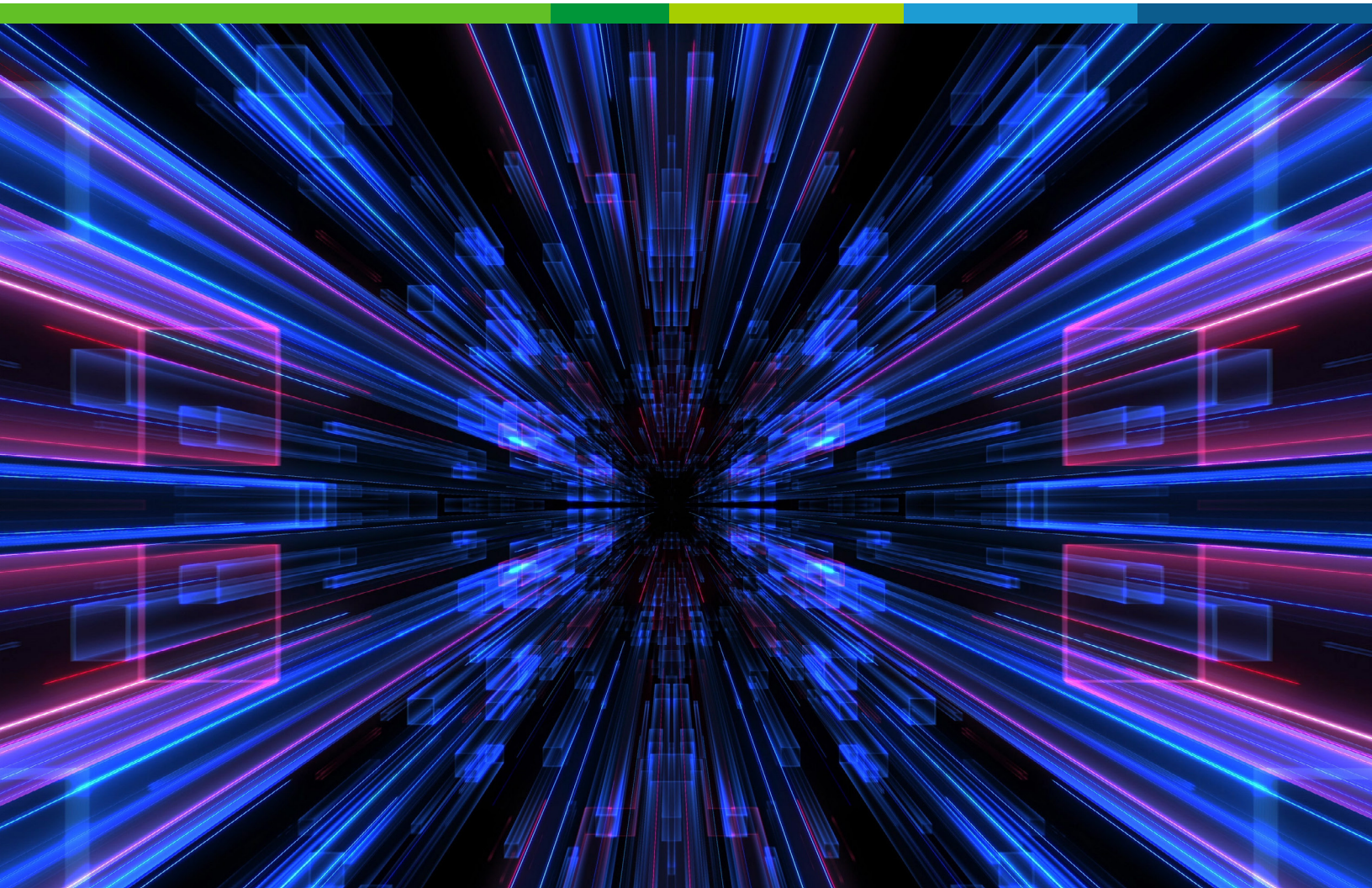


# Managing Kubernetes Traffic with F5 NGINX

## A Practical Guide

By Amir Rawdat  
Technical Marketing Manager, NGINX



# Table of Contents

Foreword .....	5
<b>1. Installing and Deploying F5 NGINX Ingress Controller and F5 NGINX Service Mesh.....</b>	<b>6</b>
What Is an Ingress Controller and Why Is It Important? .....	6
What's Special About NGINX Ingress Resources? .....	7
Prerequisites .....	7
Installation and Deployment Instructions for NGINX Ingress Controller .....	7
What Is a Service Mesh and Do I Need One? .....	10
Why Should I Try NGINX Service Mesh? .....	11
Why Integrate NGINX Ingress Controller with NGINX Service Mesh? .....	11
NGINX Service Mesh Architecture .....	12
Installation and Deployment Instructions for NGINX Service Mesh .....	13
Installation with the NGINX Service Mesh CLI .....	13
Install the NGINX Service Mesh CLI .....	13
Install NGINX Service Mesh .....	14
Installation with Helm .....	15
Prerequisites .....	15
Installing with Helm Repository .....	15
Installing with Chart Sources .....	16
Migrating from the Community Ingress Controller to F5 NGINX Ingress Controller .....	17
Option 1: Migrate Using NGINX Ingress Resources .....	17
Set Up SSL Termination and HTTP Path-Based Routing .....	18
Set Up TCP/UDP Load Balancing and TLS Passthrough .....	18
Convert Community Ingress Controller Annotations to NGINX Ingress Resources .....	18
Canary Deployments .....	19
Traffic Control .....	20
Header Manipulation .....	22
Other Proxying and Load Balancing Annotations .....	23
mTLS Authentication .....	24
Session Persistence (Exclusive to NGINX Plus) .....	24
Option 2: Migrate Using the Kubernetes Ingress Resource .....	25
Advanced Configuration with Annotations .....	25
Global Configuration with ConfigMaps .....	26
Chapter Summary .....	28

<b>2. Traffic Management Use Cases</b>	<b>29</b>
Load Balancing TLS-Encrypted Traffic with TLS Passthrough	32
Enabling Multi-Tenancy and Namespace Isolation	34
Delegation with NGINX Ingress Controller	34
Implementing Full Self-Service	35
Implementing Restricted Self-Service	37
Leveraging Kubernetes RBAC in a Restricted Self-Service Model	39
Adding Policies	40
Configuring Traffic Control and Traffic Splitting	42
Why Is Traffic Management So Vital?	42
How Do I Pick a Traffic Control or Traffic Splitting Method?	43
When Do I Use NGINX Ingress Controller vs. NGINX Service Mesh?	43
Deploying the Sample Application	43
Configuring Traffic Control	47
Configuring Rate Limiting	47
Activating Client Rate Limiting with NGINX Ingress Controller	48
Allowing Bursts of Requests with NGINX Ingress Controller	51
Activating Interservice Rate Limiting with NGINX Service Mesh	52
Configuring Circuit Breaking	54
Returning a Custom Page	58
Configuring Traffic Splitting	59
Generating Cluster-Internal Traffic to Split	59
Implementing Blue-Green Deployment	60
Blue-Green Deployment with NGINX Service Mesh	60
Blue-Green Deployment with NGINX Ingress Controller	62
Implementing Canary Deployment	63
Canary Deployment with NGINX Service Mesh	63
Canary Deployment with NGINX Ingress Controller	64
Implementing A/B Testing	65
A/B Testing with NGINX Service Mesh	65
A/B Testing with NGINX Ingress Controller	67
Implementing Debug Routing	68
Debug Routing with NGINX Service Mesh	68
Debug Routing with NGINX Ingress Controller	70
Chapter Summary	71

<b>3. Monitoring and Visibility Use Cases</b>	72
Monitoring with the NGINX Plus Live Activity Monitoring Dashboard	72
Distributed Tracing, Monitoring, and Visualization with Jaeger, Prometheus, and Grafana	74
Enabling Distributed Tracing, Monitoring, and Visualization for NGINX Service Mesh	74
Enabling Distributed Tracing for NGINX Ingress Controller	75
Enabling Monitoring and Visualization for NGINX Ingress Controller	76
Visualizing Distributed Tracing and Monitoring Data	77
Logging and Monitoring with the Elastic Stack	80
Configuring the NGINX Ingress Controller Access and Error Logs	80
Enabling Filebeat	81
Displaying NGINX Ingress Controller Log Data with Filebeat	83
Enabling Metricbeat and Displaying NGINX Ingress Controller and NGINX Service Mesh Metrics	87
Displaying Logs and Metrics with Amazon CloudWatch	89
Configuring CloudWatch	89
Creating Graphs in CloudWatch	92
Capturing Logs in CloudWatch with Fluent Bit	93
Chapter Summary	95
<b>4. Identity and Security Use Cases</b>	96
Implementing SSO with Okta	97
Prerequisites	97
Configuring Okta as the IdP	98
Configuring NGINX Ingress Controller as the Relaying Party with Okta	99
Implementing SSO with Azure Active Directory	102
Configuring Azure AD as the IdP	102
Configuring NGINX Ingress Controller as the Relaying Party with Azure AD	105
Implementing SSO with Ping Identity	109
Configuring Ping Identity as the IdP	109
Configuring NGINX Ingress Controller as the Relaying Party with Ping Identity	112
Implementing SSO for Multiple Apps	114
Deploying NGINX App Protect with NGINX Ingress Controller	115
Why Is Integrating a WAF into NGINX Ingress Controller So Significant?	115
Configuring NGINX App Protect in NGINX Ingress Controller	116
Configuring NGINX App Protect WAF with NGINX Ingress Resources	117
Configuring NGINX App Protect WAF with the Standard Ingress Resource	121
Logging	121
Resource Thresholds	123
Chapter Summary	124

# Foreword

Microservices architectures introduce several benefits to the application development and delivery process. Microservices-based apps are easier to build, test, maintain, and scale. They also reduce downtime through better fault isolation.

While container-based microservices apps have profoundly changed the way DevOps teams deploy applications, they have also introduced challenges. Kubernetes – the de facto container orchestration platform – is designed to simplify management of containerized apps, but it has its own complexities and a steep learning curve. This is because responsibility for many functions that traditionally run inside an app (security, logging, scaling, and so on) are shifted to the Kubernetes networking fabric.

To manage this complexity, DevOps teams need a data plane that gives them control of Kubernetes networking. The data plane is the key component that connects microservices to end users and each other, and managing it effectively is critical to achieving stability and predictability in an environment where modern apps are evolving constantly.

Ingress controller and service mesh are the two Kubernetes-native technologies that provide the control you need over the data plane. This hands-on guide to F5 NGINX Ingress Controller and F5 NGINX Service Mesh includes thorough explanations, diagrams, and code samples to prepare you to deploy and manage production-grade Kubernetes environments.

**Chapter 1** introduces NGINX Ingress Controller and NGINX Service Mesh and walks you through installation and deployment, including an integrated solution for managing both north-south and east-west traffic.

**Chapter 2** steps through configurations for key use cases:

- **TCP/UDP and TLS Passthrough load balancing** – Supporting TCP/UDP workloads
- **Multi-tenancy and delegation** – For safe and effective sharing of resources in a cluster
- **Traffic control** – Rate limiting and circuit breaking
- **Traffic splitting** – Blue-green and canary deployments, A/B testing, and debug routing

**Chapter 3** covers monitoring, logging, and tracing, which are essential for visibility and insight into your distributed applications. You'll learn how to export NGINX metrics to third-party tools including AWS, Elastic Stack, and Prometheus.

And of course, we can't forget about security. **Chapter 4** addresses several mechanisms for protecting your apps, including centralized authentication on the Ingress controller, integration with third-party SSO solutions, and F5 NGINX App Protect WAF policies for preventing advanced attacks and data exfiltration methods.

I'd like to thank my collaborators on this eBook: Jenn Gile for project conception and management, Sandra Kennedy for the cover design, Tony Mauro for editing, and Michael Weil for the layout and diagrams.

This is our first edition of this eBook and we welcome your input on important scenarios to include in future editions.

Amir Rawdat

Technical Marketing Engineer, F5 NGINX

# 1. Installing and Deploying F5 NGINX Ingress Controller and F5 NGINX Service Mesh

In this chapter we explain how to install and deploy NGINX Ingress Controller and NGINX Service Mesh. We also detail how to migrate from the NGINX Ingress Controller maintained by the Kubernetes community ([kubernetes/ingress-nginx](#)) to our version ([nginxinc/kubernetes-ingress](#)).

- [Installing and Deploying NGINX Ingress Controller](#)
- [Installing and Deploying NGINX Service Mesh](#)
- [Migrating from the Community Ingress Controller to NGINX Ingress Controller](#)
- [Chapter Summary](#)

## INSTALLING AND DEPLOYING NGINX INGRESS CONTROLLER

As you start off using Kubernetes, your cluster typically has just a few simple applications that serve requests from external clients and don't exchange much data with other services in the cluster. For this use case, NGINX Ingress Controller is usually sufficient on its own, and we begin with instructions for a stand-alone NGINX Ingress Controller deployment.

As your cluster topology becomes more complicated, adding a service mesh often becomes necessary. We cover installation and deployment of NGINX Service Mesh in the [next section](#).

### What Is an Ingress Controller and Why Is It Important?

In Kubernetes, the Ingress controller is a specialized load balancer that bridges between the internal network, which connects the containerized apps running within the Kubernetes cluster, and the external network outside the Kubernetes cluster. Ingress controllers are used to configure and manage external interactions with Kubernetes pods that are labeled to a specific service. Ingress controllers have many features of traditional external load balancers, like TLS termination, handling multiple domains and namespaces, and of course, load balancing traffic.

You configure the Ingress controller with the Kubernetes API. The Ingress controller integrates with Kubernetes components so that it can automatically reconfigure itself appropriately when service endpoints scale up and down. And there's another bonus! Ingress controllers can also enforce egress rules which permit outgoing traffic from certain pods only to specific external services, or ensure that traffic is secured using mTLS.

AS YOUR CLUSTER TOPOLOGY  
BECOMES MORE COMPLICATED,  
ADDING A SERVICE MESH  
OFTEN BECOMES NECESSARY



## What's Special About NGINX Ingress Resources?

NGINX Ingress Controller supports the standard Kubernetes [Ingress resource](#), but also supports [NGINX Ingress resources](#), which provide enterprise-grade features such as more flexible load-balancing options, circuit breaking, routing, header manipulation, mutual TLS (mTLS) authentication, and web application firewall (WAF). In contrast, the native Kubernetes Ingress resource facilitates configuration of load balancing in Kubernetes but does not provide those enterprise-grade features nor other customizations.

## Prerequisites

To install and deploy NGINX Ingress Controller, you need:

- A working Kubernetes environment where you have administrative privilege. See the [Kubernetes documentation](#) to get started.
- A subscription to the NGINX Ingress Controller based on NGINX Plus, if you also want to deploy NGINX Service Mesh and NGINX App Protect. To explore all use cases in later chapters of this eBook, you must have NGINX Service Mesh. If you don't already have a paid subscription, start a [30-day free trial](#) before continuing.

## Installation and Deployment Instructions for NGINX Ingress Controller

To install and deploy NGINX Ingress Controller, complete these steps:

1. Complete the indicated steps in these sections of the [NGINX Ingress Controller documentation](#):
  - **Prerequisites:** Step 2
  - **1. Configure RBAC:** Steps 1–3
  - **2. Create Common Resources:** Steps 1–3, plus the two steps in the *Create Custom Resources* subsection and the one step in the *Resources for NGINX App Protect* subsection
2. Clone the GitHub repo for this eBook, which includes configuration files for the NGINX Ingress Controller based on NGINX Plus and the [sample bookinfo application](#) used in later chapters:

```
$ git clone https://github.com/nginxinc/ebook-managing-  
kubernetes-nginx.git
```

### 3. Deploy NGINX Ingress Controller.

**Note (again!):** You must use the NGINX Ingress Controller based on NGINX Plus if you later want to deploy NGINX Service Mesh as well as explore all use cases in this guide.

- If deploying the NGINX Open Source-based NGINX Ingress Controller, apply the [nginx-ingress.yaml](#) file provided in the [nginxinc/kubernetes-ingress](#) repo on GitHub:

```
$ kubectl apply -f ./deployments/deployment/nginx-ingress.yaml
```

- If deploying the NGINX Ingress Controller based on NGINX Plus, along with NGINX App Protect:
  - a) Download the JSON Web Token (JWT) provided with your NGINX Ingress Controller subscription from [MyF5](#) (if your subscription covers multiple NGINX Ingress Controller instances, there is a separate JWT for each instance).
  - b) Create a Kubernetes Secret, which is required for pulling images from the NGINX private registry. Substitute the JWT obtained in the previous step for **<your\_JWT>**:

```
$ kubectl create secret docker-registry regcred \
  --docker-server=private-registry.nginx.com \
  --docker-username=<your_JWT> \
  --docker-password=none -n nginx-ingress
secret/regcred created
```

- c) On line 29 of **Installation-Deployment/nginx-plus-ingress.yaml**, edit the second and third elements in the path to the NGINX Ingress Controller image to match the latest version available for your operating system (do not change the first element, **private-registry.nginx.com**). Choose an image that also includes NGINX App Protect WAF. The default value is appropriate for Debian and Ubuntu.

```
29 -image: private-registry.nginx.com/nginx-ic-nap/nginx-plus-
    ingress:2.x.y
```

[View on GitHub](#)

For a list of the available images, see [NGINX Ingress Controller Technical Specifications](#).

- d) Apply **nginx-plus-ingress.yaml** to deploy NGINX Ingress Controller with NGINX App Protect:

```
$ kubectl apply -f ./Installation-Deployment/nginx-plus-
    ingress.yaml
deployment.apps/nginx-ingress created
```



4. Check that the NGINX Ingress Controller pod is up and running, as confirmed by the value **Running** in the **STATUS** column:

```
$ kubectl get pods -n nginx-ingress
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-ingress-7b8f6dfbcb-pd22b	1/1	Running	0	1m

5. Deploy a network load balancer to expose NGINX Ingress Controller on an IP address that's external to the cluster, so that external clients can reach apps and services in the cluster. For cloud deployments, the following instructions create a Kubernetes **LoadBalancer** service for this purpose.

- For AWS, create a **LoadBalancer** service:

- a) Apply the ConfigMap defined in **Installation-Deployment/nginx-config.yaml**:

```
$ kubectl apply -f ./Installation-Deployment/nginx-config.yaml
configmap/nginx-config created
```

The file includes the following keys, which enable the PROXY Protocol for proper interaction with AWS Elastic Load Balancing (ELB).

```
6  data:
7      proxy-protocol: "True"
8      real-ip-header: "proxy_protocol"
9      set-real-ip-from: "0.0.0.0/0"
```

[View on GitHub](#)

- b) Apply the **LoadBalancer** configuration defined in **Installation-Deployment/loadbalancer-aws-elb.yaml**:

```
$ kubectl apply -f ./Installation-Deployment/loadbalancer-aws-elb.yaml
service/nginx-ingress created
```

- For Azure or Google Cloud Platform, apply the **LoadBalancer** configuration defined in **Installation-Deployment/loadbalancer.yaml**:

```
$ kubectl apply -f ./Installation-Deployment/loadbalancer.yaml
service/nginx-ingress created
```

FOR ON-PREMISES  
DEPLOYMENTS, YOU NEED  
TO DEPLOY YOUR OWN  
NETWORK LOAD BALANCER

- For on-premises deployments, you need to deploy your own network load balancer that integrates with the cluster, because Kubernetes does not offer a native implementation of a **LoadBalancer** service for this use case. Network operators typically deploy systems like the following in front of an on-premises Kubernetes cluster:
  - **F5 BIG-IP** or **Citrix ADC** hardware load balancer
  - Software network load balancers like **MetalLB**

At this point NGINX Ingress Controller is deployed. You can either:

- Continue to the [next section](#) to add NGINX Service Mesh (required to explore all of the traffic-management use cases in [Chapter 2](#)).
- Continue to [Chapter 3](#) to explore observability use cases or [Chapter 4](#) for security use cases.

## INSTALLING AND DEPLOYING NGINX SERVICE MESH

As [previously mentioned](#), NGINX Ingress Controller on its own is typically sufficient for Kubernetes clusters with simple applications that serve requests from external clients and don't exchange much data with other services in the cluster. But as your cluster topology becomes more complicated, adding a service mesh often becomes helpful, if not required, for proper operation. In this section we install and deploy NGINX Service Mesh and integrate it with NGINX Ingress Controller.

Also as previously mentioned, you must use the NGINX Ingress Controller based on NGINX Plus to integrate with NGINX Service Mesh, and some use cases in [Chapter 2](#) are possible only with the combination. (For ease of reading, the remainder of this section uses the term *NGINX Ingress Controller* for the NGINX Plus-based model of the product.)

Let's start with a look at the capabilities provided by NGINX Service Mesh and NGINX Ingress Controller, when they are used, and how they can be used together.

### What Is a Service Mesh and Do I Need One?

A service mesh is a component of orchestration tools for containerized environments such as Kubernetes, typically responsible for functions that include routing traffic among containerized applications, serving as the interface for defining autonomous service-to-service mutual TLS (mTLS) policies and then enforcing them, and providing visibility into application availability and security. As such, a service mesh extends Layer 7 control down to the level of service-to-service communication. Like Kubernetes as a whole, service meshes consist of control, management, and data planes.

Service meshes typically handle traffic management and security in a way that's transparent to the containerized applications. By offloading functions like SSL/TLS and load balancing, service meshes free developers from having to implement security or service availability separately in each application. An enterprise-grade service mesh provides solutions for a variety of "problems":

- Securing traffic with end-to-end encryption and mTLS
- Orchestration via injection and sidecar management, and Kubernetes API integration
- [Management of service traffic](#), including load balancing, traffic control (rate limiting and circuit breaking), and traffic splitting (canary and blue-green deployments, A/B testing, and debug routing)

THE LARGER AND MORE  
COMPLEX THE SERVICE MESH,  
THE MORE HELPFUL AN  
INDEPENDENT MANAGEMENT  
PLANE CAN BE

IF YOU ARE READY FOR  
A SERVICE MESH, NGINX  
SERVICE MESH IS A GREAT  
OPTION BECAUSE IT IS  
LIGHTWEIGHT, TURNKEY,  
AND DEVELOPER-FRIENDLY

- **Improved monitoring and visibility** of service-to-service traffic with popular tools like Prometheus and Grafana
- **Simplified management of Kubernetes ingress and egress traffic** when integrated natively with an Ingress controller

Service meshes range in focus from small and very focused (like NGINX Service Mesh) to very large with a comprehensive set of network and cluster management tools (like Istio), and everywhere in between. The larger and more complex the service mesh, the more helpful an independent management plane can be.

At NGINX, we think it's no longer a binary question of "Do I have to use a service mesh?" but rather "When will I be ready for a service mesh?" We believe that anyone deploying containers in production and using Kubernetes to orchestrate them has the potential to reach the level of app and infrastructure maturity where a service mesh adds value. But as with any technology, implementing a service mesh before you need one just adds risk and expense that outweigh the possible benefits to your business. For our six-point readiness checklist, read [How to Choose a Service Mesh](#) on our blog.

### Why Should I Try NGINX Service Mesh?

If you are ready for a service mesh, NGINX Service Mesh is a great option because it is lightweight, turnkey, and developer-friendly. You don't need a team of people to run it. It leverages NGINX Plus as the sidecar to operate highly available and scalable containerized environments, providing a level of enterprise traffic management, performance, and scalability to the market that other sidecars don't offer. NGINX Service Mesh provides the seamless and transparent load balancing, reverse proxy, traffic routing, identity, and encryption features needed for production-grade service mesh deployments.

If you run containers on Kubernetes in production, then you can use NGINX Service Mesh to reliably deploy and orchestrate your services for many use cases and features such as configuration of mTLS between app services. For especially large, distributed app topologies, NGINX Service Mesh provides full visibility, monitoring, and security.

### Why Integrate NGINX Ingress Controller with NGINX Service Mesh?

Not all Ingress controllers integrate with all service meshes, and when they do, it's not always pretty. NGINX Service Mesh was designed to tightly and perfectly integrate with NGINX Ingress Controller, which provides benefits including:

- A unified data plane which can be managed in a single configuration, saving you time and helping avoid errors resulting in improper traffic routing and security issues
- Easier implementation of zero-trust security in Kubernetes clusters; using tools that tightly integrate with minimal effort helps you avoid potentially painful complications and configuration hacks later on

Integrating NGINX Ingress Controller with NGINX Service Mesh yields a unified data plane with production-grade security, functionality, and scale.

## NGINX Service Mesh Architecture

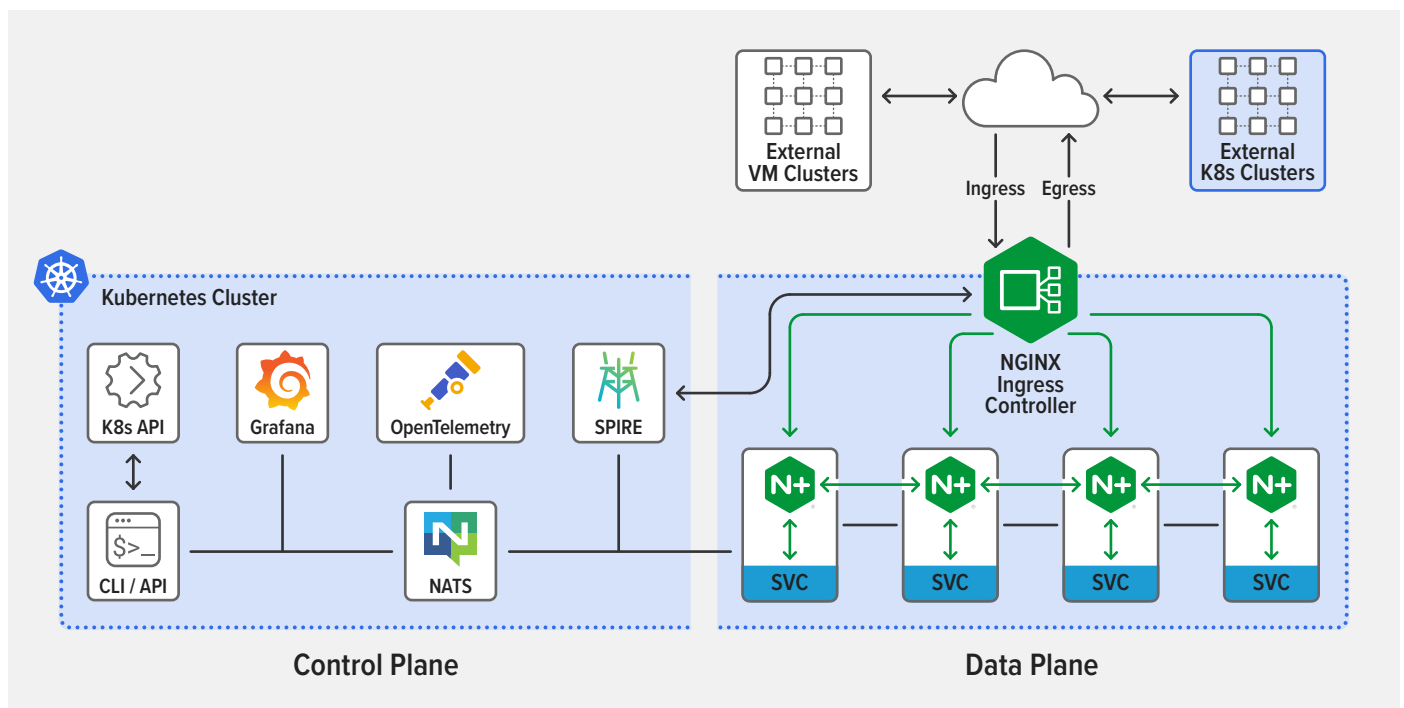
IF YOU ARE READY TO  
INTEGRATE NGINX INGRESS  
CONTROLLER WITH  
NGINX SERVICE MESH,  
WHERE DO YOU START?

If you are ready to integrate NGINX Ingress Controller with NGINX Service Mesh, where do you start? Before we step through the details of the installation, it is important to understand the architectural components of the NGINX Service Mesh.

NGINX Service Mesh consists of two key components:

- **Data plane** – Handles the traffic between services in the Kubernetes cluster and performs traffic-management functions that include load balancing, reverse proxy, traffic routing, identity, and encryption. The data plane component is implemented with sidecars, which are proxy instances responsible for intercepting and routing all traffic for their service and executing traffic-management rules.
- **Control plane** – Configures and manages the data plane, providing instant support for optimizing apps and their updates, all of which make it possible to keep the instances protected and integrated with other components.

The following diagram depicts the interaction of the control and data planes in NGINX Service Mesh.



THERE ARE SEVERAL METHODS  
AVAILABLE FOR INSTALLING  
NGINX SERVICE MESH AND  
INTEGRATING IT WITH  
NGINX INGRESS CONTROLLER

As shown in the diagram, sidecar proxies interoperate with the following open source solutions:

- [Grafana](#)
- Kubernetes Ingress controllers like [NGINX Ingress Controller](#)
- [NATS](#)
- [OpenTelemetry](#)
- SPIRE, the [SPIFFE](#) runtime environment

## Installation and Deployment Instructions for NGINX Service Mesh

There are several methods available for installing NGINX Service Mesh and integrating it with NGINX Ingress Controller. In this section we provide instructions for two popular methods:

- [Installation with the NGINX Service Mesh CLI](#)
- [Installation with Helm](#)

### Installation with the NGINX Service Mesh CLI

When installing NGINX Service Mesh with a Kubernetes manifest, there are two stages:

- [Install the NGINX Service Mesh CLI](#)
- [Install NGINX Service Mesh](#)

#### *Install the NGINX Service Mesh CLI*

The NGINX Service Mesh control plane is designed to connect to an API, a CLI, and a GUI for managing the app. Here you install the NGINX Service Mesh CLI (`nginx-meshctl`).

The following instructions apply to Linux, but you can also install the CLI on macOS and Windows; for instructions, see the [NGINX Service Mesh documentation](#).

1. Login at <https://downloads.f5.com/> (create an account if you don't already have one) and download the binary for `nginx-meshctl`:
  - Click **Find a Download**.
  - Select **NGINX\_Service\_Mesh** under **NGINX-Public**.
  - Select the file for the Linux platform: `nginx-meshctl_linux.gz`.
2. Open a terminal and run the following command to unzip the downloaded binary file:

```
$ gunzip nginx-meshctl_linux.gz
```

3. Copy the tool to the local `/usr/bin/` directory and make it executable:

```
$ sudo cp nginx-meshctl_linux /usr/bin/nginx-meshctl
$ sudo chmod +x /usr/bin/nginx-meshctl
```

4. Verify that the CLI is working – if it is, usage instructions and a list of available commands and flags is displayed.

