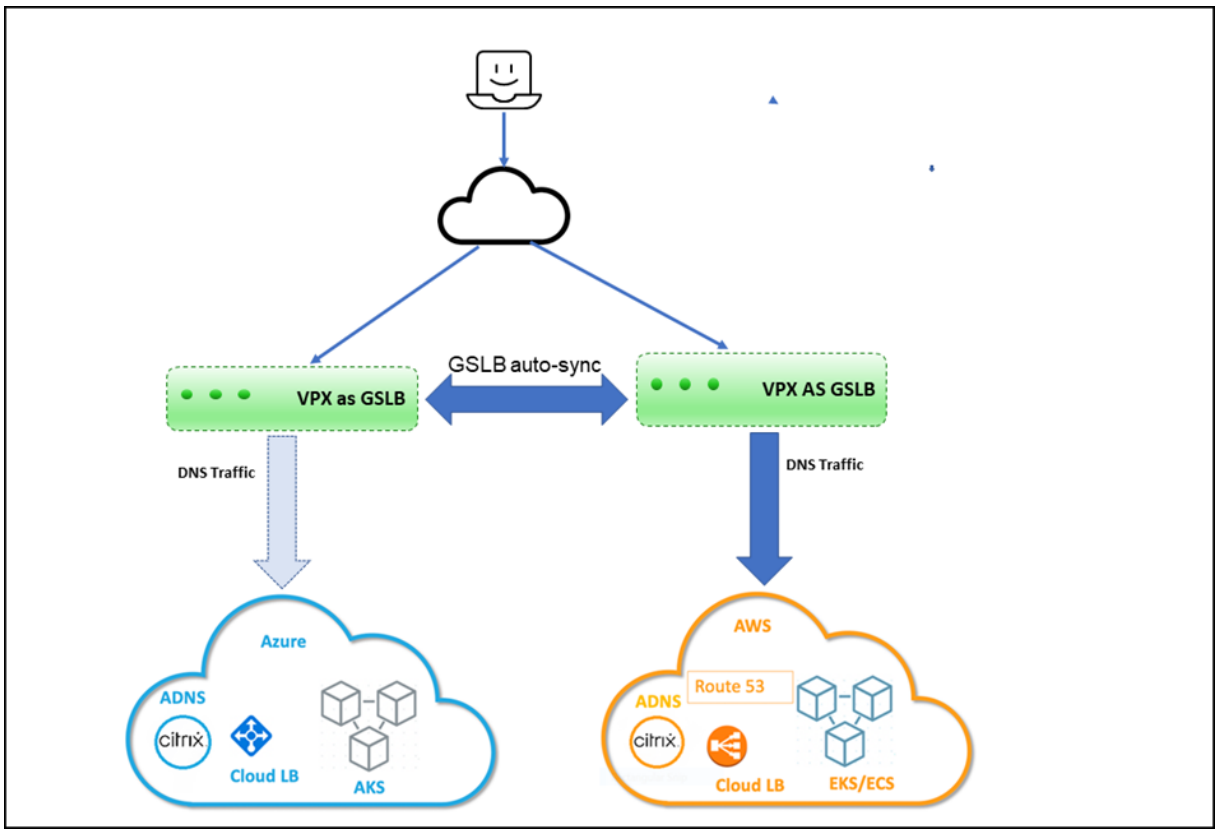
You can deploy multiple instances of the same application across multiple clouds provided by different cloud providers. This multi-cloud strategy helps you to ensure resiliency, high availability, and proximity. A multi-cloud approach also allows you to take advantage of the best of each cloud provider by reducing the risks such as vendor lock-in and cloud outages.

NetScaler with the help of the NetScaler Ingress Controller can perform multi-cloud load balancing. NetScaler 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 GSLB 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 NetScaler and NetScaler networking.
- Instances of the same application must be deployed in Kubernetes clusters on Amazon EKS and Microsoft AKS.

To deploy the multi-cloud and GSLB solution, you must perform the following tasks.

1. Deploy NetScaler VPX in AWS.
2. Deploy NetScaler VPX in Azure.
3. Configure ADNS service on NetScaler VPX deployed in AWS and AKS.
4. Configure GSLB service on NetScaler VPX deployed in AWS and AKS.
5. Apply GTP and GSE CRDs on AWS and Azure Kubernetes clusters.
6. Deploy the GSLB controller.

Deploying NetScaler VPX in AWS

You must ensure that the NetScaler VPX instances are installed in the same virtual private cloud (VPC) on the EKS cluster. It enables NetScaler VPX to communicate with EKS workloads. You can use an existing EKS subnet or create a subnet to install the NetScaler VPX instances.

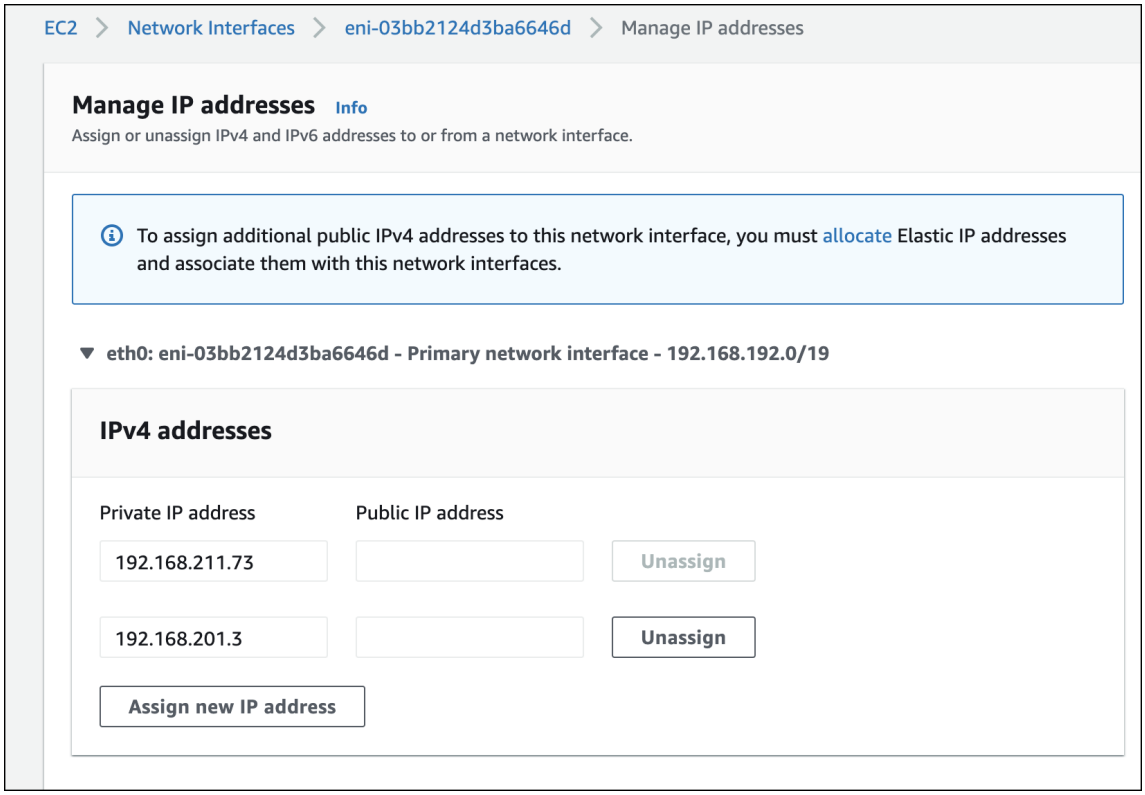
Also, you can install the NetScaler VPX instances in a different VPC. In that case, you must ensure that the VPC for EKS can communicate using VPC peering. For more information about VPC peering, see VPC peering documentation.

For high availability (HA), you can install two instances of NetScaler VPX in HA mode.

1. Install NetScaler VPX in AWS. For information on installing NetScaler VPX in AWS, see Deploy NetScaler VPX instance on AWS.

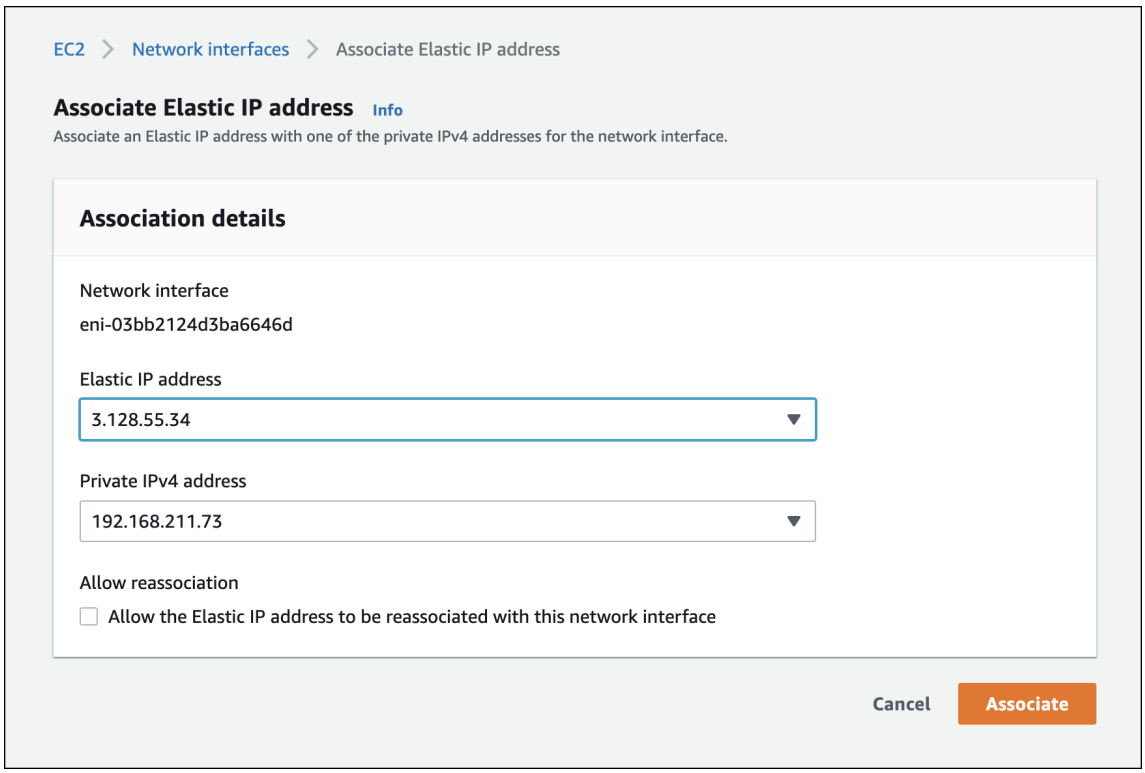
   NetScaler VPX requires a secondary public IP address other than the NSIP to run GSLB service sync and ADNS service.

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 NetScaler 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 NetScaler 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 NetScaler VPX in High Availability (HA) mode, you must perform this configuration in both of the NetScaler VPX instances.

5. Enable Content Switching (CS), Load Balancing (LB), Global Server Load Balancing(GSLB), and SSL features in NetScaler 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 NetScaler 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 NetScaler VPX using the following command:

```
add nameserver *nameserver IP*
```

Deploying NetScaler VPX in Azure

You can run a standalone NetScaler VPX instance on an AKS cluster or run two NetScaler VPX instances in High Availability mode on the AKS cluster.

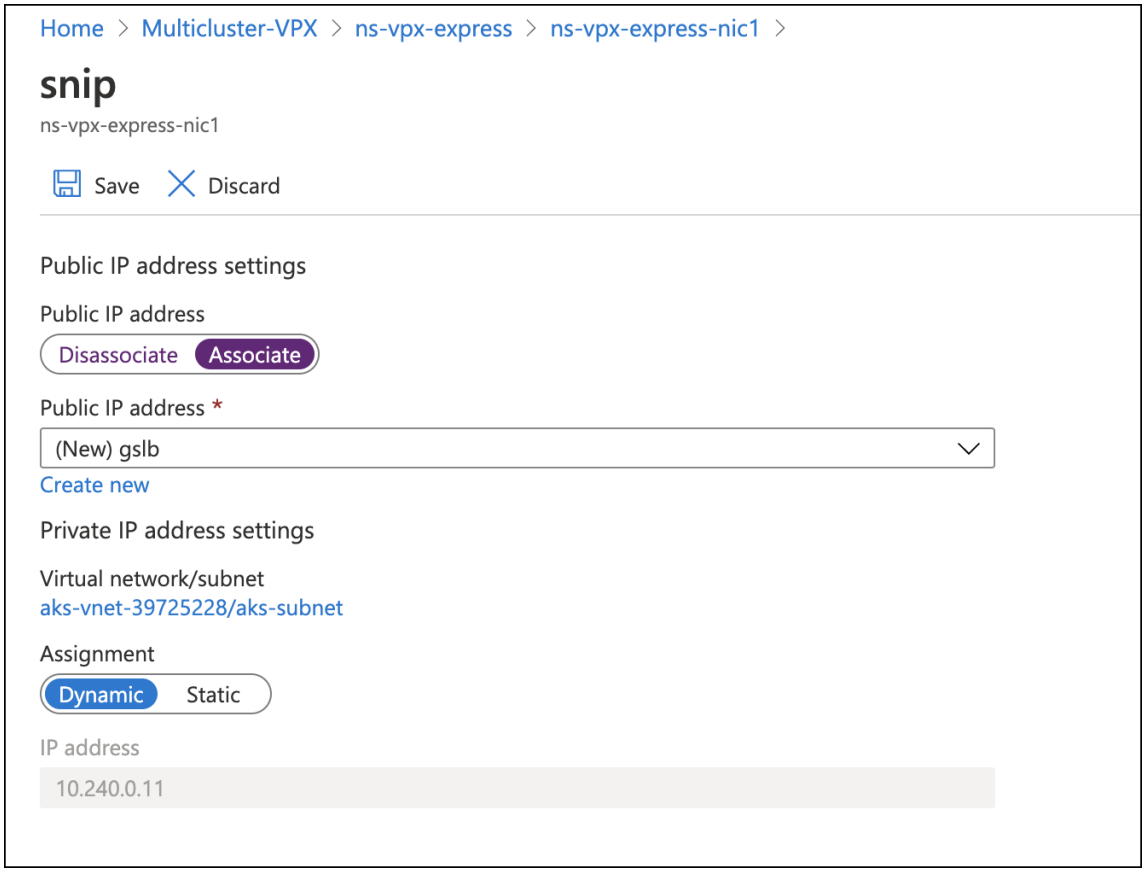
While installing, ensure that the AKS cluster must have connectivity with the VPX instances. To ensure the connectivity, you can install the NetScaler VPX in the same virtual network (VNet) on the AKS cluster in a different resource group.

While installing the NetScaler VPX, select the VNet where the AKS cluster is installed. Alternatively, you can use VNet peering to ensure the connectivity between AKS and NetScaler VPX if the VPX is deployed in a different VNet other than the AKS cluster.

1. Install NetScaler VPX in AWS. For information on installing NetScaler VPX in AKS, see Deploy a NetScaler 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 NetScaler 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 NetScaler VPX using the following command:

```
enable feature *feature*
```

To access the NetScaler 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 NetScaler 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 NetScaler VPX using the following command:

```
add nameserver *nameserver IP*
```

Configure ADNS service in NetScaler VPX deployed in AWS and Azure

The ADNS service in NetScaler 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 NetScaler 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 NetScaler 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.

Configure GSLB service in NetScaler VPX deployed in AWS and Azure

You must create GSLB sites on NetScaler VPX deployed on AWS and Azure.

1. Log in to AWS NetScaler 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:

```
add gslb site aws_site 192.168.197.18 -publicIP 3.139.156.175

add gslb site azure_site 10.240.0.11 -publicIP 23.100.28.121
```


2. Log in to Azure NetScaler VPX and configure GSLB sites. For example:

```
add gslb site aws_site 192.168.197.18 -publicIP 3.139.156.175

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:

```
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 NetScaler for performing GSLB in Kubernetes applications. These CRDs are designed for configuring NetScaler GSLB controller for applications deployed in distributed Kubernetes clusters.

GTP CRD

The GTP CRD accepts the parameters for configuring GSLB on the NetScaler 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:

```
kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-
    ingress-controller/master/gslb/Manifest/gtp-crd.yaml
```

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 GSLB endpoints of NetScalers 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:


```
kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/gslb/Manifest/gse-crd.yaml
```

Deploy GSLB controller

GSLB controller helps you to ensure the high availability of the applications across clusters in a multi-cloud environment.

You can install the GSLB controller on the AWS and Azure clusters. GSLB controller listens to GTP and GSE CRDs and configures the NetScaler for GSLB that provides high availability across multiple regions in a multi-cloud environment.

To deploy the GSLB controller, perform the following steps:

1. Create an RBAC for the GSLB controller on the AWS and Azure Kubernetes clusters.

   ```
   kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/gslb/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 NetScaler 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*.


```
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:
   name: secret-1
   key: username
name: "azure_site_password"
valueFrom:
  secretKeyRef:
   name: secret-1
   key: password
name: "aws_site_username"
valueFrom:
  secretKeyRef:
   name: secret-1
   key: username
name: "aws_site_password"
valueFrom:
  secretKeyRef:
   name: secret-1
   key: password
```

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"
name: "azure_site_username"
valueFrom:
  secretKeyRef:
   name: secret-1
   key: username
name: "azure_site_password"
valueFrom:
  secretKeyRef:
   name: secret-1
   key: password
name: "aws_site_username"
valueFrom:
  secretKeyRef:
   name: secret-1
   key: username
name: "aws_site_password"
valueFrom:
  secretKeyRef:
   name: secret-1
   key: password
```

> **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
NAME     TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)
           AGE
apache   LoadBalancer   10.0.16.231   20.62.235.193   80:32666/TCP
       3m2s
```

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_isntance.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:
     {
  }

<!--NeedCopy-->
```

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:

```
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:


```
kubectl get gse
kubectl get gse *name* -o yaml
```

Also, log in to NetScaler VPX and verify that the GSLB configuration is successfully created using the following command:

```
show gslb runningconfig
```

As the GTP CRD is configured for the traffic policy as `FAILOVER`, NetScaler VPX instances serve the traffic from the primary cluster (EKS cluster in this example).

```
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.

NetScaler 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 NetScaler VPX as GSLB and the same NetScaler VPX as ingress device too.

To deploy the multi-cloud multi-cluster ingress and load balancing with NetScaler VPX as the ingress device, you must complete the following tasks described in the previous sections:

1. Deploy NetScaler VPX in AWS
2. Deploy NetScaler VPX in Azure
3. Configure ADNS service on NetScaler VPX deployed in AWS and AKS
4. Configure GSLB service on NetScaler VPX deployed in AWS and AKS
5. Apply GTP and GSE CRDs on AWS and Azure Kubernetes clusters
6. Deploy the GSLB controller

After completing the preceding tasks, perform the following tasks:

1. Configure NetScaler VPX as Ingress Device for AWS
2. Configure NetScaler VPX as Ingress Device for Azure

Configure NetScaler VPX as Ingress device for AWS

Perform the following steps:

1. Create NetScaler VPX login credentials using Kubernetes secret

```
kubectl create secret  generic nslogin --from-literal=username='
    nsroot' --from-literal=password='<instance-id-of-vpx>'
```


The NetScaler VPX password is usually the instance-id of the VPX if you have not changed it.

2. Configure SNIP in the NetScaler VPX by connecting to the NetScaler VPX using SSH. SNIP is the secondary IP address of Citrix a VPX to which the elastic IP address is not assigned.

```
add ns ip 192.168.84.93 255.255.224.0
```

This step is required for NetScaler to interact with the pods inside the Kubernetes cluster.

3. Update the NetScaler VPX management IP address and VIP in the NetScaler Ingress Controller manifest.

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

> **Note:**
>
> If you do not have `wget` installed, you can use `fetch` or `curl`.

4. Update the primary IP address of NetScaler VPX in the `cic.yaml` in the following field.

```
# Set NetScaler NSIP/SNIP, SNIP in case of HA (mgmt has to be enabled)
 - name: "NS_IP"
   value: "X.X.X.X"
```

5. Update the NetScaler 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

```
# Set NetScaler VIP for the data traffic
- name: "NS_VIP"
  value: "X.X.X.X"
```

6. Once you have edited the YAML file with the required values deploy NetScaler Ingress Controller.

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

Configure NetScaler VPX as Ingress device for Azure

Perform the following steps:

1. Create NetScaler VPX login credentials using Kubernetes secrets.

```
kubectl create secret  generic nslogin --from-literal=username='<azure-vpx-instance-username>' --from-literal=password='<azure-vpx-instance-password>'
```


> **Note:**
>
> The NetScaler VPX user name and password should be the same as the credentials set while creating NetScaler VPX on Azure.

2. Using SSH, configure a SNIP in the NetScaler VPX, which is the secondary IP address of the NetScaler VPX. This step is required for the NetScaler 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 NetScaler VPX instance creation.
   - `vpx-instance-primary-ip-subnet` is the subnet of the primary private IP address of the NetScaler VPX instance.

   To verify the subnet of the private IP address, SSH into the NetScaler VPX instance and use the following command.

```
show ip <primary-private-ip-addess>
```

3. Update the NetScaler VPX image URL, management IP address, and VIP in the NetScaler Ingress Controller YAML file.

   a) Download the NetScaler 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 NetScaler 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 NetScaler 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 NetScaler VPX VIP in the `cic.yaml` with the private IP address of the VIP assigned during VPX Azure instance creation.

```
# Set NetScaler VIP for the data traffic
- name: "NS_VIP"
  value: "X.X.X.X"
```

4. Once you have configured NetScaler Ingress Controller with the required values, deploy the NetScaler Ingress Controller using the following command.

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

Annotations

April 7, 2024

Ingress annotations

The following ingress annotations are supported by NetScaler:

Annotations	Type	Required	Description	Default	Possible value
`ingress.citrix.com/frontend-ip`	String	Optional	Specify an IP address that needs to be used as the content switching virtual server IP address. **Note:** There are multiple ways to configure content switching virtual server. `ingress.citrix.com/frontend-ip` such as IPAM configuration, default nsVIP, and so on.	NA	Numeric IP address. For example, `1.2.3.4`
`ingress.citrix.com/frontend-ipset-name`	String	Optional	Specify the IPSET name that needs to be bound to the content switching virtual server. Use this annotation along with `ingress.citrix.com/frontend-ip`. **Note:** If you specify in the IPSET name	NA	NetScaler IPSET entity name
`ingress.citrix.com/insecure-service-type`	String	Optional	Specify the protocols among HTTP/TCP/UDP/sip_tcp/sip_udp/any for content switching virtual server.	`http`	`http`, `tcp`, `udp`, `sip_udp`, or `any`

Annotations	Type	Required	Description	Default	Possible value
`ingress.citrix.com/insecure-port`	String	Optional	Configure the port for content switching virtual server for http/tcp/udp/sip_udp/any protocols.	`80`	Valid port number
`ingress.citrix.com/secure-service-type`	String	Optional	Specify the protocols among SSL/SSL_TCP as the protocol for content switching virtual server.	`ssl`	`ssl`, `ssl_tcp`
`ingress.citrix.com/secure-port`	String	Optional	Configure the port for content switching virtual server for HTTPS traffic.	`443`	Valid port number

Annotations	Type	Required	Description	Default	Possible value
`ingress.citrix.com/insecure-termination`	String	Optional	Configure the behavior for HTTP traffic. Use `allow` to allow HTTP traffic; use `redirect` to redirect the HTTP request to HTTPS; or use `disallow` if you want to drop the HTTP traffic.	`disallow`	`allow`, `redirect`, or `disallow`
`ingress.citrix.com/default-response-code`	String	Optional	Configure NetScaler to trigger an HTTP response code when a request lands on NetScaler and if any of the following conditions are met for all the backend services defined in the ingress resource: 1. None of the content policies match. 2. All the backend service endpoints are down.	NA	Possible HTTP response codes are `404` and `503`.
`ingress.citrix.com/secure-backend`	String/JSON	Optional	Specify the certificate that you want to use for the backend application. If the value provided is either true or false, it applies for all the services in the ingress content switching policies.	`False`	As string: `True`/`False`. As JSON: `{ "<Service_Name>": "True/False", ... } '`
`ingress.citrix.com/backend-secret`	String/JSON	Optional	Specify the secret between NetScaler and the backend Kubernetes communication. If the value provided is either true or false, it applies for all the services in the ingress content switching policies. **Example:** `ingress.citrix.com`	NA	As string: `"<Kubernetes secret>"`. As JSON: `{ "<Service_Name>": "<kubernetes secret>", ... } '`

Annotations	Type	Required	Description	Default	Possible value
`ingress.citrix.com/backend-ca-secret`	String/JSON	Optional	Specify the CA certificate that you want to use for backend communication between NetScaler and Kubernetes pods.	NA	As string: `"kubernetes secret"`, As JSON: `'{ "<Service_Name>": "<kubernetes secret>", ... } '`
`ingress.citrix.com/ingress.preconfigured.citrix.com-certkey`	JSON	Optional	Specify already existing SSL certificate keys on NetScaler that needs to be configured for content switching.	NA	One or more NetScaler sslcertkey entity names
`ingress.citrix.com/lbvserver`	JSON	Optional	Configure the certificate settings/parameters of NetScaler LBVserver entity. **Example:** `ingress.citrix.com/lbvserver: '{"app1.citrix.com": {"sslprofile": "sslprofile1", "ciphersuitebinding": "ciphersuite1", "sslcertkey": "sslcertkey1"}}'`	NA	Valid NetScaler with entity certificate parameter in type default/sni/ca.
`ingress.citrix.com/ingress.servicegroup.citrix.com/monitor`	JSON	Optional	Configure the NetScaler settings/parameters of ServiceGroup entity. **Example:** `ingress.citrix.com/servicegroup: '{"app1.citrix.com": {"monitor": "monitor1"}}'`	NA	Valid NetScaler parameter in `key:value` format.
`ingress.citrix.com/kubernetes.deployment.io/ingress.ingressclass`	String	Optional	Specify the particular Ingress controller. **Example:** `ingress.citrix.com/kubernetes.deployment.io/ingress.ingressclass: "citrix"`	NA	NetScaler entity parameter in `key:value` format.
`ingress.citrix.com/path-match-method`	String	Optional	Specify the particular path matching for Ingress controller. **Example:** `ingress.citrix.com/path-match-method: "prefix"`	`prefix`	Ingress classes mentioned in `prefix` or `exact`.
`ingress.citrix.com/ipam-range`	JSON	Optional	Specify the IP address range from a set of domain name ranges specified to IPAM controller. **Example:** `ingress.citrix.com/ipam-range: '{"www.citrix.com": "10.0.0.1-10.0.0.10", "www.citrix.com": "10.0.0.11-10.0.0.20"}'`	NA	Valid IPAM controller.
`ingress.citrix.com/external-service.citrix.com/canary-weight`	String	Optional	Specify the percentage of traffic to be directed to canary version. **Example:** `ingress.citrix.com/external-service.citrix.com/canary-weight: "20"`	NA	Valid integer value.

Annotations	Type	Required	Description	Default	Possible value
`ingress.citrix.com/canary-by-header`	String	Optional	Provide an HTTP header key to direct traffic to the canary version. See this section.	NA	
`ingress.citrix.com/canary-by-header-value`	List	Optional	Provide HTTP header values to direct traffic to the canary version. See this section.	NA	List of header values as strings.
`ingress.citrix.com/bot_crd`	String/JSON	Optional	Bind the policies created by BOT CRD to the application's load balancing virtual server. See this section.	NA	As a string: `"CRD_Instance_Name"`. As JSON: `'{ "<CRD_Instance_Name>": "<Service_Name>" }'`
`ingress.citrix.com/ratelimit_crd`	String/JSON	Optional	Bind the policies created by Ratelimit CRD to the application's load balancing virtual server. See this section.	NA	As a string: `"CRD_Instance_Name"`. As JSON: `'{ "<CRD_Instance_Name>": "CRD_Instance_Name" }'`.

Example:

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

Example:

```
ingress.citrix.com/bot_crd: "botdefense"
```

binds the policy to all the services in the ingress or

```
ingress.citrix.com/bot_crd: '{ "appname": "botdefense" }'
```

binds the policy to only the front-end service.

Annotations	Type	Required	Description	Default	Possible value
			Example: `ingress.citrix.com/ratelimit_crd: "ratelimitexample"` binds the policy to all the services in the ingress or `ingress.citrix.com/ratelimit_crd: '{ "appname": "ratelimitexample" } '` binds the policy to only frontend service.		
`ingress.citrix.com/auth_crd`	String/JSON	Optional	Bind the policies created by Auth CRD to the application's load balancing virtual server. See this section.	NA	As a string: `"CRD_Instance_Name"`, As JSON: `{ "<Service_Name>":"CRD_Instance_Name"}`

Annotations	Type	Required	Description	Default	Possible value
			Example: `ingress.citrix.com/auth_crd: "authexample"` binds the policy to all the services in the ingress or `ingress.citrix.com/auth_crd: '{ "appname": "authexample" } '` binds the policy to only the front-end service.		
`ingress.citrix.com/waf_crd`	String/JSON	Optional	Bind the policies created by WAF CRD to the application's load balancing virtual server. See this section.	NA	As a string: "CRD_Instance_Name" , As JSON: `'{ "<Service_Name>":"CRD_Instance_Name" } '`

Annotations	Type	Required	Description	Default	Possible value
			Example: `ingress.citrix.com/waf_crd: "wafbasic"` binds the policy to all the services in the ingress or `ingress.citrix.com/waf_crd: '{ "appname": "wafbasic" } '` binds the policy to only the front-end service		
`ingress.citrix.com/rewrite-responder_crd`	String/JSON	Optional	Bind the policies created by Rewrite-Responder CRD to the application's load balancing virtual server. See this section.	NA	As a string: `"CRD_Instance_Name"`, As JSON: `'{ "<Service_Name>":"CRD_Instance_Name" } '`

Annotations	Type	Required	Description	Default	Possible value
			Example: `ingress.citrix.com/rewrite-responder_crd: "blockurlpolicy"` Binds the policy to all the services in the ingress or `ingress.citrix.com/rewrite-responder_crd: '{ "appname": "blockurlpolicy" } '` binds the policy to only the front-end service.		
`ingress.citrix.com/rewrite-responder_crd`	String/JSON	Optional	Bind the policies created by rewrite-responder CRD to the application's load balancing virtual server. See this section.	NA	As a string: `"CRD_Instance_Name"`. As JSON: `'{ "<Service_Name>":"CRD_Instance_Name" } '`.

Annotations	Type	Required	Description	Default	Possible value
			Example: `ingress.citrix.com/rewrite-responder_crd: "blockurlpolicy"` binds the policy to all the services in the ingress or `ingress.citrix.com/rewrite-responder_crd: '{ "appname": "blockurlpolicy" } '` binds the policy to only the front-end service.		

Service annotations

The following are the service annotations supported by NetScaler.

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 for the second port is one.

Annotations	Type	Required	Description	Default	Possible value
`service.citrix.com/frontend-ip`	String	Optional	Specify an IP adress that needs to be used as content switching virtual server IP address.	NA	Numeric IP address, for example, '1.2.3.4'

Annotations	Type	Required	Description	Default	Possible value
service.citrix.com/ipam-range	String	Optional	Select a particular IP address range specified to IPAM.	NA	Value matching any of the range names configured in IPAM
service.citrix.com/insecure-redirect	JSON	Optional	Redirect HTTP from a set of traffic to a secure port. **Example:** `service.citrix.com/insecure-redirect: '{"port": 80, "redirectURL": "HTTPS://NEWHOST/path%s"}'`	NA	NA
service.citrix.com/ssl-termination	String	Optional	Specify the NetScaler IPAM. **Example:** `service.citrix.com/ssl-termination: 'EDGE'`	NA	EDGE and REENCRYPT
service.citrix.com/service-type-<index>	String	Optional	Specify the protocol for the NetScaler annotation is entities to be used for server created.	HTTP	TCP, HTTP, SSL,UDP,ANY, SSL_TCP, SIP_UDP.
service.citrix.com/ssl-certificate-data-<index>	String	Optional	Specify the value in the server PEM format. **Example:** `service.citrix.com/ssl-certificate-data-0: '<PEM formatted server cert>'`	NA	Certificate and Data in PEM Format
service.citrix.com/ssl-key-data-<index>	String	Optional	Specify the value in the server PEM format. **Example:** `service.citrix.com/ssl-key-data-0: '<PEM formatted server key>'`	NA	Key data in PEM Format
service.citrix.com/ssl-ca-certificate-data-<index>	String	Optional	Specify the CA value to verify the client certificate in the server PEM format. **Example:** `service.citrix.com/ssl-ca-certificate-data-0: '<PEM formatted CA cert>'`	NA	CA certificate data in PEM format
service.citrix.com/backend-ca-secret	String	Optional	Specify a CA certificate for the front-end server. **Example:** `service.citrix.com/backend-ca-secret: 'client-ca-secret'`	NA	Kubernetes secret Name
service.citrix.com/secret-data-<index>	String	Optional	Specify the back end resource for an HTTPS certificate for the front-end server. **Example:** `service.citrix.com/secret-data-0: '<PEM formatted server cert>'`	NA	Kubernetes secret Name
service.citrix.com/ca-secret	String	Optional	Specify a ca-certificate annotation if authentication. For more information and example, see, https://docs.citrix.com/en-us/netscaler-ingress-controller/1.0/ssl-offload/ssl-offload-ingress-example.html	NA	Kubernetes secret Name
service.citrix.com/backend-secret	String	Optional	This server communication between NetScaler and your workload.	NA	Kubernetes secret Name
service.citrix.com/backend-ca-secret	String	Optional	Specify a CA certificate for the back-end server. **Example:** `service.citrix.com/backend-ca-secret: 'backend-ca-secret'`	NA	NetScaler sslcertkey entity name
service.citrix.com/preconfigured-certkey	String	Optional	Specify the name of a pre-configured certificate key-pair bound to the NetScaler to be used as a server certificate for a workload. **Example:** `service.citrix.com/preconfigured-certkey: 'preconfigured-certkey'`	NA	NetScaler sslcertkey entity name

Annotations	Type	Required	Description	Default	Possible value
service.citrix.com/preconfigured-backend-certkey	String	Optional	Specify the name of a preconfigured certificate key in NetScaler to be bound to the back-end SSL service group. This SSL certificate is sent to the server authentication. **Example:** service.citrix.com/preconfigured-backend-ca-certkey: 'coffee-ca-cert'	NA	NetScaler sslcertkey entity name
service.citrix.com/preconfigured-backend-ca-certkey	String	Optional	Specify the name of a preconfigured CA certificate key in NetScaler to be bound to the back-end SSL service group. This SSL certificate is sent to the server authentication. **Example:** service.citrix.com/preconfigured-backend-ca-certkey: 'coffee-preconfigured-ca-cert'	NA	NetScaler sslcertkey entity name

Smart annotations for HTTP, TCP, or SSL profiles

Annotations	Type	Required	Description	Default	Possible value
ingress.citrix.com/frontend-httpprofile	String/JSON	Optional	Create the frontend HTTP profile handshake for server authentication. **Example:** ingress.citrix.com/frontend-httpprofile: '{ "dropinvalreqs":"enabled", "websocket" : "enabled" } '	NA	**Example:** ingress.citrix.com/frontend-httpprofile: '{ "dropinvalreqs":"enabled", "websocket" : "enabled" } '

Annotations	Type	Required	Description	Default	Possible value
`ingress.citrix.com/backend-httpprofile`	String/JSON	Optional	Create the back-end HTTP profile (Server Plane).	NA	**Example:** `ingress.citrix.com/backend-httpprofile: '{ "app-1": { "dropinvalreqs":"enabled", "websocket" : "enabled" } } '`
`ingress.citrix.com/frontend-tcpprofile`	String/JSON	Optional	Create the front-end TCP profile (Client Plane)	NA	**Example:** `ingress.citrix.com/frontend-tcpprofile: '{ "ws":"enabled", "sack" : "enabled" } '`
`ingress.citrix.com/backend-tcpprofile`	String/JSON	Optional	Create the back-end TCP profile (Server Plane)	NA	**Example:**`ingress.citrix.com/backend-tcpprofile: '{ "citrix-svc":{ "ws":"enabled", "sack" : "enabled" } } '`

Annotations	Type	Required	Description	Default	Possible value
`ingress.citrix.com/frontend-sslprofile`	String/JSON	Optional	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.	NA	**Example:** `ingress.citrix.com/frontend-sslprofile: '{ "hsts":"enabled", "tls12" : "enabled" } '`
`ingress.citrix.com/backend-sslprofile`	String/JSON	Optional	Create the back-end SSL profile (Server Plane). The SSL back-end profile is required only if you use `ingress.citrix.com/secure-backend`.	NA	**Example:** `ingress.citrix.com/backend-sslprofile: '{ "citrix-svc":{ "hsts":"enabled", "tls1" : "enabled" } } '`

Smart annotations for Ingress

Smart annotation is an option provided by NetScaler Ingress Controller to efficiently enable NetScaler features using the NetScaler entity name. The NetScaler Ingress Controller converts the Ingress in Kubernetes to a set of NetScaler objects. You can efficiently control these objects using smart annotations.

> **Note**
>
> To use smart annotations, you must have a good understanding of NetScaler features and their respective entity names. For more information about NetScaler features and entity names, see NetScaler documentation.

Smart annotation takes JSON format as input. The key and value that you pass in the JSON format

must match the NetScaler NITRO format. For more information about the NetScaler NITRO API, see NetScaler 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 NetScaler Ingress Controller:

NetScaler Entity Name	Smart Annotation	Example
lbvserver	`ingress.citrix.com/lbvserver`	`ingress.citrix.com/lbvserver: '{ "appname":{ "lbmethod":"SRCIPDESTIPHASH" } } '`
servicegroup	`ingress.citrix.com/servicegroup`	`ingress.citrix.com/servicegroup: '{ "appname":{ "cip": "Enabled","cipHeader":"X-Forwarded-For" } } '`
monitor	`ingress.citrix.com/monitor`	`ingress.citrix.com/monitor: '{ "appname":{ "type":"http" } } '`
csvserver	`ingress.citrix.com/csvserver`	`ingress.citrix.com/csvserver: '{ "stateupdate": "ENABLED" } '`

For information on smart annotations for HTTP, TCP, and SSL profiles, see Configure HTTP, TCP, or SSL profiles on NetScaler.

Sample ingress YAML with smart annotations

The following sample Ingress YAML includes smart annotations to enable NetScaler features using the entities such as, lbvserver, servicegroup, and monitor:

```
kubectl apply -f - <<EOF
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: '{
 "appname":{
 "lbmethod":"LEASTCONNECTION", "persistenceType":"SOURCEIP" }
  }
 '
    ingress.citrix.com/monitor: '{
 "appname":{
 "type":"http" }
  }
 '
    ingress.citrix.com/servicegroup: '{
 "appname":{
 "usip":"yes" }
  }
 '
  name: citrix
spec:
  rules:
  - host: citrix.org
    http:
      paths:
      - backend:
          service:
            name: appname
            port:
              number: 80
        path: /
        pathType: Prefix
EOF
<!--NeedCopy-->
```

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:

Smart Annotation	Description
`ingress.citrix.com/lbvserver: '{ "appname":{ "lbmethod":" LEASTCONNECTION", " persistenceType":"SOURCEIP" } } '`	Sets the load balancing method as Least Connection and also configures Source IP address persistence.
`ingress.citrix.com/servicegroup: '{ "appname":{ "usip":"yes" } } '`	Enables Use Source IP Mode (USIP) on NetScaler device. When you enable USIP on NetScaler, it uses the client's IP address for communication with the back-end pods.

Smart Annotation	Description
`ingress.citrix.com/monitor: '{ "appname":{ "type":"http" } } '`	Creates a custom HTTP monitor for the service group.

> **Note:**
>
> 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 using the content switching policies.

Smart annotations for routes

Similar to Ingress, you can also use smart annotations with OpenShift routes. NetScaler Ingress Controller converts the routes in OpenShift to a set of NetScaler objects.

The following table details the smart annotations provided by NetScaler Ingress Controller:

NetScaler entity name	Smart annotation	Example
`lbvserver`	`route.citrix.com/lbvserver`	`route.citrix.com/lbvserver: '{ "SRCIPDESTIPHASH" } } '`
`servicegroup`	`route.citrix.com/servicegroup`	`route.citrix.com/servicegroup: '{ "clientipheaderhdr": "X-Forwarded-For" } } '`
`monitor`	`route.citrix.com/monitor`	`route.citrix.com/monitor: '{ "`

Sample route manifest with smart annotations

The following example is a route YAML file.

```
kubectl apply -f - <<EOF
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: citrix
  annotations:
    route.citrix.com/lbvserver: '{
```



```
 "appname":{
 "lbmethod":"LEASTCONNECTION", "persistenceType":"SOURCEIP" }
  }
 '
    route.citrix.com/servicegroup: '{
 "appname":{
 "usip":"yes" }
  }
 '
    route.citrix.com/monitor: '{
 "appname":{
 "type":"http" }
  }
 '
spec:
  host:  citrix.org
  port:
    targetPort: 80
  to:
    kind: Service
    name: appname
    weight: 100
  wildcardPolicy: None
EOF
<!--NeedCopy-->
```

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:

Smart annotation	Description
`route.citrix.com/lbvserver: '{ "appname":{ "lbmethod":"LEASTCONNECTION", "persistenceType":"SOURCEIP" } } '`	Sets the load balancing method as Least Connection and also configures Source IP address persistence.
`route.citrix.com/servicegroup: '{ "appname":{ "usip":"yes" } } '`	Enables Use Source IP Mode (USIP) on NetScaler. When you enable USIP on the NetScaler, it uses the IP address of the client for communication with the back-end pods.
`route.citrix.com/monitor: '{ "appname":{ "type":"http" } } '`	Creates a custom HTTP monitor for the service group.

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:

```
    service.citrix.com/insecure-redirect: '{
 "port-443": 80 }
 '
  <!--NeedCopy-->
```

Sample service definition:

```
kubectl apply -f - <<EOF
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
EOF
<!--NeedCopy-->
```

Smart annotations for services

Smart annotations for services are used to configure NetScaler with custom values for NetScaler configuration parameters. The annotations are used for services of type `LoadBalancer` and for the services in NetScaler CPX used for East-West traffic.

> **Note:**
>
> If you have configured a service with NodePort or ClusterIP for the North-South traffic, then NetScaler is configured using the applicable ingress smart annotations rather than service annotations.

Smart annotations for services take JSON format as input. The key and value that you pass in the JSON format must match the NetScaler NITRO format. For more information about the NetScaler NITRO API, see NetScaler REST APIs - NITRO Documentation.

Example smart annotation for services:

```
 service.citrix.com/lbvserver: '{
"80-tcp":{
"lbmethod":"SRCIPDESTIPHASH" }
 }
'
 <!--NeedCopy-->
```

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 describes the smart annotations for services:

NetScaler Entity Name	Smart Annotation for Service	Example
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" } } '`
analyticsprofile	`service.citrix.com/analyticsprofile`	`service.citrix.com/analyticsprofile: '{ "80-tcp":{ "webinsight": { "httpurl":"ENABLED", "httpuseragent":"ENABLED" } } } '`

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 YAML is a sample deployment and service definition for a basic apache web-server based application. It includes smart annotations for services to enable NetScaler features using the entities such as lbvserver, csvserver, servicegroup, monitor, and analyticsprofile:

```
kubectl apply -f - <<EOF
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

---
#Expose the apache web server as a service
apiVersion: apps/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" }
  }
  }
 '
    NETSCALER_VPORT: '80'
  labels:
    name: apache
spec:
  externalTrafficPolicy: Local
  type: LoadBalancer
  selector:
    name: apache
  ports:
  - name: http
    port: 80
    targetPort: http
  selector:
    app: apache
---
EOF
<!--NeedCopy-->
```

Examples

Sample ingress YAML for SIP_UDP support in insecure service type annotation

The following sample ingress YAML includes the configuration for enabling SIP over UDP support using the `ingress.citrix.com/insecure-service-type` annotation.

```
kubectl apply -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
```



```
5    annotations:
6      ingress.citrix.com/frontend-ip: 1.1.1.1
7      ingress.citrix.com/insecure-port: "5060"
8      ingress.citrix.com/insecure-service-type: sip_udp
9      ingress.citrix.com/lbvserver: '{
10   "asterisk17":{
11   "lbmethod":"CALLIDHASH","persistenceType":"CALLID" }
12    }
13   '
14    name: sip-ingress
15 spec:
16    defaultBackend:
17      service:
18        name: asterisk17
19        port:
20          number: 5060
21 EOF
22 <!--NeedCopy-->
```

ConfigMap support for the NetScaler Ingress Controller

December 31, 2023

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 NetScaler Ingress Controller supports the configuration command line arguments, and environment variables mentioned in deploying the NetScaler Ingress Controller. But, you cannot update these configurations at runtime without rebooting the NetScaler Ingress Controller pod. With ConfigMap support, you can update the configuration automatically while keeping the NetScaler Ingress Controller pod running. You do not need to restart the pod after the update.

Supported environment variables in the NetScaler Ingress Controller

The values for the following environment variables in the NetScaler Ingress Controller can be specified in a ConfigMap.

- LOGLEVEL: Specifies the log levels to control the logs generated by the NetScaler Ingress Controller (debug, info, critical, and so on). The default value is `debug`.

- NS_HTTP2_SERVER_SIDE: Enables HTTP2 for NetScaler 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 NetScaler 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 NetScaler.
- 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 NetScaler.
- 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.
- NS_ADNS_IPS: Enables configuring NetScaler as an ADNS server.

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 NetScaler.

> **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 NetScaler Ingress Controller

This example shows how to create a ConfigMap and apply the ConfigMap to the NetScaler 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 NetScaler Ingress Controller.

1. 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'
  NS_PORT: '80'
  NS_COOKIE_VERSION: '0'
  NS_HTTP2_SERVER_SIDE: 'ON'
```

2. Deploy the `cic-configmap.yaml` using the following command.

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

3. Edit the `cic.yaml` file for deploying the NetScaler Ingress Controller as a stand-alone pod and specify the following:

```
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 NetScaler 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:
        serviceAccountName: cic-k8s-role
        containers:
        - name: cic-k8s-ingress-controller
          image: "quay.io/citrix/citrix-k8s-ingress-controller
              :1.36.5"
          env:
          # Set NetScaler NSIP/SNIP, SNIP in case of HA (mgmt has
              to be enabled)
          - name: "NS_IP"
            value: "x.x.x.x"
          - name: "EULA"
            value: "yes"
          args:
            - --ingress-classes
              citrix
            - --feature-node-watch
              false
            - --configmap
              default/cic-configmap
          imagePullPolicy: Always
```

4. Deploy the NetScaler Ingress Controller as a stand-alone pod by applying the YAML.

```
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’.

```
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.

```
kubectl apply -f cic-configmap.yaml
```

7. (Optional) If you need to delete the ConfigMap, use the following command.


```
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 NetScaler Ingress Controller, use the following in the NetScaler Ingress Controller deployment YAML file.

```
    envFrom:
            - configMapRef:
              name: cic-configmap
```

Ingress configurations

December 31, 2023

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.

NetScaler Ingress Controller is built around the Kubernetes Ingress and automatically configures one or more NetScaler 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:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: virtual-host-ingress
  namespace: default
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          service:
            name: service1
            port:
              number: 80
        pathType: Prefix
        path: /
```



```
  - host: bar.foo.com
    http:
      paths:
      - backend:
          service:
            name: service2
            port:
              number: 80
        pathType: Prefix
        path: /
<!--NeedCopy-->
```

After the sample Ingress definition is deployed, all the HTTP request with a host header is load balanced by NetScaler to `service1`. And, the HTTP request with a host header is load balancer by NetScaler 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:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: path-ingress
  namespace: default
spec:
  rules:
  - host: test.example.com
    http:
      paths:
      - backend:
          service:
            name: service1
            port:
              number: 80
        path: /foo
        pathType: Prefix
      - backend:
          service:
            name: service2
            port:
              number: 80
        path: /
        pathType: Prefix
<!--NeedCopy-->
```

After the sample Ingress definition is deployed, any HTTP requests with host `test.example.com` and URL path with prefix / `foo`, NetScaler routes the request to `service1` and all other requests are routed to `service2`.

NetScaler Ingress Controller follows first match policy to evaluate paths. For effective matching, NetScaler 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.

```
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
        path: /
        pathType: Prefix
<!--NeedCopy-->
```

After the sample Ingress definition is deployed, HTTP requests to all the subdomains of `example.com` is routed to `service1` by NetScaler.

> **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` APIversion can make use of `PathType: Exact` to consider the path for the exact match.

```
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 NetScaler 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:

```
apiVersion: extension/v1beta1
kind: Ingress
metadata:
  name: path-exact-ingress
  namespace: default
  annotations:
    ingress.citrix.com/path-match-method: "exact"
spec:
  rules:
  - host:test.example.com
    http:
      paths:
      - path: /exact
        backend:
          serviceName: service1
          servicePort: 80
<!--NeedCopy-->
```

After the sample Ingress definition is deployed, HTTP requests with path `/exact` is routed by NetScaler 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.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: default-path-ingress
  namespace: default
```



```
spec:
  rules:
  - http:
      paths:
      - backend:
          service:
            name: service1
            port:
              number: 80
        path: /foo
        pathType: Prefix
      - backend:
          service:
            name: service2
            port:
              number: 80
        path: /
        pathType: Prefix
<!--NeedCopy-->
```

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

<!--NeedCopy-->
```

> **Note:**
>
> A global default back end can be specified if NetScaler CPX is load balancing the traffic. You can create a default back end per `frontend-ip:port` combination in case of NetScaler VPX or MPX is the ingress device.

Ingress class support

December 31, 2023

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.

NetScaler Ingress Controller and Ingress classes

The NetScaler Ingress Controller supports accepting multiple ingress resources, which have `kuberneters.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 NetScaler CPX. In the given YAML snippet, the following ingress classes are associated with the Ingress Controller.

- `my-custom-class`
- `Citrix`


```
spec:
    serviceAccountName: cic-k8s-role
    containers:
    - name: cic-k8s-ingress-controller
      image:"quay.io/citrix/citrix-k8s-ingress-controller:latest"
      # specify the ingress classes names to be supportedbyIngress Controller in args section.
      # First line should be --ingress-classes, andeverysubsequent line should be
      # the name of allowed ingress class. In the givenexampletwo classes named
      # "citrix" and "my-custom-class" are accepted. Thiswill 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 NetScaler Ingress Controller is configured to accept `my-custom-class`, it processes this Ingress resource.

```
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.

```
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  name: citrix
spec:
  controller: citrix.com/ingress-controller

<!--NeedCopy-->
```


An `IngressClass`resource must refer to the ingress class associated with the controller that should implement the Ingress rules as shown as follows:

```
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
<!--NeedCopy-->
```

The NetScaler Ingress Controller uses the following rules to match the Ingresses.

- If the NetScaler 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 NetScaler Ingress Controller is started with one or more ingress classes set using the `--ingress-classes` argument.
  - 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 NetScaler Ingress Controller, if the `IngressClass` resource is supported and the NetScaler Ingress Controller is deployed with the `--ingress-classes` argument, the `v1.IngressClass` resource is created by default.

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 NetScaler Ingress Controller with the allocated IP addresses, specify the command line argument `--update-ingress-status yes` when you start the NetScaler Ingress Controller. This feature is only supported for the NetScaler Ingress Controller deployed as a stand-alone pod for managing NetScaler VPX or MPX. For NetScaler CPXs deployed as sidecars, this feature is not supported.

Following is an example YAML with the `--update-ingress-status yes` command line argument enabled.

```
1 args:
2     - --feature-node-watch false
3     - --ipam citrix-ipam-controller
4     - --update-ingress-status yes
5     imagePullPolicy: Always
6 <!--NeedCopy-->
```

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, NetScaler 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 NetScaler CPX along with the ingress controller. So, the Ingress resources exposed with the NetScaler 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 NetScaler CPX is exposed as service type `LoadBalancer` to the tier-1 NetScaler VPX ingress, the ingress resources operated by the NetScaler CPX is updated with the VIP address.

This topic provides information about how to enable the ingress status update for NetScaler CPX with the NetScaler Ingress Controller as sidecar deployments.

Note: The ingress status update for the sidecar feature is supported only on services of type `LoadBalancer`.

Sample ingress output after an ingress status update

The following is a sample ingress output after the ingress status update:

```
1     $ kubectl get ingress
2
3     NAME              HOSTS               ADDRESS
                                     PORTS     AGE
4     sample-ingress    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 NetScaler CPX YAML file. You must add the argument to the `args` section of NetScaler CPX in the deployment YAML file for NetScaler CPX with the NetScaler 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

Keyword/variable	Description
`--cpx-service`	Specifies the argument for enabling this feature.
`<namespace>/<name-of-the-type-load-balancer-service-exposing-cpx>`	Specifies the format in which the argument value to be provided.
`<namespace>`	Specifies the namespace in which the service is created.
`<name-of-the-type-load-balancer-service-exposing-cpx>`	Specifies the name of the service that exposes NetScaler CPX.

> **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` should be a Kubernetes service of `type LoadBalancer`.

Service class for services of type LoadBalancer

December 31, 2023

When services of type LoadBalancer are deployed, all such services are processed by the NetScaler Ingress Controller and configured on NetScalers. However, there may be situations where you want to associate only specific services to a NetScaler 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 NetScaler 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 NetScaler Ingress Controller, then it accepts only those services of the type `LoadBalancer` that match the `service.citrix.com/class` annotation. In this case, the NetScaler 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`

```
spec:
serviceAccountName: cic-k8s-role
containers:
- name: cic-k8s-ingress-controller
  # specify the service classes to be supported by NetScaler 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
  # "svc-class1" and "svc-class2" are accepted. This will be case-insensitive.
  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 NetScaler Ingress Controller is configured to accept `svc-class1`, it configures the service on the NetScaler.

