**ISTIO – installation on kubernetes cluster**

**With bookapp example**

**www.zippyops.com**

172-172, 5th floor Old Mahabalipuram Road

(Above Axis Bank-PTC Bus Stop)  
Thuraipakkam  
Chennai 600097

🖂[zippyops@gmail.com](mailto:zippyops@gmail.com)

✆+91 7010585768

Cloud platforms provide a wealth of benefits for the organizations that use them. However, there’s no denying that adopting the cloud can put strains on DevOps teams. Developers must use microservices to architect for portability, meanwhile operators are managing extremely large hybrid and multi-cloud deployments. Istio lets you connect, secure, control, and observe services.

At a high level, Istio helps reduce the complexity of these deployments, and eases the strain on your development teams. It is a completely open source service mesh that layers transparently onto existing distributed applications. It is also a platform, including APIs that let it integrate into any logging platform, or telemetry or policy system. Istio’s diverse feature set lets you successfully, and efficiently, run a distributed microservice architecture, and provides a uniform way to secure, connect, and monitor microservices.

# Prerequisites

For this lab, we will use a standard centos 7 installation as a base image for the 3 machines needed. The machines will all be configured on the same network, this network needs to have access to the Internet.

We also need one Kubernetes master nodes. This machine will have the IP 192.168.1.163.

Finally, we will also have two Kubernetes worker nodes with the IPs 192.168.1.164 and 192.168.1.165.

For this lab we are going to use one master and two worker nodes that will work fine.

## System requirments

* 1-Master => 2 CPU , 4GB RAM
* 1-Nodes => 1 CPU, 1GB RAM

# What is a service mesh?

Istio addresses the challenges developers and operators face as monolithic applications transition towards a distributed microservice architecture. To see how, it helps to take a more detailed look at Istio’s service mesh.

The term service mesh is used to describe the network of microservices that make up such applications and the interactions between them. As a service mesh grows in size and complexity, it can become harder to understand and manage. Its requirements can include discovery, load balancing, failure recovery, metrics, and monitoring. A service mesh also often has more complex operational requirements, like A/B testing, canary rollouts, rate limiting, access control, and end-to-end authentication.

Istio provides behavioral insights and operational control over the service mesh as a whole, offering a complete solution to satisfy the diverse requirements of microservice applications.

# Why use Istio?

Istio makes it easy to create a network of deployed services with load balancing, service-to-service authentication, monitoring, and more, with [few](https://istio.io/docs/tasks/telemetry/distributed-tracing/overview/#understanding-what-happened) or no code changes in service code. You add Istio support to services by deploying a special sidecar proxy throughout your environment that intercepts all network communication between microservices, then configure and manage Istio using its control plane functionality, which includes:

* Automatic load balancing for HTTP, gRPC, WebSocket, and TCP traffic.
* Fine-grained control of traffic behavior with rich routing rules, retries, failovers, and fault injection.
* A pluggable policy layer and configuration API supporting access controls, rate limits and quotas.
* Automatic metrics, logs, and traces for all traffic within a cluster, including cluster ingress and egress.
* Secure service-to-service communication in a cluster with strong identity-based authentication and authorization.
* Istio is designed for extensibility and meets diverse deployment needs.

# Core features

Istio provides a number of key capabilities uniformly across a network of services:

## Traffic management

Istio’s easy rules configuration and traffic routing lets you control the flow of traffic and API calls between services. Istio simplifies configuration of service-level properties like circuit breakers, timeouts, and retries, and makes it a breeze to set up important tasks like A/B testing, canary rollouts, and staged rollouts with percentage-based traffic splits.

With better visibility into your traffic, and out-of-box failure recovery features, you can catch issues before they cause problems, making calls more reliable, and your network more robust – no matter what conditions you face.

## Security

Istio’s security capabilities free developers to focus on security at the application level. Istio provides the underlying secure communication channel, and manages authentication, authorization, and encryption of service communication at scale. With Istio, service communications are secured by default, letting you enforce policies consistently across diverse protocols and runtimes – all with little or no application changes.

While Istio is platform independent, using it with Kubernetes (or infrastructure) network policies, the benefits are even greater, including the ability to secure pod-to-pod or service-to-service communication at the network and application layers.

## Observability

Istio’s robust tracing, monitoring, and logging features give you deep insights into your service mesh deployment. Gain a real understanding of how service performance impacts things upstream and downstream with Istio’s monitoring features, while its custom dashboards provide visibility into the performance of all your services and let you see how that performance is affecting your other processes.

Istio’s Mixer component is responsible for policy controls and telemetry collection. It provides backend abstraction and intermediation, insulating the rest of Istio from the implementation details of individual infrastructure backends, and giving operators fine-grained control over all interactions between the mesh and infrastructure backends.

All these features let you more effectively set, monitor, and enforce SLOs on services. Of course, the bottom line is that you can detect and fix issues quickly and efficiently.

## Platform support

Istio is platform-independent and designed to run in a variety of environments, including those spanning Cloud, on-premise, Kubernetes, Mesos, and more. You can deploy Istio on Kubernetes, or on Nomad with Consul. Istio currently supports:

* Service deployment on Kubernetes
* Services registered with Consul
* Services running on individual virtual machines