```
$ nginx-meshctl help
```

### *Install NGINX Service Mesh*

You configure NGINX Service Mesh in one of three mutual TLS (mTLS) modes depending on which types of traffic need to be protected by encryption:

- **off** – mTLS is disabled and incoming traffic is accepted from any source
- **permissive** – mTLS secures communication among injected pods, which can also communicate in clear text with external services
- **strict** – mTLS secures communication among injected pods and external traffic cannot enter the Kubernetes cluster

In development environments, **off** mode is probably acceptable. For production we recommend **strict** mode, as in the following steps.

1. Deploy NGINX Service Mesh and enable **strict** mode:

```
$ nginx-meshctl deploy --sample-rate 1 --mtls-mode strict
Deploying NGINX Service Mesh...
All resources created. Testing the connection to the Service
Mesh API Server
Connected to the NGINX Service Mesh API successfully.
NGINX Service Mesh is running.
```

2. Verify that all pods are up and running in the **nginx-mesh** namespace. A list of pods like the following indicates a successful deployment.

```
$ kubectl get pods -n nginx-mesh
```

NAME	READY	STATUS	RESTARTS	AGE
grafana-7c6c88b959-7q9t1	1/1	Running	0	10s
jaeger-77457fb8d4-6mfmg	1/1	Running	0	10s
nats-server-859dfb4b6d-ljm7n	2/2	Running	0	10s
nginx-mesh-api-7fbfc8df4c-g9qmj	1/1	Running	0	10s
nginx-mesh-metrics-85b55579bb-d75k8	1/1	Running	0	10s
prometheus-8d5fb5879-hvc84	1/1	Running	0	10s
spire-agent-8lczf	1/1	Running	0	10s
spire-agent-stzcq	1/1	Running	0	10s
spire-server-0	2/2	Running	0	10s

## Installation with Helm

Helm is a popular and supported tool for automating creation, configuration, packaging, and deployment of Kubernetes resources. After fulfilling the [prerequisites](#), you can use either of two procedures:

- [Installing with Helm Repository](#)
- [Installing with Chart Sources](#)

### Prerequisites

To install NGINX Service Mesh with Helm you need:

- Access to a Kubernetes environment via the **kubectl** command line utility
- [Helm 3.0+](#)
- A clone of the [NGINX Service Mesh GitHub repository](#) on your local machine

### Installing with Helm Repository

1. Add the Helm repository:

```
$ helm repo add nginx-stable https://helm.nginx.com/stable
$ helm repo update
```



2. Install NGINX Service Mesh from the Helm repository, substituting a deployment name such as **my-service-mesh-release** for **<your\_deployment\_name>**:

```
$ helm install <your_deployment_name> nginx-stable/nginx-  
service-mesh -n nginx-mesh --create-namespace --wait  
NAME: <your_deployment_name>  
LAST DEPLOYED: Day Mon DD HH:MM:SS YYYY  
NAMESPACE: default  
STATUS: deployed  
REVISION: 1  
TEST SUITE: None  
NOTES:  
The NGINX Service Mesh has been installed.
```

#### *Installing with Chart Sources*

With chart sources, you specify configuration parameters in **values.yaml** (not included in the eBook repo), which are referenced when the following command installs the **nginx-service-mesh** Helm chart. For **<your\_deployment\_name>** substitute a name such as **my-service-mesh-release**.

```
$ cd helm-chart  
$ helm install <your_deployment_name> -f values.yaml . -n nginx-  
mesh --create-namespace --wait
```

Any configurable parameters that you do not specify in **values.yaml** are set to their default value.

For more details on chart sources and the full list of configurable parameters for the Helm chart, see the [NGINX Ingress Controller repository](#) and [NGINX Service Mesh documentation](#).

AS THEIR KUBERNETES  
DEPLOYMENT MATURES, SOME  
ORGANIZATIONS FIND THEY  
NEED ADVANCED FEATURES  
OR WANT COMMERCIAL  
SUPPORT WHILE KEEPING  
NGINX AS THE DATA PLANE

## MIGRATING FROM THE COMMUNITY INGRESS CONTROLLER TO F5 NGINX INGRESS CONTROLLER

Many organizations setting up Kubernetes for the first time start with the NGINX Ingress Controller developed and maintained by the Kubernetes community ([kubernetes/ingress-nginx](#)). As their Kubernetes deployment matures, however, some organizations find they need advanced features or want commercial support while keeping NGINX as the data plane.

One option is to migrate to the NGINX Ingress Controller based on NGINX Plus and maintained by F5 NGINX ([nginxinc/kubernetes-ingress](#)), and here we provide complete instructions so you can avoid some complications that result from differences between the two projects. (As mentioned previously, you must use the NGINX Plus-based NGINX Ingress Controller and NGINX Service Mesh if you want to explore all use cases in this guide.)

To distinguish between the two projects in the remainder of this guide, we refer to the NGINX Ingress Controller maintained by the Kubernetes community ([kubernetes/ingress-nginx](#)) as the “community Ingress controller” and the one maintained by F5 NGINX ([nginxinc/kubernetes-ingress](#)) as “NGINX Ingress Controller”.

There are two ways to migrate from the community Ingress controller to NGINX Ingress Controller:

- **Option 1: Migrate Using NGINX Ingress Resources**  
This is the optimal solution, because [NGINX Ingress resources](#) support the broader set of Ingress networking capabilities required in production-grade Kubernetes environments. For more information on NGINX Ingress resources, watch our webinar, [Advanced Kubernetes Deployments with NGINX Ingress Controller](#).
- **Option 2: Migrate Using the Kubernetes Ingress Resource**  
This option is recommended if you are committed to using the standard [Kubernetes Ingress resource](#) to define Ingress load-balancing rules.

### Option 1: Migrate Using NGINX Ingress Resources

With this migration option, you use the standard Kubernetes Ingress resource to set root capabilities and [NGINX Ingress resources](#) to enhance your configuration with increased capabilities and ease of use.

The custom resource definitions (CRDs) for NGINX Ingress resources – [VirtualServer](#), [VirtualServerRoute](#), [TransportServer](#), [GlobalConfiguration](#), and [Policy](#) – enable you to easily delegate control over various parts of the configuration to different teams (such as AppDev and security teams) as well as provide greater configuration safety and validation.

## Set Up SSL Termination and HTTP Path-Based Routing

The table maps the configuration of SSL termination and Layer 7 path-based routing in the **spec** field of the standard Kubernetes Ingress resource with the **spec** field in the NGINX VirtualServer resource. The syntax and indentation differ slightly in the two resources, but they accomplish the same basic Ingress functions.

KUBERNETES INGRESS RESOURCE	NGINX VIRTUALSERVER RESOURCE
<pre>apiVersion: networking.k8s.io/v1 kind: Ingress metadata:   name: nginx-test spec:   tls:     - hosts:       - foo.bar.com       secretName: tls-secret   rules:     - host: foo.bar.com       http:         paths:           - path: /login             backend:               serviceName: login-svc               servicePort: 80           - path: /billing             serviceName: billing-svc             servicePort: 80</pre>	<pre>apiVersion: networking.k8s.io/v1 kind: VirtualServer metadata:   name: nginx-test spec:   host: foo.bar.com   tls:     secret: tls-secret   upstreams:     - name: login       service: login-svc       port: 80     - name: billing       service: billing-svc       port: 80   routes:     - path: /login       action:         pass: login     - path: /billing       action:         pass: billing</pre>

## Set Up TCP/UDP Load Balancing and TLS Passthrough

With the community Ingress Controller, a Kubernetes [ConfigMap](#) API object is the [only way to expose TCP and UDP services](#).

With NGINX Ingress Controller, [TransportServer](#) resources define a broad range of options for TLS Passthrough and TCP and UDP load balancing. TransportServer resources are used in conjunction with [GlobalConfiguration](#) resources to control inbound and outbound connections.

For more information, see [Load Balancing TCP and UDP Traffic](#) and [Load Balancing TLS-Encrypted Traffic with TLS Passthrough](#) in Chapter 2.

## Convert Community Ingress Controller Annotations to NGINX Ingress Resources

Production-grade Kubernetes deployments often need to extend basic Ingress rules to implement advanced use cases, including canary and blue-green deployments, traffic throttling, ingress-egress traffic manipulation, and more.

PRODUCTION-GRADE  
KUBERNETES DEPLOYMENTS  
OFTEN NEED TO EXTEND  
BASIC INGRESS RULES  
TO IMPLEMENT ADVANCED  
USE CASES

EVEN AS YOU PUSH FREQUENT  
CODE CHANGES TO YOUR  
PRODUCTION CONTAINER  
WORKLOADS, YOU MUST  
CONTINUE TO SERVE YOUR  
EXISTING USERS

The community Ingress controller implements many of these use cases with Kubernetes [annotations](#). However, many of these annotations are built with custom Lua extensions that pertain to very specific NGINX Ingress resource definitions and as a result are not suitable for implementing advanced functionality in a stable and supported production environment.

In the following sections we show how to convert community Ingress controller annotations into NGINX Ingress Controller resources.

*Canary Deployments*

Even as you push frequent code changes to your production container workloads, you must continue to serve your existing users. Canary and blue-green deployments enable you to do this, and you can perform them on the NGINX Ingress Controller data plane to achieve stable and predictable updates in production-grade Kubernetes environments.

The table shows the fields in NGINX VirtualServer and VirtualServerRoute resources that correspond to community Ingress Controller [annotations for canary deployments](#).

The community Ingress controller evaluates canary annotations in this order of precedence:

- 1. `nginx.ingress.kubernetes.io/canary-by-header`
- 2. `nginx.ingress.kubernetes.io/canary-by-cookie`
- 3. `nginx.ingress.kubernetes.io/canary-by-weight`

For NGINX Ingress Controller to evaluate them the same way, they must appear in that order in the NGINX VirtualServer or VirtualServerRoute manifest.

COMMUNITY INGRESS CONTROLLER	NGINX INGRESS CONTROLLER
<code>nginx.ingress.kubernetes.io/canary: "true"</code> <code>nginx.ingress.kubernetes.io/canary-by-header: "httpHeader"</code>	<code>matches:</code> - <code>conditions:</code> - <code>header: httpHeader</code> <code>value: never</code> <code>action:</code> <code>pass: echo</code> - <code>header: httpHeader</code> <code>value: always</code> <code>action:</code> <code>pass: echo-canary</code> <code>action:</code> <code>pass: echo</code>
<code>nginx.ingress.kubernetes.io/canary: "true"</code> <code>nginx.ingress.kubernetes.io/canary-by-header: "httpHeader"</code> <code>nginx.ingress.kubernetes.io/canary-by-header-value: "my-value"</code>	<code>matches:</code> - <code>conditions:</code> - <code>header: httpHeader</code> <code>value: my-value</code> <code>action:</code> <code>pass: echo-canary</code> <code>action:</code> <code>pass: echo</code>

(continues)

COMMUNITY INGRESS CONTROLLER	NGINX INGRESS CONTROLLER
<pre> nginx.ingress.kubernetes.io/canary: "true" nginx.ingress.kubernetes.io/canary-by-cookie: "cookieName" </pre>	<pre> matches: - conditions:   - cookie: cookieName     value: never   action:     pass: echo - cookie: cookieName   value: always   action:     pass: echo-canary action:   pass: echo </pre>
<pre> nginx.ingress.kubernetes.io/canary: "true" nginx.ingress.kubernetes.io/canary-weight: "10" </pre>	<pre> splits: - weight: 90   action:     pass: echo - weight: 10   action:     pass: echo-canary </pre>

### *Traffic Control*

In microservices environments, where applications are ephemeral by nature and so more likely to return error responses, DevOps teams make extensive use of traffic-control policies – such as circuit breaking and rate and connection limiting – to prevent error conditions when applications are unhealthy or not functioning as expected.

The table shows the fields in NGINX VirtualServer and VirtualServerRoute resources that correspond to community Ingress controller annotations for [rate limiting](#), [custom HTTP errors](#), a [custom default backend](#), and [URI rewriting](#).

COMMUNITY INGRESS CONTROLLER	NGINX INGRESS CONTROLLER
<pre> nginx.ingress.kubernetes.io/custom-http-errors: "code" nginx.ingress.kubernetes.io/default-backend: "default-svc" </pre>	<pre> errorPages: - codes: [code]   redirect:     code: 301     url: default-svc </pre>
<pre> nginx.ingress.kubernetes.io/limit-connections: "number" </pre>	<pre> http-snippets:       limit_conn_zone \$binary_remote_addr zone=zone_name:size; routes: - path: /path   location-snippets:       limit_conn zone_name number; </pre>

(continues)

COMMUNITY INGRESS CONTROLLER	NGINX INGRESS CONTROLLER
<code>nginx.ingress.kubernetes.io/limit-rate: "number"</code> <code>nginx.ingress.kubernetes.io/limit-rate-after: "number"</code>	<code>location-snippets:  </code> <code>    limit_rate number;</code> <code>    limit_rate_after number;</code>
<code>nginx.ingress.kubernetes.io/limit-rpm: "number"</code> <code>nginx.ingress.kubernetes.io/limit-burst-multiplier: "multiplier"</code>	<code>rateLimit:</code> <code>    rate: number/m</code> <code>    burst: number * multiplier</code> <code>    key: \${binary_remote_addr}</code> <code>    zoneSize: size</code>
<code>nginx.ingress.kubernetes.io/limit-rps: "number"</code> <code>nginx.ingress.kubernetes.io/limit-burst-multiplier: "multiplier"</code>	<code>rateLimit:</code> <code>    rate: number/s</code> <code>    burst: number * multiplier</code> <code>    key: \${binary_remote_addr}</code> <code>    zoneSize: size</code>
<code>nginx.ingress.kubernetes.io/limit-whitelist: "CIDR"</code>	<code>http-snippets</code> <code>server-snippets</code>
<code>nginx.ingress.kubernetes.io/rewrite-target: "URI"</code>	<code>rewritePath: URI</code>

As indicated in the table, as of this writing NGINX Ingress resources do not include fields that directly translate the following four community Ingress controller annotations, and you must use snippets. Direct support for the four annotations, using **Policy** objects, is planned for future releases of NGINX Ingress Controller.

- `nginx.ingress.kubernetes.io/limit-connections`
- `nginx.ingress.kubernetes.io/limit-rate`
- `nginx.ingress.kubernetes.io/limit-rate-after`
- `nginx.ingress.kubernetes.io/limit-whitelist`

MANIPULATING HTTP  
HEADERS IS USEFUL  
IN MANY USE CASES

Header Manipulation

Manipulating HTTP headers is useful in many use cases, as they contain additional information that is important and relevant for systems involved in an HTTP transaction. For example, the community Ingress controller supports enabling and setting **cross-origin resource sharing** (CORS) headers, which are used with AJAX applications, where front-end JavaScript code from a browser is connecting to a backend app or web server.

The table shows the fields in NGINX VirtualServer and VirtualServerRoute resources that correspond to community Ingress Controller annotations for **header manipulation**.

COMMUNITY INGRESS CONTROLLER	NGINX INGRESS CONTROLLER
<code>nginx.ingress.kubernetes.io/enable-cors: "true"</code> <code>nginx.ingress.kubernetes.io/cors-allow-credentials: "true"</code> <code>nginx.ingress.kubernetes.io/cors-allow-headers: "X-Forwarded-For"</code> <code>nginx.ingress.kubernetes.io/cors-allow-methods: "PUT, GET, POST, OPTIONS"</code> <code>nginx.ingress.kubernetes.io/cors-allow-origin: "*"</code> <code>nginx.ingress.kubernetes.io/cors-max-age: "seconds"</code>	<b>responseHeaders:</b> add: - name: Access-Control-Allow-Credentials value: "true" - name: Access-Control-Allow-Headers value: "X-Forwarded-For" - name: Access-Control-Allow-Methods value: "PUT, GET, POST, OPTIONS"  - name: Access-Control-Allow-Origin value: "*" - name: Access-Control-Max-Age value: "seconds"



THERE ARE OTHER PROXYING  
AND LOAD-BALANCING  
FUNCTIONALITIES YOU  
MIGHT WANT TO CONFIGURE

Other Proxying and Load Balancing Annotations

There are other proxying and load-balancing functionalities you might want to configure in NGINX Ingress Controller depending on the specific use case. These functionalities include setting load-balancing algorithms and timeouts and buffering settings for proxied connections.

The table shows the statements in the **upstream** field of NGINX VirtualServer and VirtualServerRoute resources that correspond to community Ingress Controller annotations for custom NGINX load balancing, proxy timeouts, proxy buffering, and routing connections to a service’s Cluster IP address and port.

COMMUNITY INGRESS CONTROLLER	NGINX INGRESS CONTROLLER
nginx.ingress.kubernetes.io/load-balance	lb-method
nginx.ingress.kubernetes.io/proxy-buffering	buffering
nginx.ingress.kubernetes.io/proxy-buffers-number nginx.ingress.kubernetes.io/proxy-buffer-size	buffers
nginx.ingress.kubernetes.io/proxy-connect-timeout	connect-timeout
nginx.ingress.kubernetes.io/proxy-next-upstream	next-upstream
nginx.ingress.kubernetes.io/proxy-next-upstream-timeout	next-upstream-timeout
nginx.ingress.kubernetes.io/proxy-read-timeout	read-timeout
nginx.ingress.kubernetes.io/proxy-send-timeout	send-timeout
nginx.ingress.kubernetes.io/service-upstream	use-cluster-ip

A SERVICE MESH IS  
PARTICULARLY USEFUL  
IN A STRICT ZERO-TRUST  
ENVIRONMENT

mTLS Authentication

As [previously noted](#), a service mesh is particularly useful in a strict zero-trust environment, where distributed applications inside a cluster communicate securely by mutually authenticating. What if we need to impose that same level of security on traffic entering and exiting the cluster?

We can configure mTLS authentication at the Ingress Controller layer so that the end systems of external connections authenticate each other by presenting a valid certificate.

The table shows the fields in NGINX Policy resources that correspond to community Ingress Controller annotations for [client certificate authentication](#) and [backend certificate authentication](#).

COMMUNITY INGRESS CONTROLLER	NGINX INGRESS CONTROLLER
nginx.ingress.kubernetes.io/auth-tls-secret: secretName nginx.ingress.kubernetes.io/auth-tls-verify-client: "on" nginx.ingress.kubernetes.io/auth-tls-verify-depth: "1"	<b>ingressMTLS:</b> clientCertSecret: secretName verifyClient: "on" verifyDepth: 1
nginx.ingress.kubernetes.io/proxy-ssl-secret: "secretName" nginx.ingress.kubernetes.io/proxy-ssl-verify: "on off" nginx.ingress.kubernetes.io/proxy-ssl-verify-depth: "1" nginx.ingress.kubernetes.io/proxy-ssl-protocols: "TLSv1.2" nginx.ingress.kubernetes.io/proxy-ssl-ciphers: "DEFAULT" nginx.ingress.kubernetes.io/proxy-ssl-name: "server-name" nginx.ingress.kubernetes.io/proxy-ssl-server-name: "on off"	<b>egressMTLS:</b> tlsSecret: secretName verifyServer: true false verifyDepth: 1 protocols: TLSv1.2 ciphers: DEFAULT sslName: server-name serverName: true false

Session Persistence (Exclusive to NGINX Plus)

The table shows the fields in NGINX Policy resources that are exclusive to the NGINX Ingress Controller based on NGINX Plus and correspond to community Ingress Controller annotations for [session persistence](#) (affinity).

COMMUNITY INGRESS CONTROLLER	NGINX INGRESS CONTROLLER
nginx.ingress.kubernetes.io/affinity: "cookie" nginx.ingress.kubernetes.io/session-cookie-name: "cookieName" nginx.ingress.kubernetes.io/session-cookie-expires: "x" nginx.ingress.kubernetes.io/session-cookie-path: "/route" nginx.ingress.kubernetes.io/session-cookie-secure: "true"	<b>sessionCookie:</b> enable: true name: cookieName path: /route expires: xh secure: true

## Option 2: Migrate Using the Kubernetes Ingress Resource

The second option for migrating from the community Ingress controller to NGINX Ingress Controller is to use only [annotations](#) and [ConfigMaps](#) in the standard Kubernetes Ingress resource and potentially rely on “[master/minion](#)”-style processing. This keeps all the configuration in the **Ingress** object.

**Note:** With this method, do not alter the **spec** field of the Ingress resource.

### Advanced Configuration with Annotations

The following table outlines the community Ingress controller annotations that correspond directly to annotations supported by NGINX Ingress Controller (for both the NGINX Open Source-based and NGINX Plus-based models).

COMMUNITY INGRESS CONTROLLER	NGINX INGRESS CONTROLLER	NGINX DIRECTIVE
<a href="#">nginx.ingress.kubernetes.io/configuration-snippet</a> :	<a href="#">nginx.org/location-snippets</a> :	N/A
<a href="#">nginx.ingress.kubernetes.io/load-balance</a> <sup>1</sup>	<a href="#">nginx.org/lb-method</a>	Default: <a href="#">random</a> two least_conn
<a href="#">nginx.ingress.kubernetes.io/proxy-buffering</a> : "on off"	<a href="#">nginx.org/proxy-buffering</a> : "True False"	<a href="#">proxy_buffering</a>
<a href="#">nginx.ingress.kubernetes.io/proxy-buffers-number</a> : "number" <a href="#">nginx.ingress.kubernetes.io/proxy-buffer-size</a> : "xk"	<a href="#">nginx.org/proxy-buffers</a> : "number 4k 8k" <a href="#">nginx.org/proxy-buffer-size</a> : "4k 8k"	<a href="#">proxy_buffers</a> <a href="#">proxy_buffer_size</a>
<a href="#">nginx.ingress.kubernetes.io/proxy-connect-timeout</a> : "seconds"	<a href="#">nginx.org/proxy-connect-timeout</a> : "secondss"	<a href="#">proxy_connect_timeout</a>
<a href="#">nginx.ingress.kubernetes.io/proxy-read-timeout</a> : "seconds"	<a href="#">nginx.org/proxy-read-timeout</a> : "secondss"	<a href="#">proxy_read_timeout</a>
<a href="#">nginx.ingress.kubernetes.io/proxy-send-timeout</a> : "seconds"	<a href="#">nginx.org/proxy-send-timeout</a> : "secondss"	<a href="#">proxy_send_timeout</a>
<a href="#">nginx.ingress.kubernetes.io/rewrite-target</a> : "URI"	<a href="#">nginx.org/rewrites</a> : "serviceName=svc rewrite=URI"	<a href="#">rewrite</a>
<a href="#">nginx.ingress.kubernetes.io/server-snippet</a> :	<a href="#">nginx.org/server-snippets</a> :	N/A
<a href="#">nginx.ingress.kubernetes.io/ssl-redirect</a> : "true false"	<a href="#">ingress.kubernetes.io/ssl-redirect</a> : "True False"	N/A <sup>2</sup>

1. The community Ingress controller uses Lua to implement some of its load-balancing algorithms. NGINX Ingress Controller doesn't have an equivalent for all of them.
2. Redirects HTTP traffic to HTTPS. The community Ingress controller implements this with Lua code, while NGINX Ingress Controller uses native NGINX **if** conditions.

The following table outlines the community Ingress controller annotations that correspond directly to annotations supported by the NGINX Ingress Controller based on NGINX Plus.

COMMUNITY INGRESS CONTROLLER	NGINX INGRESS CONTROLLER
<code>nginx.ingress.kubernetes.io/affinity: "cookie"</code> <code>nginx.ingress.kubernetes.io/session-cookie-name: "cookie_name"</code> <code>nginx.ingress.kubernetes.io/session-cookie-expires: "seconds"</code> <code>nginx.ingress.kubernetes.io/session-cookie-path: "/route"</code>	<code>nginx.com/sticky-cookie-services:</code> <code>"serviceName=example-svc cookie_name</code> <code>expires=time path=/route"</code>

**Note:** The NGINX Ingress Controller based on NGINX Plus has additional annotations for features that the community Ingress controller doesn't support at all, including [active health checks](#) and [authentication using JSON Web Tokens \(JWTs\)](#).

## Global Configuration with ConfigMaps

The following table maps community Ingress controller ConfigMap keys to their directly corresponding NGINX Ingress Controller ConfigMap keys. Note that a handful of ConfigMap key names are identical. Also, both the community Ingress controller and NGINX Ingress Controller have ConfigMaps keys that the other does not (not shown in the table).

COMMUNITY INGRESS RESOURCE	NGINX INGRESS CONTROLLER
<code>disable-access-log</code>	<code>access-log-off</code>
<code>error-log-level</code>	<code>error-log-level</code>
<code>hsts</code>	<code>hsts</code>
<code>hsts-include-subdomains</code>	<code>hsts-include-subdomains</code>
<code>hsts-max-age</code>	<code>hsts-max-age</code>
<code>http-snippet</code>	<code>http-snippets</code>
<code>keep-alive</code>	<code>keepalive-timeout</code>
<code>keep-alive-requests</code>	<code>keepalive-requests</code>
<code>load-balance</code>	<code>lb-method</code>
<code>location-snippet</code>	<code>location-snippets</code>
<code>log-format-escape-json: "true"</code>	<code>log-format-escaping: "json"</code>
<code>log-format-stream</code>	<code>stream-log-format</code>
<code>log-format-upstream</code>	<code>log-format</code>
<code>main-snippet</code>	<code>main-snippets</code>
<code>max-worker-connections</code>	<code>worker-connections</code>
<code>max-worker-open-files</code>	<code>worker-rlimit-nofile</code>

(continues)

COMMUNITY INGRESS RESOURCE	NGINX INGRESS CONTROLLER
<code>proxy-body-size</code>	<code>client-max-body-size</code>
<code>proxy-buffering</code>	<code>proxy-buffering</code>
<code>proxy-buffers-number: "number"</code> <code>proxy-buffer-size: "size"</code>	<code>proxy-buffers: number size</code>
<code>proxy-connect-timeout</code>	<code>proxy-connect-timeout</code>
<code>proxy-read-timeout</code>	<code>proxy-read-timeout</code>
<code>proxy-send-timeout</code>	<code>proxy-send-timeout</code>
<code>server-name-hash-bucket-size</code>	<code>server-names-hash-bucket-size</code>
<code>server-name-hash-max-size</code>	<code>server-names-hash-max-size</code>
<code>server-snippet</code>	<code>server-snippets</code>
<code>server-tokens</code>	<code>server-tokens</code>
<code>ssl-ciphers</code>	<code>ssl-ciphers</code>
<code>ssl-dh-param</code>	<code>ssl-dhparam-file</code>
<code>ssl-protocols</code>	<code>ssl-protocols</code>
<code>ssl-redirect</code>	<code>ssl-redirect</code>
<code>upstream-keepalive-connections</code>	<code>keepalive</code>
<code>use-http2</code>	<code>http2</code>
<code>use-proxy-protocol</code>	<code>proxy-protocol</code>
<code>variables-hash-bucket-size</code>	<code>variables-hash-bucket-size</code>
<code>worker-cpu-affinity</code>	<code>worker-cpu-affinity</code>
<code>worker-processes</code>	<code>worker-processes</code>
<code>worker-shutdown-timeout</code>	<code>worker-shutdown-timeout</code>

## CHAPTER SUMMARY

We defined what an *Ingress controller* and *service mesh* are and what they do, explained why it's beneficial to deploy them together, and showed how to install NGINX Ingress Controller and NGINX Service Mesh.

Let's summarize some of the key concepts from this chapter:

- An Ingress controller is a tightly integrated traffic-management solution for Kubernetes environments that bridges between applications in the Kubernetes cluster and clients on external networks.
- A service mesh is a layer of infrastructure that enables secure and high-performance communication between application services, while also providing visibility and insights.
- NGINX Service Mesh comes in handy when setting up zero-trust production environments, especially those with large-scale distributed app topologies.
- NGINX Service Mesh has two key components: the data plane and the control plane. The data plane (implemented as sidecar proxy instances) manages the traffic between instances, and its behavior is configured and controlled by the control plane.
- You can migrate from the community Ingress controller to NGINX Ingress Controller using either custom NGINX Ingress resources or the standard Kubernetes Ingress resource with annotations and ConfigMaps. The former option supports a broader set of networking capabilities and so is more suitable for production-grade Kubernetes environments.
- NGINX Ingress resources not only enable configuration of load balancing, but also provide additional customization, including circuit breaking, routing, header manipulation, mTLS authentication, and web application firewall (WAF).

## 2. Traffic Management Use Cases

In this chapter we show how to configure NGINX Ingress Controller for several traffic management use cases.

- [Load Balancing TCP and UDP Traffic](#)
- [Load Balancing TLS-Encrypted Traffic with TLS Passthrough](#)
- [Enabling Multi-Tenancy and Namespace Isolation](#)
- [Configuring Traffic Control and Traffic Splitting](#)
- [Chapter Summary](#)

### LOAD BALANCING TCP AND UDP TRAFFIC

NGINX Ingress Controller supports TCP and UDP load balancing, so you can use it to manage traffic for a wide range of apps and utilities based on those protocols, including:

- **MySQL, LDAP, and MQTT** – TCP-based apps used by many popular applications
- **DNS, syslog, and RADIUS** – UDP-based utilities used by edge devices and non-transactional applications

TCP and UDP load balancing with NGINX Ingress Controller is also an effective solution for distributing network traffic to Kubernetes applications in the following circumstances:

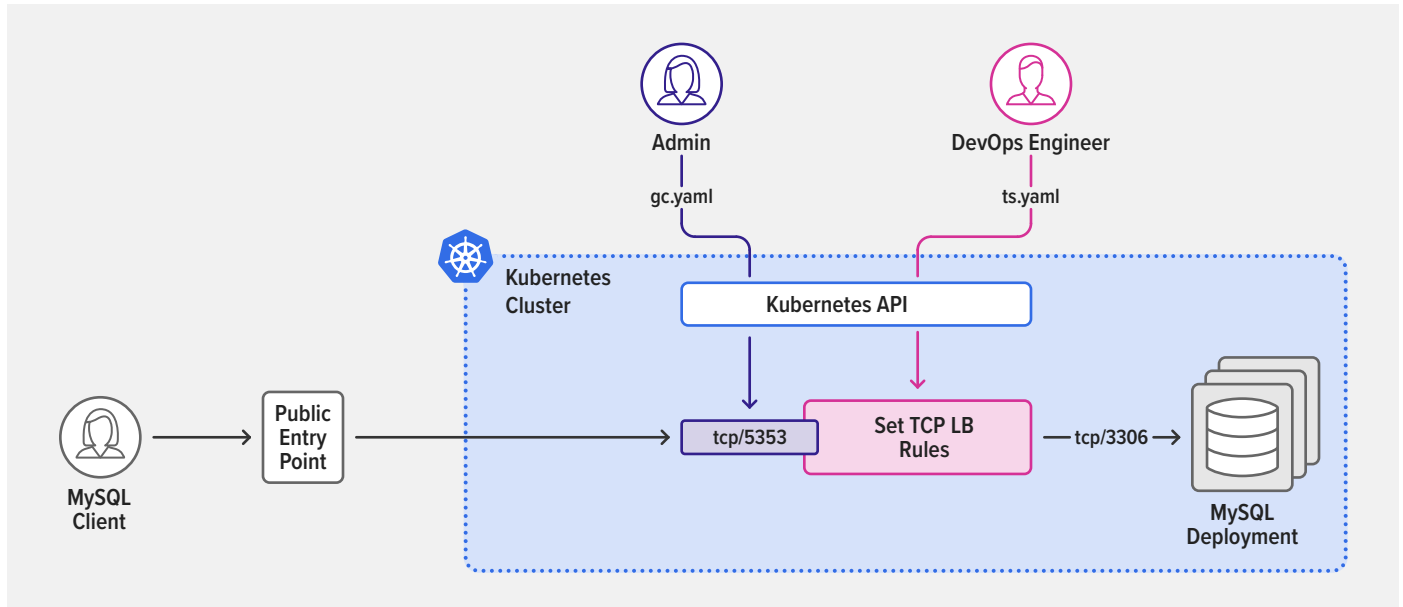
- You are using end-to-end encryption (EE2E) and having the application handle encryption and decryption rather than NGINX Ingress Controller
- You need high-performance load balancing for applications that are based on TCP or UDP
- You want to minimize the amount of change when migrating an existing network (TCP/UDP) load balancer to a Kubernetes environment

NGINX Ingress Controller comes with two NGINX Ingress resources that support TCP/UDP load balancing:

- **GlobalConfiguration** resources are typically used by cluster administrators to specify the TCP/UDP ports (**listeners**) that are available for use by DevOps teams. Note that each NGINX Ingress Controller deployment can only have one GlobalConfiguration resource.
- **TransportServer** resources are typically used by DevOps teams to configure TCP/UDP load balancing for their applications. NGINX Ingress Controller listens only on ports that were instantiated by the administrator in the GlobalConfiguration resource. This prevents conflicts between ports and provides an extra layer of security by ensuring DevOps teams expose to public external services only ports that the administrator has predetermined are safe.



The following diagram depicts a sample use case for the GlobalConfiguration and TransportServer resources. In **gc.yaml**, the cluster administrator defines TCP and UDP listeners in a GlobalConfiguration resource. In **ts.yaml**, a DevOps engineer references the TCP listener in a TransportServer resource that routes traffic to a MySQL deployment.



The GlobalConfiguration resource in **gc.yaml** defines two listeners: a UDP listener on port 514 for connection to a syslog service and a TCP listener on port 5353 for connection to a MySQL service.