```
apiVersion: v1
kind: Service
metadata:
     name: apache
     annotations:
         service.citrix.com/class: 'svc-class1'
     labels:
```



```
8          name: apache
9  spec:
10    type: LoadBalancer
11    selector:
12        name: apache
13    ports:
14    - name: http
15      port: 80
16      targetPort: http
17    selector:
18    app: apache
19 <!--NeedCopy-->
```

Configure HTTP, TCP, or SSL profiles on NetScaler

December 31, 2023

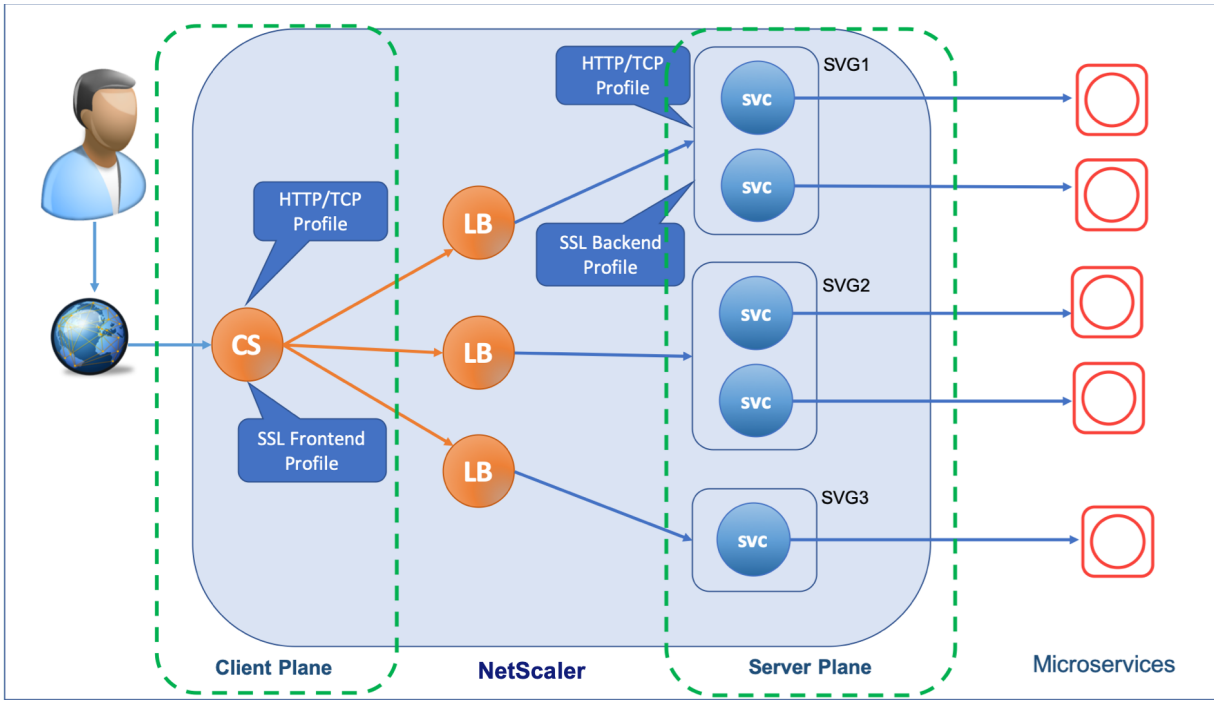
Configurations such as, HTTP, TCP, or SSL for a NetScaler appliance can be specified using individual entities such as HTTP profile, TCP profile, or SSL profile respectively. The profile is a collection of settings pertaining to the individual protocols, for example, HTTP profile is a collection of HTTP settings. 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.

NetScaler Ingress Controller enables you to configure HTTP, TCP, or SSL related configuration on the Ingress NetScaler using profiles.

Understand NetScaler configuration in Kubernetes environment

In a Kubernetes environment, the Ingress NetScaler 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 NetScaler to the microservices in Kubernetes (highlighted as `Server Plane` in the diagram).



The NetScaler 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 NetScaler Ingress Controller provides the following two smart annotations for HTTP profile. You can use these annotations to define the HTTP settings for the NetScaler. When you deploy an ingress that includes these annotations, the NetScaler Ingress Controller creates an HTTP profile derived from the default HTTP profile (`nshttp_default_profile`) configured on the NetScaler. Then, it applies the parameters that you have provided in the annotations to the new HTTP profile and applies the profile to the NetScaler.

Smart annotation	Description	Sample
`ingress.citrix.com/frontend-httpprofile`	Use this annotation to create the front-end HTTP profile (Client Plane)	`ingress.citrix.com/frontend-httpprofile: '{ "dropinvalreqs":"enabled", "websocket" : "enabled" } '`

Smart annotation	Description	Sample
`ingress.citrix.com/backend-httpprofile`	Use this annotation to create the back-end HTTP profile (Server Plane). **Note:** Ensure that you manually enable the HTTP related global parameters on the NetScaler. For example, to use HTTP2 at the back end (Server Plane), ensure that you can enable `HTTP2Serverside` global parameter in the NetScaler. For more information, see Configurating HTTP2.	`ingress.citrix.com/backend-httpprofile: '{ "app-1": { " dropinvalreqs":" enabled", "websocket" : "enabled" } } '`

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 NetScaler Ingress Controller provides the following two smart annotations for TCP profile. You can use these annotations to define the TCP settings for the NetScaler. When you deploy an ingress that includes these annotations, the NetScaler Ingress Controller creates a TCP profile derived from the default TCP profile (`nstcp_default_profile`) configured on the NetScaler. Then, it applies the parameters that you have provided in the annotations to the new TCP profile and applies the profile to the NetScaler.

Smart annotation	Description	Sample
`ingress.citrix.com/frontend-tcpprofile`	Use this annotation to create the front-end TCP profile (Client Plane)	`ingress.citrix.com/frontend-tcpprofile: '{ "ws":"enabled", " sack" : "enabled" } '`

Smart annotation	Description	Sample
`ingress.citrix.com/backend-tcpprofile`	Use this annotation to create the back-end TCP profile (Server Plane)	`ingress.citrix.com/backend-tcpprofile: ' { "citrix-svc":{ "ws":"enabled", "sack" : "enabled" } } '`

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 NetScaler, by default, SSL profile is not enable on the Ingress NetScaler. Ensure that you manually enable SSL profile on the NetScaler. Enabling the SSL profile overrides all the existing SSL related setting on the NetScaler, 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.
- 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 NetScaler, a default front end profile (`ns_default_ssl_profile_front`) 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 NetScaler.

The NetScaler 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:

Smart annotation	Description	Sample
`ingress.citrix.com/frontend-sslprofile`	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.	`ingress.citrix.com/frontend-sslprofile: '{ "hsts":"enabled", "tls12" : "enabled" } '`
`ingress.citrix.com/backend-sslprofile`	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.	`ingress.citrix.com/backend-sslprofile: ' { "citrix-svc":{ " hsts":"enabled", " tls1" : "enabled" } } '`

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:

```
#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:

```
#Sample ingress manifest for the front-end configuration for an SSL 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/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:
      #Presense 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 NetScaler 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-httpprofile` smart annotation in the ingress definition.

To use an existing profile on NetScaler or use a built-in HTTP profile.

```
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 NetScaler.

Alternatively, you can set the profile parameters as specified as follows. See the HTTP profile NITRO documentation for all possible key-values.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cic-configmap
  labels:
    app: citrix-ingress-controller
data:
  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 NetScaler or use a built-in TCP profile:

```
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 NetScaler.

Alternatively, you can set the profile parameters as follows. See the NetScaler TCP profile NITRO documentation for all possible key values.


```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cic-configmap
  labels:
    app: citrix-ingress-controller
data:
  FRONTEND_TCP_PROFILE: |
    config:
      sack: 'ENABLED'
      nagle: 'ENABLED'

<!--NeedCopy-->
```

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 NetScaler using the `set ssl parameter -defaultProfile ENABLED` command. Make sure that NetScaler Ingress Controller is restarted after enabling the default profile. The default profile is automatically enabled when NetScaler 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 NetScaler or use a built-in SSL profile,

```
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 NetScaler.

> **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.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cic-configmap
  labels:
    app: citrix-ingress-controller
data:
  FRONTEND_SSL_PROFILE: |
    config:
      tls13: 'ENABLED'
      hsts: 'ENABLED'
<!--NeedCopy-->
```

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 NetScaler or user-created cipher groups in this field. For information about the list of cyphers available in the NetScaler, see Ciphers in NetScaler.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cic-configmap
  labels:
    app: citrix-ingress-controller
data:
  FRONTEND_SSL_PROFILE: |
    config:
      tls13: 'ENABLED'
      ciphers:
      - TLS1.3-AES256-GCM-SHA384
      - TLS1.3-CHACHA20-POLY1305-SHA256
<!--NeedCopy-->
```

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

```
#The values for the parameters are for demonstration purpose only.
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-httpprofile: '{
 "apache":{
 "markhttp09inval": "disabled" }
  }
 '
    ingress.citrix.com/backend-tcpprofile: '{
 "apache":{
 "sack":"enabled" }
  }
 '
    ingress.citrix.com/frontend-ip: 'VIP_IP'
    ingress.citrix.com/insecure-port: "80"
  name: apache-ingress
spec:
  rules:
  - host: www.apachetest.com
    http:
      paths:
      - backend:
          service:
            name: apache
            port:
              number: 80
        path: /
        pathType: Prefix
<!--NeedCopy-->
```

Sample backend ingress manifest with TLS configuration

```
#The values for the parameters are for demonstration purpose only.

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-httpprofile: '{
 "hotdrink":{
 "markhttp09inval": "disabled" }
  }
 '
    ingress.citrix.com/backend-sslprofile: '{
 "hotdrink":{
 "snienable": "enabled" }
  }
 '
```



```
18      ingress.citrix.com/backend-tcpprofile: '{
19   "hotdrink":{
20   "sack":"enabled" }
21    }
22   '
23      ingress.citrix.com/frontend-ip: 'VIP_IP'
24      ingress.citrix.com/secure-backend: '{
25   "hotdrink":"true" }
26   '
27      ingress.citrix.com/secure-port: "443"
28    name: hotdrink-ingress
29  spec:
30    rules:
31    - host: hotdrinks.beverages.com
32      http:
33        paths:
34        - backend:
35            service:
36              name: hotdrink
37              port:
38                number: 443
39          path: /
40          pathType: Prefix
41    tls:
42    - secretName: hotdrink.secret
43  <!--NeedCopy-->
```

Using built-in or existing user-defined profiles on the Ingress NetScaler

You can use the individual smart annotations to configure the built-in profiles or existing user-defined profiles on the Ingress NetScaler 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 NetScaler. The following is a sample ingress annotation:

```
1  ingress.citrix.com/frontend-httpprofile: "http_preconf_profile1"
```

Where, ‘http_preconf_profile1’is the profile that exists on the Ingress NetScaler.

For the back-end configuration, you must provide the name of the built-in or existing profile on the Ingress NetScaler 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 NetScaler and `citrix-svc` is the back-end service name.

Sample HTTP profile

```
ingress.citrix.com/frontend-httpprofile: "http_preconf_profile"
ingress.citrix.com/backend-httpprofile: '{
 "citrix-svc": "http_preconf_profile" }
 '
```

Sample TCP profile

```
ingress.citrix.com/frontend-tcpprofile: "tcp_preconf_profile"
ingress.citrix.com/backend-tcpprofile: '{
 "citrix-svc":"tcp_preconf_profile" }
 '
```

Sample SSL profile

```
ingress.citrix.com/frontend-sslprofile: "ssl_preconf_profile"
ingress.citrix.com/backend-sslprofile: '{
 "citrix-svc":"ssl_preconf_profile" }
 '
```

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, back-end and TLS is not defined.

   A sample YAML (`ingress1.yaml`) is provided as follows:

```
  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" }
```



```
14  '
15       ingress.citrix.com/frontend-sslprofile: '{
16  "hsts":"enabled", "tls13" : "enabled" }
17  '
18   spec:
19     tls:
20     - hosts:
21     rules:
22     - host:
23   <!--NeedCopy-->
```

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 back-end defined which creates the load balancing resource definition.

   A sample YAML (ingress2.yaml) is provided as follows:

```
1   apiVersion: networking.k8s.io/v1
2   kind: Ingress
3   metadata:
4     name: ingress-vpx2
5     annotations:
6     kubernetes.io/ingress.class: "vpx"
7     ingress.citrix.com/insecure-termination: "allow"
8     ingress.citrix.com/frontend-ip: "10.221.36.190"
9   spec:
10    tls:
11    - secretName: <hotdrink-secret>
12    rules:
13    - host:  hotdrink.beverages.com
14      http:
15        paths:
16        - path:
17          backend:
18            serviceName: frontend-hotdrinks
19            servicePort: 80
20  <!--NeedCopy-->
```

4. Deploy the back-end ingress resource.

```
1  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.

```
1  # show cs vserver <k8s150-10.221.36.190_443_ssl>
2
3    k8s150-10.221.36.190_443_ssl (10.221.36.190:443) - SSL Type:
       CONTENT
4    State: UP
```



```
5    Last state change was at Thu Apr 22 20:14:44 2021
6    Time since last state change: 0 days, 00:10:56.850
7    Client Idle Timeout: 180 sec
8    Down state flush: ENABLED
9    Disable Primary Vserver On Down : DISABLED
10   Comment: uid=
        QEYQI2LDW5WR4A6P3NSZ37XICKOJKV4HPEM2H4PSK4HWA3JQWCLQ====
11   TCP profile name: k8s150-10.221.36.190_443_ssl
12   HTTP profile name: k8s150-10.221.36.190_443_ssl
13   Appflow logging: ENABLED
14   State Update: DISABLED
15   Default:   Content Precedence: RULE
16   Vserver IP and Port insertion: OFF
17   L2Conn: OFF Case Sensitivity: ON
18   Authentication: OFF
19   401 Based Authentication: OFF
20   Push: DISABLED Push VServer:
21   Push Label Rule: none
22   Persistence: NONE
23   Listen Policy: NONE
24   IcmpResponse: PASSIVE
25   RHIstate:  PASSIVE
26   Traffic Domain: 0
27
28   1)  Content-Switching Policy: k8s150-ingress-vpx1_tier-2-
        adc_443_k8s150-frontend-hotdrinks_tier-2-adc_80_svc
        Priority: 200000004   Hits: 0
29   Done
```

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 NetScaler using the `set ssl parameter -defaultProfile ENABLED` command. Make sure that NetScaler 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:

```
1 apiVersion: networking.k8s.io/v1
2 kind: Ingress
3 metadata:
4   name: ingress-vpx1
5   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: '{
  "snienable": "enabled", "hsts":"enabled", "tls13" : "enabled" }
  '
spec:
  tls:
  - hosts:
  rules:
  - host:

<!--NeedCopy-->
```

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 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:

```
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:
  - 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.

```yaml
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:
      - 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 NetScaler using the `set ssl parameter -defaultProfile ENABLED` command. Make sure that NetScaler Ingress Controller is restarted after enabling default profile.

Set default SSL profile on NetScaler using the command `set ssl parameter -defaultProfile ENABLED` before deploying NetScaler Ingress Controller. If you have already deployed NetScaler

Ingress Controller, then redeploy it. For more information about the SSL default profile, see documentation.

For information on supported Ciphers on the NetScaler appliances, see Ciphers available on the NetScaler appliances.

For information about securing cipher, see securing cipher.

A sample YAML (cat frontend_ingress.yaml) is provided as follows:

```
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" : [{
 "ciphername": "test", "cipherpriority" :"1" }
 ] }
 '
spec:
  tls:
  - hosts:
  rules:
   - host:
<!--NeedCopy-->
```

Log levels

December 31, 2023

The logs generated by NetScaler Ingress Controller are available as part of kubernetes logs. You can specify NetScaler Ingress Controller to log in the following log levels:

- CRITICAL
- ERROR
- WARNING
- INFO

- DEBUG

By default, NetScaler Ingress Controller is set to log in `INFO` log level. If you want to specify NetScaler Ingress Controller to log in a particular log level then you need to specify the log level in the NetScaler Ingress Controller deployment YAML file before deploying the NetScaler Ingress Controller. You can specify the log level in the `spec` section of the YAML file as follows:

```
apiVersion: v1
kind: Pod
metadata:
  name: citrixingresscontroller
  labels:
    app: citrixingresscontroller
spec:
      serviceAccountName: cpx
      containers:
      - name: citrixingresscontroller
        image: "quay.io/citrix/citrix-k8s-ingress-controller:1.36.5"
        env:
        # Set kube api-server URL
        - name: "kubernetes_url"
          value: "https://10.x.x.x:6443"
        # Set NetScaler Management IP
        - name: "NS_IP"
          value: "10.x.x.x"
        # Set log level
        - name: "LOGLEVEL"
          value: "DEBUG"
        - name: "EULA"
          value: "yes"
        args:
        - --feature-node-watch
          true
        imagePullPolicy: Always
<!--NeedCopy-->
```

Modify the log levels

To modify the log level configured on the NetScaler Ingress Controller instance, you need to delete the instance and update the log level value in the following section and redeploy the NetScaler Ingress Controller instance:

```
# Set log level
- name: "LOGLEVEL"
  value: "XXXX"
<!--NeedCopy-->
```

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

December 31, 2023

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 NetScaler 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 NetScaler.

Service annotation	Description
`service.citrix.com/frontend-tcpprofile`	Use this annotation to create the front-end TCP profile (Client Plane).
`service.citrix.com/backend-tcpprofile`	Use this annotation to create the back-end TCP profile (Server Plane).

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`).

Service annotation	Sample
`service.citrix.com/frontend-tcpprofile` `service.citrix.com/backend-tcpprofile`	`service.citrix.com/frontend-tcpprofile: enabled` `service.citrix.com/backend-tcpprofile: enabled`

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.

```
service.citrix.com/frontend-tcpprofile: "tcp_preconf_profile"
service.citrix.com/backend-tcpprofile: '{
 "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.

```
  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.

SSL certificate for services of type LoadBalancer through the Kubernetes secret resource

December 31, 2023

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 NetScaler Ingress Controller default certificate

If the SSL certificate is not provided, you can use the default NetScaler 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 NetScaler Ingress Controller YAML file.

Default NetScaler Ingress Controller

```
        --default-ssl-certificate <NAMESPACE>/<SECRET_NAME>
```

Service annotations for SSL certificate as Kubernetes secrets

NetScaler Ingress Controller provides the following service annotations to use SSL certificates stored as Kubernetes secrets for services of type `LoadBalancer`.

Service annotation	Description
`service.citrix.com/secret`	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.

Service annotation	Description
`service.citrix.com/ca-secret`	Use this annotation to provide a CA certificate for client certificate authentication. This certificate is bound to the front-end SSL virtual server in NetScaler.
`service.citrix.com/backend-secret`	Use this annotation if the back-end communication between NetScaler 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.
`service.citrix.com/backend-ca-secret`	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 NetScaler.
`service.citrix.com/preconfigured-certkey`	Use this annotation to specify the name of the preconfigured cert key in the NetScaler to be used as a front-end server certificate.
`service.citrix.com/preconfigured-ca-certkey`	Use this annotation to specify the name of the preconfigured cert key in the NetScaler to be used as a CA certificate for client certificate authentication. This certificate is bound to the front-end SSL virtual server in NetScaler.
`service.citrix.com/preconfigured-backend-certkey`	Use this annotation to specify the name of the preconfigured cert key in the NetScaler to be bound to the back-end SSL service group. This certificate is sent to the server during the SSL handshake for server authentication.
`service.citrix.com/preconfigured-backend-ca-certkey`	Use this annotation to specify the name of the preconfigured CA cert key in the NetScaler to bound to back-end SSL service group for server authentication.

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.

```
        service.citrix.com/secret: hotdrink-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/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.

```
 # port-proto: secret
 service.citrix.com/backend-ca-secret: '{
"443-tcp": "coffee-ca", "8443-tcp": "tea-ca" }
'

 # portname: secret

 service.citrix.com/backend-ca-secret: '{
"coffee-443": "coffee-ca", "tea-8443": "tea-ca" }
'

 # applicable to all ports

 service.citrix.com/backend-ca-secret: "hotdrink-ca-secret"
```

BGP advertisement of external IP addresses for type LoadBalancer services and Ingresses using NetScaler CPX

December 31, 2023

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. 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.

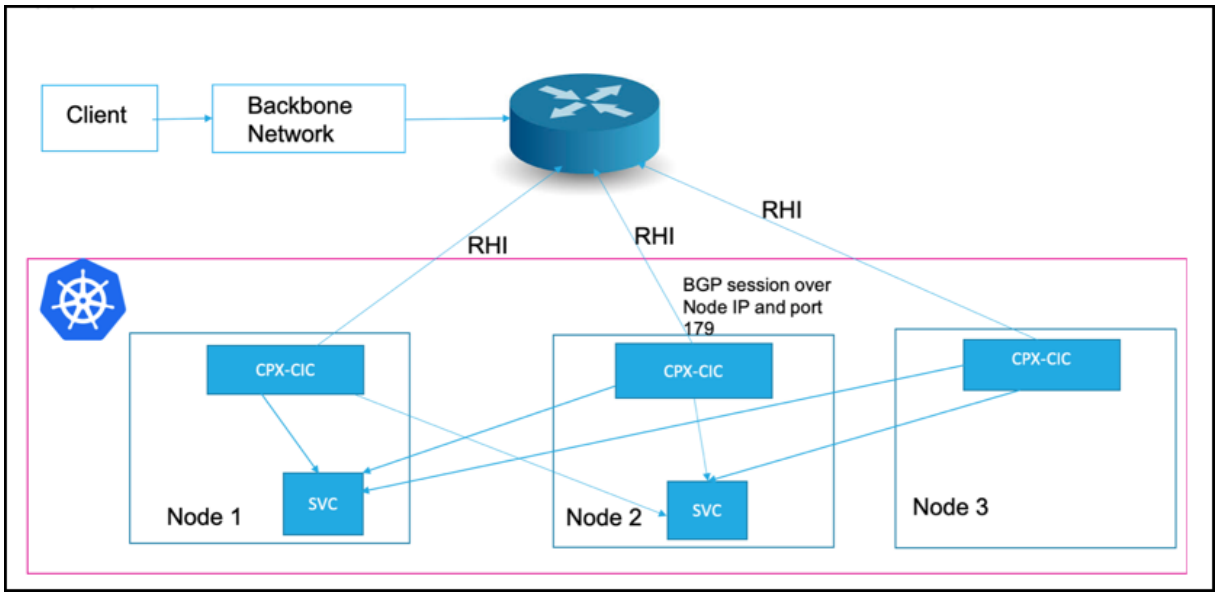
You can expose the services of type `LoadBalancer` using the NetScaler Ingress Controller and Tier-1 NetScaler devices such as NetScaler VPX or MPX. The NetScaler VPX or MPX residing outside the Kubernetes cluster load balances the incoming traffic to the Kubernetes services. For more information

on such a deployment, see Expose services of type `LoadBalancer`.

However, it may not be always feasible to use an external ADC device to expose the service of type LoadBalancer in an on-prem environment. Some times, it is desirable to manage all related resources from the Kubernetes cluster itself without any external component. The NetScaler Ingress Controller provides a way to expose the service of type LoadBalancer using NetScaler 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, NetScaler CPX is deployed as a daemonset on the Kubernetes nodes in host mode. NetScaler 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 NetScaler CPX with this mode, NetScaler CPX adds iptables rules for each service of type LoadBalancer on Kubernetes nodes. The traffic destined to the external IP address is routed to NetScaler CPX pods.

The following diagram explains a deployment where NetScaler CPX is exposing a service of type LoadBalancer:



As shown in the diagram, NetScaler 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 NetScaler Ingress Controller configures the NetScaler 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 NetScaler 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 NetScaler Ingress Controller which acts as the VIP for all ingress resources. When an Ingress resource is added, NetScaler CPX advertises the `NS_VIP` to external routers through BGP to attract the traffic. Once traffic comes to the `NS_VIP`, NetScaler CPX performs the content switching and load balancing as specified in the ingress resource.

> **Note:**
>
> For this solution to work, the NetScaler Ingress Controller must run as a root user and must have the `NET_ADMIN` capability.

Deploy NetScaler CPX solution for services of type LoadBalancer

This procedure explains how to deploy NetScaler CPX as a daemonset in the host network to expose services of type LoadBalancer.

This configuration includes the following tasks:

- Deploy NetScaler CPX with the NetScaler Ingress Controller as sidecar
- 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 NetScaler CPX and the NetScaler Ingress Controller.

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

2. Download the citrix-k8s-cpx-ingress.yml using the following command.

```
wget  https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/docs/configure/cpx-bgp-router/citrix-k8s-cpx-ingress.yml
```

3. Edit the `citrix-k8s-cpx-ingress.yaml` file and specify the required values.

- The argument `–configmap` specifies the ConfigMap location for the NetScaler Ingress Controller in the form of `namespace/name`.
- The argument `--ipam citrix-ipam-controller` can be specified if you are running the for automatic IP address allocation.
- (Optional) `nodeSelector` to select the nodes where you need to run the NetScaler 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 NetScaler CPX and the NetScaler Ingress Controller.

```
kubectl apply -f citrix-k8s-cpx-ingress.yml
```

5. Create a ConfigMap (configmap.yaml) with the BGP configuration which is passed as an argument to the NetScaler 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 NetScaler CPX to connect
   - The autonomous system (AS number) of the router
   - The AS number for NetScaler 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.

> **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 NetScaler to configure different NetScaler functionalities.

Following is an example for configuration of service of type LoadBalancer.

```
apiVersion: v1
kind: Service
metadata:
  name: kuard-service
  annotations:
    # This uses IPAM to allocate an IP from range 'Dev'
    # 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
  selector:
    app: kuard
  ports:
  - port: 80
    targetPort: 8080
    name: http
  - port: 443
    targetPort: 8443
    name: https
```

8. Apply the service of type LoadBalancer.

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

Once the service is applied, the NetScaler 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 NetScaler Ingress Controller.

You must have the following information to configure BGP routing:

- The router IP address so that NetScaler CPX can connect to it
- The autonomous system (AS number) of the router
- The AS number for NetScaler CPX

In the following ConfigMap for the BGP configuration, the `bgpConfig` field represents the BGP configuration.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config
  labels:
    app: cic
data:
  NS_BGP_CONFIG: |
    bgpConfig:
```



```
10      - bgpRouter:
11          localAS: 100
12          neighbor:
13          - address: x.x.x.x
14            remoteAS: 100
15            advertisementInterval: 10
16            ASOriginationInterval: 10
17  <!--NeedCopy-->
```

The following table explains the various fields of the `bgpConfig` field.

Field	Description	Type	Default value	Required
`nodeSelector`	If the `nodeSeclector` field is present, then the BGP router configuration is applicable for nodes which matches the nodeSelector field. `nodeSelector` accepts comma separated `key=value` pairs where each key represents a label name and the value is the label value. For example: nodeSelector: datacenter=ds1,rack-rack1	string		No

Field	Description	Type	Default value	Required
`bgpRouter`	Specifies the BGP configuration. For information on different fields of the `bgpRouter`, see the following table.	bgpRouter		Yes

The following table explains the fields for the `bgpRouter` field.

Field	Description	Type	Default value	Required
`localAS`	AS number for the NetScaler CPX	integer		Yes
`neighbor`	Neighbor router BGP configuration.	neighbor		Yes

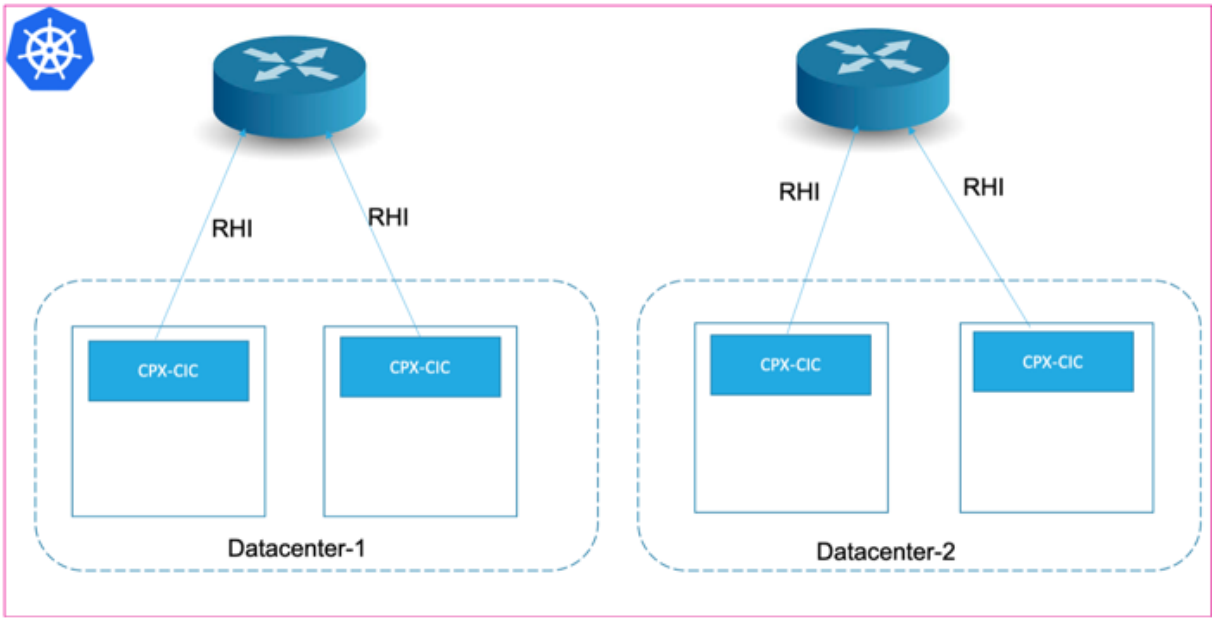
The following table explains the `neighbor` field.

Field	Description	Type	Default value	Required
`address`	IP address for the neighbor router.	string		Yes
`remoteAS`	AS number of the neighbor router.	integer		Yes
`advertisementInterval`	This field sets a minimum interval between the sending of BGP routing updates (in seconds).	integer	10 seconds	Yes

Field	Description	Type	Default value	Required
`ASOriginationInterval`	This field sets the interval of sending AS origination routing updates (in seconds).	integer	10 seconds	Yes

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:

```
apiVersion: v1
kind: 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.

```
metadata:
    annotations:
        service.citrix.com/frontend-ip: 172.217.163.17
```

- Specifying an IP address in the `spec.loadBalancerIP` field of the service specification as follows.

```
spec:
    loadBalancerIP: 172.217.163.17
```

- By automatically assigning a virtual IP address to the service using the IPAM controller provided by NetScaler. 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 , see the documentation.

Service annotation configuration

The NetScaler Ingress Controller provides many service annotations to leverage the various functionalities of the NetScaler. 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.

```
metadata:
    annotations:
        service.citrix.com/service-type-0: 'HTTP'
        service.citrix.com/service-type-1: 'SSL'
```

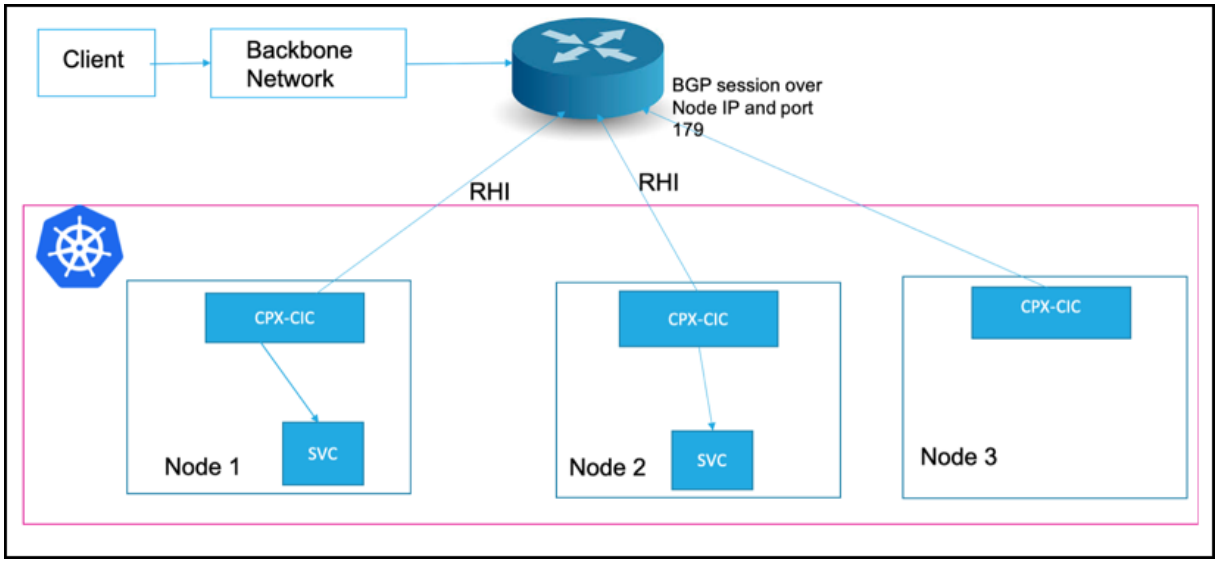
With the help of various annotations provided by the NetScaler Ingress Controller, you can leverage various ADC functionalities like SSL offloading, HTTP rewrite and responder policies, and other custom resource definitions (CRDs).

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 NetScaler Ingress Controller adds all the service pods as a back-end for the load balancing virtual service in NetScaler CPX. This step ensures better high availability and equal distribution to the service pod instances. All nodes running NetScaler 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.



Using Ingress resources

The NetScaler Ingress Controller provides an nt variable `NS_VIP`, which is the external IP Address for all ingress resources. Whenever an ingress resource is added, NetScaler CPX advertises the ingress IP address to the external routers.
The NetScaler 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 NetScaler CPX and the NetScaler Ingress Controller.

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

2. Download the citrix-k8s-cpx-ingress.yml using the following command.

```
wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/docs/configure/cpx-bgp-router/citrix-k8s-cpx-ingress.yml
```

3. Edit the `citrix-k8s-cpx-ingress.yml` file and specify the required values.
   - The argument `–configmap` specifies the ConfigMap location for the NetScaler 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 NetScaler CPX and the NetScaler Ingress Controller.


```
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.

```
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 NetScaler CPX as BGP router. For more information, see the Citrix Helm chart documentation.

Troubleshooting

- By default. NetScaler 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 NetScaler Ingress Controller and NetScaler 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 NetScaler CPX for NSIP and gateway respectively. The same IP address must also be specified as part of the NetScaler Ingress Controller environment variable `NS_IP` to establish the communication between the NetScaler Ingress Controller and NetScaler CPX.
- By default, BGP on NetScaler CPX runs on port 179 and all the BGP traffic coming to the TCP port 179 is handled by NetScaler 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 NetScaler Ingress Controller `BGP_PORT`.
- Use source IP (USIP) mode of NetScaler 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"}'

NetScaler CPX integration with MetalLB in layer 2 mode for on-premises Kubernetes clusters

December 31, 2023

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.

NetScaler provides different options to support the type `LoadBalancer` services in an on-premises environment including:

- Using an external NetScaler VPX or NetScaler 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 NetScaler CPX daemonset running inside the Kubernetes cluster along with a router supporting ECMP over BGP. ECMP router load balances the traffic to multiple NetScaler CPX instances. NetScaler 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 NetScaler CPX.
- Expose the NetScaler 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 NetScaler CPX service on service ports to NetScaler CPX pods. Ingress resources can be configured using the NetScaler 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 NetScaler CPX to achieve ingress capability.

This documentation shows how you can leverage MetalLB along with NetScaler 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 NetScaler CPX service. MetalB 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 NetScaler 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 NetScaler 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.

```
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 NetScaler 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
```

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.

```
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-->
```

NetScaler CPX service installation

Once the metal LB is successfully installed, you can install the NetScaler CPX deployment and a service of type `LoadBalancer`.

To install NetScaler CPX, you can either use the YAML file or Helm charts.

To install NetScaler CPX using the YAML file, perform the following steps:

1. Download the NetScaler CPX deployment manifests.

   ```
   wget https://github.com/citrix/citrix-k8s-ingress-controller/blob/
      master/deployment/baremetal/citrix-k8s-cpx-ingress.yml
   ```

2. Edit the NetScaler 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.

```
kubectl apply – f citrix-k8s-cpx-ingress.yaml
```

4. View the service using the following command:

```
kubectl get svc cpx-service -output yaml
```

You can see that MetalLB allocates an external IP address to the NetScaler CPX service as follows:

```
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.

```
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:

```
curl -v http://192.168.1.240 -H 'host: testdomain.com'
```

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

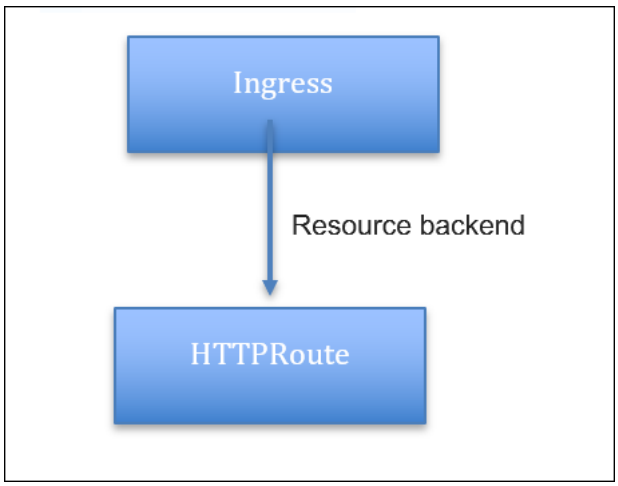
Advanced content routing for Kubernetes Ingress using the HTTPRoute CRD

December 31, 2023

Kubernetes native Ingress offers basic host and path-based routing which is supported by the NetScaler 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 NetScaler 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, NetScaler 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`.


```
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.

```
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.

```
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:
      backend:
        kube:
          service: kuard-purple
          port: 80
<!--NeedCopy-->
```

After you have defined the HTTP routes in the YAML file, deploy the YAML file. In this example, `httproute` is the YAML definition.

```
kubectl apply -f httproute.yaml
```

Profile support for the Listener CRD

December 31, 2023

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, NetScaler 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.

```
    apiVersion: citrix.com/v1
    kind: Listener
    metadata:
      name: test-listener
      namespace: default
    spec:
      vip: x.x.x.x
      port: 80
      protocol: http
      policies:
        httpprofile:
          config:
            websocket: "ENABLED"
<!--NeedCopy-->
```

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`, NetScaler 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.

> **Note:**
>
> The name is auto-generated.

The following example shows how to enable `tcpfastopen` and `HyStart` 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: 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:

```
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, NetScaler 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 NetScaler before NetScaler Ingress Controller is launched.

To enable the advanced SSL profile, use the following command in the NetScaler 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, NetScaler 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.

```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:
      config:
        tls13: "ENABLED"
        hsts: "ENABLED"
```



```
<!--NeedCopy-->
```

You can specify a built-in or pre-configured SSL profile name 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: 443
  certificates:
  - secret:
      name: my-cert
  protocol: https
  policies:
    sslprofile:
      preconfigured: 'ns_default_ssl_profile_secure_frontend'
<!--NeedCopy-->
```

SSL ciphers

The Ingress NetScaler 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.

```
  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-->
```

For information about the list of cipher suites available in NetScaler, see SSL profile infrastructure.

Ensure that NetScaler 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 NetScaler before you apply it to Listener.

```
  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:
      - 'SECURE'
      - 'HIGH'
      - 'MY-CUSTOM-CIPHERS'
<!--NeedCopy-->
```

In the preceding example, `SECURE` and `HIGH` are built-in cipher groups in NetScaler. `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 NetScaler 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 NetScaler. The user-defined cipher group can be used only in a Tier-1 NetScaler.

Analytics profile

Analytics profile enables NetScaler to export the type of transactions or data to an external platform. If you are using NetScaler Observability Exporter to collect metrics and transactions data and export it to endpoints such 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 NetScaler Observability Exporter. Analytics configuration support using ConfigMap.

The following example shows how to enable `webinsight` and `tcpinsight` in the analytics profile.

```
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
       - type: tcpinsight
<!--NeedCopy-->
```

The following example shows how to select the additional parameters for the type of `webinsight` which you want to be exported to NetScaler Observability Exporter. For information about the valid key-value pair, see Analytics Profile.

```
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.

```
  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:
       preconfigured:
       - 'custom-websingiht-analytics-profile'
       - 'custom-tcpinsight-analytics-profile'
<!--NeedCopy-->
```

IP address management using the for Ingress resources

December 31, 2023

IPAM controller is an application provided by NetScaler for IP address management and it runs in parallel to the NetScaler 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 NetScaler Ingress Controller configures the IP address allocated to the Ingress resource as a virtual IP address (VIP) in NetScaler MPX or VPX.

The IPAM controller requires the VIP CustomResourceDefinition (CRD) provided by NetScaler. The VIP CRD is used for internal communication between the NetScaler 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 NetScaler 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 NetScaler VIP CRD which enables communication between the NetScaler 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 NetScaler Ingress Controller

Perform the following steps to deploy the NetScaler Ingress Controller with the IPAM controller argument.

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 NetScaler 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 NetScaler Ingress Controller. Here, you don't need to specify `NS_VIP`.
   - Specify the IPAM controller as an argument using the following:

     args:
     - –ipam
     citrix-ipam-controller

Here is a snippet of a sample NetScaler Ingress Controller YAML file with the IPAM controller argument:

> **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.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: cic-k8s-ingress-controller
spec:
      serviceAccountName: cic-k8s-role
      containers:
      - name: cic-k8s-ingress-controller
        image: "quay.io/citrix/citrix-k8s-ingress-controller:1.36.5"
        env:
          - name: "NS_IP"
            value: "x.x.x.x"
          - name: "NS_USER"
            valueFrom:
            secretKeyRef:
              name: nslogin
              key: username
          - name: "NS_PASSWORD"
            valueFrom:
            secretKeyRef:
              name: nslogin
              key: password
          - name: "EULA"
            value: "yes"
          - name: POD_NAME
            valueFrom:
            fieldRef:
              apiVersion: v1
              fieldPath: metadata.name
          - name: POD_NAMESPACE
            valueFrom:
            fieldRef:
              apiVersion: v1
              fieldPath: metadata.namespace
        args:
          - --ipam citrix-ipam-controller
        imagePullPolicy: Always
```

3. Deploy the NetScaler 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 NetScaler Ingress Controller, see the Deploy NetScaler 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 envirnment 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:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: guestbook-ingress
  annotations:
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.

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

For Ingress without any frontend-ip annotation, the order of IP assignment is as follows:

- If the default `NS_VIP` environment variable is provided, NetScaler Ingress Controller makes a request to IPAM controller only if the range-name (`ingress.citrix.com/ipam-range:`) is provided in the ingress. If the annotation is not provided, `NS_VIP` is used for that ingress.
- If the default `NS_VIP` environment variable is not provided, NetScaler Ingress Controller always make a request to IPAM controller for IP assignment.

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

December 31, 2023

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:

```
ingress.citrix.com/rewrite-responder_crd: <Rewritepolicy Custom-resoure-instance-name>
```

Example:

```
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:

```
ingress.citrix.com/rewrite-responder_crd: '{
<Kubernetes-service-name>: <Rewritepolicy Custom-resoure-instance-name> }
'
```

Example:

```
ingress.citrix.com/rewrite-responder_crd: '{
 "frontendsvc": "blockurlpolicy", "backendsvc": "addresponseheaders" }
'
```

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.

Annotation	Examples	Description
`ingress.citrix.com/bot_crd`	`ingress.citrix.com/bot_crd: '{ "frontend": "botdefense" } '`	Applies the `botdefense` policy to the traffic incoming to the front-end service.
`ingress.citrix.com/auth_crd`	`ingress.citrix.com/auth_crd: '{ "frontend": "authexample" } '`	Applies the `authexample` policy to the front-end service.

Annotation	Examples	Description
`ingress.citrix.com/waf_crd`	`ingress.citrix.com/waf_crd: "wafbasic"`	Applies the WAF policy `wafbasic` to all services in the Ingress
`ingress.citrix.com/ratelimit_crd`	`ingress.citrix.com/ratelimit_crd: "throttlecoffeeperclientip"`	Applies the rate limit policy `throttlecoffeeperclientip` to all services in the Ingress.

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:

```
service.citrix.com/rewrite-responder: <Rewritepolicy Custom-resoure-instance-name>
```

Listener CRD support for Ingress through annotation

December 31, 2023

Ingress is a standard Kubernetes resource that specifies HTTP routing capability to back-end Kubernetes services. NetScaler 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 NetScaler by NetScaler 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.

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. NetScaler 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 NetScaler.

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 NetScaler, 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 required if the Listener resource is in the same namespace as that of Ingress.

Following is an example for the annotation:

```
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:
      - type: webinsight
        parameters:
          allhttpheaders: "ENABLED"
    csvserverConfig:
      rhistate: 'ACTIVE'
```

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 NetScaler 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 NetScaler CPX is used as an ingress device. For NetScaler VPX, VIP is the same as the pod IP address which is automatically configured by NetScaler Ingress Controller.

2. Apply the Listener resource.


```
kubectl apply -f listener.yaml
```

3. Create an Ingress resource (`ingress.yaml`) by referring to the Listener 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:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: kuard
            port:
              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
```



```
2   kind: Ingress
3   metadata:
4     name: my-ingress
5     namespace: default
6     annotations:
7       ingress.citrix.com/listener: my-listener
8       kubernetes.io/ingress.class: "citrix"
9   spec:
10    tls:
11    - secretName: my-secret
12      hosts:
13      - example.com
14    rules:
15    - host: example.com
16      http:
17        paths:
18        - path: /
19          pathType: Prefix
20          backend:
21            service:
22              name: kuard
23              port:
24                number: 80
```

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.

```
1  apiVersion: citrix.com/v1
2  kind: Listener
3  metadata:
4    name: my-listener
5    namespace: default
6  spec:
7    ingressClass: citrix
8    certificates:
9    - secret:
10       name: my-secret
11     # Secret named 'my-secret' in current namespace bound as default
         certificate
12     default: true
13   - secret:
14       # Secret 'other-secret' in demo namespace bound as SNI
           certificate
15       name: other-secret
16       namespace: demo
17   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
```

In the Ingress resource, secrets are not specified as shown in the following example.

```
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:
  # TLS field is empty as the certs are specified in Listener
  rules:
  - host: example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: kuard
            port:
              number: 80
```

Configuring consistent hashing algorithm using NetScaler Ingress Controller

December 31, 2023

Load balancing algorithms define the criteria that the NetScaler 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 NetScaler.
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 NetScaler documentation.

You can now configure the consistent hashing algorithm on NetScaler using NetScaler Ingress Controller. This configuration is enabled with in the NetScaler Ingress Controller using a ConfigMap.

Configure hashing algorithm

A new parameter NS_LB_HASH_ALGO is introduced in the NetScaler 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 NetScaler 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.

```
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 NetScaler Ingress Controller

December 31, 2023

A DNS address record is a mapping of the domain name to the IP address.
When you want to use NetScaler as a DNS resolver, you can add the DNS records on NetScaler using NetScaler Ingress Controller.

For more information on creating DNS records on NetScaler, see the NetScaler documentation.

Adding DNS records for Ingress resources

You need to enable the following environment variable during the NetScaler 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 NetScaler.

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, NetScaler Ingress Controller adds the DNS record on NetScaler. 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, NetScaler Ingress Controller removes the DNS records from the NetScaler.
NetScaler 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 NetScaler:

```
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
  selector:
    app: guestbook
<!--NeedCopy-->
```


Open policy agent support for Kubernetes with NetScaler

December 31, 2023

Open policy agent (OPA) is an open source, general-purpose policy engine that unifies policy enforcement across different technologies and systems. OPA provides a high-level declarative language that lets you specify policy as code and simple APIs to offload policy decision-making from your software. Using OPA, you can decouple policy decision-making from policy enforcement. You can use OPA to enforce policies through NetScaler in a Kubernetes environment.

With OPA, you can create a centralized policy-decision making system for an environment involving multiple NetScalers or multiple devices which are distributed. The advantage of this approach is you have to make changes only on the OPA server for any decision specific changes applicable to multiple devices.

For more information on OPA, see the OPA documentation.

The OPA integration on NetScaler can be supported through HTTP callout, where OPA can be used with or without authentication. An HTTP callout is an HTTP or HTTPS request that the NetScaler appliance generates and sends to an external application as part of the policy evaluation.

For more information on the HTTP callout support, see the HTTP callout documentation.

For more information regarding authentication support, see the Authentication and authorization policies for Kubernetes with NetScaler.

The following diagram provides an overview of how to integrate OPA with the NetScaler cloud native solution.

OPA integration

In the OPA integration diagram, each number represents the corresponding task in the following list:

1. Creating the required Kubernetes objects using Kubernetes commands. This step should include creating the CRD to send the HTTP callout to the OPA server.
2. Configuring NetScaler. NetScaler is automatically configured by NetScaler Ingress Controller based on the created Kubernetes objects.
3. Sending user request for resources from client. The user might get authenticated if authentication CRDs are created.
4. Sending HTTP callout to OPA server in JSON format from NetScaler carrying authorization parameters.
5. Sending authorization decision from OPA server based on the rules defined in REGO, the policy language for OPA.

6. Sending response to the client based on the authorization decision.

Example use cases

Example 1: Allow or deny access to resources based on the client source IP address

Following is an example HTTP callout policy to the OPA server using rewrite policy CRD to allow or deny access to resources based on the client source IP address and the corresponding OPA rules.

In the example, the OPA server responds with `"result": true` if the client source IP address is 192.2.162.0/24, else it responds with `"result":false`.

```
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
  name: calloutexample
spec:
  responder-policies:
    - servicenames:
        - frontend
      responder-policy:
        respondwith:
          http-payload-string: '"HTTP/1.1 401 Access denied\r\n\r\n"' # Access is denied if the respose from OPA server contains false.
        respond-criteria: 'sys.http_callout("callout_name").CONTAINS("false")'
        comment: 'Invalid access'
  httpcallout_policy:
    - name: callout_name
      server_ip: "192.2.156.160" #OPA Server IP
      server_port: 8181 #OPA Server Port
      http_method: 'POST'
      host_expr: "\"192.2.156.160\""
      url_stem_expr: "\"/v1/data/example/allow\"" #URL stem expression to be used
      body_expr: '"{
 \"input\": {
 \"clientinfo\": [{
 \"id\": \"ci\", \"ip\": [\""+ CLIENT.IP.SRC +"\"] }
 ] }
  }
 "' #JSON to OPA server carrying client IP
      headers:
      - name: Content-Type
        expr: '"application/json"'
      return_type: TEXT
      result_expr: "HTTP.RES.BODY(100)"

<!--NeedCopy-->
```


Following are the rules defined through the Rego policy language on the OPA server for the HTTP callout policy for this example:

```
package example

default allow = false                                # unless otherwise defined, allow is false

allow = true {
                                        # allow is true if...
    count(violation) != 0                            # the ip matches regex.
 }


violation[client.id] {
                                # a client is in the violation set if...
    client := input.clientinfo[_]
    regex.match("192.2.162.", client.ip[_])          # the client is not part of 192.2.162.0/24 network.
 }
```

Example 2: Allow or deny access based on user group after authentication

Following is an example HTTP callout policy to the OPA server using rewrite policy CRD to allow or deny access to resources based on user group after authentication and the corresponding OPA rules.

In this example, the OPA server responds with `"result":true` if the user is part of the `beverages` group, else it responds with `"result":false`.

Following is the HTTP callout policy to the OPA server through the rewrite policy CRD.

```
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
  name: calloutexample
spec:
  responder-policies:
    - servicenames:
        - frontend
      responder-policy:
        respondwith:
          http-payload-string: '"HTTP/1.1 401 Access denied\r\n\r\n"' # Access is denied if the respose from OPA server contains false.
        respond-criteria: 'sys.http_callout("callout_name").CONTAINS("false")'
        comment: 'Invalid access'

  httpcallout_policy:
```



```
    - name: callout_name
      server_ip: "192.2.156.160" #OPA Server IP
      server_port: 8181 #OPA Server Port
      http_method: 'POST'
      host_expr: "\"192.2.156.160\""
      url_stem_expr: "\"/v1/data/example/allow\"" #URL stem expression to be used
      body_expr: '"{
 \"input\": {
 \"users\": [{
 \"name\": \""+ AAA.USER.NAME +"\", \"group\": [\""+ AAA.USER.GROUPS +"\"] }
 ] }
  }
 "' #JSON to OPA server carrying username and group information
      headers:
      - name: Content-Type
        expr: '"application/json"'
      return_type: TEXT
      result_expr: "HTTP.RES.BODY(100)"
<!--NeedCopy-->
```

Following are the rules defined through the Rego language on the OPA server for this example:

```
    package example

    default allow = false                                # unless otherwise defined, allow is false

    allow = true {
                                        # allow is true if...
        count(isbeveragesuser) != 0                      # the user is part of beverages group.
     }


    isbeveragesuser[user.name] {
                          # a user is beverages user...
        user := input.users[_]
        user.group[_] == "beverages"                     # if it is part of beverages group.
     }
```

You can perform authentication using the request header (401 based) or through forms based.

Following is a sample authentication policy using request header-based authentication. In this policy, local authentication is used.

```
apiVersion: citrix.com/v1beta1
kind: authpolicy
metadata:
  name: localauth
spec:
```



```
    servicenames:
    - frontend

    authentication_mechanism:
      using_request_header: 'ON'

    authentication_providers:

        - name: "local-auth-provider"
          basic_local_db:
              use_local_auth: 'YES'

    authentication_policies:

        - resource:
            path: []
            method: []
          provider: ["local-auth-provider"]

    authorization_policies:

        - resource:
            path: []
            method: []
            claims: []
<!--NeedCopy-->
```

Following is a sample authentication policy using form-based authentication. In this policy, local-based authentication is used.

