Red Hat OpenShift Service Mesh 3.0

Installing

Installing OpenShift Service Mesh

Last Updated: 2025-09-16

Red Hat OpenShift Service Mesh 3.0 Installing

Installing OpenShift Service Mesh

Legal Notice

Abstract

This documentation provides information about installing OpenShift Service Mesh.

Table of Contents

CHAPTER 1. SUPPORTED PLATFORMS AND CONFIGURATIONS

Before you can install Red Hat OpenShift Service Mesh 3.0.4, you must subscribe to OpenShift Container Platform and install OpenShift Container Platform in a supported configuration. If you do not have a subscription on your Red Hat account, contact your sales representative for more information.

1.1. SUPPORTED PLATFORMS

Version {MaistraVersion} Service Mesh control planes are supported on the following platform versions:

- Red Hat OpenShift Container Platform version 4.14 or later
- Red Hat OpenShift Dedicated version 4
- Azure Red Hat OpenShift (ARO) version 4
- Red Hat OpenShift Service on AWS (ROSA)

The Red Hat OpenShift Service Mesh Operator supports multiple versions of **Istio**.

If you are installing Red Hat OpenShift Service Mesh on a restricted network, follow the instructions for your chosen OpenShift Container Platform infrastructure.

For additional information about Red Hat OpenShift Service Mesh lifecycle and supported platforms, refer to the Support Policy.

1.2. SUPPORTED CONFIGURATIONS FOR SERVICE MESH

Red Hat OpenShift Service Mesh supports the following configurations:

- This release of Red Hat OpenShift Service Mesh is supported on OpenShift Container Platform x86_64, IBM Z®, IBM Power®, and Advanced RISC Machine (ARM).
- Configurations where all Service Mesh components are contained within a single OpenShift Container Platform cluster.
- Configurations that do not integrate external services such as virtual machines.

NOTE

Red Hat OpenShift Service Mesh does not support the **EnvoyFilter** configuration except where explicitly documented.

1.3. SUPPORTED NETWORK CONFIGURATIONS

You can use the following OpenShift networking plugins for the Red Hat OpenShift Service Mesh:

- OpenShift-SDN.
- OVN-Kubernetes. See About the OVN-Kubernetes network plugin for more information.

- Third-Party Container Network Interface (CNI) plugins that have been certified on OpenShift Container Platform and passed Service Mesh conformance testing. See Certified OpenShift CNI Plug-ins for more information.

1.4. SUPPORTED CONFIGURATIONS FOR KIALI

- The Kiali console is supported on Google Chrome, Microsoft Edge, Mozilla Firefox, or Apple Safari browsers.
- The **openshift** authentication strategy is the only supported authentication configuration when Kiali is deployed with Red Hat OpenShift Service Mesh (OSSM). The **openshift** strategy controls access based on the user's role-based access control (RBAC) roles of the OpenShift Container Platform.

1.5. ADDITIONAL RESOURCES

- OpenShift Operator Life Cycles
- About OpenShift Container Platform installation
- Installing OpenShift Container Platform on AWS
- Installing OpenShift Container Platform on AWS with user-provisioned infrastructure
- Installing OpenShift Container Platform on bare metal
- Installing OpenShift Container Platform on vSphere
- Installing OpenShift Container Platform on IBM Z® and IBM® LinuxONE
- Installing OpenShift Container Platform on IBM Power®

CHAPTER 2. INSTALLING OPENSHIFT SERVICE MESH

Installing OpenShift Service Mesh consists of three main tasks: installing the OpenShift Operator, deploying Istio, and customizing the Istio configuration. Then, you can also choose to install the sample **bookinfo** application to push data through the mesh and explore mesh functionality.

WARNING

Before installing OpenShift Service Mesh 3, make sure you are not running OpenShift Service Mesh 3 and OpenShift Service Mesh 2 in the same cluster, because it causes conflicts unless configured correctly. To migrate from OpenShift Service Mesh 2, see Migrating from OpenShift Service Mesh 2.6 .

2.1. ABOUT DEPLOYING ISTIO USING THE RED HAT OPENSHIFT SERVICE MESH OPERATOR

To deploy Istio using the Red Hat OpenShift Service Mesh Operator, you must create an **Istio** resource. Then, the Operator creates an **IstioRevision** resource, which represents one revision of the Istio control plane. Based on the **IstioRevision** resource, the Operator deploys the Istio control plane, which includes the **istiod Deployment** resource and other resources.

The Red Hat OpenShift Service Mesh Operator may create additional instances of the **IstioRevision** resource, depending on the update strategy defined in the **Istio** resource.