## Integration and customization

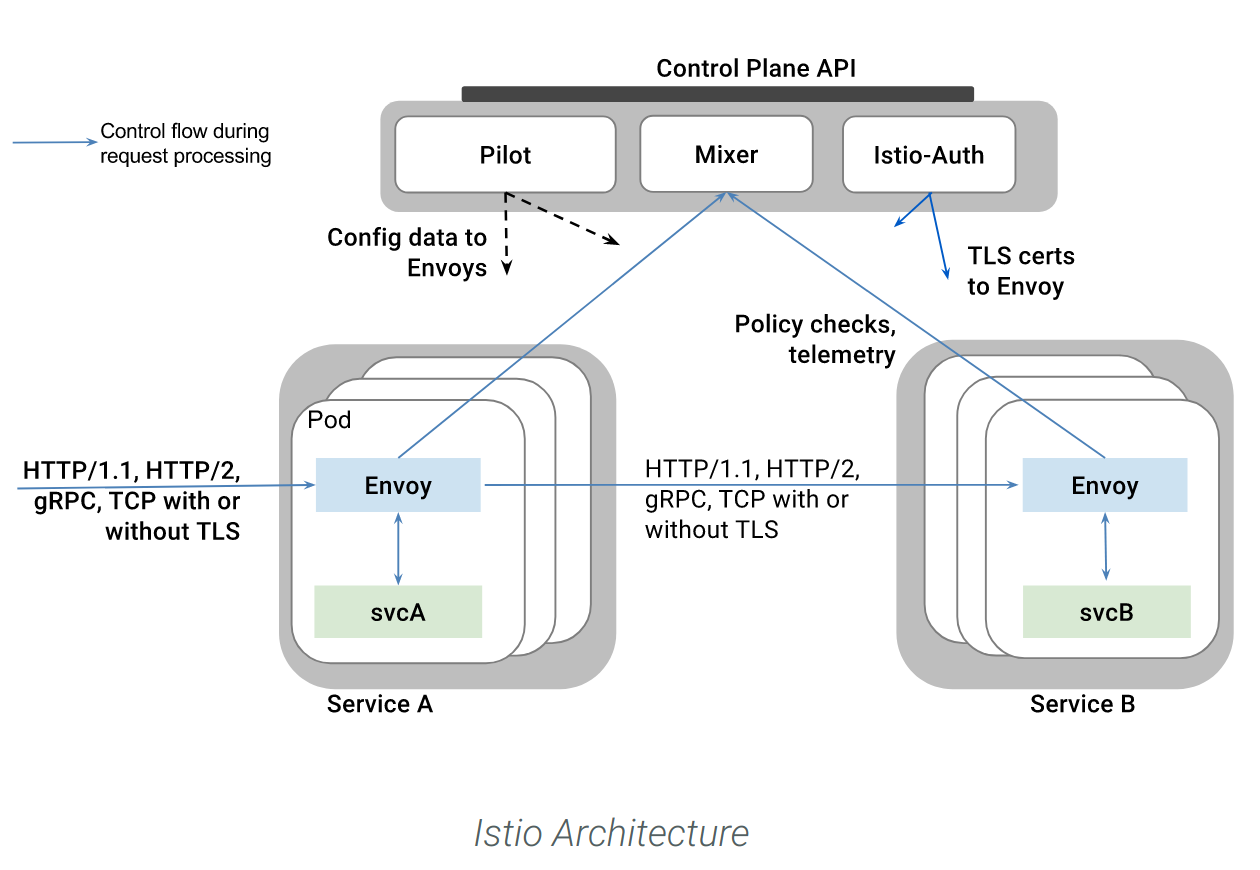
The policy enforcement component of Istio can be extended and customized to integrate with existing solutions for ACLs, logging, monitoring, quotas, auditing, and more.

# Architecture

An Istio service mesh is logically split into a data plane and a control plane.

* The **data plane** is composed of a set of intelligent proxies ([Envoy](https://www.envoyproxy.io/)) deployed as sidecars. These proxies mediate and control all network communication between microservices along with [Mixer](https://istio.io/docs/concepts/policies-and-telemetry/), a general-purpose policy and telemetry hub.
* The **control plane** manages and configures the proxies to route traffic. Additionally, the control plane configures Mixers to enforce policies and collect telemetry.

The following diagram shows the different components that make up each plane:



## Envoy

Istio uses an extended version of the [Envoy](https://envoyproxy.github.io/envoy/) proxy. Envoy is a high-performance proxy developed in C++ to mediate all inbound and outbound traffic for all services in the service mesh. Istio leverages Envoy’s many built-in features, for example:

* Dynamic service discovery
* Load balancing
* TLS termination
* HTTP/2 and gRPC proxies
* Circuit breakers
* Health checks
* Staged rollouts with %-based traffic split
* Fault injection
* Rich metrics

Envoy is deployed as a sidecar to the relevant service in the same Kubernetes pod. This deployment allows Istio to extract a wealth of signals about traffic behavior as [attributes](https://istio.io/docs/concepts/policies-and-telemetry/#attributes). Istio can, in turn, use these attributes in [Mixer](https://istio.io/docs/concepts/policies-and-telemetry/) to enforce policy decisions, and send them to monitoring systems to provide information about the behavior of the entire mesh.

The sidecar proxy model also allows you to add Istio capabilities to an existing deployment with no need to rearchitect or rewrite code. You can read more about why we chose this approach in our [Design Goals](https://istio.io/docs/concepts/what-is-istio/#design-goals).

## Mixer

[Mixer](https://istio.io/docs/concepts/policies-and-telemetry/) is a platform-independent component. Mixer enforces access control and usage policies across the service mesh, and collects telemetry data from the Envoy proxy and other services. The proxy extracts request level [attributes](https://istio.io/docs/concepts/policies-and-telemetry/#attributes), and sends them to Mixer for evaluation. You can find more information on this attribute extraction and policy evaluation in our [Mixer Configuration documentation](https://istio.io/docs/concepts/policies-and-telemetry/#configuration-model).

Mixer includes a flexible plugin model. This model enables Istio to interface with a variety of host environments and infrastructure backends. Thus, Istio abstracts the Envoy proxy and Istio-managed services from these details.

## Pilot

[Pilot](https://istio.io/docs/concepts/traffic-management/#pilot-and-envoy) provides service discovery for the Envoy sidecars, traffic management capabilities for intelligent routing (e.g., A/B tests, canary rollouts, etc.), and resiliency (timeouts, retries, circuit breakers, etc.).

Pilot converts high level routing rules that control traffic behavior into Envoy-specific configurations, and propagates them to the sidecars at runtime. Pilot abstracts platform-specific service discovery mechanisms and synthesizes them into a standard format that any sidecar conforming with the [Envoy data plane APIs](https://github.com/envoyproxy/data-plane-api) can consume. This loose coupling allows Istio to run on multiple environments such as Kubernetes, Consul, or Nomad, while maintaining the same operator interface for traffic management.