```

1 apiVersion: k8s.nginx.org/v1alpha1
2 kind: GlobalConfiguration
3 metadata:
4   name: nginx-configuration
5   namespace: nginx-ingress
6 spec:
7   listeners:
8     - name: syslog-udp
9       port: 514
10      protocol: UDP
11     - name: mysql-tcp
12       port: 5353
13       protocol: TCP

```

Lines 6–8 of the TransportServer resource in **ts.yaml** reference the TCP listener defined in **gc.yaml** by name (**mysql-tcp**) and lines 9–14 define the routing rule that sends TCP traffic to the **mysql-db** upstream.

```
1 apiVersion: k8s.nginx.org/v1alpha1
2 kind: TransportServer
3 metadata:
4   name: mysql-tcp
5 spec:
6   listener:
7     name: mysql-tcp
8     protocol: TCP
9   upstreams:
10  - name: mysql-db
11    service: mysql
12    port: 3306
13  action:
14    pass: mysql-db
```

In this example, a DevOps engineer uses the MySQL client to verify that the configuration is working, as confirmed by the output with the list of tables in the **rawdata\_content\_schema** database inside the MySQL deployment.

```
$ echo "SHOW TABLES" | mysql -h <external_IP_address> -P <port>
-u <user> -p rawdata_content_schema
Enter Password: <password>
Tables_in_rawdata_content_schema
authors
posts
```

TransportServer resources for UDP traffic are configured similarly; for a complete example, see [Basic TCP/UDP Load Balancing](#) in the NGINX Ingress Controller repo on GitHub. Advanced NGINX users can extend the TransportServer resource with native NGINX configuration using the **stream-snippets** ConfigMap key, as shown in the [Support for TCP/UDP Load Balancing](#) example in the repo.

## LOAD BALANCING TLS-ENCRYPTED TRAFFIC WITH TLS PASSTHROUGH

Beyond TCP and UDP load balancing, you can use NGINX Ingress Controller for TLS Passthrough, which means load balancing encrypted TLS traffic on port 443 among different applications without decryption.

There are several other options for load balancing TLS-encrypted traffic with NGINX Ingress Controller:

- **TLS termination** – NGINX Ingress Controller terminates inbound TLS connections and routes them unencrypted to service endpoints, using either an NGINX VirtualServer or a standard Kubernetes Ingress resource. This option is not secure if hackers are able to access your private network and view traffic between NGINX Ingress Controller and backend services.
- **End-to-end encryption (E2EE)** – NGINX Ingress Controller terminates inbound TLS connections and establishes separate TLS connections with service endpoints, using a VirtualServer resource. This option adds an extra layer of security because data crossing the network is always encrypted. It also enables NGINX Ingress Controller to make additional Layer 7 routing decisions and transform the headers or body.
- **TCP load balancing with a TransportServer resource** – NGINX Ingress Controller load balances TCP connections that are bound to a specific port defined by the **listeners** field in a GlobalConfiguration resource. Connections can be either clear text or encrypted (in the second case the backend app decrypts them).

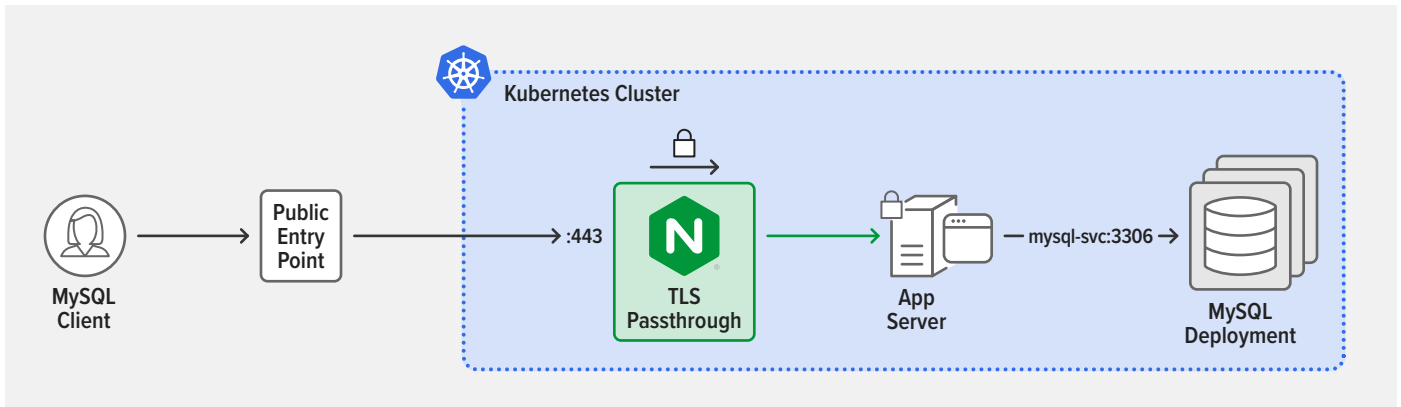
TLS PASSTHROUGH IS AN ALTERNATIVE TO E2EE WHEN ENCRYPTING TRAFFIC IN THE LOCAL AREA NETWORK IS IMPORTANT

TLS Passthrough is an alternative to E2EE when encrypting traffic in the local area network is important. The difference is that with TLS Passthrough NGINX Ingress Controller does not terminate inbound TLS connections, instead forwarding them encrypted to service endpoints. As with E2EE, the service endpoints do the decryption and so require the certificate and keys. A limitation with TLS Passthrough compared to E2EE is that NGINX Ingress Controller cannot make additional Layer 7 routing decisions or transform the headers and body because it doesn't decrypt traffic coming from the client.

TLS Passthrough is also an alternative to TCP load balancing with a TransportServer resource (the third option above). The difference is that with the third option the GlobalConfiguration resource can specify multiple ports for load balancing TCP/UDP connections, but only one TransportServer resource can reference the GlobalConfiguration resource. With TLS Passthrough, there is a single built-in listener that exposes port 443 only, but many TransportServer resources can reference the built-in listener for load balancing encrypted TLS connections.

TLS Passthrough is also extremely useful when the backend configures and performs the process for TLS verification of the client, and it is not possible to move authentication to NGINX Ingress Controller.

We can extend the example in [Load Balancing TCP and UDP Traffic](#) by adding TLS Passthrough and deploying an app server or reverse proxy that decrypts the secure traffic forwarded by NGINX Ingress Controller and connects to the MySQL deployment.



The following TransportServer resource for TLS Passthrough references a built-in listener named **tls-passthrough** and sets the protocol to **TLS\_PASSTHROUGH** (lines 7–8). This exposes port 443 on NGINX Ingress Controller for load balancing TLS-encrypted traffic. Users can establish secure connections with the application workload by accessing the hostname **app.example.com** (line 9), which resolves to NGINX Ingress Controller's public entry point. NGINX Ingress Controller passes the TLS-secured connections to the **secure-app** upstream for decryption (lines 10–15).

```
1 apiVersion: k8s.nginx.org/v1alpha1
2 kind: TransportServer
3 metadata:
4   name: secure-app
5 spec:
6   listener:
7     name: tls-passthrough
8     protocol: TLS_PASSTHROUGH
9   host: app.example.com
10  upstreams:
11    - name: secure-app
12      service: secure-app
13      port: 8443
14  action:
15    pass: secure-app
```

For more information about features you can configure in TransportServer resources, see the [NGINX Ingress Controller documentation](#).

AS ORGANIZATIONS SCALE UP, DEVELOPMENT AND OPERATIONAL WORKFLOWS IN KUBERNETES GET MORE COMPLEX

NGINX INGRESS CONTROLLER SUPPORTS MULTIPLE MULTI-TENANCY MODELS

## ENABLING MULTI-TENANCY AND NAMESPACE ISOLATION

As organizations scale up, development and operational workflows in Kubernetes get more complex. It's generally more cost-effective – and can be more secure – for teams to share Kubernetes clusters and resources, rather than each team getting its own cluster. But there can be critical damage to your deployments if teams don't share those resources in a safe and secure manner or hackers exploit your configurations.

Multi-tenancy practices and namespace isolation at the network and resource level help teams share Kubernetes resources safely. You can also significantly reduce the magnitude of breaches by isolating applications on a per-tenant basis. This method helps boost resiliency because only subsections of applications owned by specific teams can be compromised, while systems providing other functionalities remain intact.

NGINX Ingress Controller supports multiple multi-tenancy models, but we see two primary patterns. The **infrastructure service provider pattern** typically includes multiple NGINX Ingress Controller deployments with physical isolation, while the **enterprise pattern** typically uses a shared NGINX Ingress Controller deployment with namespace isolation. In this section we explore the enterprise pattern in depth; for information about running multiple NGINX Ingress Controllers see our [documentation](#).

### Delegation with NGINX Ingress Controller

NGINX Ingress Controller supports both the standard Kubernetes Ingress resource and custom NGINX Ingress resources, which enable both more sophisticated traffic management and delegation of control over configuration to multiple teams. The custom resources are [VirtualServer](#), [VirtualServerRoute](#), [GlobalConfiguration](#), [TransportServer](#), and [Policy](#).

With NGINX Ingress Controller, cluster administrators use [VirtualServer](#) resources to provision Ingress domain (hostname) rules that route external traffic to backend applications, and [VirtualServerRoute](#) resources to delegate management of specific URLs to application owners and DevOps teams.

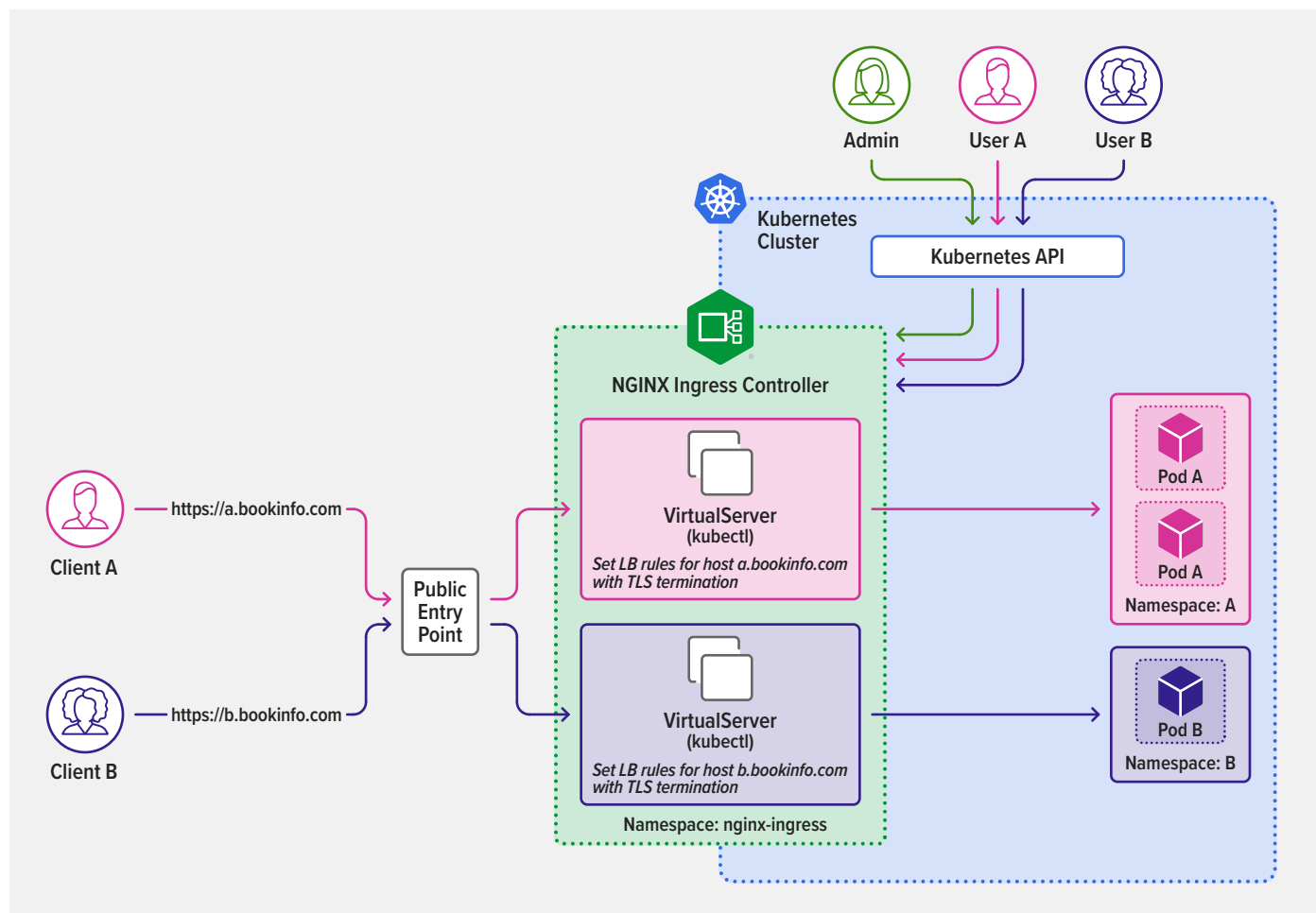
There are two models you can choose from when implementing multi-tenancy in your Kubernetes cluster: [full self-service](#) and [restricted self-service](#).

IN A FULL SELF-SERVICE  
MODEL, ADMINISTRATORS  
ARE NOT INVOLVED IN  
DAY-TO-DAY CHANGES TO  
NGINX INGRESS CONTROLLER  
CONFIGURATION

## Implementing Full Self-Service

In a full self-service model, administrators are not involved in day-to-day changes to NGINX Ingress Controller configuration. They are responsible only for deploying NGINX Ingress Controller and the Kubernetes **Service** which exposes the deployment externally. Developers then deploy applications within an assigned namespace without involving the administrator. Developers are responsible for managing TLS secrets, defining load-balancing configuration for domain names, and exposing their applications by creating VirtualServer or standard **Ingress** resources.

To illustrate this model, we replicate the **bookinfo** application with two subdomains, **a.bookinfo.com** and **b.bookinfo.com**, as depicted in the following diagram. Once the administrator installs and deploys NGINX Ingress Controller in the **nginx-ingress** namespace (highlighted in green), teams DevA (pink) and DevB (purple) create their own VirtualServer resources and deploy applications isolated within their namespaces (A and B respectively).



Teams DevA and DevB set Ingress rules for their domains to route external connections to their applications.

Team DevA applies the following VirtualServer resource to expose applications for the **a.bookinfo.com** domain in the **A** namespace.

```
1 apiVersion: k8s.nginx.org/v1
2 kind: VirtualServer
3 metadata:
4   name: bookinfo
5   namespace: A
6 spec:
7   host: a.bookinfo.com
8   upstreams:
9     - name: productpageA
10      service: productpageA
11      port: 9080
12   routes:
13     - path: /
14       action:
15         pass: productpageA
```

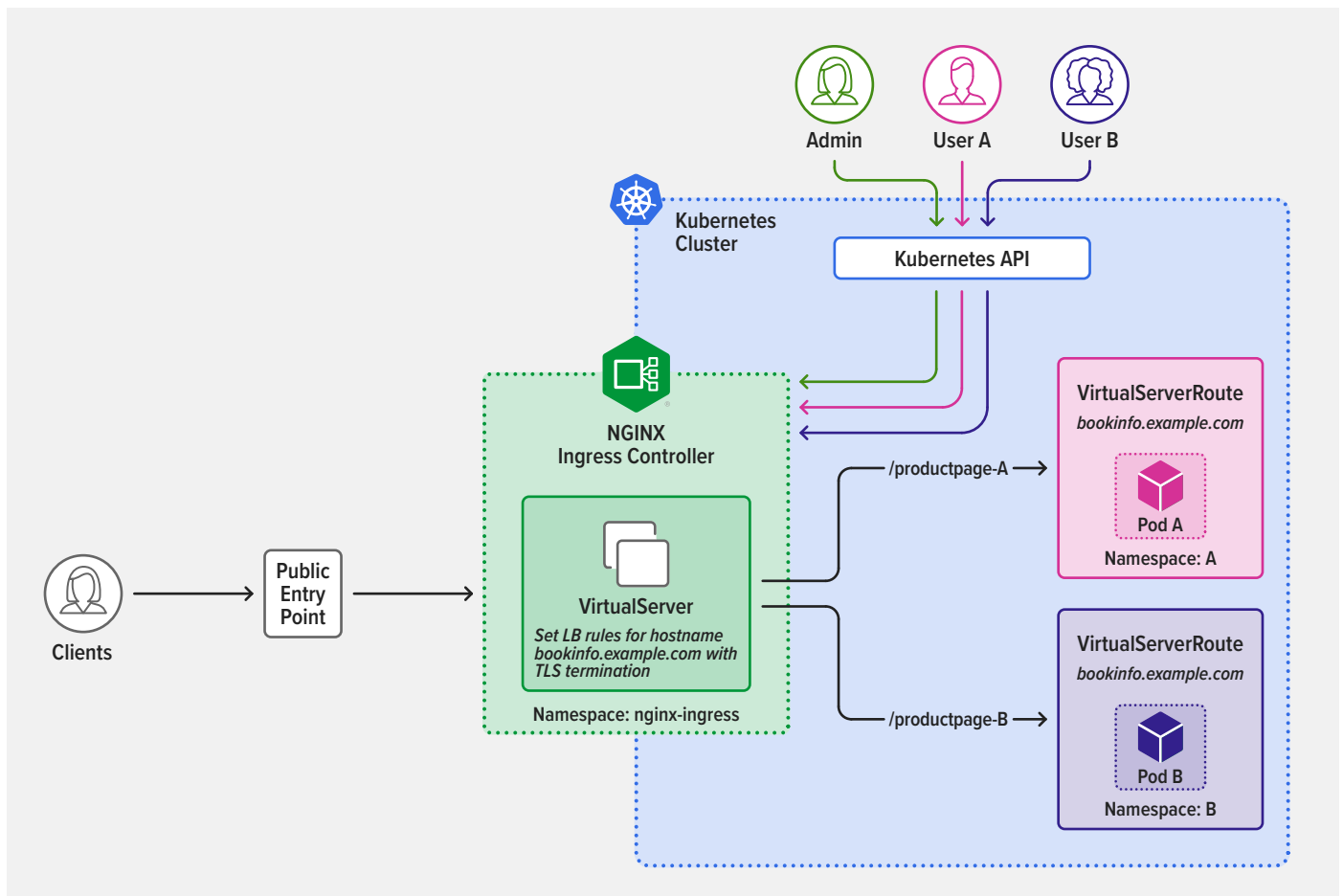
Similarly, team DevB applies the following VirtualServer resource to expose applications for the **b.bookinfo.com** domain in the **B** namespace.

```
1 apiVersion: k8s.nginx.org/v1
2 kind: VirtualServer
3 metadata:
4   name: bookinfo
5   namespace: B
6 spec:
7   host: b.bookinfo.com
8   upstreams:
9     - name: productpageB
10      service: productpageB
11      port: 9080
12   routes:
13     - path: /
14       action:
15         pass: productpageB
```



## Implementing Restricted Self-Service

In a restricted self-service model, administrators configure `VirtualServer` resources to route traffic entering the cluster to the appropriate namespace, but delegate configuration of the applications in the namespaces to the responsible development teams. Each such team is responsible only for its application subroute as instantiated in the `VirtualServer` resource and uses `VirtualServerRoute` resources to define traffic rules and expose application subroutes within its namespace.



As illustrated in the diagram, the cluster administrator installs and deploys NGINX Ingress Controller in the **nginx-ingress** namespace (highlighted in green), and defines a `VirtualServer` resource that sets path-based rules referring to `VirtualServerRoute` resource definitions.

This VirtualServer resource definition sets two path-based rules that refer to VirtualServerRoute resource definitions for two subroutes, **/productpage-A** and **/productpage-B**.

```
1  apiVersion: k8s.nginx.org/v1
2  kind: VirtualServer
3  metadata:
4    name: example
5  spec:
6    host: bookinfo.example.com
7    routes:
8      - path: /productpage-A
9        route: A/ingress
10     - path: /productpage-B
11       route: B/ingress
```

The developer teams responsible for the apps in namespaces **A** and **B** then define VirtualServerRoute resources to expose application subroutes within their namespaces. The teams are isolated by namespace and restricted to deploying application subroutes set by VirtualServer resources provisioned by the administrator:

- Team DevA (pink in the diagram) applies the following VirtualServerRoute resource to expose the application subroute rule set by the administrator for **/productpage-A**.

```
1  apiVersion: k8s.nginx.org/v1
2  kind: VirtualServerRoute
3  metadata:
4    name: ingress
5    namespace: A
6  spec:
7    host: bookinfo.example.com
8    upstreams:
9      - name: productpageA
10        service: productpageA-svc
11        port: 9080
12    subroutes:
13      - path: /productpage-A
14        action:
15          pass: productpageA
```

- Team DevB (purple) applies the following VirtualServerRoute resource to expose the application subroute rule set by the administrator for **/productpage-B**.

```
1  apiVersion: k8s.nginx.org/v1
2  kind: VirtualServerRoute
3  metadata:
4    name: ingress
5    namespace: B
6  spec:
7    host: bookinfo.example.com
8    upstreams:
9      - name: productpageB
10        service: productpageB-svc
11        port: 9080
12    subroutes:
13      - path: /productpage-B
14        action:
15          pass: productpageB
```

For more information about features you can configure in VirtualServer and VirtualServerRoute resources, see the [NGINX Ingress Controller documentation](#).

**Note:** You can use [mergeable Ingress types](#) to configure cross-namespace routing, but in a restricted self-service delegation model that approach has three downsides compared to VirtualServer and VirtualServerRoute resources:

1. It is less secure.
2. As your Kubernetes deployment grows becomes larger and more complex, it becomes increasingly prone to accidental modifications, because mergeable Ingress types do not prevent developers from setting Ingress rules for hostnames within their namespace.
3. Unlike VirtualServer and VirtualServerRoute resources, mergeable Ingress types don't enable the primary ("master") Ingress resource to control the paths of "minion" Ingress resources.

## Leveraging Kubernetes RBAC in a Restricted Self-Service Model

You can use Kubernetes role-based access control (RBAC) to regulate a user's access to namespaces and NGINX Ingress resources based on the roles assigned to the user.

For instance, in a restricted self-service model, only administrators with special privileges can safely be allowed to access VirtualServer resources – because those resources define the entry point to the Kubernetes cluster, misuse can lead to system-wide outages.

Developers use VirtualServerRoute resources to configure Ingress rules for the application routes they own, so administrators set RBAC policies that allow developers to create only those resources. They can even restrict that permission to specific namespaces if they need to regulate developer access even further.

YOU CAN USE KUBERNETES  
ROLE-BASED ACCESS  
CONTROL (RBAC) TO  
REGULATE A USER'S ACCESS

## NGINX POLICY RESOURCES ARE ANOTHER TOOL FOR ENABLING DISTRIBUTED TEAMS TO CONFIGURE KUBERNETES

In a full self-service model, developers can safely be granted access to VirtualServer resources, but again the administrator might restrict that permission to specific namespaces.

For more information on RBAC authorization, see the [Kubernetes documentation](#).

### Adding Policies

NGINX Policy resources are another tool for enabling distributed teams to configure Kubernetes in multi-tenancy deployments. Policy resources enable functionalities like authentication using [OAuth](#) and [OpenID Connect](#) (OIDC), rate limiting, and web application firewall (WAF). Policy resources are referenced in VirtualServer and VirtualServerRoute resources to take effect in the Ingress configuration.

For instance, a team in charge of identity management in a cluster can define [JSON Web Token](#) (JWT) or OIDC policies like this one (defined in **Identity-Security/okta-oidc-policy.yaml**) for using Okta as the OIDC identity provider (IdP), which we discuss in detail in [Chapter 4](#).

```
1 apiVersion: k8s.nginx.org/v1
2 kind: Policy
3 metadata:
4   name: okta-oidc-policy
5 spec:
6   oidc:
7     clientID: client-id
8     clientSecret: okta-oidc-secret
9     authEndpoint: https://your-okta-domain/oauth2/v1/authorize
10    tokenEndpoint: https://your-okta-domain/oauth2/v1/token
11    jwksURI: https://your-okta-domain/oauth2/v1/keys
```

[View on GitHub](#)

NetOps and DevOps teams can use VirtualServer or VirtualServerRoute resources to reference those policies, as in this example (**Identity-Security/okta-oidc-bookinfo-vs.yaml**).