2.1.1. About Istio control plane update strategies

The update strategy affects how the update process is performed. The **spec.updateStrategy** field in the **Istio** resource configuration determines how the OpenShift Service Mesh Operator updates the Istio control plane. When the Operator detects a change in the **spec.version** field or identifies a new minor release with a configured **vX.Y-latest** alias, it initiates an upgrade procedure. For each mesh, you select one of two strategies:

- **InPlace**
- **RevisionBased**

InPlace is the default strategy for updating OpenShift Service Mesh.

2.2. INSTALLING THE SERVICE MESH OPERATOR

WARNING

For clusters without OpenShift Service Mesh instances, install the Service Mesh Operator. OpenShift Service Mesh operates cluster-wide and needs a scope configuration to prevent conflicts between Istio control planes. For clusters with OpenShift Service Mesh 3 or later, see "Deploying multiple service meshes on a single cluster".

Prerequisites

- You have deployed a cluster on OpenShift Container Platform 4.14 or later.
- You are logged in to the OpenShift Container Platform web console as a user with the cluster-admin role.

Procedure

1. In the OpenShift Container Platform web console, navigate to the **Operators** → **OperatorHub** page.
2. Search for the Red Hat OpenShift Service Mesh 3 Operator.
3. Locate the Service Mesh Operator, and click to select it.
4. When the prompt that discusses the community operator opens, click **Continue**.
5. Click **Install**.
6. On the **Install Operator** page, perform the following steps:
   a. Select **All namespaces on the cluster (default)**as the **Installation Mode**. This mode installs the Operator in the default **openshift-operators** namespace, which enables the Operator to watch and be available to all namespaces in the cluster.
   b. Select **Automatic** as the **Approval Strategy**. This ensures that the Operator Lifecycle Manager (OLM) handles the future upgrades to the Operator automatically. If you select the **Manual** approval strategy, OLM creates an update request. As a cluster administrator, you must then manually approve the OLM update request to update the Operator to the new version.
   c. Select an **Update Channel**.
      - Choose the **stable** channel to install the latest stable version of the Red Hat OpenShift Service Mesh 3 Operator. It is the default channel for installing the Operator.
      - To install a specific version of the Red Hat OpenShift Service Mesh 3 Operator, choose the corresponding **stable-<version>** channel. For example, to install the Red Hat OpenShift Service Mesh Operator version 3.0.x, use the **stable-3.0** channel.
7. Click **Install** to install the Operator.

Verification

1. Click **Operators** → **Installed Operators** to verify that the Service Mesh Operator is installed. **Succeeded** should show in the **Status** column.

Additional resources

- Deploying multiple service meshes on a single cluster

2.2.1. About Service Mesh custom resource definitions

Installing the Red Hat OpenShift Service Mesh Operator also installs custom resource definitions (CRD) that administrators can use to configure Istio for Service Mesh installations. The Operator Lifecycle Manager (OLM) installs two categories of CRDs: Sail Operator CRDs and Istio CRDs.

Sail Operator CRDs define custom resources for installing and maintaining the Istio components required to operate a service mesh. These custom resources belong to the **sailoperator.io** API group and include the **Istio**, **IstioRevision**, **IstioCNI**, and **ZTunnel** resource kinds. For more information on how to configure these resources, see the **sailoperator.io** API reference documentation.

Istio CRDs are associated with mesh configuration and service management. These CRDs define custom resources in several **istio.io** API groups, such as **networking.istio.io** and **security.istio.io**. The CRDs also include various resource kinds, such as **AuthorizationPolicy**, **DestinationRule**, and **VirtualService**, that administrators use to configure a service mesh.

2.3. ABOUT ISTIO DEPLOYMENT

To deploy Istio, you must create two resources: **Istio** and **IstioCNI**. The **Istio** resource deploys and configures the Istio Control Plane. The **IstioCNI** resource deploys and configures the Istio Container Network Interface (CNI) plugin. You should create these resources in separate projects; therefore, you must create two projects as part of the Istio deployment process.

You can use the OpenShift web console or the OpenShift CLI (oc) to create a project or a resource in your cluster.

NOTE

In the OpenShift Container Platform, a project is essentially a Kubernetes namespace with additional annotations, such as the range of user IDs that can be used in the project. Typically, the OpenShift Container Platform web console uses the term project, and the CLI uses the term namespace, but the terms are essentially synonymous.

2.3.1. Creating the Istio project using the web console

The Service Mesh Operator deploys the Istio control plane to a project that you create. In this example, **istio-system** is the name of the project.

Prerequisties

- The Red Hat OpenShift Service Mesh Operator must be installed.
- You are logged in to the OpenShift Container Platform web console as cluster-admin.