## Citadel

[Citadel](https://istio.io/docs/concepts/security/) enables strong service-to-service and end-user authentication with built-in identity and credential management. You can use Citadel to upgrade unencrypted traffic in the service mesh. Using Citadel, operators can enforce policies based on service identity rather than on relatively unstable layer 3 or layer 4 network identifiers. Starting from release 0.5, you can use [Istio’s authorization feature](https://istio.io/docs/concepts/security/#authorization) to control who can access your services.

## Galley

Galley is Istio’s configuration validation, ingestion, processing and distribution component. It is responsible for insulating the rest of the Istio components from the details of obtaining user configuration from the underlying platform (e.g. Kubernetes).

## Design Goals

* A few key design goals informed Istio’s architecture. These goals are essential to making the system capable of dealing with services at scale and with high performance.
* **Maximize Transparency**: To adopt Istio, an operator or developer is required to do the minimum amount of work possible to get real value from the system. To this end, Istio can automatically inject itself into all the network paths between services. Istio uses sidecar proxies to capture traffic and, where possible, automatically program the networking layer to route traffic through those proxies without any changes to the deployed application code. In Kubernetes, the proxies are injected into pods and traffic is captured by programming iptables rules. Once the sidecar proxies are injected and traffic routing is programmed, Istio can mediate all traffic. This principle also applies to performance. When applying Istio to a deployment, operators see a minimal increase in resource costs for the functionality being provided. Components and APIs must all be designed with performance and scale in mind.
* **Extensibility**: As operators and developers become more dependent on the functionality that Istio provides, the system must grow with their needs. While we continue to add new features, the greatest need is the ability to extend the policy system, to integrate with other sources of policy and control, and to propagate signals about mesh behavior to other systems for analysis. The policy runtime supports a standard extension mechanism for plugging in other services. In addition, it allows for the extension of its vocabulary to allow policies to be enforced based on new signals that the mesh produces.
* **Portability**: The ecosystem in which Istio is used varies along many dimensions. Istio must run on any cloud or on-premises environment with minimal effort. The task of porting Istio-based services to new environments must be trivial. Using Istio, you are able to operate a single service deployed into multiple environments. For example, you can deploy on multiple clouds for redundancy.
* **Policy Uniformity**: The application of policy to API calls between services provides a great deal of control over mesh behavior. However, it can be equally important to apply policies to resources which are not necessarily expressed at the API level. For example, applying a quota to the amount of CPU consumed by an ML training task is more useful than applying a quota to the call which initiated the work. To this end, Istio maintains the policy system as a distinct service with its own API rather than the policy system being baked into the proxy sidecar, allowing services to directly integrate with it as needed.

# Ports used by Istio

The following ports and protocols are used by Istio. Ensure that there are no TCP headless services using a TCP port used by one of Istio’s services.

| **Port** | **Protocol** | **Used by** | **Description** |
| --- | --- | --- | --- |
| 8060 | HTTP | Citadel | GRPC server |
| 9090 | HTTP | Prometheus | Prometheus |
| 9091 | HTTP | Mixer | Policy/Telemetry |
| 9093 | HTTP | Citadel |  |
| 15000 | TCP | Envoy | Envoy admin port (commands/diagnostics) |
| 15001 | TCP | Envoy | Envoy |
| 15004 | HTTP | Mixer, Pilot | Policy/Telemetry - mTLS |
| 15010 | HTTP | Pilot | Pilot service - XDS pilot - discovery |
| 15011 | TCP | Pilot | Pilot service - mTLS - Proxy - discovery |
| 15014 | HTTP | Citadel, Mixer, Pilot | Control plane monitoring |
| 15030 | TCP | Prometheus | Prometheus |
| 15090 | HTTP | Mixer | Proxy |
| 42422 | TCP | Mixer | Telemetry - Prometheus |

# Download and prepare for the installation

Istio is installed in its own istio-system namespace and can manage services from all other namespaces.

1. Go to the [Istio release](https://github.com/istio/istio/releases" \t "_blank) page to download the installation file corresponding to your OS. On a macOS or Linux system, you can run the following command to download and extract the latest release automatically:

$ curl -L https://git.io/getLatestIstio | ISTIO\_VERSION=1.1.1 sh -

Move to the Istio package directory. For example, if the package is istio-1.1.1:

$ cd istio-1.1.1

The installation directory contains:

* Installation YAML files for Kubernetes in install/
* Sample applications in samples/
* The istioctl client binary in the bin/ directory. istioctl is used when manually injecting Envoy as a sidecar proxy.

The istio.VERSION configuration file

Add the istioctl client to your PATH environment variable, on a macOS or Linux system:

$ export PATH=$PWD/bin:$PATH

# installation steps

Install all the Istio [Custom Resource Definitions](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#customresourcedefinitions) (CRDs) using kubectl apply, and wait a few seconds for the CRDs to be committed in the Kubernetes API-server:

$ for i in install/kubernetes/helm/istio-init/files/crd\*yaml; do kubectl apply -f $i; done

**Install one of the following variants of the demo profile**

**Permissive mutual TLS**

When using the **permissive mutual TLS** mode, all services accept both plain text and mutual TLS traffic. Clients send plain text traffic unless configured for [mutual migration](https://istio.io/docs/tasks/security/mtls-migration/#configure-clients-to-send-mutual-tls-traffic). Visit our [mutual TLS permissive mode page](https://istio.io/docs/concepts/security/#permissive-mode) for more information.

Choose this variant for:

Clusters with existing applications, or

Applications where services with an Istio sidecar need to be able to communicate with other non-Istio Kubernetes services

Run the following command to install this variant:

$ kubectl apply -f install/kubernetes/istio-demo.yaml

**Strict mutual TLS**

This variant will enforce [mutual TLS authentication](https://istio.io/docs/concepts/security/#mutual-tls-authentication) between all clients and servers.

Use this variant only on a fresh Kubernetes cluster where all workloads will be Istio-enabled. All newly deployed workloads will have Istio sidecars installed.