```
1 apiVersion: k8s.nginx.org/v1
2 kind: VirtualServer
3 metadata:
4   name: bookinfo-vs
5 spec:
6   host: bookinfo.example.com
7   tls:
8     secret: bookinfo-secret
9   upstreams:
10  - name: backend
11    service: productpage
12    port: 9080
```

(continues)

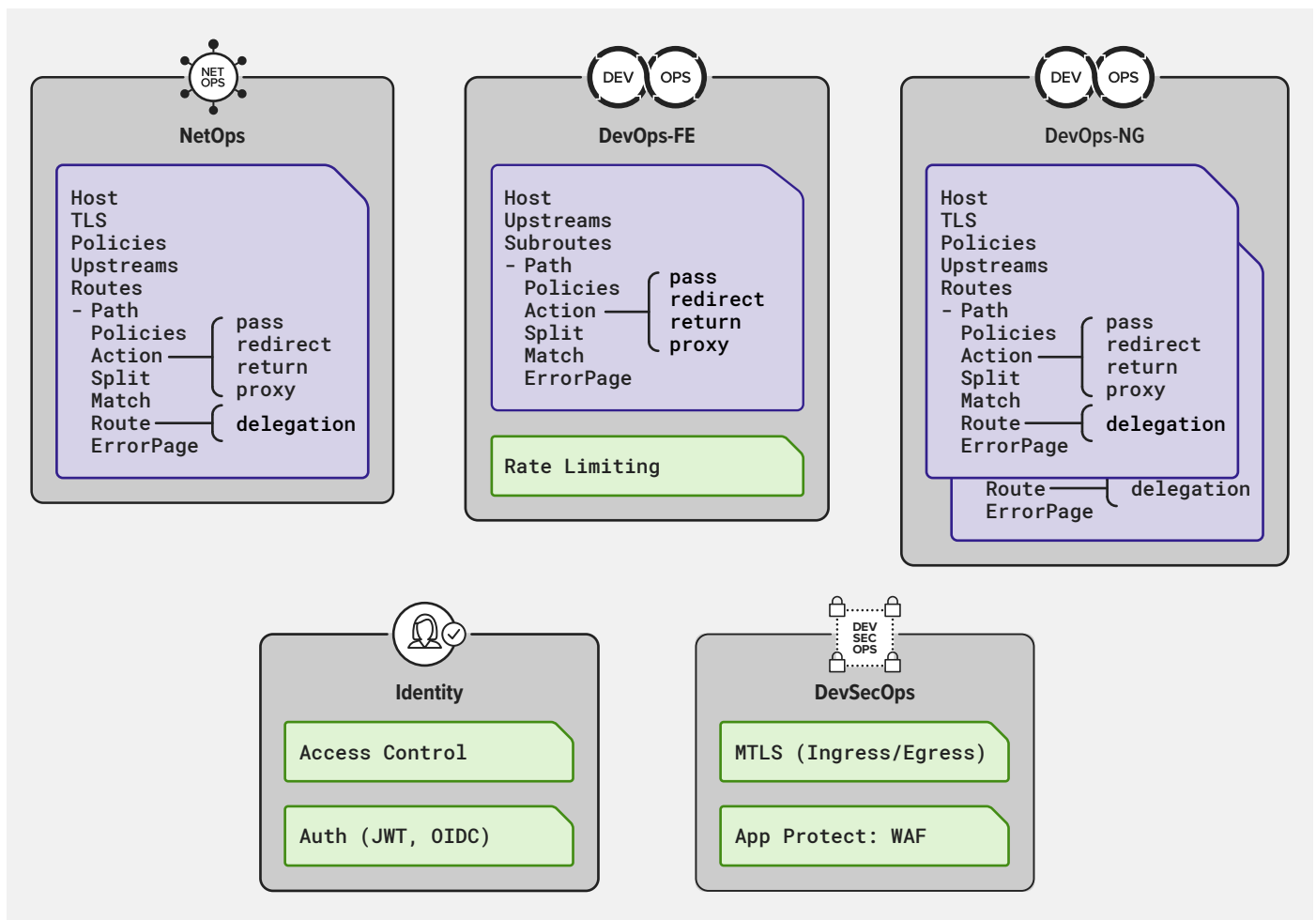
```

13 routes:
14   - path: /
15     policies:
16       - name: okta-oidc-policy
17     action:
18       pass: backend

```

[View on GitHub](#)

Together, the NGINX Policy, VirtualServer, and VirtualServerRoute resources enable distributed configuration architectures, where administrators can easily delegate configuration to other teams. Teams can assemble modules across namespaces and configure the NGINX Ingress Controller with sophisticated use cases in a secure, scalable, and manageable fashion.



For more information about Policy resources, see the [NGINX Ingress Controller documentation](#).

## CONFIGURING TRAFFIC CONTROL AND TRAFFIC SPLITTING

**“Art and science have their meeting point in method”**

– Edward G. Bulwer-Lytton

Traffic control and traffic splitting are two key traffic-management approaches that have become critical in modern application topologies.

This section uses a sample application called **bookinfo**, originally created by Istio, to illustrate how traffic control and traffic splitting affect application behavior. Some of the use cases leverage features that are available only when you deploy both NGINX Service Mesh and the NGINX Ingress Controller based on NGINX Plus, but many are also available with just the NGINX Ingress Controller based on NGINX Open Source.

We have prepared a GitHub repo that includes all the files for deploying the **bookinfo** app and implementing the sample use cases in this section as well as [Chapter 4](#). To get started, see [Deploying the Sample Application](#).

### Why Is Traffic Management So Vital?

Customer satisfaction and “always on” accessibility of resources are paramount for most companies delivering services online. Loss of customers, loss of revenue **up to \$550,000 for each hour of downtime**, and loss of employee productivity are all factors that directly hurt not only the bottom line but also the reputation of the company. If a company isn’t successfully using modern app-delivery technologies and approaches to manage its online traffic, customers are quick to react on social media, and you just don’t want to be known as “that company”.

Traffic-management strategy is therefore a critical piece of planning and delivering a modern application architecture. It requires a change of approach in how traffic is treated, such as controlling services rather than packets, or adapting traffic-management rules dynamically via the Kubernetes API. Whether you’re limiting the number of requests to your app to avoid cascade failures or testing a new app service for stability with real-world traffic loads, managing traffic without downtime is a science, an art, and most certainly a method . . . well two, actually.

TRAFFIC-MANAGEMENT  
STRATEGY IS A CRITICAL  
PIECE OF PLANNING AND  
DELIVERING A MODERN  
APPLICATION ARCHITECTURE

TRAFFIC CONTROL AND TRAFFIC SPLITTING ARE BOTH ESSENTIAL FOR MAXIMIZING APPLICATION PERFORMANCE

## How Do I Pick a Traffic Control or Traffic Splitting Method?

Traffic control and traffic splitting are both essential for maximizing application performance, but the method to choose depends on your goals:

- To protect services from being overwhelmed with requests, use **rate limiting** (traffic control)
- To prevent cascading failure, use **circuit breaking** (traffic control)
- To upgrade to a new application version without downtime, use **blue-green deployment** (traffic splitting)
- To test how a new application version handles load by gradually increasing the amount of traffic directed to it, use a **canary release** (traffic splitting)
- To determine which version of an application users prefer, use **A/B testing** (traffic splitting)
- To expose a new application or feature only to a defined set of users, use **debug routing** (traffic splitting)

You can implement all of these methods with NGINX Ingress Controller and NGINX Service Mesh, configuring robust traffic routing and splitting policies in seconds.

## When Do I Use NGINX Ingress Controller vs. NGINX Service Mesh?

Both NGINX Ingress Controller and NGINX Service Mesh help you implement robust traffic control and traffic splitting in seconds. However, they are not equally suitable for all use cases and app architectures. As a general rule of thumb:

- NGINX Ingress Controller is appropriate when there is no service-to-service communication in your cluster or apps are direct endpoints from NGINX Ingress Controller.
- NGINX Service Mesh is appropriate when you need to control east-west traffic within the cluster, for example when testing and upgrading individual microservices.

## Deploying the Sample Application

In this section you use NGINX Ingress Controller to expose the sample **bookinfo** application, using the `VirtualServer` resource defined in **Traffic-Management/bookinfo-vs.yaml**.

**VirtualServer** and **VirtualServerRoute** are custom NGINX resources, introduced in NGINX Ingress Controller 1.5. They enable use cases – including traffic splitting and advanced content-based routing – that are possible with the standard **Kubernetes Ingress resource** only by using **annotations**, which are limited in scope.

BOTH NGINX INGRESS CONTROLLER AND NGINX SERVICE MESH HELP YOU IMPLEMENT ROBUST TRAFFIC CONTROL AND TRAFFIC SPLITTING IN SECONDS

Line 8 of **bookinfo-vs.yaml** references the Kubernetes Secret for the **bookinfo** app, which is defined in **Traffic-Management/bookinfo-secret.yaml**. For the purposes of the sample application, the Secret is self-signed; in a production environment we strongly recommend that you use real keys and certificates generated by a Certificate Authority.

Lines 9–16 of **bookinfo-vs.yaml** define the routing rule that directs requests for **bookinfo.example.com** to the **productpage** service.

```
1 apiVersion: k8s.nginx.org/v1
2 kind: VirtualServer
3 metadata:
4   name: bookinfo-vs
5 spec:
6   host: bookinfo.example.com
7   tls:
8     secret: bookinfo-secret
9   upstreams:
10  - name: backend
11    service: productpage
12    port: 9080
13  routes:
14  - path: /
15    action:
16      pass: backend
```

[View on GitHub](#)

Here is **bookinfo-secret.yaml**, with a self-signed key and certificate for the purposes of this example:

```
1 apiVersion: v1
2 kind: Secret
3 metadata:
4   name: bookinfo-secret
5 type: kubernetes.io/tls
6 data:
7   tls.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0...
8   tls.key: LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVk...
```

[View on GitHub](#)



Deploy the sample application:

1. Load the key and certificate, and activate the VirtualServer resource for **bookinfo**:

```
$ kubectl apply -f ./Traffic-Management/bookinfo-secret.yaml
secret/bookinfo-secret created
$ kubectl apply -f ./Traffic-Management/bookinfo-vs.yaml
virtualserver.k8s.nginx.org/bookinfo-vs created
```

2. To enable external clients to access resources in the cluster via NGINX Ingress Controller, you need to advertise a public IP address as the entry point for the cluster.

In on-premises deployments, this is the IP address of the hardware or software load balancer you configured in [Step 5 of Installation and Deployment Instructions for NGINX Ingress Controller](#).

In cloud deployments, this is the public IP address of the **LoadBalancer** service you created in Step 5.

Obtain the public IP address of the **LoadBalancer** service (the output is spread across two lines for legibility):

```
$ kubectl get svc nginx-ingress -n nginx-ingress
NAME                TYPE           CLUSTER-IP ...
nginx-ingress       LoadBalancer  203.0.x.5   ...

... EXTERNAL-IP          PORT(S)              AGE
... a309c13t-2.elb.amazonaws.com  80:32148/TCP,443:32001/TCP  1h
```

For Azure and Google Cloud Platform, the public IP address is reported in the **EXTERNAL-IP** field. For AWS, however, the public IP address of the Network Load Balancer (NLB) is not static and the **EXTERNAL-IP** field instead reports its DNS name, as in the sample output above. To find the public IP address, run the **nslookup** command (here the public address is 203.0.x.66):

```
$ nslookup a309c13t-2.elb.amazonaws.com
Server:      198.51.100.1
Address:     198.51.100.1#53

Non-authoritative answer:
Name:        a309c13t-2.elb.amazonaws.com
Address:     203.0.x.66
```

3. Edit the local `/etc/hosts` file, adding an entry for **bookinfo.example.com** with the public IP address. For example:

```
203.0.x.66    bookinfo.example.com
```

4. Deploy the sample **bookinfo** application, defined in **Traffic-Management/bookinfo.yaml**:

```
$ kubectl apply -f ./Traffic-Management/bookinfo.yaml
service/details created
serviceaccount/bookinfo-details created
deployment.apps/details-v1 created
service/ratings created
serviceaccount/bookinfo-ratings created
deployment.apps/ratings-v1 created
service/reviews created
serviceaccount/bookinfo-reviews created
deployment.apps/reviews-v1 created
service/productpage created
serviceaccount/bookinfo-productpage created
deployment.apps/productpage-v1 created
```

5. Verify that all pods have a sidecar injected, as indicated by **nginx-mesh-sidecar** in the **CONTAINERS** field:

```
$ kubectl get pods -o custom-columns=NAME:.metadata.name,CONTAINERS:.spec.containers[*].name
NAME                                CONTAINERS
details-v1-847c7999fb-9vvbv         details,nginx-mesh-sidecar
productpage-v1-764fd8c446-kxskn     productpage,nginx-mesh-sidecar
ratings-v1-7c46bc6f4d-vpf74        ratings,nginx-mesh-sidecar
reviews-v1-988d86446-zvwvc         reviews-v1,nginx-mesh-sidecar
```

6. Deploy a **bash** container (defined in **Traffic-Management/bash.yaml**) in the cluster:

```
$ kubectl apply -f ./Traffic-Management/bash.yaml
deployment.apps/bash created
```

7. To verify that the **bookinfo** app is running, fetch its main page from within the **bash** container:

```
$ kubectl exec deploy/bash -it -c bash -- bash
$ curl -I -k http://productpage:9080/productpage?u=normal
HTTP/1.1 200 OK
Server: nginx/1.21.5
Date: Day, DD Mon HH:MM:SS YYYY TZ
Content-Type: text/html; charset=utf-8
Content-Length: 4183
Connection: keep-alive
X-Mesh-Request-ID: d8d3418d7b701363e6830155b520bf3e
```

8. To verify that external clients can access the app by connecting to NGINX Ingress Controller, navigate to **https://bookinfo.example.com/** in a browser.

In this guide we **deploy NGINX Service Mesh in mTLS strict mode**. If you use **off** or **permissive** mode, an alternative way to verify that clients can access the app is to run this command to port-forward the product page to your local environment, and then open **http://localhost:9080/** in your browser.

```
$ kubectl port-forward svc/productpage 9080
Forwarding from 127.0.0.1:9080 -> 9080
Forwarding from [::1]:9080 -> 9080
```

## Configuring Traffic Control

Traffic control refers to the act of regulating the flow of traffic to your apps in terms of source, volume, and destination. It's a necessity when running Kubernetes in production because it allows you to protect your infrastructure and apps from attacks and traffic spikes. In simple terms, it's always an advantage to regulate the traffic coming to your app or service. Traffic control incorporates two techniques:

- **Rate limiting**
- **Circuit breaking**

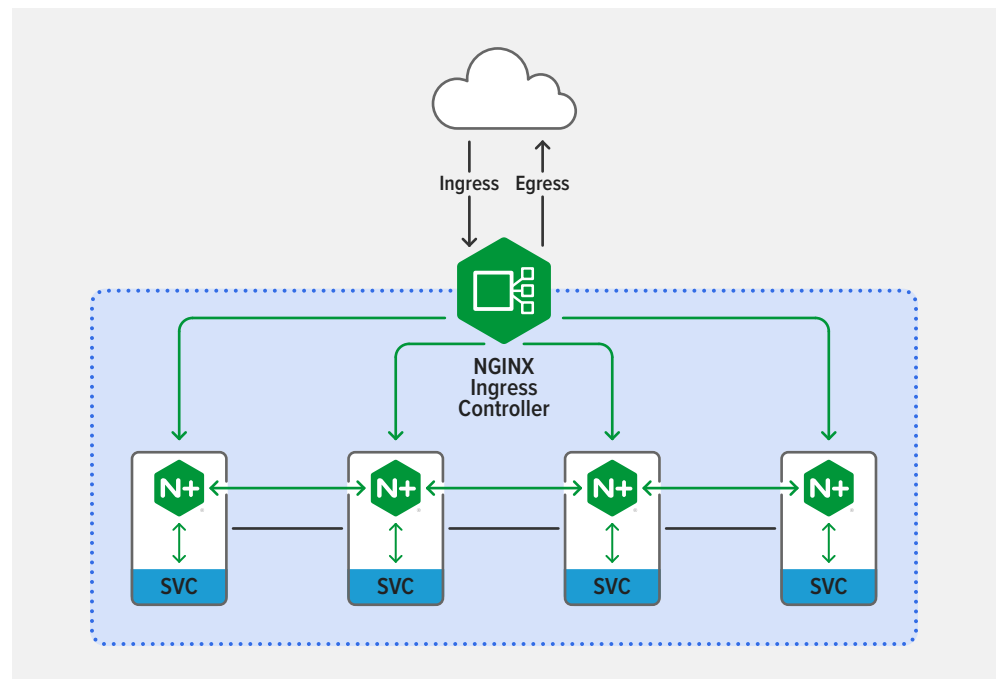
### Configuring Rate Limiting

An app that's receiving more HTTP requests than it can handle is likely to crash. And it completely doesn't matter whether the requests are intentionally harmful (a brute-force attempt to guess passwords, say) or good news in disguise (eager customers flocking to your Cyber Monday sale) – either way, your services are at stake. Rate limiting protects against overload by limiting the number of requests your app accepts from each client within a defined period of time.

AN APP THAT'S RECEIVING  
MORE HTTP REQUESTS  
THAN IT CAN HANDLE  
IS LIKELY TO CRASH

### Activating Client Rate Limiting with NGINX Ingress Controller

In non-Kubernetes environments, and many Kubernetes environments as well, excessive requests from external clients represent the biggest threat. In this case, it makes sense to implement rate limiting with NGINX Ingress Controller, because it can classify clients on many criteria and rate limit based on those criteria. (If you need to limit requests from other services in the Kubernetes cluster, apply rate limiting with NGINX Service Mesh as described in [Activating Interservice Rate Limiting with NGINX Service Mesh](#).)



A simple way to rate-limit all clients is to create an NGINX Ingress Controller [Policy](#) resource and apply it to `VirtualServer` and `VirtualServerRoute` resources.

A `VirtualServer` or `VirtualServerRoute` can reference multiple rate-limiting policies. When there is more than one policy, NGINX Ingress Controller configures NGINX to use all of them, with the most restrictive one setting the actual enforced rate. The values of the `dryRun`, `logLevel`, and `rejectCode` parameters in the first referenced policy are inherited by the additional policies.

1. Create a rate-limiting policy. This sample policy (**Traffic-Management/nic-rate-limit-policy.yaml**) accepts 1 request per second from each IP address; additional requests arriving within that second are rejected with a **503 (Service Unavailable)** response code:

```
1  apiVersion: k8s.nginx.org/v1
2  kind: Policy
3  metadata:
4    name: nic-rate-limit-policy
5  spec:
6    rateLimit:
7      rate: 1r/s
8      zoneSize: 10M
9      key: ${binary_remote_addr}
10     logLevel: warn
11     rejectCode: 503
12     dryRun: false
```

[View on GitHub](#)

2. To enable NGINX Ingress Controller to apply the policy, include a reference to it in **Traffic-Management/bookinfo-vs-rate-limit.yaml**:

```
17  policies:
18    - name: nic-rate-limit-policy
```

[View on GitHub](#)

3. Apply the changes to the **bookinfo** application you exposed in [Deploying the Sample Application](#):

```
$ kubectl apply -f ./Traffic-Management/nic-rate-limit-policy.yaml
policy.k8s.nginx.org/nic-rate-limit-policy created

$ kubectl apply -f ./Traffic-Management/bookinfo-vs-rate-limit.yaml
virtualserver.k8s.nginx.org/bookinfo-vs configured
```

4. To verify that the policy is in effect, use the **Traffic-Management/generate-traffic.sh** script, which generates and directs traffic to the Ingress Controller:

```
1 #!/bin/bash
2
3 # get IP address of NGINX Ingress Controller
4 IC_IP=$(kubectl get svc -n nginx-ingress -o jsonpath="{.items[0].
  status.loadBalancer.ingress[0].ip}")
5 [ -z "$IC_IP" ] && IC_IP=$(kubectl get svc -n nginx-ingress -o
  jsonpath="{.items[0].status.loadBalancer.ingress[0].hostname}")
6
7 # send 300 requests to bookinfo
8 for i in $(seq 1 300);
9 do curl -I -k https://$IC_IP:443/productpage?u=normal -H "host:
  bookinfo.example.com";
10 done
```

[View on GitHub](#)

5. Run the script on your local machine. Requests that exceed the rate limit get rejected with error code **503 (Service Unavailable)** as for the second request in this example:

```
$ bash generate-traffic.sh
HTTP/1.1 200 OK
Server: nginx/1.21.5
Date: Day, DD Mon HH:MM:SS YYYY TZ
Content-Type: text/html; charset=utf-8
Content-Length: 4183
Connection: keep-alive
X-Mesh-Request-ID: c9df5e030d3c6871745527ea93e403b8

HTTP/1.1 503 Service Unavailable
Server: nginx/1.21.5
Date: Day, DD Mon HH:MM:SS YYYY TZ
Content-Type: text/html
Content-Length: 197
Connection: keep-alive
```

TO PRESERVE A  
SATISFACTORY USER  
EXPERIENCE, YOU OFTEN  
NEED TO MAKE RATE-  
LIMITING POLICIES  
MORE FLEXIBLE

### *Allowing Bursts of Requests with NGINX Ingress Controller*

To preserve a satisfactory user experience, you often need to make rate-limiting policies more flexible, for example to accommodate “bursty” apps. Such apps tend to send multiple requests in rapid succession followed by a period of inactivity. If the rate-limiting policy is set such that it always rejects bursty traffic, many legitimate client requests don’t succeed.

To avoid this, instead of immediately rejecting requests that exceed the limit, you can buffer them in a queue and service them in a timely manner. The **burst** field in a **rateLimit** policy defines how many requests a client can make in excess of the rate, and requests that exceed **burst** are rejected immediately. We can also control how quickly the queued requests are sent. The **noDelay** field sets the number of queued requests (which must be smaller than **burst**) NGINX Ingress Controller proxies to the app without delay; the remaining queued requests are delayed to comply with the defined rate limit.

1. Create a policy that allows bursting. In this update (**Traffic-Management/nic-rate-limit-policy-burst.yaml**) to the policy introduced in the [previous section](#), the combination of the **rate** of 10 requests per second (line 7), the **noDelay** parameter (line 13), and the **burst** value of 10 (line 14) means that NGINX Ingress Controller immediately proxies up to 20 requests per second for each client.

```
1  apiVersion: k8s.nginx.org/v1
2  kind: Policy
3  metadata:
4    name: nic-rate-limit-policy
5  spec:
6    rateLimit:
7      rate: 10r/s
8      zoneSize: 10M
9      key: ${binary_remote_addr}
10     logLevel: warn
11     rejectCode: 503
12     dryRun: false
13     noDelay: true
14     burst: 10
```

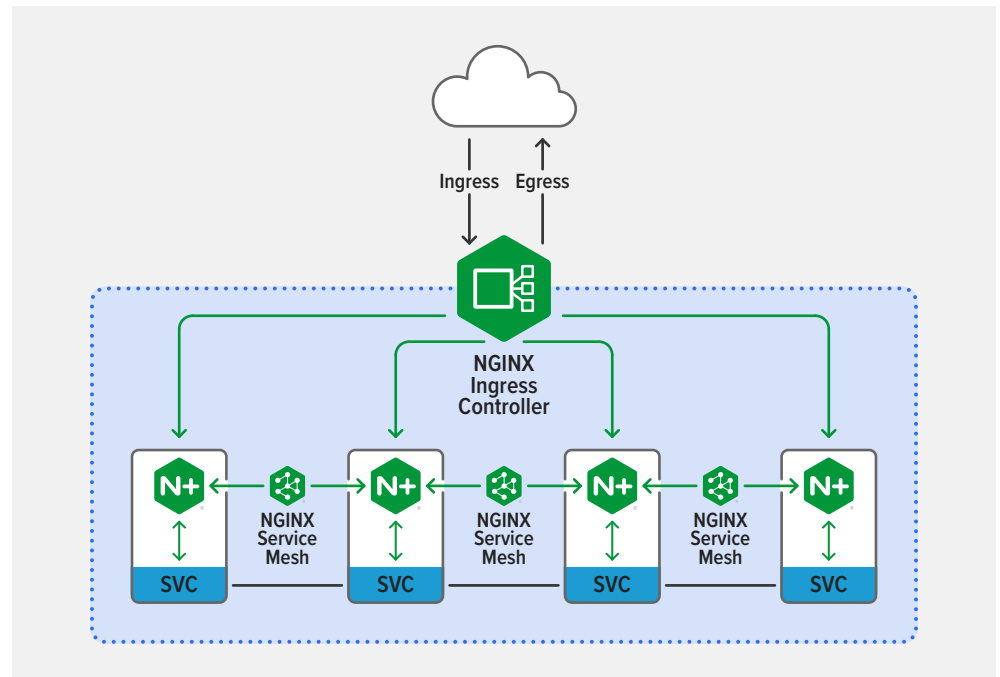
[View on GitHub](#)

2. Verify that the new rate limit is in effect by running the **Traffic-Management/generate-traffic.sh** script as in [Step 4](#) of the previous section.

Rate limiting with NGINX Ingress Controller is implemented with the NGINX [Limit Requests](#) module. For a more detailed explanation of how rate limiting works, see the [NGINX blog](#).

### Activating Interservice Rate Limiting with NGINX Service Mesh

Traffic between services in a Kubernetes cluster doesn't necessarily fall under the scope of the NGINX Ingress Controller, making NGINX Service Mesh the more appropriate way to rate limit it.



As with NGINX Ingress Controller, you create a rate-limiting policy to have NGINX Service Mesh limit the number of requests an app accepts from each service within a defined period of time.

The NGINX Service Mesh **RateLimit** object takes different parameters from the NGINX Ingress Controller **rateLimit** policy:

- **destination** – Service for which the rate limit is being set
- **sources** – Group of client services subject to the rate limit
- **rate** – Allowed number of requests per second or minute from each client



This policy definition in **Traffic-Management/nsm-rate-limit.yaml** sets a limit of 10 requests per minute, or 1 every 6 seconds (line 16).

```
1 apiVersion: specs.smi.nginx.com/v1alpha1
2 kind: RateLimit
3 metadata:
4   name: nsm-rate-limit
5   namespace: default
6 spec:
7   destination:
8     kind: Service
9     name: productpage
10    namespace: default
11   sources:
12   - kind: Deployment
13     name: bash
14     namespace: default
15   name: 10rm
16   rate: 10r/m
17   burst: 0
18   delay: nodelay
```

[View on GitHub](#)

1. Apply the policy:

```
$ kubectl apply -f ./Traffic-Management/nsm-rate-limit.yaml
ratelimit.specs.smi.nginx.com/nsm-rate-limit created
```

2. To test the policy, initialize a **bash** container:

```
$ kubectl exec deploy/bash -it -c bash -- bash
```

3. Run the following `curl` command several times in rapid succession in the `bash` container to verify the rate limit is being imposed. As shown for the second request, the error code **503 (Service Unavailable)** indicates the request was rejected because it exceeded the limit.

```
$ curl -I -k http://productpage:9080/productpage?u=normal
HTTP/1.1 200 OK
Server: nginx/1.21.5
Date: Day, DD Mon HH:MM:SS YYYY TZ
Content-Type: text/html; charset=utf-8
Content-Length: 5183
Connection: keep-alive
X-Mesh-Request-ID: 27c4030698264b7136f2218002d9933f

$ curl -I -k http://productpage:9080/productpage?u=normal
HTTP/1.1 503 Service Unavailable
Server: nginx/1.21.5
Date: Day, DD Mon HH:MM:SS YYYY TZ
Content-Type: text/html
Content-Length: 198
Connection: keep-alive
```

## Configuring Circuit Breaking

When a service is unavailable or experiencing high latency, it can take a long time for incoming requests to time out and for the clients to receive an error response. Such long timeouts can potentially cause a cascading failure, in which the outage of one service leads to timeouts at other services and ultimately failure of the application as a whole.

The circuit breaker pattern helps prevent cascading failure by:

1. Detecting and isolating the service that failed
2. Sending a predefined response to clients without waiting for a timeout
3. Redirecting failed responses and timeouts to an external or backup service (a failsafe) that handles requests differently

To enable a circuit breaker with NGINX Service Mesh, you set a limit on the number of errors that occur within a defined period. When the number of failures exceeds the limit, the circuit breaker starts returning an error response to clients as soon as a request arrives. You can also define a custom informational page to return when your service is not functioning correctly or under maintenance, as detailed in [Returning a Custom Page](#).

## THE USE OF A CIRCUIT BREAKER CAN IMPROVE THE PERFORMANCE OF AN APPLICATION

The circuit breaker continues to intercept and reject requests for the defined amount of time before allowing a limited number of requests to pass through as a test. If those requests are successful, the circuit breaker stops throttling traffic. Otherwise, the clock resets and the circuit breaker again rejects requests for the defined time.

The use of a circuit breaker can improve the performance of an application by eliminating calls to a failed component that would otherwise time out or cause delays, and it can often mitigate the impact of a failed non-essential component.

The following lines in **Traffic-Management/broken-deployment.yaml** simulate a service failure in which the release of version 2 of the **reviews** service is followed by a command (lines 44–45) that causes the associated pod to crash and start returning error code **502 (Bad Gateway)**.

```
18 apiVersion: apps/v1
19 kind: Deployment
25 spec:
31   template:
32     metadata:
33       labels:
34         app: reviews-v2
34         version: v2
36   spec:
37     serviceName: bookinfo-reviews
38     containers:
39     - name: reviews-v2
40       image: docker.io/istio/examples-bookinfo-reviews-v2:1.15.0
41       imagePullPolicy: IfNotPresent
42       ports:
43       - containerPort: 9080
44       command: ["/bin/sh", "-c"]
45       args: ["timeout --signal=SIGINT 5 /opt/ibm/wlp/bin/server run
              defaultServer"]
```

[View on GitHub](#)

1. Apply the failure simulation. In the **STATUS** column of the output from **kubectl get pods**, the value **CrashLoopBackOff** for the **reviews-v2** pod indicates that it is repeatedly starting up and crashing. When you send a **curl** request to the **reviews** service, you get a **502 (Bad Gateway)** error response:

```
$ kubectl apply -f ./Traffic-Management/broken-deployment.yaml
service/reviews configured
deployment.apps/reviews-v2 created
```

*(continues)*

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
bash-5bbdcb458d-tbzrb	2/2	Running	0	42h
details-v1-847c7999fb-47fsw	2/2	Running	0	9d
maintenance-v1-588566b84f-57wrr	2/2	Running	0	3h53m
productpage-v1-764fd8c446-px5p9	2/2	Running	0	4m47s
ratings-v1-7c46bc6f4d-qjqff	2/2	Running	0	9d
reviews-v1-76ddd45467-vvw56	2/2	Running	0	9d
reviews-v2-7fb86bc686-5jkhq	1/2	CrashLoopBackOff	9	2m

```

$ kubectl exec deploy/bash -it -c bash -- bash
$ curl -I -k http://reviews:9080/health
HTTP/1.1 502 Bad Gateway
Server: nginx/1.21.5
Date: Day, DD Mon HH:MM:SS YYYY TZ
Content-Type: text/html
Content-Length: 158
Connection: keep-alive

```

2. Configure an NGINX Service Mesh **CircuitBreaker** object to route traffic away from the **reviews-v2** pod and to **reviews-v1** instead, preventing clients from receiving the **502** error. This configuration (defined in **Traffic-Management/nsm-circuit-breaker.yaml**) trips the circuit when there are more than 3 errors within 30 seconds.