```
apiVersion: citrix.com/v1beta1
kind: authpolicy
metadata:
  name: localauth
spec:
    servicenames:
    - frontend

    authentication_mechanism:
      using_forms:
        authentication_host: "fqdn_authenticaton_host"
        authentication_host_cert:
          tls_secret: authhost-tls-cert-secret
        vip: "192.2.156.156"

    authentication_providers:

        - name: "local-auth-provider"
          basic_local_db:
              use_local_auth: 'YES'

    authentication_policies:

```



```
24          - resource:
25              path: []
26              method: []
27            provider: ["local-auth-provider"]
28
29
30      authorization_policies:
31
32          - resource:
33              path: []
34              method: []
35              claims: []
36
37  <!--NeedCopy-->
```

Example 3: Allow or deny access based on authentication attributes obtained during authentication

Following is an example HTTP callout policy to the OPA server using the rewrite policy CRD to allow or deny access based on authentication attributes obtained during authentication and the corresponding OPA rules.

In the example, the OPA server responds with `"result":true'` if the user `memberof` attribute contains `grp1`, else it responds with `"result":false`.

The following is the sample HTTP callout policy to the OPA server through the rewrite policy CRD:

```
1  apiVersion: citrix.com/v1
2  kind: rewritepolicy
3  metadata:
4    name: calloutexample
5  spec:
6    responder-policies:
7      - servicenames:
8          - frontend
9        responder-policy:
10         respondwith:
11           http-payload-string: '"HTTP/1.1 401 Access denied\r\n\r\n"' #
                Access is denied if the respose from OPA server contains
                false.
12         respond-criteria: 'sys.http_callout("callout_name").CONTAINS("
              false")'
13         comment: 'Invalid access'
14
15   httpcallout_policy:
16     - name: callout_name
17       server_ip: "192.2.156.160" #OPA Server IP
18       server_port: 8181 #OPA Server Port
19       http_method: 'POST'
20       host_expr: "\"192.2.156.160\""
```



```
      url_stem_expr: "\"/v1/data/example/allow\"" #URL stem expression to be used
      body_expr: '"{
 \"input\": {
 \"users\": [{
 \"name\": \""+ AAA.USER.NAME +"\", \"attr\": [\""+ aaa.user.attribute("memberof") +"\"] }
 ] }
  }
 "' #JSON to OPA server carrying username and "memberof" attribute information
      headers:
      - name: Content-Type
        expr: '"application/json"'
      return_type: TEXT
      result_expr: "HTTP.RES.BODY(100)"
<!--NeedCopy-->
```

Following are the rules defined through the Rego language on the OPA server for this example:

```
    package example

    default allow = false                                # unless otherwise defined, allow is false

    allow = true {
                                        # allow is true if...
        count(isbeveragesuser) != 0                      # the user is part of grp1.
     }


    isbeveragesuser[user.name] {
                          # a user is part of allow group...
        user := input.users[_]
        regex.match("CN=grp1", user.attr[_])             # if it is part of grp1 group.  }
```

You can perform authentication using request header (401 based) or through forms based. In this example, LDAP authentication is used, where the user `memberof` attribute is obtained from the LDAP server during authentication.

Following is a sample authentication policy using request header-based authentication.

```
apiVersion: citrix.com/v1beta1
kind: authpolicy
metadata:
  name: ldapauth
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
              attributes_to_save: memberof #memberof attribute to be
                 obtained from LDAP server for user

    authentication_policies:
        - resource:
            path: []
            method: []
          provider: ["ldap-auth-provider"]

    authorization_policies:
        - resource:
             path: []
             method: []
             claims: []
<!--NeedCopy-->
```

Following is a sample authentication policy using form-based authentication.

```
apiVersion: citrix.com/v1beta1
kind: authpolicy
metadata:
  name: authhotdrinks
spec:
    servicenames:
    - frontend

    authentication_mechanism:
      using_forms:
        authentication_host: "fqdn_authenticaton_host"
        authentication_host_cert:
          tls_secret: authhost-tls-cert-secret
        vip: "192.2.156.156"

    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
              attributes_to_save: memberof #memberof attribute to be
                 obtained from LDAP server for user
```



```

    authentication_policies:

        - resource:
            path: []
            method: []
          provider: ["ldap-auth-provider"]
<!--NeedCopy-->
```

Exporting metrics directly to Prometheus

December 31, 2023

NetScaler Ingress Controller now supports exporting metrics directly from NetScaler to Prometheus. With NetScaler Ingress Controller, you can automate the configurations required on NetScaler for exporting metrics directly.

Once you export the metrics, you can visualize the exported NetScaler metrics for easier interpretation and understanding using tools such as Grafana.

Configuring direct export of metrics from NetScaler CPX to Prometheus

To enable NetScaler Ingress Controller to configure NetScaler CPX to support direct export of metrics to Prometheus, you need to perform the following steps:

1. Create a Kubernetes secret to enable read-only access for a user. This step is required for NetScaler CPX to export metrics to Prometheus.

```
kubectl create secret generic prom-user --from-literal=username=<prometheus-username> --from-literal=password=<prometheus-password>
<!--NeedCopy-->
```

2. Deploy NetScaler CPX with NetScaler Ingress Controller using the following Helm commands:

```
helm repo add netscaler https://netscaler.github.io/netscaler-helm-charts/

helm install my-release netscaler/netscaler-cpx-with-ingress-controller --set license.accept=yes,nsic.prometheusCredentialSecret=<Secret-for-read-only-user-creation>,analyticsConfig.required=true,analyticsConfig.timeseries.metrics.enable=true,analyticsConfig.timeseries.port=5563,analyticsConfig.timeseries.metrics.mode=prometheus,analyticsConfig.timeseries.metrics.enableNativeScrape=true

```



```
<!--NeedCopy-->
```

The new parameters specified in the command are explained as follows:

- `nsic.prometheusCredentialSecret`: Specifies the Kubernetes secret name for creating the read only user for native Prometheus support.
- `analyticsConfig.timeseries.metrics.enableNativeScrape`: Set this value to **`true`** for directly exporting metrics to Prometheus

3. Add the appropriate Prometheus scrape job under `scrape_configs` in the Prometheus configuration depending on the Prometheus deployment.

   - If your Prometheus server is outside the Kubernetes cluster, add a scrape job under `scrape_configs` in the Prometheus configuration. For a sample Prometheus scrape job, see the Prometheus integration documentation.
   - If your Prometheus server is within the same Kubernetes cluster, add a new Prometheus job to configure Prometheus for directly exporting from a NetScaler CPX pod. For more information, see kubernetes_sd_config. A sample Prometheus job is given as follows:

```
- job_name: 'kubernetes-cpx'
  scheme: http
  metrics_path: /nitro/v1/config/systemfile
  params:
    args: ['filename:metrics_prom_ns_analytics_time_series_profile.log,filelocation:/var/nslog']
    format: ['prometheus']
  basic_auth:
    username:  # Prometheus username set in nsic.prometheusCredentialSecret
    password:  # Prometheus password set in nsic.prometheusCredentialSecret
  scrape_interval: 30s
  kubernetes_sd_configs:
  - role: pod
  relabel_configs:
  - source_labels: [__meta_kubernetes_pod_annotation_netscaler_prometheus_scrape]
    action: keep
    regex: true
  - source_labels: [__address__, __meta_kubernetes_pod_annotation_netscaler_prometheus_port]
    action: replace
    regex: ([^:]+)(?::\d+)?;(\d+)
    replacement: $1:$2
    target_label: __address__
  - source_labels: [__meta_kubernetes_namespace]
    action: replace
    target_label: kubernetes_namespace
  - source_labels: [__meta_kubernetes_pod_name]
```



```
    action: replace
    target_label: kubernetes_pod_name

<!--NeedCopy-->
```

> **Note:**
>
> For more information on Prometheus integration, see the NetScaler Prometheus integration documentation.

Configuring direct export of metrics from NetScaler VPX or NetScaler MPX to Prometheus

To enable NetScaler Ingress Controller to configure NetScaler VPX or NetScaler MPX to support direct export of metrics to Prometheus, you need to perform the following steps:

1. Deploy NetScaler Ingress Controller as a stand-alone pod using the Helm command:

```
    helm repo add netscaler https://netscaler.github.io/netscaler-helm-charts/

    helm install my-release netscaler/citrix-cloud-native --set cic.enabled=true,cic.nsIP=<NSIP>,cic.license.accept=yes,cic.adcCredentialSecret=<Secret-for-NetScaler-credentials>,cic.analyticsConfig.required=true,cic.analyticsConfig.timeseries.metrics.enable=true,cic.analyticsConfig.timeseries.port=5563,cic.analyticsConfig.timeseries.metrics.mode=prometheus,cic.analyticsConfig.timeseries.metrics.enableNativeScrape=true
<!--NeedCopy-->
```

2. Create a system user with read only access for NetScaler VPX. For more details on the user creation, see the NetScaler Prometheus integration documentation.
3. Add a scrape job under `scrape_configs` in the prometheus configuration for enabling Prometheus to scrape from NetScaler VPX. For a sample Prometheus scrape job, see Prometheus configuration.

> **Note:**
>
> The scrape configuration section specifies a set of targets and configuration parameters describing how to scrape them. For more information on NetScaler specific parameters used in the configuration, see the NetScaler documentation.

Configure static route on Ingress NetScaler VPX or MPX

December 31, 2023

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 NetScaler 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 NetScaler instance and the pods for the ingress traffic to reach the microservices.

One of the ways to achieve network connectivity between pods and NetScaler VPX or MPX instance outside the Kubernetes cluster is to configure routes on the NetScaler instance to the overlay network.

You can either do this manually or NetScaler Ingress Controller provides an option to automatically configure the network.

> **Note:**
>
> Ensure that the NetScaler 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 NetScaler instance

Perform the following:

1. On the master node in the Kubernetes cluster, get the podCIDR using the following command:

```
# kubectl get nodes -o jsonpath="{
 range .items[*] }
 {
 'podNetwork: ' }
 {
 .spec.podCIDR }
 {
 '\t' }
 {
 'gateway: ' }
 {
 .status.addresses[0].address }
 {
 '\n' }
 {
 end }
 "

  podNetwork: 10.244.0.0/24    gateway: 10.106.162.108
```



```
20   podNetwork: 10.244.2.0/24    gateway: 10.106.162.109
21   podNetwork: 10.244.1.0/24    gateway: 10.106.162.106
```

If you are using **Calico** CNI then use the following command to get the podCIDR:

```
1  # kubectl get nodes -o jsonpath="{
2   range .items[*] }
3   {
4   'podNetwork: ' }
5   {
6   .metadata.annotations.projectcalico\.org/IPv4IPIPTunnelAddr }
7   {
8   '\tgateway: ' }
9   {
10  .metadata.annotations.projectcalico\.org/IPv4Address }
11  {
12  '\n' }
13  "
14
15   podNetwork: 192.168.109.0       gateway: 10.106.162.108/24
16   podNetwork: 192.168.174.0       gateway: 10.106.162.109/24
17   podNetwork: 192.168.76.128      gateway: 10.106.162.106/24
```

2. Log on to the NetScaler instance.
3. Add route on the NetScaler instance using the podCIDR information. Use the following command:

```
1  add route <pod_network> <podCIDR_netmask> <gateway>
```

For example,

```
1  add route 192.244.0.0 255.255.255.0 192.106.162.108
2
3  add route 192.244.2.0 255.255.255.0 192.106.162.109
4
5  add route 192.244.1.0 255.255.255.0 192.106.162.106
```

Automatically configure route on the NetScaler instance

In the citrix-k8s-ingress-controller.yaml file, you can use an argument, `feature-node-watch` to automatically configure route on the associated NetScaler 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:

spec:
serviceAccountName: cic-k8s-role
containers:

- name: cic-k8s-ingress-controller
image: “quay.io/citrix/citrix-k8s-ingress-controller:1.36.5”
feature-node-watch argument configures route(s) on the Ingress NetScaler
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:

```
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 NetScaler using node controller

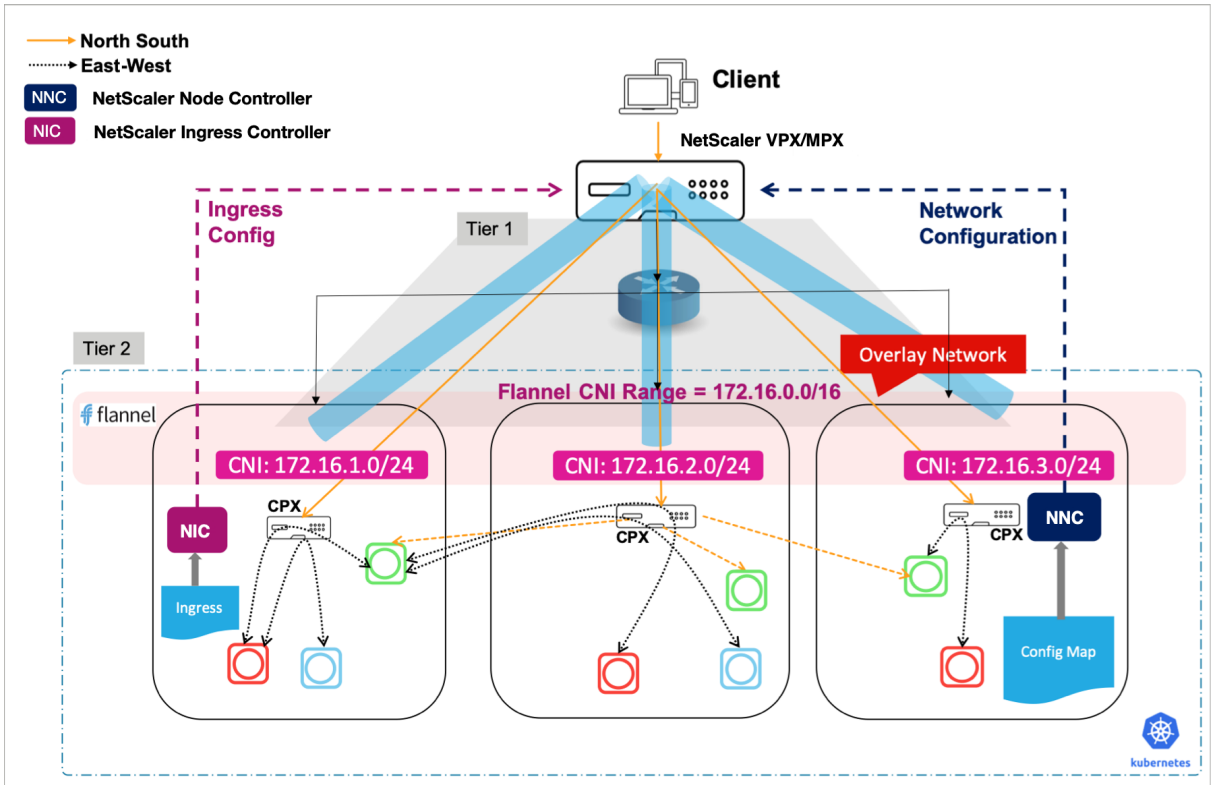
December 31, 2023

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 NetScaler 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 NetScaler.

NetScaler provides a node controller that you can use to create a VXLAN based overlay network between the Kubernetes nodes and the Ingress NetScaler as shown in the following diagram:



> **Note:**
>
> NetScaler Node Controller does not work in a setup where a NetScaler cluster is configured as an ingress device. NetScaler Node Controller requires to establish a Virtual Extensible LAN (VXLAN) tunnel between NetScaler and Kubernetes nodes to configure routes and creating a VXLAN on a NetScaler cluster is not supported.

To establish network connectivity using node controller:

1. Deploy the NetScaler 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 NetScaler 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 NetScaler Ingress Controller is deployed successfully using the following command:

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

2. Deploy the node controller. For information on how to deploy the node controller, see Deploy the Citrix k8s node controller.

Expose Service of type NodePort using Ingress

December 31, 2023

In a single-tier deployment, the Ingress NetScaler (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 NetScaler instance and pods.

As pods run on overlay network, the pod IP addresses are private IP addresses and the Ingress NetScaler 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 NetScaler 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 NetScaler 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-->
```

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:

```
#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 NetScaler 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`.

```
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
<!--NeedCopy-->
```

7. Deploy the Ingress object.

```
kubectl create -f vpx-ingress.yaml
```

Configure pod to pod communication using Calico

December 31, 2023

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:

```
> kubeadm init --pod-network-cidr=192.168.0.0/16
> export KUBECONFIG=/etc/kubernetes/admin.conf
> 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 NetScaler

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 NetScaler 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 NetScaler IP address and the AS number is the AS number that is used in the Ingress NetScaler.

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# ETCD_ENDPOINTS=http://10.102.33.194:6666 ./calicoctl.1 get bgpConfiguration
NAME      LOGSEVERITY   MESHENABLED   ASNUMBER
default   Info          false         64512
```

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:

```
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 NetScaler
    asNumber: 500  # AS number configured on the Ingress NetScaler
<!--NeedCopy-->
```

Deploy the definition file using the following command:

```
> kubectl create -f bgp.yml
```

Add the BGP configurations on the Ingress NetScaler Perform the following:

1. Log on to the NetScaler command-line interface.
2. Enable the BGP feature using the following command:

   ```
   > en feature bgp
     Done
   ```

3. Type `vtysh` and press **Enter**.

   ```
   > vtysh
   ns#
   ```

4. Change to config terminal using the `conf t` command:

   ```
   ns#conf t
   Enter configuration commands, one per line. End with CNTL/Z.
   ns(config)#
   ```

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 adress 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
ns(config-router)# 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 NetScaler 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/OwnedIP  State   Traffic Domain  Type
        -------          -------          ---------------  -----   --------------  ----
1)      0.0.0.0          0.0.0.0          10.102.33.1      UP      0               STATIC
2)      127.0.0.0        255.0.0.0        127.0.0.1        UP      0               PERMANENT
3)      10.102.33.0      255.255.255.0    10.102.33.208    UP      0               DIRECT
4)      192.168.1.0      255.255.255.0    10.102.33.198    UP      0               BGP
5)      192.168.43.128   255.255.255.192  10.102.33.198    UP      0               BGP
6)      192.168.71.64    255.255.255.192  10.102.33.202    UP      0               BGP
 Done
>
```

Once the route is installed, the NetScaler is able to communicate with services that are present in the Kubernetes cluster:


```
> sh lb vserver web-ingress.default.80-web-frontend.default.http-lb
        web-ingress.default.80-web-frontend.default.http-lb (0.0.0.0:0) - HTTP  Type: ADDRESS
        State: UP
        Last state change was at Mon Jul  9 06:23:04 2018
        Time since last state change: 0 days, 00:05:41.530
        Effective State: UP
        Client Idle Timeout: 180 sec
        Down state flush: ENABLED
        Disable Primary Vserver On Down : DISABLED
        Appflow logging: ENABLED
        Port Rewrite : DISABLED
        No. of Bound Services :  2 (Total)       2 (Active)
        Configured Method: LEASTCONNECTION
        Current Method: Round Robin, Reason: Bound service's state changed to UP        BackupMethod: ROUNDROBIN
        Mode: IP
        Persistence: NONE
        Vserver IP and Port insertion: OFF
        Push: DISABLED  Push VServer:
        Push Multi Clients: NO
        Push Label Rule: none
        L2Conn: OFF
        Skip Persistency: None
        Listen Policy: NONE
        IcmpResponse: PASSIVE
        RHIstate: PASSIVE
        New Service Startup Request Rate: 0 PER_SECOND, Increment Interval: 0
        Mac mode Retain Vlan: DISABLED
        DBS_LB: DISABLED
        Process Local: DISABLED
        Traffic Domain: 0
        TROFS Persistence honored: ENABLED
        Retain Connections on Cluster: NO

Bound Service Groups:
1)      Group Name: web-ingress.default.80-web-frontend.default.http-svcgrp

                1) web-ingress.default.80-web-frontend.default.http-svcgrp (192.168.43.129: 80) - HTTP State: UP        Weight: 1
                2) web-ingress.default.80-web-frontend.default.http-svcgrp (192.168.71.65: 80) - HTTP State: UP Weight: 1
Done
```

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.1 get bgpPeer
NAME                      PEERIP          NODE        ASN
bgppeer-global-3040  10.102.33.208    (global)      500
```

View the BGP node status

You can view the status of a BGP node using the following command:

```
>calicoctl node status
IPV4 BGP status
+---------------+-----------+-------+----------+-------------+
| PEER ADDRESS | PEER TYPE | STATE | SINCE | INFO |
+---------------+-----------+-------+----------+-------------+
| 10.102.33.208 | global | up | 16:38:14 | Established |
+---------------+-----------+-------+----------+-------------+
```


Enhancements for Kubernetes service of type LoadBalancer support in the NetScaler Ingress Controller

February 8, 2024

Kubernetes service of type LoadBalancer support in the NetScaler 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

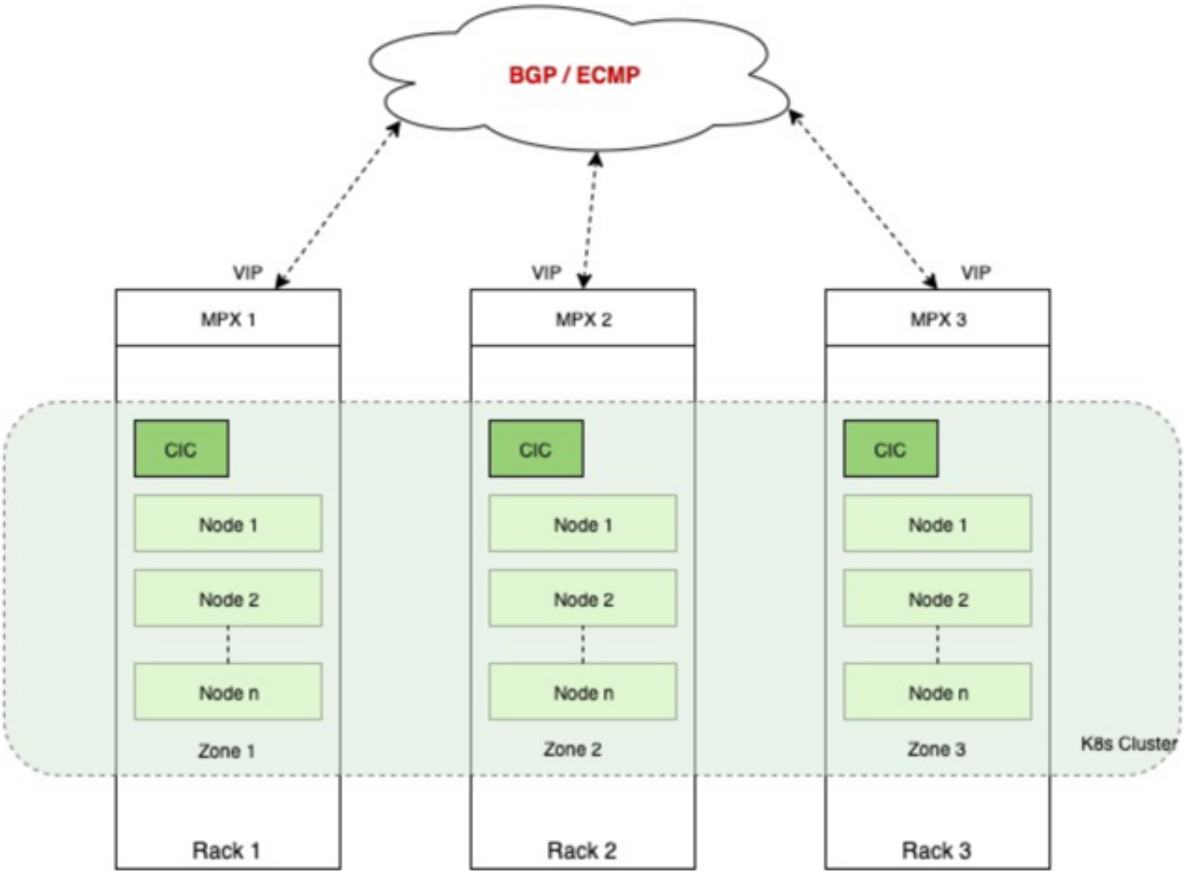
Support for automatic configuration of BGP RHI on NetScaler

Route health injection (RHI) allows the NetScaler 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 NetScaler to support RHI. Using NetScaler Ingress Controllers deployed in a Kubernetes environment, you can automate the configuration on NetScalers to advertise VIPs.

When a service of type `LoadBalancer` is created, the NetScaler Ingress Controller configures a VIP on the NetScaler for the service. If BGP RHI support is enabled for the NetScaler Ingress Controller, it automatically configures NetScaler to advertise the VIP to the BGP network. Using the `service.citrix.com/vipparams` annotation, you can enable IP parameters for the VIP. For example, see the `service.YAML` file in the step 5 of Configuring BGP RHI on NetScalers using the NetScaler Ingress Controller. For information on the supported IP parameters, see nsip configuration.

Advertise and recall VIPs based on the availability of pods

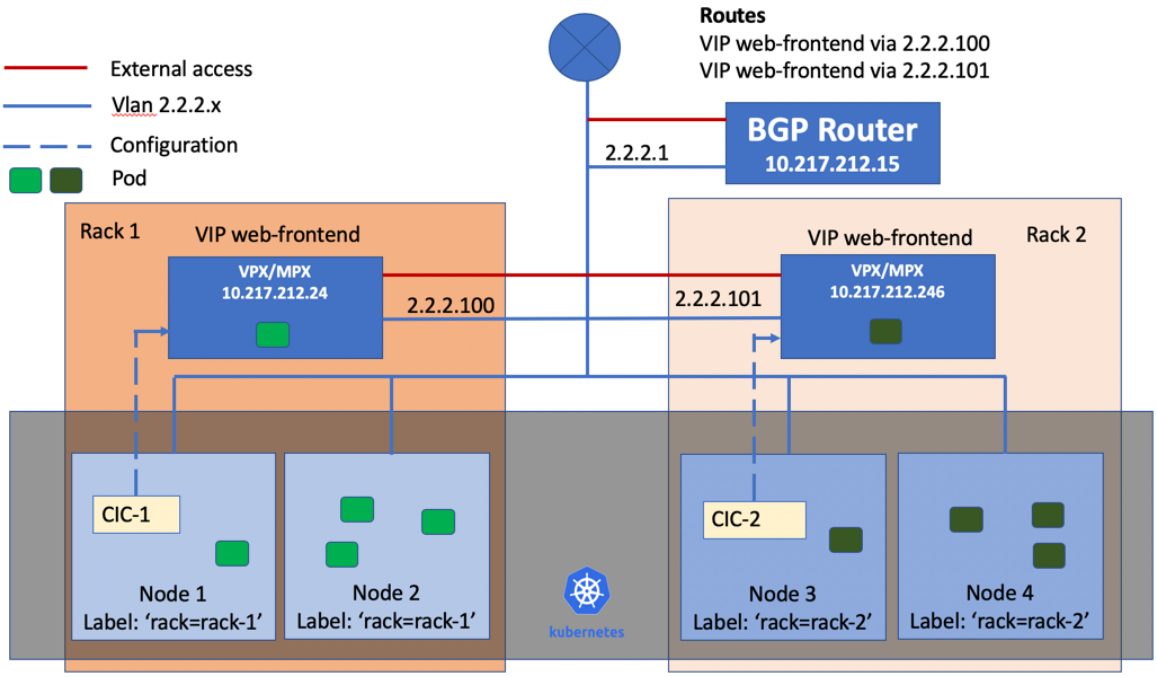
In the topology as shown in the following diagram, nodes in a Kubernetes cluster are physically distributed across three different racks. They are logically grouped into three zones. Each zone has a NetScaler MPX as the Tier-1 ADC and a NetScaler Ingress Controller on the same in the Kubernetes cluster. NetScaler Ingress Controllers in all zones listen to the same Kubernetes API server. So, whenever a service of type `LoadBalancer` is created, all NetScalers in the cluster advertises the same IP address to the BGP fabric. Even, if there is no workload on a zone, the NetScaler in that zone still advertises the IP address.



NetScaler 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 NetScaler Ingress Controller can identify nodes belonging to the same zone. The NetScaler 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 NetScaler on the zone.

Configuring BGP RHI on NetScalers using the NetScaler Ingress Controller

This topic provides information on how to configure BGP RHI on NetScalers using the NetScaler Ingress Controller based on a sample topology. In this topology, nodes in a Kubernetes cluster are deployed across two zones. Each zone has a NetScaler VPX or MPX as the Tier-1 ADC and a NetScaler Ingress Controller for configuring ADC in the Kubernetes cluster. The ADCs are peered using BGP with the upstream router.



Prerequisites

- Configure NetScaler 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 NetScaler Ingress Controller configuration YAML files as follows:

   For zone 1:

   ```
   - name: "NODE_LABELS"
     value: "rack-1"
   - name: "BGP_ADVERTISEMENT"
     value: "True"
   ```

   For zone 2:


```
  - name: "NODE_LABELS"
    value: "rack-2"
  - name: "BGP_ADVERTISEMENT"
    value: "True"
```

A sample `cic.yaml` file for deploying the NetScaler Ingress Controller on zone 1 is provided as follows:

```
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:1.36.5"

    env:
    # Set NetScaler NSIP/SNIP, SNIP in case of HA (mgmt has to be
       enabled)
    - name: "NS_IP"
      value: "10.217.212.24"
   # Set username for Nitro
    - name: "NS_USER"
      valueFrom:
        secretKeyRef:
          name: nslogin
          key: username
   # Set user password for Nitro
    - name: "NS_PASSWORD"
      valueFrom:
        secretKeyRef:
        name: nslogin
        key: password
    - name: "EULA"
      value: "yes"
    - name: "NODE_LABELS"
      value: "rack=rack-1"
    - name: "BGP_ADVERTISEMENT"
      value: "True"
args:
  - --ipam
    citrix-ipam-controller
imagePullPolicy: Always
```

3. Deploy the NetScaler Ingress Controller using the following command.

> **Note:**
>
> You need to deploy the NetScaler Ingress Controller on both racks (per zone).

```
Kubectl create -f cic.yaml
```

4. Deploy a sample application using the `web-frontend-lb.yaml` file.

   Kubectl create -f web-frontend-lb.yaml

   The content of the `web-frontend-lb.yaml` is as follows:

```
apiVersion: v1
kind: Deployment
metadata:
  name: web-frontend
spec:
  selector:
    matchLabels:
      app: web-frontend
  replicas: 4
  template:
    metadata:
      labels:
        app: web-frontend
    spec:
      containers:
      - name: web-frontend
        image: 10.217.6.101:5000/web-test:latest
        ports:
          - containerPort: 80
        imagePullPolicy: Always
```

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:

```
apiVersion: v1
kind: Service
metadata:
  name: web-frontend
  annotations:
    service.citrix.com/class: 'cic-vpx'
    service.citrix.com/frontend-ip: 1.1.1.1
    service.citrix.com/vipparams: '{
 "vserverrhilevel": "ONE_VSERVER", "hostroute": "ENABLED", "metric
    ": 10 }
 '
  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 NetScalers 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

2)   10.217.212.22:30126 State: UP Server Name: 10.217.212.22 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.
```

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 NetScaler Ingress Controller

April 11, 2024

NetScaler Ingress Controller provides option to configure TLS certificates for NetScaler 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. NetScaler determines which certificate to present to the client based on the domain name in the TLS handshake.

NetScaler Ingress Controller handles the certificates in the following three ways:

- NetScaler Ingress Controller default Certificate
- Preconfigured certificates
- TLS section in the Ingress YAML

Prerequisite

For handling TLS certificates using NetScaler Ingress Controller, you need to enable TLS support in NetScaler 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 NetScaler for the application

NetScaler Ingress Controller uses the **TLS** section in the ingress definition as an enabler for TLS support with NetScaler.

> **Note:**
>
> If there is a default certificate or if there are 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.

NetScaler Ingress Controller default certificate

The default secrets provided in NetScaler Ingress Controller can be used to configure SSL and SSL SNI certificates in NetScaler.
You can use **`default`**`-sni-certificate` and **`default`**`-ssl-sni-certificate` arguments to provide a secret to configure non-SNI and SNI certificates respectively. When you specify the argu-

ments in the NetScaler Ingress Controller deployment YAML file, provide the secret name and the namespace where the secret has been deployed in the cluster as following:

- Argument to add in the YAML file to use the default certificate as a non-SNI certificate: `--default-ssl-certificate <NAMESPACE>/<SECRET_NAME>`.
- Argument to add in the YAML file to use the default certificate as an SNI certificate: `--default-ssl-sni-certificate <NAMESPACE>/<SECRET_NAME>`

> **Note:**
>
> If you deploy NetScaler Ingress Controller using a Helm chart or operators, both the default secrets must be deployed in the same namespace where NetScaler Ingress Controller is being deployed. Provide the non-SNI secret name in the `defaultSSLCertSecret` parameter and the SNI secret name in the `defaultSSLSNICertSecret` parameter during the NetScaler Ingress Controller installation. For example, `defaultSSLCertSecret: <non-SNI secret name>`; `defaultSSLSNICertSecret: <SNI secret name>`.

The following is a sample NetScaler Ingress Controller YAML definition file that contains a TLS secret (`hotdrink.secret`) picked from the `ssl` namespace and provided as the NetScaler 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 NetScaler 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"
      value: "nsroot"
    # Set user password for Nitro
    - name: "NS_PASSWORD"
      value: "nsroot"
<!--NeedCopy-->
```

For information about the behaviour of NetScaler Ingress Controller in different scenarios related to Kubernetes ingress in Kubernetes cluster, see the following table.

Default SSL Secret in NetScaler Ingress Controller	Default SSL SNI Secret in NetScaler Ingress Controller	Host in Ingress	Secret in Ingress	Actions
Yes	No	Not provided	Not provided	Bind non-SNI secret as a non-SNI certificate in SSL virtual server.
No	Yes	Not provided	Not Provided	Bind default SNI secret as an SNI certificate in SSL virtual server.
Yes	Yes	Not Provided	Not provided	• Bind non-SNI secret as a non-SNI certificate. • Bind SNI secret as an SNI certificate.
Yes	No	Provided	Not provided	Bind non-SNI secret as a non-SNI certificate in SSL virtual server.
No	Yes	Provided	Not provided	Bind SNI secret as an SNI certificate in SSL virtual server.
Yes	Yes	Provided	Not provided	• Use host as secret as SNI since multiple secrets are getting bound. • Bind non-SNI secret as a non-SNI certificate in SSL virtual server. • Bind SNI secret as an

Default SSL Secret in NetScaler Ingress Controller	Default SSL SNI Secret in NetScaler Ingress Controller	Host in Ingress	Secret in Ingress	Actions
Anything	Anything	Not provided	Provided	Bind secret provided as a non-SNI certificate in SSL virtual server.
Anything	Anything	Provided	Provided	Bind secret provided as an SNI certificate in SSL virtual server.
Anything	Anything	Provided/Not provided	Multiple	Bind all the secrets provided as SNI certificates in SSL virtual server.

For information about behaviour of NetScaler Ingress Controller in different scenarios related to OpenShift route in OpenShift cluster, see the following table.

Default SSL Secret in NetScaler Ingress Controller	Default SSL SNI Secret in NetScaler Ingress Controller	Route type	Key and Cert in Route	Actions
Yes	No	Edge	Not Provided	Bind non-SNI secret as a non-SNI certificate in SSL virtual server.
No	Yes	Edge	Not Provided	Bind default SNI secret as an SNI certificate in SSL virtual server.
Yes	Yes	Edge	Not Provided	• Bind non-SNI secret as a non-SNI certificate. • Bind SNI secret as an SNI certificate.

Default SSL Secret in NetScaler Ingress Controller	Default SSL SNI Secret in NetScaler Ingress Controller	Route type	Key and Cert in Route	Actions
Yes	No	Reencrypt	Not Provided	Bind non-SNI secret as a non-SNI certificate in SSL virtual server.
No	Yes	Reencrypt	Not provided	Bind SNI secret as a SNI certificate in SSL virtual server.
Yes	Yes	Reencrypt	Not provided	• Bind non-SNI secret as a non-SNI certificate in SSL virtual server. • Bind SNI secret as an SNI certificate.
Yes	No	Passthrough	Not provided	Bind non-SNI secret as an SNI certificate.
No	Yes	Passthrough	Not provided	Bind SNI secret as an SNI certificate in SSL virtual server.
Yes	Yes	Passthrough	Not provided	• Bind non-SNI secret as an SNI certificate. • Also, bind non-SNI secret as an SNI certificate in SSL virtual server.
Anything	Anything	Edge	Provided	Bind secret provided as an SNI certificate in SSL virtual server.
Anything	Anything	Reencrypt	Provided	Bind secret provided as an SNI certificate in SSL virtual server.
Anything	Anything	Passthrough	OpenShift doesn't allow	NA

Preconfigured certificates

NetScaler Ingress Controller allows you to use the certkeys that are already configured on the NetScaler. You must provide the details about the certificate using the following annotation in your ingress definition:

```
    ingress.citrix.com/preconfigured-certkey : '{
 \"certs\": \[ {
 \"name\": \"<name>\", \"type\": \"default|sni|ca\" }
  ]  }
 '
```

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:

```
    ingress.citrix.com/preconfigured-certkey : '{
 "certs": [ {
 "name": "certkey1", "type": "default" }
 , {
 "name": "certkey2", "type": "sni" }
  ]  }
  '
```

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 NetScaler and bind them to the applications that are managed by NetScaler Ingress Controller. NetScaler 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 NetScaler Ingress Controller uses these secrets.

With the host section

If the secret name is provided with the host section, NetScaler Ingress Controller binds the secret as an SNI certificate.

```
spec:
  tls:
```



```
  - secretName: fruitjuice.secret
    hosts:
    - items.fruit.juice
<!--NeedCopy-->
```

Without the host section

If the secret name is provided without the host section, NetScaler Ingress Controller binds the secret as a default certificate.

```
spec:
  tls:
  - secretName: colddrink.secret
<!--NeedCopy-->
```

> **Note:**
>
> If there are more than one secret given then NetScaler Ingress Controller binds all the certificates as SNI enabled certificates.

Points to note

1. When, multiple secrets are provided to the NetScaler 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 NetScaler Ingress Controller binds the NetScaler 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 NetScaler.

4. If SNI enabled for SSL virtual server then:

   - Non-SNI (Default) certificate is used for the following HTTPs requests:

     ```
     curl -1 -v -k https://1.1.1.1/

     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:


```
curl -1 -v -k https://items.colddrink.beverages/
```

If any request is received that does not match with certificates, CN name fails.

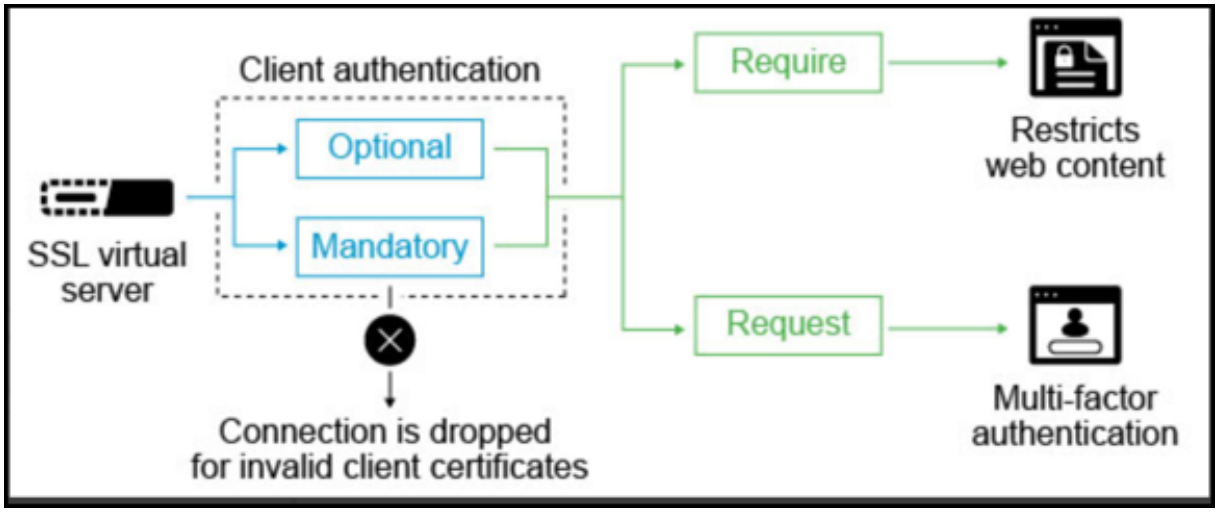
TLS client authentication support in NetScaler

December 31, 2023

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 NetScaler SSL-based virtual servers. With client authentication enabled on a NetScaler SSL virtual server, the NetScaler 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 NetScaler.



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 NetScaler requests the client certificate, but the connection is not dropped. The NetScaler 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 NetScaler.

   The NetScaler Ingress Controller uses the **TLS** section in the Ingress definition as an enabler for TLS support with NetScaler.
   The following is a sample snippet of the Ingress definition:

```
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:

```
    kubectl create secret generic tls-ca --from-file=tls.crt=
       cacerts.pem
```

> **Note:**
>
> You must specify ‘tls.crt=’while creating a secret. This file is used by the NetScaler Ingress Controller while parsing a CA secret.

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.

```
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.

```
ingress.citrix.com/frontend_sslprofile: '{
 "clientauth":"ENABLED", “ clientcert ” : “ optional ” }
 '
```

> **Note:**
>
> The `frontend_sslprofile` only supports the front end Ingress configuration. For more information, see front end configuration.

TLS server authentication support in NetScaler using the NetScaler Ingress Controller

December 31, 2023

Server authentication allows a client to verify the authenticity of the web server that it is accessing. Usually, the NetScaler 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 NetScaler device becomes the SSL client and performs the following:

- 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 NetScaler 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 NetScaler.

   The NetScaler Ingress Controller uses the **TLS** section in the Ingress definition as an enabler for TLS support with NetScaler.
   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 NetScaler 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 NetScaler 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 NetScaler.

```
ingress.citrix.com/backend-ca-secret: '{
"coffee-beverage":"coffee-ca", "tea-beverage":"tea-ca" }
```

Install, link, and update certificates on a NetScaler using the NetScaler Ingress Controller

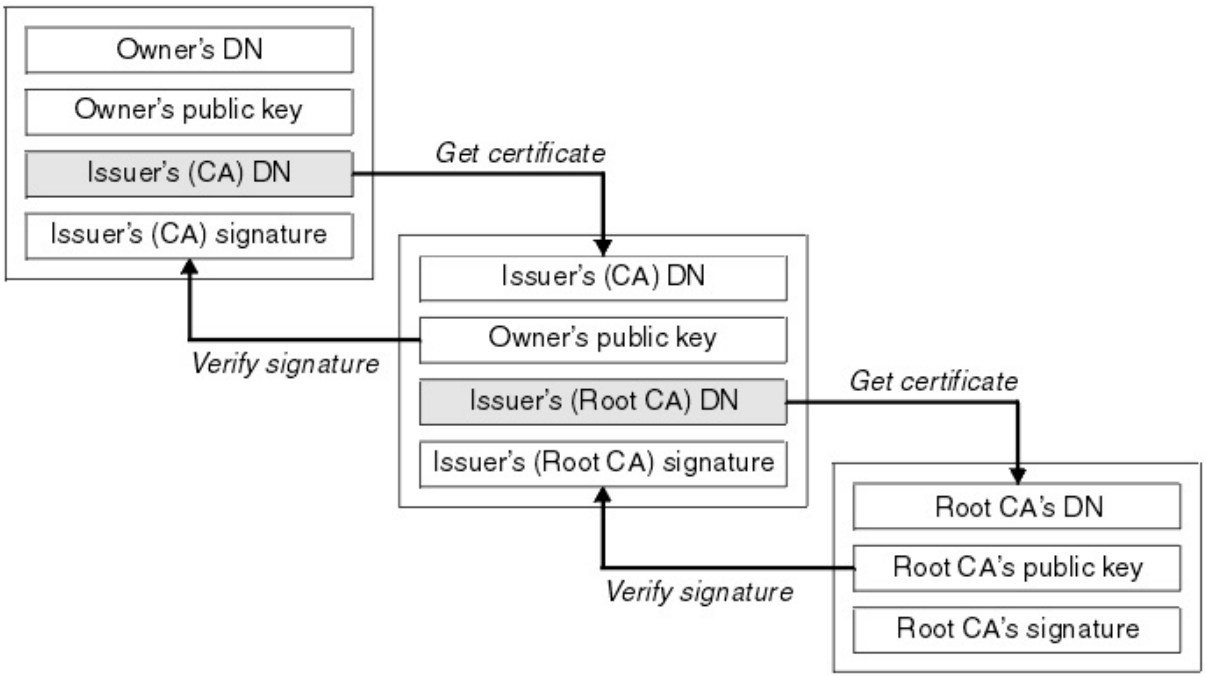
December 31, 2023

On the Ingress NetScaler, 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.

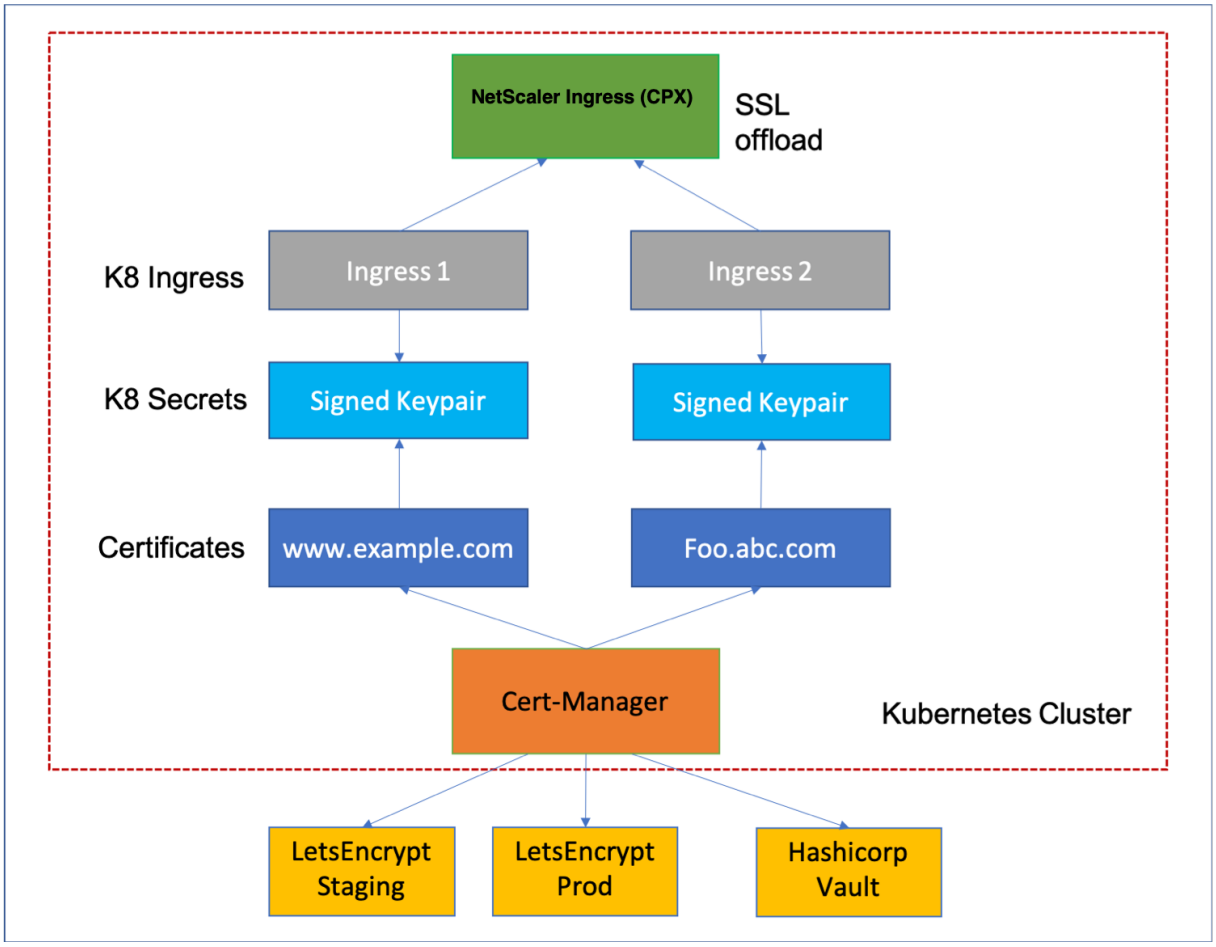
If the intermediate certificate is not installed on the server (where the SSL certificate is installed) it may prevent some browsers, mobile devices, and applications from trusting the SSL certificate. To make the SSL certificate compatible with all clients, it is necessary that the intermediate certificate is installed.



Certificates linking in Kubernetes

The NetScaler 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 NetScaler. 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:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: frontendssl
spec:
  rules:
  - host: frontend.com
    http:
      paths:
      - backend:
          service:
            name: frontend
            port:
              number: 443
```



```
        path: /web-frontend/frontend.php
        pathType: Prefix
  tls:
  - secretName: certchain1

<!--NeedCopy-->
```

On the NetScaler, you can verify if certificates are added to the NetScaler. Perform the following:

1. Log on to the NetScaler command-line interface.
2. Verify if certificates are added to the NetScaler using the following command:

   ```
   >show certkey
   ```

   For sample outputs, see the NetScaler 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
   ```

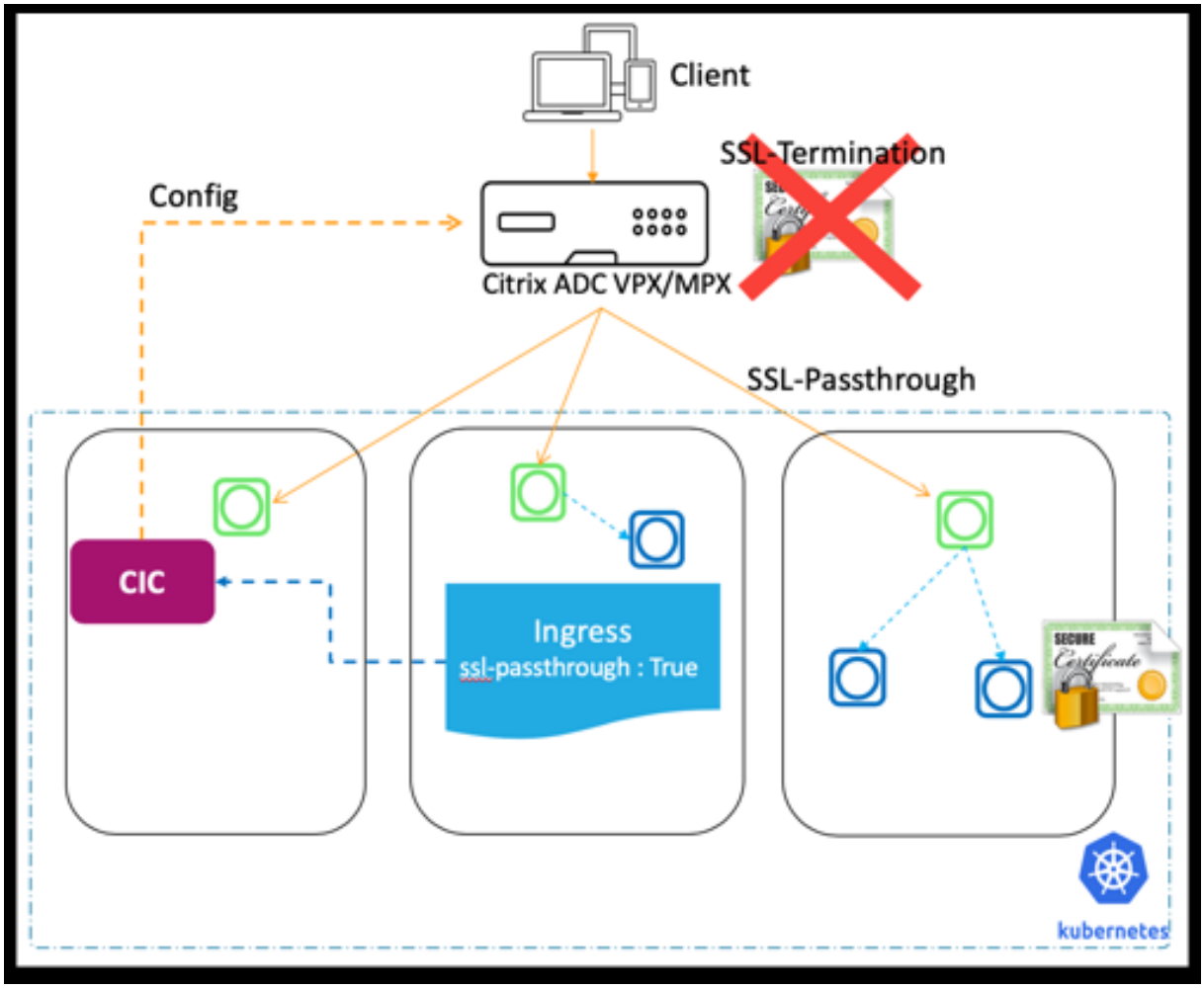
Configure SSL passthrough using Kubernetes Ingress

December 31, 2023

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 NetScaler and SSL traffic is passed through the NetScaler to the back end server. SSL certificate at the back end server is used for the SSL handshake.

The NetScaler Ingress Controller provides the following Ingress annotation that you can use to enable SSL passthrough on the Ingress NetScaler:

```
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 passthrough uses host name (wildcard host name is also supported) and ignores paths given in Ingress.

> **Note:**
>
> The NetScaler 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 NetScaler, 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.