Run the following command to install this variant:

$ kubectl apply -f install/kubernetes/istio-demo-auth.yaml

## Verifying the installation

Ensure the following Kubernetes services are deployed and verify they all have an appropriate CLUSTER-IP except the jaeger-agent service

$ kubectl get svc -n istio-system

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

grafana ClusterIP 172.21.211.123 <none> 3000/TCP 2m

istio-citadel ClusterIP 172.21.177.222 <none> 8060/TCP,9093/TCP 2m

istio-egressgateway ClusterIP 172.21.113.24 <none> 80/TCP,443/TCP 2m

istio-galley ClusterIP 172.21.132.247 <none> 443/TCP,9093/TCP 2m

istio-ingressgateway LoadBalancer 172.21.144.254 52.116.22.242 80:31380/TCP,443:31390/TCP,31400:31400/TCP,15011:32081/TCP,8060:31695/TCP,853:31235/TCP,15030:32717/TCP,15031:32054/TCP 2m

istio-pilot ClusterIP 172.21.105.205 <none> 15010/TCP,15011/TCP,8080/TCP,9093/TCP 2m

istio-policy ClusterIP 172.21.14.236 <none> 9091/TCP,15004/TCP,9093/TCP 2m

istio-sidecar-injector ClusterIP 172.21.155.47 <none> 443/TCP 2m

istio-telemetry ClusterIP 172.21.196.79 <none> 9091/TCP,15004/TCP,9093/TCP,42422/TCP 2m

jaeger-agent ClusterIP None <none> 5775/UDP,6831/UDP,6832/UDP 2m

jaeger-collector ClusterIP 172.21.135.51 <none> 14267/TCP,14268/TCP 2m

jaeger-query ClusterIP 172.21.26.187 <none> 16686/TCP 2m

kiali ClusterIP 172.21.155.201 <none> 20001/TCP 2m

prometheus ClusterIP 172.21.63.159 <none> 9090/TCP 2m

tracing ClusterIP 172.21.2.245 <none> 80/TCP 2m

zipkin ClusterIP 172.21.182.245 <none> 9411/TCP 2m

**Note**: If your cluster is running in an environment that does not support an external load balancer (e.g., minikube), theEXTERNAL-IP of istio-ingressgateway will say <pending>. To access the gateway, use the service’s NodePort, or use port-forwarding instead.

Ensure corresponding Kubernetes pods are deployed and have a STATUS of Running:

$ kubectl get pods -n istio-system

NAME READY STATUS RESTARTS AGE

grafana-f8467cc6-rbjlg 1/1 Running 0 1m

istio-citadel-78df5b548f-g5cpw 1/1 Running 0 1m

istio-cleanup-secrets-release-1.1-20190308-09-16-8s2mp 0/1 Completed 0 2m

istio-egressgateway-78569df5c4-zwtb5 1/1 Running 0 1m

istio-galley-74d5f764fc-q7nrk 1/1 Running 0 1m

istio-grafana-post-install-release-1.1-20190308-09-16-2p7m5 0/1 Completed 0 2m

istio-ingressgateway-7ddcfd665c-dmtqz 1/1 Running 0 1m

istio-pilot-f479bbf5c-qwr28 2/2 Running 0 1m

istio-policy-6fccc5c868-xhblv 2/2 Running 2 1m

istio-security-post-install-release-1.1-20190308-09-16-bmfs4 0/1 Completed 0 2m

istio-sidecar-injector-78499d85b8-x44m6 1/1 Running 0 1m

istio-telemetry-78b96c6cb6-ldm9q 2/2 Running 2 1m

istio-tracing-69b5f778b7-s2zvw 1/1 Running 0 1m

kiali-99f7467dc-6rvwp 1/1 Running 0 1m

prometheus-67cdb66cbb-9w2hm 1/1 Running 0 1m

## Deploy your application

You can now deploy your own application or one of the sample applications provided with the installation like Bookinfo.

**Note:**The application must use either the HTTP/1.1 or HTTP/2.0 protocols for all its HTTP traffic; HTTP/1.0 is not supported.

## Bookinfo application

This example deploys a sample application composed of four separate microservices used to demonstrate various Istio features. The application displays information about a book, similar to a single catalog entry of an online book store. Displayed on the page is a description of the book, book details (ISBN, number of pages, and so on), and a few book reviews.

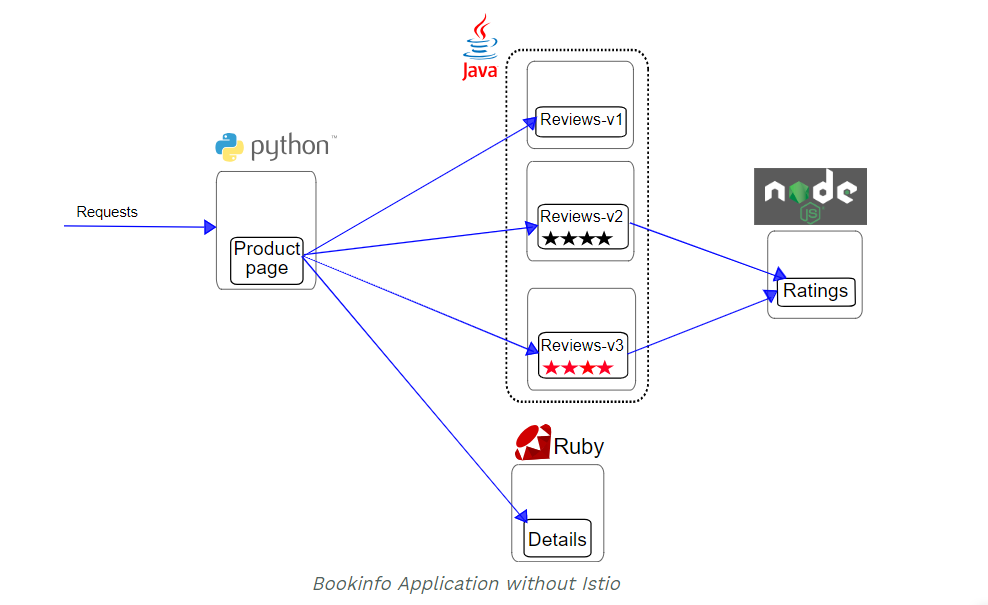
The Bookinfo application is broken into four separate microservices:

* productpage. The productpage microservice calls the details and reviews microservices to populate the page.
* details. The details microservice contains book information.
* reviews. The reviews microservice contains book reviews. It also calls the ratings microservice.
* ratings. The ratings microservice contains book ranking information that accompanies a book review.