Procedure

1. In the OpenShift Container Platform web console, click **Home** → **Projects**.
2. Click **Create Project**.
3. At the prompt, enter a name for the project in the **Name** field. For example, **istio-system**. The other fields provide supplementary information to the **Istio** resource definition and are optional.
4. Click **Create**. The Service Mesh Operator deploys Istio to the project you specified.

2.3.2. Creating the Istio resource using the web console

Create the Istio resource that will contain the YAML configuration file for your Istio deployment. The Red Hat OpenShift Service Mesh Operator uses information in the YAML file to create an instance of the Istio control plane.

Prerequisties

- The Service Mesh Operator must be installed.
- You are logged in to the OpenShift Container Platform web console as cluster-admin.

Procedure

1. In the OpenShift Container Platform web console, click **Operators** → **Installed Operators**.
2. Select **istio-system** in the **Project** drop-down menu.
3. Click the Service Mesh Operator.
4. Click **Istio**.
5. Click **Create Istio**.
6. Select the **istio-system** project from the **Namespace** drop-down menu.
7. Click **Create**. This action deploys the Istio control plane.
   When **State: Healthy** appears in the **Status** column, Istio is successfully deployed.

2.3.3. Creating the IstioCNI project using the web console

The Service Mesh Operator deploys the Istio CNI plugin to a project that you create. In this example, **istio-cni** is the name of the project.

Prerequisties

- The Red Hat OpenShift Service Mesh Operator must be installed.
- You are logged in to the OpenShift Container Platform web console as cluster-admin.

Procedure

1. In the OpenShift Container Platform web console, click **Home** → **Projects**.
2. Click **Create Project**.

3. At the prompt, you must enter a name for the project in the **Name** field. For example, **istio-cni**. The other fields provide supplementary information and are optional.

4. Click **Create**.

2.3.4. Creating the IstioCNI resource using the web console

Create an Istio Container Network Interface (CNI) resource, which contains the configuration file for the Istio CNI plugin. The Service Mesh Operator uses the configuration specified by this resource to deploy the CNI pod.

Prerequisties

- The Red Hat OpenShift Service Mesh Operator must be installed.

- You are logged in to the OpenShift Container Platform web console as cluster-admin.

Procedure

1. In the OpenShift Container Platform web console, click **Operators → Installed Operators**.

2. Select **istio-cni** in the **Project** drop-down menu.

3. Click the Service Mesh Operator.

4. Click **IstioCNI**.

5. Click **Create IstioCNI**.

6. Ensure that the name is **default**.

7. Click **Create**. This action deploys the Istio CNI plugin.
   When **State: Healthy** appears in the **Status** column, the Istio CNI plugin is successfully deployed.

2.4. SCOPING THE SERVICE MESH WITH DISCOVERY SELECTORS

Service Mesh includes workloads that meet the following criteria:

- The control plane has discovered the workload.

- The workload has an Envoy proxy sidecar injected.

By default, the control plane discovers workloads in all namespaces across the cluster, with the following results:

- Each proxy instance receives configuration for all namespaces, including workloads not enrolled in the mesh.

- Any workload with the appropriate pod or namespace injection label receives a proxy sidecar.

In shared clusters, you might want to limit the scope of Service Mesh to only certain namespaces. This approach is especially useful if multiple service meshes run in the same cluster.

2.4.1. About discovery selectors

With discovery selectors, the mesh administrator can control which namespaces the control plane can access. By using a Kubernetes label selector, the administrator sets the criteria for the namespaces visible to the control plane, excluding any namespaces that do not match the specified criteria.

NOTE

Istiod always opens a watch to OpenShift for all namespaces. However, discovery selectors ignore objects that are not selected very early in its processing, minimizing costs.

The **discoverySelectors** field accepts an array of Kubernetes selectors, which apply to labels on namespaces. You can configure each selector for different use cases:

- Custom label names and values. For example, configure all namespaces with the label **istio-discovery=enabled**.
- A list of namespace labels by using set-based selectors with OR logic. For instance, configure namespaces with **istio-discovery=enabled** OR **region=us-east1**.
- Inclusion and exclusion of namespaces. For example, configure namespaces with **istio-discovery=enabled** AND the label **app=helloworld**.

NOTE

Discovery selectors are not a security boundary. Istiod continues to have access to all namespaces even when you have configured the **discoverySelector** field.

Additional resources

- Label selectors (Kubernetes documentation)
- Resources that support set-based requirements (Kubernetes documentation)

2.4.2. Scoping a Service Mesh by using discovery selectors

If you know which namespaces to include in the Service Mesh, configure **discoverySelectors** during or after installation by adding the required selectors to the **meshConfig.discoverySelectors** section of the **Istio** resource. For example, configure Istio to discover only namespaces labeled **istio-discovery=enabled**.

Prerequisites

- The OpenShift Service Mesh operator is installed.
- An Istio CNI resource is created.