```

1 apiVersion: specs.smi.nginx.com/v1alpha1
2 kind: CircuitBreaker
3 metadata:
4   name: nsm-circuit-breaker
5   namespace: default
6 spec:
7   destination:
8     kind: Service
9     name: reviews
10    namespace: default
11   errors: 3
12   timeoutSeconds: 30
13   fallback:
14     service: default/reviews-v1
15     port: 9080

```

[View on GitHub](#)

3. Apply the circuit breaker. The `curl` request to the `reviews` service succeeds with status code **200 (OK)** even though the output from `kubectl get pods` shows that the `reviews-v2` pod is still down:

```
$ kubectl apply -f ./Traffic-Management/nsm-circuit-breaker.yaml
service/reviews-backup created
circuitbreaker.specs.smi.nginx.com/nsm-circuit-breaker created

$ kubectl get pods
NAME                                READY STATUS    RESTARTS AGE
bash-5bbdcb458d-tbzrb              2/2   Running      0      42h
details-v1-847c7999fb-47fsw        2/2   Running      0      9d
maintenance-v1-588566b84f-57wrr    2/2   Running      0      3h53m
productpage-v1-764fd8c446-px5p9    2/2   Running      0      4m47s
ratings-v1-7c46bc6f4d-qjqff        2/2   Running      0      9d
reviews-v1-76ddd45467-vvw56        2/2   Running      0      9d
reviews-v2-7fb86bc686-5jkhq        1/2   CrashLoopBackOff 9      26m

$ kubectl exec deploy/bash -it -c bash -- bash
$ curl -I -k http://reviews:9080/health
HTTP/1.1 200 OK
Server: nginx/1.21.5
Date: Day, DD Mon HH:MM:SS YYYY TZ
Content-Type: text/html; charset=utf-8
Content-Length: 4063
Connection: keep-alive
X-Mesh-Request-ID: 574e93b8b1736f7dcfd866bca547d370
```

**Note:** The NGINX Service Mesh circuit breaker relies on passive health checks to monitor the status of service endpoints. With the configuration shown above, it marks the broken deployment as unhealthy when more than 3 requests issued from the `bash` container trigger an error response during a 30-second period.

When implementing the circuit breaker pattern with the NGINX Ingress Controller based on NGINX Plus, you can use active health checks instead. For more information, see the [NGINX blog](#).

## A CIRCUIT BREAKER WITH A BACKUP SERVICE IMPROVES THE USER EXPERIENCE

### *Returning a Custom Page*

A circuit breaker with a backup service improves the user experience by reducing the number of error messages clients see, but it doesn't eliminate such messages entirely. We can enhance the user experience further by returning a response that's more helpful than an error code when failure occurs.

Consider, for example, an application with a web or mobile interface that presents a list of ancillary items – comments on an article, recommendations, advertisements, and so on – in addition to the information specifically requested by clients. If the Kubernetes service that generates this list fails, by default it returns error code **502 (Bad Gateway)**. You can create a more appropriate response for the circuit breaker to send, such as a redirect to a URL that explains the failure.

The following lines in **Traffic-Management/bookinfo-vs-circuit-breaker.yaml** amend the `VirtualServer` configuration for the **bookinfo** app with a redirect to a “Sorry, we’re doing maintenance” page for error code **502**:

```
17  errorPages:
18    - codes: [502]
19      redirect:
20        code: 301
21        url: https://cdn.f5.com/maintenance/f5.com/SorryPage.html
```

[View on GitHub](#)

1. Apply the new configuration that performs the redirect:

```
$ kubectl apply -f ./Traffic-Management/bookinfo-vs-circuit-
breaker.yaml
virtualserver.k8s.nginx.org/bookinfo-vs configured
```

2. Turn off the **productpage-v1** service to cause a failure. Now a **curl** request to the **bookinfo** app results in a redirect (code **301 Moved Permanently**) to the maintenance page rather than a **502** error:

```
$ kubectl scale --replicas=0 deployment/productpage-v1
deployment.apps/productpage-v1 scaled

$ curl -k -I https://bookinfo.example.com
HTTP/1.1 301 Moved Permanently
Server: nginx/1.21.5
Date: Day, DD Mon HH:MM:SS YYYY TZ
Content-Type: text/html
Content-Length: 169
Connection: keep-alive
Location: https://cdn.f5.com/maintenance/f5.com/SorryPage.html
```

## Configuring Traffic Splitting

Traffic splitting is a technique which directs different proportions of incoming traffic to two versions of a backend app running simultaneously in an environment (usually the current production version and an updated version). One use case is testing the stability and performance of the new version as you gradually increase the amount of traffic to it. Another is seamlessly updating the app version by changing routing rules to transfer all traffic at once from the current version to the new version. Traffic splitting techniques include:

- Blue-green deployment
- Canary deployment
- A/B testing
- Debug routing

## Generating Cluster-Internal Traffic to Split

All of the examples in this section use a script called **traffic.sh** to generate the traffic split by the various configurations. Here you define the script within the **bash** container in which you generate traffic in the following sections.

1. Connect to a **bash** container:

```
$ kubectl exec deploy/bash -it -c bash -- bash
```

TRAFFIC SPLITTING DIRECTS  
DIFFERENT PROPORTIONS OF  
INCOMING TRAFFIC TO TWO  
VERSIONS OF A BACKEND APP

BLUE-GREEN DEPLOYMENT  
ENABLES YOU TO SWITCH  
USER TRAFFIC TO A  
NEW VERSION SUCH THAT  
USERS DON'T EXPERIENCE  
ANY DOWNTIME

2. Create the script:

```
$ cat << 'EOF' > traffic.sh
#!/bin/bash
for i in $(seq 1 1000);
do
curl http://reviews:9080/health;
curl -H "User-Agent: Firefox" http://reviews:9080/health;
sleep 0.1;
done
EOF
```

## Implementing Blue-Green Deployment

Suppose you have a new version of an app and are ready to deploy it. Blue-green deployment enables you to switch user traffic to a new version such that users don't experience any downtime and might not even notice that a new version is servicing their requests. To do this, keep the old version (blue) running while simultaneously deploying the new version (green) alongside in the same environment. Then change the routing rule to direct traffic to the new version.

### *Blue-Green Deployment with NGINX Service Mesh*

The following example implements a blue-green deployment that directs traffic to the new version of a sample service (**reviews-v2-1**) without removing the old one (**reviews-v1**). Keeping **reviews-v1** in place means it's easy to roll back if **reviews-v2-1** has problems or fails.

1. Generate traffic to the **reviews** service (if necessary, repeat the instructions in [Generating Cluster-Internal Traffic to Split](#) to create the script):

```
$ bash traffic.sh
```

2. The **nginx-meshctl top** command shows that all traffic is flowing to the **reviews-v1** service:

```
$ nginx-meshctl top
Deployment      Incoming Success  Outgoing Success  NumRequests
bash           100.00%
reviews-v1     100.00%
```



3. Define a **TrafficSplit** object in **Traffic-Management/nsm-blue-green.yaml** that points all traffic to the **reviews-v2-1** service:

```
1 apiVersion: split.smi-spec.io/v1alpha3
2 kind: TrafficSplit
3 metadata:
4   name: reviews
5 spec:
6   service: reviews
7   backends:
8     - service: reviews-v1
9       weight: 0
10    - service: reviews-v2-1
11      weight: 100
```

[View on GitHub](#)

4. Apply the **TrafficSplit** resource:

```
$ kubectl apply -f ./Traffic-Management/nsm-blue-green.yaml
trafficsplit.split.smi-spec.io/reviews created
service/reviews configured
service/reviews-v1 created
service/reviews-v2-1 created
deployment.apps/reviews-v2-1 created
```

5. Run **nginx-meshctl top** again to verify that all traffic is flowing to the **reviews-v2-1** service:

```
$ nginx-meshctl top
Deployment      Incoming Success  Outgoing Success  NumRequests
reviews-v2-1   100.00%
bash            100.00%
               129
```

### Blue-Green Deployment with NGINX Ingress Controller

Blue-green deployment with NGINX Ingress Controller is similar to NGINX Service Mesh, except that traffic originates from outside the cluster rather than from inside the **bash** container. (You can generate that traffic with the **Traffic-Management/generate-traffic.sh** script; see [Step 4](#) in *Activating Client Rate Limiting with NGINX Ingress Controller*.)

1. Define a VirtualServer resource in **Traffic-Management/bookinfo-vs-blue-green.yaml** that directs almost all traffic to the **reviews-v2-1** service:

```
1  apiVersion: k8s.nginx.org/v1
2  kind: VirtualServer
3  metadata:
4    name: reviews
5  spec:
6    host: reviews.example.com
7    upstreams:
8      - name: reviews-v1
9        service: reviews-v1
10       port: 9080
11      - name: reviews-v2-1
12        service: reviews-v2-1
13        port: 9080
14    routes:
15      - path: /
16        splits:
17          - weight: 1
18            action:
19              pass: reviews-v1
20          - weight: 99
21            action:
22              pass: reviews-v2-1
```

[View on GitHub](#)

2. Apply the resource:

```
$ kubectl apply -f ./Traffic-Management/bookinfo-vs-blue-green.yaml
```

## A CANARY DEPLOYMENT MINIMIZES POSSIBLE NEGATIVE EFFECTS ON THE USER BASE

### Implementing Canary Deployment

After deploying a new version of an app, it's often prudent to test how it performs before switching over 100% of user traffic to it. A canary deployment minimizes possible negative effects on the user base as a whole, by initially directing a small percentage of user traffic to the new version and the bulk of traffic to the existing stable version. If the updated version successfully handles traffic from the test group, you increase the proportion of traffic directed to it. Usually the increase is done incrementally with a test period for each increment, but you can also switch over all users at once, as in a blue-green deployment. Like blue-green deployment, canary deployment lets you quickly revert to the old version in case of issues with the new version.

#### *Canary Deployment with NGINX Service Mesh*

The following example uses NGINX Service Mesh to implement a canary deployment that directs 10% of traffic to the new version (**reviews-v2-1**) of the sample service and the rest to old one (**reviews-v1**).

1. Define a **TrafficSplit** object in **Traffic-Management/nsm-canary.yaml** that leverages the **weight** field to set the traffic proportions:

```
1  apiVersion: split.smi-spec.io/v1alpha3
2  kind: TrafficSplit
3  metadata:
4    name: reviews
5  spec:
6    service: reviews
7    backends:
8      - service: reviews-v1
9        weight: 90
10     - service: reviews-v2-1
11       weight: 10
```

[View on GitHub](#)

2. Generate traffic to the **reviews** service (if necessary, repeat the instructions in [Generating Cluster-Internal Traffic to Split](#) to create the script):

```
$ bash traffic.sh
```

3. Apply the traffic-splitting configuration:

```
$ kubectl apply -f ./Traffic-Management/nsm-canary.yaml
trafficsplit.split.smi-spec.io/reviews configured
service/reviews-v3 created
deployment.apps/reviews-v3 created
```

4. Run the `nginx-meshctl top` command to verify the split: about 10% of requests (16) are going to the `reviews-v2-1` service:

```
$ nginx-meshctl top
Deployment      Incoming Success  Outgoing Success  NumRequests
bash           100.00%
reviews-v1     100.00%           164
reviews-v2-1   100.00%           16
```

### Canary Deployment with NGINX Ingress Controller

Canary deployment with NGINX Ingress Controller is similar to NGINX Service Mesh, except that traffic originates from outside the cluster rather than from inside the `bash` container. (You can generate that traffic with the **Traffic-Management/generate-traffic.sh** script; see [Step 4](#) in *Activating Client Rate Limiting with NGINX Ingress Controller*.)

1. Define a `VirtualServer` resource in **Traffic-Management/bookinfo-vs-canary.yaml** that splits traffic between the `reviews-v1` and `reviews-v2-1` services:

```
1  apiVersion: k8s.nginx.org/v1
2  kind: VirtualServer
3  metadata:
4    name: reviews
5  spec:
6    host: reviews.example.com
7    upstreams:
8      - name: reviews-v1
9        service: reviews-v1
10       port: 9080
11      - name: reviews-v2-1
12        service: reviews-v2-1
13        port: 9080
14    routes:
15      - path: /
16        splits:
17          - weight: 90
18            action:
19              pass: reviews-v1
20          - weight: 10
21            action:
22              pass: reviews-v2-1
```

[View on GitHub](#)

2. Apply the resource:

```
$ kubectl apply -f ./Traffic-Management/bookinfo-vs-canary.yaml
```

YOU'VE JUST RELEASED  
A PREVIEW VERSION OF AN  
APP AND WANT TO EVALUATE  
ITS PERFORMANCE AND  
USER APPEAL

## Implementing A/B Testing

Suppose you've just released a preview version of an app and want to evaluate its performance and user appeal before swapping it in for the current app version. Evaluation criteria can include feedback directly from end users (perhaps in response to a survey) and performance metrics (latency percentile distribution, requests or connections per second, and so on). This is a common use case for A/B testing, which – like blue-green deployment – directs users to different versions.

### *A/B Testing with NGINX Service Mesh*

The following example uses NGINX Service Mesh to split traffic between two app versions based on which browser the client is using.

1. Define an **HTTPRouteGroup** object in **Traffic-Management/nsm-ab-testing.yaml** that groups users based on the value of the HTTP **User-Agent** header, assigning users of the Firefox browser to the test group.

```
1  apiVersion: specs.smi-spec.io/v1alpha3
2  kind: HTTPRouteGroup
3  metadata:
4    name: reviews-testgroup-rg
5    namespace: default
6  spec:
7    matches:
8      - name: test-users
9      headers:
10       - user-agent: ".*Firefox.*"
```

[View on GitHub](#)

2. Define a **TrafficSplit** object in **nsm-ab-testing.yaml** that directs traffic from users in the test group to the **reviews-v3** version of the app:

```
12 apiVersion: split.smi-spec.io/v1alpha3
13 kind: TrafficSplit
14 metadata:
15   name: reviews
16 spec:
17   service: reviews
18   backends:
19     - service: reviews-v1
20       weight: 0
21     - service: reviews-v3
22       weight: 100
23   matches:
24     - kind: HTTPRouteGroup
25       name: reviews-testgroup-rg
```

[View on GitHub](#)

3. Generate traffic to the **reviews** service (if necessary, repeat the instructions in [Generating Cluster-Internal Traffic to Split](#) to create the script):

```
$ bash traffic.sh
```

Recall that **traffic.sh** includes this command to generate traffic from the test group by setting the **User-Agent** header to **Firefox**:

```
curl -H "User-Agent: Firefox" http://reviews:9080/health;
```

4. Apply the configuration and check the traffic split:

```
$ kubectl apply -f ./Traffic-Management/nsm-ab-testing.yaml
httprouategroup.specs.smi-spec.io/reviews-testgroup-rg created
trafficsplit.split.smi-spec.io/reviews configured

$ nginx-meshctl top
```

Deployment	Incoming Success	Outgoing Success	NumRequests
bash	100.00%		65
reviews-v3	100.00%		65
reviews-v1	100.00%		65

### A/B Testing with NGINX Ingress Controller

A/B testing with NGINX Ingress Controller is similar to NGINX Service Mesh, except that traffic originates from outside the cluster rather than from inside the **bash** container. (You can generate that traffic with the **Traffic-Management/generate-traffic.sh** script; see [Step 4](#) in *Activating Client Rate Limiting with NGINX Ingress Controller*.)

1. Define a VirtualServer resource in **Traffic-Management/bookinfo-vs-ab-testing.yaml** that directs traffic from Firefox users to the **reviews-v3** service and from users of other browsers to **reviews-v1**:

```
1  apiVersion: k8s.nginx.org/v1
2  kind: VirtualServer
3  metadata:
4    name: reviews
5  spec:
6    host: reviews.example.com
7    upstreams:
8      - name: reviews-v1
9        service: reviews-v1
10       port: 9080
11      - name: reviews-v2-1
12        service: reviews-v2-1
13        port: 9080
14      - name: reviews-v3
15        service: reviews-v3
16        port: 9080
17    routes:
18      - path: /
19        matches:
20          - conditions:
21            - header: "user-agent"
22              value: ".*Firefox.*"
23            action:
24              pass: reviews-v3
25          action:
26            pass: reviews-v1
```

[View on GitHub](#)

2. Apply the resource:

```
$ kubectl apply -f ./Traffic-Management/bookinfo-vs-ab-testing.yaml
```

YOU'VE ADDED A NEW  
FEATURE TO THE SAMPLE  
REVIEWS SERVICE AND WANT  
TO TEST HOW THE FEATURE  
PERFORMS IN PRODUCTION

## Implementing Debug Routing

Suppose you've added a new feature to the sample **reviews** service and want to test how the feature performs in production. This is a use case for debug routing, which restricts access to the service to defined group of users, based on Layer 7 attributes such as a session cookie, session ID, or group ID. This makes the updating process safer and more seamless.

### *Debug Routing with NGINX Service Mesh*

The following example uses NGINX Service Mesh to direct traffic from users who have a session cookie to a development version of the app.

1. Define an **HTTPRouteGroup** object in **Traffic-Management/nsm-debug-routing.yaml** that identifies **GET** requests for **/api/reviews** that include a session cookie.

```
1  apiVersion: specs.smi-spec.io/v1alpha3
2  kind: HTTPRouteGroup
3  metadata:
4    name: reviews-session-cookie
5    namespace: default
6  spec:
7    matches:
8      - name: get-session-cookie
9        headers:
10         - Cookie: "session_token=xxx-yyy-zzz"
11      - name: get-api-requests
12        pathRegex: "/api/reviews"
13        methods:
14          - GET
```

[View on GitHub](#)

2. Define a **TrafficSplit** object in **nsm-debug-routing.yaml** that directs requests identified by the **HTTPRouteGroup** object to the development version of the service, **reviews-v3**.

```
16 apiVersion: split.smi-spec.io/v1alpha3
17 kind: TrafficSplit
18 metadata:
19   name: reviews
20 spec:
21   service: reviews
22   backends:
23     - service: reviews-v1
24       weight: 0
25     - service: reviews-v3
26       weight: 100
27   matches:
28     - kind: HTTPRouteGroup
29       name: reviews-session-cookie
```

[View on GitHub](#)



3. Connect to a **bash** container:

```
$ kubectl exec deploy/bash -it -c bash -- bash
```

4. Create a traffic-generation script:

```
$ cat << 'EOF' > traffic.sh
#!/bin/bash
for i in $(seq 1 1000);
do
curl http://reviews:9080/health;
curl -i -H 'Cookie:session_token=xxx-yyy-zzz' http://
review:9080/health;
sleep 0.1;
done
EOF
```

5. Apply the configuration and check the traffic split.

```
$ kubectl apply -f ./Traffic-Management/nsm-debug-routing.yaml
httprouategroup.specs.smi-spec.io/reviews-session-cookie
unchanged
trafficsplit.split.smi-spec.io/reviews configured

$ nginx-meshctl top
Deployment      Incoming Success  Outgoing Success  NumRequests
bash            100.00%
reviews-v1      100.00%           65
reviews-v3      100.00%           65
```

### Debug Routing with NGINX Ingress Controller

Debug routing with NGINX Ingress Controller is similar to NGINX Service Mesh, except that traffic originates from outside the cluster rather than from inside the **bash** container. (You can generate that traffic with the **Traffic-Management/generate-traffic.sh** script; see [Step 4](#) in *Activating Client Rate Limiting with NGINX Ingress Controller*.)

1. Define a VirtualServer resource in **Traffic-Management/bookinfo-vs-debug-routing.yaml** that directs traffic users who have a session cookie to the **reviews-v3** service:

```
1  apiVersion: k8s.nginx.org/v1
2  kind: VirtualServer
3  metadata:
4    name: reviews
5  spec:
6    host: reviews.example.com
7    upstreams:
8      - name: reviews-v1
9        service: reviews-v1
10       port: 9080
11      - name: reviews-v2-1
12        service: reviews-v2-1
13        port: 9080
14      - name: reviews-v3
15        service: reviews-v3
16        port: 9080
17    routes:
18      - path: /api/reviews
19        matches:
20          - conditions:
21            - header: "cookie"
22              value: "session_token=xxx-yyy-zzz"
23            - variable: $request_method
24              value: GET
25          action:
26            pass: reviews-v3
27        action:
28          pass: reviews-v1
```

[View on GitHub](#)

2. Apply the resource:

```
$ kubectl apply -f ./Traffic-Management/bookinfo-vs-debug-
routing.yaml
```

## CHAPTER SUMMARY

Application traffic management is an important contributor to successful operation of apps and services. In this chapter we used NGINX Ingress Controller and NGINX Service Mesh to apply traffic control – for better app performance and resilience – and traffic splitting for seamless upgrades without downtime and testing of new app versions and features.

Let's summarize some of the key concepts from this chapter:

- NGINX Ingress Controller supports TCP and UDP load balancing for use cases involving TCP- and UDP-based apps and utilities.
- TLS Passthrough is an effective option for Layer 4 (TCP) routing where the encrypted traffic is passed to application workloads in Kubernetes.
- Multi-tenancy and delegation are especially important for organizations leveraging Kubernetes that are looking to scale up while reducing operational costs.
- Traffic control and traffic splitting are two basic categories of traffic-management methods.
  - Traffic control refers to the act of regulating the flow of traffic to your apps in terms of source, volume, and destination, using rate limiting and circuit breaking.
    - Rate limiting refers to setting the maximum number of requests that an app or service accepts from each client within a certain time period. Set rate limits with NGINX Ingress Controller for traffic from cluster-external clients, and with NGINX Service Mesh for traffic from other services within the Kubernetes cluster.
    - Circuit breaking refers to a mechanism that detects when an app or service is unavailable or experiencing high latency and immediately returns an error to clients instead of waiting for a timeout period as is the usual default behavior. This improves user experience and averts the cascading failures that occur when multiple services time out waiting for responses from other services.
  - Traffic splitting refers to dividing traffic between different versions of an app or service that are running simultaneously. Methods include blue-green deployment, canary deployment, A/B testing, and debug routing.
    - Blue-green deployment enables seamless app upgrades by directing all traffic to the new version of an app while keeping the old version running as a backup.
    - Canary deployment directs a small proportion of traffic to a new app version to verify that it performs well, while the rest of the traffic continues to go to the current version. When the new version proves stable, it receives increasing amounts of traffic.
    - A/B testing helps determine which version of an app users prefer. A group of users is defined based on characteristics (such as the value of an HTTP header) and their requests are always sent to one of the versions.
    - Debug routing is great for verifying the performance or stability of a new app version or feature by directing traffic to it from a group of users selected on the basis of a Layer 7 attributes (such as the session cookie they are using).

## 3. Monitoring and Visibility Use Cases

In this chapter we explore how to use NGINX and third-party tools and services for monitoring, visibility, tracing, and the insights required for successful traffic management in Kubernetes.

- [Monitoring with the NGINX Plus Live Activity Monitoring Dashboard](#)
- [Distributed Tracing, Monitoring, and Visualization with Jaeger, Prometheus, and Grafana](#)
- [Logging and Monitoring with the Elastic Stack](#)
- [Displaying Logs and Metrics with Amazon CloudWatch](#)
- [Chapter Summary](#)

### MONITORING WITH THE NGINX PLUS LIVE ACTIVITY MONITORING DASHBOARD

MONITORING AND VISIBILITY  
ARE CRUCIAL FOR SUCCESSFUL  
APP DELIVERY

Monitoring and visibility are crucial for successful app delivery, as tracking of app availability and request processing helps you identify issues quickly and resolve them in a timely way.

For this reason, by default the [NGINX Plus API and live monitoring dashboard](#) are enabled for the NGINX Ingress Controller based on NGINX Plus.

(The [stub\\_status](#) module is enabled by default for the NGINX Ingress Controller based on NGINX Open Source, but no dashboard is provided.)

The NGINX Plus live monitoring dashboard is enabled on port 8080 by default, but you can designate a different port by adding the following line to the **args** section (starting on line 66) of **Installation-Deployment/nginx-plus-ingress.yaml**:

```
- --nginx-status-port=<port_number>
```

[View on GitHub](#)

Although we don't recommend it, you can disable statistics gathering completely by adding this line to the **args** section:

```
- --nginx-status=false
```

[View on GitHub](#)

To access the live activity monitoring dashboard:

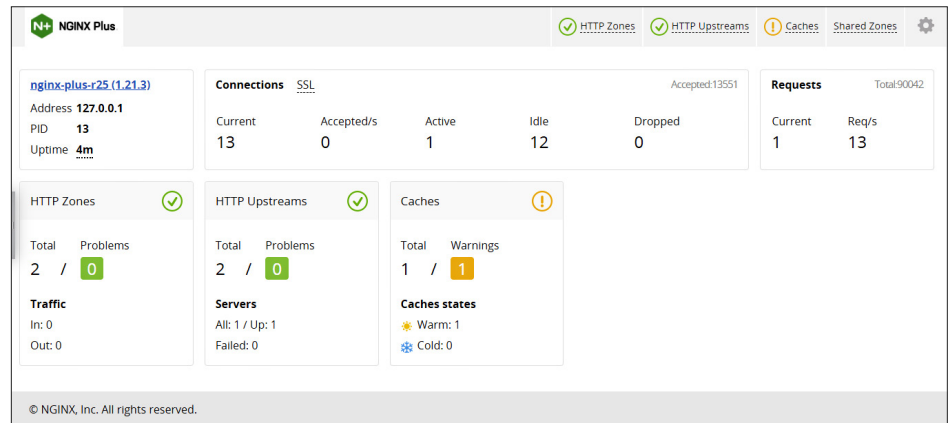
1. Apply the dashboard configuration:

```
$ kubectl apply -f ./Installation-Deployment/nginx-plus-ingress.yaml
```

2. Run the `kubectl port-forward` command to **forward connections** made to port 8080 on your local machine to port 8080 of the specified NGINX Ingress Controller pod:

```
$ kubectl port-forward deploy/nginx-ingress 8080 -n nginx-ingress
```

3. Open the dashboard in your browser at <http://localhost:8080/dashboard.html>.



The main page of the dashboard displays summary metrics, which you can explore in fine-grained detail, down to the level of a single pod, on the tabs:

- **HTTP Zones** – Statistics for each `server{}` and `location{}` block in the `http{}` context that includes the `status_zone` directive
- **HTTP Upstreams** – Statistics for each `upstream{}` block in the `http{}` context that includes the `zone` directive
- **Caches** – Statistics for each cache
- **Shared Zones** – The amount of memory currently used by each shared memory zone

For more information about the tabs, see the [NGINX Plus documentation](#).

TRAFFIC MANAGEMENT  
TOOLS SUCH AS LOAD  
BALANCERS, REVERSE  
PROXIES, AND INGRESS  
CONTROLLERS GENERATE  
A LOT OF INFORMATION

## DISTRIBUTED TRACING, MONITORING, AND VISUALIZATION WITH JAEGER, PROMETHEUS, AND GRAFANA

Although a microservices-based application looks like a single entity to its clients, internally it's a [daisy-chain network](#) of several microservices involved in completing a request from end users. How can you troubleshoot issues as requests are routed through this potentially complex network? [Distributed tracing](#) is a method for tracking services that shows detailed session information for all requests as they are processed, helping you diagnose issues with your apps and services.

Traffic management tools such as load balancers, reverse proxies, and Ingress controllers generate a lot of information about the performance of your services and applications. You can configure NGINX Ingress Controller and NGINX Service Mesh to feed such information to third-party monitoring tools, which among other features give you extra insight with visualization of performance over time.

In this section we show how to deploy three of the most popular tools:

- [Jaeger](#) for distributed tracing
- [Prometheus](#) for monitoring and alerting
- [Grafana](#) for analytics and visualization of data collected by Prometheus

### Enabling Distributed Tracing, Monitoring, and Visualization for NGINX Service Mesh

Distributed tracing, monitoring, and visualization are enabled by default for NGINX Service Mesh, but a server for each of Jaeger, Prometheus, and Grafana must be deployed in the cluster.

At the time of writing, NGINX Service Mesh automatically deploys default Jaeger, Prometheus, and Grafana servers. These deployments are intended for onboarding and evaluation, and might not be feature-complete or robust enough for production environments. In addition, by default the data they gather and present does not persist.

For production environments, we recommend that you separately deploy Jaeger, Prometheus, and Grafana. In the **Monitoring-Visibility** directory of the eBook repo, we provide configuration for each server: [jaeger.yaml](#), [prometheus.yaml](#), and [grafana.yaml](#).

#### Notes:

- The next planned release of NGINX Service Mesh, version 1.5, will not create default deployments of these servers, making the commands in Step 1 below mandatory even for non-production environments.
- Large-scale production deployments of Jaeger need an Elasticsearch cluster for backend storage. The sample Jaeger deployment defined in [jaeger.yaml](#) stores data in memory and is not recommended for use in production. For information about deploying Jaeger with Elasticsearch, see the [jaegertracing repo](#) on GitHub.

Enable distributed tracing, monitoring, and visualization for NGINX Service Mesh:

1. Create the **monitoring** namespace and configure Grafana, Prometheus, and Jaeger:

```
$ kubectl create ns monitoring
$ kubectl apply -f ./Monitoring-Visibility/grafana.yaml
$ kubectl apply -f ./Monitoring-Visibility/prometheus.yaml
$ kubectl apply -f ./Monitoring-Visibility/jaeger.yaml
```

2. Connect Grafana, Prometheus, and Jaeger to NGINX Service Mesh:

```
$ nginx-meshctl remove
$ nginx-meshctl deploy --sample-rate 1 --prometheus-address
"prometheus-service.monitoring:9090" --tracing-address
"jaeger.monitoring:6831"
```

For more information, see the [NGINX Service Mesh documentation](#).

## Enabling Distributed Tracing for NGINX Ingress Controller

NGINX Ingress Controller supports distributed tracing with a third-party [OpenTracing module](#) that works with Datadog, Jaeger, and Zipkin. Distributed tracing is disabled by default.

To enable distributed tracing with OpenTracing and Jaeger:

1. Include lines 10–11 in the **data** section of the ConfigMap for NGINX Ingress Controller, in **Monitoring-Visibility/nginx-config.yaml**:

```
6 data:
10   enable-opentracing: "true"
11   jaeger-collector-host: jaeger.monitoring.svc.cluster.local
```

[View on GitHub](#)

2. Apply the ConfigMap:

```
$ kubectl apply -f ./Monitoring-Visibility/nginx-config.yaml
```

3. Run the **kubectl port-forward** command to forward connections made to port 16686 on your local machine to the Jaeger service in the **monitoring** namespace:

```
$ kubectl port-forward -n monitoring svc/jaeger 16686
```

For more information, see the [NGINX Ingress Controller documentation](#).

## Enabling Monitoring and Visualization for NGINX Ingress Controller

Metrics from the NGINX Ingress Controller based on NGINX Plus (as well as general latency metrics) are exposed in Prometheus format at **/metrics**, on port 9913 by default. To change the port, see Step 1.

To enable metrics collection:

1. Include the following settings in the configuration for the NGINX Ingress Controller based on NGINX Plus, in **Installation-Deployment/nginx-plus-ingress.yaml**:

- A label with **nginx-ingress** as the resource name:

```
13 labels:
14   app: nginx-ingress
15   nsm.nginx.com/deployment: nginx-ingress
```

[View on GitHub](#)

- Annotations that define how Prometheus scrapes metrics:

```
17 annotations:
20   prometheus.io/scrape: "true"
21   prometheus.io/port: "9113"
22   prometheus.io/scheme: "http"
```

[View on GitHub](#)

- These arguments to the NGINX Ingress Controller initialization command line:

```
66 args:
71   - -enable-prometheus-metrics
72   - -enable-latency-metrics
```

[View on GitHub](#)

- (Optional.) This argument in the **args** section (starting on line 66), if you want to change the port on which metrics are exposed from the default of 9113:

```
- - prometheus-metrics-listen-port=<port_number>
```

[View on GitHub](#)

2. Apply the configuration:

```
$ kubectl apply -f ./Installation-Deployment/nginx-plus-ingress.yaml
```



3. Port-forward connections made to the Prometheus UI on port 9090 and to Grafana on port 3000 of your local machine to the Prometheus and Grafana services in the **monitoring** namespace:

```
$ kubectl port-forward -n monitoring svc/prometheus-service 9090
$ kubectl port-forward -n monitoring svc/grafana 3000
```

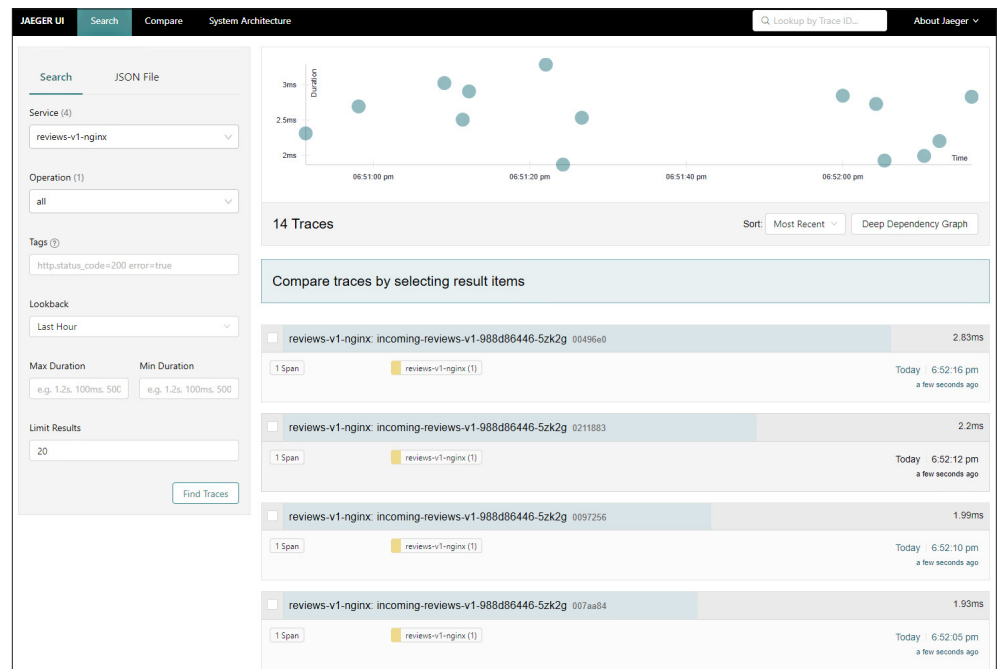
## Visualizing Distributed Tracing and Monitoring Data

THERE ARE SEVERAL  
WAYS TO VISUALIZE  
DATA FROM DISTRIBUTED  
TRACING AND MONITORING

There are several ways to visualize data from distributed tracing and monitoring of NGINX Ingress Controller and NGINX Service Mesh.

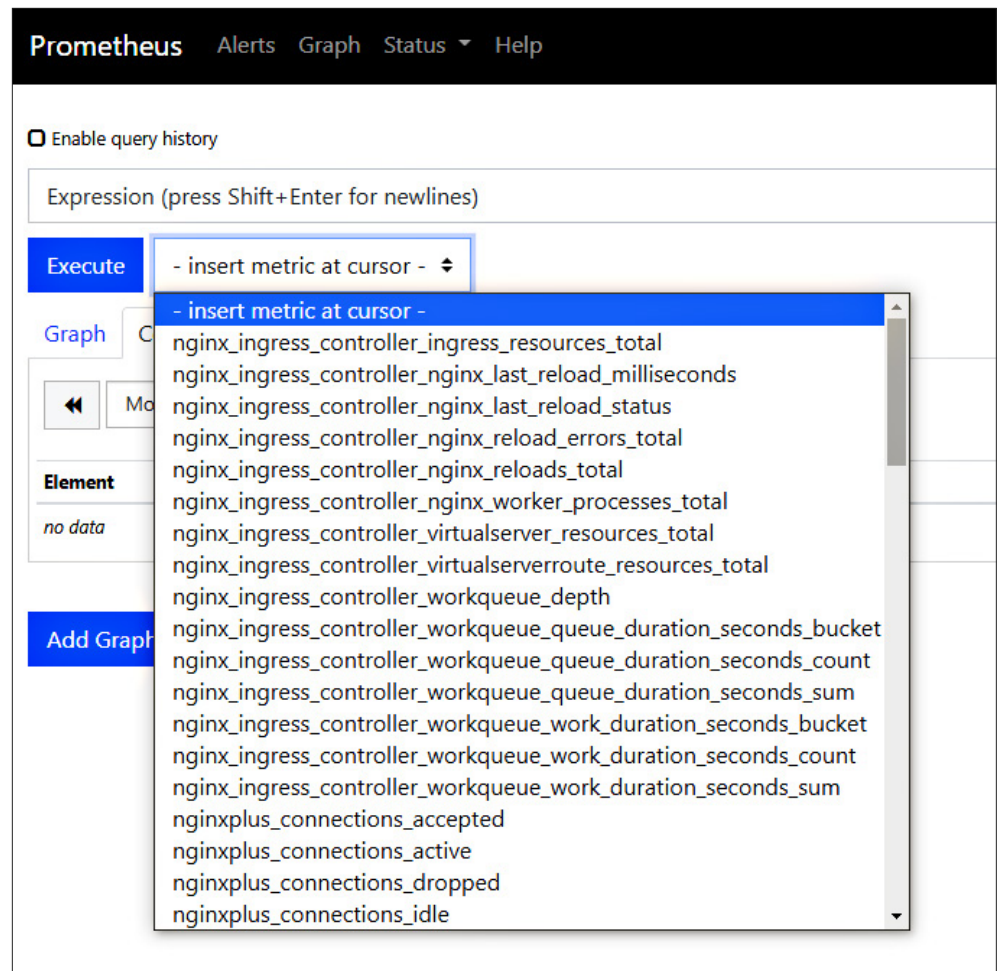
To display distributed tracing data for NGINX Ingress Controller, open the Jaeger dashboard in a browser at **http://localhost:16686**.

This sample Jaeger dashboard shows details for four requests, with the most recent at the top. The information includes how much time a response took, and the length of time between responses. In the example, the most recent response, with ID **00496e0**, took 2.83ms, starting about 4 seconds after the previous response.



To display metrics for NGINX Ingress Controller and NGINX Service Mesh with Prometheus, open the Prometheus dashboard in a browser at <http://localhost:9090>. Select the metrics you want to include in the dashboard from the **Execute** drop-down menu.

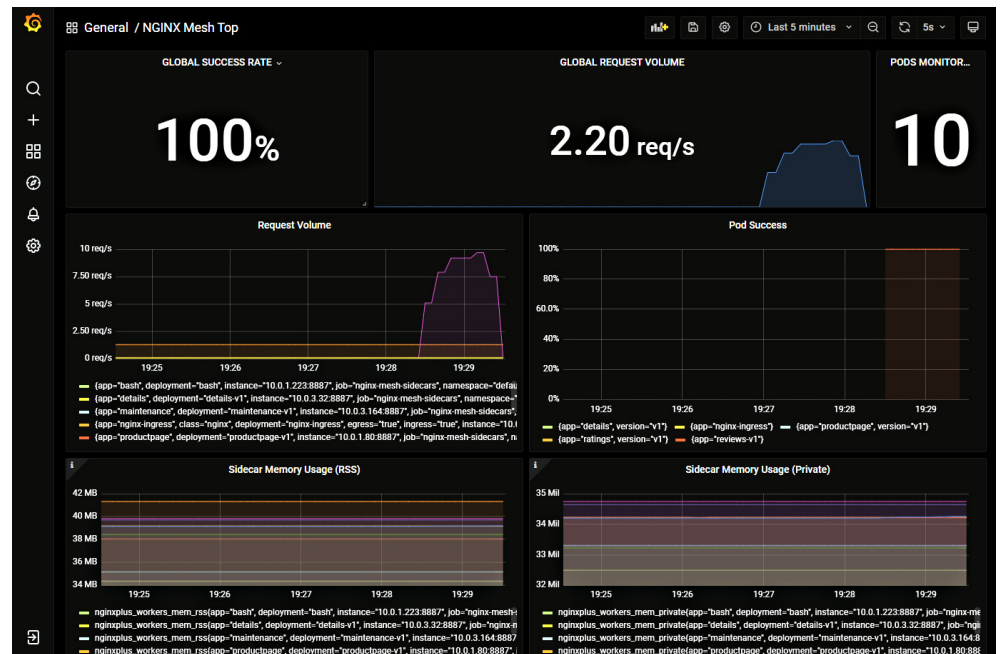
Prometheus metrics exported from the NGINX Ingress Controller are prefixed with **nginx\_ingress** and metrics exported from the NGINX Service Mesh sidecars are prefixed with **nginxplus**. For example, **nginx\_ingress\_controller\_upstream\_server\_response\_latency\_ms\_count** is specific to NGINX Ingress Controller, while **nginxplus\_upstream\_server\_response\_latency\_ms\_count** is specific to NGINX Service Mesh sidecars.



For descriptions of available metrics, see:

- [Available Metrics](#) in the NGINX Ingress Controller documentation
- [NGINX Plus Ingress Controller Metrics](#) in the NGINX Service Mesh documentation
- [NGINX Prometheus Exporter](#) on GitHub

To display metrics for NGINX Ingress Controller and NGINX Service Mesh with Grafana, open the Grafana UI in a browser at **http://localhost:3000**. Add Prometheus as a **data source** and create a **dashboard**. This example includes global success rate and request volume per second, memory usage, and more:



## LOGGING AND MONITORING WITH THE ELASTIC STACK

The [Elastic Stack](#) (formerly called the ELK stack) is a popular open source logging tool made up of three base tools:

- [Elasticsearch](#) – Search and analytics engine
- [Logstash](#) – Server-side data processing pipeline
- [Kibana](#) – Visualization engine for charts and graphs

In this section we explain how to collect and visualize NGINX Ingress Controller logs with Elastic Stack, using the [Filebeat module for NGINX](#). Filebeat monitors the log files or locations that you specify, collects log events, and forwards them to either Elasticsearch or Logstash for indexing.

### Configuring the NGINX Ingress Controller Access and Error Logs

NGINX Ingress Controller writes to two logs:

- Access log – Information about client requests recorded right after the request is processed. To customize the information included in the access log entries, add these ConfigMap keys to the **data** section (starting on line 6) of **Monitoring-Visibility/nginx-config.yaml**:
  - **log-format** for HTTP and HTTPS traffic
  - **stream-log-format** for TCP, UDP, and TLS Passthrough traffic

For an example of log-entry customization, see the [NGINX Ingress Controller](#) repo on GitHub. For a list of all the NGINX built-in variables you can include in log entries, see the [NGINX reference documentation](#).

Although we do not recommend that you disable access logging, you can do so by including this key in the **data** section of **nginx-config.yaml**:

```
access-log-off: "true"
```

[View on GitHub](#)

Run:

```
$ kubectl apply -f ./Monitoring-Visibility/nginx-config.yaml
```

- Error log – Information about error conditions at the severity levels you configure with the **error-log-level** ConfigMap key.

To enable debug logging, include this key in the **data** section (starting on line 6) of **Monitoring-Visibility/nginx-config.yaml**:

```
error-log-level: "debug"
```

[View on GitHub](#)

Also include this line in the **args** section (starting on line 66) of **Installation-Deployment/nginx-plus-ingress.yaml**. This starts NGINX Ingress Controller in debug mode.

```
- -nginx-debug
```

[View on GitHub](#)

Run these commands:

```
$ kubectl apply -f ./Monitoring-Visibility/nginx-config.yaml
$ kubectl apply -f ./Installation-Deployment/nginx-plus-ingress.yaml
```

To view NGINX Ingress Controller logs, run:

```
$ kubectl logs deploy/nginx-ingress -n nginx-ingress -f
```

## Enabling Filebeat

**Filebeat** is a module in the Elastic stack that “monitors the log files or locations that you specify, collects log events, and forwards them either to Elasticsearch or Logstash for indexing”.

Here we use two Filebeat features:

- The **module for NGINX**, which parses the NGINX access and error logs
- The **autodiscover** feature, which tracks containers as their status changes (they spin up and down or change locations) and adapts logging settings automatically

To enable the Filebeat NGINX module with the autodiscover feature:

1. Sign in to your **Elastic Cloud account** ([start a free trial](#) if you don’t already have an account) and create a deployment. Record the username and password for the deployment in a secure location, as you need the password in the next step and cannot retrieve it after the deployment is created.

2. Configure the Filebeat NGINX module with the autodiscover feature. The **templates** section (starting on line 14) of **Monitoring-Visibility/elk/filebeat.yaml** directs the autodiscover subsystem to start monitoring new services when they initialize.

On line 26, replace ***ccloud\_ID*** with the Cloud ID associated with your Elastic Cloud deployment. (To access the Cloud ID, select **Manage this deployment** in the left-hand navigation column in Elastic Cloud. The value appears in the **Cloud ID** field on the page that opens.)

On line 27, replace ***password*** with the password associated with the deployment, which you noted in Step 1.

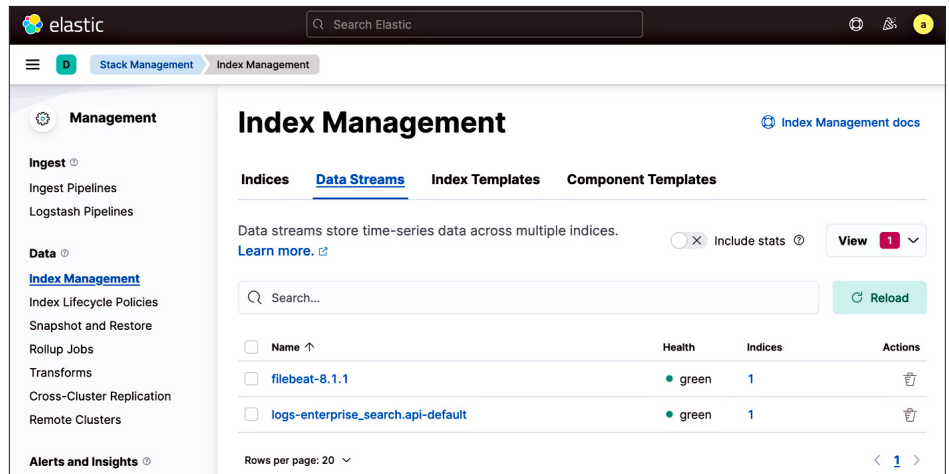
```
11 filebeat.autodiscover:
12   providers:
13   - type: kubernetes
14     templates:
15       - condition:
16         equals:
17           kubernetes.container.name: "nginx-plus-ingress"
18         config:
19         - module: nginx
20           access:
21             enabled: true
22           input:
23             type: container
24             paths:
25               - /var/log/containers/*-
26                 ${data.kubernetes.container.id}.log
27 cloud.id: "ccloud_ID"
28 cloud.auth: "elastic:password"
```

[View on GitHub](#)

3. Apply the Filebeat configuration:

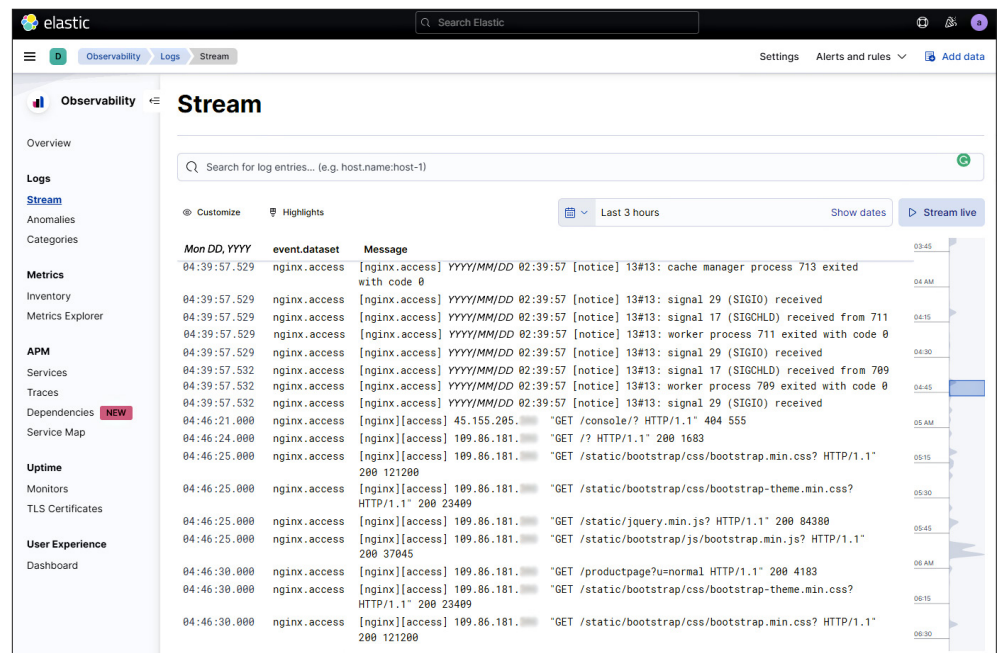
```
$ kubectl apply -f ./Monitoring-Visibility/elk/filebeat.yaml
```

- Confirm that your Filebeat deployment appears on the **Data Streams** tab of the Elastic Cloud **Index Management** page. (To navigate to the tab, click **Stack Management** in the **Management** section of the left-hand navigation column. Then click **Index Management** in the navigation column and **Data Streams** on the **Index Management** page.) In the screenshot, the deployment is called **filebeat-8.1.1**.

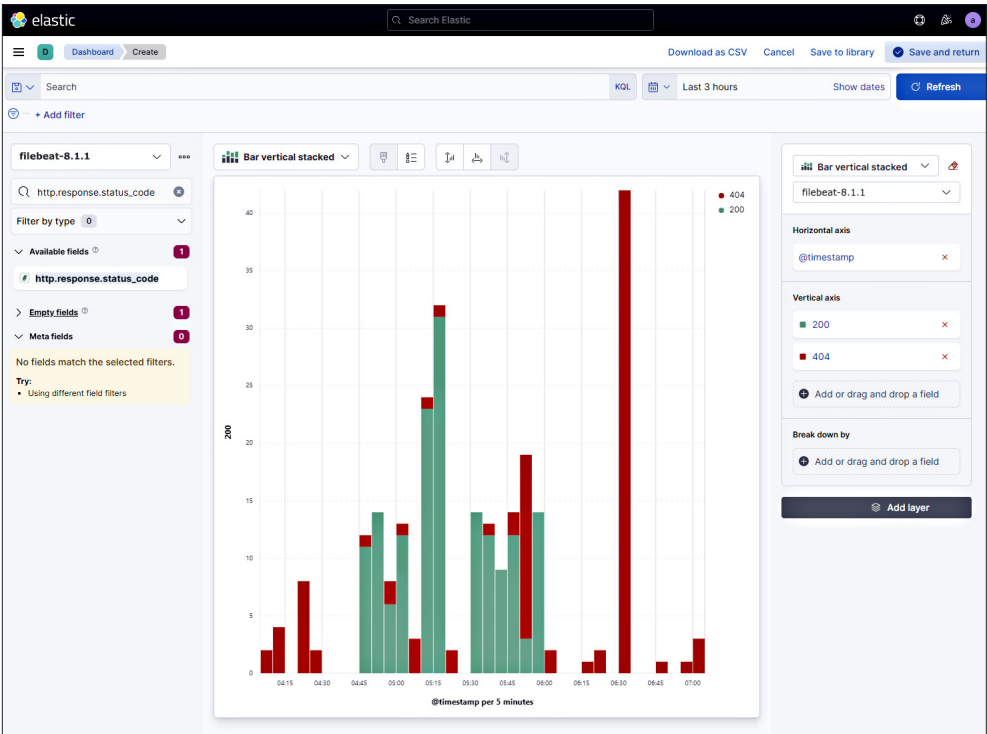


## Displaying NGINX Ingress Controller Log Data with Filebeat

To display the NGINX Ingress Controller access and error logs that Filebeat has forwarded to Elasticsearch, access the **Stream** page. (In the left-hand navigation column, click **Logs** in the **Observability** section. The **Stream** page opens by default.)



To display statistics about log entries, navigate to the Kibana **Dashboards** tab (in the left-hand navigation column, click **Dashboard** in the **Analytics** section). The following sample chart displays the number of log entries in each 5-minute period for requests that resulted in status code **200** and **404**.





The Filebeat module for NGINX comes with a pre-configured dashboard. To load the Filebeat dashboards, run the following command in the Filebeat pod:

```
$ kubectl exec -it <filebeat_pod_name> -n kube-system -- bash

$ ./filebeat setup -e --dashboards \
> -E output.elasticsearch.hosts=['<Elastic_Cloud_host>:9200'] \
> -E output.elasticsearch.username=elastic \
> -E output.elasticsearch.password=<password> \
> -E setup.kibana.host=<Elastic_Cloud_host>:5601 \
> -c /etc/filebeat.yaml
```

Navigate to **Dashboards** and search for **nginx** to see the available dashboards:

## Dashboards

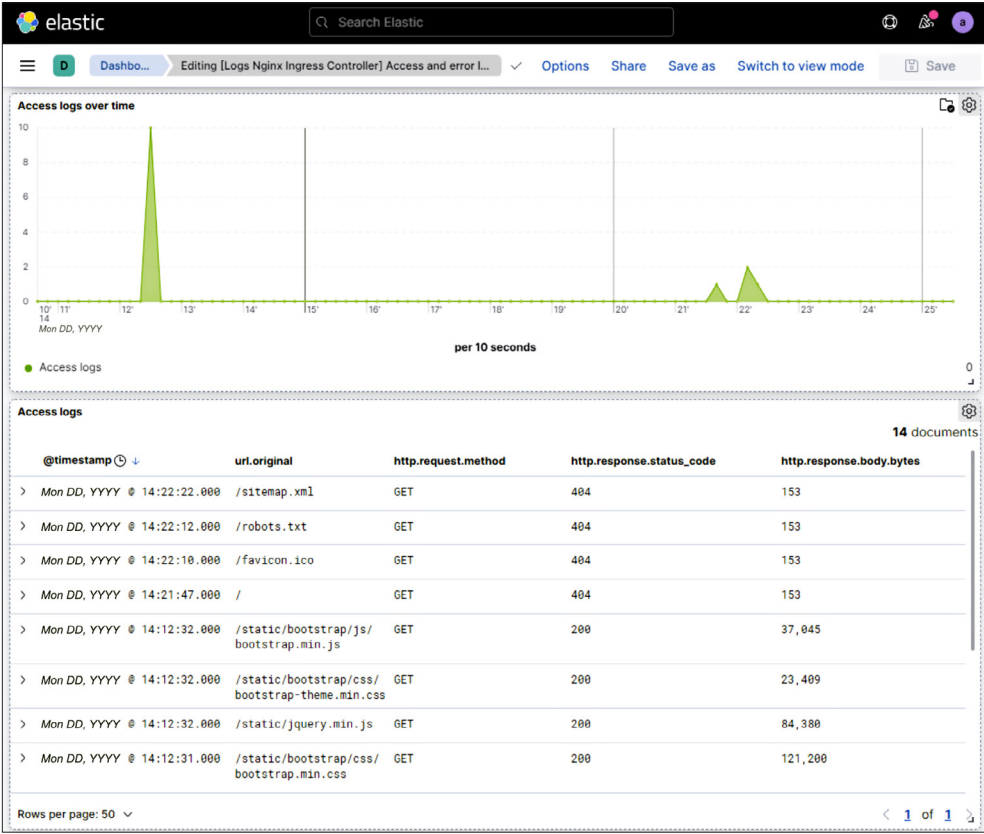
Create new dashboard

🔍 nginx

<input type="checkbox"/>	Title	Description	Actions
<input type="checkbox"/>	<a href="#">[Filebeat Nginx] [ML] Remote IP Count Explorer ECS</a>	Machine learning dashboard, for the Filebeat Nginx module	Edit
<input type="checkbox"/>	<a href="#">[Filebeat Nginx] [ML] Remote IP URL Explorer ECS</a>	Machine Learning dashboard for the Filebeat Nginx module	Edit
<input type="checkbox"/>	<a href="#">[Filebeat Nginx] Overview ECS</a>	Dashboard for the Filebeat Nginx module	Edit
<input type="checkbox"/>	<a href="#">[Filebeat Nginx] Access and error logs ECS</a>	Dashboard for the Filebeat Nginx module	Edit
<input type="checkbox"/>	<a href="#">[Metricbeat Nginx] Overview ECS</a>	Overview dashboard for the Nginx module in Metricbeat	Edit

Rows per page: 20 ▾

Select the **[Filebeat NGINX] Access and error logs ECS** dashboard. This sample dashboard includes a graph that plots the number of NGINX Ingress access logs over time and scrollable lists of log entries.



## Enabling Metricbeat and Displaying NGINX Ingress Controller and NGINX Service Mesh Metrics

Just as the Filebeat module for NGINX exports NGINX Ingress Controller logs to Elasticsearch, the [Metricbeat module for NGINX](#) scrapes Prometheus metrics from NGINX Ingress Controller and NGINX Service Mesh and sends them to Elasticsearch.

To enable the Metricbeat NGINX module with the autodiscover feature:

1. Sign in to your [Elastic Cloud account](#) if you have not already done so. (For information about creating an account, as well as the autodiscover feature, see [Enabling Filebeat](#).)
2. Configure the Metricbeat NGINX module with the autodiscover feature to scrape metrics and display them in the Elastic Metrics Explorer. The `templates` section (starting on line 14) of **Monitoring-Visibility/elk/metricbeat.yaml** directs the autodiscover subsystem to start monitoring new services when they initialize.

By default, the NGINX Service Mesh sidecar and NGINX Ingress Controller expose metrics in Prometheus format at **/metrics**, on ports 8887 and 9113 respectively.

To change the defaults, edit lines 20–21 for NGINX Service Mesh and lines 29–30 for NGINX Ingress Controller.

On line 31, replace `ccloud_ID` with the Cloud ID associated with your Elastic Cloud deployment. (To access the Cloud ID, select **Manage this deployment** from the left-hand navigation column in Elastic Cloud. The value appears in the **Cloud ID** field on the page that opens.)

On line 32, replace `password` with the password associated with the deployment.

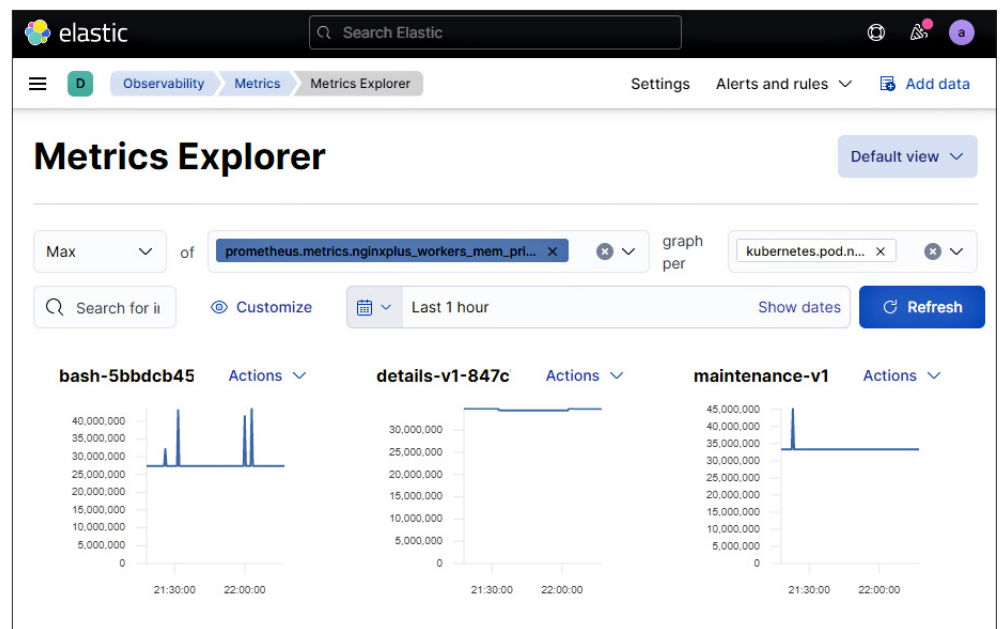
```
11 metricbeat.autodiscover:
12   providers:
13     - type: kubernetes
14       templates:
15         - condition.equals:
16             kubernetes.container.name: "nginx-mesh-sidecar"
17         config:
18           - module: prometheus
19             period: 10s
20             hosts: ["${data.host}:8887"]
21             metrics_path: /metrics
22     - type: kubernetes
23       templates:
24         - condition.equals
25             kubernetes.container.name: "nginx-plus-ingress"
26         config:
27           - module: prometheus
28             period: 10s
29             hosts: ["${data.host}:9113"]
30             metrics_path: /metrics
31   cloud.id: "ccloud_ID"
32   cloud.auth: "elastic:password"
```

[View on GitHub](#)

3. Apply the Metricbeat configuration:

```
$ kubectl apply -f ./Monitoring-Visibility/elk/metricbeat.yaml
```

This sample screenshot displays the **Maximum** value of the **prometheus.metrics.nginxplus\_workers\_mem\_private** metric during each time period, grouped by **kubernetes.pod.name**.



For more information about logging and the Elastic Stack, see:

- [Logging](#) in the NGINX Ingress Controller documentation
- [How to monitor NGINX web servers with the Elastic Stack](#) on the Elastic blog
- [Run Filebeat on Kubernetes](#) and [Run Metricbeat on Kubernetes](#) in the Elastic documentation
- [Nginx module](#) (Filebeat) and [Nginx module](#) (Metricbeat) in the Elastic documentation

## DISPLAYING LOGS AND METRICS WITH AMAZON CLOUDWATCH

Amazon CloudWatch is a monitoring and observability service that provides a unified view of your NGINX Ingress Controller and NGINX Service Mesh deployment in the CloudWatch console.

### Configuring CloudWatch

To configure and use CloudWatch, you create two configurations: a standard Prometheus `<scrape_config>` configuration and a CloudWatch agent configuration.

1. Complete the instructions in [Enabling Distributed Tracing, Monitoring, and Visualization for NGINX Service Mesh](#) and [Enabling Monitoring and Visualization for NGINX Ingress Controller](#).
2. Configure Prometheus scraping and an embedded metric format (EMF) processor rule for sending NGINX Ingress Controller metrics to CloudWatch, in **Monitoring-Visibility/cloudwatch/cw-metrics.yaml**. (Similar EMF configuration for NGINX Service Mesh is on lines 68–75.)

For descriptions of the fields, see *CloudWatch agent configuration for Prometheus* in the [CloudWatch documentation](#).

On line 65, change `nginx-demo-cluster` to match your cluster name.

```
53 data:
54   cwagentconfig.json: |
55     {
56       "logs": |
57         "metrics_collected": {
58           "prometheus": {
59             "prometheus_config_path": "/etc/prometheusconfig/
60 prometheus.yaml",
61             "log_group_name": "nginx-metrics",
62             "cluster_name": "nginx-demo-cluster",
63             "emf_processor": {
64               "metric_declaration": [
65                 {
66                   "source_labels": ["job"],
67                   "label_matcher": "nic",
68                   "dimensions": [["PodNamespace", "PodName"]],
69                   "metric_selectors": [
70                     "nginx*"
71                   ]
72                 }
73               ]
74             }
75           }
76         },
77       }
78     }
```

[View on GitHub](#)

3. Configure Prometheus to scrape metrics from NGINX Ingress Controller (the **nginx-plus-ingress** pod) once per minute (similar configuration for NGINX Service Mesh is on lines 103–125):

```
97 data:
98   prometheus.yaml: |
99     global:
100       scrape_interval: 1m
101       scrape_timeout: 5s
102     scrape_configs:
126     - job_name: nic
127       sample_limit: 10000
128       kubernetes_sd_configs:
129       - role: pod
130       relabel_configs:
131         - source_labels: [ __meta_kubernetes_pod_container_name ]
132           action: keep
133           regex: '^nginx-plus-ingress$'
134         - action: replace
135           source_labels:
136             - __meta_kubernetes_namespace
137           target_label: PodNamespace
138         - action: replace
139           source_labels:
140             - __meta_kubernetes_pod_name
141           target_label: PodName
142         - action: labelmap
143           regex: __meta_kubernetes_pod_label_(.+)
```

[View on GitHub](#)

4. Specify your AWS credentials by replacing:

- `<< AWS_access_key >>` on line 153 with your AWS access key
- `<< AWS_secret_access_key >>` on line 154 with your secret access key

(For instructions about creating and accessing AWS access keys, see the [AWS documentation](#).)