There are 3 versions of the reviews microservice:

* Version v1 doesn’t call the ratings service.
* Version v2 calls the ratings service, and displays each rating as 1 to 5 black stars.
* Version v3 calls the ratings service, and displays each rating as 1 to 5 red stars.

The end-to-end architecture of the application is shown below.

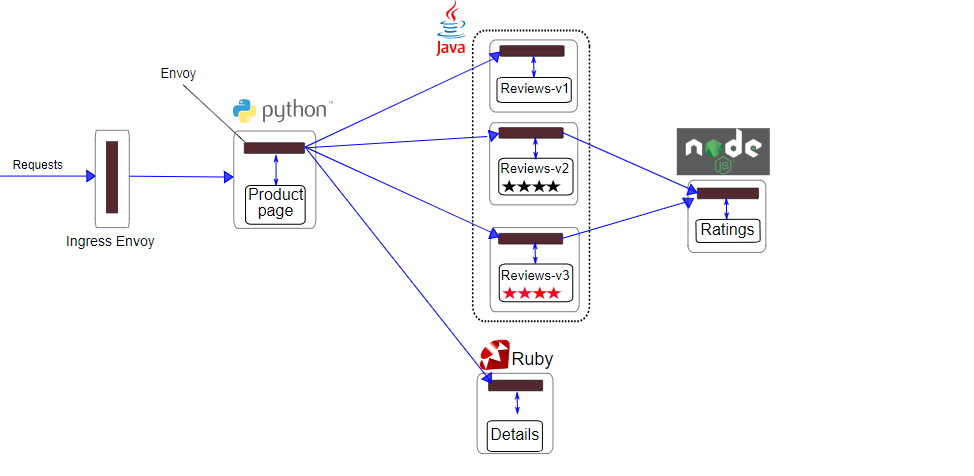


This application is polyglot, i.e., the microservices are written in different languages. It’s worth noting that these services have no dependencies on Istio, but make an interesting service mesh example, particularly because of the multitude of services, languages and versions for the reviews service.

## Before you begin

To run the sample with Istio requires no changes to the application itself. Instead, we simply need to configure and run the services in an Istio-enabled environment, with Envoy sidecars injected along side each service. The needed commands and configuration vary depending on the runtime environment although in all cases the resulting deployment will look like this:

All of the microservices will be packaged with an Envoy sidecar that intercepts incoming and outgoing calls for the services, providing the hooks needed to externally control, via the Istio control plane, routing, telemetry collection, and policy enforcement for the application as a whole.



## If you are running on Kubernetes

Change directory to the root of the Istio installation.

$ cd istio-1.1.1

The default Istio installation uses [automatic sidecar injection](https://istio.io/docs/setup/kubernetes/additional-setup/sidecar-injection/#automatic-sidecar-injection). Label the namespace that will host the application with istio-injection=enabled:

$ kubectl label namespace default istio-injection=enabled

Deploy your application using the kubectl command:

$ kubectl apply -f [samples/bookinfo/platform/kube/bookinfo.yaml](https://raw.githubusercontent.com/istio/istio/release-1.1/samples/bookinfo/platform/kube/bookinfo.yaml)

**Note**: If you disabled automatic sidecar injection during installation and rely on [manual sidecar injection](https://istio.io/docs/setup/kubernetes/additional-setup/sidecar-injection/#manual-sidecar-injection), use the istioctl kube-inject command to modify the bookinfo.yaml file before deploying your application. For more information please visit the istioctl [reference documentation](https://istio.io/docs/reference/commands/istioctl/#istioctl-kube-inject).

$ istioctl kube-inject -f samples/bookinfo/platform/kube/bookinfo.yaml | kubectl apply -f -

The command launches all four services shown in the bookinfo application architecture diagram. All 3 versions of the reviews service, v1, v2, and v3, are started.

Confirm all services and pods are correctly defined and running:

$ kubectl get services

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

details 10.0.0.31 <none> 9080/TCP 6m

kubernetes 10.0.0.1 <none> 443/TCP 7d

productpage 10.0.0.120 <none> 9080/TCP 6m

ratings 10.0.0.15 <none> 9080/TCP 6m

reviews 10.0.0.170 <none> 9080/TCP 6m

and

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

details-v1-1520924117-48z17 2/2 Running 0 6m

productpage-v1-560495357-jk1lz 2/2 Running 0 6m

ratings-v1-734492171-rnr5l 2/2 Running 0 6m

reviews-v1-874083890-f0qf0 2/2 Running 0 6m

reviews-v2-1343845940-b34q5 2/2 Running 0 6m

reviews-v3-1813607990-8ch52 2/2 Running 0 6m

To confirm that the Bookinfo application is running, send a request to it by a curl command from some pod, for example from ratings:

$ kubectl exec -it $(kubectl get pod -l app=ratings -o jsonpath='{.items[0].metadata.name}') -c ratings -- curl productpage:9080/productpage | grep -o "<title>.\*</title>"

<title>Simple Bookstore App</title>

## Determining the ingress IP and port

Now that the Bookinfo services are up and running, you need to make the application accessible from outside of your Kubernetes cluster, e.g., from a browser. An [Istio Gateway](https://istio.io/docs/concepts/traffic-management/" \l "gateways) is used for this purpose.

Define the ingress gateway for the application:

$ kubectl apply -f [samples/bookinfo/networking/bookinfo-gateway.yaml](https://raw.githubusercontent.com/istio/istio/release-1.1/samples/bookinfo/networking/bookinfo-gateway.yaml)

Confirm the gateway has been created:

$ kubectl get gateway

NAME AGE

bookinfo-gateway 32s

## Confirm the app is accessible from outside the cluster

To confirm that the Bookinfo application is accessible from outside the cluster, run the following curl command:

$ curl -s http://${GATEWAY\_URL}/productpage | grep -o "<title>.\*</title>"

<title>Simple Bookstore App</title>

You can also point your browser to http://$GATEWAY\_URL/productpage to view the Bookinfo web page. If you refresh the page several times, you should see different versions of reviews shown in productpage, presented in a round robin style (red stars, black stars, no stars), since we haven’t yet used Istio to control the version routing.