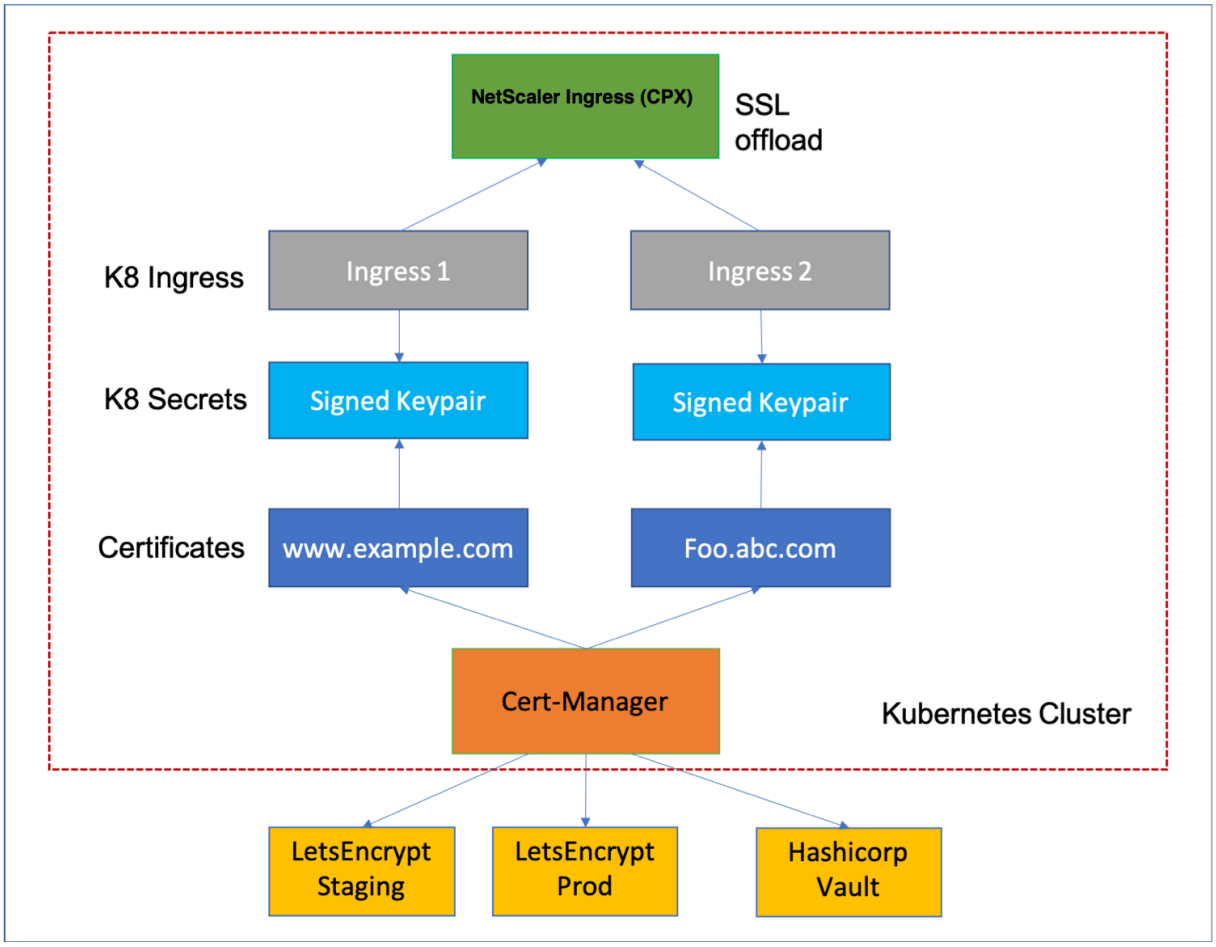
```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    ingress.citrix.com/frontend-ip: x.x.x.x
    ingress.citrix.com/insecure-termination: redirect
    ingress.citrix.com/secure-backend: "True"
    ingress.citrix.com/ssl-passthrough: "True"
    kubernetes.io/ingress.class: citrix
  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

December 31, 2023

NetScaler 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 NetScaler Ingress Controller to configure SSL virtual server on the NetScaler.



For detailed configurations, refer:

- Deploying HTTPS web applications on Kubernetes with NetScaler Ingress Controller and Let's Encrypt using cert-manager
- Deploying HTTPS web application on Kubernetes with NetScaler Ingress Controller and HashiCorp Vault using cert-manager

Deploy HTTPS web application on Kubernetes with the NetScaler Ingress Controller and Let's Encrypt using cert-manager

December 31, 2023

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: DNS01 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 NetScaler 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 NetScaler MPX, VPX, or CPX deployed in Tier 1 or Tier 2 deployment model.

  In the Tier 1 deployment model, NetScaler MPX or VPX is used as an Application Delivery Controller (ADC). The NetScaler Ingress Controller running in Kubernetes cluster configures the virtual services for services running on Kubernetes cluster. NetScaler 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 NetScaler (VPX/MPX) running outside the Kubernetes cluster. This service is created to forward the traffic to NetScaler CPX instances running in the Kubernetes cluster. NetScaler CPX ends the SSL session and load-balances the traffic to actual service pods.
- Deployed the NetScaler 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:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: kuard
  name: kuard
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kuard
  template:
    metadata:
      labels:
        app: kuard
    spec:
      containers:
      - image: gcr.io/kuar-demo/kuard-amd64:1
        imagePullPolicy: Always
        name: kuard
```



```
                  ports:
                  - containerPort: 8080
                    protocol: TCP
<!--NeedCopy-->
```

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

```
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:

```
          apiVersion: v1
          kind: Service
          metadata:
            name: kuard
          spec:
            ports:
            - port: 80
              targetPort: 8080
              protocol: TCP
            selector:
              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
NAME    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kuard   ClusterIP   10.103.49.171   <none>        80/TCP    13s
```

5. Expose this service to outside world by creating an Ingress that is deployed on NetScaler 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 NetScaler 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:**
>
> You must change the value of `spec.rules.host` to the domain that you control. Ensure that a DNS entry exists to route the traffic to NetScaler CPX or VPX.

1. Deploy the Ingress using the following command:

```
% kubectl apply -f ingress.yml
ingress.extensions/kuard created

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 NetScaler CPX or VPX by using the following command:

```
$ 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=MPPL57E3AFY6NMNDGDKN2VT57HEZVOV53Z7DWKH44X2SGLIH4ZWQ====
  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:
  Push Label Rule: none
  Persistence: NONE
  Listen Policy: NONE
  IcmpResponse: PASSIVE
  RHIstate:  PASSIVE
  Traffic Domain: 0
Done

root@cpx-ingress-55c88788fd-qd4rg/# exit
exit
```

3. Verify that the webpage is correctly being served when requested using the `curl` command.

```
% 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, an `Issuer`, scoped to a single namespace, and a `ClusterIssuer`, with cluster-wide scope.

For the NetScaler Ingress Controller to use the Ingress from any namespace, use `ClusterIssuer`. Alternatively, you can also create an `Issuer` 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 called `issuer-letsencrypt-staging.yaml` with the following configuration:

```
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:
      # 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 NetScaler Ingress Controller. If the NetScaler 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`:

```
% kubectl describe issuer letsencrypt-staging

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 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.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    certmanager.io/cluster-issuer: letsencrypt-staging
    kubernetes.io/ingress.class: citrix
  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
```

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.yml

ingress.extensions/kuard configured
% kubectl get ingress kuard
NAME    HOSTS                ADDRESS   PORTS     AGE
kuard   kuard.example.com             80, 443   4h39m
```

Create a certificate CRD resource Alternatively, you can deploy a certificate CRD object independent of the Ingress object. Documentation of "certificate"CRD can be found at HTTP validation.

1. Create the `certificate.yaml` file with the following configuration:

```
        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>
API Version:  cert-manager.io/v1alpha2
Kind:         Certificate
Metadata:
  Creation Timestamp:  2020-01-04T17:36:26Z
  Generation:          1
  Owner References:
    API Version:           extensions/v1beta1
    Block Owner Deletion:  true
    Controller:            true
    Kind:                  Ingress
    Name:                  kuard
    UID:                   2cafa1b4-2ef7-11ea-8ba9-06bea3f4b04a
  Resource Version:        81263
  Self Link:               /apis/cert-manager.io/v1alpha2/
     namespaces/default/certificates/kuard-example-tls
  UID:                     bbfa5e51-2f18-11ea-8ba9-06bea3f4b04a
Spec:
  Dns Names:
    acme.cloudpst.net
  Issuer Ref:
    Group:      cert-manager.io
    Kind:       ClusterIssuer
    Name:       letsencrypt-staging
  Secret Name:  kuard-example-tls
Status:
  Conditions:
    Last Transition Time:  2020-01-04T17:36:28Z
    Message:               Certificate is up to date and has not
       expired
    Reason:                Ready
```



```
    Status:                True
    Type:                  Ready
  Not After:               2020-04-03T16:36:27Z
Events:
  Type    Reason        Age   From          Message
  ----    ------        ----  ----          -------
  Normal  GeneratedKey  24m   cert-manager  Generated a new private key
  Normal  Requested     24m   cert-manager  Created new CertificateRequest resource "kuard-example-tls-3030465986"
  Normal  Issued        24m   cert-manager  Certificate issued successfully
```

3. Verify that the secret resource is created.

```
% kubectl get secret  kuard-example-tls
  NAME                TYPE                DATA   AGE
  kuard-example-tls   kubernetes.io/tls   3      3m13s
```

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
<!--NeedCopy-->
```

   > **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
```



```
letsencrypt-staging   8d
```

7. Verify if the `ClusterIssuer` is properly registered using the command `kubectl describe issuer letsencrypt-staging`:

```
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:

```
certmanager.io/cluster-issuer: "letsencrypt-staging"
<!--NeedCopy-->
```

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-staging
    kubernetes.io/ingress.class: citrix
  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
<!--NeedCopy-->
```

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 Encypt 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:

```
        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-->
```

   After successful validation of the domain name, certificate READY status is set to True.

2. Verify that the certificate is issued.

```
% kubectl get certificate kuard-example-tls

   NAME           READY   SECRET              AGE
   -example-tls   True    kuard-example-tls   10m
```

   You can watch the progress of the certificate as it is issued, using the following command:

   `% kubectl describe certificates kuard-example-tls | tail -n 6`

```
   Not After:                2020-04-04T13:34:23Z
   Events:
   Type    Reason     Age    From          Message
   ----    ------     ----   ----          -------
```



```
    Normal  Requested  11m    cert-manager  Created new CertificateRequest resource "kuard-example-tls-3030465986"
    Normal  Issued     7m21s  cert-manager  Certificate issued successfully
```

Verify certificate in NetScaler

Letsencrypt 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:

```
% kubectl get secret kuard-example-tls

NAME                TYPE                DATA   AGE
kuard-example-tls   kubernetes.io/tls   3      30m
```

   The NetScaler Ingress Controller picks up the secret and binds the certificate to the content switching virtual server on the NetScaler 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 NetScaler 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.
Use 'kubectl describe pod/cpx-ingress-55c88788fd-n2x9r -n default' to see all of the containers in this pod.

% 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
```



```
18   OCSP Stapling: DISABLED
19   HSTS: DISABLED
20   HSTS IncludeSubDomains: NO
21   HSTS Max-Age: 0
22   HSTS Preload: NO
23   SSLv3: ENABLED  TLSv1.0: ENABLED  TLSv1.1: ENABLED  TLSv1.2:
        ENABLED  TLSv1.3: DISABLED
24   Push Encryption Trigger: Always
25   Send Close-Notify: YES
26   Strict Sig-Digest Check: DISABLED
27   Zero RTT Early Data: DISABLED
28   DHE Key Exchange With PSK: NO
29   Tickets Per Authentication Context: 1
30 , P_256, P_384, P_224, P_5216) CertKey Name: k8s-
      GVWNYGVZKKRHKF7MZVTLOAEZYBS Server Certificate for SNI
31
32 7) Cipher Name: DEFAULT
33   Description: Default cipher list with encryption strength >= 128
        bit
34 Done
35
36 % cli_script.sh 'sh certkey'
37 1) Name: k8s-GVWNYGVZKKRHKF7MZVTLOAEZYBS
38   Cert Path: k8s-GVWNYGVZKKRHKF7MZVTLOAEZYBS.crt
39   Key Path: k8s-GVWNYGVZKKRHKF7MZVTLOAEZYBS.key
40   Format: PEM
41   Status: Valid,   Days to expiration:89
42   Certificate Expiry Monitor: ENABLED
43   Expiry Notification period: 30 days
44   Certificate Type: "Client Certificate" "Server Certificate"
45   Version: 3
46   Serial Number: 03B2B57EA9E61A93F1D05EA3272FA95203C2
47   Signature Algorithm: sha256WithRSAEncryption
48   Issuer:  C=US,O=Let's Encrypt,CN=Let's Encrypt Authority X3
49   Validity
50     Not Before: Jan  5 13:34:23 2020 GMT
51     Not After : Apr  4 13:34:23 2020 GMT
52   Subject:  CN=acme.cloudpst.net
53   Public Key Algorithm: rsaEncryption
54   Public Key size: 2048
55   Ocsp Response Status: NONE
56 2) Name: k8s-GVWNYGVZKKRHKF7MZVTLOAEZYBS_ic1
57   Cert Path: k8s-GVWNYGVZKKRHKF7MZVTLOAEZYBS.crt_ic1
58   Format: PEM
59   Status: Valid,   Days to expiration:437
60   Certificate Expiry Monitor: ENABLED
61   Expiry Notification period: 30 days
62   Certificate Type: "Intermediate CA"
63   Version: 3
64   Serial Number: 0A0141420000015385736A0B85ECA708
65   Signature Algorithm: sha256WithRSAEncryption
66   Issuer:  O=Digital Signature Trust Co.,CN=DST Root CA X3
67   Validity
```



```
68      Not Before: Mar 17 16:40:46 2016 GMT
69      Not After : Mar 17 16:40:46 2021 GMT
70    Subject:  C=US,O=Let's Encrypt,CN=Let's Encrypt Authority X3
71    Public Key Algorithm: rsaEncryption
72    Public Key size: 2048
73    Ocsp Response Status: NONE
74  Done
```

The HTTPS webserver is now UP with a fake LE signed certificate. Next step is to move to production with the actual Let's Encrypt certificates.

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`

```
1  apiVersion: cert-manager.io/v1alpha2
2  kind: ClusterIssuer
3  metadata:
4    name: letsencrypt-prod
5  spec:
6    acme:
7      # You must replace this email address with your own.
8      # Let's Encrypt will use this to contact you about expiring
9      # certificates, and issues related to your account.
10     email: user@example.com
11     server: https://acme-v02.api.letsencrypt.org/directory
12     privateKeySecretRef:
13       # Secret resource used to store the account's private key.
14       name: example-issuer-account-key
15     # Add a single challenge solver, HTTP01 using citrix
16     solvers:
17     - http01:
18         ingress:
19           class: citrix
20 <!--NeedCopy-->
```

> **Note:**
>
> Replace `user@example.com` with your email address.

Deploy the file using the following command:

```
1  % kubectl apply -f letsencrypt-prod.yaml
2
```



```
 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:
  Type    Reason        Age   From          Message
  ----    ------        ----  ----          -------
  Normal  OrderCreated  22s   cert-manager  Created Order resource "
      kuard-example-tls-1754626579"
```

Also you can view the major certificate events using the `kubectl events` command:


```
kubectl get events

LAST SEEN   TYPE     REASON              KIND          MESSAGE
36s         Normal   Started             Challenge     Challenge scheduled for processing
36s         Normal   Created             Order         Created Challenge resource "kuard-example-tls-1754626579-0" for domain "acme.cloudpst.net"
38s         Normal   OrderCreated        Certificate   Created Order resource "kuard-example-tls-1754626579"
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:

```
% kubectl get po -n cert-manager

NAME                                       READY   STATUS      RESTARTS   AGE
cert-manager-76d48d47bf-5w4vx              1/1     Running     0          23h
cert-manager-webhook-67cfb86d56-6qtxr      1/1     Running     0          23h
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:

```
% kubectl logs -f cert-manager-76d48d47bf-5w4vx -n cert-manager
```

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.cloudpst.net
              certmanager.k8s.io/common-name: acme.cloudpst.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
ca.crt:   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 NetScaler Ingress Controller

If a Kubernetes secret is generated and complete, but it is not uploaded to the NetScaler, you can analyze the logs from the NetScaler Ingress Controller using the following command.

```
% kubectl logs -f cpx-ingress-685c8bc976-zgz8q
```

Deploy an HTTPS web application on Kubernetes with NetScaler Ingress Controller and HashiCorp Vault using cert-manager

December 31, 2023

For ingress resources deployed with the NetScaler 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:

- NetScaler 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 NetScaler MPX, VPX, or CPX in Tier 1 or Tier 2 deployment model.

  In the Tier 1 deployment model, NetScaler MPX or VPX is used as an Application Delivery Controller (ADC). The NetScaler Ingress Controller running in the Kubernetes cluster configures the virtual services for the services running on the Kubernetes cluster. NetScaler 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 NetScaler (VPX/MPX) running outside the Kubernetes cluster to forward the traffic to NetScaler CPX instances running in the Kubernetes cluster. NetScaler CPX ends the SSL session and load-balances the traffic to actual service pods.
- Deployed NetScaler 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 NetScaler CPX used as the ingress device. When a NetScaler VPX or MPX is used as the ingress device, the steps are the same except the steps to verify the ingress configuration in the NetScaler.

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.

```
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.

```
% 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

NAME                           TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
service/cert-manager-webhook   ClusterIP   10.108.161.154   <none>        443/TCP   13d

NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/cert-manager          1/1     1            1           13d
deployment.apps/cert-manager-webhook  1/1     1            1           13d

NAME                                             DESIRED   CURRENT   READY   AGE
replicaset.apps/cert-manager-77fd74fb64          1         1         1       13d
replicaset.apps/cert-manager-webhook-67bf86d45   1         1         1       13d

NAME                                             COMPLETIONS   DURATION   AGE
job.batch/cert-manager-webhook-ca-sync           1/1           22s        13d
job.batch/cert-manager-webhook-ca-sync-1549756800   1/1        21s        10d
job.batch/cert-manager-webhook-ca-sync-1550361600   1/1        19s        3d8h

NAME                                       SCHEDULE    SUSPEND   ACTIVE   LAST SCHEDULE   AGE
cronjob.batch/cert-manager-webhook-ca-sync   @weekly     False     0        3d8h            13d
```


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.

```
        apiVersion: apps/v1
        kind: Deployment
        metadata:
          name: kuard
        spec:
          replicas: 1
          selector:
            matchLabels:
              app: kuard
          template:
            metadata:
              labels:
                app: kuard
            spec:
              containers:
              - image: gcr.io/kuar-demo/kuard-amd64:1
                imagePullPolicy: Always
                name: kuard
                ports:
                - containerPort: 8080
<!--NeedCopy-->
```

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
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.

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



```
8           targetPort: 8080
9           protocol: TCP
10        selector:
11          app: kuard
12 <!--NeedCopy-->
```

4. Deploy and verify the service using the following command.

```
1 % kubectl create -f service.yaml
2 service/kuard created
3 % kubectl get svc kuard
4 NAME    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
5 kuard   ClusterIP   10.103.49.171   <none>        80/TCP    13s
```

5. Expose this service to the outside world by creating an Ingress that is deployed on NetScaler CPX or VPX as Content switching virtual server.

> **Note:**
>
> Ensure that you change `kubernetes.io/ingress.class` to your ingress class on which NetScaler Ingress Controller is started.

```
1     apiVersion: networking.k8s.io/v1
2     kind: Ingress
3     metadata:
4       annotations:
5         kubernetes.io/ingress.class: citrix
6       name: kuard
7     spec:
8       rules:
9       - host: kuard.example.com
10        http:
11          paths:
12          - backend:
13              service:
14                name: kuard
15                port:
16                  number: 80
17            path: /
18            pathType: Prefix
19  <!--NeedCopy-->
```

> **Note:**
>
> Change the value of `spec.rules.host` to the domain that you control. Ensure that a DNS entry exists to route the traffic to NetScaler CPX or VPX.

6. Deploy the Ingress using the following command.

```
1 % kubectl apply -f ingress.yml
2 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 NetScaler 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
     VBGFO7NZXV7SCI4LSDJML2Q5X6FSNK6NXQPWGMDOYGBW2IMOGQ====
  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:
  Push Label Rule: none
  Listen Policy: NONE
  IcmpResponse: PASSIVE
  RHIstate:  PASSIVE
  Traffic Domain: 0
Done
root@cpx-ingress-5b85d7c69d-ngd72:/# exit
exit
```

8. Verify if the page is correctly being served when requested using the `curl` command.

```
% 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
```

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 }
 " pki

# Set the max TTL for the root CA to 10 years
% vault secrets tune -max-lease-ttl=87600h "${
 PKI_ROOT }
 "

% vault write -format=json "${
 PKI_ROOT }
 "/root/generate/internal \
 common_name="${
 DOMAIN }
  CA root" ttl=87600h | tee \
>(jq -r .data.certificate > ca.pem) \
>(jq -r .data.issuing_ca > issuing_ca.pem) \
>(jq -r .data.private_key > ca-key.pem)

#Configure the CA and CRL URLs:

% vault write "${
 PKI_ROOT }
 "/config/urls \
       issuing_certificates="${
 VAULT_ADDR }
 /v1/${
```



```
PKI_ROOT }
/ca" \
      crl_distribution_points="${
VAULT_ADDR }
/v1/${
PKI_ROOT }
/crl"
<!--NeedCopy-->
```

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:

```
    % 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-->
```

2. Generate CSR for `${ DOMAIN }` that needs to be signed by the root CA. The key is stored internally to the Vault. Use the following command:

```
      % vault write -format=json "${
PKI_INT }
"/intermediate/generate/internal \
      common_name="${
DOMAIN }
 CA intermediate" ttl=26280h | tee \
      >(jq -r .data.csr > pki_int.csr) \
      >(jq -r .data.private_key > pki_int.pem)

   <!--NeedCopy-->
```

3. Generate and sign the `${ DOMAIN }` certificate as an intermediate CA using root CA, store it as `intermediate.cert.pem`. Use the following command:

```
    % vault write -format=json "${
PKI_ROOT }
"/root/sign-intermediate csr=@pki_int.csr
            format=pem_bundle ttl=26280h \
```



```
                | jq -r '.data.certificate' > intermediate.cert.pem
  <!--NeedCopy-->
```

If you are using an external root CA, skip the preceding step and sign the CSR manually using the root CA.

4. Once the CSR is signed and the root CA returns a certificate, it needs to be added back into the Vault using the following command:

```
    % vault write "${
PKI_INT }
"/intermediate/set-signed certificate=@intermediate.cert.pem
 <!--NeedCopy-->
```

5. Set the CA and CRL location using the following command.

```
    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.

```
    # with a Max TTL of 90 days
    vault write ${
PKI_INT }
/roles/kube-ingress \
             allowed_domains=${
```



```
DOMAIN }
 \
             allow_subdomains=true \
             max_ttl="2160h" \
             require_cn=false
 <!--NeedCopy-->
```

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.

```yaml
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: cert-manager-vault-approle
  namespace: cert-manager
data:
  secretId: "NmExNzRjMjAtZjZkZS1hNTNjLTc0ZDItNjAxOGZjY2VmZjY0Cg=="
```


> **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.

2. 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:
        path: approle
        roleId: "db02de05-fa39-4855-059b-67221c5c2f63"
        secretRef:
          name: cert-manager-vault-approle
          key: secretId
```

   `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.


```
% 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.

```
        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
```



```
20          tls:
21          - hosts:
22            - kuard.example.com
23            secretName: kuard-example-tls
24 <!--NeedCopy-->
```

2. Deploy the modified ingress as follows.

```
1   % kubectl apply -f ingress.yml
2   ingress.extensions/kuard created
3
4   % kubectl get ingress kuard
5   NAME    HOSTS               ADDRESS   PORTS     AGE
6   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.

```
1   % kubectl describe certificates kuard-example-tls  | grep -A5
       Events
2   Events:
3   Type    Reason      Age   From          Message
4   ----    ------      ----  ----          -------
5   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.

```
1 apiVersion: cert-manager.io/v1
2 kind: Certificate
3 metadata:
4   name: kuard-example-tls
5   namespace: default
6 spec:
7   secretName: kuard-example-tls
```



```
  issuerRef:
    kind: ClusterIssuer
    name: vault-issuer
  commonName: kuard.example.com
  duration: 720h
  #Renew before 7 days of expiry
  renewBefore: 168h
  commonName: kuard.example.com
  dnsNames:
  - www.kuard.example.com
```

The certificate has CN=`kuard.example.com` and SAN=`Kuard.example.com`,`www.kuard.example.com`.
`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.

   % 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
Events:
  Type    Reason      Age   From          Message
  ----    ------      ----  ----          -------
  Normal  CertIssued  48s   cert-manager  Certificate issued
     successfully > **Note** > > You may encounter some errors due to
     the Vault policies. If you encounter any such errors, return to
     the Vault and fix it.
```

After successful signing, a `kubernetes.io/tls` secret is created with the `secretName` specified in the `Certificate` resource.

```
% kubectl get secret kuard-example-tls
NAME                TYPE                DATA   AGE
kuard-exmaple-tls   kubernetes.io/tls   3      4m20s
```

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.


```
      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
              pathType: Prefix
              path: /
        tls:
        - hosts:
          - kuard.example.com
          secretName: kuard-example-tls
```

2. Deploy the ingress using the following command.

```
% 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 NetScaler

Once the certificate is successfully generated, NetScaler 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 NetScaler CPX and verify if the Certificate is bound to the SSL virtual server.

```
% 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
  Session Reuse: ENABLED Timeout: 120 seconds
  Cipher Redirect: DISABLED
  SSLv2 Redirect: DISABLED
  ClearText Port: 0
```



```
11    Client Auth: DISABLED
12    SSL Redirect: DISABLED
13    Non FIPS Ciphers: DISABLED
14    SNI: ENABLED
15    OCSP Stapling: DISABLED
16    HSTS: DISABLED
17    HSTS IncludeSubDomains: NO
18    HSTS Max-Age: 0
19    SSLv2: DISABLED  SSLv3: ENABLED  TLSv1.0: ENABLED  TLSv1.1:
         ENABLED  TLSv1.2: ENABLED  TLSv1.3: DISABLED
20    Push Encryption Trigger: Always
21    Send Close-Notify: YES
22    Strict Sig-Digest Check: DISABLED
23    Zero RTT Early Data: DISABLED
24    DHE Key Exchange With PSK: NO
25    Tickets Per Authentication Context: 1
26  Done
27
28  root@cpx-ingress-668bf6695f-4fwh8:/# cli_script.sh 'shsslvs k8s
       -10.244.3.148:443:ssl'
29  exec: shsslvs k8s-10.244.3.148:443:ssl
30
31    Advanced SSL configuration for VServer k8s-10.244.3.148:443:ssl:
32    DH: DISABLED
33    DH Private-Key Exponent Size Limit: DISABLED Ephemeral RSA:
         ENABLED Refresh Count: 0
34    Session Reuse: ENABLED Timeout: 120 seconds
35    Cipher Redirect: DISABLED
36    SSLv2 Redirect: DISABLED
37    ClearText Port: 0
38    Client Auth: DISABLED
39    SSL Redirect: DISABLED
40    Non FIPS Ciphers: DISABLED
41    SNI: ENABLED
42    OCSP Stapling: DISABLED
43    HSTS: DISABLED
44    HSTS IncludeSubDomains: NO
45    HSTS Max-Age: 0
46    SSLv2: DISABLED  SSLv3: ENABLED  TLSv1.0: ENABLED  TLSv1.1:
         ENABLED  TLSv1.2: ENABLED  TLSv1.3: DISABLED
47    Push Encryption Trigger: Always
48    Send Close-Notify: YES
49    Strict Sig-Digest Check: DISABLED
50    Zero RTT Early Data: DISABLED
51    DHE Key Exchange With PSK: NO
52    Tickets Per Authentication Context: 1
53  , P_256, P_384, P_224, P_5216) CertKey Name: k8s-
       LMO3O3U6KC6WXKCBJAQY6K6X6JO Server Certificate for SNI
54
55  7) Cipher Name: DEFAULT
56    Description: Default cipher list with encryption strength >= 128
         bit
57  Done
```



```

root@cpx-ingress-668bf6695f-4fwh8:/# cli_script.sh 'sh certkey k8s-LMO3O3U6KC6WXKCBJAQY6K6X6JO'
exec: sh certkey k8s-LMO3O3U6KC6WXKCBJAQY6K6X6JO
  Name: k8s-LMO3O3U6KC6WXKCBJAQY6K6X6JO Status: Valid,   Days to expiration:0
  Version: 3
  Serial Number: 524C1D9306F784A2F5277C05C2A120D5258D9A2F
  Signature Algorithm: sha256WithRSAEncryption
  Issuer:  CN=example.com CA intermediate
  Validity
    Not Before: Feb 26 06:48:39 2019 GMT
    Not After : Feb 27 06:49:09 2019 GMT
  Certificate Type: "Client Certificate" "Server Certificate"
  Subject:  CN=kuard.example.com
  Public Key Algorithm: rsaEncryption
  Public Key size: 2048
  Ocsp Response Status: NONE
  2) URI:http://127.0.0.1:8200/v1/pki_int/crl
  3) VServer name: k8s-10.244.3.148:443:ssl Server Certificate for SNI
Done
```

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.

```
% curl -sS -D - https://kuard.example.com -k -o /dev/null
HTTP/1.1 200 OK
Content-Length: 1472
Content-Type: text/html
Date: Tue, 11 May 2021 20:39:23 GMT
```

Enable NetScaler certificate validation in the NetScaler Ingress Controller

December 31, 2023

The NetScaler Ingress Controller provides an option to ensure secure communication between the NetScaler Ingress Controller and NetScaler by using the HTTPS protocol. You can achieve this by using pre-loaded certificates in the NetScaler. As an extra measure to avoid any possible man-in-the-middle (MITM) attack, the NetScaler Ingress Controller also allows you to validate the SSL server certificate provided by the NetScaler.

To enable certificate signature and common name validation of the ADC server certificate by the NetScaler Ingress Controller, security administrators can optionally install signed (or self-signed) certificates in the NetScaler and configure the NetScaler Ingress Controller with the corresponding CA certificate bundle. Once the validation is enabled and CA certificate bundles are configured, the NetScaler Ingress Controller starts validating the certificate (including certificate name validation). If the validation fails, the NetScaler 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 NetScaler Ingress Controller as follows.

Prerequisites

- For enabling certificate validation, you must configure a NetScaler with proper SSL server certificates (with proper server name or IP address in certificate subject). For more information, see NetScaler documentation.
- The CA certificate for the installed server certificate-key pair is used to configure the NetScaler Ingress Controller to enable validation of these certificates.

Configure the NetScaler 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 NetScaler Ingress Controller can access it on a mounted storage volume.

To generate a Kubernetes secret for an existing certificate, use the following `kubectl` command:

```
$ 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:

```
  apiVersion: v1
  kind: Secret
  metadata:
    name: ciccacert
    data:
```



```
6          myCA.pem: <base64 encoded cert>
```

The following is a sample YAML file with the NetScaler Ingress Controller configuration for enabling certificate validation.

```
1  kind: Pod
2  metadata:
3    name: cic
4    labels:
5      app: cic
6  spec:
7    serviceAccountName: cpx
8    # Make secret available as a volume
9    volumes:
10   - name: certs
11     secret:
12       secretName: ciccacert
13   containers:
14   - name: cic
15     image: "xxxx"
16     imagePullPolicy: Always
17     args: []
18     # Mounting certs in a volume path
19     volumeMounts:
20     - name: certs
21       mountPath: <Path to mount the certificate>
22       readOnly: true
23     env:
24     # Set NetScaler ADM Management IP
25     - name: "NS_IP"
26       value: "xx.xx.xx.xx"
27     # Set port for Nitro
28     - name: "NS_PORT"
29       value: "xx"
30     # Set Protocol for Nitro
31     - name: "NS_PROTOCOL"
32       # Enable HTTPS protocol for secure communication
33       value: "HTTPS"
34     # Set username for Nitro
35     - name: "NS_USER"
36       value: "nsroot"
37     # Set user password for Nitro
38     - name: "NS_PASSWORD"
39       value: "nsroot"
40     # Certificate validation configurations
41     - name: "NS_VALIDATE_CERT"
42       value: "yes"
43     - name: "NS_CACERT_PATH"
44       value: " <Mounted volume path>/myCA.pem"
45 <!--NeedCopy-->
```

As specified in the example YAML file, following are the specific changes required for enabling certificate validation in the NetScaler 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

December 31, 2023

While communicating with the API server from NetScaler Ingress Controller or GSLB ingress, you have the option to disable the API server certificate verification on NetScaler Ingress Controller.

Disable API server certificate verification on NetScaler Ingress Controller or GSLB ingress

When you deploy NetScaler Ingress Controller using YAML, you can disable the API server certificate verification by providing the following argument in the NetScaler Ingress Controller deployment YAML file.


```
1  args:
2     - --disable-apiserver-cert-verify
3       true
```

When you deploy NetScaler Ingress Controller using Helm charts, the parameter `disableAPIServerCertVeri` can be mentioned as `True` in the Helm values file as follows:

```
1     disableAPIServerCertVerify: True
```

Create a self-signed certificate and linking into Kubernetes secret

December 31, 2023

Use the steps in the procedure to create a self-signed certificate using OpenSSL and link into Kubernetes secret. You can use this secret to secure your Ingress.

Create a self-signed certificate

You can create a TLS secret by using the following steps. In this procedure, a self-signed certificate and key are created.
You can link it to the Kubernetes secret and use that secret in the Ingress for securing the Ingress.

```
1     openssl genrsa -out cert_key.pem 2048
2     openssl req -new -key cert_key.pem -out cert_csr.pem -subj "/CN=
         example.com"
3     openssl x509 -req -in cert_csr.pem -sha256 -days 365 -extensions
         v3_ca -signkey cert_key.pem -CAcreateserial -out cert_cert.pem
```

> **Note:**
>
> Here, `example.com` is used for reference. You must replace `example.com` with the required domain name.

In the example, the generated certificate has a validity of one year as the days are mentioned as 365 days.

Linking the certificate to a Kubernetes secret

Perform the following steps to link the certificate to the Kubernetes secret.

1. Run the following command to create a Kubernetes secret based on the TLS certificate that you have created.


```
kubectl create secret tls tls-secret --cert=cert_cert.pem --key=
    cert_key.pem
```

2. Run the following command to view the secret that contains the TLS certificate information:

```
kubectl get secret tls-secret
```

Deploy the Ingress

Create and apply the Ingress configuration. The following YAML can be used for reference.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-demo
  namespace: netscaler
  annotations:
   kubernetes.io/ingress.class: "netscaler"
spec:
  tls:
  - secretName: tls-secret
    hosts:
      - "example.com"
  rules:
  - host:  "example.com"
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: service-test
            port:
              number: 80
```

View metrics of NetScalers using Prometheus and Grafana

December 31, 2023

You can use the NetScaler Metrics Exporter and Prometheus-Operator to monitor NetScaler VPX or CPX ingress devices and NetScaler CPX (east-west) devices.

NetScaler Metrics Exporter

NetScaler Metrics Exporter is a simple server that collects NetScaler stats and exports them to Prometheus using HTTP. You can then add Prometheus as a data source to Grafana and graphically view the NetScaler stats. For more information see, NetScaler Metrics Exporter.

> **Note:**
>
> NetScaler Metrics Exporter supports exporting metrics from the admin partitions of NetScaler.

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:

```
1  $ kubectl get pods -n monitoring
2  NAME                                  READY     STATUS    RESTARTS
       AGE
3  alertmanager-main-0                   2/2       Running   0          2
       h
4  alertmanager-main-1                   2/2       Running   0          2
       h
5  alertmanager-main-2                   2/2       Running   0          2
       h
6  grafana-5b68464b84-5fvxq              1/1       Running   0          2
       h
7  kube-state-metrics-6588b6b755-d6ftg   4/4       Running   0          2
       h
8  node-exporter-4hbcp                   2/2       Running   0          2
       h
9  node-exporter-kn9dg                   2/2       Running   0          2
       h
10 node-exporter-tpxhp                   2/2       Running   0          2
       h
11 prometheus-k8s-0                      3/3       Running   1          2
       h
12 prometheus-k8s-1                      3/3       Running   1          2
       h
```



```
prometheus-operator-7d9fd546c4-m8t7v   1/1       Running   0          2
    h
```

> **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
kind: Service
metadata:
  labels:
    prometheus: k8s
  name: prometheus-k8s
  namespace: monitoring
spec:
  type: NodePort
  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:

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

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 NetScaler Metrics Exporter

This topic describes how to integrate the NetScaler Metrics Exporter with NetScaler VPX or CPX ingress or NetScaler CPX (east-west) devices.

Configure NetScaler Metrics Exporter for NetScaler VPX Ingress device

To monitor an ingress NetScaler VPX device, the NetScaler Metrics Exporter is run as a pod within the Kubernetes cluster. The IP address of the NetScaler VPX ingress device is provided as an argument to the NetScaler 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:
```



```
13          - "--target-nsip=<IP_of_VPX>"
14          - "--port=8888"
15        volumeMounts:
16        - name: nslogin
17          mountPath: "/mnt/nslogin"
18          readOnly: true
19        securityContext:
20          readOnlyRootFilesystem: true
21    volumes:
22    - name: nslogin
23      secret:
24        secretName: nslogin
25  ---
26  kind: Service
27  apiVersion: v1
28  metadata:
29    name: exporter-vpx-ingress
30    labels:
31      service-type: citrix-adc-monitor
32  spec:
33    selector:
34      app: exporter-vpx-ingress
35    ports:
36      - name: exporter-port
37        port: 8888
38        targetPort: 8888
39  <!--NeedCopy-->
```

The IP address and the port of the NetScaler VPX device needs to be provided in the `--target-nsip` parameter. For example, `--target-nsip=10.0.0.20`.

Configure NetScaler Metrics Exporter for NetScaler CPX Ingress device

To monitor a NetScaler CPX ingress device, the NetScaler Metrics Exporter is added as a sidecar to the NetScaler CPX.The following is a sample YAML file of a NetScaler CPX ingress device with the exporter as a side car:

```
1  ---
2  apiVersion: apps/v1
3  kind: Deployment
4  metadata:
5    labels:
6      app: cpx-ingress
7    name: cpx-ingress
8  spec:
9    replicas: 1
10   selector:
11     matchLabels:
12       app: cpx-ingress
13   template:
14     metadata:
```



```
15        annotations:
16          NETSCALER_AS_APP: "True"
17        labels:
18          app: cpx-ingress
19      spec:
20        containers:
21        - env:
22          - name: EULA
23            value: "YES"
24          - name: NS_PROTOCOL
25            value: HTTP
26          - name: NS_PORT
27            value: "9080"
28          #Define the NITRO port here
29          image: quay.io/citrix/citrix-k8s-cpx-ingress:13.0-52.24
30          imagePullPolicy: IfNotPresent
31          name: cpx-ingress
32          ports:
33          - containerPort: 80
34            name: http
35            protocol: TCP
36          - containerPort: 443
37            name: https
38            protocol: TCP
39          - containerPort: 9080
40            name: nitro-http
41            protocol: TCP
42          - containerPort: 9443
43            name: nitro-https
44            protocol: TCP
45          securityContext:
46            privileged: true
47        # Adding exporter as a sidecar
48        - args:
49          - --target-nsip=192.0.0.2
50          - --port=8888
51          - --secure=no
52          env:
53          - name: NS_USER
54            value: nsroot
55          - name: NS_PASSWORD
56            value: nsroot
57          image: quay.io/citrix/citrix-adc-metrics-exporter:1.4.8
58          imagePullPolicy: IfNotPresent
59          name: exporter
60          securityContext:
61            readOnlyRootFilesystem: true
62        serviceAccountName: cpx
63  ---
64  kind: Service
65  apiVersion: v1
66  metadata:
67    name: exporter-cpx-ingress
```



```
  labels:
    service-type: citrix-adc-monitor
spec:
  selector:
    app: cpx-ingress
  ports:
    - name: exporter-port
      port: 8888
      targetPort: 8888
<!--NeedCopy-->
```

Here, the exporter uses the local IP address (`192.0.0.2`) to fetch metrics from the NetScaler CPX.

Configure NetScaler Metrics Exporter for NetScaler CPX (east-west) device

To monitor a NetScaler CPX (east-west) device, the NetScaler Metrics Exporter is added as a sidecar to the NetScalerCPX.The following is a sample YAML file of a NetScaler CPX (east-west) device with the exporter as a side car:

```
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:
      - env:
        - 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
        env:
        - name: NS_USER
          value: nsroot
        - name: NS_PASSWORD
          value: nsroot
        image: quay.io/citrix/citrix-adc-metrics-exporter:1.4.8
        imagePullPolicy: IfNotPresent
        name: exporter
        securityContext:
          readOnlyRootFilesystem: true
      serviceAccountName: cpx
---
kind: Service
apiVersion: v1
metadata:
  name: exporter-cpx-ew
  labels:
    service-type: citrix-adc-monitor
spec:
  selector:
    app: cpx-ew
  ports:
    - name: exporter-port
      port: 8888
      targetPort: 8888
<!--NeedCopy-->
```

Here, the exporter uses the local IP (`192.168.0.2`) to fetch metrics from the NetScaler CPX (east-west) device.

ServiceMonitors to detect NetScaler

The NetScaler Metrics Exporter helps collect data from the NetScaler VPX or CPX ingress and NetScaler 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.

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
```



```
  name: citrix-adc-servicemonitor
  labels:
    servicemonitor: citrix-adc
spec:
  endpoints:
  - interval: 30s
    port: exporter-port
  selector:
    matchLabels:
      service-type: citrix-adc-monitor
  namespaceSelector:
    matchNames:
    - monitoring
    - default
<!--NeedCopy-->
```

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:

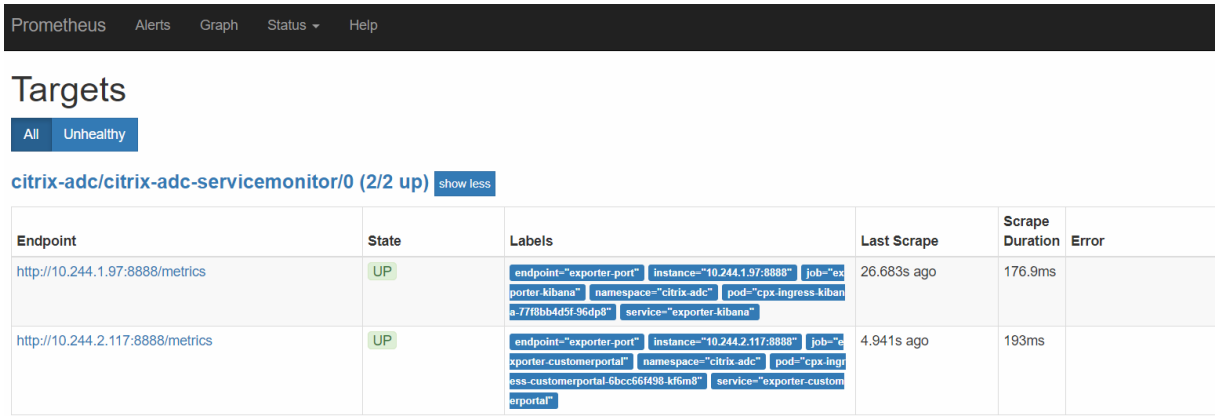
```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus-k8s
rules:
- apiGroups:
  - ""
  resources:
  - nodes/metrics
  - namespaces
  - services
  - endpoints
  - pods
  verbs: ["*"]
- nonResourceURLs:
  - /metrics
  verbs:  ["*"]
<!--NeedCopy-->
```

To provide additional privileges Prometheus, deploy the sample YAML using the following command:

```
kubectl apply -f prometheus-clusterRole.yaml
```

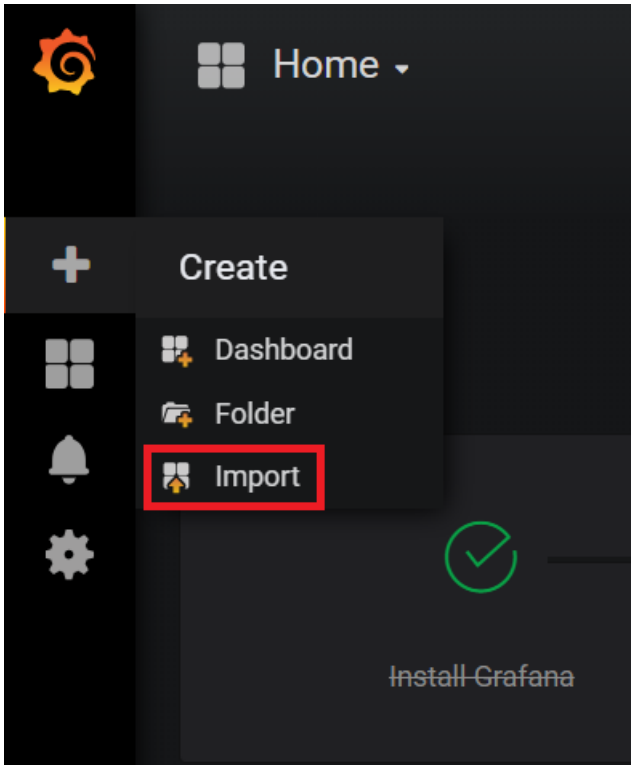

View the metrics in grafana

The NetScaler instances that are detected for monitoring appears in the **Targets** page of the prometheus container. You can be access the **Targets** page using the following URL: `http://<k8s_cluster_ip>:<prometheus_nodeport>/targets`:

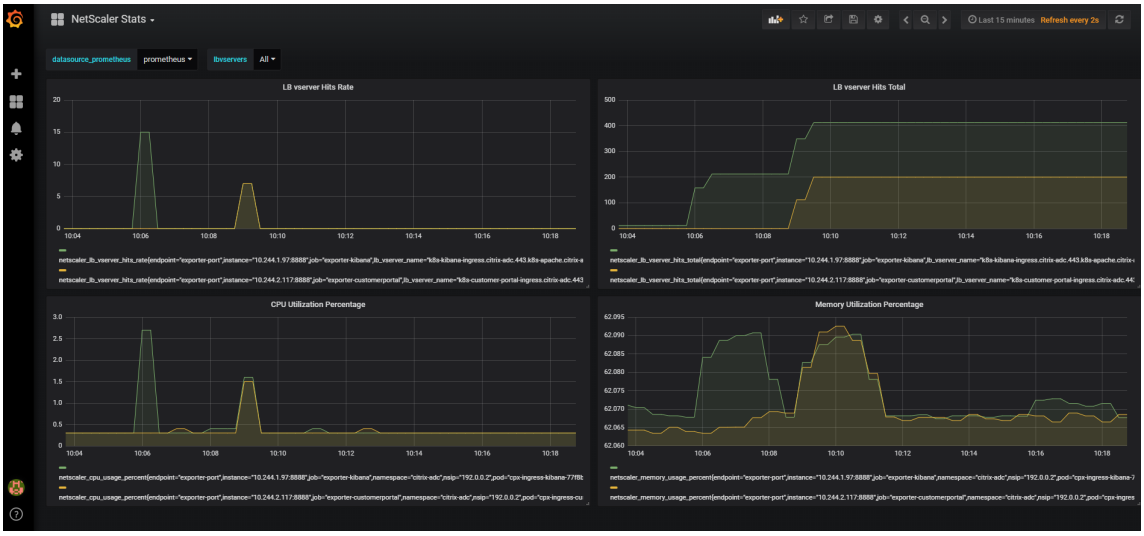


To view the metrics graphically:

1. Log into grafana using `http://<k8s_cluster_ip>:<grafafa_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

December 31, 2023

Analytics from NetScaler 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 NetScaler 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:

```
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.

```
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:

```
service.citrix.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.

```
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
  type: LoadBalancer
  selector:
    name: apache
  ports:
  - name: http
    port: 80
    targetPort: http
  selector:
    app: apache
<!--NeedCopy-->
```

For information about annotations, see the annotation documentation.

Analytics using NetScaler ADM

NetScaler 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.

NetScaler ADM analytics provide an easy and scalable way to get various insights out of the data from NetScaler instances to describe, predict, and improve the application performance. You can use one or more analytics features simultaneously on NetScaler 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 NetScaler ADM and Kubernetes cluster or managed instances in your data center or cloud. It makes NetScaler instances discoverable by NetScaler ADM.
- Ensure that an appropriate license is available and auto licensing is enabled on ADM.

Analytics with open source tools

NetScaler can be integrated with various open source tools for observability using NetScaler Observability Exporter. NetScaler Observability Exporter is a container which collects metrics and transac-

tions from NetScalers 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 NetScalers.
For more information on NetScaler Observability Exporter, see the NetScaler Observability Exporter documentation.

Analytics configuration support using ConfigMap

December 31, 2023

You can use NetScaler Observability Exporter to export metrics and transactions from NetScaler CPX, MPX, or VPX and analyze the exported data to get meaningful insights. The NetScaler Observability Exporter support is enabled with in the NetScaler Ingress Controller configuration. You can now enable the NetScaler Observability Exporter configuration with in the NetScaler 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 NetScaler 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 NetScaler Observability Exporter service depending on whether the service is running on a virtual machine or as a Kubernetes service.
    If the NetScaler Observability Exporter instance is running on a virtual machine this parameter specifies the IP address. If the NetScaler 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 NetScaler. 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 NetScaler.
    * `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 NetScaler.
    * `enable`: Set this value to **`true`** to enable audit log data. The default value is **`false`**.
  - `events`: Enables exporting events from the NetScaler.
    * `enable`: Set this value to **`true`** to enable exporting events. The default value is **`false`**.

- `transactions`: Enables exporting transactions from NetScaler.
  - `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 NetScaler Ingress Controller is running and you need to reboot the NetScaler 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 NetScaler Ingress Controller is running.

> **Note:**
>
> When the user specifies value for a service as `namespace/service name`, NetScaler Ingress Controller derives the endpoint associated to that service and dynamically bind them to the transactional service group in NetScaler tier-1 ADC . If a user specifies the value for a service as IP address, the IP address is direclty bound to the transactional service group. NetScaler 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 NetScaler 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.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cic-configmap
  labels:
    app: citrix-ingress-controller
data:
  LOGLEVEL: 'info'
  NS_PROTOCOL: 'http'
  NS_PORT: '80'
  NS_HTTP2_SERVER_SIDE: 'ON'
  NS\_ANALYTICS\_CONFIG: |
    distributed_tracing:
      enable: 'false'
      samplingrate: 100
    endpoint:
      server: '1.1.1.1'
      service: 'default/coe-kafka'
    timeseries:
      port: 5563
      metrics:
        enable: 'false'
        mode: 'avro'
      auditlogs:
        enable: 'false'
      events:
        enable: 'false'
    transactions:
      enable: 'true'
      port: 5557
```

For more information on how to configure ConfigMap support on the NetScaler Ingress Controller, see configuring ConfigMap support for the NetScaler Ingress Controller.

Schema for NS_ANALYTICS_CONFIG

Following is the schema for `NS_ANALYTICS_CONFIG`. The attributes should confirm with this schema.

```
type: map
mapping:
  NS_ANALYTICS_CONFIG:
    required: no
    type: map
    mapping:
```



```
7         endpoint:
8           required: yes
9           type: map
10          mapping:
11            server:
12              required: yes
13              type: str
14        distributed_tracing:
15          required: no
16          type: map
17          mapping:
18            enable:
19              required: yes
20              type: str
21              enum:
22                - 'true'
23                - 'false'
24            samplingrate:
25              required: no
26              type: int
27              range:
28                max: 100
29                min: 0
30        timeseries:
31          required: no
32          type: map
33          mapping:
34            port:
35              required: no
36              type: int
37            metrics:
38              required: no
39              type: map
40              mapping:
41                enable:
42                  required: yes
43                  type: str
44                  enum:
45                    - 'true'
46                    - 'false'
47                mode:
48                  required: yes
49                  type: str
50                  enum:
51                    - prometheus
52                    - avro
53                    - influx
54            auditlogs:
55              required: no
56              type: map
57              mapping:
58                enable:
59                  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

December 31, 2023

The following table describes some of the common issues and workarounds.

Problem	Log	Workaround
NetScaler instance is not reachable	**2019-01-10 05:05:27,250 - ERROR - [nitrointerface.py:login_logout:94] (MainThread) Exception: HTTPConnectionPool(host=' 10.106.76.200', port=80): Max retries exceeded with url: /nitro/v1/config/login (Caused by NewConnectionError('<urllib3.connection.HTTPConnection object at 0x7f4d45bd63d0>: Failed to establish a new connection: [Errno 113] No route to host',))**	Ensure that the NetScaler is up and running, and you can ping the NSIP address.
Wrong user name password	**2019-01-10 05:03:05,958 - ERROR - [nitrointerface.py:login_logout:90] (MainThread) Nitro Exception::login_logout::errorcode=354,message=Invalid username or password**	
SNIP is not enabled with management access	**2019-01-10 05:43:03,418 - ERROR - [nitrointerface.py:login_logout:94] (MainThread) Exception: HTTPConnectionPool(host=' 10.106.76.242', port=80): Max retries exceeded with url: /nitro/v1/config/login (Caused by NewConnectionError('<urllib3.connection.HTTPConnection object at 0x7f302a8cfad0>: Failed to establish a new connection: [Errno 110] Connection timed out',))**	Ensure that you have enabled the management access in NetScaler (for NetScaler VPX high availability) and set the IP address, **NSIP**, with management access enabled.