```
150 data:
151   credentials: |
152     [AmazonCloudWatchAgent]
153     aws_access_key_id = << AWS_access_key >>
154     aws_secret_access_key = << AWS_secret_access_key >>
```

[View on GitHub](#)

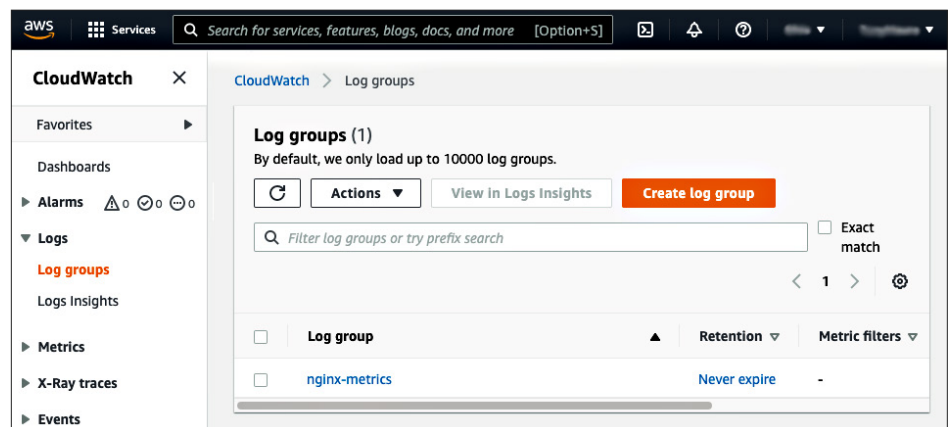
If you have an AWS session token, add this line directly below line 154:

```
aws_session_token = << your_AWS_session_token >>
```

5. Apply the CloudWatch configuration:

```
$ kubectl apply -f ./Monitoring-Visibility/cloudwatch/
cw-metrics.yaml
```

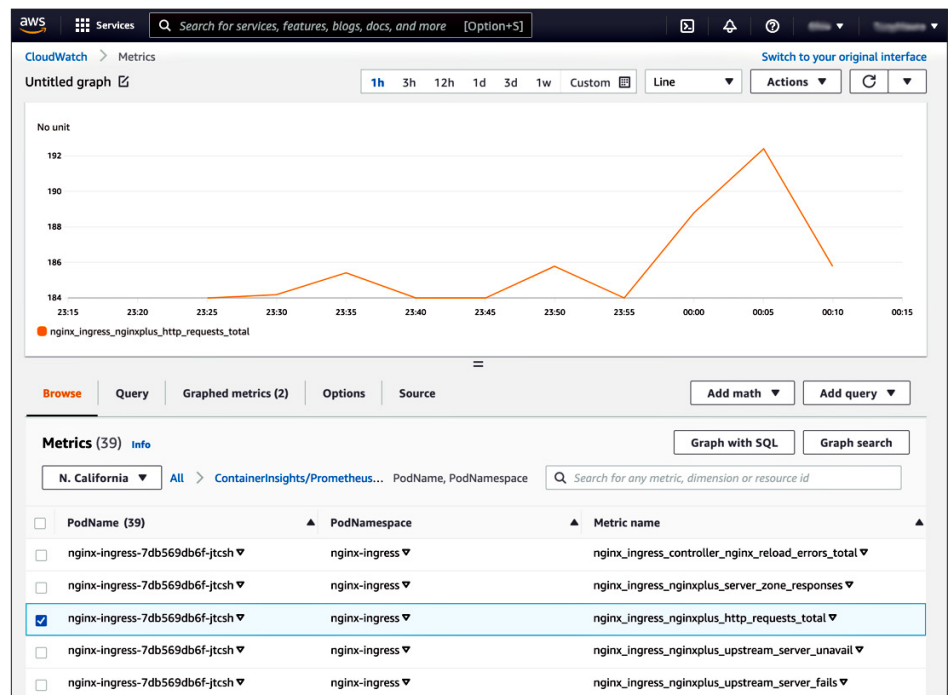
With this configuration in place, the **nginx-metrics** log group appears on the **Log Groups** tab.



## Creating Graphs in CloudWatch

To graph metrics on a CloudWatch dashboard:

1. In the left-hand navigation column, select **All metrics** in the **Metrics** section.
2. On the page that opens, select the **ContainerInsights/Prometheus** custom namespace on the **Browse** tab in the lower part of the page.
3. On the **Browse** tab, click the checkbox at the left end of the row for a metric to graph it in the upper part of the page. This screenshot displays a graph of the **nginx\_ingress\_nginxplus\_http\_requests\_total** metric.





## Capturing Logs in CloudWatch with Fluent Bit

There are two ways to send logs from your containers to CloudWatch: Fluent Bit and Fluentd. Here we use Fluent Bit because it has the following advantages over Fluentd:

- A smaller resource footprint and more resource-efficient usage of memory and CPU
- The image is developed and maintained by AWS, resulting in quicker adoption of new Fluent Bit image features and faster reaction to bugs or other issues

To export logs to CloudWatch using Fluent Bit:

1. Configure logging with with Fluent Bit, in

**Monitoring-Visibility/cloudwatch/cw-fluentbit.yaml:**

```
66 nginx-ingress.conf: |
67   [INPUT]
68     Name          tail
69     Tag           nic.data
70     Path          /var/log/containers/nginx-ingress*.log
71     Parser        docker
72     DB            /var/fluent-bit/state/flb_log.db
73     Mem_Buf_Limit 5MB
74     Skip_Long_Lines On
75     Refresh_Interval 10
76   [FILTER]
77     Name          parser
78     Match         nic.*
79     Key_Name      log
80     Parser        nginx_nic
81     Reserve_Data  On
82     Preserve_Key  On
83   [OUTPUT]
84     Name          cloudwatch_logs
85     Match         nic.*
86     region        ${AWS_REGION}
87     log_group_name nginx-logs
88     log_stream_prefix ${HOST_NAME}-
89     auto_create_group true
90     extra_user_agent container-insights
```

[View on GitHub](#)

2. Apply the Fluent Bit configuration:

```
$ kubectl apply -f ./Monitoring-Visibility/cloudwatch/
cw-fluentbit.yaml
```

With this configuration in place, the CloudWatch log stream is formatted as follows:

```
▶ YYYY-MM-DDT01:55:34.852Z {"log":"Accept: application/openmetrics-text; \" while readin
▼ YYYY-MM-DDT01:55:42.000Z {"remote":"198.51.100.5","host":"-","user":"-","method":"GET"
  {
    "remote": "198.51.100.5",
    "host": "-",
    "user": "-",
    "method": "GET",
    "path": "/",
    "code": "302",
    "size": "145",
    "referer": "-",
    "agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrom
    "http_x_forwarded_for": "\\\"-\\\"\\n\",
    "log": "198.51.100.5 - -[DD/Mon/YYYY:01:55:42 +0000] \\\"GET / HTTP/1.1\\\" 302 145 \\\"-\\\" \\\"Mozilla
    "stream": "stdout",
    "time": "YYYY-MM-DDT01:55:42.879931691Z"
  }
}
```

No newer events at this moment. *Auto retry paused. Resume*

For more information about CloudWatch, see:

- [Scraping additional Prometheus sources and importing those metrics in the CloudWatch documentation](#)
- [A sample CloudWatch configuration on GitHub used by AWS's One Observability Workshop](#)
- [Set up NGINX with sample traffic on Amazon EKS and Kubernetes in the Cloud Watch documentation](#)

## CHAPTER SUMMARY

We discussed why actionable insights into app and service performance are crucial to successful traffic management in Kubernetes clusters – they help you quickly identify and resolve issues that worsen user experience. We showed how to configure tools for tracing, monitoring, and visibility of NGINX Ingress Controller and NGINX Service Mesh.

Let's summarize some of the key concepts from this chapter:

- With the NGINX Ingress Controller based on NGINX Plus, the NGINX Plus API and live activity monitoring dashboard are enabled by default and report key load-balancing and performance metrics for insight into app performance and availability.
- The Jaeger distributed tracing service tracks and shows detailed session information for all requests as they are processed, which is key to diagnosing and troubleshooting issues with apps and services.
- Prometheus and Grafana combine as a comprehensive monitoring tool. Prometheus monitors performance and generates alerts about problems, while Grafana generates graphs and other visualizations of the data collected by Prometheus.
- The Elastic Stack (formerly the ELK stack) is a popular open source logging tool made up of Elasticsearch for search and analytics, Logstash as the data processing pipeline, and Kibana for charts and graphs.
- Amazon CloudWatch is a monitoring and observability tool for Kubernetes clusters. It tracks abnormal behavior and performs automated mitigations, visualizes metrics, and highlights situations to troubleshoot.

AUTHENTICATING USER IDENTITIES AND ENFORCING PERMISSIONS ARE PROBABLY THE MOST COMMON WAY TO PREVENT UNAUTHORIZED ACCESS

WE SHOW HOW TO IMPLEMENT A FULL-FLEDGED SSO SOLUTION

## 4. Identity and Security Use Cases

In this chapter we explore techniques for verifying user identity and securing applications. We show how to configure tools for both capabilities that are examples of the available solutions.

While there are many ways to protect applications, authenticating user identities and enforcing permissions are probably the most common way to prevent unauthorized access to application resources. App developers commonly leverage a third-party identity provider to manage user credentials and digital profiles. Identity providers eliminate the need for app developers and administrators to write and manage bespoke solutions for authenticating users' digital identities and controlling access to application resources.

Identity-provider solutions increase overall user satisfaction by enabling single sign-on (SSO). Users do not need to input profile data separately for each app and then remember the associated unique usernames and passwords. Instead, one paired username and password enables access to all apps. The identity provider enforces consistent security criteria for identity attributes like passwords, reducing end-user frustration with the registration process which can lead to abandonment.

[OpenID Connect \(OIDC\)](#) is an authentication protocol for SSO built on the industry-standard OAuth 2.0 protocol. We show how to implement a full-fledged SSO solution that supports the [OIDC Authorization Code Flow](#), with the NGINX Ingress Controller based on [F5 NGINX Plus](#) as the relaying party which analyzes user requests to determine the requester's level of authorization and routes requests to the appropriate app service. (For ease of reading, the remainder of this chapter uses the term *NGINX Ingress Controller* for the NGINX Plus-based model.)

We provide instructions for implementing SSO with three OIDC identity providers (IdPs): Okta, Azure Active Directory (AD), and Ping Identity.

Finally, we show how to “shift security left” by integrating [F5 NGINX App Protect WAF](#) with NGINX Ingress Controller.

- [Implementing SSO with Okta](#)
- [Implementing SSO with Azure Active Directory](#)
- [Implementing SSO with Ping Identity](#)
- [Implementing SSO for Multiple Apps](#)
- [Deploying NGINX App Protect with NGINX Ingress Controller](#)
- [Chapter Summary](#)

## IMPLEMENTING SSO WITH OKTA

In this section you use the Okta CLI to preconfigure **Okta** as the OIDC identity provider (IdP) for SSO and then configure NGINX Ingress Controller as the relaying party.

- [Prerequisites](#)
- [Configuring Okta as the IdP](#)
- [Configuring NGINX Ingress Controller as the Relaying Party with Okta](#)

### Prerequisites

1. Download the **Okta CLI software** to the local machine.
2. Create an Okta Developer account:

```
$ okta register
First name: <your_first_name>
Last name: <your_last_name>
Email address: <your_email_address>
Country: <your_country>
Creating new Okta Organization, this may take a minute:
An account activation email has been sent to you.

Check your email
```

3. Click the **Activate** button in the email.
4. In the browser window that opens, the email address you provided in Step 2 appears in the upper righthand corner. Click the down-arrow to the right of it and note the value that appears below your email address in the pop-up (here, **dev-609627xx.okta.com**).



5. In the browser window, click the **Create Token** button. Follow Steps 3–5 of [Create the Token](#) in the Okta documentation and record the token value.

## Configuring Okta as the IdP

1. Sign in to your Okta Developer account using the Okta CLI, substituting these values:
  - **<your\_okta\_domain>** – URL starting with **https://**, followed by the **dev-xxxxxxx.okta.com** value you obtained in Step 4 of [Prerequisites](#).
  - **<your\_okta\_API\_token>** – The token value you obtained in Step 5 of [Prerequisites](#).

```
$ okta login
Okta Org URL: <your_okta_domain>
Okta API token: <your_okta_API_token>
```

2. Create an app integration for the **bookinfo** sample app. In response to the prompts, type **1 (Web)** and **5 (Other)**:

```
$ okta apps create --app-name=bookinfo --redirect-
uri=https://bookinfo.example.com/_codexch
Type of Application
(The Okta CLI only supports a subset of application types
and properties):
> 1: Web
> 2: Single Page App
> 3: Native App (mobile)
> 4: Service (Machine-to-Machine)
Enter your choice [Web]: 1
Type of Application
> 1: Okta Spring Boot Starter
> 2: Spring Boot
> 3: JHipster
> 4: Quarkus
> 5: Other
Enter your choice [Other]: 5
Configuring a new OIDC Application, almost done:
Created OIDC application, client-id: 0oa1mi...OrfQAg5d7
Okta application configuration has been written to:
<current_directory>/.okta.env
```

3. Obtain the integrated app's client ID and Kubernetes Secret from the **OKTA\_OAUTH2\_CLIENT\_ID** and **OKTA\_OAUTH2\_CLIENT\_SECRET** fields in **.okta.env**:

```
$ cat .okta.env
export OKTA_OAUTH2_ISSUER="https://dev-xxxxxxx.okta.com/
oauth2/default"
export OKTA_OAUTH2_CLIENT_ID="0oa4go...b735d7"
export OKTA_OAUTH2_CLIENT_SECRET="CMRErvVMJKM...PINeaoFZZ6I"
```

4. Base64-encode the secret:

```
$ echo CMRErvVMJKM...PINeafZZ6I | base64
Q01SRXJ2Vk...YW9mW1o2SQo
```

## Configuring NGINX Ingress Controller as the Relaying Party with Okta

1. Edit **Identity-Security/okta-client-secret.yaml**, replacing the **client-secret** parameter on line 7 with the Base64-encoded Secret you generated in Step 4 of [Configuring Okta as the IdP](#) (just above).

```
1 apiVersion: v1
2 kind: Secret
3 metadata:
4   name: okta-oidc-secret
5 type: nginx.org/oidc
6 data:
7   client-secret: client-secret
```

[View on GitHub](#)

2. Apply the YAML file containing the Secret:

```
$ kubectl apply -f ./Identity-Security/okta-client-secret.yaml
```

3. Obtain the URLs for the integrated app's authorization endpoint, token endpoint, and JSON Web Key (JWK) file from the Okta configuration. Run the following **curl** command, piping the output to the indicated **python** command to output the entire configuration in an easily readable format. The output is abridged to show only the relevant fields. In the command, for **<your\_okta\_domain>** substitute the value (**https://dev-xxxxxxx.okta.com**) you used in Step 1 of [Configuring Okta as the IdP](#).

```
$ curl -s https://<your_okta_domain>/well-known/openid-configuration | python -m json.tool
{
  "authorization_endpoint":
  "https://<your_okta_domain>/oauth2/v1/authorize",
  ...
  "jwks_uri": "https://<your_okta_domain>/oauth2/v1/keys",
  ...
  "token_endpoint": "https://<your_okta_domain>/oauth2/v1/token",
  ...
}
```

4. Edit the NGINX Ingress OIDC Policy in **Identity-Security/okta-oidc-policy.yaml**, replacing the parameters as indicated:

- **client-id** on line 7 – The value you obtained from the `OKTA_OAUTH2_CLIENT_ID` field in [Step 3](#) of *Configuring Okta as the IdP*
- **your-okta-domain** on lines 9–11 – The value used for `<your-okta-domain>` in the command in the previous step

```
1 apiVersion: k8s.nginx.org/v1
2 kind: Policy
3 metadata:
4   name: okta-oidc-policy
5 spec:
6   oidc:
7     clientID: client-id
8     clientSecret: okta-oidc-secret
9     authEndpoint: https://your-okta-domain/oauth2/v1/authorize
10    tokenEndpoint: https://your-okta-domain/oauth2/v1/token
11    jwksURI: https://your-okta-domain/oauth2/v1/keys
```

[View on GitHub](#)

5. Apply the policy:

```
$ kubectl apply -f ./Identity-Security/okta-oidc-policy.yaml
```



6. Apply the VirtualServer resource (**Identity-Security/okta-oidc-bookinfo-vs.yaml**) that references **okta-oidc-policy**:

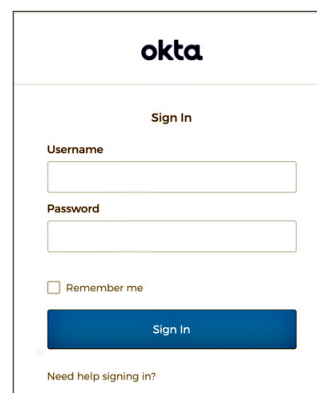
```
$ kubectl apply -f ./Identity-Security/okta-oidc-bookinfo-vs.yaml
```

The VirtualServer resource references **okta-oidc-policy** on line 16. The **path** definition on line 14 means that users who request a URL starting with **/** are authenticated before the request is proxied to the upstream called **backend** (line 18) which maps to the **productpage** service (lines 10–11):

```
1 apiVersion: k8s.nginx.org/v1
2 kind: VirtualServer
3 metadata:
4   name: bookinfo-vs
5 spec:
6   host: bookinfo.example.com
7   tls:
8     secret: bookinfo-secret
9   upstreams:
10  - name: backend
11    service: productpage
12    port: 9080
13  routes:
14  - path: /
15    policies:
16  - name: okta-oidc-policy
17  action:
18    pass: backend
```

[View on GitHub](#)

7. Test the SSO deployment by navigating to **https://bookinfo.example.com** (the hostname of NGINX Ingress Controller) in a browser. You are redirected to the Okta login portal, where you can enter the credentials for your Okta developer account to gain access to the backend application (the **productpage** service).

The image shows the Okta Sign In web form. At the top is the 'okta' logo. Below it is the 'Sign In' heading. The form contains two input fields: 'Username' and 'Password'. Below the password field is a checkbox labeled 'Remember me'. At the bottom of the form is a blue 'Sign In' button. Below the button is a link that says 'Need help signing in?'.

## IMPLEMENTING SSO WITH AZURE ACTIVE DIRECTORY

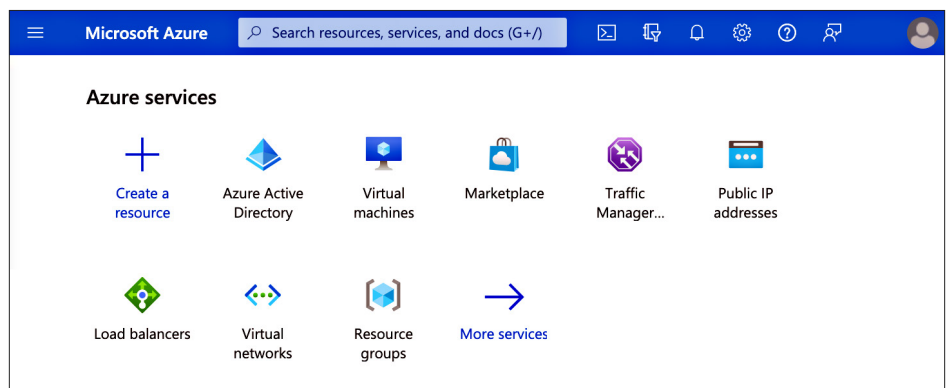
In this section you use the [Microsoft Azure portal](#) to preconfigure [Azure Active Directory \(AD\)](#) as the OIDC identity provider (IdP) for SSO and then configure NGINX Ingress Controller as the relaying party.

- [Configuring Azure AD as the IdP](#)
- [Configuring NGINX Ingress Controller as the Relaying Party with Azure AD](#)

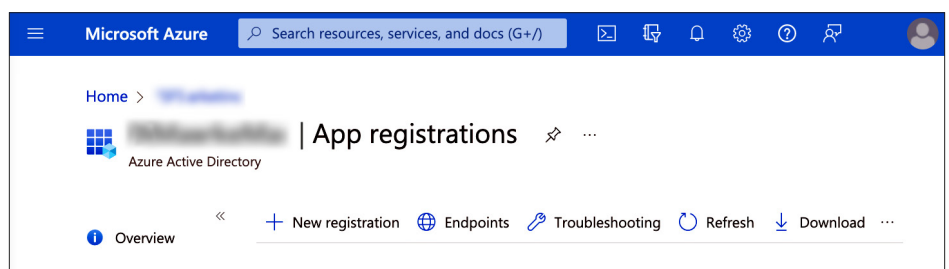
### Configuring Azure AD as the IdP

**Note:** The instructions and screenshots in this section are accurate as of the time of publication, but are subject to change by Microsoft.

1. Create a [Microsoft Azure account](#) if you don't already have one.
2. Log in at [Azure Portal](#) and click on the **Azure Active Directory** icon in the **Azure services** section.



3. In the left-hand navigation column, click **App registrations**.
4. On the **App registrations** tab that opens, click **+ New registration**.



5. On the **Register an application** page that opens:

- a) Type a value in the **Name** box (here, **oidc-demo-app**).
- b) Click the top radio button (**Single tenant**) in the **Supported account types** section.
- c) In the **Redirect URI** section:
  - i) Select **Web** from the left-hand drop-down menu.
  - ii) Type **https://bookinfo.example.com/\_codexch** in the right-hand box.
- d) Click the **Register** button.

Microsoft Azure Search resources, services, and docs (G+)

Home > >

## Register an application ...

**\* Name**  
The user-facing display name for this application (this can be changed later).

oidc-demo-app ✓

**Supported account types**  
Who can use this application or access this API?

☒ Accounts in this organizational directory only (F5 Marketing only - Single tenant)  
☐ Accounts in any organizational directory (Any Azure AD directory - Multitenant)  
☐ Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)  
☐ Personal Microsoft accounts only

[Help me choose...](#)

**Redirect URI (optional)**  
We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

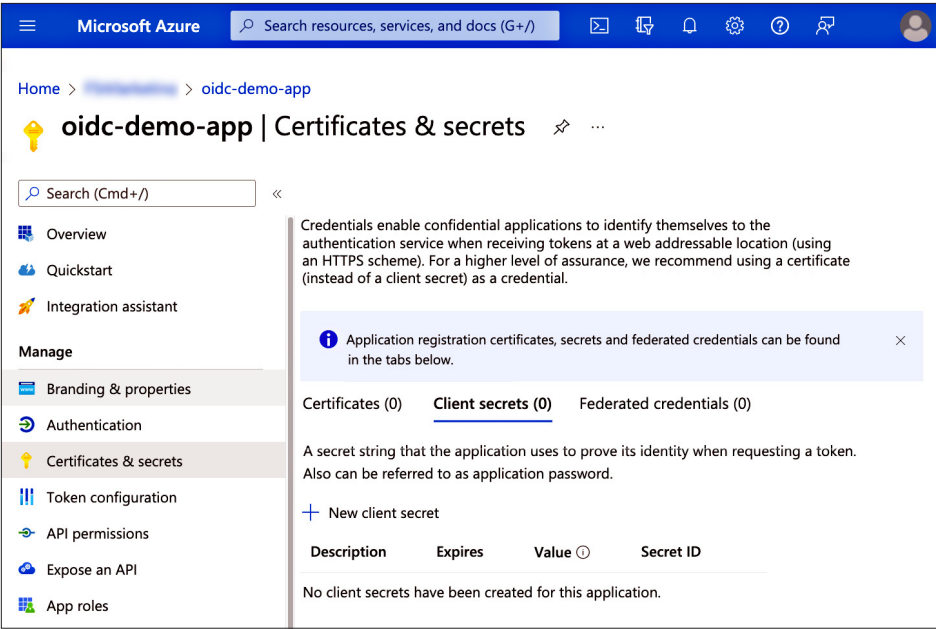
Web ✓ https://bookinfo.example.com/\_codexch ✓

Register an app you're working on here. Integrate gallery apps and other apps from outside your organization by adding from [Enterprise applications](#).

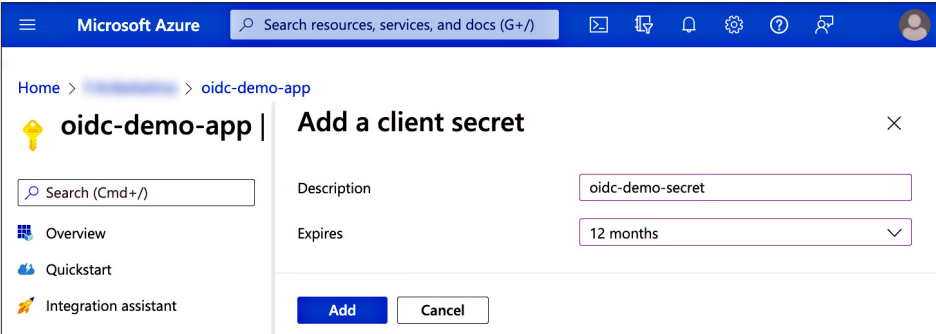
By proceeding, you agree to the [Microsoft Platform Policies](#)

**Register**

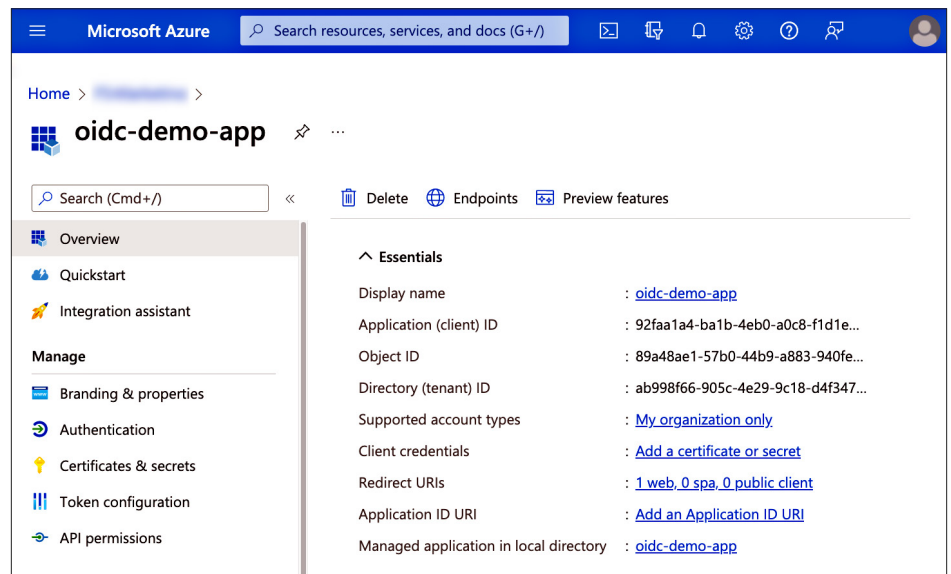
6. A page for the app (here, **oidc-demo-app**) opens. In the left-hand navigation column, click **Certificates & secrets**.
7. On the **Certificates & secrets** tab that appears, click **+ New client secret**.



8. On the **Add a client secret** card that opens, enter a name in the **Description** field (here, **oidc-demo-secret**) and select an expiration time from the **Expires** drop-down menu (here, **12 months**). Click the **Add** button.



9. The new secret appears in the table on the **Certificates & secrets** tab. Copy the character string in the **Value** column to a safe location; you cannot access it after you leave this page.
10. In the left-hand navigation column, click **Overview** and note the values in these fields:
  - **Application (client) ID**
  - **Directory (tenant) ID**



11. In a terminal, Base64-encode the secret you obtained in Step 9.

```
$ echo eqw7Q~jTWK...vFPzYWezxL | base64
ZXF3N1F+a1RX...e1lXZxp4TAo=
```

## Configuring NGINX Ingress Controller as the Relaying Party with Azure AD

1. Edit **Identity-Security/ad-client-secret.yaml**, replacing **client-secret** on line 7 with the Base64-encoded Secret you generated in Step 11 of the previous section (just above):

```
1 apiVersion: v1
2 kind: Secret
3 metadata:
4   name: ad-oidc-secret
5   type: nginx.org/oidc
6   data:
7     client-secret: client-secret
```

[View on GitHub](#)

2. Apply the YAML file containing the Secret:

```
$ kubectl apply -f ./Identity-Security/ad-client-secret.yaml
```

3. Obtain the URLs for the registered app's authorization endpoint, token endpoint, and JSON Web Key (JWK) file. Run the following `curl` command, piping the output to the indicated `python` command to output the entire configuration in an easily readable format. The output is abridged to show only the relevant fields. In the command, make the following substitutions as appropriate:

- For **<tenant>**, substitute the value from the **Directory (tenant) ID** field obtained in [Step 10](#) of the previous section.
- Be sure to include **v2.0** in the path to obtain [Azure AD Endpoint V2 endpoints](#).
- If you are using an [Azure national cloud](#) rather than the Azure “global” cloud, substitute your [Azure AD authentication endpoint](#) for **login.microsoftonline.com**.
- If your app has [custom signing keys](#) because you're using the Azure AD [claims-mapping](#) feature in a multi-tenant environment, also append the **appid** query parameter to get the **jwks\_uri** value that is specific to your app's signing key. For **<app\_id>**, substitute the value from the **Application (client) ID** field obtained in [Step 10](#) of the previous section.