## Apply default destination rules

Run the following command to create default destination rules for the Bookinfo services:

If you **did not enable mutual TLS**, execute this command:

$ kubectl apply -f [samples/bookinfo/networking/destination-rule-all.yaml](https://raw.githubusercontent.com/istio/istio/release-1.1/samples/bookinfo/networking/destination-rule-all.yaml)

If you **did enable mutual TLS**, execute this command:

$ kubectl apply -f [samples/bookinfo/networking/destination-rule-all-mtls.yaml](https://raw.githubusercontent.com/istio/istio/release-1.1/samples/bookinfo/networking/destination-rule-all-mtls.yaml)

Wait a few seconds for the destination rules to propagate.

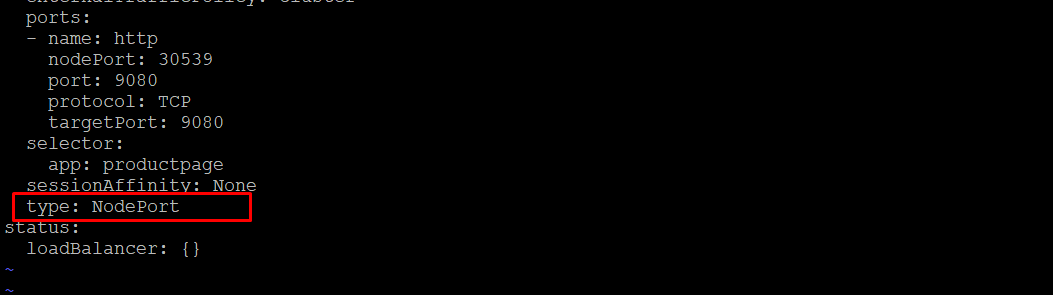
You can display the destination rules with the following command:

$ kubectl get destinationrules -o yaml

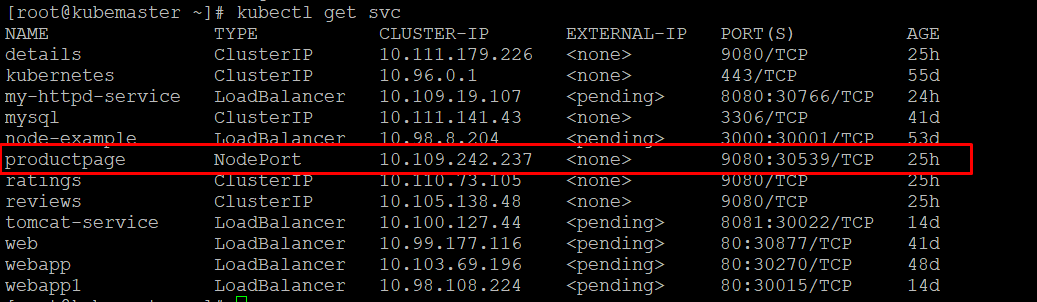
To add the Node port for the product page follow the instruction below:

$ kubectl edit svc productpage

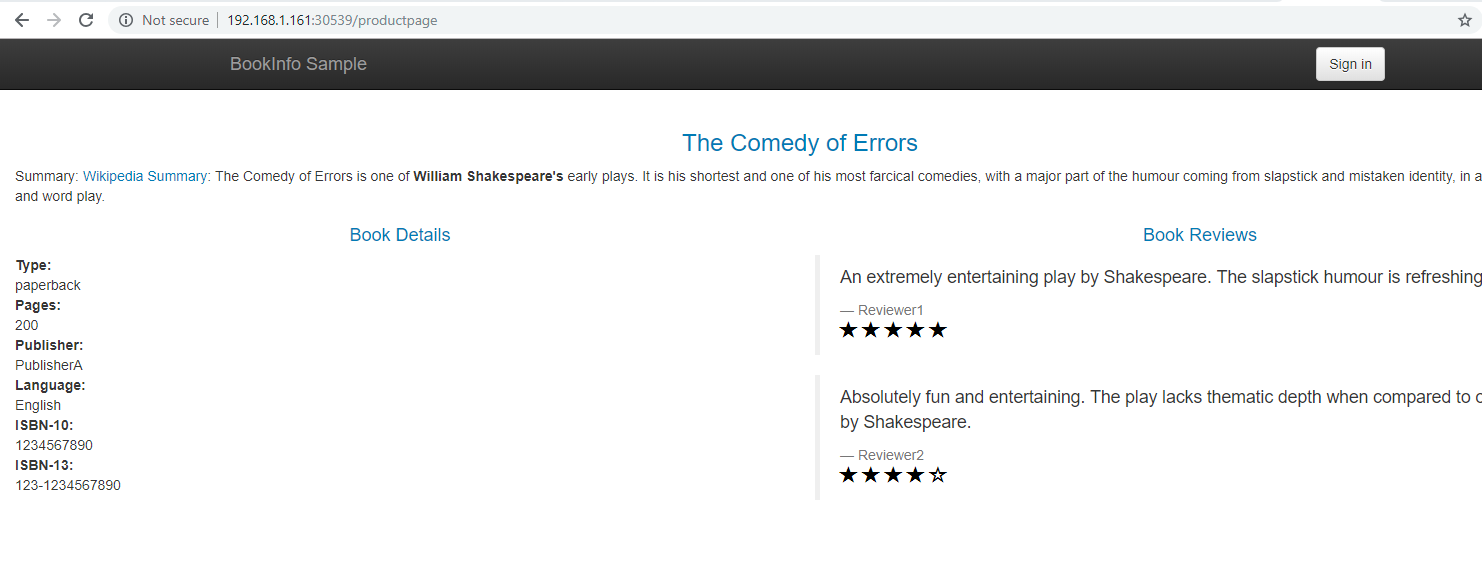
At last change ClusterIP to Nodeport



$ kubectl get svc



Open in the browser



# Control Routing

One of the main features of Istio is its traffic management. As a Microservice architectures scale, there is a requirement for more advanced service-to-service communication control.

## User Based Testing / Request Routing

One aspect of traffic management is controlling traffic routing based on the HTTP request, such as user agent strings, IP address or cookies.

The example below will send all traffic for the user "jason" to the reviews:v2, meaning they'll **only see the black stars**.