Procedure

1. Add a label to the namespace containing the Istio control plane, for example, the **istio-system** system namespace.

```
$ oc label namespace istio-system istio-discovery=enabled
```

2. Modify the **Istio** control plane resource to include a **discoverySelectors** section with the same label.

```
kind: Istio
apiVersion: sailoperator.io/v1
metadata:
  name: default
spec:
  namespace: istio-system
  values:
    meshConfig:
      discoverySelectors:
        - matchLabels:
            istio-discovery: enabled
```

3. Apply the Istio CR:

```
$ oc apply -f istio.yaml
```

4. Ensure that all namespaces that will contain workloads that are to be part of the Service Mesh have both the **discoverySelector** label and, if needed, the appropriate Istio injection label.

NOTE

Discovery selectors help restrict the scope of a single Service Mesh and are essential for limiting the control plane scope when you deploy multiple Istio control planes in a single cluster.

Next steps

- Deploying the Bookinfo application

2.5. ABOUT THE BOOKINFO APPLICATION

Installing the **bookinfo** example application consists of two main tasks: deploying the application and creating a gateway so the application is accessible outside the cluster.

You can use the **bookinfo** application to explore service mesh features. Using the **bookinfo** application, you can easily confirm that requests from a web browser pass through the mesh and reach the application.

The **bookinfo** application displays information about a book, similar to a single catalog entry of an online book store. The application displays a page that describes the book, lists book details (ISBN, number of pages, and other information), and book reviews.

The **bookinfo** application is exposed through the mesh, and the mesh configuration determines how the microservices comprising the application are used to serve requests. The review information comes from one of three services: **reviews-v1**, **reviews-v2** or **reviews-v3**. If you deploy the **bookinfo** application without defining the **reviews** virtual service, then the mesh uses a round robin rule to route requests to a service.

By deploying the **reviews** virtual service, you can specify a different behavior. For example, you can specify that if a user logs into the **bookinfo** application, then the mesh routes requests to the **reviews-v2** service, and the application displays reviews with black stars. If a user does not log into the **bookinfo** application, then the mesh routes requests to the **reviews-v3** service, and the application displays reviews with red stars.

For more information, see Bookinfo Application in the upstream Istio documentation.

2.5.1. Deploying the Bookinfo application

Prerequisites

- You have deployed a cluster on OpenShift Container Platform 4.15 or later.
- You are logged in to the OpenShift Container Platform web console as a user with the **cluster-admin** role.
- You have access to the OpenShift CLI (oc).
- You have installed the Red Hat OpenShift Service Mesh Operator, created the Istio resource, and the Operator has deployed Istio.
- You have created IstioCNI resource, and the Operator has deployed the necessary IstioCNI pods.

Procedure

1. In the OpenShift Container Platform web console, navigate to the **Home → Projects** page.
2. Click **Create Project**.
3. Enter **bookinfo** in the **Project name** field.
   The **Display name** and **Description** fields provide supplementary information and are not required.
4. Click **Create**.
5. Apply the Istio discovery selector and injection label to the **bookinfo** namespace by entering the following command:

   ```
   $ oc label namespace bookinfo istio-discovery=enabled istio-injection=enabled
   ```

   NOTE

   In this example, the name of the Istio resource is **default**. If the Istio resource name is different, you must set the **istio.io/rev** label to the name of the Istio resource instead of adding the **istio-injection=enabled** label.

6. Apply the **bookinfo** YAML file to deploy the **bookinfo** application by entering the following command:

   ```
   oc apply -f https://raw.githubusercontent.com/openshift-service-mesh/istio/release-
   1.24/samples/bookinfo/platform/kube/bookinfo.yaml -n bookinfo
   ```

Verification

1. Verify that the **bookinfo** service is available by running the following command:

   ```
   $ oc get services -n bookinfo
   ```

   Example output

   ```
   NAME          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
   details       ClusterIP   172.30.137.21    <none>        9080/TCP   44s
   productpage   ClusterIP   172.30.2.246     <none>        9080/TCP   43s
   ratings       ClusterIP   172.30.33.85     <none>        9080/TCP   44s
   reviews       ClusterIP   172.30.175.88    <none>        9080/TCP   44s
   ```

2. Verify that the **bookinfo** pods are available by running the following command:

   ```
   $ oc get pods -n bookinfo
   ```

   Example output

   ```
   NAME                             READY   STATUS    RESTARTS   AGE
   details-v1-698d88b-km2jg         2/2     Running   0          66s
   productpage-v1-675fc69cf-cvxv9   2/2     Running   0          65s
   ratings-v1-6484c4d9bb-tpx7d      2/2     Running   0          65s
   reviews-v1-5b5d6494f4-wsrwp      2/2     Running   0          65s
   reviews-v2-5b667bcbf8-4lsfd      2/2     Running   0          65s
   reviews-v3-5b9bd44f4-44hr6       2/2     Running   0          65s
   ```

   When the **Ready** columns displays **2/2**, the proxy sidecar was successfully injected. Confirm that **Running** appears in the **Status** column for each pod.