Problem	Log	Workaround
Error while parsing annotations	**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}**	
Wrong port for NITRO access	**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('<urllib3.connection.HTTPConnection object at 0x7fc592cb8b10>: Failed to establish a new connection: [Errno 111] Connection refused',))**	Verify if the correct port is specified for NITRO access. By default, NetScaler Ingress Controller uses the port **80** for communication.
Ingress class is wrong	**2019-01-10 05:27:27,149 - INFO - [kubernetes.py:get_all_ingresses:1329] (MainThread) Unsupported Ingress class for ingress object web-ingress.default**	Verify that the ingress file belongs to the ingress class that NetScaler Ingress Controller monitors. See the following log for information about the ingress classes listened by NetScaler Ingress Controller

Problem	Log	Workaround
Kubernetes API is not reachable	**2019-01-10 05:32:09,729 - ERROR - [kubernetes.py:_get:222] (Thread-1) Error while calling /services:HTTPSConnectionPool(host='10.106.76.237', port=6443): Max retries exceeded with url: /api/v1/services (Caused by NewConnectionError('<urllib3.connection.VerifiedHTTPSConnection object at 0x7fb3013e7dd0>: Failed to establish a new connection: [Errno 111] Connection refused',))**	Check if the kubernetes_url is correct. Use the command, `kubectl cluster-info` to get the URL information. Ensure that the Kubernetes main node is running at `https://kubernetes_master_address:6443` and the Kubernetes API server pod is up and running.
Incorrect service port specified in the YAML file		Provide the correct port details in the ingress YAML file and reapply to solve the issue.
Load balancing virtual server and service group are created but they are down		Check for the service name and port used in the YAML file. For NetScaler VPX, ensure that `--feature-node-watch` is set to **`true`**, when bringing up the NetScaler Ingress Controller.
CS virtual server is not getting created for NetScaler VPX.		Use the annotation, `ingress.citrix.com/frontend-ip`, in the ingress YAML file for NetScaler VPX.
Incorrect secret provided in the TLS section in the ingress YAML file	**2019-01-10 09:30:50,673 - INFO - [kubernetes.py:_get:231] (MainThread) Resource not found: /secrets/default-secret12345 namespace default**	Correct the values in the YAML file and reapply to solve the issue.

Problem	Log	Workaround
	2019-01-10 09:30:50,673 - INFO - [kubernetes.py:get_secret:1712] (MainThread) Failed to get secret for the app default-secret12345.default	
The `feature-node-watch` argument is specified, but static routes are not added in the NetScaler VPX	**ERROR - [nitrointerface.py:add_ns_route:4495] (MainThread) Nitro Exception::add_ns_route::errorcode=604, message=The gateway is not directly reachable**	This error occurs when `feature-node-watch` is enabled and the NetScaler VPX and Kubernetes cluster are not in the same network. You must remove the `--feature-node-watch` argument from the NetScaler Ingress Controller YAML file. Static routes do not work when the NetScaler VPX and Kubernetes cluster are in different network. Use node controller to create tunnels between NetScaler VPX and cluster nodes.
CRD status not updated	**ERROR - [crdinfrautils.py:update_crd_status:42] (MainThread) Exception during CRD status update for negrwaddmuloccmod: 403 Client Error: Forbidden for url: `https://10.96.0.1:443/apis/citrix.com/v1/namespaces/default/rewritepolicies/negrwaddmuloccmod/status`**	Verify that permission to push CRD status is provided in the RBAC. The permission should be similar to the following

Problem	Log	Workaround
		• apiGroups: [“citrix.com”] resources: [“rewritepolicies/status”, “canarycrds/status”, “authpolicies/status”, “ratelimits/status”, “listeners/status”, “httproutes/status”, “wafs/status”]
NetScaler Ingress Controller event not updated	**ERROR - [clienthelper.py:post:94] (MainThread) Reuqest /events to api server is forbidden**	Verify that the permission to update the NetScaler Ingress Controller pod events is provided in the RBAC.
		• apiGroups: [””] resources: [“events”] verbs: [“create”]
Rewrite-responder policy not added	**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] <**	Such errors are due to incorrect expressions in rewrite-responder CRDs. Fix the expression and reapply the CRD.
	ERROR - [config_dispatcher.py:__dispatch_config_pack:324] (Dispatcher) Status: 104, ErrorCode: 3098, Reason: Nitro Exception: Invalid expression data type [ent.ip.src^, Offset 13]	

Problem	Log	Workaround
Application of a CRD failed. The NetScaler Ingress Controller converts a CRD into a set of configurations to configure the NetScaler to the desired state as per the specified CRD. If the configuration fails, then the CRD instance may not get applied on the NetScaler.	**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'** **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 cant be updated.]** **2020-07-13 08:49:07,620 - INFO - [config_dispatcher.py:__dispatch_config_pack:263] (Dispatcher) Processing of ConfigPack 'default.k8service.kuard-service.add_spec'failed**	Log shows that the NITRO command has failed. The same log appears in NetScaler as well. Check the NetScaler `ns.log` and search for the error using the `grep` command. Log shows the NetScaler 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.

Troubleshooting - Prometheus and Grafana Integration

Problem	Description	Workaround
Grafana dashboard has no plots	If the graphs on the Grafana dashboards do not have any values plotted, then Grafana is unable to obtain statistics from its datasource.	Check if the Prometheus datasource is saved and working properly. On saving the datasource after providing the Name and IP, a `Data source is working` message appears in green indicating the datasource is reachable and detected. If the dashboard is created using `sample_grafana_dashboard.json`, ensure that the name given to the Prometheus datasource begins with the word `prometheus` in the lowercase. Check the Targets page of Prometheus to see if the required target exporter is in `DOWN` state.
DOWN: Context deadline exceeded	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 `scrape_timeout`.	If you are using the Prometheus Operator, `scrape_timeout` is adjusted automatically and the error means that the exporter itself is not reachable. If a standalone Prometheus container or pod is used, try increasing the `scrape_interval` and `scrape_timeout` values in the `/etc/prometheus/prometheus.cfg` file to increase the time interval for collecting the metrics.

Troubleshooting - OpenShift feature node watch

Problem: While using OpenShift-ovn CNI `feature-node-watch` is not adding correct routes.

Description: NetScaler 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 NetScaler Ingress Controller along with `route.openshift.io` to run in the OpenShift environment with OVN CNI.

```
- apiGroups: ["config.openshift.io"]
  resources: ["networks"]
  verbs: ["get", "list"]
```

2. NetScaler Ingress Controller looks for the following two annotations added by OVN, make sure that it exists on the
cluster nodes.

```
"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 NetScaler VPX.

Problem: While using OpenShift-sdn CNI, feature-node-watch is not adding correct routes.

Description: NetScaler 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 NetScaler Ingress Controller along with `route.openshift.io` to run in the OpenShift environment with SDN CNI.

```
- apiGroups: ["network.openshift.io"]
  resources: ["hostsubnets"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["config.openshift.io"]
  resources: ["networks"]
  verbs: ["get", "list"]
```

2. NetScaler Ingress Controller looks for the following CRD and specification.

```
    oc get hostsubnets.network.openshift.io <cluster node-name> -
       ojson
```



```

    {
  "apiVersion": "network.openshift.io/v1",
        "host": <cluster node-name,
        "hostIP": "x.x.x.x",
        "kind": "HostSubnet",
        "metadata": {

    "annotations": {

                    ...
     }
,
        "subnet": "10.129.0.0/23"
     }
```

3. If the CRD does not exist with the expected specification, `feature-node-watch` might not work for OpenShfit-SDN CNI. In that case, you must manually configure the static routes on NetScaler VPX.

Troubleshooting the NetScaler Ingress Controller during runtime

December 31, 2023

You can debug the NetScaler 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 NetScaler 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 NetScaler Ingress Controller.

```
Kubectl describe pods <citrix-k8s-ingress-controller pod name> -n <
    namespace of pod>
```

You can view the events under the events section.

In this example, the NetScaler has been deliberately made unreachable and the same information can be seen under the events section.

```
kubectl describe pods cic-vpx-functionaltest -n functionaltest

Name:         cic-vpx-functionaltest
Namespace:    functionaltest

Events:
Type     Reason     Age   From                    Message
----     ------     ----  ----                    -------
Normal   Pulled     33m   kubelet, rak-asp4-node2 Container image "citrix-ingress-controller:latest" already present on machine
Normal   Created    33m   kubelet, rak-asp4-node2 Created container cic-vpx-functionaltest
Normal   Started    33m   kubelet, rak-asp4-node2 Started container cic-vpx-functionaltest
Normal   Scheduled  33m   default-scheduler       Successfully assigned functionaltest/cic-vpx-functionaltest to rak-asp4-node2

Normal   Created    33m   CIC ENGINE, cic-vpx-functionaltest CONNECTED: NetScaler:<NetScaler IP>:80
Normal   Created    33m   CIC ENGINE, cic-vpx-functionaltest SUCCESS: Test LB Vserver Creation on NetScaler:
Normal   Created    33m   CIC ENGINE, cic-vpx-functionaltest SUCCESS: ENABLING INIT features on NetScaler:
Normal   Created    33m   CIC ENGINE, cic-vpx-functionaltest SUCCESS: GET Default VIP from NetScaler:
Warning  Created    17s   CIC ENGINE, cic-vpx-functionaltest UNREACHABLE: NetScaler: Check Connectivity::<NetScaler IP>:80
```

You can use the events section to check the flow of events within the NetScaler Ingress Controller. Events provide information on the flow of events. For further debugging, you should check the logs of the NetScaler Ingress Controller pod.

Log based debugging

You can change the log level of the NetScaler Ingress Controller at runtime using the ConfigMap feature. For changing the log level during runtime, see the ConfigMap documentation.

To check logs on the NetScaler Ingress Controller, use the following command.

```
kubectl logs <citrix-k8s-ingress-controller> -n namespace
```


Call Home enablement for the NetScaler Ingress Controller in NetScaler

December 31, 2023

Sometimes, NetScaler 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 NetScaler is enabled for the NetScaler Ingress Controller.
The Call Home feature is enabled by default and requires no specific configuration by users. When the latest version of the NetScaler Ingress Controller is deployed, a string map is configured on the NetScaler with the NetScaler Ingress Controller specific information.

Upgrade NetScaler Ingress Controller

December 31, 2023

This topic explains how to upgrade the NetScaler Ingress Controller instance for NetScaler CPX with the NetScaler Ingress Controller as sidecar and NetScaler Ingress Controller standalone deployments.

Upgrade NetScaler CPX with NetScaler Ingress Controller as a sidecar

To upgrade a NetScaler CPX with the NetScaler 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 NetScaler Ingress Controller and NetScaler CPX image under `containers` section to the following:
   - NetScaler CPX version: 13.0-83.27 (`quay.io/citrix/citrix-k8s-cpx-ingress:13.0-83.27`)
   - NetScaler Ingress Controller version: 1.29.5 (`quay.io/citrix/citrix-k8s-ingress-controller:1.29.5`)
2. Update the `CluterRole` as follows:

```
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"]
  # 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: ["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"]
    verb s: ["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 NetScaler Ingress Controller to version 1.21.9

To upgrade a standalone NetScaler Ingress Controller instance, you can either modify the **YAML** definition file or use the Helm chart.

If you want to upgrade NetScaler Ingress Controller to version 1.21.9 by modifying the **YAML** definition file, perform the following:

1. Change the version for the NetScaler Ingress Controller image under `containers` section. For example, consider you have the following YAML file.

```
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:

```
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
  - 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: ["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.

IP address management using the IPAM controller

April 4, 2024

The IPAM controller is a container provided by NetScaler for IP address management and it runs in parallel to NetScaler Ingress Controller as 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. NetScaler Ingress Controller configures the IP address allocated to the service as a virtual IP address (VIP) in NetScaler MPX or VPX.
Using this IP address, you can externally access the service.

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, 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, and 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.

NetScaler IPAM 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. NetScaler provides a way to load balance such services using the NetScaler Ingress Controller and NetScaler.

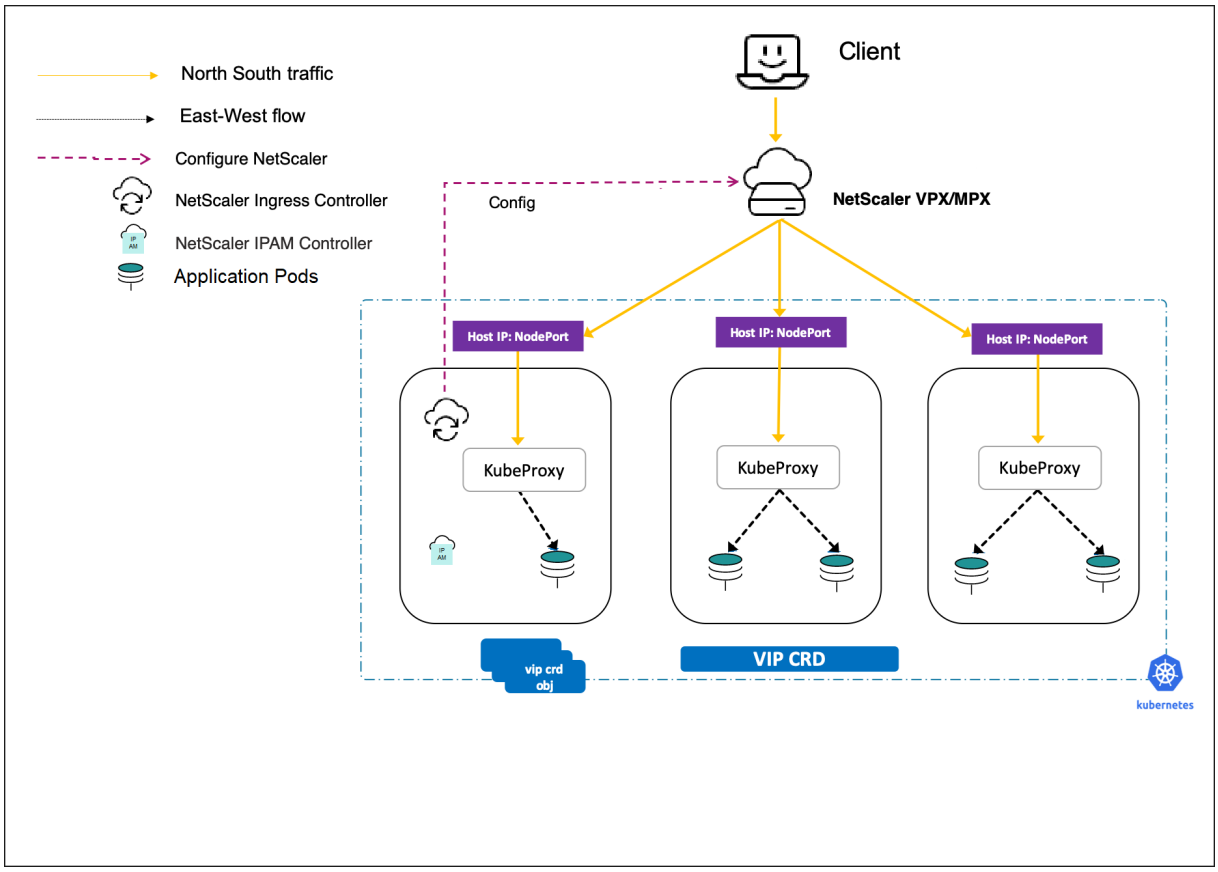
In the NetScaler solution for services of type `LoadBalancer`, NetScaler Ingress Controller deployed inside the Kubernetes cluster configures a NetScaler deployed outside the cluster to load balance the incoming traffic. Using the NetScaler solution, you can load balance the incoming traffic to the Kubernetes cluster regardless of whether the deployment is on bare metal, on-premises, or public cloud.

Since the NetScaler Ingress Controller provides flexible IP address management that enables multi-tenancy for NetScalers, you can use a single NetScaler 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.

IP address management using the IPAM controller

NetScaler IPAM controller requires the VIP CRD provided by NetScaler. The VIP CRD contains fields for service-name, namespace, and IP address. The VIP CRD is used for internal communication between the NetScaler 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 NetScaler 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 NetScaler Ingress Controller automatically configures the NetScaler.

> **Note:**
>
> Custom resource definitions (CRDs) offered by NetScaler
> also support 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.

Expose 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 NetScaler Ingress Controller.
2. Deploy NetScaler IPAM controller.
3. Deploy a sample application.
4. Expose the sample application using service of type LoadBalancer.
5. Access the service.

Deploy NetScaler Ingress Controller

Perform the following steps to deploy the NetScaler Ingress Controller with the IPAM controller argument.

1. Add the NetScaler Helm chart repository to your local registry using the following command.

```
helm repo add netscaler https://netscaler.github.io/netscaler-helm-charts/
<!--NeedCopy-->
```

2. Install NetScaler Ingress Controller using the following command.

```
helm install netscaler-ingress-controller netscaler/netscaler-ingress-controller --set nsIP=<NS_IP>,license.accept=yes,adcCredentialSecret=<>,ingressClass[0]=netscaler,serviceClass[0]=netscaler,ipam=true,crds.install=true -n netscaler
<!--NeedCopy-->
```

   For detailed information about deploying and configuring NetScaler Ingress Controller using Helm charts, see the Helm chart repository.

Deploy IPAM controller

1. Add the NetScaler Helm chart repository to your local registry using the following command.

```
helm repo add netscaler https://netscaler.github.io/netscaler-helm-charts/
<!--NeedCopy-->
```

2. Install NetScaler IPAM controller using the following command.

```
 helm install netscaler-ipam-controller netscaler/netscaler-ipam-controller --set vipRange='[{
"Prod": ["10.1.2.0 - 10.1.2.255"] }
]' -n netscaler
 <!--NeedCopy-->
```

Deploy a sample application

Perform the following steps 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 choice.

1. Deploy a sample application using the following command:

```
kubectl apply -f - <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache
  namespace: netscaler
  labels:
    name: apache
spec:
  selector:
    matchLabels:
      app: apache
  replicas: 2
  template:
    metadata:
      labels:
        app: apache
    spec:
      containers:
      - name: apache
        image: httpd:latest
        ports:
        - name: http
          containerPort: 80
```



```
          imagePullPolicy: IfNotPresent
    EOF
    <!--NeedCopy-->
```

2. Verify if the pods are running using the following command:

```
kubectl get pods
<!--NeedCopy-->
```

Expose the sample application using service of type LoadBalancer

Perform the following steps to create a service of type `LoadBalancer`:

1. Deploy a service to expose `apache` application, for which the IP address is allocated from the `Prod` VIP range specified during the IPAM installation.

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Service
metadata:
  name: apache
  namespace: netscaler
  labels:
    name: apache
  annotations:
    service.citrix.com/class: 'netscaler'
    service.citrix.com/ipam-range: 'Prod'
spec:
  externalTrafficPolicy: Local
  type: LoadBalancer
  ports:
  - name: http
    port: 80
    targetPort: http
  selector:
    app: apache

EOF
<!--NeedCopy-->
```

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. NetScaler Ingress Controller configures the IP address allocated to the service as a virtual IP address (VIP) in NetScaler.

2. View the service using the following command:

```
kubectl get service apache --output yaml
```



```
<!--NeedCopy-->
```

Output:

```
apiVersion: v1
kind: Service
metadata:
 annotations:
   NETSCALER_VPORT: “80”
 creationTimestamp: “2019-10-04T18:40:42Z”
 labels:
   name: apache
 name: apache
 namespace: default
 resourceVersion: “22245641”
 selfLink: /api/v1/namespaces/default/services/apache
 uid: 7811c632-e6d6-11e9-8359-527c8bde541f
spec:
 clusterIP: 10.108.104.133
 externalTrafficPolicy: Local
 healthCheckNodePort: 32646
 ports:
 - name: http
   nodePort: 31408
   port: 80
   protocol: TCP
   targetPort: http
 selector:
   app: apache
 sessionAffinity: None
 type: LoadBalancer
status:
 loadBalancer:
   ingress:
   - ip: 10.105.158.196
```

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 following curl command to access the service:

```
curl <IP_address>
<!--NeedCopy-->
```


The response should be:

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

Securing Ingress

December 31, 2023

The topic covers the various ways to secure your Ingress using NetScaler and the annotations provided by the NetScaler Ingress Controller.

The following table lists the TLS use cases with sample annotations that you can use to secure your Ingress using the Ingress NetScaler and the NetScaler Ingress Controller:

Use cases	Sample annotations
Enable TLS v1.3 protocol	`ingress.citrix.com/frontend-sslprofile: '{ "tls13 tls13sessionticketsperauthcontext":"1", "dhekeyexchangewithpsk":"yes" } '`
HTTP strict transport security (HSTS)	`ingress.citrix.com/frontend-sslprofile: '{ "hsts" : "157680000", "includesubdomain":"yes" }`
OCSP stapling	`ingress.citrix.com/frontend-sslprofile: '{ "ocsps '`
Set client authentication to mandatory	`ingress.citrix.com/frontend-sslprofile: '{ "clien clientauth" : "mandatory" } '`
TLS session ticket extension	`ingress.citrix.com/frontend-sslprofile: '{ "sessi ticket" : "enabled", "sessionticketlifetime : "300" } '`
SSL session reuse	`ingress.citrix.com/frontend-sslprofile: '{ "sessr sessionreusetimeout : "120" } '`
Cipher groups	`ingress.citrix.com/frontend-sslprofile:'{ "sniena ciphers" : [{ "ciphername": "secure", "cipherpriority" :"1" } , { "ciphername": "secure", "ci ] } '`
Cipher redirect	`ingress.citrix.com/frontend-sslprofile:'{ "sniena ciphers" : [{ "ciphername": "secure", "cipherpriority" :"1" } ], "cipherredirect":"enabled", "c redirecturl" } '`

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 NetScaler.

The `tls13SessionTicketsPerAuthContext` parameter enables you to set the number of tickets the Ingress NetScaler issues anytime TLS 1.3 is negotiated, ticket-based resumption is enabled,

and either a handshake completes or post-handhsake 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 NetScaler 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.

```
ingress.citrix.com/frontend-sslprofile: '{
 "tls13":"enabled", "tls13sessionticketsperauthcontext":"1", "
    dhekeyexchangewithpsk":"yes" }
 '
```

HTTP strict transport security (HSTS)

The Ingress NetScaler 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 NetScaler support for HSTS.

Using the annotations for SSL profiles, you can enable HSTS in an SSL front-end profile on the Ingress NetScaler. The following is a sample ingress annotation:

```
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 NetScaler 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 NetScaler 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:

```
ingress.citrix.com/frontend-sslprofile: '{
 "ocspstapling":"enabled" }
 '
```

> **Note:**
>
> To use OCSP stapling, you must add an OCSP responder on the NetScaler appliance.

Set Client authentication to mandatory

Using the annotations for SSL profiles, you can enable client authentication, the Ingress NetScaler 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 NetScaler 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:

```
ingress.citrix.com/frontend-sslprofile: '{
 "clientauth":"enabled", "clientcert" : "mandatory" }
 '
```

> **Note:**
>
> Make sure that you bind the client-certificate to the SSL virtual server on the Ingress NetScaler.

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 NetScaler 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 NetScaler, using the `sessionticketlifetime` parameter.

The following is the sample ingress annotation:

```
ingress.citrix.com/frontend-sslprofile: '{
 "sessionticket" : "enabled", "sessionticketlifetime : "300" }
 '
```

SSL session reuse

You can reuse an existing SSL session on a NetScaler 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 NetScaler.

The following is the sample ingress annotation:

```
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 NetScaler 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 NetScaler.

The built-in cipher groups can be used in Tier-1 and Tier-2 NetScaler, and the user-defined cipher group can be used only in Tier-1 NetScaler.

To use a user-defined cipher group, ensure that the NetScaler 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:

```
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 NetScaler 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`.

```
ingress.citrix.com/frontend-sslprofile:'{
"snienable": "enabled", "ciphers" : [{
"ciphername": "secure", "cipherpriority" :"1" }
], "cipherredirect":"enabled", "cipherurl": "https://redirecturl" }
'
```

TCP use cases

April 25, 2024

This topic covers various ways to configure TCP parameters on NetScaler using smart annotations for services of type `LoadBalancer` and ingress using the annotations in NetScaler Ingress Controller.

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 front-end TCP profiles can be attached to the client-side content switching virtual server and the back-end TCP profiles can be configured for a service group.

TCP profile support for services of type LoadBalancer

NetScaler 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 NetScaler.

Service annotation	Description
`service.citrix.com/frontend-tcpprofile`	Use this annotation to create the front-end TCP profile (client plane).
`service.citrix.com/backend-tcpprofile`	Use this annotation to create the back-end TCP profile (server plane).

User-defined TCP profiles

Using service annotations for TCP, you can create custom profiles with the same name as that of content switching virtual server or service group and bind them to the corresponding virtual server (`frontend-tcpprofile`) and service group (`backend-tcpprofile`).

Service annotation	Sample
`service.citrix.com/frontend-tcpprofile` `service.citrix.com/backend-tcpprofile`	`service.citrix.com/frontend-tcpprofile: 'enable-tcp'` `service.citrix.com/backend-tcpprofile: 'enable-tcp'`

Built-in TCP profiles

Built-in TCP profiles do not create any profiles but bind a given profile name in the annotation to the corresponding virtual server (frontend-tcpprofile) and service group (backend-tcpprofile).

Examples for built-in TCP profiles:

```
service.citrix.com/frontend-tcpprofile: "tcp_preconf_profile"
service.citrix.com/backend-tcpprofile: '{ "citrix-svc" : "tcp_preconf_profile
" }
```

Example: Service of type LoadBalancer with 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 YAML file.

> **Note:**
>
> For information about exposing services of type `LoadBalancer`, see service of type `LoadBalancer`.

Deploy a sample application (tea-beverage.yaml) using the following command:

```
kubectl apply -f - <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tea-beverage
  labels:
    name: tea-beverage
spec:
  selector:
    matchLabels:
      name: tea-beverage
  replicas: 2
  template:
    metadata:
      labels:
        name: tea-beverage
    spec:
      containers:
      - name: tea-beverage
        image: quay.io/citrix-duke/hotdrinks:latest
        ports:
        - name: tea-80
          containerPort: 80
        - name: tea-443
          containerPort: 443
        imagePullPolicy: Always
---
apiVersion: v1
kind: Service
metadata:
  name: tea-beverage
  annotations:
    service.citrix.com/frontend-ip: 10.105.158.194
    service.citrix.com/frontend-tcpprofile: '{
 "ws" : "enabled", "sack" : "enabled" }
 '
    service.citrix.com/backend-tcpprofile: '{
 "ws" : "enabled", "sack" : "enabled" }
 '
spec:
  type: LoadBalancer
  ports:
  - name: tea-80
    port: 80
    targetPort: 80
  selector:
    name: tea-beverage
EOF
<!--NeedCopy-->
```

After the application is deployed, the corresponding entities and profiles are created on NetScaler.

Run the following commands on NetScaler to verify the same: `show cs vserver k8s-tea-beverage_80_default_svc` and `show servicegroup k8s-tea-beverage_80_sgp_f4lezsannv`
.

```
        # show cs vserver k8s-tea-beverage_80_default_svc
          k8s-tea-beverage_80_default_svc (10.105.158.194:80) - TCP
              Type: CONTENT
          State: UP
          Last state change was at Wed Apr  3 09:37:59 2024
          Time since last state change: 0 days, 00:00:09.790
          Client Idle Timeout: 9000 sec
          Down state flush: ENABLED
          Disable Primary Vserver On Down : DISABLED
          Comment: uid=
              VIGQWRCYKCM6WFYX2GFKRVT3ZF6JSFISPW6XM24JADBXEYRLITOQ====
          **TCP profile name: k8s-tea-beverage_80_default_svc**
          Appflow logging: ENABLED
          State Update: DISABLED
          Default: k8s-tea-
              beverage_80_lbv_f4lezsannvu7tk2ftpjbhi4hza2tvdnk Content
              Precedence: RULE
          L2Conn: OFF Case Sensitivity: ON
          Authentication: OFF
          401 Based Authentication: OFF
          HTTP Redirect Port: 0 Dtls : OFF
          Persistence: NONE
          Listen Policy: NONE
          IcmpResponse: PASSIVE
          RHIstate:  PASSIVE
          Traffic Domain: 0

          1) Default Target LB: k8s-tea-
              beverage_80_lbv_f4lezsannvu7tk2ftpjbhi4hza2tvdnk Hits: 0
          Done
<!--NeedCopy-->
```

```
        # show servicegroup k8s-tea-
            beverage_80_sgp_f4lezsannvu7tk2ftpjbhi4hza2tvdnk
          k8s-tea-beverage_80_sgp_f4lezsannvu7tk2ftpjbhi4hza2tvdnk -
              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
          Monitoring Owner: 0
          TCP Buffering(TCPB): NO
          HTTP Compression(CMP): NO
          Idle timeout: Client: 9000 sec Server: 9000 sec
          Client IP: DISABLED
          Cacheable: NO
          SC: ???
          SP: OFF
          Down state flush: ENABLED
```



```
            Monitor Connection Close : NONE
            Appflow logging: ENABLED
            TCP profile name: k8s-tea-beverage_80_sgp_f4lezsannvu7tk2ftpjbhi4hza2tvdnk
            ContentInspection profile name: ???
            Process Local: DISABLED
            Traffic Domain: 0
            Comment: "lbsvc:tea-beverage,svcport:80,ns:default"


            1)   10.146.107.38:30524 State: UP Server Name: 10.146.107.38 Server ID: None Weight: 1 Order: Default
              Last state change was at Wed Apr  3 09:38:00 2024
              Time since last state change: 0 days, 00:02:27.660

              Monitor Name: tcp-default State: UP Passive: 0
              Probes: 30 Failed [Total: 0 Current: 0]
              Last response: Success - TCP syn+ack received.
              Response Time: 0.000 millisec
          Done
    <!--NeedCopy-->
```

> **Note:**
>
> The TCP profile is supported for single-port services.

Configure TCP profiles using Ingress annotations

The following table lists some of the TCP use cases with sample annotations:

Use case	Sample annotation
Silently drop idle TCP connections	`ingress.citrix.com/frontend-tcpprofile: '{ "dropestconnontimeout" : "enabled" }'`
Delayed TCP connection acknowledgments	`ingress.citrix.com/frontend-tcpprofile: '{ "delayedack" : "150" }'`
Client side Multipath TCP session management	`ingress.citrix.com/backend-tcpprofile: '{ "mptcp": "enabled", "mptcpsessiontimeout": "7200" }'`
TCP Optimization	`ingress.citrix.com/frontend-tcpprofile: '{ "dropestconnontimeout" : "enabled", "dropestconnontimeout" : "enabled" }'`
Selective acknowledgment	`ingress.citrix.com/backend-tcpprofile: '{ "citrix-svc" : { "sack" : "enabled" } }'`

Forward acknowledgment	`ingress.citrix.com/frontend_tcpprofile: '{ "fack" : "enabled" }'`
	`ingress.citrix.com/backend_tcpprofile: '{ "citrix-svc" : { "sack" : "enabled" } }'`
	`ingress.citrix.com/backend_tcpprofile: '{ "citrix-svc" : { "fack" : "enabled" } }'`

Use case	Sample annotation
Window Scaling	`ingress.citrix.com/frontend_tcpprofile: '{ "ws" : "enabled", "wsval" : "9" } '`
Maximum Segment Size	`ingress.citrix.com/frontend_tcpprofile: '{ "mss" : "1460", "maxpktpermss" : "512" } '`
Keep-Alive	`ingress.citrix.com/frontend_tcpprofile: '{ "ka" : "enabled", "kaprobeupdatelastactivity" : "enabled", "kaconnidletime": "900", "kamaxprobes" : "3", "kaprobeinterval" : "75" } '`
bufferSize	`ingress.citrix.com/backend_tcpprofile: '{ "citrix-svcn": { "ws": "enabled", "wsval": "9" } } '`
MPTCP	`ingress.citrix.com/frontend_tcpprofile: '{ "mptcp" : "enabled", "bufferSize" : "8190" } '`
flavor	`ingress.citrix.com/backend_tcpprofile: '{ "citrix-mptcpdropdataonpreestsf": "enabled", "mptcpfastopen": "enabled" } '`
Dynamic receive buffering	`ingress.citrix.com/frontend_tcpprofile: '{ "flavor" : "westwood" } '`
Defending TCP against spoofing attacks	`ingress.citrix.com/backend_tcpprofile: '{ "citrix-dynamicreceivebuffering": "enabled", "mapbetweenbackendsocketandtcpcb": "enabled" } '`
	`ingress.citrix.com/frontend_tcpprofile: '{ "rstwindowattenuate": "enabled", "spoofsyndrop" : "enabled" } '`

> **Note:**
>
> The above Ingress annotations can also be used with service annotations in the format described earlier.

Silently drop idle TCP connections

In a network, when a large number of TCP connections become idle, NetScaler 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 NetScaler 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 NetScaler. 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 NetScaler just drops the connection.

Using the annotations for TCP profiles, you can enable or disable the `drophalfclosedconnontimeout` and `dropestconnontimeout` on NetScaler. The following is a sample annotation of TCP profile to enable these parameters:

```
ingress.citrix.com/frontend-tcpprofile: '{ "drophalfclosedconnontimeout
" : "enable", "dropestconnontimeout" : "enable" } '
```

```
ingress.citrix.com/backend-tcpprofile: '{ "citrix-svc" : { "drophalfclosedconn
" : "enable", "dropestconnontimeout" : "enable" } } '
```

Delayed TCP connection acknowledgments

To avoid sending several ACK packets, NetScaler supports TCP delayed acknowledgment mechanism. It sends delayed ACK with a default timeout of 100 ms. NetScaler 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:

```
ingress.citrix.com/frontend-tcpprofile: '{ "delayedack" : "150" } '
```

```
ingress.citrix.com/backend-tcpprofile: '{ "citrix-svc" : { "delayedack
" : "150" } } '
```

Client side Multipath TCP session management

You can perform TCP configuration on NetScaler for Multipath TCP (MPTCP) connections between the client and NetScaler. MPTCP connections are not supported between NetScaler and the back-end communication. Both the client and NetScaler appliance must support the same MPTCP version.

You can enable MPTCP and set the MPTCP session timeout (`mptcpsessiontimeout`) in seconds using TCP profiles in NetScaler. 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 NetScaler. The following is a sample annotation of TCP profile to enable MPTCP and set the `mptcpsessiontimeout` parameter value to 7200 on NetScaler:


```
ingress.citrix.com/frontend-tcpprofile: '{ "mptcp" : "enabled", "
mptcpsessiontimeout" : "7200" } '
```

```
ingress.citrix.com/backend-tcpprofile: '{ "citrix-svc" : { "mptcp" :
"enabled", "mptcpsessiontimeout" : "7200" } } '
```

TCP optimization

Most of the relevant TCP optimization capabilities of NetScaler are exposed through a corresponding TCP profile. Using the annotations for TCP profiles, you can enable the following TCP optimization capabilities on NetScaler:

- `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 NetScaler:

  ```
  ingress.citrix.com/frontend_tcpprofile: '{ "sack" : "enabled" } '
  ```

  ```
  ingress.citrix.com/backend_tcpprofile: '{ "citrix-svc" : { "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 NetScaler:

  ```
  ingress.citrix.com/frontend_tcpprofile: '{ "fack" : "enabled" } '
  ```

  ```
  ingress.citrix.com/backend_tcpprofile: '{ "citrix-svc" : { "fack"
   : "enabled" } } '
  ```

- `Window Scaling (WS)`: TCP window scaling allows increasing the TCP receive window size beyond 65535 bytes. It helps improve 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 NetScaler:

  ```
  ingress.citrix.com/frontend_tcpprofile: '{ "ws" : "enabled", "
  wsval" : "9" } '
  ```

  ```
  ingress.citrix.com/backend_tcpprofile: '{ "citrix-svc" : { "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.

- `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 NetScaler:

  ```
  ingress.citrix.com/frontend_tcpprofile: '{ "mss" : "1460", "
  maxpktpermss" : "512" } '
  ```

  ```
  ingress.citrix.com/backend_tcpprofile: '{ "citrix-svc" : { "mss"
  : "1460", "maxpktpermss" : "512" } } '
  ```

  Where:

  - `mss` is the MSS to use for the TCP connection. Minimum value: 0; Maximum value: 9176.
  - `maxpktpermss` is the maximum number of TCP packets allowed per maximum segment size (MSS). Minimum value: 0; Maximum value: 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 NetScaler:

  ```
  ingress.citrix.com/frontend_tcpprofile: '{ "ka" : "enabled", "
  kaprobeupdatelastactivity" : "enabled", "kaconnidletime": "900",
  "kamaxprobes" : "3", "kaprobeinterval" : "75" } '
  ```

  ```
  ingress.citrix.com/backend_tcpprofile: '{ "citrix-svc" : { "
  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:

  ```
  ingress.citrix.com/frontend_tcpprofile: '{ "bufferSize" : "8190" } '
  ```

  ```
  ingress.citrix.com/backend_tcpprofile: '{ "citrix-svc" : { "bufferSize" : "8190" } } '
  ```

- `Multipath TCP (MPTCP)`: Enable MPTCP and set the optional MPTCP configuration. The following is a sample annotation of TCP profile to enable MPTCP and se the optional MPTCP configurations:

  ```
  ingress.citrix.com/frontend_tcpprofile: '{ "mptcp" : "enabled", "mptcpdropdataonpreestsf" : "enabled", "mptcpfastopen": "enabled", "mptcpsessiontimeout" : "7200" } '
  ```

  ```
  ingress.citrix.com/backend_tcpprofile: '{ "citrix-svc" : { "mptcp" : "enabled", "mptcpdropdataonpreestsf" : "enabled", "mptcpfastopen": "enabled", "mptcpsessiontimeout" : "7200" } } '
  ```

  Where:

  - `mptcpdropdataonpreestsf` is used to silently dropping the data on Pre-Established subflow. When enabled, DSS data packets are dropped silently instead of dropping the connection when data is received on pre-established subflow. Possible values: ENABLED, DISABLED. Default value: DISABLED.
  - `mptcpfastopen` can be enabled so that DSS data packets are accepted before receiving the third ack of SYN handshake. Possible values: ENABLED, DISABLED. Default value: DISABLED

- `flavor`: Set the TCP congestion control algorithm. Possible values: Default, BIC, CUBIC, Westwood, and Nile. Default value: Default. The following sample annotation of TCP profile sets the TCP congestion control algorithm:

  ```
  ingress.citrix.com/frontend_tcpprofile: '{ "flavor" : "westwood" } '
  ```

  ```
  ingress.citrix.com/backend_tcpprofile: '{ "citrix-svc" : { "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" } '
```

```
ingress.citrix.com/backend_tcpprofile: '{ "citrix-svc" : { "
dynamicReceiveBuffering" : "enabled" } } '
```

Defend TCP against spoofing attacks

You can enable NetScaler to defend TCP against spoof attacks using the `rstWindowAttenuate` parameter in TCP profiles. By default, the `rstWindowAttenuate` parameter is disabled. This parameter is enabled to protect NetScaler against spoofing. If you enable `rstWindowAttenuate`, it replies with corrective acknowledgment (ACK) for an invalid sequence number. Possible values: Enabled, Disabled. Additionally, `spoofSynDrop` parameter can be used to enable or disable drop of invalid SYN packets to protect against spoofing. When disabled, established connections will be reset when a SYN packet is received. The default value for this parameter is ENABLED.

The following is a sample annotation of TCP profile to enable `rstWindowAttenuate` on NetScaler:

```
ingress.citrix.com/frontend_tcpprofile: '{ "rstwindowattenuate" : "
enabled", "spoofsyndrop" : "enabled" } '
```

```
ingress.citrix.com/backend_tcpprofile: '{ "citrix-svc" : { "rstwindowattenuate
" : "enabled", "spoofsyndrop" : "enabled" } } '
```

Example for applying TCP profile using Ingress annotation

This example shows how to apply TCP profiles.

1. Deploy the front-end ingress resource with the TCP profile. In this Ingress resource, backend and TLS are not defined.

```
kubectl apply -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: frontend-ingress
  annotations:
    ingress.citrix.com/insecure-termination: "allow"
    ingress.citrix.com/frontend-ip: "10.221.36.190"
    ingress.citrix.com/frontend-tcpprofile: '{
 "ws" : "enabled", "sack" : "enabled" }
```



```
'
spec:
  tls:
  - hosts:
  rules:
  - host:
EOF
<!--NeedCopy-->
```

2. Deploy the secondary ingress resource with the same front-end IP address. Back end and TLS are defined, which creates the load balancing resource definition.

```
kubectl apply -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: backend-ingress
  annotations:
    ingress.citrix.com/insecure-termination: "allow"
    ingress.citrix.com/frontend-ip: "10.221.36.190"
spec:
  tls:
  - secretName: <hotdrink-secret>
  rules:
  - host:  hotdrink.beverages.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend-hotdrinks
            port:
              number: 80
EOF
<!--NeedCopy-->
```

3. After the ingress resources are deployed, the corresponding entities, profiles are created on NetScaler.

   Run the following command on NetScaler: `show cs vserver k8s-10.221.36.190_443_ssl`.

```
        # show cs vserver k8s-10.221.36.190_443_ssl

          k8s-10.221.36.190_443_ssl (10.221.36.190:443) - SSL Type:
              CONTENT
          State: UP
          Last state change was at Wed Apr  3 04:21:38 2024
          Time since last state change: 0 days, 00:00:57.420
          Client Idle Timeout: 180 sec
          Down state flush: ENABLED
          Disable Primary Vserver On Down : DISABLED
```



```
10          Comment: uid=
               XMX2KPYG2GUJIHGTLVCPA7QVXDUBDRMJFTAWNCPAA2TVXB33EL5A====
11          TCP profile name: k8s-10.221.36.190_443_ssl
12          Appflow logging: ENABLED
13          State Update: DISABLED
14          Default: Content Precedence: RULE
15          Vserver IP and Port insertion: OFF
16          L2Conn: OFF Case Sensitivity: ON
17          Authentication: OFF
18          401 Based Authentication: OFF
19          Push: DISABLED Push VServer:
20          Push Label Rule: none
21          HTTP Redirect Port: 0 Dtls : OFF
22          Persistence: NONE
23          Listen Policy: NONE
24          IcmpResponse: PASSIVE
25          RHIstate:  PASSIVE
26          Traffic Domain: 0
27
28          1) Content-Switching Policy: k8s-
               backend_80_csp_2k75kfjrr6ptgzwtncozwxdjqrpbvicz Rule: HTTP
               .REQ.HOSTNAME.SERVER.EQ("hotdrink.beverages.com") && HTTP.
               REQ.URL.PATH.SET_TEXT_MODE(IGNORECASE).STARTSWITH("/")
               Priority: 200000008 Hits: 0
29          Done
30    <!--NeedCopy-->
```

> **Note:**
>
> For an exhaustive list of the various TCP parameters supported on NetScaler, refer to Supported TCP Parameters. The key and value that you pass in the JSON format must match the NetScaler NITRO format. For more information on the NetScaler NITRO API, see NetScaler 14.1 REST APIs - NITRO Documentation for TCP profiles.

HTTP use cases

December 31, 2023

This topic covers various HTTP use cases that you can configure on the Ingress NetScaler using the annotations in the NetScaler Ingress Controller.

The following table lists the HTTP use cases with sample annotations:

Use case	Sample annotation

Configuring HTTP/2	`ingress.citrix.com/frontend-httpprofile: '{ "http2":"enabled" } '`
	`ingress.citrix.com/backend-httpprofile: '{ "apache":{ " http2direct" : "enabled" } '`
	`ingress.citrix.com/backend-httpprofile: '{ "apache":{ " http2direct" : "enabled", "altsvc ":"enabled" } '`

Handling HTTP session timeouts	`ingress.citrix.com/frontend-httpprofile: '{ "reqtimeout" : "10", "reqtimeoutaction":"drop" } '`
	`ingress.citrix.com/frontend-httpprofile: '{ "reqtimeout" : "10", "adptimeout" : "enable" } '`
	`ingress.citrix.com/backend-httpprofile: '{ "apache":{ " reusepooltimeout" : "20000" } } '`

Configuring HTTP/2

The Ingress NetScaler HTTP/2 on the client side as well on the server side. For more information, see HTTP/2 support on NetScaler. For an HTTP load balancing configuration on the Ingress NetScaler, it uses one of the following methods to start communicating with the client/server using HTTP/2.

The Ingress NetScaler 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 NetScaler Ingress Controller provides annotations to configure the HTTP profile on the Ingress NetScaler. You use these annotations to configure the various HTTP load balancing configuration on the Ingress NetScaler 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 NetScaler.

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 NetScaler. The following is a sample annotation of the HTTP profile to configure the HTTP/2 upgrade method on the Ingress NetScaler:

```
ingress.citrix.com/frontend-httpprofile: '{
 "http2":"enabled" }
 '
```

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 NetScaler. The following is a sample annotation of the HTTP profile to configure the direct HTTP/2 method on the Ingress NetScaler:

```
ingress.citrix.com/backend-httpprofile: '{
 "apache":{
 "http2direct" : "enabled" }
  }
 '
```

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 for the HTTP profile to configure the direct HTTP/2 using alternative service (ALT-SVC) method on the Ingress NetScaler:


```
ingress.citrix.com/backend-httpprofile: '{
 "apache":{
 "http2direct" : "enabled", "altsvc":"enabled" }
  }
 '
```

Handling HTTP session timeouts

To handle the different types 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 NetScaler, you can configure the following timeouts for these scenarios:

- reqTimeout and reqTimeoutAction
- adptTimeout
- reusePoolTimeout

reqTimeout and reqTimeoutAction

In NetScaler, 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 the defined time, the specified request timeout action in the `reqTimeoutAction` is executed. The minimum timeout value that 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 the HTTP profile to configure the HTTP request timeout and HTTP request timeout action on the Ingress NetScaler:

```
ingress.citrix.com/frontend-httpprofile: '{
 "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:

```
ingress.citrix.com/frontend-httpprofile: '{
 "reqtimeout" : "10", "adptimeout" : "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 that 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:

```
ingress.citrix.com/backend-httpprofile: '{
 "apache":{
 "reusepooltimeout" : "20000" }
  }
 '
```

HTTP callout with the rewrite and responder policy

December 31, 2023

An HTTP callout allows NetScaler 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 NetScaler documentation.

You can initiate the HTTP callout through the following expressions with the rewrite and responder CRD provided by NetScaler:

- `sys.http_callout()`: This expression is used for blocking the call when the httpcallout agent response needs to be evaluated.
- `sys.non_blocking_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.

Parameter	Description
`name`	Specifies the name of the callout, maximum is up to 32 characters.
`server_ip`	Specifies the IP Address of the server (callout agent) to which the callout is sent.
`server_port`	Specifies the Port of the server (callout agent) to which the callout is sent.
`http_method`	Specifies the method used in the HTTP request that this callout sends. The default value is GET.
`host_expr`	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("Host")) 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.
`url_stem_expr`	Specifies a string expression for generating the URL stem. The string expression can contain a literal string (for example, "/mysite/index.html") or an expression that derives the value (for example, http.req.url).

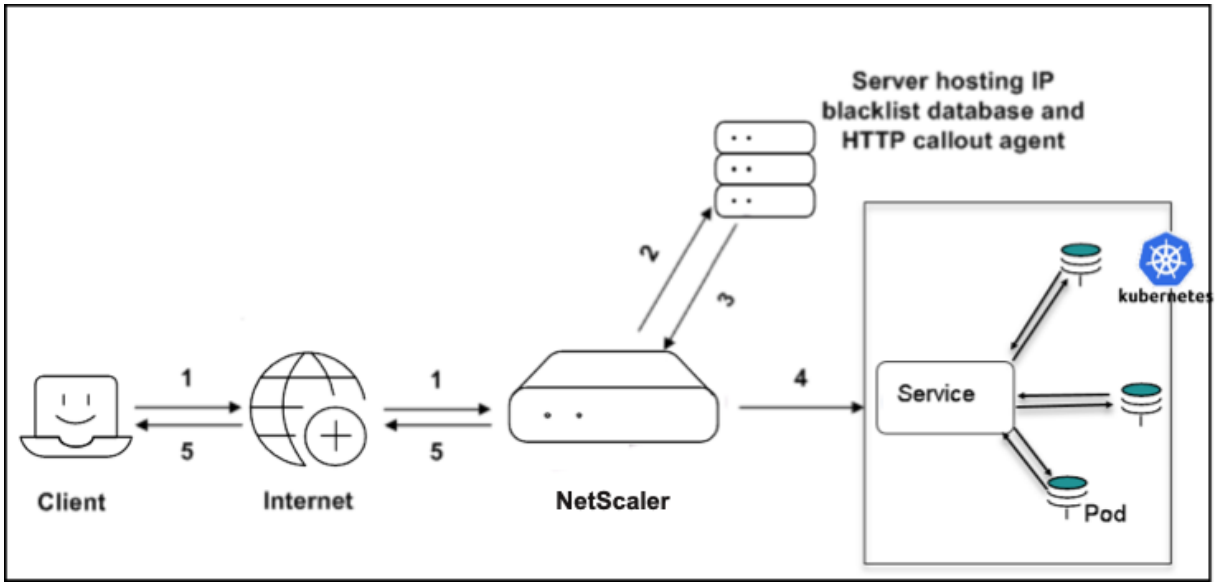
Parameter	Description
headers	Specifies one or more headers to insert into the HTTP request. Each header name and exp, where exp is an expression that is evaluated at runtime to provide the value for the named header.
parameters	Specifies one or more query parameters to insert into the HTTP request URL (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.
body_expr	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 (for example, client.ip.src).
full_req_expr	Specifies the exact HTTP request, in the form of an expression, which the NetScaler 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.
scheme	Specifies the type of scheme for the callout server. Example: HTTP, HTTPS
return_type	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.

Parameter	Description
`cache_for_secs`	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.
`result_expr`	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 `HTTP.RES`. The operations in this expression must match the return type. For example, if you configure a return type of `TEXT`, the result expression must be a text based expression. If the return type is `NUM`, the result expression (result_expr) must return a numeric value, as in the following example: `http.res.body(10000).length`
`comment`	Specifies any comments to preserve the information about this HTTP callout.

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:

> **Note:**
>
> You must deploy the rewrite and responder CRD before deploying the `ip_validate_responder` YAML file.

```yaml
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"'
```

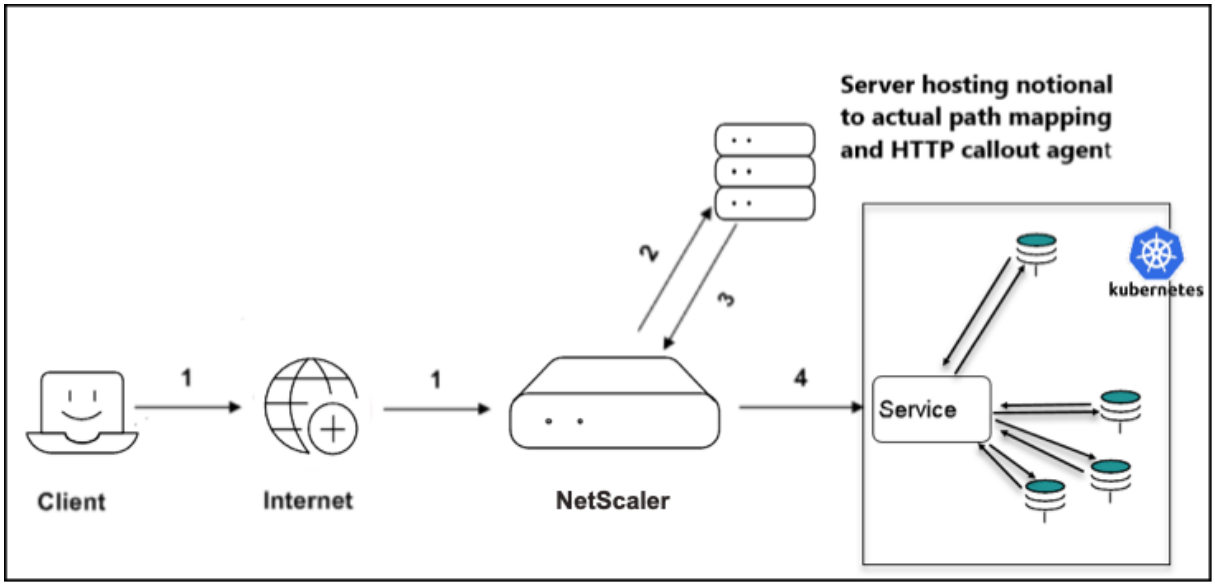


```
12          respond-criteria: 'sys.http_callout("blocklist_callout").
               CONTAINS("IP Matched")' #Callout name needs to be given in
               double quotes to pick httpcallout_policy
13          comment: 'Invalid access'
14
15    httpcallout_policy:
16      - name: blocklist_callout
17        server_ip: "192.2.156.160"
18        server_port: 80
19        http_method: GET
20        host_expr: '"192.2.156.160"'
21        url_stem_expr: '"/validateIP.pl"'
22        headers:
23        - name: X-Request
24          expr: '"Callout Request"'
25        parameters:
26        - name: Cip
27          expr: 'CLIENT.IP.SRC'
28        return_type: TEXT
29        result_expr: 'HTTP.RES.BODY(100)'
30  <!--NeedCopy-->
```

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.

> **Note:**
>
> You must deploy the rewrite and responder CRD before deploying the `path_rewrite` YAML file.