$ cat samples/bookinfo/networking/virtual-service-reviews-test-v2.yaml

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

name: reviews

spec:

hosts:

- reviews

http:

- match:

- headers:

end-user:

exact: jason

route:

- destination:

host: reviews

subset: v2

- route:

- destination:

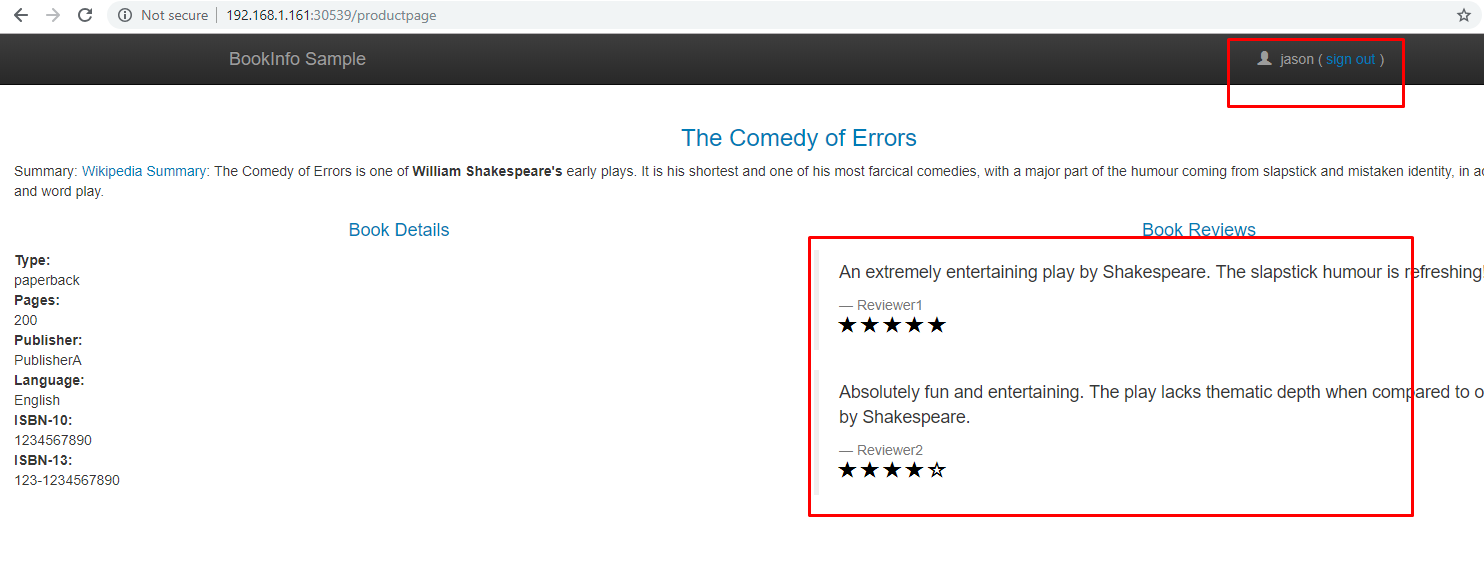
host: reviews

subset: v1

Similarly to deploying Kubernetes configuration, routing rules can be applied using istioctl.

$ kubectl apply -f samples/bookinfo/networking/virtual-service-reviews-test-v2.yaml

Visit the product page and sign in as Jason (reload the page only black star will display