3. Verify that the **bookinfo** application is running by sending a request to the **bookinfo** page. Run the following command:

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

2.5.2. About accessing the Bookinfo application using a gateway

The Red Hat OpenShift Service Mesh Operator does not deploy gateways. Gateways are not part of the control plane. As a security best-practice, Ingress and Egress gateways should be deployed in a different namespace than the namespace that contains the control plane.

You can deploy gateways using either the Gateway API or the gateway injection method.

2.5.3. Accessing the Bookinfo application by using Istio gateway injection

Gateway injection uses the same mechanisms as Istio sidecar injection to create a gateway from a **Deployment** resource that is paired with a **Service** resource. The **Service** resource can be made accessible from outside an OpenShift Container Platform cluster.

Prerequisites

- You are logged in to the OpenShift Container Platform web console as **cluster-admin**.

- The Red Hat OpenShift Service Mesh Operator must be installed.
- The Istio resource must be deployed.

Procedure

1. Create the **istio-ingressgateway** deployment and service by running the following command:

```
$ oc apply -n bookinfo -f ingress-gateway.yaml
```

NOTE

This example uses a sample **ingress-gateway.yaml** file that is available in the Istio community repository.

2. Configure the **bookinfo** application to use the new gateway. Apply the gateway configuration by running the following command:

```
$ oc apply -f https://raw.githubusercontent.com/openshift-service-mesh/istio/release-
1.24/samples/bookinfo/networking/bookinfo-gateway.yaml -n bookinfo
```

NOTE

To configure gateway injection with the **bookinfo** application, this example uses a sample gateway configuration file that must be applied in the namespace where the application is installed.

3. Use a route to expose the gateway external to the cluster by running the following command:

```
$ oc expose service istio-ingressgateway -n bookinfo
```

4. Modify the YAML file to automatically scale the pod when ingress traffic increases.

Example configuration

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  labels:
    istio: ingressgateway
    release: istio
  name: ingressgatewayhpa
  namespace: bookinfo
spec:
  maxReplicas: 5 (1)
  metrics:
  - resource:
      name: cpu
      target:
        averageUtilization: 80
        type: Utilization
    type: Resource
  minReplicas: 2
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: istio-ingressgateway
```

1 This example sets the the maximum replicas to **5** and the minimum replicas to **2**. It also creates another replica when utilization reaches 80%.

5. Specify the minimum number of pods that must be running on the node.

Example configuration

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  labels:
    istio: ingressgateway
    release: istio
  name: ingressgatewaypdb
  namespace: bookinfo
spec:
  minAvailable: 1 (1)
  selector:
    matchLabels:
      istio: ingressgateway
```

1 This example ensures one replica is running if a pod gets restarted on a new node.

6. Obtain the gateway host name and the URL for the product page by running the following command:

```
$ HOST=$(oc get route istio-ingressgateway -n bookinfo -o jsonpath='{.spec.host}')
```

7. Verify that the **productpage** is accessible from a web browser by running the following command:

```
$ echo productpage URL: http://$HOST/productpage
```

2.5.4. Accessing the Bookinfo application by using Gateway API

The Kubernetes Gateway API deploys a gateway by creating a **Gateway** resource. In OpenShift Container Platform 4.15 and later, Red Hat OpenShift Service Mesh implements the Gateway API custom resource definitions (CRDs). However, in OpenShift Container Platform 4.18 and earlier, the CRDs are not installed by default. Hence, in OpenShift Container Platform 4.15 through 4.18, you must manually install the CRDs. Starting with OpenShift Container Platform 4.19, these CRDs are automatically installed and managed, and you can no longer create, update, or delete them.

For details about enabling Gateway API for Ingress in OpenShift Container Platform 4.19 and later, see "Configuring ingress cluster traffic" in the OpenShift Container Platform documentation.

NOTE

Red Hat provides support for using the Kubernetes Gateway API with Red Hat OpenShift Service Mesh. Red Hat does not provide support for the Kubernetes Gateway API custom resource definitions (CRDs). In this procedure, the use of community Gateway API CRDs is shown for demonstration purposes only.

Prerequisites

- You are logged in to the OpenShift Container Platform web console as **cluster-admin**.
- The Red Hat OpenShift Service Mesh Operator must be installed.
- The Istio resource must be deployed.