```yaml
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
  name: getvalidpath
spec:
  rewrite-policies:
    - servicenames:
        - frontend
      rewrite-policy:
        operation: replace
        target: http.req.url
        modify-expression: 'sys.http_callout("mapping_callout")' # Callout name needs to be given in double quotes to pick httpcallout_policy
        comment: 'Get the valid path'
        direction: REQUEST
        rewrite-criteria: 'TRUE'

  httpcallout_policy:
    - name: mapping_callout
      server_ip: "192.2.156.160"
      server_port: 80
      http_method: GET
      host_expr: '"192.2.156.160"'
      url_stem_expr: '"/getPath.pl"'
      headers:
      - name: X-Request
        expr: '"Callout Request"'
      parameters:
      - name: path
        expr: 'http.req.url'
      return_type: TEXT
      result_expr: '"HTTP.RES.BODY(500).AFTER_STR(\"<newpath>\").BEFORE_STR(\"</newpath>\")"'
<!--NeedCopy-->
```


Configure session affinity or persistence on the Ingress NetScaler

December 31, 2023

Session affinity or persistence settings on the Ingress NetScaler 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 NetScaler 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 NetScaler 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 NetScaler 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 NetScaler 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 NetScaler, using the following Ingress annotation provided by the NetScaler Ingress Controller:

```
ingress.citrix.com/lbvserver: '{
 "apache":{
 "persistenceType":"COOKIEINSERT", "timeout":"20", "cookiename":"
    k8s_cookie" }
  }
 '
```


Where:

- `timeout` specifies the duration of persistence. If session cookies are used with a `timeout` value of 0, no expiry time is specified by NetScaler 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, `COOKIEINSERT` 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 NetScaler, you can set persistence to an load balancing virtual server, that creating a stickiness for the subsequest requests from the same client.

The following is a sample Ingress annotation to configure source IP address persistence:

```
ingress.citrix.com/lbvserver: '{
 "apache":{
 "persistenceType":"SOURCEIP", "timeout":"10" }
  }
 '
```

SSL session ID persistence

When SSL session ID persistence is configured, the NetScaler 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:

```
ingress.citrix.com/lbvserver: '{
 "apache":{
 "persistenceType":"SSLSESSION" }
  }
 '
```


Destination IP address-based persistence

In this type of persistence, when the Ingress NetScaler 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:

```
ingress.citrix.com/lbvserver: '{
 "apache":{
 "persistenceType":"DESTIP" }
  }
 '
```

Source and destination IP address-based persistence

In this type of persistence, when the NetScaler 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:

```
ingress.citrix.com/lbvserver: '{
 "apache":{
 "persistenceType":"SRCIPDESTIP" }
  }
 '
```

Allowlisting or blocklisting IP addresses

December 31, 2023

`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 NetScaler 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 NetScaler.

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:

```
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:

```
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")'
```



```
12          comment: 'Allowlist certain IP addresses'
13    patset:
14      - name: allowlistip
15        values: [ '10.xxx.170.xx', '10.xxx.16.xx' ]
16  <!--NeedCopy-->
```

Then, deploy the YAML file (`allowlist-ip.yaml`) using the following command:

```
1  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:

```
1  apiVersion: citrix.com/v1
2  kind: rewritepolicy
3  metadata:
4    name: allowlistip
5  spec:
6    responder-policies:
7      - servicenames:
8          - frontend
9        responder-policy:
10         respondwith:
11           http-payload-string: '"HTTP/1.1 403 Forbidden\r\n\r\n" + "
                 Client: " + CLIENT.IP.SRC + " is not authorized to access
                 URL:" + HTTP.REQ.URL.HTTP_URL_SAFE +"\n"'
12         respond-criteria: '!client.ip.src.TYPECAST_text_t.equals_any("
               allowlistip")'
13         comment: 'Allowlist a list of IP addresses'
14   patset:
15     - name: allowlistip
16       values: [ '10.xxx.170.xx',  '10.xxx.16.xx' ]
17 <!--NeedCopy-->
```

Then, deploy the YAML file (`allowlist-ip-403.yaml`) using the following command:

```
1  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\r\n" + "
              Client: " + 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: 'Allowlist certain IPs'
<!--NeedCopy-->
```

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:

```
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:

```
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:

```
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\r\n" + "
              Client: " + 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-->
```

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:

```
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
  name: allowlistsub
spec:
  responder-policies:
    - servicenames:
        - frontend
      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:

```
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

December 31, 2023

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 . The 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.

For the ExternalDNS integration to work, the `external-dns.alpha.kubernetes.io/hostname` annotation must contain the host name.

> **Note:**
>
> For ExtenalDNS 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:

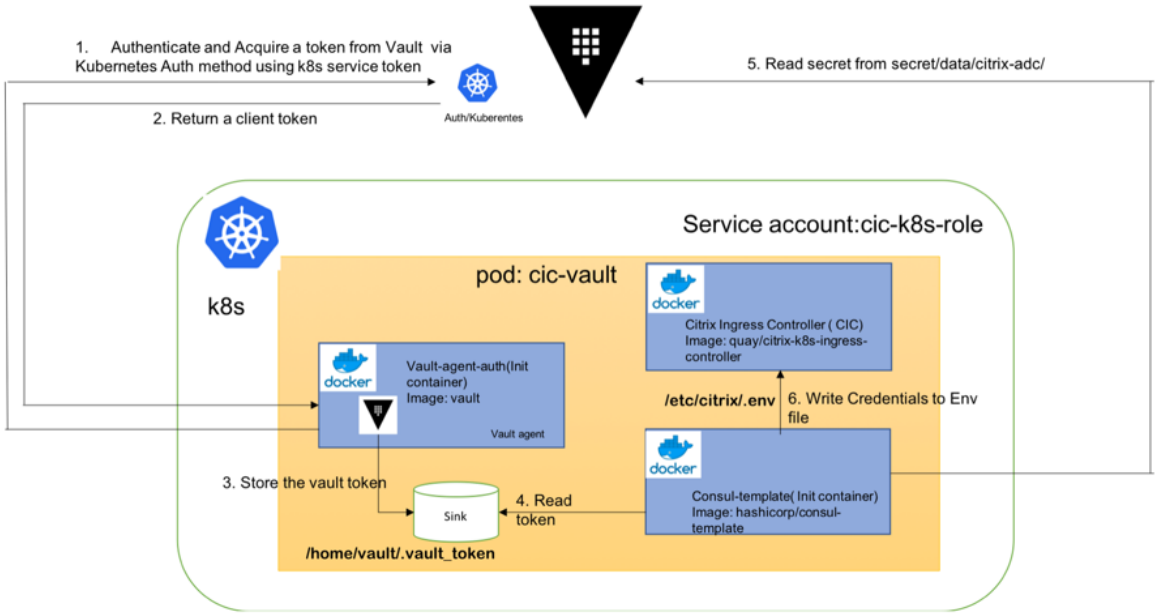
   ```
   kubectl create -f <service-name>.yml
   ```

Using NetScaler credentials stored in a Vault server for the NetScaler Ingress Controller

December 31, 2023

In most organizations, tier 1 NetScaler Ingress devices and Kubernetes clusters are managed by separate teams. Usually, network administrators manage tier 1 NetScaler Ingress devices, while developers manage Kubernetes clusters. The NetScaler Ingress Controller requires NetScaler credentials such as NetScaler user name and password to configure the NetScaler. You can specify NetScaler credentials as part of the NetScaler Ingress Controller specification and store the ADC credentials as Kubernetes secrets. However, you can also store NetScaler credentials in a Vault server and pass credentials to the NetScaler Ingress Controller to minimize any security risk. This topic provides information on how to use NetScaler credentials stored in a Vault server for the NetScaler Ingress Controller.

The following diagram explains the steps for using NetScaler credentials which are stored in a Vault server with the NetScaler 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 NetScaler credentials from a Vault server for the NetScaler Ingress Controller

Perform the following tasks to use NetScaler credentials from a Vault server for the NetScaler 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 NetScaler credentials for the NetScaler 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


serviceaccount/cic-k8s-role created
clusterrole.rbac.authorization.k8s.io/cic-k8s-role configured
clusterrolebinding.rbac.authorization.k8s.io/cic-k8s-role
    configured
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'] }
 ")
<!--NeedCopy-->
```

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-->
```

4. Get a Kubernetes CA signed certificate to communicate with Kubernetes API.

```
      export SA_CA_CRT=$(kubectl get secret $VAULT_SA_NAME -o
          jsonpath="{
 .data['ca\.crt'] }
 " | base64 --decode; echo)
<!--NeedCopy-->
```


Create a key vault secret and setup Kubernetes authentication on the Vault server

Log in to the Vault server and perform the following steps to create a Key Vault secret and setup Kubernetes authentication.

1. Review the sample vault policy file citrix-adc-kv-ro.hcl and create a read-only policy, `citrix-adc-kv-ro` in Vault.

```
$ tee citrix-adc-kv-ro.hcl <<EOF
# If working with K/V v1
path "secret/citrix-adc/*"
{

    capabilities = ["read", "list"]
 }

# If working with K/V v2
path "secret/data/citrix-adc/*"
{

    capabilities = ["read", "list"]
 }

EOF

#  Create a policy named citrix-adc-kv-ro
$  vault policy write citrix-adc-kv-ro citrix-adc-kv-ro.hcl
```

2. Create a KV secret with NetScaler credentials at the `secret/citrix-adc/` path.

```
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`).

```
# $ vault auth enable kubernetes
```

4. Specify how to communicate with the Kubernetes cluster.

```
$ 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.

```
$ 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 NetScaler Ingress Controller

Perform the following steps to leverage Vault auto-authentication.

1. Review the provided Vault Agent configuration file, `vault-agent-config.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.

```
vault {

    renew_token = false
    vault_agent_token_file = "/home/vault/.vault-token"
    retry {

        backoff = "1s"
     }

 }


template {

    destination = "/etc/citrix/.env"
    contents = <<EOH
    NS_USER=
    NS_PASSWORD=

    EOH
 }
```

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.

```
template {

    destination = "/etc/citrix/.env"
    contents = <<EOH
    NS_USER=
    NS_PASSWORD=

    EOH
         }
```

3. Create a Kubernetes config-map from vault-agent-config.hcl and consul-template-config.hcl.

```
kubectl create configmap example-vault-agent-config --from-
   file=./vault-agent-config.hcl --form-file=./consul-template
   -config.hcl
```

4. Create a NetScaler 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 NetScaler Ingress Controller for authentication with tier 1 NetScaler.

```
kubectl apply citrix-k8s-ingress-controller-vault.yaml
```

The `citrix-k8s-ingress-controller-vault.yaml` file is as follows:

```
apiVersion: v1
kind: Pod
metadata:
annotations:
name: cic-vault
namespace: default
spec:
containers:
- args:
    - --ingress-classes tier-1-vpx
    - --feature-node-watch true
    env:
    - name: NS_IP
    value: <Tier 1 ADC IP-ADDRESS>
    - name: EULA
    value: "yes"
    image: in-docker-reg.eng.citrite.net/cpx-dev/kumar-cic
        :latest
    imagePullPolicy: Always
    name: cic-k8s-ingress-controller
    volumeMounts:
    - mountPath: /etc/citrix
    name: shared-data
initContainers:
- args:
    - agent
    - -config=/etc/vault/vault-agent-config.hcl
    - -log-level=debug
    env:
    - name: VAULT_ADDR
    value: <VAULT URL>
    image: vault
    imagePullPolicy: Always
    name: vault-agent-auth
    volumeMounts:
    - mountPath: /etc/vault
    name: config
    - mountPath: /home/vault
    name: vault-token
- args:
    - -config=/etc/consul-template/consul-template-config.
        hcl
    - -log-level=debug
    - -once
    env:
    - name: HOME
    value: /home/vault
    - name: VAULT_ADDR
```



```
47              value: <VAULT_URL>
48              image: hashicorp/consul-template:alpine
49              imagePullPolicy: Always
50              name: consul-template
51              volumeMounts:
52              - mountPath: /home/vault
53              name: vault-token
54              - mountPath: /etc/consul-template
55              name: config
56              - mountPath: /etc/citrix
57              name: shared-data
58          serviceAccountName: vault-auth
59          volumes:
60          - emptyDir:
61              medium: Memory
62              name: vault-token
63          - configMap:
64              defaultMode: 420
65              items:
66              - key: vault-agent-config.hcl
67                  path: vault-agent-config.hcl
68              - key: consul-template-config.hcl
69          name: example-vault-agent-config
70              name: config
71          - emptyDir:
72              medium: Memory
73              name: shared-data
```

If the configuration is successful, the Vault server fetches a token and passes it on to a Consul template container. The Consul template uses the token to read NetScaler credentials and write it as an environment variable in the path `/etc/citrix/.env`. The NetScaler Ingress Controller uses these credentials for communicating with the tier 1 NetScaler.

Verify that the NetScaler Ingress Controller is running successfully using credentials fetched from the Vault server.

How to use Kubernetes secrets for storing NetScaler credentials

December 31, 2023

In most organizations, Tier 1 NetScaler Ingress devices and Kubernetes clusters are managed by separate teams. The NetScaler Ingress Controller requires NetScaler credentials such as NetScaler user name and password to configure the NetScaler. Usually, NetScaler credentials are specified as environment variables in the NetScaler Ingress Controller pod specification. But, another secure option is to use Kubernetes secrets to store the NetScaler 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 NetScaler 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 NetScaler 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:

```
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 NetScaler Ingress Controller pod specification.

- Use secret data as environment variables in the NetScaler Ingress Controller pod specification
- Use a secret volume mount to pass credentials to the NetScaler Ingress Controller

Use secret data as environment variables in the NetScaler Ingress Controller pod specification

You can use secret data from the Kubernetes secret as the values for the environment variables in the NetScaler Ingress Controller deployment specification.

A snippet of the YAML file is shown as follows.

```
  - 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 NetScaler Ingress Controller deployment with value of environment variables sourced from the secret object.

```
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:
         # Set NetScaler NSIP/SNIP, SNIP in case of HA (mgmt has to be
             enabled)
         - name: "NS_IP"
           value: "x.x.x.x"
         # Set username for Nitro
         - 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
         # Set log level
         - name: "EULA"
           value: "yes"
        imagePullPolicy: Always
<!--NeedCopy-->
```


Use a secret volume mount to pass credentials to the NetScaler Ingress Controller

Alternatively, you can also use a volume mount using the secret object as a source for the NetScaler credentials. The NetScaler 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.

```
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:
         # Set NetScaler NSIP/SNIP, SNIP in case of HA (mgmt has to be enabled)
         - name: "NS_IP"
           value: "x.x.x.x"
         # Set log level
         - name: "EULA"
           value: "yes"
        volumeMounts:
        # name must match the volume name below
          - name: secret-volume
            mountPath: /etc/citrix
        imagePullPolicy: Always
      # The secret data is exposed to Containers in the Pod through a Volume.
      volumes:
      - name: secret-volume
        secret:
          secretName: adc-credentials
<!--NeedCopy-->
```


Use NetScaler credentials stored in a Hashicorp Vault server

You can also use the NetScaler credentials stored in a Hashicorp Vault server for the NetScaler Ingress Controller and push the credentials through a sidecar container.

For more information, see Using NetScaler credentials stored in a Vault server.

How to load balance ingress traffic to TCP or UDP based application

April 8, 2024

In a Kubernetes environment, an ingress object 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, UDP, and SSL. Hence, any non-HTTP applications such as DNS, FTP or LDAP cannot be exposed using the standard Kubernetes ingress.

NetScaler 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, NetScaler Ingress Controller configures NetScaler 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`: This annotation enables L4 load balancing with TCP, UDP, or ANY as a protocol for NetScaler.
- `ingress.citrix.com/insecure-port`: This annotation configures the port for HTTP, TCP or UDP traffic. It 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 annotations.

You can also use the standard Kubernetes solution of creating a service of type `LoadBalancer` with NetScaler. You can find out more about Service Type LoadBalancer in NetScaler.

Sample: Ingress definition for TCP-based ingress.

```
kubectl apply -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    ingress.citrix.com/insecure-port: '6379'
    ingress.citrix.com/insecure-service-type: tcp
  name: redis-master-ingress
spec:
  ingressClassName: guestbook
```



```
  defaultBackend:
    service:
      name: redis-master-pods
      port:
        number: 6379
---
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  name: guestbook
spec:
  controller: citrix.com/ingress-controller
 EOF
<!--NeedCopy-->
```

Sample: Ingress definition for UDP-based ingress.

```
kubectl apply -f - <<EOF
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 defination
EOF
<!--NeedCopy-->
```

Sample: Service definition where the service port name is defined as `udp-53`:

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Service
metadata:
  name: bind
  labels:
    app: bind
spec:
  ports:
  - name: udp-53
    port: 53
    targetPort: 53
    protocol: UDP
  selector:
    name: bind
EOF
```



```
<!--NeedCopy-->
```

Load balance ingress traffic based on SSL over TCP

NetScaler Ingress Controller provides `ingress.citrix.com/secure-service-type: ssl_tcp` annotation that you can use to load balance ingress traffic based on SSL over TCP.

Sample: Ingress definition for SSL over TCP based Ingress.

```
kubectl apply -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    ingress.citrix.com/secure-service-type: "ssl_tcp"
    ingress.citrix.com/secure-backend: '{
 "frontendcolddrinks":"True" }
 '
  name: colddrinks-ingress
spec:
  ingressClassName: colddrink
  defaultBackend:
    service:
      name: frontend-colddrinks
      port:
        number: 443
  tls:
  - secretName: "colddrink-secret"
---
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  name: colddrink
spec:
  controller: citrix.com/ingress-controller
EOF
<!--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 NetScaler. The ECV (extended content validation) monitors help in checking whether the
application returns expected content or not. NetScaler Ingress Controller provides `ingress.citrix.com/monitor` annotation that can be used to monitor the health of the backend service.

Also, the application performance can be improved by using persistence methods such as `Source IP`. You can use these NetScaler features through Smart Annotations in Kubernetes.

The following ingress resource example uses smart annotations:

```
kubectl apply -f - <<EOF
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: '{
 "mongodb-svc":{
 "lbmethod":"SRCIPDESTIPHASH" }
  }
 '
    ingress.citrix.com/monitor: '{
 "mongodbsvc":{
 "type":"tcp-ecv" }
  }
 '
  name: mongodb
spec:
  rules:
  - host: mongodb.beverages.com
    http:
      paths:
      - backend:
          service:
            name: mongodb-svc
            port:
              number: 80
        path: /
        pathType: Prefix
EOF
<!--NeedCopy-->
```

For more information about the different deployment options supported by NetScaler Ingress Controller, see Deployment topologies.

For more information about deploying NetScaler Ingress Controller:

- Deploy the NetScaler Ingress Controller using Helm charts

How to expose non-standard HTTP ports in the NetScaler CPX service

Sometimes you need to expose ports other than 80 and 443 in a NetScaler CPX service for allowing TCP or UDP traffic on other ports.

This section provides information on how to expose other non-standard HTTP ports on the NetScaler CPX service when you deploy it in the Kubernetes cluster.

For Helm chart deployments

To expose non-standard HTTP ports while deploying NetScaler CPX with ingress controller using Helm charts, see the Helm chart installation guide.

For deployments using the OpenShift operator

For deployments using the OpenShift operator, you need to edit the YAML definition to create CPX with ingress controller as specified in the step 6 of Deploy the NetScaler Ingress Controller as a sidecar with NetScaler CPX using NetScaler Operator and specify the ports as shown in the following example:

```
servicePorts:
  - port: 80
    protocol: TCP
    name: http
  - port: 443
    protocol: TCP
    name: https
  - port: 6379
    protocol: TCP
    name: tcp
<!--NeedCopy-->
```

The following sample configuration is an example for deployment using the OpenShift Operator. The service port definitions are highlighted in green.


```
  nitroReadTimeout: 20
  nodeSelector:
    key: ''
    value: ''
  nsConfigDnsRec: false
  nsCookieVersion: '0'
  nsDnsNameserver: ''
  nsGateway: 192.168.1.1
  nsHTTP2ServerSide: 'OFF'
  nsIP: 192.168.1.2
  nsLbHashAlgo:
    hashAlgorithm: DEFAULT
    hashFingers: 256
    required: false
  nsProtocol: http
  nsSvcLbDnsRec: false
  openshift: true
  optimizeEndpointBinding: false
  podAnnotations: {}
  profileHttpFrontend: {}
  profileSslFrontend: {}
  profileTcpFrontend: {}
  pullPolicy: IfNotPresent
  rbacRole: false
  replicaCount: 1
  resources: {}
  routeLabels: ''
  serviceAccount:
    create: true
  serviceAnnotations: {}
  servicePorts:
  - port: 80
    protocol: TCP
    name: http
  - port: 443
    protocol: TCP
    name: https
  - port: 6379
    protocol: TCP
    name: tcp
  serviceSpec:
    externalTrafficPolicy: Cluster
    loadBalancerIP: ''
    loadBalancerSourceRanges: []
  serviceType:
    loadBalancer:
      enabled: false
```

How to set up dual-tier deployment

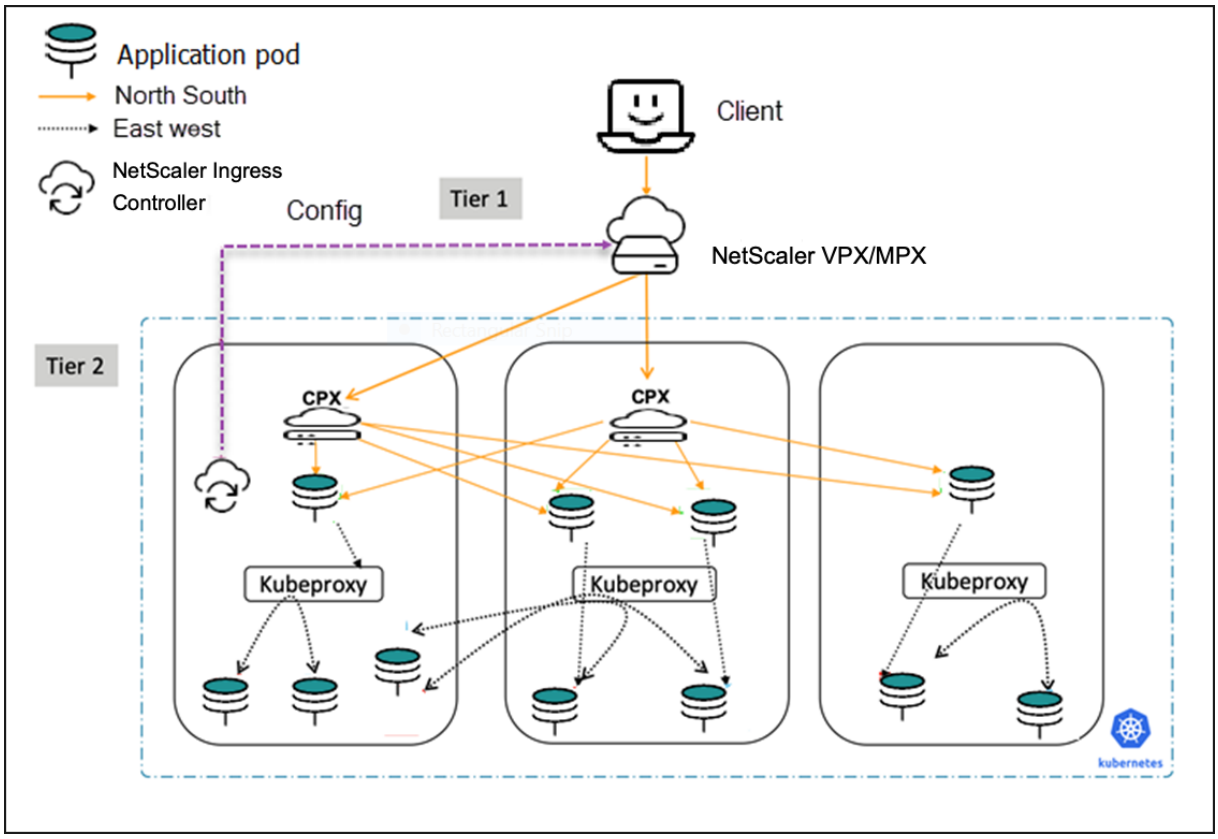
December 31, 2023

In a dual-tier deployment, NetScaler VPX or MPX is deployed outside the Kubernetes cluster (Tier-1) and NetScaler CPXs are deployed inside the Kubernetes cluster (Tier-2).

NetScaler MPX or VPX devices in Tier-1 proxy the traffic (North-South) from the client to NetScaler CPXs in Tier-2. The Tier-2 NetScaler CPX then routes the traffic to the microservices in the Kubernetes cluster. The NetScaler Ingress Controller deployed as a standalone pod configures the Tier-1 NetScaler. And, the sidecar NetScaler Ingress Controller in one or more NetScaler CPX pods configures the associated NetScaler 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:



Setup process

The NetScaler Ingress Controller repo provides a sample Apache microservice and manifests for NetScaler CPX for Tier-2, ingress object for Tier-2 NetScaler CPX, NetScaler Ingress Controller, and an ingress object for Tier-1 NetScaler 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 NetScaler MPX or VPX on a multi-NIC deployment mode outside the Kubernetes cluster.
   - For instructions to deploy NetScaler MPX, see NetScaler documentation.
   - For instructions to deploy NetScaler VPX, see Deploy a NetScaler VPX instance.

   Perform the following after you deploy NetScaler VPX or MPX:

a) Configure an IP address from the subnet of the Kubernetes cluster as SNIP on the NetScaler. For information on configuring SNIPs in NetScaler, see Configuring Subnet IP Addresses (SNIPs).

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 NetScaler Ingress Controller YAML file to enable NetScaler Ingress Controller to configure the Tier-1 NetScaler.

> **Note:**
>
> It is not mandatory to use SNIP as `NS_IP`. If the management IP address of the NetScaler is reachable from NetScaler 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 NetScaler VPX. As NetScaler 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 NetScaler VPX to handle this scenario.

d) Create a NetScaler system user account specific to NetScaler Ingress Controller. NetScaler Ingress Controller uses the system user account to automatically configure the Tier-1 NetScaler.

e) Configure your on-premises firewall or security groups on your cloud to allow inbound traffic to the ports required for NetScaler. 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:

```
kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/dual-tier/manifest/apache.yaml
```

4. Deploy NetScaler CPX as Tier-2 ingress. Use the following command:

```
kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/dual-tier/manifest/tier-2-cpx.yaml
```

5. Create an ingress object for the Tier-2 NetScaler CPX. Use the following command:

```
kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/dual-tier/manifest/ingress-tier-2-cpx.yaml
```

6. Deploy the NetScaler Ingress Controller for Tier-1 NetScaler. Perform the following:

a) Download the NetScaler Ingress Controller manifest file. Use the following command:


```
wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/dual-tier/manifest/tier-1-vpx-cic.yaml
```

b) Edit the NetScaler Ingress Controller manifest file and enter the values for the following environmental variables:

Environment Variable	Mandatory or Optional	Description
NS_IP	Mandatory	The IP address of the NetScaler appliance. For more details, see Prerequisites.
NS_USER and NS_PASSWORD	Mandatory	The user name and password of the NetScaler VPX or MPX appliance used as the Ingress device. For more details, see Prerequisites.
EULA	Mandatory	The End User License Agreement. Specify the value as `Yes`.
LOGLEVEL	Optional	The log levels to control the logs generated by NetScaler 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
NS_PROTOCOL and NS_PORT	Optional	Defines the protocol and port that must be used by NetScaler Ingress Controller to communicate with NetScaler. By default, NetScaler Ingress Controller uses HTTPS on port 443. You can also use HTTP on port 80.

Environment Variable	Mandatory or Optional	Description
ingress-classes	Optional	If multiple ingress load balancers are used to load balance different ingress resources. You can use this environment variable to specify NetScaler Ingress Controller to configure NetScaler associated with specific ingress class. For information on Ingress classes, see Ingress class support
NS_VIP	Optional	NetScaler Ingress Controller uses the IP address provided in this environment variable to configure a virtual IP address to the NetScaler that receives Ingress traffic.

c) Deploy the updated NetScaler Ingress Controller manifest file. Use the following command:

```
kubectl create -f tier-1-vpx-cic.yaml
```

7. Create an ingress object for the Tier-1 NetScaler. Use the following command:

```
kubectl create -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/dual-tier/manifest/ingress-tier-1-vpx.yaml
```

8. Update DNS server details in the cloud or on-premises to point your website to the VIP of the Tier-1 NetScaler.

   For example: `citrix-ingress.com 192.250.9.1`

   Where `192.250.9.1` is the VIP of the Tier-1 NetScaler and `citrix-ingress.com` is the microservice running in your Kubernetes cluster.

9. Access the URL of the microservice to verify the deployment.

Set up dual-tier deployment using one step deployment manifest file

For easy deployment, the NetScaler 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:

```
wget https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/deployment/dual-tier/manifest/all-in-one-dual-tier-demo.yaml
```

2. Edit the all-in-one deployment manifest file and enter the values for the following environmental variables:

Environment Variable	Mandatory or Optional	Description
NS_IP	Mandatory	The IP address of the NetScaler appliance. For more details, see Prerequisites.
NS_USER and NS_PASSWORD	Mandatory	The user name and password of the NetScaler VPX or MPX appliance used as the Ingress device. For more details, see Prerequisites.
EULA	Mandatory	The End User License Agreement. Specify the value as `Yes`.
LOGLEVEL	Optional	The log levels to control the logs generated by NetScaler 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

Environment Variable	Mandatory or Optional	Description
NS_PROTOCOL and NS_PORT	Optional	Defines the protocol and port that must be used by NetScaler Ingress Controller to communicate with NetScaler. By default, NetScaler Ingress Controller uses HTTPS on port 443. You can also use HTTP on port 80.
ingress-classes	Optional	If multiple ingress load balancers are used to load balance different ingress resources. You can use this environment variable to specify NetScaler Ingress Controller to configure NetScaler associated with specific ingress class. For information on Ingress classes, see [Ingress class support](/en-us/netscaler-k8s-ingress-controller/configure/ingress-classes.html
NS_VIP	Optional	NetScaler Ingress Controller uses the IP address provided in this environment variable to configure a virtual IP address to the NetScaler that receives Ingress traffic. **Note:** NS_VIP acts as a fallback when the frontend-ip annotation is not provided in Ingress yaml. Not supported for Type Loadbalancer service.

3. Deploy the updated all-in-one deployment manifest file. Use the following command:

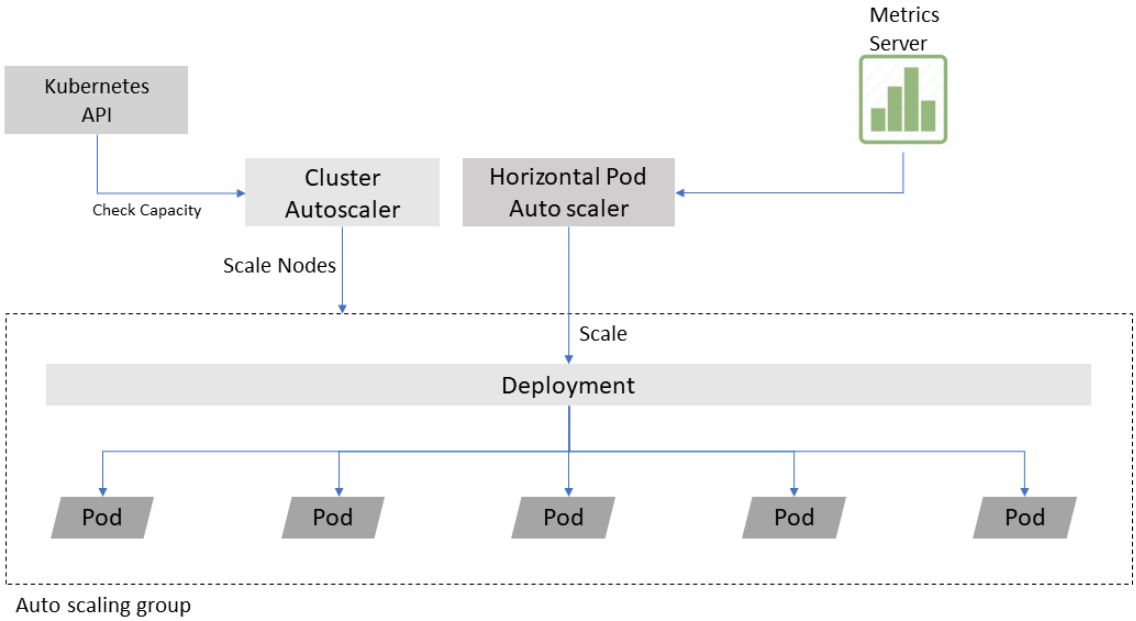
```
kubectl create -f all-in-one-dual-tier-demo.yaml
```


Horizontal pod autoscaler for NetScaler CPX with custom metrics

December 31, 2023

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.



NetScaler provides a custom-metric based HPA solution for NetScaler CPX.
By default, the metrics server only gives CPU and memory metrics for a pod.
NetScaler 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.

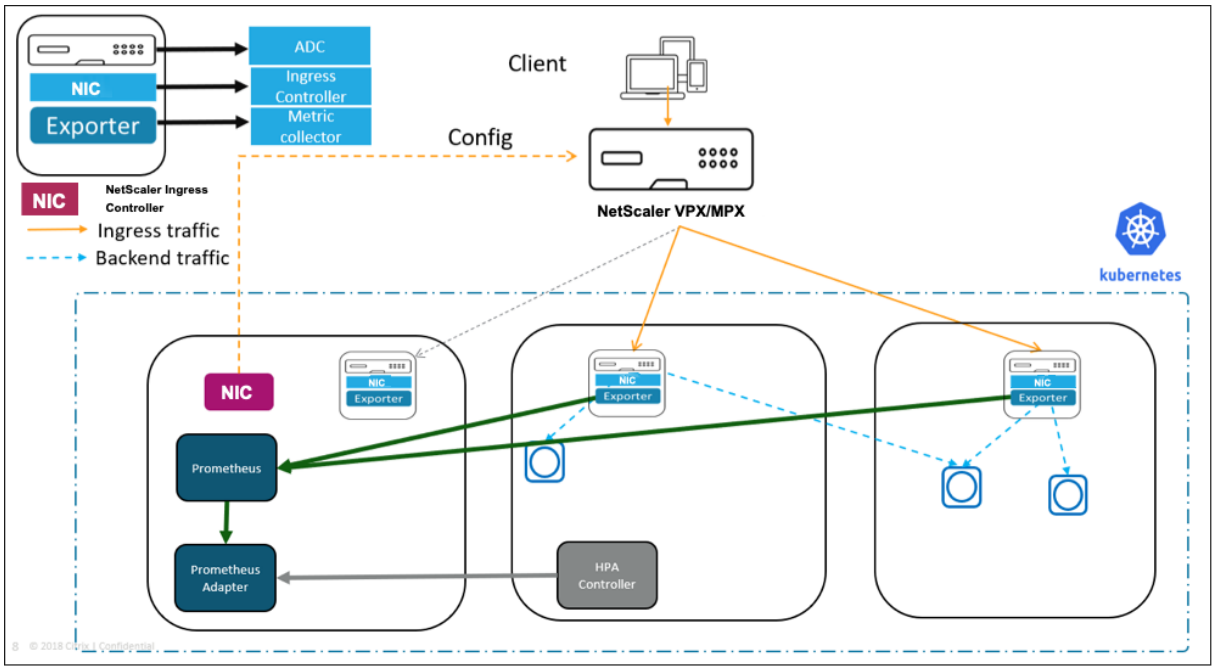
NetScaler CPX HPA solution

NetScaler CPX HPA solution consists of the following components:

- NetScaler VPX: NetScaler VPX or MPX is deployed at Tier-1 and load balances the client requests among the NetScaler CPX pods inside the cluster.

- NetScaler CPX: NetScaler CPX deployed inside the cluster acts as a Tier-2 load balancer for the endpoint application pods. The NetScaler CPX pod is running along with the NetScaler Ingress Controller and NetScaler metric exporter as sidecars.
- NetScaler Ingress Controller: The NetScaler Ingress Controller is an ingress controller which is built around the Kubernetes Ingress and automatically configures NetScaler based on the Ingress resource configuration. The NetScaler Ingress Controller deployed as a stand-alone pod configures the NetScaler VPX and other instances configures NetScaler CPXs.
- NetScaler Metrics Exporter: The NetScaler Metrics Exporter exports the application performance metrics to the open-source monitoring system Prometheus. The NetScaler Metrics Exporter collects metrics from NetScaler CPX and exposes it in a format that Prometheus can understand.
- Prometheus: Prometheus is an open-source systems monitoring and alerting toolkit. Prometheus is used to collect metrics from NetScaler 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 NetScaler CPX HPA solution works.



The Tier-1 NetScaler VPX load balances the NetScaler CPXs at Tier-2. NetScaler 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 NetScaler CPX deployment and creates another NetScaler CPX pod to handle the load.

Deploy NetScaler CPX HPA solution

Perform the following steps to deploy the NetScaler CPX HPA solution.

1. Clone the citrix-k8s-ingress-controller repository from GitHub using the following command.

   ```
   1 git clone https://github.com/citrix/citrix-k8s-ingress-controller.
       git
   ```

   After cloning, change your directory to the HPA folder with the following command.

   ```
   1 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 NetScaler VPX.

   - `VPX_IP`: IP address of the NetScaler VPX
   - `VPX_PASSWORD`: The password of the `nsroot` user on the NetScaler VPX
   - `VIRTUAL_IP_VPX`: The IP address on which the sample guesbook application is accessed.

3. Create all the required resources by running the `create_all.sh` file.

   ```
   1 ./create_all.sh
   ```

   This step creates the following resources:

   - Prometheus and Grafana for monitoring
   - NetScaler CPX with the NetScaler Ingress Controller and metrics exporter as sidecars
   - NetScaler Ingress Controller as a stand-alone pod to configure NetScaler VPX
   - A sample guestbook application
   - HPA controller for monitoring the NetScaler 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 NetScaler 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 NetScaler CPX autoscale deployment.

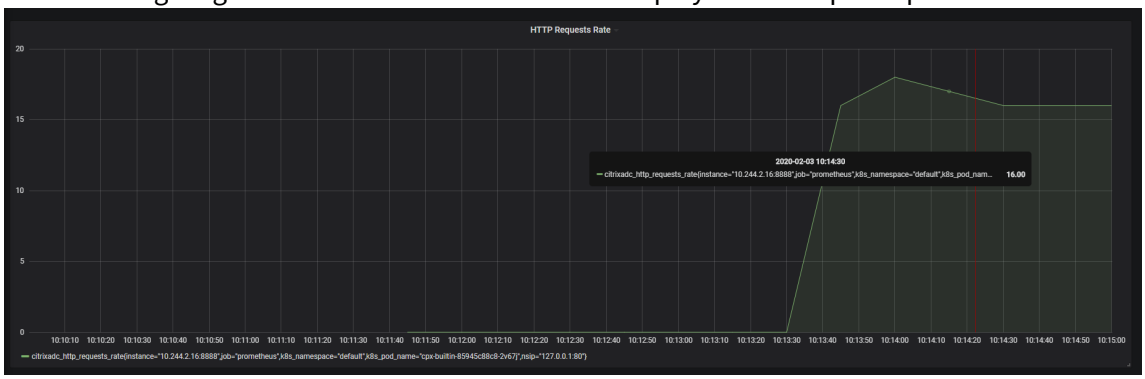
   The NetScaler CPX deployment HPA has been configured in such a way that when the average HTTP requests rate of the NetScaler CPX goes above 20 requests per second, it autoscales. You can use the following scripts provided in the HPA folder for sending traffic:

- `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 NetScaler CPX.

```
./16_curl.sh
```

The following diagram a Grafana dashboard which displays HTTP requests per second.



The following output shows the HPA state with 16 HTTP RPS.

```
NAME          REFERENCE                TARGETS    MINPODS    MAXPODS    REPLICAS    AGE
cpx-builtin   Deployment/cpx-builtin   16/20      1          3          1           4m9s
```

b. Run the `30_curl.sh` script to send 30 HTTP requests per second to NetScaler CPX.

```
./30_curl.sh
```

When you run this script, the threshold of 20 requests that was set has been crossed and the NetScaler 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 NetScaler CPX pods.

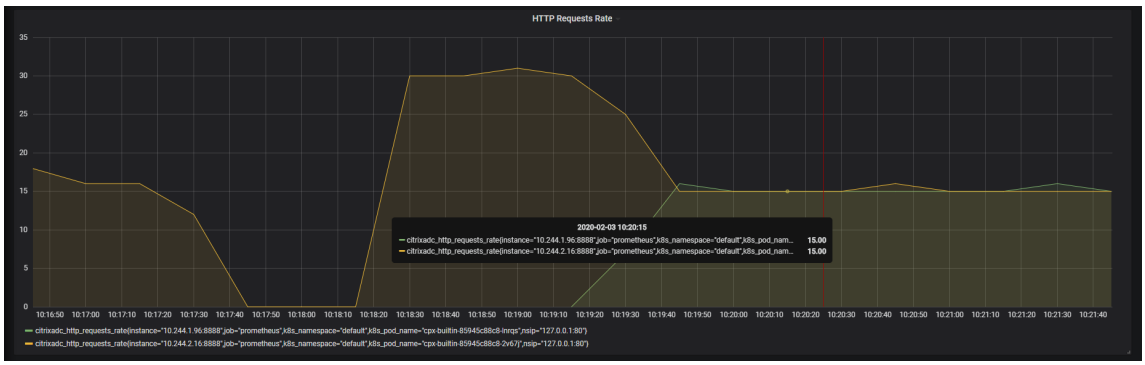
The following output shows the state of HPA when the target is crossed.

```
NAME          REFERENCE                TARGETS    MINPODS    MAXPODS    REPLICAS    AGE
cpx-builtin   Deployment/cpx-builtin   30/20      1          3          1           8m4s
```

The following output shows that the number of replicas of NetScaler CPX have gone up to 2 and the average value of HTTP RPS comes down to 15.

```
NAME          REFERENCE                TARGETS    MINPODS    MAXPODS    REPLICAS    AGE
cpx-builtin   Deployment/cpx-builtin   15/20      1          3          2           9m24s
```

The following diagram shows a Grafana dashboard with two NetScaler CPXs load balancing the traffic.



6. Clean up by executing the `delete_all.sh` script.

```
./delete_all.sh
```

> **Note:**
>
> If the Tier-1 NetScaler VPX is not present, use NodePort to expose the NetScaler CPX service.

Deploy Direct Server Return

December 31, 2023

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 NetScaler, see the NetScaler documentation.

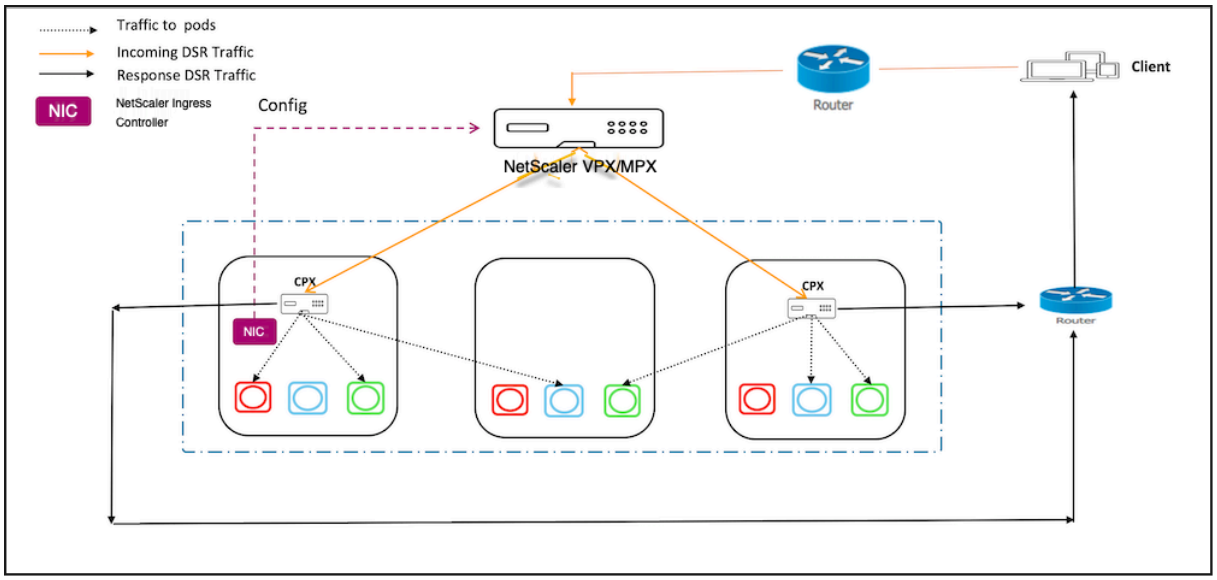
DSR solution is useful in the following situations:

- 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 NetScaler

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 NetScaler

Perform the steps in the following sections to deploy DSR for applications deployed on the Kubernetes cluster.

Deploy NetScaler 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 node controller network is configured as 192.168.1.0/24. Hence, the command to create IP tunnel is provided as `add iptunnel dsr 192.168.1.254 255.255.255.0 *`. You need to specify the value according to your CNC configuration.

3. Deploy NetScaler CPX on the namespace `dsr`.

```
kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/example/dsr/KubernetesConfig/citrix-k8s-cpx-ingress.yml  -n dsr
```

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.

```
kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/example/dsr/KubernetesConfig/guestbook-all-in-one.yaml  -n dsr
```

2. Expose the guestbook application using Ingress.

   a) Download the guestbook ingress YAML file using the following command.

```
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.

```
ingress.citrix.com/frontend-ip: "<ip-address>"
```

   c) Save the YAML file and deploy the Ingress resource using the following command.

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

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 node controller using the following command.

```
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 node controller.

3. Save the YAML file and deploy the node controller.

```
kubectl create -f citrix-k8s-node-controller.yaml -n dsr
```

Deploy the NetScaler Ingress Controller for Tier-1 ADC and expose NetScaler CPX as a service

Perform the following steps to deploy the NetScaler Ingress Controller as a stand-alone pod and create an Ingress resource for Tier-2 NetScaler CPX.

1. Download the NetScaler Ingress Controller YAML file using the following command.

```
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 NetScaler Ingress Controller.

   - NS_IP
   - NS_USER
   - NS_PASSWORD

   For more information, see Deploy the NetScaler Ingress Controller using YAML.

3. Save the YAML file and deploy the NetScaler Ingress Controller.

```
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 NetScaler CPX.

```
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.

```
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:

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.

```
sysctl -w net.ipv4.conf.all.rp_filter=0
sysctl -w net.ipv4.conf.cni0.rp_filter=0
sysctl -w net.ipv4.conf.eth0.rp_filter=0
sysctl -w net.ipv4.conf.cni0.rp_filter=0
sysctl -w net.ipv4.conf.default.rp_filter=0
```

Support for admission controller webhooks

December 31, 2023

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.

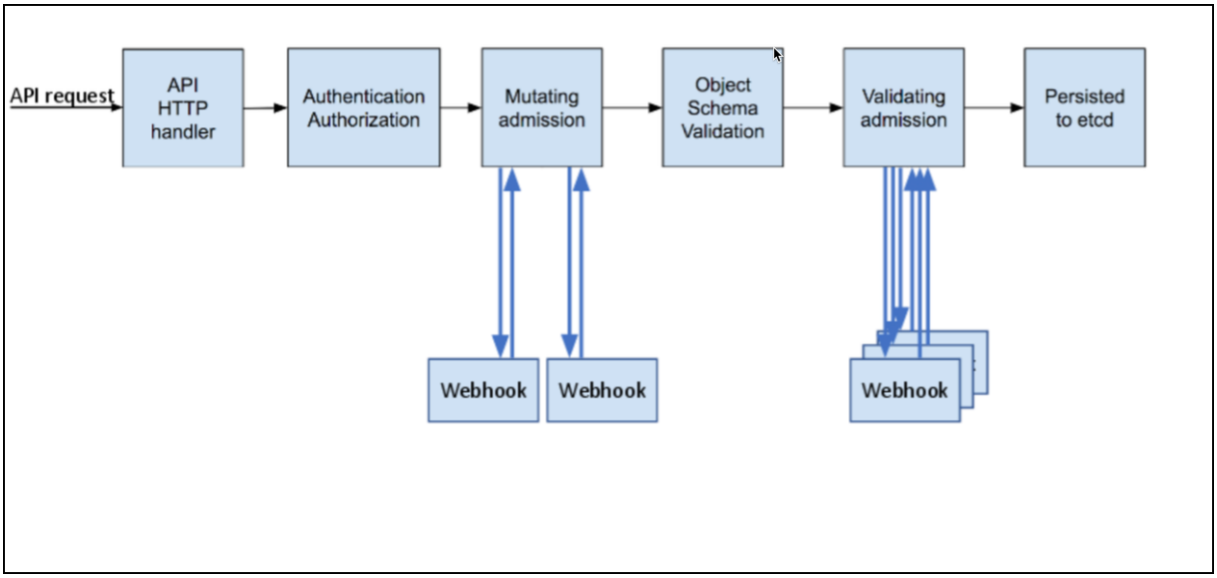
Using the Admission controller webhooks Kubernetes cluster administrators can create additional plug-ins to the admission chain of API server without recompiling them. Admission controller webhooks can be executed whenever a resource is created, updated, or deleted.

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:

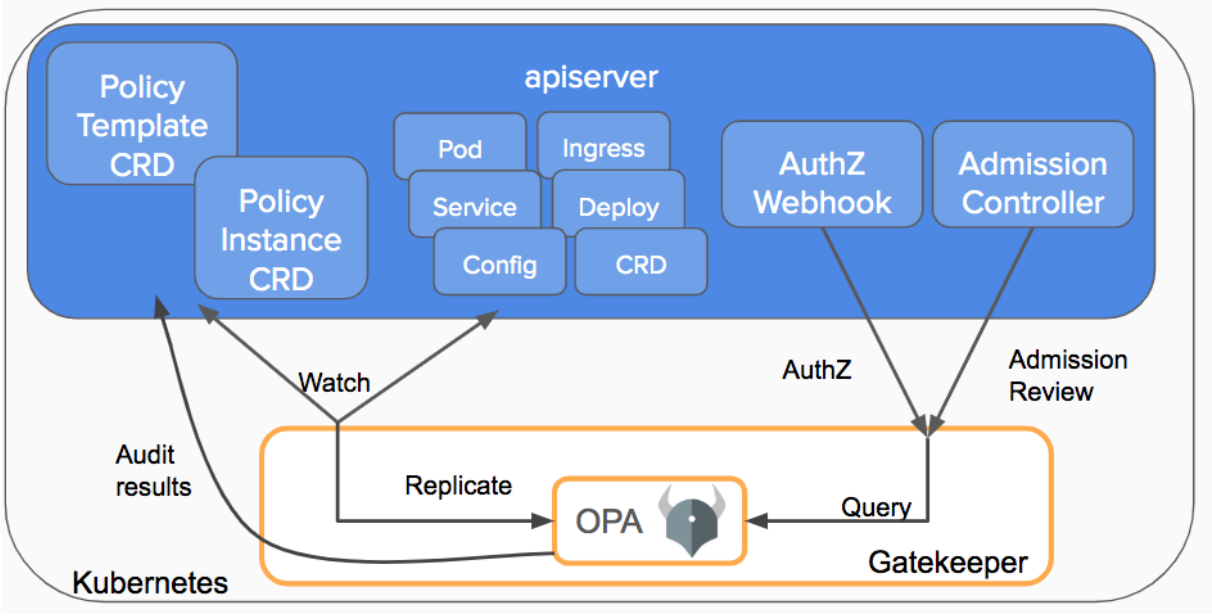
- To mandate a reasonable security baseline across an entire namespace or cluster mandating. For example, disallowing containers from running as root or making sure the container's root filesystem is always mounted as read-only.
- To enforce the adherence to certain standard and practices for labels, 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 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.



(image credit)

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)
- 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:

  ```
  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:

```
minikube start --extra-config=apiserver.enable-admission-plugins=
    NamespaceLifecycle,LimitRanger,ServiceAccount,
    DefaultStorageClass,DefaultTolerationSeconds,NodeRestriction,
    MutatingAdmissionWebhook,ValidatingAdmissionWebhook`
```

- Ensure that you have cluster administrator permissions.

```
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 `HttpsOnly` as a sample policy using Gatekeeper. The `HttpsOnly` policy allows only an Ingress configuration with HTTPS.

1. Install Gatekeeper using the following command.

> **Note:**
>
> In this step, Gatekeeper is installed using a prebuilt image. You can install Gatekeeper using various methods mentioned in the Gatekeeper installation.

```
# kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/master/deploy/gatekeeper.yaml
```

You can verify the installation using the following command.

```
kubectl get crd | grep -i constraintsonstrainttemplates.templates.gatekeeper.sh
```

You can check all the constraint templates using the following command:

```
kubectl get constrainttemplates.templates.gatekeeper.sh
```

2. Apply the `httpsonly` constraint template.

```
kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/docs/how-to/webhook/httpsonly/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-https-only] 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-https-only] 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
kubectl delete -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/docs/how-to/webhook/httpsonly/template.yaml
kubectl delete -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/master/deploy/gatekeeper.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 NetScaler Ingress Controller

December 31, 2023

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 NetScaler Ingress Controller to support ConfigMap.

```
        args:
          - --ingress-classes
            citrix
          - --configmap
            default/cic-configmap
```

4. Deploy the NetScaler Ingress Controller as a stand-alone pod by applying the edited YAML file.

```
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.

```
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`).