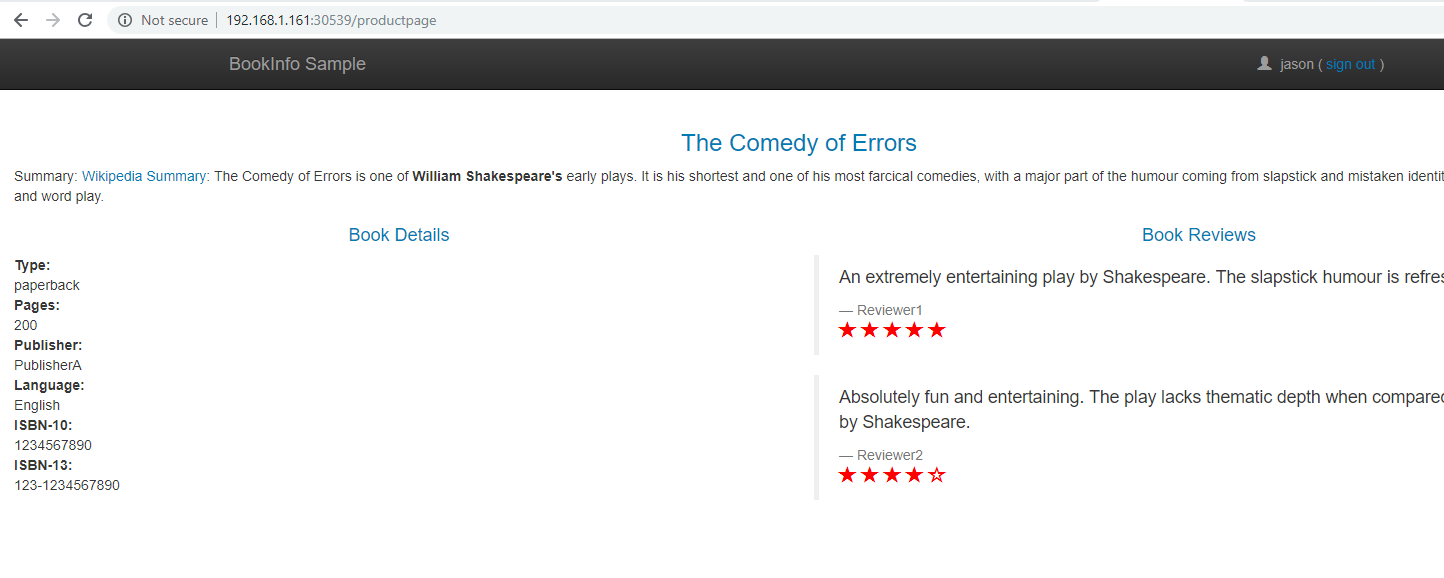


## Traffic Shaping for Canary Releases

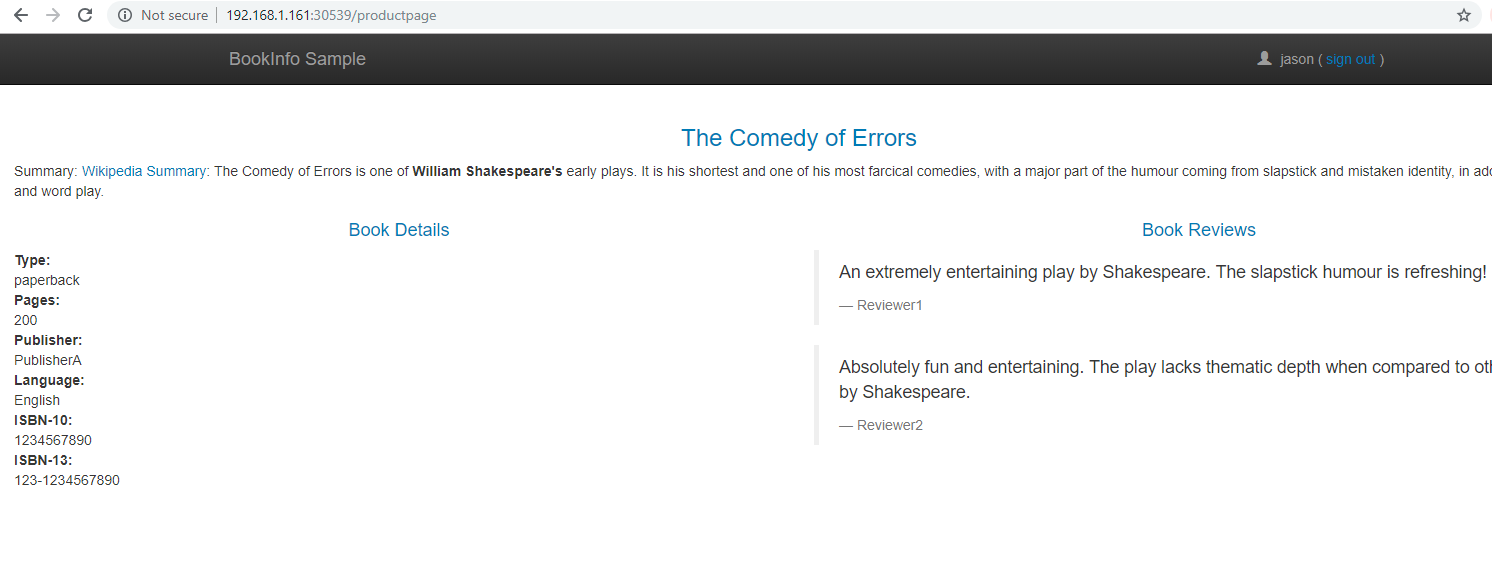
The ability to split traffic for testing and rolling out changes is important. This allows for A/B variation testing or deploying canary releases.

The rule below ensures that 50% of the traffic goes to reviews:v1 (no stars), or reviews:v3 (red stars).

kubectl apply -f samples/bookinfo/networking/virtual-service-reviews-50-v3.yaml



After reload the page



Logout of user Jason otherwise the above configuration will take priority

Note: The weighting is not round robin, multiple requests may go to the same service.

## New Releases

Given the above approach, if the canary release were successful then we'd want to move 100% of the traffic to reviews:v3.

$ cat samples/bookinfo/networking/virtual-service-reviews-v3.yaml

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

name: reviews

spec:

hosts:

- reviews

http:

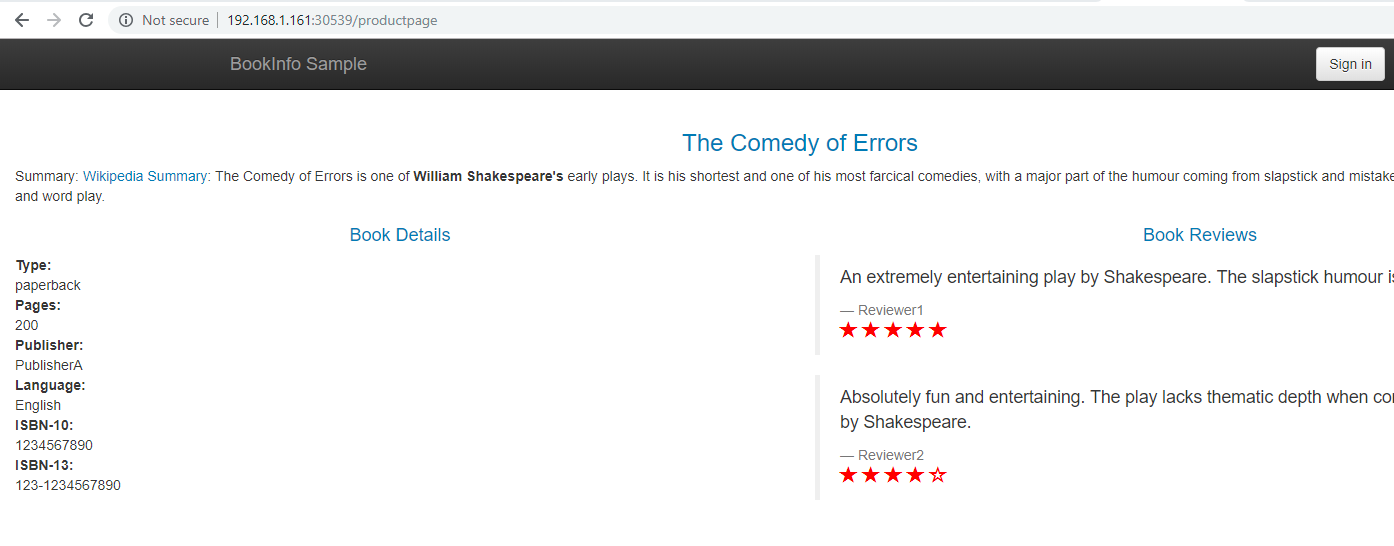
- route:

- destination:

host: reviews

subset: v3

kubectl apply -f samples/bookinfo/networking/virtual-service-reviews-v3.yaml



Note: whenever you deploy a services a sidecar get generated automatically inside the service

To check

$ Kubectl get pods <pod name> -o yaml

Its gives output with istio injection on code