Procedure

1. Enable the Gateway API CRDs for OpenShift Container Platform 4.18 and earlier, by running the following command:

```
$ oc get crd gateways.gateway.networking.k8s.io &> /dev/null ||  { oc kustomize
"github.com/kubernetes-sigs/gateway-api/config/crd?ref=v1.0.0" | oc apply -f -; }
```

2. Create and configure a gateway by using the **Gateway** and **HTTPRoute** resources by running the following command:

```
$ oc apply -f https://raw.githubusercontent.com/openshift-service-mesh/istio/release-
1.24/samples/bookinfo/gateway-api/bookinfo-gateway.yaml -n bookinfo
```

NOTE

To configure a gateway with the **bookinfo** application by using the Gateway API, this example uses a sample gateway configuration file that must be applied in the namespace where the application is installed.

3. Ensure that the Gateway API service is ready, and has an address allocated by running the following command:

```
$ oc wait --for=condition=programmed gtw bookinfo-gateway -n bookinfo
```

4. Retrieve the host by running the following command:

```
$ export INGRESS_HOST=$(oc get gtw bookinfo-gateway -n bookinfo -o
jsonpath='{.status.addresses[0].value}')
```

5. Retrieve the port by running the following command:

```
$ export INGRESS_PORT=$(oc get gtw bookinfo-gateway -n bookinfo -o
jsonpath='{.spec.listeners[?(@.name=="http")].port}')
```

6. Retrieve the gateway URL by running the following command:

```
$ export GATEWAY_URL=$INGRESS_HOST:$INGRESS_PORT
```

7. Obtain the gateway host name and the URL of the product page by running the following command:

```
$ echo "http://${GATEWAY_URL}/productpage"
```

Verification

- Verify that the **productpage** is accessible from a web browser.

Additional resources

- Configuring ingress cluster traffic

2.6. CUSTOMIZING ISTIO CONFIGURATION

The **values** field of the **Istio** custom resource definition, which was created when the control plane was deployed, can be used to customize Istio configuration using Istio's **Helm** configuration values. When you create this resource using the OpenShift Container Platform web console, it is pre-populated with configuration settings to enable Istio to run on OpenShift.

Procedure

1. Click **Operators** → **Installed Operators**.
2. Click **Istio** in the **Provided APIs** column.
3. Click the **Istio** instance, named **default**, in the **Name** column.
4. Click **YAML** to view the **Istio** configuration and make modifications.

For a list of available configuration for the **values** field, refer to Istio's artifacthub chart documentation.

- Base parameters
- Istiod parameters
- Gateway parameters
- CNI parameters
- ZTunnel parameters

Additional resources

- Service Mesh 3.0 Operator community documentation

2.7. ABOUT ISTIO HIGH AVAILABILITY

Running the Istio control plane in High Availability (HA) mode prevents single points of failure, and ensures continuous mesh operation even if an **istiod** pod fails. By using HA, if one **istiod** pod becomes unavailable, another one continues to manage and configure the Istio data plane, preventing service outages or disruptions. HA provides scalability by distributing the control plane workload, enables graceful upgrades, supports disaster recovery operations, and protects against zone-wide mesh outages.

There are two ways for a system administrator to configure HA for the Istio deployment:

- Defining a static replica count: This approach involves setting a fixed number of **istiod** pods, providing a consistent level of redundancy.
- Using autoscaling: This approach dynamically adjusts the number of **istiod** pods based on resource utilization or custom metrics, providing more efficient resource consumption for fluctuating workloads.

2.7.1. Configuring Istio HA by using autoscaling

Configure the Istio control plane in High Availability (HA) mode to prevent a single point of failure, and ensure continuous mesh operation even if one of the **istiod** pods fails. Autoscaling defines the minimum and maximum number of Istio control plane pods that can operate. OpenShift Container Platform uses these values to scale the number of control planes in operation based on resource utilization, such as CPU or memory, to efficiently respond to the varying number of workloads and overall traffic patterns within the mesh.

Prerequisites

- You are logged in to the OpenShift Container Platform web console as a user with the **cluster-admin** role.
- You have installed the Red Hat OpenShift Service Mesh Operator.
- You have deployed the Istio resource.