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

Following is a sample content for the `grpc-service.yaml` file.

```
  apiVersion: apps/v1
  kind: Deployment
  metadata:
    name: grpc-service
  spec:
    replicas: 1
    selector:
      matchLabels:
        app: grpc-service
    template:
      metadata:
        labels:
          app: grpc-service
      spec:
        containers:
        - image: registry.cn-hangzhou.aliyuncs.com/acs-sample/grpc
          -server:latest
          imagePullPolicy: Always
          name: grpc-service
          ports:
          - containerPort: 50051
            protocol: TCP
```



```
22          restartPolicy: Always
23    ---
24    apiVersion: v1
25    kind: Service
26    metadata:
27      name: grpc-service
28    spec:
29      ports:
30      - port: 50051
31        protocol: TCP
32        targetPort: 50051
33      selector:
34        app: grpc-service
35      sessionAffinity: None
36      type: NodePort
```

7. Create a certificate for the gRPC Ingress configuration.

```
1  openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout tls.
      key -out tls.crt -subj "/CN=grpc.example.com/O=grpc.example.com
      "
2
3  kubectl create secret tls grpc-secret --key tls.key --cert tls.crt
4
5  secret "grpc-secret" created
```

8. Enable HTTP2 using Ingress annotations. See HTTP/2 support for steps to enable HTTP2 using the NetScaler 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:

```
1    apiVersion: networking.k8s.io/v1
2    kind: Ingress
3    metadata:
4      annotations:
5        ingress.citrix.com/frontend-httpprofile: '{
6  "http2":"enabled", "http2direct" : "enabled" }
7  '
8        ingress.citrix.com/frontend-ip: 192.0.2.1
9        ingress.citrix.com/secure-port: "443"
10       kubernetes.io/ingress.class: citrix
11     name: frontend-ingress
12   spec:
13     rules:
14     - {
15   }
16
17     tls:
```



```
    - {
  }
```

- 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).

  kubectl apply -f backend-ingress.yaml

The content of the `backend-ingress.yaml` file is provided as follows:

```
  apiVersion: networking.k8s.io/v1
  kind: Ingress
  metadata:
    annotations:
      ingress.citrix.com/backend-httpprofile: '{
 "grpc-service":{
 "http2": "enabled", "http2direct" : "enabled" }
  }
 '
      ingress.citrix.com/frontend-ip: 192.0.2.2
      ingress.citrix.com/secure-port: "443"
      kubernetes.io/ingress.class: citrix
    name: grpc-ingress
  spec:
    rules:
    - host: grpc.example.com
      http:
        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.

```
grpcurl -v -insecure -d '{
 "name": "gRPC" }
 ' grpc.example.com:443 helloworld.Greeter.SayHello
```

The output of the command is shown as follows:

```
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.

   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.

```
kubectl create -f ratelimit-crd-object.yaml
```

4. Test gRPC traffic using the `grpcurl` command.


```
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:

```
Error invoking method "helloworld.Greeter.SayHello": failed to
    query for service descriptor "helloworld.Greeter": rpc error:
    code = Unavailable desc =

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.

```
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.

```
kubectl create -f rewrite-crd-object.yaml
```

4. Test the gRPC traffic using the `grpcurl` command.

```
grpcurl -v -insecure -d '{
 "name": "gRPC" }
 ' grpc.example.com:443 helloworld.Greeter.SayHello
```


This command adds a session id in the gRPC request response.

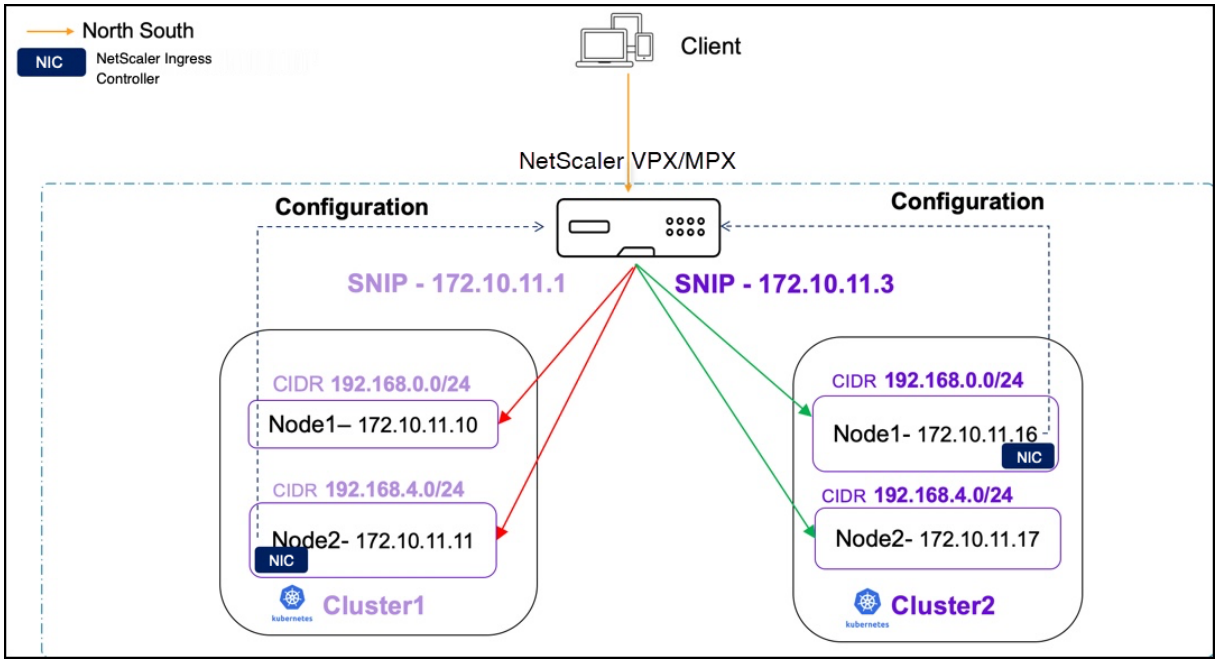
```
Resolved method descriptor:
rpc SayHello ( .helloworld.HelloRequest ) returns ( .helloworld.
   HelloReply );

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

December 31, 2023

When you are using a single NetScaler to load balance multiple Kubernetes clusters, the NetScaler Ingress Controller adds pod CIDR networks through static routes. These routes establish networking connectivity between Kubernetes pods and NetScaler. However, when the pod CIDRs overlap there may be route conflicts. NetScaler 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 NetScaler Ingress Controller. Using net profile, the SNIP is bound to all service groups created by the same NetScaler 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 NetScaler VPX or MPX.



Configure PBR using the NetScaler 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 NetScaler Ingress Controller deployment YAML file during bootup or using ConfigMap.

Perform the following steps to deploy the NetScaler Ingress Controller and configure PBR using ConfigMap.

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 NetScaler 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 NetScaler Ingress Controller](/en-us/netscaler-k8s-ingress-controller/cic-yaml.html) documentation.
```

3. Deploy the NetScaler 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 NetScaler 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:
  - 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 NetScaler after deploying the NetScaler Ingress Controller

This validation example uses a two node Kubernetes cluster with the NetScaler Ingress Controller deployed along with the following ConfigMap with two SNIPs.


```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cic-configmap
  namespace: default
data:
   NS_SNIPS: '["1.2.3.4","5.6.7.8"]'
```

You can verify that the NetScaler Ingress Controller adds the following configurations to the ADC:

1. An `IPset` of all `NS_SNIPs` provided by the ConfigMap is added.

```
> show ipset k8s-pbr_ipset

1)       Name: k8s-pbr_ipset


       IP:1.2.3.4

       IP:5.6.7.8
 Done
```

2. A net profile is added with the `SrcIP` set to the `IPset`.

```
> show netprofile k8s-pbr_netprof

1)      Name: k8s-pbr_netprof

       SrcIP: k8s-pbr_ipset
       SrcIPPersistency: DISABLED
       OverrideLSN: DISABLED
       MBF: DISABLED
       Proxy Protocol: DISABLED
       Proxy Protocol version: V1
       Proxy Protocol After TLS: DISABLED
 Done
```

3. The service group added by the NetScaler Ingress Controller contains the net profile set.


```
> show servicegroup
1)      k8s-frontend_80_sgp_5udfadwklqkfukj3rsuoao5b6wxauf5j - HTTP
        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
        Monitoring Owner: 0
        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: ???
        Network profile name: k8s-pbr_netprof
        Process Local: DISABLED
        Traffic Domain: 0
        Comment: "ing:web-ingress,ingport:80,ns:default,svc:frontend,svcport:80"
 Done
```

4. Finally, the NetScaler Ingress Controller adds PBRs.


```
> shpbr
1)      Name: k8s-1.2.3.4_10.106.175.86
        Action: ALLOW                          Hits: 0
        srcIP = 1.2.3.4
        destIP = 192.168.0.0-192.168.0.255
        srcMac:
        Protocol:
        Vlan:                                 Interface:
        Active Status: ENABLED                 Applied Status: APPLIED
        Priority: 30
        NextHop: 10.106.175.86

2)      Name: k8s-5.6.7.8_10.106.175.86
        Action: ALLOW                          Hits: 0
        srcIP = 5.6.7.8
        destIP = 192.168.0.0-192.168.0.255
        srcMac:
        Protocol:
        Vlan:                                 Interface:
        Active Status: ENABLED                 Applied Status: APPLIED
        Priority: 40
        NextHop: 10.106.175.86

3)      Name: k8s-1.2.3.4_10.106.175.87
        Action: ALLOW                          Hits: 0
        srcIP = 1.2.3.4
        destIP = 192.168.16.0-192.168.16.255
        srcMac:
        Protocol:
        Vlan:                                 Interface:
        Active Status: ENABLED                 Applied Status: APPLIED
        Priority: 50
        NextHop: 10.106.175.87

4)      Name: k8s-5.6.7.8_10.106.175.87
        Action: ALLOW                          Hits: 0
        srcIP = 5.6.7.8
        destIP = 192.168.16.0-192.168.16.255
        srcMac:
        Protocol:
        Vlan:                                 Interface:
        Active Status: ENABLED                 Applied Status: APPLIED
        Priority: 60
        NextHop: 10.106.175.87
```

Here:

- 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 NetScaler 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 NetScaler Ingress Controller to validate the configuration.


```
root@flannel-master:~/pbr-demo# logs -f cic-vpx | grep pbr
2021-05-06 05:46:41,506  - INFO - [NSPBRConfig.py:configure_snips_netprofile_for_pbr:95] (MainThread) Successfully Added IP 5.6.7.8 to ADC
2021-05-06 05:46:41,539  - INFO - [NSPBRConfig.py:configure_snips_netprofile_for_pbr:95] (MainThread) Successfully Added IP 1.2.3.4 to ADC
2021-05-06 05:46:41,701  - INFO - [nitrointerface.py:add_ns_pbroute:4709] (MainThread) PBRoute for 192.168.0.0-192.168.0.255 with gateway:10.106.175.86 srcip:5.6.7.8 added to ADC
2021-05-06 05:46:41,739  - INFO - [nitrointerface.py:add_ns_pbroute:4709] (MainThread) PBRoute for 192.168.0.0-192.168.0.255 with gateway:10.106.175.86 srcip:1.2.3.4 added to ADC
2021-05-06 05:46:41,771  - INFO - [nitrointerface.py:add_ns_pbroute:4709] (MainThread) PBRoute for 192.168.16.0-192.168.16.255 with gateway:10.106.175.87 srcip:5.6.7.8 added to ADC
2021-05-06 05:46:41,802  - INFO - [nitrointerface.py:add_ns_pbroute:4709] (MainThread) PBRoute for 192.168.16.0-192.168.16.255 with gateway:10.106.175.87 srcip:1.2.3.4 added to ADC
```

Configure PBR using the node controller

You can configure PBR using the node controller for multiple Kubernetes clusters. When you are using a single NetScaler to load balance multiple Kubernetes clusters with 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, node controller needs to works in conjunction with the NetScaler Ingress Controller to bind the net profile to the service group.

Perform the following steps to configure PBR using the node controller:

1. While starting the node controller, provide the `CLUSTER_NAME` as an environment variable. Specifying this variable indicates that it is a multi-cluster deployment and the node controller configures PBR instead of static routes.

   Example:

   ```
   - name: CLUSTER_NAME
     value: "dev-cluster"
   ```

2. While deploying the NetScaler Ingress Controller, provide the `CLUSTER_NAME` as an environment variable. This value should be the same as the value provided in node controller.

   Example:

   ```
   - name: CLUSTER_NAME
     value: "dev-cluster "
   ```

3. Specify the argument `--enable-cnc-pbr` as `True` in the arguments section of the NetScaler Ingress Controller deployment YAML file. When you specify this argument, NetScaler Ingress Controller is aware that the node controller is configuring PBR on the NetScaler.

   Example:

   ```
   args:
    - --enable-cnc-pbr True
   ```

> **Note:**
>
> - The value provided for `CLUSTER_NAME` in the node controller and NetScaler 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 NetScaler VPX or MPX.

Validate PBR configuration on a NetScaler after deploying the node controller

This validation example uses a two node Kubernetes cluster with node controller and NetScaler Ingress Controller deployed.

You can verify that the following configurations are added to the ADC by node controller:

1. A net profile is added, with the value of `srcIP` set to the SNIP added by node controller while creating the VXLAN tunnel network between the NetScaler and Kubernetes nodes.

```
> show netprofile

1)      Name: cnc-cluster1_netprof

        SrcIP: 162.1.11.254
        SrcIPPersistency: DISABLED
        OverrideLSN: DISABLED
        MBF: DISABLED
        Proxy Protocol: DISABLED
        Proxy Protocol version: V1
 Done
```

2. NetScaler Ingress Controller binds the net profile to the service groups it creates.

```
> shservicegroup
1)      k8s-frontend_80_sgp_5udfadwklqkfukj3rsuoao5b6wxauf5j - HTTP
        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
        Monitoring Owner: 0
        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: ???
        Network profile name: cnc-cluster1_netprof
        Process Local: DISABLED
        Traffic Domain: 0
        Comment: "ing:web-ingress,ingport:80,ns:default,svc:frontend,svcport:80"
 Done
```

3. Finally, node controller adds PBRs.


```
> show pbr
1)      Name: cnc-cluster1_192.168.16.0-192.168.16.255
        Action: ALLOW                          Hits: 337
        srcIP = 162.1.11.254
        destIP = 192.168.16.0-192.168.16.255
        srcMac:
        Protocol:
        Vlan:                                 Interface:
        Active Status: ENABLED                 Applied Status: APPLIED
        Priority: 10
        NextHop: 162.1.11.2

2)      Name: cnc-cluster1_192.168.0.0-192.168.0.255
        Action: ALLOW                          Hits: 674
        srcIP = 162.1.11.254
        destIP = 192.168.0.0-192.168.0.255
        srcMac:
        Protocol:
        Vlan:                                 Interface:
        Active Status: ENABLED                 Applied Status: APPLIED
        Priority: 20
        NextHop: 162.1.11.1

 Done
```

Here:

- The number of PBRs is equal to number of Kuberntes nodes. In this case, it adds two PBRs.
- The `srcIP` of the PBR is the `SNIP` added by node controller in tunnel network . The `destIP` is the CNI overlay subnet range of the Kubernete node. The `NextHop` is the IP address of Kubernetes node's VXLAN Tunnel interface.

  > **Note:**
  >
  > 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 node controller configuration.

Single tier NetScaler Ingress solution for MongoDB

December 31, 2023

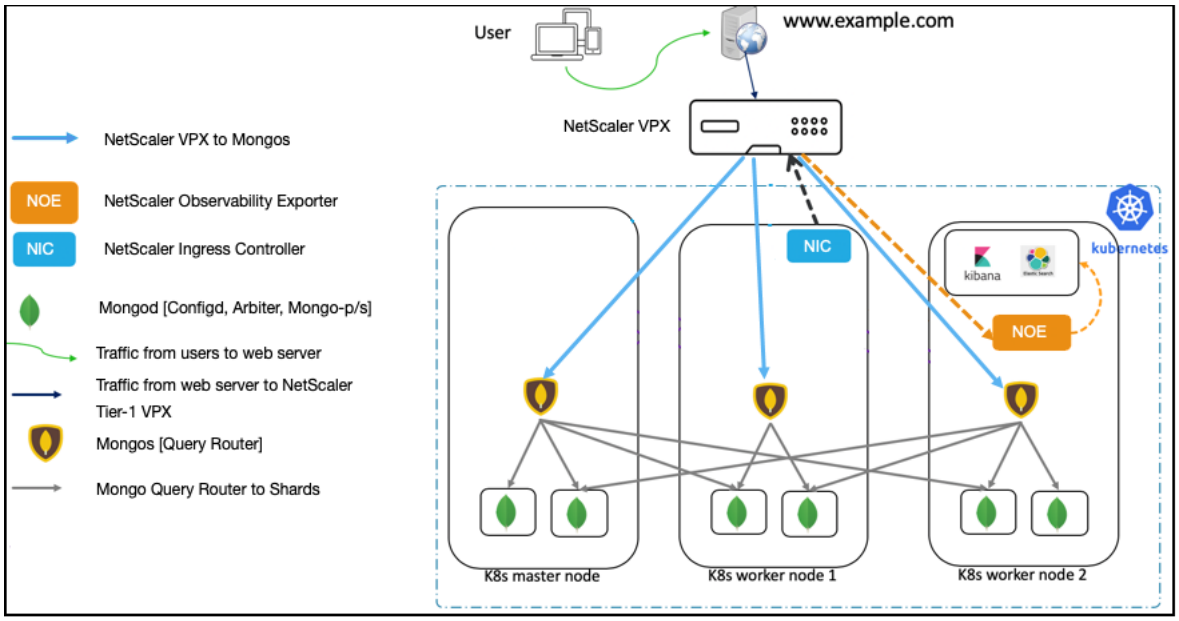
MongoDB is one of the most popular NoSQL databases which is designed to process and store massive amounts of unstructured data. Cloud-native applications widely use MongoDB as a NoSQL database in the Kubernetes platform.

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.

NetScaler 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 NetScalers. NetScaler 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 NetScaler Observability Exporter, you can export the MongoDB transactions to Elasticsearch and visualize them using Kibana dashboards to get deeper insights.

The following diagram explains NetScaler Ingress solution for MongoDB using a single-tier deployment of NetScaler.



In this solution, a NetScaler VPX is deployed outside the Kubernetes cluster (Tier-1) and NetScaler Observability Exporter is deployed inside the Kubernetes cluster.

The Tier-1 NetScaler VPX routes the traffic (North-South) from MongoDB clients to Mongo DB query routers (Mongos) in the Kubernetes cluster. NetScaler Observability Exporter is deployed inside the Kubernetes cluster.

As part of this deployment, an Ingress resource is created for NetScaler VPX (Tier-1 Ingress). The Tier-1 Ingress resource defines rules to enable load balancing for MongoDB traffic on NetScaler VPX and specifies the port for Mongo. Whenever MongoDB traffic arrives on the specified port on a NetScaler 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 NetScaler Observability Exporter instance aggregates transactions from NetScaler 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 NetScaler Observability Exporter.

Prerequisites

You must complete the following steps before deploying the NetScaler 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.
- 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 NetScaler VPX instance outside the Kubernetes cluster. For instructions on how to deploy NetScaler VPX, see Deploy a NetScaler VPX instance.

Perform the following after you deploy the NetScaler VPX:

1. Configure an IP address from the subnet of the Kubernetes cluster as SNIP on the NetScaler. For information on configuring SNIPs in NetScaler, 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 NetScaler Ingress Controller YAML file to enable the NetScaler Ingress Controller to configure the Tier-1 NetScaler.

   > **Note:**
   >
   > It is not mandatory to use SNIP as `NS_IP`. If the management IP address of the NetScaler is reachable from the NetScaler Ingress Controller then you can use the management IP address as `NS_IP`.

3. Create a NetScaler system user account specific to the NetScaler Ingress Controller. The NetScaler Ingress Controller uses the system user account to automatically configure the Tier-1 NetScaler.
4. Configure NetScaler VPX to forward DNS queries to CoreDNS pod IP addresses in the Kubernetes cluster.

   ```
   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 NetScaler VPX as:

```
add dns nameServer 192.244.0.3
add dns nameServer 192.244.0.2
```

Deploy the NetScaler Ingress solution for MongoDB

When you deploy the NetScaler Ingress solution for MongoDB, you deploy the following components in the Kubernetes cluster:

- A stand-alone NetScaler Ingress Controller for NetScaler VPX
- An Ingress resource for NetScaler VPX
- NetScaler Observability Exporter

Perform the following steps to deploy the NetScaler Ingress solution for MongoDB.

1. Create a Kubernetes secret with the user name and password for NetScaler VPX.

```
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.

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

3. Deploy the NetScaler Ingress Controller as a pod using the following steps.

   a) Download the NetScaler Ingress Controller manifest file. Use the following command:

```
wget https://raw.githubusercontent.com/citrix/citrix-k8s-
    ingress-controller/master/deployment/dual-tier/manifest/
    tier-1-vpx-cic.yaml
```

   b) Edit the NetScaler Ingress Controller manifest file and enter the values for the following environmental variables:

Environment Variable	Mandatory or Optional	Description
NS_IP	Mandatory	The IP address of the NetScaler appliance. For more details, see Prerequisites.
NS_USER and NS_PASSWORD	Mandatory	The user name and password of the NetScaler VPX or MPX appliance used as the Ingress device.

Environment Variable	Mandatory or Optional	Description
EULA	Mandatory	The End User License Agreement. Specify the value as `Yes`.
LOGLEVEL	Optional	The log levels to control the logs generated by NetScaler Ingress Controller. By default, the value is set to DEBUG. The supported values are: CRITICAL, ERROR, WARNING, INFO, and DEBUG.
NS_PROTOCOL and NS_PORT	Optional	Defines the protocol and port that must be used by the NetScaler Ingress Controller to communicate with NetScaler. By default, the NetScaler Ingress Controller uses HTTPS on port 443. You can also use HTTP on port 80.
ingress-classes	Optional	If multiple Ingress load balancers are used to load balance different Ingress resources. You can use this environment variable to specify the NetScaler Ingress Controller to configure NetScaler associated with a specific Ingress class. For information on Ingress classes, see Ingress class support
NS_VIP	Optional	NetScaler Ingress Controller uses the IP address provided in this environment variable to configure a virtual IP address to the NetScaler that receives Ingress traffic.

c) Specify or modify the following arguments in the NetScaler Ingress Controller YAML file.


```
1 args:
2  - --configmap
3     default/cic-configmap
4  - --ingress-classes
5     tier-1-vpx
```

d) Deploy the updated NetScaler Ingress Controller manifest file using the following command:

```
1 kubectl create -f tier-1-vpx-cic.yaml
```

4. Create an Ingress object for the Tier-1 NetScaler using the tier-1-vpx-ingress.yaml file.

```
1 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, NetScaler Ingress Controller configures the NetScaler 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 NetScaler VPX can route traffic to it. Here, `mongodb-mongos` is the service for MongoDB query routers.

```
1   apiVersion: networking.k8s.io/v1
2   kind: Ingress
3   metadata:
4     annotations:
5       ingress.citrix.com/analyticsprofile: '{
6  "tcpinsight": {
7  "tcpBurstReporting":"DISABLED" }
8   }
9  '
10      ingress.citrix.com/insecure-port: "27017"
11      ingress.citrix.com/insecure-service-type: mongo
12      ingress.citrix.com/insecure-termination: allow
13      kubernetes.io/ingress.class: tier-1-vpx
14    name: vpx-ingress
15  spec:
16    defaultBackend:
17        service:
18          name: mongodb-mongos
19          port:
20            number: 27017
```

5. Deploy NetScaler Observability Exporter with Elasticsearch as the endpoint using the coe-es-mongo.yaml file.

```
1 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.

```
apiVersion: v1
kind: 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",
                    "APPFW": "none",
                    "BOT": "none",
                    "VIDEOOPT": "none",
                    "BURST_CQA": "none",
                    "SLA": "none",
                    "MONGO": "all"
                 }
 ,
                "ProcessAlways": "no",
                "ProcessYieldTimeOut": "500",
                "MaxConnections": "512",
                "ElkMaxSendBuffersPerSec": "64",
                "JsonFileDump": "no"
              }

          }

      }

<!--NeedCopy-->
```


Verify the deployment of NetScaler 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.

```
root@kubenode222:~# kubectl  get pods,svc --all-namespaces -o wide
NAMESPACE     NAME                                          READY   STATUS    RESTARTS   AGE     IP              NODE          NOMINATED NODE   READINESS GATES
default       pod/cic-k8s-ingress-controller                1/1     Running   0          3h18m   10.244.1.44     kubenode223   <none>           <none>
default       pod/coe-es-66bb7f795d-7hftr                   1/1     Running   0          15d     10.244.1.32     kubenode223   <none>           <none>
default       pod/elasticsearch-5775b46ccd-gnt97            1/1     Running   0          90d     10.244.1.20     kubenode223   <none>           <none>
default       pod/kibana-5954647d85-6f4mj                   1/1     Running   0          90d     10.244.1.19     kubenode223   <none>           <none>
default       pod/mongo-786f4cb565-b49fm                    1/1     Running   0          95d     10.244.1.7      kubenode223   <none>           <none>
kube-system   pod/coredns-74ff55c5b-jtrnc                   1/1     Running   0          96d     10.244.0.3      kubenode222   <none>           <none>
kube-system   pod/coredns-74ff55c5b-rqvv4                   1/1     Running   0          96d     10.244.0.2      kubenode222   <none>           <none>
kube-system   pod/etcd-kubenode222                          1/1     Running   0          96d     10.102.217.222  kubenode222   <none>           <none>
kube-system   pod/kube-apiserver-kubenode222                1/1     Running   0          96d     10.102.217.222  kubenode222   <none>           <none>
kube-system   pod/kube-controller-manager-kubenode222       1/1     Running   0          96d     10.102.217.222  kubenode222   <none>           <none>
kube-system   pod/kube-flannel-ds-j4qrr                     1/1     Running   0          95d     10.102.217.222  kubenode222   <none>           <none>
kube-system   pod/kube-flannel-ds-nm556                     1/1     Running   0          95d     10.102.217.223  kubenode223   <none>           <none>
kube-system   pod/kube-proxy-9q8n9                          1/1     Running   0          95d     10.102.217.223  kubenode223   <none>           <none>
kube-system   pod/kube-proxy-n755g                          1/1     Running   0          96d     10.102.217.222  kubenode222   <none>           <none>
kube-system   pod/kube-scheduler-kubenode222                1/1     Running   0          96d     10.102.217.222  kubenode222   <none>           <none>
```

You can use the `kubectl get ingress` command as shown in the following example to get information about the Ingress objects deployed.

```
# kubectl get ingress

 NAME                HOSTS    ADDRESS    PORTS    AGE
 vpx-ingress         *                   80       22d
```

Verify observability for MongoDB traffic

This topic provides information on how to get visibility into MongoDB transactions using the NetScaler 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 NetScaler 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 NetScaler VPX.

   For example:

   ```
   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.

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 NetScaler Observability Exporter as a scrape target.

Following is a snippet of NetScaler Observability Exporter YAML file (coe-es-mongodb.yaml) with these annotations.

```
template:
  metadata:
    name: coe-es
    labels:
      app: coe-es
    annotations:
      prometheus.io/scrape: "true"
      prometheus.io/port: "5563"
```


Alternatively, you can manually add NetScaler 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.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cic-configmap
  namespace: default
data:
   NS_ANALYTICS_CONFIG: |
     distributed_tracing:
       enable: 'false'
       samplingrate: 0
     endpoint:
       server: 'coe-es.default.svc.cluster.local'
     timeseries:
       port: 5563
       metrics:
         enable: 'true'
         mode: 'prometheus'
       auditlogs:
         enable: 'false'
       events:
         enable: 'false'
     transactions:
       enable: 'true'
       port: 5557
<!--NeedCopy-->
```

In this YAML file, the following configuration enables metrics for Prometheus.

```
 metrics:
    enable: 'true'
    mode: 'prometheus'
```

Canary and blue-green deployment using NetScaler VPX and Azure pipelines for Kubernetes based applications

December 31, 2023

This topic provides information on how to achieve canary and blue-green deployment for Kubernetes applications using NetScaler VPX and Azure pipelines.

Canary deployment using NetScaler 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 NetScaler VPX and NetScaler 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, NetScaler VPX is deployed on the Azure platform to enable load balancing of an application and achieve canary deployment using NetScaler VPX. For more information on how to deploy NetScaler on Microsoft Azure, see the NetScaler documentation link.

Canary deployment using NetScaler

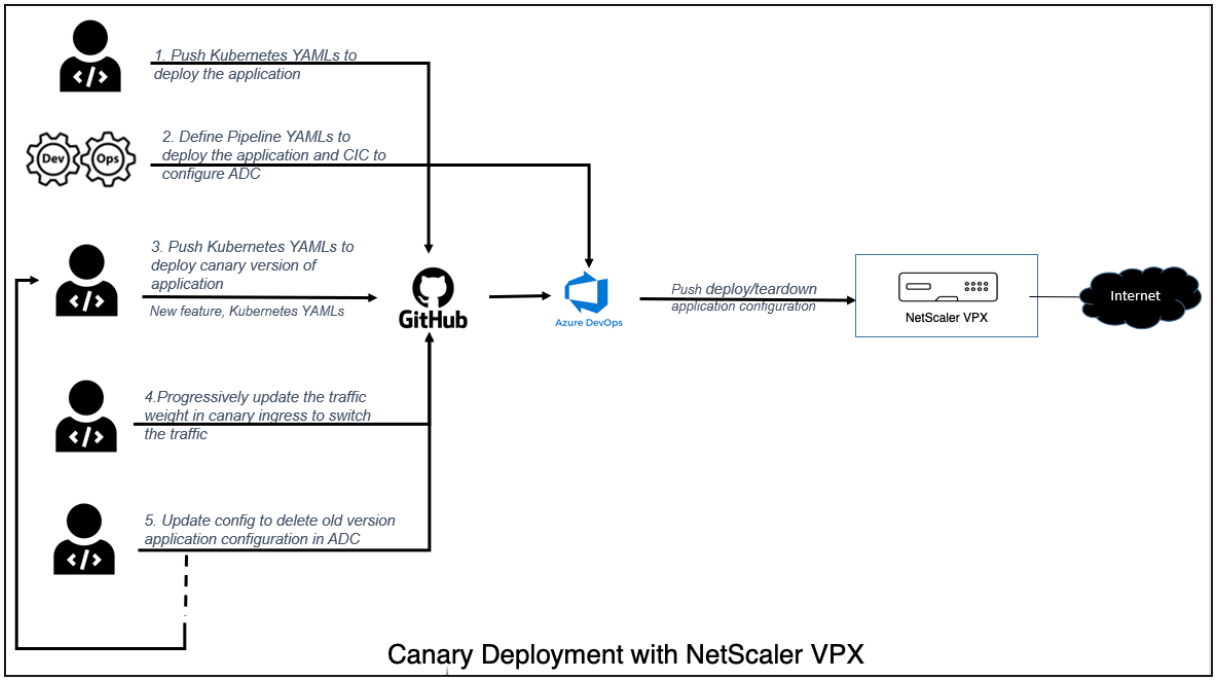
You can achieve canary deployment using NetScaler with Ingress annotations which is a rule based canary deployment. In this approach, you need to define an additional Ingress object with specific annotations to indicate that the application request needs to be served based on the rule based canary deployment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

- 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 NetScaler VPX with Azure pipelines

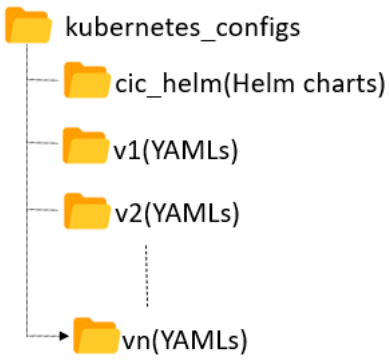
Citrix proposes a solution for canary deployment using NetScaler VPX and NetScaler Ingress Controller with Azure pipelines for Kubernetes based applications.



In this solution, there are three configuration directories:

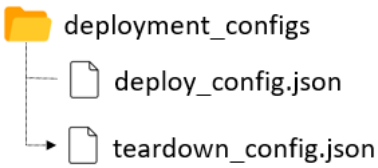
- `kubernetes_configs`
- `deployment_configs`
- `pipeline_configs`

kubernetes_configs This directory includes the version based application specific deployment YAML files and the Helm based configuration files to deploy NetScaler Ingress Controller which is responsible to push NetScaler configuration to achieve canary deployment.

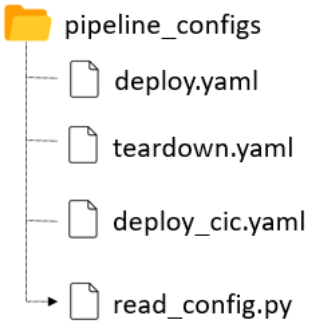


> **Note:**
>
> You can download the latest Helm charts from the NetScaler Ingress Controller Helm charts repository and place it under the `cic_helm` directory.

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.



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 NetScaler and Azure pipelines.

Prerequisites Ensure that:

- NetScaler 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 NetScaler 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 NetScaler 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.

```
{


 "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 NetScaler.
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 NetScaler 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 NetScaler 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

December 31, 2023

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 NetScaler to resolve the domain names and leverage the features of NetScaler.

Configure NetScaler to reach external services

You can configure NetScaler as a domain name resolver using NetScaler Ingress Controller. When you configure NetScaler as domain name resolver, you need to resolve:

- Reachability of NetScaler from microservices
- Domain name resolution at NetScaler to reach external services

Configure a service for reachability from Kubernetes cluster to NetScaler

To reach NetScaler from microservices, you have to define a headless service which would be resolved to a NetScaler service and thus the connectivity between microservices and NetScaler establishes.

```yaml
apiversion: v1
kind: Service
metadata:
  name: external-svc
spec:
  selector:
    app: cpx
  ports:
    - protocol: TCP
      port: 80
```

Configure NetScaler as a domain name resolver using NetScaler Ingress Controller

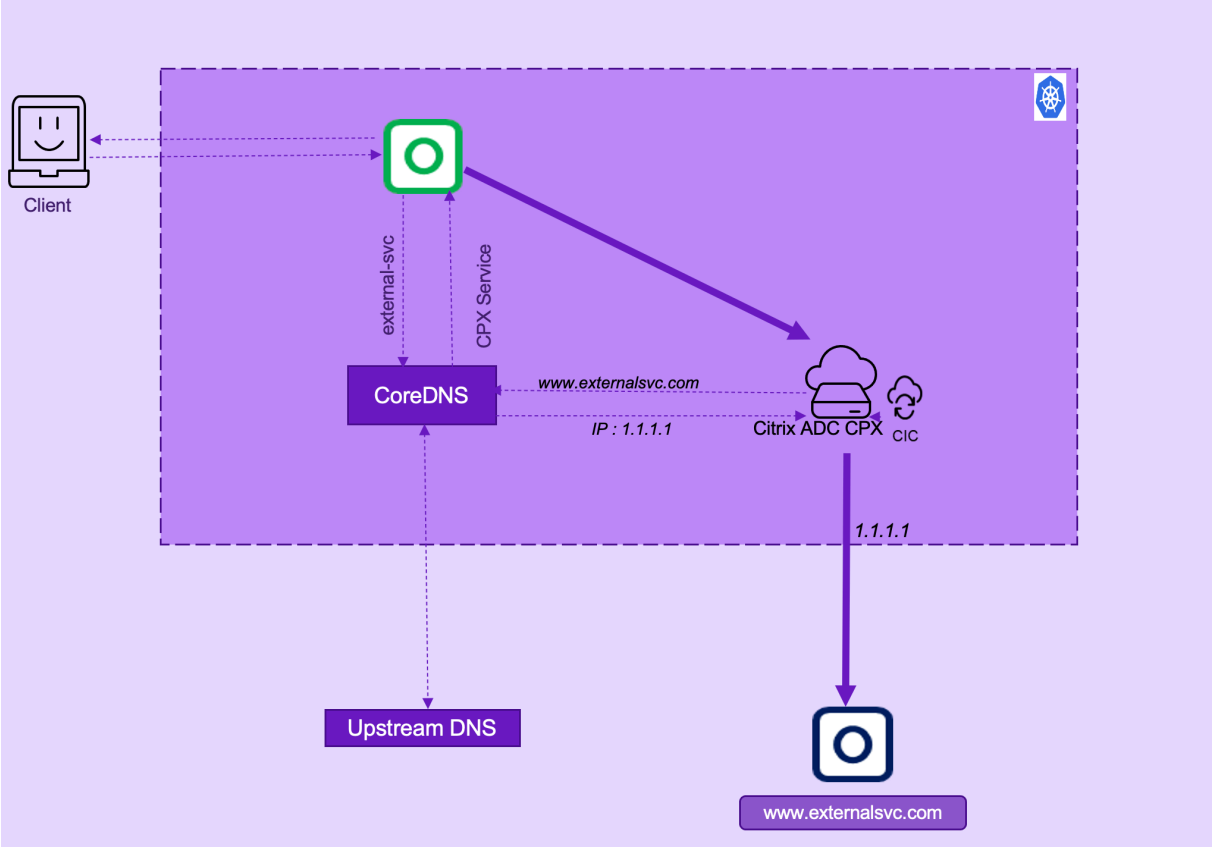
You can configure NetScaler through NetScaler Ingress Controller to create a domain based service group using the ingress annotation `ingress.citrix.com/external-service`. The value for `ingress.citrix.com/external-service` is a list of external name services with their corresponding domain names. For NetScaler VPX, name servers are configured on NetScaler using the ConfigMap.

> **Note:**
>
> ConfigMaps are used to configure name servers on NetScaler only for NetScaler VPX. For NetScaler CPX, CoreDNS forwards the name resolution request to the upstream DNS server.

Traffic management using NetScaler CPX

The following diagram explains NetScaler CPX deployment to reach external services. An Ingress is deployed where the external service annotation is specified to configure DNS on NetScaler CPX.

Note: A ConfigMap is used to configure name servers on NetScaler VPX.



In this deployment:

1. A microservice sends the DNS query for www.externalsvc.com which would get resolved to the NetScaler CPX service.
2. NetScaler CPX resolves www.externalsvc.com and reaches external service.

Following are the steps to configure NetScaler CPX to load balance external services:

1. Define a headless service to reach NetScaler.

```
apiVersion: v1
```



```
kind: Service
metadata:
  name: external-svc
spec:
  selector:
    app: cpx
  ports:
    - protocol: TCP
      port: 80
```

2. Define an ingress and specify the external-service annotation as specified in the dbs-ingress.yaml file. When you specify this annotation, NetScaler Ingress Controller creates DNS servers on NetScaler and binds the servers to the corresponding service group.

```
    annotations:
      ingress.citrix.com/external-service: '{
 "external-svc": {
 "domain": "www.externalsvc.com" }
  }
 '
```

3. Add the IP address of the DNS server on NetScaler using ConfigMap.

> **Note:**
>
> This step is applicable only for NetScaler VPX.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nameserver-cmap
  namespace: default
data:
  NS_DNS_NAMESERVER: '[]'
<!--NeedCopy-->
```

Support for external name service across namespaces

December 31, 2023

Namespaces are used to isolate resources within a Kubernetes cluster. Sometimes, services in a different namespace might have to access a service located in another namespace. In such scenarios, you can use the `ExternalName` service provided by Kubernetes. An `ExternalName` service is a special service that does not have selectors and instead uses DNS names.

In the service definition, the `externalName` field must point to the namespace and also to the service which we are trying to access on that namespace. Citrix ingress controller supports services of

type `ExternalName` when you have to access services within the cluster.

When you create the `ExternalName` service, the following criteria must be met:

- The `externalName` field in the service definition must follow the format: `svc: <name-of-the-service>.<namespace-of-the-service>.svc.cluster.local`
- The port number in the `ExternalName` service must exactly match the port number of the targeted service.

> **Note:**
>
> When the service of an application is outside the Kubernetes cluster and you have created an `ExternalName` service, you can resolve the domain name using the Traffic management for external services feature.

Sample ExternalName service

In this example, a `mysql` service is running in the default namespace and a sample `ExternalName` service is created to access the `mysql` service from the `namespace1` namespace.

The following is a sample service definition for a MySQL service running in the default namespace.

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
  namespace: default
spec:
  clusterIP: None
  ports:
  - port: 3306
    protocol: TCP
    targetPort: 3306
  selector:
    app: mysql
  type: ClusterIP

<!--NeedCopy-->
```

The following is a sample `ExternalName` service definition to access the `mysql` service from the `namespace1` namespace.

```
kind: Service
apiVersion: v1
metadata:
  name: dbservice
  namespace: namespace1
spec:
```



```
  type: ExternalName
  externalName: mysql.default.svc.cluster.local
  ports:
  - port: 3306
    protocol: TCP
    targetPort: 3306
<!--NeedCopy-->
```

In the example, the service points to the namespace where `mysql` is deployed as specified in the field `externalName: mysql.default.svc.cluster.local`. Here `mysql` is the service name and **`default`** is the namespace. You can see that the port name is also the same as the `mysql` service.

Supported platforms and deployments

December 31, 2023

This topic provides details about various Kubernetes platforms, deployment topologies, features, and CNIs supported in Cloud-Native deployments that include NetScaler and NetScaler Ingress Controller.

Kubernetes platforms

NetScaler 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

NetScaler platforms

The following table lists the NetScaler platforms supported by the NetScaler Ingress Controller:

NetScaler Platform	Versions
NetScaler MPX	11.1–61.7 and later
NetScaler VPX	11.1–61.7 and later
NetScaler CPX	12.1–51.16 and later

Supported deployment topologies on platforms (on-premises)

The following table lists the various deployment topologies supported by the NetScaler Ingress Controller on the supported Kubernetes (on-premises) platforms:

Deployment Topologies	Kubernetes	Red Hat OpenShift	PKS
Single-Tier (NetScaler MPX or VPX in tier-1)	Yes	Yes	Yes
Dual-Tier (NetScaler MPX or VPX in tier-1 and NetScaler CPXs in tier-2)	Yes	Yes	Yes
Service mesh lite	Yes	Yes	Yes
Services of type LoadBalancer	Yes	Yes	Yes
Services of type NodePort	Yes	Yes	Yes

Supported deployment topologies on cloud platforms

The following table lists the various deployment topologies supported by the NetScaler Ingress Controller on the supported cloud platforms:

Deployment Topologies	GKE	EKS	AKS (Basic mode - Kubenet)	AKS (Advanced mode - Azure CNI)
Single-Tier Cloud topology (NetScaler VPX in tier-1)	Yes	Yes	Yes	Yes
Dual-Tier Cloud topology (NetScaler VPX in tier-1 and NetScaler CPXs in tier-2)	Yes	No	Yes	Yes
Dual-Tier Cloud topology (Cloud LB in tier-1 and NetScaler CPXs in tier-2)	Yes	No	Yes	Yes

Supported NetScaler Ingress Controller feature on platforms

The following table lists the NetScaler Ingress Controller features supported on various cloud-native platforms:

NetScaler Ingress Controller features	Kubernetes	Google Cloud	AWS	Azure	Red Hat OpenShift	PKS
TCP Ingress	Yes	Yes	Yes	Yes	Yes	Yes
UDP Ingress	Yes	Yes	Yes	Yes	Yes	Yes

NetScaler Ingress Controller features	Kubernetes	Google Cloud	AWS	Azure	Red Hat OpenShift	PKS
SSL Ingress	Yes	Yes	Yes	Yes	Yes	Yes
TCP over SSL Ingress	Yes	Yes	Yes	Yes	Yes	Yes
HTTP, TCP, or SSL profiles	Yes	Yes	Yes	Yes	Yes	Yes
NodePort support	Yes	Yes	Yes	Yes	Yes	Yes
Type LoadBalancer support	Yes	No	Yes	No	Yes	Yes
Rewrite and Responder CRD	Yes	Yes	Yes	Yes	Yes	Yes
Rate limit CRD	Yes	Yes	Yes	Yes	Yes	Yes
Auth CRD	Yes	Yes	Yes	Yes	Yes	Yes
Advanced content routing	Yes	Yes	Yes	Yes	Yes	Yes
WAF CRD	Yes	Yes	Yes	Yes	Yes	Yes
Bot CRD	Yes	Yes	Yes	Yes	Yes	Yes
OpenShift Routes	N/A	N/A	N/A	N/A	Yes	N/A

NetScaler Ingress Controller features	Kubernetes	Google Cloud	AWS	Azure	Red Hat OpenShift	PKS
OpenShift router sharding	N/A	N/A	N/A	N/A	Yes	N/A
Simplified canary using Ingress	Yes	Yes	Yes	Yes	Yes	Yes

The following table lists the NetScaler Ingress Controller features supported on the respective NetScaler Ingress Controller versions and NetScaler versions:

NetScaler Ingress Controller features	NetScaler Ingress Controller versions	NetScaler MPX or VPX versions	NetScaler CPX versions
TCP Ingress	1.1.1 and later	11.1–61.7 and later	12.1–51.16 and later
UDP Ingress	1.1.1 and later	11.1–61.7 and later	12.1–51.16 and later
SSL Ingress	1.1.1 and later	11.1–61.7 and later	12.1–51.16 and later
TCP over SSL Ingress	1.1.1 and later	11.1–61.7 and later	12.1–51.16 and later
HTTP, TCP, or SSL profiles	1.4.392	11.1–61.7 and later	12.1–51.16 and later
NodePort support	1.1.1 and later	11.1–61.7 and later	12.1–51.16 and later
Type LoadBalancer support	1.2.0 and later	11.1–61.7 and later	12.1–51.16 and later
Rewrite and Responder CRD	1.1.1 and later	11.1–61.7 and later	12.1–51.16 and later
Rate limit CRD	1.4.392	11.1–61.7 and later	12.1–51.16 and later
Auth CRD	1.4.392	11.1–61.7 and later	12.1–51.16 and later
Advanced content routing	1.7.46	12.1–51.16 and later	12.1–51.16 and later
WAF CRD	1.9.2	13.0–65.4 and later	13.0–65.4 and later

NetScaler Ingress Controller features	NetScaler Ingress Controller versions	NetScaler MPX or VPX versions	NetScaler CPX versions
Bot CRD	1.11.3	NetScaler VPX version 13.0.67.39 and later	Not supported
OpenShift Routes	1.1.3 and later	12.1–51.16 and later	13.0–36.28 and later
OpenShift router sharding	1.2.0 and later	12.1–51.16 and later	13.0–36.28 and later
Simplified canary using Ingress	Version 1.13.15 and later	11.1–61.7 and later	12.1–51.16 and later

Container network interfaces (CNIs) for NetScaler CPX

The following table lists the Container network interfaces (CNIs) supported by NetScaler CPX:

Container network interfaces (CNI)	NetScaler CPX versions
Flannel	12.1–51.16 and later
Kubenet	12.1–51.16 and later
Calico	13.0–36.28
Canal	13.0–36.28
Calico on GKE	12.1–51.16 and later
OVS	13.0–36.28
Weave	12.1–51.16 and later
Cilium	13-0-71-40 and later

Supported container runtime interfaces for NetScaler CPX

The following table lists the container runtime interfaces (CRIs) supported by NetScaler CPX.

CRI	Supported versions of NetScaler CPX
Docker	11.1 and later
CRI-O	13.0–47.103 and later

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 NetScaler CPX/VPX/MPX which supports different components of the NetScaler 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.

Product/Component	NetScaler CPX/VPX/MPX	NetScaler Ingress Controller	NetScaler Connectivity Observability Exporter	Citrix ingress adaptor	Citrix node controller	ADM agent	ADM service	ADM on-prem	NetScaler Metrics Exporter	–	-	-	-	–	–	–	–
NetScaler CPX/VPX/MPX	NA	NetScaler Ingress Controller version 1.1.1 onwards is supported with CPX version 12.1+ onwards and VPX/MPX 11.1+ onwards	COE version 1.0.001 onwards is supported with VPX/MPX/CPX 13.0 onwards	CIA version 1.0-alpha onwards is supported with CPX/VPX/MPX 12.1+ onwards	CPX/VPX/MPX 12.0 onwards	CPX/VPX/MPX 13.0–47.22 onwards	CPX/VPX/MPX 13.0–47.22 onwards	CPX/VPX/MPX 11.1 onwards	CPX/VPX/MPX 12.1 onwards								

NetScaler Ingress Controller	CPX 12.1+ onwards and VPX/MPX 11.1+ onwards supports NetScaler Ingress Controller version 1.1.1 onwards	NA	COE version 1.0.001 and onwards is supported with NetScaler Ingress Controller version 1.5.6 onwards	NA	NA	NA	NA	NA	NA
NetScaler Observability Exporter	CPX/VPX/MPX 13.0 onwards is supported with COE version 1.0.001 onwards	NetScaler Ingress Controller version 1.5.6 onwards is supported with COE version 1.0.001 onwards		CIA version 1.2.0-beta onwards is supported with COE version 1.0.001 onwards	NA	NA	NA	NA	NA

Citrix ingestion adaptor	CPX/VPX/MPX 12.1+ onwards is supported with CIA version 1.2.0-beta onwards	CPX/VPX/MPX CIA version 1.0.001 is supported with CIA version 1.2.0-beta onwards	NA	NA	NA	NA	NA	NA	NA
node controller	CPX/VPX/MPX 12.0 onwards	CPX/VPX/MPX	NA	NA	NA	NA	NA	NA	NA
ADM agent	CPX/VPX/MPX 13.0–47.22 onwards	CPX/VPX/MPX	NA	NA	NA	NA	NA	NA	NA
ADM service	CPX/VPX/MPX 13.0–47.22 onwards	CPX/VPX/MPX	NA	NA	NA	NA	NA	NA	NA
ADM on-prem	CPX/VPX/MPX 11.1 onwards	CPX/VPX/MPX	NA	NA	NA	NA	NA	NA	NA
NetScaler Metrics Exporter	CPX/VPX/MPX 12.1 onwards	CPX/VPX/MPX	NA	NA	NA	NA	NA	NA	NA

> **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 NetScaler

December 31, 2023

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.

NetScaler provides a Kubernetes CustomResourceDefinition (CRD) called the **Auth CRD** that you can use with the NetScaler Ingress Controller to define authentication policies on the ingress NetScaler.

Auth CRD definition

The Auth CRD is available in the NetScaler 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 NetScaler.

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.
- 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.

Attribute	Description
`authentication_host`	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 NetScaler with Ingress/service type LoadBalancer or the VIP address of the Listener CRD.
`authentication_host_cert`	Specifies the name of the SSL certificate to be used with the `authentication_host`. This certificate is mandatory while performing authentication using the form.
`ingress_name`	Specifies the Ingress name for which the authentication using forms is applicable.
`lb_service_name`	Specifies the name of the service of type LoadBalancer for which the authentication using forms is applicable.
`listener_name`	The name of the Listener CRD for which the authentication using forms is applicable.

Attribute	Description
`vip`	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.

> **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 NetScaler.

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 NetScaler as an access credential, the NetScaler validates the token using the configured values. If the token validation is successful then NetScaler grants access to the client.

OAuth authentication attributes The following are the attributes for OAuth authentication:

Attribute	Description
`Issuer`	The identity (usually a URL) of the server whose tokens need to be accepted for authentication.

Attribute	Description
`jwks_uri`	The URL of the endpoint that contains JWKs (JSON Web Key) for JWT (JSON Web Token) verification.
`audience`	The identity of the service or application for which the token is applicable.
`token_in_hdr`	The custom header name where the token is present. The default value is the `Authorization` header. **Note:** You can specify more than one header.
`token_in_param`	The query parameter where the token is present.
`signature_algorithms`	Specifies the list of signature algorithms which are allowed. By default HS256, RS256, and RS512 algorithms are allowed.
`introspect_url`	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.
`client_credentials`	The name of the Kubernetes secrets object that contains the client id and client secret required to authenticate with the authentication server.
`claims_to_save`	The list of claims to be saved. Claims are used to create authorization policies.

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.

Attribute	Description
`metadata_url`	Specifies the URL that is used to get OAUTH or OIDC provider metadata.
`user_field`	Specifies the attribute in the token from which the user name should be extracted. By default, NetScaler examines the email attribute for user ID.

Attribute	Description
`default_group`	Specifies the group assigned to the request if authentication succeeds. This group is in addition to any extracted groups from the token.
`grant_type`	Specifies the type of flow to the token end point. The default value is `CODE`.
`pkce`	Specifies whether to enable Proof Key for Code Exchange (PKCE). The default value is `ENABLED`.
`token_ep_auth_method`	Specifies the authentication method to be used with the token end point. The default value is `client_secret_post`.

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 NetScaler. On successful validation of the client identity, the NetScaler grants access to the client.

The following are the attributes for SAML authentication.

Attribute	Description
`metadata_url`	Specifies the URL used for obtaining SAML metadata.
`metadata_refresh_interval`	Specifies the interval in minutes for fetching metadata from the specified metadata URL.
`signing_cert`	Specifies the SSL certificate to sign requests from the service provider (SP) to the identity provider (IdP).
`audience`	Specifies the identity of the service or application for which the token is applicable.
`issuer_name`	Specifies the name used in requests sent from SP to IdP to identify the NetScaler.
`binding`	Specifies the transport mechanism of the SAML message. The default value is `POST`.
`artifact_resolution_service_url`	Specifies the URL of the artifact resolution service on IdP.

Attribute	Description
`logout_binding`	Specifies the transport mechanism of the SAML logout. The default value is `POST`.
`reject_unsigned_assertion`	Rejects unsigned SAML assertions. If this value is `ON`, it rejects assertion without signature.
`user_field`	Specifies the SAML user ID specified in the SAML assertion
`default_authentication_group`	Specifies the default group that is chosen when the authentication succeeds in addition to extracted groups.
`skewtime`	Specifies the allowed clock skew time in minutes on an incoming SAML assertion.
`attributes_to_save`	Specifies the list of attribute names separated by commas which needs to be extracted and stored as key-value pairs for the session on NetScaler.

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.

Attribute	Description
`server_ip`	Specifies the IP address assigned to the LDAP server.
`server_name`	Specifies the LDAP server name as an FQDN.
`server_port`	Specifies the port on which the LDAP server accepts connections. The default value is 389.

Attribute	Description
`base`	Specifies the base node on which to start LDAP searches. If the LDAP server is running locally, the default value of base is `dc=netscaler`, `dc=com`.
`server_login_credentials`	Specifies the Kubernetes secret object providing credentials to log in to the LDAP server. The secret data should have user name and password.
`login_name`	Specifies the **LDAP login name** attribute. The NetScaler uses the LDAP login name to query external LDAP servers or Active Directories.
`security_type`	Specifies the type of security used for communications between the NetScaler and the LDAP server. The default is TLS.
`validate_server_cert`	Validates LDAP server certificates. The default value is `NO`.
`hostname`	Specifies the host name for the LDAP server. If `validate_server_cert` is `ON`, this value must be the host name on the certificate from the LDAP. A host name mismatch causes a connection failure.
`sub_attribute_name`	Specifies the LDAP group subattribute name. This attribute is used for group extraction from the LDAP server.
`group_attribute_name`	Specifies the LDAP group attribute name. This attribute is used for group extraction on the LDAP server.
`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.