```
$ curl -s https://login.microsoftonline.com/<tenant>/v2.0/.  
well-known/openid-configuration?appid=<app_id>  
{  
  "authorization_endpoint":  
    "https://login.microsoftonline.com/<tenant>/oauth2/v2.0/  
authorize",  
  ...  
  "jwks_uri":  
    "https://login.microsoftonline.com/<tenant>/discovery/v2.0/  
keys?appid=<app_id> ",  
  ...  
  "token_endpoint": "https://login.microsoftonline.com/  
<tenant>/oauth2/v2.0/token",  
  ...  
}
```

4. Edit the NGINX Ingress OIDC Policy in **Identity-Security/ad-oidc-policy.yaml**, replacing the parameters as indicated:

- **ad-client-id** on line 7 – The value from the **Application (client) ID** field obtained in [Step 10](#) of the previous section
- **token** on lines 9–11 – The value used for **<tenant>** in the command in the previous step (as obtained from the **Directory (tenant) ID** field in [Step 10](#) of the previous section)
- **appid** on line 11 – The value used for **<app\_id>** in the command in the previous step (and the same as **ad-client-id** on line 7)

```
1 apiVersion: k8s.nginx.org/v1
2 kind: Policy
3 metadata:
4   name: ad-oidc-policy
5 spec:
6   oidc:
7     clientID: ad-client-id
8     clientSecret: ad-oidc-secret
9     authEndpoint: https://login.microsoftonline.com/token/oauth2/
10    v2.0/authorize
11    tokenEndpoint: https://login.microsoftonline.com/token/oauth2/
12    v2.0/token
13    jwksURI: https://login.microsoftonline.com/token/discovery/v2.0/
14    keys?appid=appid
```

[View on GitHub](#)

5. Apply the policy:

```
$ kubectl apply -f ./Identity-Security/ad-oidc-policy.yaml
```

6. Apply the VirtualServer resource (**Identity-Security/ad-oidc-bookinfo-vs.yaml**) that references **ad-oidc-policy**:

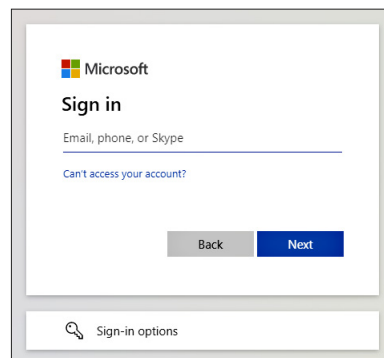
```
$ kubectl apply -f ./Identity-Security/ad-oidc-bookinfo-vs.yaml
```

The VirtualServer resource references **ad-oidc-policy** on line 16. The **path** definition on line 14 means that users who request a URL starting with **/** are authenticated before the request is proxied to the upstream called **backend** (line 18) which maps to the **productpage** service (lines 10–11):

```
1  apiVersion: k8s.nginx.org/v1
2  kind: VirtualServer
3  metadata:
4    name: bookinfo-vs
5  spec:
6    host: bookinfo.example.com
7    tls:
8      secret: bookinfo-secret
9    upstreams:
10   - name: backend
11     service: productpage
12     port: 9080
13   routes:
14   - path: /
15     policies:
16     - name: ad-oidc-policy
17     action:
18       pass: backend
```

[View on GitHub](#)

7. Test the SSO deployment by navigating to **https://bookinfo.example.com** (the hostname of NGINX Ingress Controller) in a browser. You are redirected to the Microsoft login portal, where you can enter the credentials for your Microsoft account to gain access to the backend application (the **productpage** service).



For more information about OIDC with Azure, see the [Microsoft documentation](#).



## IMPLEMENTING SSO WITH PING IDENTITY

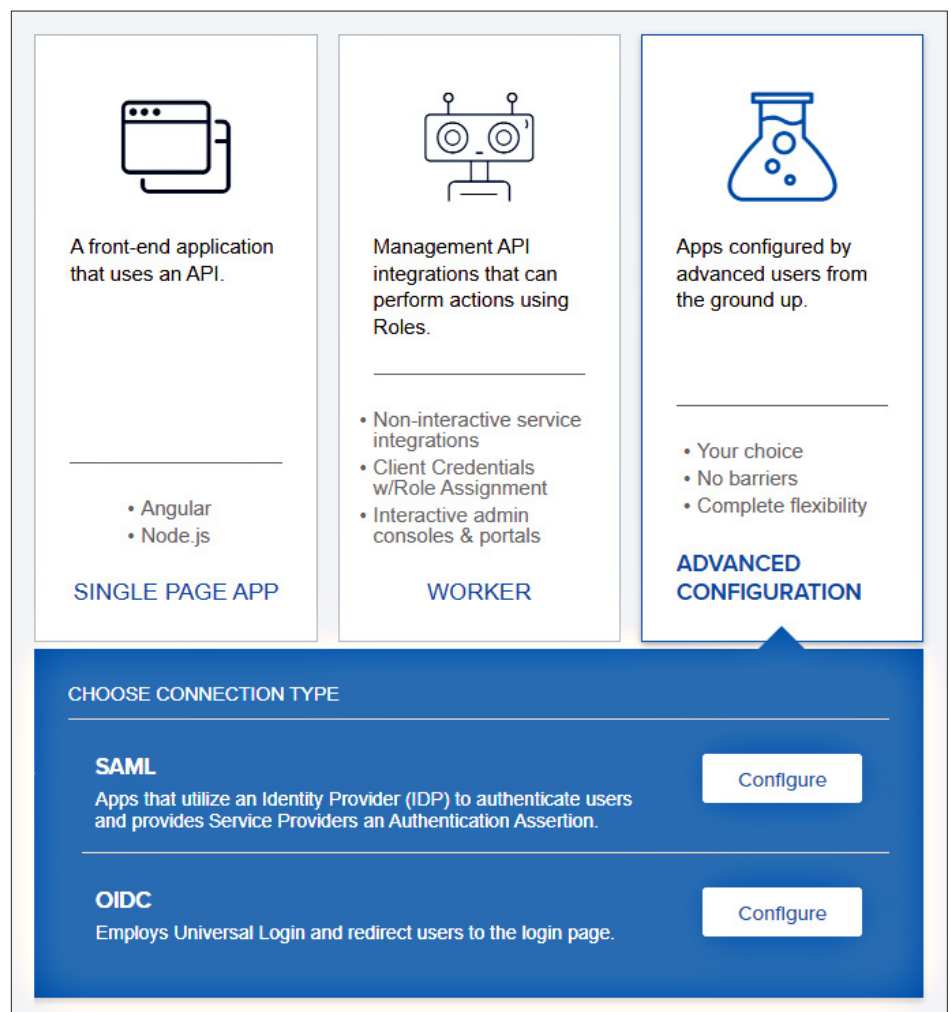
In this section you use the Ping Identity portal to preconfigure Ping Identity as the OIDC identity provider (IdP) for SSO and then configure NGINX Ingress Controller as the relaying party.

- [Configuring Ping Identity as the IdP](#)
- [Configuring NGINX Ingress Controller as the Relaying Party with Ping Identity](#)

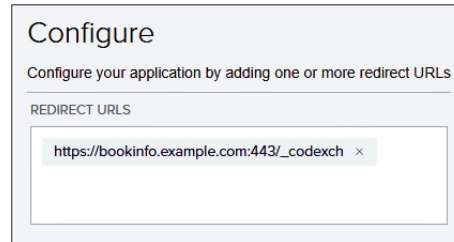
### Configuring Ping Identity as the IdP

**Note:** The instructions and screenshots in this section are accurate for the PingOne for Customers product as of the time of publication, but are subject to change by Ping Identity.

1. Authenticate at the Ping Identity Portal.
2. Navigate to the Applications page and create an application.
3. From the **ADVANCED CONFIGURATION** box, click the **Configure** button for the OIDC connection type.

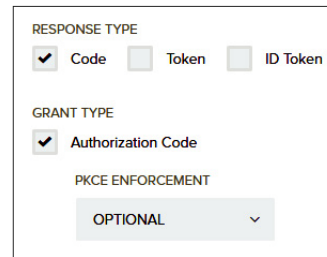


4. In the **REDIRECT URLS** field, type **https://bookinfo.example.com:443/\_codexch**:



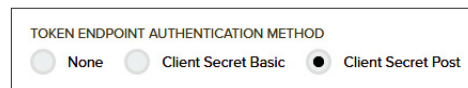
The screenshot shows a 'Configure' dialog box with the instruction 'Configure your application by adding one or more redirect URLs'. Below this is a section titled 'REDIRECT URLS' containing a text input field with the URL 'https://bookinfo.example.com:443/\_codexch' and a close button (x).

5. Edit the application configuration:
- a) In the **RESPONSE TYPE** section, click the **Code** box.
  - b) In the **GRANT TYPE** section, click the **Authorization Code** box.
  - c) In the **PKCE ENFORCEMENT** section, select **OPTIONAL** from the drop-down menu.



The screenshot shows three sections of the configuration page: 'RESPONSE TYPE' with radio buttons for 'Code' (checked), 'Token', and 'ID Token'; 'GRANT TYPE' with radio buttons for 'Authorization Code' (checked) and another unlabeled option; and 'PKCE ENFORCEMENT' with a dropdown menu currently showing 'OPTIONAL'.

6. In the **TOKEN ENDPOINT AUTHENTICATION METHOD** section, click the **Client Secret Post** radio button.



The screenshot shows the 'TOKEN ENDPOINT AUTHENTICATION METHOD' section with three radio buttons: 'None', 'Client Secret Basic', and 'Client Secret Post' (which is selected).

7. Access the **Configuration** tab for the sample application (here, **demo-oidc**) and note the following:
  - In the **URL** section, the alphanumeric string that follows **https://auth.pingone.com/** in each field (in the screenshot, **21b8bf42-a22a...**):
  - In the **GENERAL** section, the value in both the **CLIENT ID** and **CLIENT SECRET** fields (to see the actual client secret, click the eye icon).

**demo-oidc**  
 Client ID: 42cc2908-13cd-4930-ba07-a436302a

Profile
 Configuration
 Resources
 Policies
 Attribute Mappings
 Access

^ URL
 

AUTHORIZATION URL:	https://auth.pingone.com/21b8bf42-a22a-4b9e-bb2a-95a061/as/authorize
TOKEN ENDPOINT:	https://auth.pingone.com/21b8bf42-a22a-4b9e-bb2a-95a061/as/token
JWKS ENDPOINT:	https://auth.pingone.com/21b8bf42-a22a-4b9e-bb2a-95a061/as/jwks
USERINFO ENDPOINT:	https://auth.pingone.com/21b8bf42-a22a-4b9e-bb2a-95a061/as/userinfo
SIGNOFF ENDPOINT:	https://auth.pingone.com/21b8bf42-a22a-4b9e-bb2a-95a061/as/signoff
OIDC DISCOVERY ENDPOINT:	https://auth.pingone.com/21b8bf42-a22a-4b9e-bb2a-95a061/as/.well-known
TOKEN INTROSPECTION ENDPOINT:	https://auth.pingone.com/21b8bf42-a22a-4b9e-bb2a-95a061/as/introspect
TOKEN REVOCATION ENDPOINT:	https://auth.pingone.com/21b8bf42-a22a-4b9e-bb2a-95a061/as/revoke
ISSUER:	https://auth.pingone.com/21b8bf42-a22a-4b9e-bb2a-95a061/as

^ GENERAL
 

CLIENT ID:	42cc2908-13cd-4930-ba07-a436302a
CLIENT SECRET:	.....

8. In a terminal, Base64-encode the secret you obtained from the **CLIENT SECRET** field in the previous step.

```
$ echo e4sZ0f...cG0aXVdywK | base64
YWE2M0E+zkPW...dkkWYWo3SZn=
```

## Configuring NGINX Ingress Controller as the Relaying Party with Ping Identity

1. Edit **Identity-Security/ping-client-secret.yaml**, replacing the `client-secret` parameter on line 7 with the Base64-encoded Kubernetes Secret you generated in Step 8 of the previous section (just above):

```
1 apiVersion: v1
2 kind: Secret
3 metadata:
4   name: ping-oidc-secret
5 type: nginx.org/oidc
6 data:
7   client-secret: client-secret
```

[View on GitHub](#)

2. Apply the YAML file containing the Secret:

```
$ kubectl apply -f ./Identity-Security/ping-client-secret.yaml
```

3. Edit the NGINX Ingress OIDC Policy in **Identity-Security/ping-oidc-policy.yaml**, replacing the parameters as indicated:

- **ping-client-id** on line 7 – The value from the **CLIENT ID** field obtained in [Step 7](#) of *Configuring Ping Identity as the IdP*
- **token** on lines 9–11 – The alphanumeric string from the items in the **URL** section obtained in [Step 7](#) of *Configuring Ping Identity as the IdP* (in this example, it is **21b8bf42-a22a...**)

```
1 apiVersion: k8s.nginx.org/v1
2 kind: Policy
3 metadata:
4   name: ping-oidc-policy
5 spec:
6   oidc:
7     clientId: ping-client-id
8     clientSecret: ping-oidc-secret
9     authEndpoint: https://auth.pingone.com/token/as/authorize
11    tokenEndpoint: https://auth.pingone.com/token/as/token
12    jwksURI: https://auth.pingone.com/token/as/jwks
```

[View on GitHub](#)

4. Apply the policy:

```
$ kubectl apply -f ./Identity-Security/ping-oidc-policy.yaml
```

5. Apply the VirtualServer resource (**Identity-Security/ping-oidc-bookinfo-vs.yaml**) that references **ping-oidc-policy**:

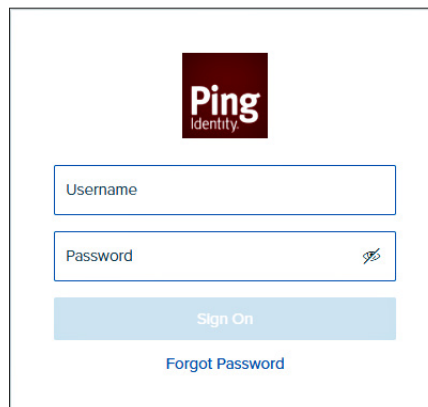
```
$ kubectl apply -f ./Identity-Security/ping-oidc-bookinfo-vs.yaml
```

The VirtualServer resource references **ping-oidc-policy** on line 16. The **path** definition on line 14 means that users who request a URL starting with **/** are authenticated before the request is proxied to the upstream called **backend** (line 18) which maps to the **productpage** service (lines 10–11):

```
1  apiVersion: k8s.nginx.org/v1
2  kind: VirtualServer
3  metadata:
4    name: bookinfo-vs
5  spec:
6    host: bookinfo.example.com
7    tls:
8      secret: bookinfo-secret
9    upstreams:
10     - name: backend
11       service: productpage
12       port: 9080
13    routes:
14     - path: /
15       policies:
16        - name: ping-oidc-policy
17        action:
18          pass: backend
```

[View on GitHub](#)

6. Test the SSO deployment by navigating to **https://bookinfo.example.com** (the hostname of NGINX Ingress Controller) in a browser. You are redirected to the Ping Identity login portal, where you can enter the credentials for your Ping Identity account to gain access to the backend application (the **productpage** service).

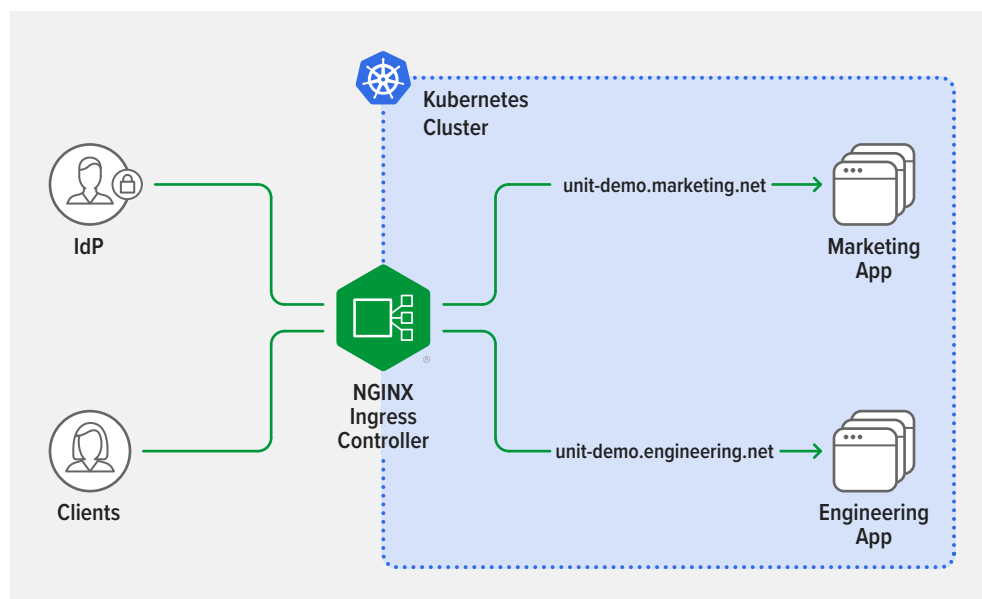
A screenshot of the Ping Identity login portal. At the top center is the Ping Identity logo, which consists of a red square with the word "Ping" in white and "Identity" in smaller white text below it. Below the logo are two input fields: "Username" and "Password". The "Password" field has a small icon of a crossed-out key on its right side. Below these fields is a blue "Sign On" button. At the bottom, there is a blue link that says "Forgot Password".

WHAT IF YOUR SYSTEM  
SCALES TO TENS OR EVEN  
HUNDREDS OF APPLICATIONS  
THAT USERS NEED TO ACCESS  
USING THE SAME SET OF  
CREDENTIALS?

## IMPLEMENTING SSO FOR MULTIPLE APPS

In the previous three sections, we showed how to move the authentication process from the application layer to three third-party OIDC IdPs (Okta, Microsoft Azure Active Directory, and Ping Identity). What if we want to enable each user to use the same set of credentials to access more than one application or application service? What if your system scales to tens or even hundreds of applications that users need to access using the same set of credentials? Single sign-on (SSO) is the solution that addresses this problem.

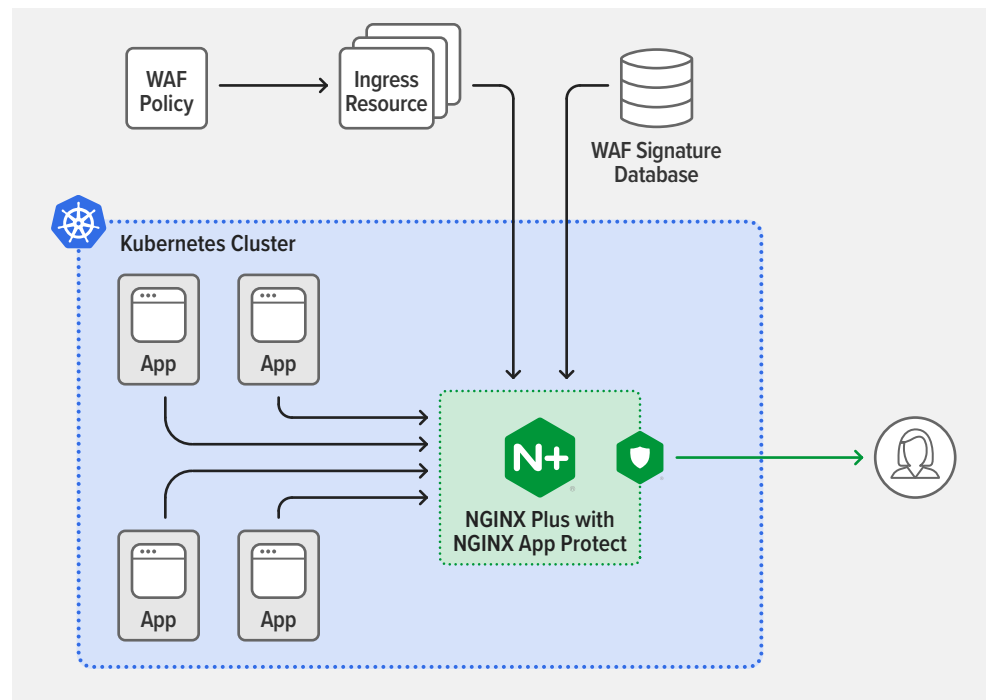
You can easily add new application integrations by defining other OIDC policies if necessary and referencing the policies in VirtualServer resources. In the following diagram, there are two subdomains, **unit-demo.marketing.net** and **unit-demo.engineering.net**, which both resolve to the external IP address of NGINX Ingress Controller. NGINX Ingress Controller routes requests to either the **Marketing** app or the **Engineering** app based on the subdomain. Once the identity of a user is verified, the user can access both applications until the session ID token issued from the IdP is expired or no longer valid.



LET'S LOOK AT HOW TO  
IMPROVE APPLICATION  
SECURITY

## DEPLOYING NGINX APP PROTECT WITH NGINX INGRESS CONTROLLER

Having discussed authentication approaches, let's look at how to improve application security. **NGINX App Protect Web Application Firewall (WAF)** provides advanced protection for apps and APIs against attacks by bad actors, with minimal configuration and management overhead. The diagram shows how NGINX App Protect WAF can be embedded in NGINX Ingress Controller:



### Why Is Integrating a WAF into NGINX Ingress Controller So Significant?

There are several benefits from integrating NGINX App Protect WAF into NGINX Ingress Controller:

- **Securing the application perimeter** – In a well-architected Kubernetes deployment, the Ingress controller is the sole point of entry for data-plane traffic flowing to services running within Kubernetes, making it an ideal location for security enforcement.
- **Consolidating the data plane** – Embedding the WAF within the Ingress controller eliminates the need for a separate WAF device. This reduces complexity, cost, and the number of points of failure.

- **Consolidating the control plane** – WAF configuration can be managed with the Kubernetes API, making it significantly easier to automate CI/CD processes. The Ingress controller configuration complies with Kubernetes role-based access control (RBAC) practices, so you can securely delegate the WAF configuration to a dedicated DevSecOps team.

The configuration objects for NGINX App Protect are consistent across both NGINX Ingress Controller (using YAML files) and NGINX Plus (using JSON). A master configuration can easily be translated and deployed to either device, making it even easier to manage WAF configuration as code and deploy it to any application environment.

## Configuring NGINX App Protect in NGINX Ingress Controller

You installed NGINX App Protect along with NGINX Ingress Controller in Step 3 of [Installation and Deployment Instructions for NGINX Ingress Controller](#).

NGINX App Protect is configured in NGINX Ingress Controller with three custom resources:

- **APPolicy** defines a WAF policy for NGINX App Protect to apply. An APPolicy WAF policy is the YAML version of a standalone, JSON-formatted NGINX App Protect policy.
- **APLogConf** defines the logging behavior of NGINX App Protect.
- **APUserSig** defines a custom signature to protect against an attack type not covered by the [standard signature sets](#).

The NGINX Ingress Controller image also includes an NGINX App Protect signature set, which is embedded at build time.

NGINX App Protect policies protect your web applications against many threats, including the [OWASP Top Ten](#), cross-site scripting (XSS), SQL injections, evasion techniques, information leakage (with [Data Guard](#)), and more.

You can configure NGINX App Protect in two ways using either [NGINX Ingress resources](#) or the [standard Ingress resource](#).



## Configuring NGINX App Protect WAF with NGINX Ingress Resources

To configure NGINX App Protect WAF with NGINX Ingress resources, you define policies, logging configuration, and custom signatures with resources like the following from the eBook repo. You then reference these resources in an NGINX Ingress Controller Policy resource and the Policy resource in a VirtualServer resource.

This APPolicy resource (**Identity-Security/app-protect/ap-dataguard-alarm-policy.yaml**) enables Data Guard protection in blocking mode (lines 17–21).

```
1 apiVersion: appprotect.f5.com/v1beta1
2 kind: APPolicy
3 metadata:
4   name: dataguard-alarm
5 spec:
6   policy:
16     applicationLanguage: utf-8
17     blocking-settings:
18       violations:
19         - alarm: true
20           block: false
21           name: VIOL_DATA_GUARD
22     data-guard:
23       creditCardNumbers: true
24       enabled: true
25       enforcementMode: ignore-urls-in-list
29       maskData: true
30       usSocialSecurityNumbers: true
31     enforcementMode: blocking
32     name: dataguard-alarm
33     template:
34       name: POLICY_TEMPLATE_NGINX_BASE
```

[View on GitHub](#)

The accompanying APLogConf resource (**Identity-Security/app-protect/ap-logconf.yaml**) configures the log to include entries for all requests.

```
1 apiVersion: appprotect.f5.com/v1beta1
2 kind: APLogConf
3 metadata:
4   name: logconf
5 spec:
6   content:
7     format: default
8     max_message_size: 64k
9     max_request_size: any
10  filter:
11    request_type: all
```

[View on GitHub](#)

This APUserSig resource (**Identity-Security/app-protect/ap-apple-uds.yaml**) defines a signature that blocks all requests with a string payload that matches the regex **apple** (line 13).

```
1 apiVersion: appprotect.f5.com/v1beta1
2 kind: APUserSig
3 metadata:
4   name: apple
5 spec:
6   signatures:
7   - accuracy: medium
8     attackType:
9       name: Brute Force Attack
10      description: Medium accuracy user defined signature with tag (Fruits)
11      name: Apple_medium_acc
12      risk: medium
13      rule: content:"apple"; nocase;
14      signatureType: request
15      systems:
16      - name: Microsoft Windows
17      - name: Unix/Linux
18    tag: Fruits
```

[View on GitHub](#)

The APPolicy and APLogConf resources are applied by references to **dataguard-alarm** on line 8 and **logconf** on line 11 in the Policy resource defined in **Identity-Security/app-protect/waf.yaml**:

```
1 apiVersion: k8s.nginx.org/v1
2 kind: Policy
3 metadata:
4   name: waf-policy
5 spec:
6   waf:
7     enable: true
8     apPolicy: "default/dataguard-alarm"
9     securityLog:
10       enable: true
11       apLogConf: "default/logconf"
12       logDest: "syslog:server=syslog-svc.default:514"
```

[View on GitHub](#)

The WAF policy is applied in turn by a reference to it on line 8 of the VirtualServer resource defined in **Identity-Security/app-protect/bookinfo-vs.yaml**.

```
1 apiVersion: k8s.nginx.org/v1
2 kind: VirtualServer
3 metadata:
4   name: bookinfo-vs
5 spec:
6   host: bookinfo.example.com
7   policies:
8     - name: waf-policy
```

[View on GitHub](#)

1. Apply the APPolicy, APLogConf, and APUserSig resources.

```
$ kubectl apply -f ./Identity-Security/app-protect/ap-dataguard-
alarm-policy.yaml
$ kubectl apply -f ./Identity-Security/app-protect/ap-logconf.yaml
$ kubectl apply -f ./Identity-Security/app-protect/ap-apple-uds.yaml
```

2. Activate the APPolicy, APLogConf, and APUserSig resources by applying the WAF Policy resource (defined in **waf.yaml**) that references them and the VirtualServer resource (defined in **bookinfo-vs.yaml**) that references the Policy.

```
$ kubectl apply -f ./Identity-Security/app-protect/waf.yaml
$ kubectl apply -f ./Identity-Security/app-protect/bookinfo-vs.yaml
```

After the resources are referenced, App Protect inspects and potentially blocks all requests handled by NGINX Ingress Controller. With this approach, administrators can have ownership over the full scope of the Ingress configuration with VirtualServer resources, while delegating responsibilities to other teams who reference Policy resources. Additionally, administrators can leverage Kubernetes RBAC to configure [namespace](#) and [resource isolation](#) among teams such as NetOps Admin, DevOps, and DevSecOps.



## Configuring NGINX App Protect WAF with the Standard Ingress Resource

As an alternative to NGINX Ingress resources, you can use annotations in a standard Kubernetes Ingress resource to reference NGINX App Protect policies, as in this example (**Identity-Security/app-protect/bookinfo-ingress.yaml**):

```
1 apiVersion: networking.k8s.io/v1beta1
2 kind: Ingress
3 metadata:
4   name: bookinfo-ingress
5   annotations:
6     appprotect.f5.com/app-protect-policy: "default/dataguard-alarm"
7     appprotect.f5.com/app-protect-enable: "True"
8     appprotect.f5.com/app-protect-security-log-enable: "True"
9     appprotect.f5.com/app-protect-security-log: "default/logconf"
10    appprotect.f5.com/app-protect-security-log-destination:
        "syslog:server=syslog-svc.default:514"
```

[View on GitHub](#)

To activate the policy and log settings, apply the Ingress resource:

```
$ kubectl apply -f ./Identity-Security/app-protect/bookinfo-ingress.yaml
```

## Logging

The logs for NGINX App Protect WAF and NGINX Ingress Controller are separate by design, to accommodate the delegation of responsibility to different teams such as DevSecOps and application owners.

NGINX Ingress Controller logs are forwarded to the local standard output, as for all Kubernetes containers.

To send NGINX App logs to a syslog destination:

- If using NGINX Ingress resources, set the **logDest** field of the WAF Policy resource to the cluster IP address of the syslog service.
- If you are using a standard Kubernetes Ingress resource, set the **app-protect-security-log-destination** annotation to the cluster IP address of the syslog service.

In the APLogConf resource you can specify which logs to push to the syslog pod.

Here's a sample syslog deployment (**Identity-Security/app-protect/syslog.yaml**):

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: syslog
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: syslog
10   template:
11     metadata:
12       labels:
13         app: syslog
14     spec:
15       containers:
16       - name: syslog
17         image: balabit/syslog-ng:3.35.1
18         ports:
19           - containerPort: 514
20           - containerPort: 601
21   ---
22   apiVersion: v1
23   kind: Service
24   metadata:
25     name: syslog-svc
26   spec:
27     ports:
28       - port: 514
29         targetPort: 514
30         protocol: TCP
31     selector:
32       app: syslog
```

[View on GitHub](#)

To configure logging to syslog, apply the resource:

```
$ kubectl apply -f ./Identity-Security/app-protect/syslog.yaml
```

YOU CAN ALSO USE  
NGINX APP PROTECT  
TO SET RESOURCE  
PROTECTION THRESHOLDS

## Resource Thresholds

You can also use NGINX App Protect to set resource protection thresholds for both CPU and memory utilization by NGINX App Protect processes. This is particularly important in multi-tenant environments such as Kubernetes which rely on resource sharing and can potentially suffer from the “noisy neighbor” problem. The following sample ConfigMap sets resource thresholds:

```
1 kind: ConfigMap
2 apiVersion: v1
3 metadata:
4   name: nginx-config
5   namespace: nginx-ingress
6 data:
7   app_protect_physical_memory_util_thresholds: "high=100 low=10"
8   app_protect_cpu_thresholds: "high=100 low=50"
9   app_protect_failure_mode_action: "drop"
```

For thresholds with **high** and **low** parameters, the former parameter sets the percent utilization at which App Protect enters failure mode and the latter the percent utilization at which it exits failure mode. Here the **high** and **low** parameters are set to 100% and 10% respectively for memory utilization and to 100% and 50% for CPU utilization.

The **app\_protect\_failure\_mode\_action** field controls how NGINX App Protect handles requests while in failure mode:

- **drop** – App Protect rejects requests, returning **503 (Service Unavailable)** and closing the connection
- **pass** – App Protect forwards requests without inspecting them or enforcing any policies

VERIFYING USER IDENTITY  
AND ENFORCING SECURITY  
ARE KEY TO SUCCESSFUL  
OPERATION FOR ALL TYPES  
OF USERS AND APPLICATIONS

## CHAPTER SUMMARY

Verifying user identity and enforcing security are key to successful operation for all types of users and applications. We showed how to implement authentication of user identities by integrating with third-party OIDC IdPs and how to secure applications with NGINX App Protect WAF.

Let's summarize some of the key concepts from this chapter:

- Authenticating user identities and enforcing authorization controls are as crucial for protecting applications and APIs as any other protection technique.
- Offloading authentication and authorization from applications to a third-party identity provider provides not only a wealth of benefits for app developers, admins, and end users alike, but is also easy to implement and a wise architectural choice for most implementations.
- Working with an OIDC identity provider (IdP), NGINX Ingress Controller operates as the relaying party which enforces access controls on incoming traffic and routes it to the appropriate services in the cluster.
- Integrating NGINX Ingress Controller with an IdP involves configuring the IdP to recognize the application, defining NGINX Policy resources, and referencing the policies in NGINX VirtualServer resources.
- NGINX App Protect integrates into NGINX Ingress Controller to secure the application perimeter quickly and reliably, helping to protect web applications against bad actors.