Procedure

1. In the OpenShift Container Platform web console, click **Installed Operators**.
2. Click Red Hat OpenShift Service Mesh 3 Operator.
3. Click **Istio**.
4. Click the name of the Istio installation. For example, **default**.
5. Click **YAML**.
6. Modify the **Istio** custom resource (CR) similar to the following example:

   Example configuration

```
apiVersion: sailoperator.io/v1
kind: Istio
metadata:
  name: default
spec:
  namespace: istio-system
  values:
    pilot:
```

```
    autoscaleMin: 2 (1)
    autoscaleMax: 5 (2)
    cpu:
      targetAverageUtilization: 80 (3)
    memory:
      targetAverageUtilization: 80 (4)
```

1. Specifies the minimum number of Istio control plane replicas that always run.
2. Specifies the maximum number of Istio control plane replicas, allowing for scaling based on load. To support HA, there must be at least two replicas.
3. Specifies the target CPU utilization for autoscaling to 80%. If the average CPU usage exceeds this threshold, the Horizontal Pod Autoscaler (HPA) automatically increases the number of replicas.
4. Specifies the target memory utilization for autoscaling to 80%. If the average memory usage exceeds this threshold, the HPA automatically increases the number of replicas.

Verification

- Verify the status of the Istio control pods by running the following command:

  ```
  $ oc get pods -n istio-system -l app=istiod
  ```

  Example output

  ```
  NAME                     READY   STATUS    RESTARTS   AGE
  istiod-7c7b6564c9-nwhsg   1/1     Running   0          70s
  istiod-7c7b6564c9-xkmsl   1/1     Running   0          85s
  ```

  Two **istiod** pods are running. Two pods, the minimum requirement for an HA Istio control plane, indicates that a basic HA setup is in place.

2.7.1.1. API settings for Service Mesh HA autoscaling mode

Use the following **istio** custom resource definition (CRD) parameters when you configure a service mesh for High Availability (HA) by using autoscaling.

Table 2.1. HA API parameters

Parameter	Description
autoScaleMin	Defines the minimum number of **istiod** pods for an istio deployment. Each pod contains one instance of the Istio control plane. OpenShift only uses this parameter when the Horizontal Pod Autoscaler (HPA) is enabled for the Istio deployment. This is the default behavior.

Parameter	Description
autoScaleMax	Defines the maximum number of **istiod** pods for an Istio deployment. Each pod contains one instance of the Istio control plane. For OpenShift to automatically scale the number of **istiod** pods based on load, you must set this parameter to a value that is greater than the value that you defined for the **autoScaleMin** parameter. You must also configure metrics for autoscaling to work properly. If no metrics are configured, the autoscaler does not scale up or down. OpenShift only uses this parameter when Horizontal Pod Autoscaler (HPA) is enabled for the Istio deployment. This is the default behavior.
cpu.targetAverageUtilization	Defines the target CPU utilization for the **istiod** pod. If the average CPU usage exceeds the threshold that this parameter defines, the HPA automatically increases the number of replica pods.
memory.targetAverageUtilization	Defines the target memory utilization for the **istiod** pod. If the average memory usage exceeds the threshold that this parameter defines, the HPA automatically increases the number of replica pods.
behavior	You can use the **behavior** field to define additional policies that OpenShift uses to scale Istio resources up or down. For more information, see Configurable Scaling Behavior.

Additional resources

- Horizontal Pod Autoscaling(Kubernetes documentation)

2.7.2. Configuring Istio HA by using replica count

Configure the Istio control plane in High Availability (HA) mode to prevent a single point of failure, and ensure continuous mesh operation even if one of the **istiod** pods fails. The replica count defines a fixed number of Istio control plane pods that can operate. Use replica count for mesh environments where the control plane workload is relatively stable or predictable, or when you prefer to manually scale the **istiod** pod.

Prerequisites

- You are logged in to the OpenShift Container Platform web console as a user with the **cluster-admin** role.

- You have installed the Red Hat OpenShift Service Mesh Operator.
- You have deployed the Istio resource.

Procedure

1. Obtain the name of the **Istio** resource by running the following command:

   ```
   $ oc get istio -n istio-sytem
   ```

   Example output

   ```
   NAME     REVISIONS   READY   IN USE   ACTIVE REVISION   STATUS    VERSION
   AGE
   default  1           1       0        default           Healthy   v1.24.6   24m
   ```

   The name of the **Istio** resource is **default**.

2. Update the **Istio** custom resource (CR) by adding the **autoscaleEnabled** and **replicaCount** parameters by running the following command:

   ```
   $ oc patch istio default -n istio-system --type merge -p '
   spec:
     values:
       pilot:
         autoscaleEnabled: false (1)
         replicaCount: 2 (2)
   '
   ```

   (1) Specifies a setting that disables autoscaling and ensures that the number of replicas remains fixed.

   (2) Specifies the number of Istio control plane replicas. To support HA, there must be at least two replicas.

Verification

1. Verify the status of the Istio control pods by running the following command:

   ```
   $ oc get pods -n istio-system -l app=istiod
   ```

   Example output

   ```
   NAME                     READY   STATUS    RESTARTS   AGE
   istiod-7c7b6564c9-nwhsg   1/1     Running   0          70s
   istiod-7c7b6564c9-xkmsl   1/1     Running   0          85s
   ```

   Two **istiod** pods are running, which is the minimum requirement for an HA Istio control plane and indicates that a basic HA setup is in place.