Attribute	Description
`auth_timeout`	Specifies the number of seconds the NetScaler 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 NetScaler.

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:

Attribute	Description
`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. 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.
`provider`	Specifies the authentication mechanism that needs to be used. If the authentication mechanism is not provided, then authentication is not performed.

The following attributes are for authentication policies with expression format:

Attribute	Description
`expression`	Specifies NetScaler expression to be evaluated based on authentication
`provider`	Specifies the authentication mechanism that needs to be used. If the authentication mechanism is not provided, then authentication is not performed.

> **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:

Attribute	Description
`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.
`claims`	Specifies the claims required to access a specific API endpoint. `name` indicates the claim name and `values` indicate the required permissions. You can have more than one claim. If an empty list is specified, it implies that authorization is not required.

Attribute	Description
	Note: Any claim that needs to be used for authorization, should be saved as part of authentication.

The following are the attributes for authorization policies with expression format:

Attribute	Description
`expression`	Specifies an expression to be evaluated for authorization.

> **Note:**
>
> NetScaler 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 end point, 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:

```
kubectl create -f auth-crd.yaml
```

For example:

```
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 NetScaler 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 NetScaler Ingress Controller applies the authentication policy configuration on the Ingress NetScaler 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'

    authentication_policies:
        - resource:
            path:
              - '/orders/'
              - '/shipping/'
            method: [GET, POST]
          provider: ["local-auth-provider"]

        # skip authentication for this
        - resource:
            path:
              - '/products/'
            method: [GET]
          provider: []

    authorization_policies:
        # skip authorization
        - resource:
            path: []
            method: []
            claims: []
```


The sample policy definition performs the following:

- NetScaler performs the local authentication on the requests to the following:
  - **GET** or **POST** operation on orders and shipping end points.
- NetScaler does not perform the authentication for **GET** operation on the **products** endpoint.
- NetScaler 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).

```
apiVersion: citrix.com/v1beta1
kind: authpolicy
metadata:
  name: authexample
spec:
    servicenames:
    - frontend

    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"]
        - resource:
            path:
              - '/orders/'
            method: [GET]
            claims:
              - name: "scope"
                values: ["read"]
        # skip authorization, no claims required
        - resource:
            path:
              - '/shipping/'
            method: [GET]
            claims: []
```

The sample policy definition performs the following:

- NetScaler performs JWT verification on the requests to the following:
  - The **GET** or **POST** operation on **orders** and **shipping** endpoints.
- NetScaler skips authentication for the **GET** operation on the **products** endpoint.
- NetScaler requires the scope claim with `read` and `write` permissions for **POST** operation on **orders** and **shipping** endpoints.
- NetScaler requires the scope claim with the read permission for **GET** operation on the **orders** endpoint.
- NetScaler 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/
            v2.0keys"
```



```
        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)

```
apiVersion: citrix.com/v1beta1
kind: authpolicy
metadata:
  name: authexample
spec:
    servicenames:
    - frontend

    authentication_providers:
        - name: "introspect-provider"
          oauth:
            issuer: "ns-idp"
            jwks_uri: "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:

- NetScaler performs the oAuth introspection as specified in the provider `introspect-provider` for all requests.

- NetScaler requires the scope claim with `read` and `write` permissions for all **POST** requests.
- NetScaler 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:

```
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:

```
      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.

```
apiVersion: citrix.com/v1beta1
kind: authpolicy
metadata:
  name: samlexample
spec:
    servicenames:
```



```
    - frontend

    authentication_mechanism:
      using_forms:
        authentication_host: "fqdn_authenticaton_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: []

<!--NeedCopy-->
```

The sample policy definition performs the following:

- NetScaler performs SAML authentication as specified in the provider `saml-auth-provider` for all requests.
  Note: Granular authentication is not supported for the forms mechanism.
- NetScaler requires the group claim with `admin` permission for all **POST** requests.
- NetScaler 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 NetScaler 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.

```
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:

        #default - no authorization requirements
        - resource:
            path: []
            method: []
            claims: []
<!--NeedCopy-->
```

The sample policy definition performs the following:

- NetScaler 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.

- NetScaler 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.

```
apiVersion: citrix.com/v1beta1
kind: authpolicy
metadata:
  name: ldapexample
spec:
    servicenames:
    - frontend

    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

        - name: "local-auth-provider"
          basic_local_db:
              use_local_auth: 'YES'

    authentication_policies:

        - resource:
            path: []
            method: []
          provider: ["ldap-auth-provider"]


    authorization_policies:

        - resource:
            path: []
            method: []
            claims: []
<!--NeedCopy-->
```

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:

```
    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.

```
apiVersion: citrix.com/v1beta1
kind: authpolicy
metadata:
  name: ldapexample
spec:
    servicenames:
    - frontend

    authentication_mechanism:
      using_forms:
        authentication_host: "fqdn_authenticaton_host"
        authentication_host_cert:
          tls_secret: authhost-tls-cert-secret
        vip: "192.2.156.156"

    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

    authentication_policies:

        - resource:
            path: []
            method: []
          provider: ["ldap-auth-provider"]

    authorization_policies:

        - resource:
            path: []
            method: []
            claims: []

<!--NeedCopy-->
```

The sample policy definition performs the following:

- NetScaler performs the LDAP authentication for entire traffic (all requests).
- NetScaler does not apply any authorization permission.

The following is an example for `LDAP_secret.yaml`.

```
apiVersion: v1
kind: Secret
metadata:
  name: ldapcredential
type: Opaque
stringData:
  username: 'ldap_server_username'
  password: 'ldap_server_password'

<!--NeedCopy-->
```

Example for NetScaler expression support with Auth CRD

This example shows how you can specify NetScaler expressions along with authentication and authorization policies:

```
  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"]

```



```
28
29        authorization_policies:
30            # ALLOW the session only if the authenticated user is
                associated with attribute "memberof" having value "grp4"
31            - expression: 'aaa.user.attribute("memberof").contains("grp4
                ")'
```

Rate limiting in Kubernetes using NetScaler

December 31, 2023

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 NetScaler.

NetScaler provides a Kubernetes CustomResourceDefinitions (CRDs) called the **Rate limit CRD** that you can use with the NetScaler Ingress Controller to configure the rate limiting configurations on the NetScalers 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 NetScaler 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 NetScaler 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 NetScaler that acts as an API gateway.

Rate limit CRD attributes

The following table lists the various attributes provided in the Rate limit CRD:

CRD attribute	Description
`servicename`	The list of Kubernetes services to which you want to apply the rate limit policies.
`selector_keys`	The traffic selector keys that filter the traffic to identify the API requests against which the throttling is applied and monitored. **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. **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.
`req_threshold`	The maximum number of requests that are allowed in the given time slice (request rate).
`timeslice`	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.
`limittype`	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 ***burst*** mode.
`throttle_action`	It allows you to define the throttle action that needs to be taken on the traffic that is throttled for crossing the configured threshold.

CRD attribute	Description
	DROP: Drops the requests above the configured traffic limits. **RESET:** Resets the connection for the requests crossing the configured limit. **RESPOND:** Responds with the standard "***429 Too many requests***"response.
`redirect_url`	This attribute is an optional attribute that is required only if `throttle_action` is configured with the value `REDIRECT`.
`logpackets`	Enables audit logs.
`logexpression`	Specifies the default-syntax expression that defines the format and content of the log message.
`loglevel`	Specifies the severity level of the log message that is generated.

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:

```
kubectl create -f ratelimit-crd.yaml
```

For example,

```
root@master:~# kubectl create -f ratelimit-crd.yaml

customresourcedefinition.apiextensions.k8s.io/ratelimits.citrix.com created

root@master:~# kubectl get crd

NAME CREATED AT
ratelimits.citrix.com 2019-08-27T01:06:30Z
```


How to write a rate-based policy configuration

After you have deployed the CRD provided by NetScaler 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 NetScaler Ingress Controller applies the rate-based policy configuration on the Ingress NetScaler 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 NetScaler 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:

```
apiVersion: citrix.com/v1beta1
kind: ratelimit
metadata:
  name: throttle-req-per-clientip
spec:
  servicenames:
    - frontend
  selector_keys:
   basic:
    path:
     - "/api/v1/products"
     - "/api/v1/orders/"
    per_client_ip: true
  req_threshold: 15
  timeslice: 60000
  throttle_action: "RESPOND"
  logpackets:
    logexpression: "http.req.url"
    loglevel: "INFORMATIONAL"
<!--NeedCopy-->
```

> **Note:**
>
> You can initiate multiple Kubernetes objects for different paths that require different rate limit configurations.

After you have defined the policy configuration, deploy the `.yaml` file using the following command:


```
root@master:~#kubectl create -f ratelimit-example1.yaml
ratelimit.citrix.com/throttle-req-per-clientip created
```

The NetScaler Ingress Controller applies the policy configuration on the Ingress NetScaler device.

Limit API requests to calender APIs

Consider a scenario wherein you want to define a rate-based policy in a NetScaler 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 calender APIs. Create a `.yaml` file called `ratelimit-example2.yaml` and use the appropriate CRD attributes to define the rate-based policy as follows:

```
apiVersion: citrix.com/v1beta1
kind: ratelimit
metadata:
  name: throttle-calendarapi-perapikey
spec:
  servicenames:
    - frontend
  selector_keys:
    basic:
      path:
        - "/api/v1/calender"
      method:
        - "GET"
        - "POST"
      header_name: "X-API-Key"
  req_threshold: 5
  throttle_action: "RESPOND"
  logpackets:
    logexpression: "rate exceeded, you may want to configure higher limit"
    loglevel: "INFORMATIONAL"
<!--NeedCopy-->
```

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 NetScaler Ingress Controller applies the policy configuration on the Ingress NetScaler device.

Use Rewrite and Responder policies in Kubernetes

February 29, 2024

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 NetScaler device to deploy these policies.

NetScaler provides Kubernetes CustomResourceDefinitions (CRDs) that you can use with the NetScaler Ingress Controller to automate the configurations and deployment of these policies on the NetScalers used as Ingress devices.

The Rewrite and Responder CRD provided by NetScaler is designed to expose a set of tools used in front-line NetScalers. 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 NetScaler 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 NetScaler Rewrite and Responder CRD

The NetScaler 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:

```
kubectl create -f rewrite-responder-policies-deployment.yaml
```

For example,

```
root@master:~# kubectl create -f rewrite-responder-policies-deployment.
    yaml
customresourcedefinition.apiextensions.k8s.io/rewritepolicies.citrix.
    com created
```


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 **NetScaler** 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 NetScaler command and attributes.

CRD attribute	NetScaler command	NetScaler attribute
rewrite-criteria	Add rewrite policy	rule
default-action	Add rewrite policy	undefAction
operation	Add rewrite action	type
target	Add rewrite action	target
modify-expression	Add rewrite action	stringBuilderExpr
multiple-occurence-modify	Add rewrite action	Search
additional-multiple-occurence-modify	Add rewrite action	RefineSearch
Direction	Bind lb vserver	Type

Responder policy

The following table lists the **CRD** attributes that you can use to define a responder policy. Also, the table lists the corresponding NetScaler command and attributes.

CRD attribute	NetScaler command	NetScaler attribute
Redirect	Add responder action	Type (the value of type)
url	Add responder action	Target
redirect-status-code	Add responder action	responseStatusCode
redirect-reason	Add responder action	reasonPhrase
Respond-with	Add responder action	Type (the value of type)

CRD attribute	NetScaler command	NetScaler attribute
http-payload-string	Add responder action	Target
Noop	Add responder policy	Action (the value of action)
Reset	Add responder policy	Action (the value of action)
Drop	Add responder policy	Action (the value of action)
Respond-criteria	Add responder policy	Rule
Default-action	Add responder policy	undefAction

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 NetScaler command and attributes.

CRD attribute	NetScaler command	NetScaler attribute
Logexpression	Add audit message action	stringBuilderExpr
Loglevel	Add audit message action	Loglevel

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 NetScaler command and attributes.

CRD attribute	NetScaler command	NetScaler attribute
Name	Add policy dataset	Name
Type	Add policy dataset	Type
Values	Bind policy dataset	Value

Patset

CRD attribute	NetScaler command	NetScaler attribute
Name	Add policy patset	Name
Values	Bind policy patset	string

String map

CRD attribute	NetScaler command	NetScaler attribute
Name	Add policy stringmap	Name
Key	Bind policy stringmap	Key
Value	Bind policy stringmap	Value

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.

CRD attribute	NetScaler command	NetScaler attribute	Supported values	Default value
goto-priroty-expression	Bind lb vserver	gotoPriorityExpression	NEXT and END	End

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 NetScaler 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 NetScaler Ingress Controller applies the policy configuration on the Ingress NetScaler 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.
- 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 NetScaler 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:

```
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
  name: targeturlrewrite
spec:
  rewrite-policies:
    - servicenames:
        - citrix-svc
      logpackets:
        logexpression: "http.req.url"
        loglevel: INFORMATIONAL
      rewrite-policy:
        operation: replace
        target: 'http.req.url'
        modify-expression: '"new-url-for-the-application"'
        comment: 'Target URL Rewrite - rewrite the url of the HTTP
            request'
        direction: REQUEST
        rewrite-criteria: 'http.req.is_valid'
<!--NeedCopy-->
```

After you have defined the policy configuration, deploy the `.yaml` file using the following command:

```
kubectl create -f target-url-rewrite.yaml
```

After you deploy the `.yaml` file, the NetScaler Ingress Controller applies the policy configuration on the Ingress NetScaler device.

On the master node in the Kubernetes cluster, you can verify the status of the applied rewrite policy CRD using the following command:


```
Kubectl get rewritepolicies.citrix.com targeturlrewrite
```

You can view the status as follows:

```
kubectl get rewritepolicies.citrix.com targeturlrewrite
NAME                STATUS    MESSAGE
targeturlrewrite    Success   CRD Activated
```

If there are issues while creating or applying the CRD, the same can be debugged using the citrix-k8s-ingress-controller logs.

```
kubectl logs citrixingresscontroller
```

Also, you can verify whether the configuration is applied on the NetScaler by using the following steps.

1. Log on to the NetScaler command-line.
2. Use the following command to verify if the configuration is applied to the NetScaler:

   ```
   show run | grep `lb vserver`
   add lb vserver k8s-citrix_default_80_k8s-citrix-svc_default_80_svc
       HTTP 0.0.0.0 0 -persistenceType NONE -cltTimeout 180
   bind lb vserver k8s-citrix_default_80_k8s-citrix-
      svc_default_80_svc k8s-citrix_default_80_k8s-citrix-
      svc_default_80_svc
   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_defau` 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`)

```
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
  name: blocklisturls
spec:
  responder-policies:
    - servicenames:
        - citrix-svc
      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'
14
15
16    patset:
17      - name: blocklistUrls
18        values:
19          - '/app1'
20          - '/app2'
21          - '/app3'
22  <!--NeedCopy-->
```

In this example, if NetScaler receives any URL that matches the `/app1`, `/app2`, or `/app3` strings defined in the `patset`, NetScaler blocks the URL.

Policy with audit logs enabled

Following is a sample policy with audit logs enabled (block-list-urls-audit-log.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         logpackets:
10          logexpression: "http.req.url"
11          loglevel: INFORMATIONAL
12        responder-policy:
13          respondwith:
14            http-payload-string: '"HTTP/1.1 401 Access denied"'
15          respond-criteria: 'http.req.url.equals_any("blocklistUrls")'
16          comment: 'Blocklist certain Urls'
17
18
19    patset:
20      - name: blocklistUrls
21        values:
22          - '/app1'
23          - '/app2'
24          - '/app3'
25  <!--NeedCopy-->
```


Multiple policy configurations

You can add multiple policy configurations in a single `.yaml` file and apply the policies to the NetScaler 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
      values:
        - 10.1.1.100
        - 1.1.1.1 - 1.1.1.100
        - 2.2.2.2/10

  stringmap:
    - name: modifyurls
```



```
      comment: Urls to be modified string
      values:
        - key: '"/app1/"'
          value: '"/internal-app1/"'
        - key: '"/app2/"'
          value: '"/internal-app2/"'
<!--NeedCopy-->
```

The example contains two responder policies and a rewrite policy, based on these policies NetScaler performs the following:

- Any requests that match the client IP addresses specified in the `redirectIPs` dataset, that is, `10.1.1.100`, IP addresses in the range `1.1.1.1 - 1.1.1.100` and IP addresses in the subnet `2.2.2.2/10`, 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:

```
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
name: addresponseheaders
spec:
rewrite-policies:
  - servicenames:
      - frontend
    rewrite-policy:
      operation: insert_before_all
      target: http.res.full_header
      modify-expression: '"\r\nx-port: "+client.tcp.srcport+"\r\nx-ip
          :"+client.ip.dst+"\r\nx-new-dummy-header: Sending_a_gift"'
```



```
      multiple-occurence-modify: 'text("\r\n\r\n")'
      comment: 'Response header rewrite'
      direction: RESPONSE
      rewrite-criteria: 'http.req.url.contains("/citrix-app/")'
<!--NeedCopy-->
```

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
<
<html>
<head>
<title> Front End App - v1 </title>


TRIMMED
.......
```

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:


```
apiVersion: citrix.com/v1
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-occurence-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:

```
$ 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
< x-request-time:Fri, 29 Mar 2019 13:27:40 GMT =============> NEW
    HEADER ADDED
< x-using-citrix-ingress-controller: true  ===============> NEW HEADER
    ADDED
<
<html>
<head>
<title> Front End App - v1 </title>
<style>

```



```
TRIMMED
........
```

Replace host name in the request

You can define a rewrite policy as shown in the following example YAML (`http_request_modify_prefixaspr.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:
  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:

```
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.

```
curl http://app.cic-citrix.org/prefix/foo/bar
```

Output:

```
▼ Hypertext Transfer Protocol
  ▼ GET /prefix/foo/bar HTTP/1.1\r\n
    ▶ [Expert Info (Chat/Sequence): GET /prefix/foo/bar HTTP/1.1\r\n]
      Request Method: GET
      Request URI: /prefix/foo/bar
      Request Version: HTTP/1.1
    Host: citrix-service-app\r\n
    User-Agent: curl/7.54.0\r\n
    Accept: */*\r\n
    \r\n
    [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:

```
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
 name: httpapprootrequestmodify
spec:
 rewrite-policies:
   - servicenames:
       - frontend
     rewrite-policy:
       operation: replace
       target: http.req.url
       modify-expression: '"/citrix-approot/"'
       comment: 'HTTP app root request modify'
       direction: REQUEST
       rewrite-criteria: http.req.url.eq("/")
<!--NeedCopy-->
```

Create a YAML file (`http_approot_request_modify.yaml`) with the rewrite policy definition and deploy the YAML file using the following command:

kubectl create -f http_approot_request_modify.yaml

Using the `curl` command, you can verify if the application root is modified as per your requirement:

```
curl -vvv http://app.cic-citrix.org/
```

Output:

```
▼ Hypertext Transfer Protocol
  ▶ GET /citrix-approot/ HTTP/1.1\r\n
    Host: app.cic-citrix.org\r\n
    User-Agent: curl/7.54.0\r\n
    Accept: */*\r\n
    \r\n
    [Full request URI: http://app.cic-citrix.org/citrix-approot/]
    [HTTP request 1/1]
    [Response in frame: 126]
```

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:


```
apiVersion: citrix.com/v1
kind: rewritepolicy
metadata:
name: httpurlreplacestring
spec:
rewrite-policies:
 - servicenames:
     - frontend
   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
<!--NeedCopy-->
```

Create a YAML file (`http_url_replace_string.yaml`) with the rewrite policy definition and deploy the YAML using the following command:

```
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:

```
curl http://app.cic-citrix.org/something/simple/citrix
```

Output:

```
▼ Hypertext Transfer Protocol
  ▶ GET /simple/citrix HTTP/1.1\r\n
    Host: app.cic-citrix.org\r\n
    User-Agent: curl/7.54.0\r\n
    Accept: */*\r\n
    \r\n
    [Full request URI: http://app.cic-citrix.org/simple/citrix]
    [HTTP request 1/1]
    [Response in frame: 17]
```

Example 2:

```
curl http://app.cic-citrix.org/something
```

Or,

```
curl http://app.cic-citrix.org/something/
```


Output:

```
▼ Hypertext Transfer Protocol
  ▶ GET / HTTP/1.1\r\n
    Host: app.cic-citrix.org\r\n
    User-Agent: curl/7.54.0\r\n
    Accept: */*\r\n
    \r\n
    [Full request URI: 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
kind: 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'

    - servicenames:
        - frontend
      rewrite-policy:
        operation: replace
        target: HTTP.REQ.HEADER("X-Forwarded-For")
        modify-expression: 'HTTP.REQ.HEADER("X-Forwarded-For").APPEND(",").APPEND(CLIENT.IP.SRC)'
        comment: 'HTTP Append X-Forwarded-For IPs'
        direction: REQUEST
        rewrite-criteria: 'HTTP.REQ.HEADER("X-Forwarded-For").EXISTS'
<!--NeedCopy-->
```

Create a YAML file (`http_x_forwarded_for_insert.yaml`) with the rewrite policy definition and deploy the YAML file using the following command:

```
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:

```
▼ Hypertext Transfer Protocol
  ▶ GET / HTTP/1.1\r\n
    Host: app.cic-citrix.org\r\n
    User-Agent: curl/7.54.0\r\n
    Accept: */*\r\n
    X-Forwarded-For: 10.150.16.22\r\n
    \r\n
    [Full request URI: http://app.cic-citrix.org/]
    [HTTP request 1/1]
    [Response in frame: 7]
```

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:

```
▼ Hypertext Transfer Protocol
  ▶ GET / HTTP/1.1\r\n
    Host: app.cic-citrix.org\r\n
    User-Agent: curl/7.54.0\r\n
    Accept: */*\r\n
    X-Forwarded-For: 1.1.1.1,10.150.16.22\r\n
    \r\n
    [Full request URI: http://app.cic-citrix.org/]
    [HTTP request 1/1]
    [Response in frame: 65]
```

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_redirec.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'

<!--NeedCopy-->
```

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 -vvv  http://app.cic-citrix.org
* Rebuilt URL to: 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/1.1 302 Found : Moved Temporarily
< Location: https://app.cic-citrix.org:443/   =======> Redirected to
    HTTPS
< Connection: close
< Cache-Control: no-cache
< Pragma: no-cache
<
* Closing connection 0
```

Example 2:

```
$ curl -vvv  http://app.cic-citrix.org/simple
*   Trying 10.102.33.176...
* TCP_NODELAY set
* Connected to app.cic-citrix.org (10.102.33.176) port 80 (#0)
> GET /simple HTTP/1.1
> Host: app.cic-citrix.org
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 302 Found : Moved Temporarily
< Location: https://app.cic-citrix.org:443/simple     ========>
    Redirected to HTTPS
< Connection: close
< Cache-Control: no-cache
< Pragma: no-cache
<
```



```
* Closing connection 0
```

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
```



```
28            rewrite-criteria: 'http.req.is_valid.and(HTTP.REQ.HOSTNAME.EQ
                ("www.citrite.org"))'`
29  <!--NeedCopy-->
```

Verification You can verify whether the following curl request `http://www.citrite.org/something/simple/citrix` is modified to `http://app.cic-citrix.org/simple/citrix`.

Example: Modifying the requested URL

```
1  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 NetScaler 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 NetScaler. For more information, see the HTTP callout documentation.

Related articles

- Feature Documentation
  - NetScaler Rewrite Feature Documentation
  - NetScaler Responder Feature Documentation
- Developer Documentation
  - NetScaler Rewrite Policy
  - NetScaler Rewrite Action
  - NetScaler Responder Policy

- NetScaler Responder Action
- NetScaler Audit Message Action
- NetScaler Policy Dataset

Advanced content routing for Kubernetes with NetScaler

December 31, 2023

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 NetScaler ADC as a custom resource definition (CRD) API for Kubernetes.

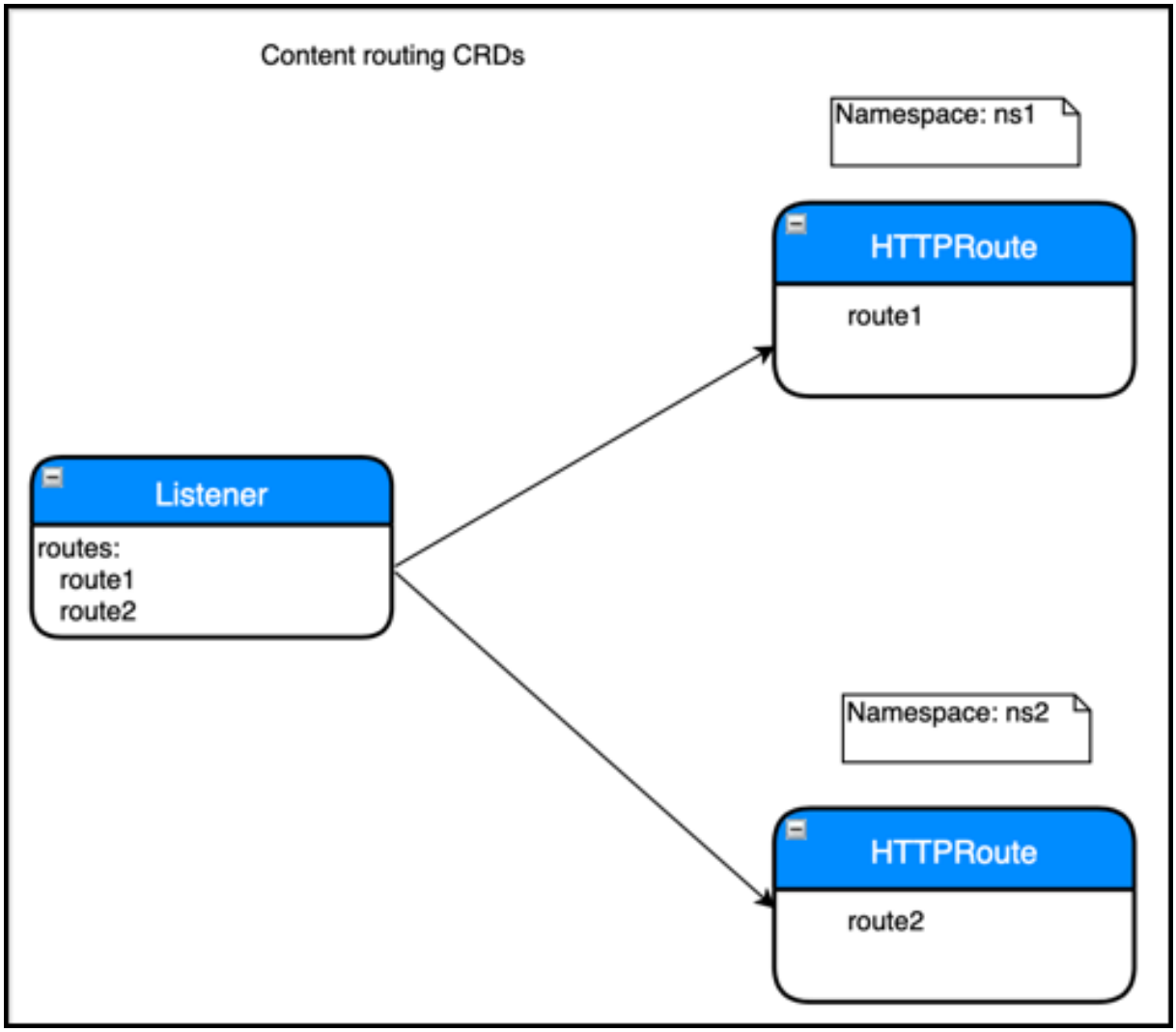
Using content routing CRDs, you can route the traffic based on the following parameters:

- Hostname
- URL path
- HTTP headers
- Cookie
- Query parameters
- HTTP method
- NetScaler 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 NetScaler 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.

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 NetScaler 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 NetScaler Ingress Controller applies the listener configuration on the Ingress NetScaler device.

Following is a sample Listener CRD object definition named as `Listener-crd.yaml`.

```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"
```



```
24    - "1.1.1.1"
25    policies:
26      httpprofile:
27        config:
28          websocket: "ENABLED"
29      tcpprofile:
30        config:
31          sack: "ENABLED"
32      sslprofile:
33        config:
34          ssl3: "ENABLED"
35      sslciphers:
36      - SECURE
37      - MEDIUM
38      analyticsprofile:
39        config:
40        - type: webinsight
41          parameters:
42             allhttpheaders: "ENABLED"
43      csvserverConfig:
44        rhistate: 'ACTIVE'
45    routes:
46      # Attach the policies from the below Routes
47    - name: domain1-route
48      namespace: default
49    - name: domain2-route
50      namespace: default
51    - labelSelector:
52        # Attach all HTTPRoutes with label route=my-route
53        route: my-route
54    # Default action when traffic matches none of the policies in the
        HTTPRoute
55    defaultAction:
56      backend:
57        kube:
58          namespace: default
59          port: 80
60          service: default-service
61          backendConfig:
62            lbConfig:
63              # Use round robin LB method for default service
64              lbmethod: ROUNDROBIN
65            servicegroupConfig:
66              # Client timeout of 20 seconds
67              clttimeout: "20"
68  <!--NeedCopy-->
```

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.

Now, NetScaler 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.

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`.

```
apiVersion: citrix.com/v1
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
            lbConfig:
              lbmethod: ROUNDROBIN
  - name: path-routing
    match:
    - path:
        prefix: /
    action:
      backend:
        kube:
          service: default-app
          port: 80
<!--NeedCopy-->
```

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.

```
Kubectl create -f  Route-crd.yaml
```

Once you deploy the YAML file, the NetScaler Ingress Controller applies the HTTP route configuration on the Ingress NetScaler 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.

```
routes:
 - name: route-1
   namespace: default
 - name: route-2
   namespace: default
```

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 NetScaler. 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:

```
routes:
- labelSelector:
    team: team1
```


Configure web application firewall policies with the NetScaler Ingress Controller

December 31, 2023

NetScaler 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 NetScaler Ingress Controller on the NetScaler VPX, MPX, SDX, and CPX. The WAF CRD enables communication between the NetScaler Ingress Controller and NetScaler 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.

WAF CRD definition

The WAF CRD is available in the NetScaler 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 NetScaler.

WAF CRD attributes

The following table lists the various attributes provided in the WAF CRD:

CRD attribute	Description
`commonchecks`	Specifies a list of common security checks, which are applied irrespective of the content type.
`block_urls`	Protects URLs.
`buffer_overflow`	Protects buffer overflow.
`content_type`	Protects content type.
`htmlchecks`	Specifies a list of security checks to be applied for HTML content types.
`cross_site_scripting`	Prevents cross site scripting attacks.
`sql_injection`	Prevents SQL injection attacks.
`form_field_consistency`	Prevents form tampering.
`csrf`	Prevents cross side request forgery (CSRF) attacks.

CRD attribute	Description
`cookie_consistency`	Prevents cookie tampering or session takeover.
`field_format`	Validates the form submission.
`fileupload_type`	Prevents malicious file uploads.
`jsonchecks`	Specifies security checks for JSON content types.
`xmlchecks`	Specifies security checks for XML content types.
`wsi`	Protects web services interoperability.
`redirect_url`	Redirects URL when block is enabled on protection.
`servicenames`	Specifies the services to which the WAF policies are applied.
`application_type`	Protects application types.
`signatures`	Specifies the location of the external signature file.
`html_error_object`	Specifies the location of the customized error page to respond when HTML or common violations are attempted.
`xml_error_object`	Specifies the location of the customized error page to respond when XML violations are attempted.
`json_error_object`	Specifies the location of the customized error page to respond when JSON violations are attempted.
`ip_reputation`	Enables the IP reputation feature.
`target`	Determines the traffic to be inspected by the WAF. If you do not specify the traffic targeted, all traffic is inspected by default.
`paths`	Specifies the list of HTTP URLs to be inspected.
`method`	Specifies the list of HTTP methods to be inspected.
`header`	Specifies the list of HTTP headers to be inspected.

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:

```
kubectl create -f waf-crd.yaml
```

For example,

```
root@master:~# kubectl create -f waf-crd.yaml
customresourcedefinition.apiextensions.k8s.io/wafpolicies.citrix.
    com created
<!--NeedCopy-->
```

How to write a WAF configuration

After you have deployed the WAF CRD provided by NetScaler 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 NetScaler Ingress Controller applies the WAF configuration on the Ingress NetScaler device.

The following are some examples for writing web appliction 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 NetScaler 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:

```
apiVersion: citrix.com/v1
kind: waf
metadata:
    name: wafhtmlxsssql
spec:
    servicenames:
        - frontend
    application_type: HTML
    html_page_url: "http://x.x.x.x/crd/error_page.html"
    security_checks:
        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:

```
1  apiVersion: citrix.com/v1
2  kind: waf
3  metadata:
4      name: wafcontenttype
5  spec:
6      servicenames:
7          - frontend
8      application_type: HTML
9      html_error_object: "http://x.x.x.x/crd/error_page.html"
10     security_checks:
11         common:
12           content_type: "on"
13     relaxations:
14         common:
15           content_type:
16             types:
17                 - custom_cnt_type
18                 - image/crd
19
20 <!--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*.

```
1  apiVersion: citrix.com/v1
2  kind: waf
3  metadata:
4      name: wafhtmlsigxsssql
5  spec:
6      servicenames:
7          - frontend
8      application_type: HTML
9      signatures: "http://x.x.x.x/crd/sig.xml"
10     html_error_object: "http://x.x.x.x/crd/error_page.html"
11     security_checks:
```



```
        html:
          cross_site_scripting: "on"
          sql_injection: "on"

<!--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.

```
apiVersion: citrix.com/v1
kind: 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-->
```

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.

```
apiVersion: citrix.com/v1
kind: waf
metadata:
    name: wafurlchecks
spec:
    servicenames:
        - frontend
    application_type: HTML
    html_error_object: "http://x.x.x.x/crd/error_page.html"
    target:
```



```
        path:
            - /
    security_checks:
        common:
          allow_url: "on"
          block_url: "on"
    relaxations:
        common:
          allow_url:
            urls:
                - payment.php
                - cover.php
    enforcements:
        common:
          block_url:
            urls:
                - "^[^?]*(passwd|passwords?)([.][^/?]*)?([?].*)?$"
                - "^[^?]*(htaccess|access_log)([.][^/?]*)?([~])?([?].*)
                  ?$"
<!--NeedCopy-->
```

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.

```
apiVersion: citrix.com/v1
kind: waf
metadata:
    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"
            secure_logging: "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.

```
apiVersion: citrix.com/v1
kind: waf
metadata:
    name: wafforms
spec:
    servicenames:
        - frontend
    application_type: HTML
    html_error_object: "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"

<!--NeedCopy-->
```

Protect forms and headers

The following is an example of a WAF CRD configuration for protecting both forms and headers.

```
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:
        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
            max_url_len: 1024
    ip_reputation: on


<!--NeedCopy-->
```

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.

```
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"
```



```
24          xml:
25            dos: "on"
26            wsi: "on"
27            attachment: "on"
28            format: "on"
29      relaxations:
30          common:
31            allow_url:
32              urls:
33                  - "^[^?]+[.](html?|shtml|js|gif|jpg|jpeg|png|swf|pif|
                      pdf|css|csv)$"
34                  - "^[^?]+[.](cgi|aspx?|jsp|php|pl)([?].*)?$"
35
36  <!--NeedCopy-->
```

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.

```
1  apiVersion: citrix.com/v1
2  kind: waf
3  metadata:
4      name: wafadvanced
5  spec:
6      servicenames:
7          - frontend
8      security_checks:
9          common:
10           allow_url: "on"
11           block_url: "on"
12           buffer_overflow: "on"
13           content_type: "on"
14           cookie_consistency: "on"
15           multiple_headers:
16             action: ["log"]
17         html:
18           cross_site_scripting: "on"
19           field_format: "on"
20           sql_injection: "on"
21           form_field_consistency: "on"
22           csrf: "on"
23           fileupload_type: "on"
24         json:
25           dos: "on"
26           sql_injection: "on"
```



```
27            cross_site_scripting: "on"
28          xml:
29            dos: "on"
30            wsi: "on"
31            validation: "on"
32            attachment: "on"
33            format: "on"
34      settings:
35          common:
36            allow_url:
37              closure: "on"
38  <!--NeedCopy-->
```

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.

```
1  apiVersion: citrix.com/v1
2  kind: waf
3  metadata:
4      name: wafiprep
5  spec:
6      application_type: html
7      servicenames:
8          - frontend
9      ip_reputation: "on"
10
11 <!--NeedCopy-->
```

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.

```
1  apiVersion: citrix.com/v1
2  kind: waf
3  metadata:
4      name: wafiprepcategory
5  spec:
6      application_type: html
7      servicenames:
8          - frontend
9      ip_reputation:
10         action: block
11         threat-categories:
12             - SPAM_SOURCES
13             - WINDOWS_EXPLOITS
14             - WEB_ATTACKS
```



```
            - BOTNETS
            - SCANNERS
            - DOS
            - REPUTATION
            - PHISHING
            - PROXY
            - NETWORK
            - CLOUD_PROVIDERS
            - MOBILE_THREATS

<!--NeedCopy-->
```

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.

```
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
            document:
              max_len: 20000000
            array:
              max_len: 5
            key:
              max_count: 10000
              max_len: 12
            string:
              max_len: 1000000


<!--NeedCopy-->
```

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.

```
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
            array:
              max_len: 10000
            key:
              max_count: 10000
              max_len: 128
            string:
              max_len: 1000000

<!--NeedCopy-->
```

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.

```
apiVersion: citrix.com/v1
kind: waf
metadata:
    name: wafxmldos
spec:
    servicenames:
        - frontend
    application_type: XML
    xml_error_object: "http://x.x.x.x/crd/error_page.xml"
    security_checks:
        xml:
          dos: "on"
```



```
13      settings:
14          xml:
15            dos:
16              attribute:
17                  max_attributes: 1024
18                  max_name_len: 128
19                  max_value_len: 128
20              element:
21                  max_elements: 1024
22                  max_children: 128
23                  max_depth: 128
24              file:
25                  max_size: 2123
26                  min_size: 9
27              entity:
28                  max_expansions: 512
29                  max_expansions_depth: 9
30              namespace:
31                  max_namespaces: 16
32                  max_uri_len: 256
33              soaparray:
34                  max_size: 1111
35              cdata:
36                  max_size: 65
37
38  <!--NeedCopy-->
```

Protect 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)

```
1  apiVersion: citrix.com/v1
2  kind: waf
3  metadata:
4      name: wafxml
5  spec:
6      servicenames:
7          - frontend
8      application_type: XML
9      xml_error_object: "http://x.x.x.x/crd/error_page.json"
10     security_checks:
```



```
11          xml:
12            dos: "on"
13            sql_injection: "on"
14            cross_site_scripting: "off"
15            wsi:
16              action: ["block"]
17            validation: "on"
18            attachment: "on"
19            format:
20              action: ["block"]
21      settings:
22          xml:
23            dos:
24              attribute:
25                  max_attributes: 1024
26                  max_name_len: 128
27                  max_value_len: 128
28              element:
29                  max_elements: 1024
30                  max_children: 128
31                  max_depth: 128
32              file:
33                  max_size: 2123
34                  min_size: 9
35              entity:
36                  max_expansions: 512
37                  max_expansions_depth: 9
38              namespace:
39                  max_namespaces: 16
40                  max_uri_len: 256
41              soaparray:
42                  max_size: 1111
43              cdata:
44                  max_size: 65
45            wsi:
46              checks: ["R1000","R1003"]
47            validation:
48              soap_envelope: "on"
49              validate_response: "on"
50            attachment:
51              url:
52                  max_size: 1111
53              content_type:
54                  value: "crd_test"
55
56  <!--NeedCopy-->
```


Configure bot management policies with the NetScaler Ingress Controller

December 31, 2023

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 NetScaler, you can configure the bot management policies with the NetScaler Ingress Controller on the NetScaler VPX. The Bot CRD enables communication between the NetScaler Ingress Controller and NetScaler 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 `.yaml` file to define the bot policy.

Bot CRD definition

The Bot CRD is available in the NetScaler 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 NetScaler.

Bot CRD attributes

The following table lists the various attributes provided in the Bot CRD:

CRD attribute	Description
security_checks	List of security checks to be applied for incoming traffic.
allow_list	List of allowed IP, subnet, and policy expressions.
block_list	List of disallowed IP, subnet, and policy expressions.
device_fingerprint	Inserts javascript and collects the client browser and device parameters.
trap	Inserts hidden URLs in the response.
tps	Prevents bots which cause unusual spike in requests based on configured percentage change in transactions.
reputation	Prevents access to bad IPs based on configured reputation categories.
ratelimit	Prevents bots based on rate limit.
redirect_url	Redirect URL when block is enabled on protection.
servicenames	Name of the services to which the bot policies are applied.
signatures	Location of external bot signature file.
target	Determines which traffic to be inspected by the bot. If you do not specify the traffic targeted, every traffic is inspected by default.
paths	List of HTTP URLs to be inspected.
method	List of HTTP methods to be inspected.
header	List of HTTP headers to be inspected.

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
customresourcedefinition.apiextensions.k8s.io/bots.citrix.com created
<!--NeedCopy-->
```

How to write a Bot configuration

After you have deployed the Bot CRD provided by NetScaler 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 NetScaler Ingress Controller applies the bot configuration on the Ingress NetScaler 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 NetScaler to enable bot for blocking malicious traffic, you can create a YAML file called `botblocklist.yaml` and use the appropriate CRD attributes to define the bot policy as follows:

```
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:
                value:  http.req.url.contains("/robots.txt")
                action:
                    - "reset"
```



```
                        - "log"
<!--NeedCopy-->
```

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:

```
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

NetScaler 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:

```
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 NetScaler 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:

```
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:

```
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:

```
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-->
```

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

`botiprepcategory.yaml` and use the appropriate CRD attributes to define the bot policy as follows:

```
apiVersion: citrix.com/v1
kind: bot
metadata:
    name: botiprepcategory
spec:
    servicenames:
        - frontend
    redirect_url: "/error_page.html"
    security_checks:
       reputation: "ON"
    bindings:
      reputation:
        categories:
            - SPAM_SOURCES:
                action:
                    - "log"
                    - "redirect"
            - MOBILE_THREATS
            - SPAM_SOURCES
<!--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:

```
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 NetScaler Ingress Controller

December 31, 2023

NetScaler 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 NetScaler Ingress Controller on the NetScaler.

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.


```
Client                                   Upstream
   |        OPTIONS         |                |
   | ———————————————————————> |                |
   |       CORS_RESP        |                |
   | <————————————————————— |                |
   |        GET /foo/       |                |
   | ———————————————————————> | ———————————————> |
   |                        |       RESP     |
   | <————————————————————————————————————— |
```

CORS CRD definition

The CORS CRD is available in the NetScaler 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 NetScaler 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:

Attribute	Description
`servicenames`	Specifies the list of Kubernetes services to which you want to apply the CORS policies.
`allow_origin`	Specifies the list of allowed origins. Incoming origin is screened against this list.
`allow_methods`	Specifies the list of allowed methods as part of the CORS protocol.
`allow_headers`	Specifies the list of allowed headers as part of the CORS protocol.
`max_age`	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.

Attribute	Description
allow_credentials	Specifies whether the response can be shared when the credentials mode of the request is "include". The default value is 'true'.

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:

```
kubectl create -f cors-crd.yaml
```

For example:

```
$ kubectl create -f cors-crd.yaml
customresourcedefinition.apiextensions.k8s.io/corspolicies.citrix.com created
$ kubectl get crd
NAME                          CREATED AT
corspolicies.citrix.com       2021-05-21T20:01:13Z
```

How to write a CORS policy configuration

After you have deployed the CORS CRD provided by NetScaler 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. NetScaler 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`.

```
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
<!--NeedCopy-->
```

After you have defined the policy configuration, deploy the `.yaml` file using the following commands:

```
user@master:~/cors$ kubectl create -f corspolicy-example.yaml
corspolicy.citrix.com/corspolicy-example created
```

The NetScaler Ingress Controller applies the policy configuration on the Ingress NetScaler device.

Enable request retry feature using AppQoE for NetScaler Ingress Controller

December 31, 2023

When a NetScaler 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 NetScaler 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 NetScaler initiates the same request to the next available service. For more information request retry feature, see the NetScaler documentation

Now, you can configure request retry on NetScaler with NetScaler Ingress Controller.
Custom Resource Definitions (CRDs) are the primary way of configuring policies in cloud native deployments. Using the AppQoE CRD provided by NetScaler, you can configure request-retry policies on NetScaler with the NetScaler Ingress Controller. The AppQoE CRD enables communication between the NetScaler Ingress Controller and NetScaler for enforcing AppQoE policies.

AppQoE CRD definition

The AppQoE CRD is available in the NetScaler 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 NetScaler.

The following are the attributes provided in the AppQoE CRD:

Attribute	Description
`servicenames`	Specifies the list of Kubernetes services to which you want to apply the AppQoE policies.
`on-reset`	Specifies whether to set retry on connection Reset or Not
`on-timeout`	Specifies the time in milliseconds for retry
`number-of-retries`	Specifies the number of retries
`appqoe-criteria`	Specifies the expression for evaluating traffic.
`direction`	Specifies the bind point for binding the AppQoE policy.

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:

```
kubectl create -f appqoe-crd.yaml
```

How to write a AppQoE policy configuration

After you have deployed the AppQoE CRD provided by NetScaler 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.

```
apiVersion: citrix.com/v1
kind: appqoepolicy
metadata:
  name: targeturlappqoe
spec:
  appqoe-policies:
    - servicenames:
```



```
8           - apache
9         appqoe-policy:
10          operation-retry:
11            onReset: 'YES'
12            onTimeout: 33
13          number-of-retries: 2
14          appqoe-criteria: 'HTTP.REQ.HEADER("User-Agent").CONTAINS("
              Android")'
15          direction: REQUEST
```

After you have defined the policy configuration, deploy the `.yaml` file using the following commands:

$ kubectl create -f appqoe-example.yaml

Configuring wildcard DNS domains through NetScaler Ingress Controller

December 31, 2023

Wildcard DNS domains are used to handle requests for non-existent domains and subdomains. In a DNS zone, you can use wildcard domains to redirect queries for all non-existent domains or subdomains to a particular server, instead of creating a separate Resource Record (RR) for each domain. The most common use of a wildcard DNS domain is to create a zone that can be used to forward mail from the internet to some other mail system.

For more information on wildcard DNS domains, see the NetScaler documentation.

Now, you can configure wildcard DNS domains on a NetScaler with NetScaler Ingress Controller. Custom Resource Definitions (CRDs) are the primary way of configuring policies in cloud native deployments. Using the Wildcard DNS CRD provided by NetScaler, you can configure wildcard DNS domains on a NetScaler with the NetScaler Ingress Controller. The Wildcard DNS CRD enables communication between NetScaler Ingress Controller and NetScaler for supporting wild card domains.

Usage guidelines and restrictions

- For fully qualified domain names (FQDNs), there are multiple ways to add DNS records. You can either enable the `NS_CONFIG_DNS_REC` variable for NetScaler Ingress Controller for the Ingress resource or use the wildcard DNS CRD. However, you should make sure that they are configured through either CRD or ingress in order to avoid multiple IP mappings to the same domain.
- It is recommended to use the Wildcard DNS CRD for the wildcard DNS configurations.

- You cannot configure wildcard DNS entries in the DNS address record through ingress if the `NS_CONFIG_DNS_REC` is enabled for NetScaler Ingress Controller.

Wildcard DNS CRD definition

The Wildcard DNS CRD is available in the NetScaler Ingress Controller GitHub repo at wildcarddnsentry.yaml. The **Wildcard DNS CRD provides** attributes for the various options that are required to configure wildcard DNS entries on NetScaler.

The following are the attributes provided in the Wildcard DNS CRD:

Attribute	Description
`domain`	Specifies the wild card domain name configured for the zone.
`dnsaddrec`	Specifies the DNS Address record with the IPv4 address of the wildcard domain.
`dnsaaaarec`	Specifies the DNS AAAA record with the IPV6 address of the wildcard domain.
`soarec`	Specifies the SOA record configuration details.
`nsrec`	Specifies the name server configuration details.

Deploy the Wildcard DNS CRD

Perform the following to deploy the Wildcard DNS CRD:

1. Download the Wildcard DNS CRD.
2. Deploy the Wildcard DNS CD using the following command:

```
kubectl create -f wildcarddnsentry.yaml
```

How to write a Wildcard DNS configuration policy

After you have deployed the Wildcard DNS CRD provided by NetScaler in the Kubernetes cluster, you can define the wildcard DNS related configuration in a `yaml` file. In the `.yaml` file, use `wildcarddnsentry` in the kind field and in the `spec` section add the Wildcard DNS CRD attributes based on your requirement for the policy configuration.

The following is a sample YAML file definition that configures a SOA record, NS record, DNS zone, and address and AAAA Records on NetScaler.


```
apiVersion:
citrix.com/v1
kind: wildcarddnsentry
metadata:
  name: sample-config
spec:
  zone:
    domain: configexample
    dnsaddrec:
      domain-ip: 1.1.1.1
      ttl: 3600
    dnsaaaarec:
      domain-ip: '2001::.1'
      ttl: 3600
    soarec:
      origin-server: n2.configexample.com
      contact: admin.configexample.com
      serial: 100
      refresh: 3600
      retry: 3
      expire: 3600
    nsrec:
      nameserver: n1.configexample.com
      ttl: 3600
<!--NeedCopy-->
```

After you have defined the DNS configuration, deploy the `wildcarddns-example.yaml` file using the following command.

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

Entity name change

December 31, 2023

While adding the NetScaler entities, the NetScaler Ingress Controller maintains unique names per Ingress, service or namespace. Sometimes, it results in NetScaler entities with large names even exceeding the name limits in NetScaler.

Now, the naming format in the NetScaler 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 NetScaler Ingress Controller version 1.12.

Entity	Old naming format	New naming format	Description/Comments
`csvserver` (ingress)	`k8s-192.2.170.67_80_http`	`k8s-192.2.170.67_80_http`	no changes
`csvserver` (`type LoadBalancer`)	`k8s-apache_default_80_svc`	`k8s-apache_80_default_svc`	Now, the port is followed by a namespace
`lbvserver` (`type LoadBalancer`)	`k8s-apache_default_80_svc_k8s-apache_default_80_lbv`	`k8s-apache_80_lbv_wlikqxeo5vunbthsoj4lxegk7cdc` Comment: `lbsvc:apache,svcport:80,ns:`**`default`**	The comment for `type LoadBalancer` is now different
`servicegroup` (`type LoadBalancer`)	`k8s-apache_default_80_svc_k8s-apache_default_80_svc`	`k8s-apache_80_sgp_wlikqxeo5vunbthsoj4lxegk7cdc`	The suffix `sgp` is added
`cspolicy` or `csaction` or `responder policy`	`k8s-web-ingress_default_443_k8s-frontend_default_80_svc`	`k8s-frontend_80_csp_267xxi-ph5rn6hoygvrqrzpm4k6`	Moved service-name, service-port to the beginning, added suffix of cs, hashed ingress-name, ingress-port, and namespace

Entity	Old naming format	New naming format	Description/Comments
lbvserver (ingress)	k8s-web-ingress_default_443_k8s-frontend_default_80_svc	k8s-frontend_80_lbv_267pneiak5rw6hoygvrqrzpm4k6 Comment: {"ns:web-ingress,ingport:5080,ns:**default**,svc:frontend,svcport:80}	Suffix lbv and comment added to the entity
servicegroup (ingress)	k8s-web-ingress_default_443_k8s-frontend_default_80_svc	k8s-frontend_80_sgp_267pneiak5rw6hoygvrqrzpm4k6	Suffix sgp is added.
lbvserver (UDP)	k8s-web-ingress_default_9053-udp_k8s-bind_default_53-udp_svc	k8s-bind_53-udp_lbv_uyomblblagtxtw3xs63ik6wkpfmw	-udp is still appended to the port as earlier.

When you upgrade from an older version of the NetScaler Ingress Controller to the latest version, the NetScaler Ingress Controller renames all the entities with the new naming format. However, the NetScaler Ingress Controller does not handle the downgrade from the latest version to an older version.

Licensing

December 31, 2023

For licensing the NetScaler CPX, you need to provide the following information in the YAML for the NetScaler Application Delivery Management (ADM) to automatically pick the licensing information:

- LS_IP (License server IP) –Specify the NetScaler 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:
        - name: EULA
          value: "YES"
        - name: NS_PROTOCOL
          value: HTTP
        - name: NS_PORT
          value: "9080"
        - name: LS_IP
          value: <ADM IP>
        - name: LS_PORT
          value: "27000"
        - name: PLATFORM
          value: CP1000
        image: cpx-ingress:latest
        imagePullPolicy: Always
        name: cpx-ingress
        ports:
        - containerPort: 80
          name: http
          protocol: TCP
        - containerPort: 443
          name: https
          protocol: TCP
        - containerPort: 9080
          name: nitro-http
          protocol: TCP
        - containerPort: 9443
          name: nitro-https
          protocol: TCP
        securityContext:
```



```
            privileged: true
<!--NeedCopy-->
```

Deployment using Helm charts and NetScaler deployment builder

December 31, 2023

For deploying NetScaler 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 NetScaler cloud native deployments using NetScaler deployment builder, which is a GUI.

The following topologies are supported by the NetScaler deployment builder:

- Single-Tier
  - Ingress
  - Service type LoadBalancer
- Dual-Tier
  - NetScaler CPX as NodePort
  - NetScaler CPX as service of type LoadBalancer
- Multi-cluster Ingress
- Service mesh

For detailed information on how to use the NetScaler deployment builder, see the NetScaler deployment builder blog.