CHAPTER 3. SIDECAR INJECTION

To use Istio's capabilities within a service mesh, each pod needs a sidecar proxy, configured and managed by the Istio control plane.

3.1. ABOUT SIDECAR INJECTION

Sidecar injection is enabled using labels at the namespace or pod level. These labels also indicate the specific control plane managing the proxy. When you apply a valid injection label to the pod template defined in a deployment, any new pods created by that deployment automatically receive a sidecar. Similarly, applying a pod injection label at the namespace level ensures any new pods in that namespace include a sidecar.

NOTE

Injection happens at pod creation through an admission controller, so changes appear on individual pods rather than the deployment resources. To confirm sidecar injection, check the pod details directly using **oc describe**, where you can see the injected Istio proxy container.

3.2. IDENTIFYING THE REVISION NAME

The label required to enable sidecar injection is determined by the specific control plane instance, known as a revision. Each revision is managed by an **IstioRevision** resource, which is automatically created and managed by the **Istio** resource, so manual creation or modification of **IstioRevision** resources is generally unnecessary.

The naming of an **IstioRevision** depends on the **spec.updateStrategy.type** setting in the **Istio** resource. If set to **InPlace**, the revision shares the **Istio** resource name. If set to **RevisionBased**, the revision name follows the format **<Istio resource name>-v<version>**. Typically, each **Istio** resource corresponds to a single **IstioRevision**. However, during a revision-based upgrade, multiple **IstioRevision** resources may exist, each representing a distinct control plane instance.

To see available revision names, use the following command:

```
$ oc get istiorevisions
```

You should see output similar to the following example:

Example output

```
NAME             READY   STATUS    IN USE   VERSION   AGE
my-mesh-v1-23-0   True    Healthy   False    v1.23.0   114s
```

3.2.1. Enabling sidecar injection with default revision

When the service mesh's **IstioRevision** name is **default**, it's possible to use the following labels on a namespace or a pod to enable sidecar injection:

Resource	Label	Enabled value	Disabled value
Namespace	**istio-injection**	**enabled**	**disabled**
Pod	**sidecar.istio.io/inject**	**true**	**false**

NOTE

You can also enable injection by setting the **istio.io/rev: default** label in the namespace or pod.

3.2.2. Enabling sidecar injection with other revisions

When the **IstioRevision** name is not **default**, use the specific **IstioRevision** name with the **istio.io/rev** label to map the pod to the desired control plane and enable sidecar injection. To enable injection, set the **istio.io/rev: default** label in either the namespace or the pod, as adding it to both is not required.

For example, with the revision shown above, the following labels would enable sidecar injection:

Resource	Enabled label	Disabled label
Namespace	**istio.io/rev=my-mesh-v1-23-0**	**istio-injection=disabled**
Pod	**istio.io/rev=my-mesh-v1-23-0**	**sidecar.istio.io/inject="false"**

NOTE

When both **istio-injection** and **istio.io/rev** labels are applied, the **istio-injection** label takes precedence and treats the namespace as part of the default revision.

3.3. ENABLING SIDECAR INJECTION

To demonstrate different approaches for configuring sidecar injection, the following procedures use the Bookinfo application.

Prerequisites

- You have installed the Red Hat OpenShift Service Mesh Operator, created an **Istio** resource, and the Operator has deployed Istio.
- You have created the **IstioCNI** resource, and the Operator has deployed the necessary **IstioCNI** pods.
- You have created the namespaces that are to be part of the mesh, and they are discoverable by the Istio control plane.
- Optional: You have deployed the workloads to be included in the mesh. In the following examples, the Bookinfo has been deployed to the **bookinfo** namespace, but sidecar injection (step 5) has not been configured. For more information, see "Deploying the Bookinfo application".

3.3.1. Enabling sidecar injection with namespace labels

In this example, all workloads within a namespace receive a sidecar proxy injection, making it the best approach when the majority of workloads in the namespace should be included in the mesh.

Procedure

1. Verify the revision name of the Istio control plane using the following command:

   ```
   $ oc get istiorevisions
   ```

   You should see output similar to the following example:

   Example output

   ```
   NAME     TYPE    READY   STATUS    IN USE   VERSION   AGE
   default   Local   True    Healthy   False    v1.23.0   4m57s
   ```

   Since the revision name is default, you can use the default injection labels without referencing the exact revision name.

2. Verify that workloads already running in the desired namespace show **1/1** containers as **READY** by using the following command. This confirms that the pods are running without sidecars.

   ```
   $ oc get pods -n bookinfo
   ```

   You should see output similar to the following example:

   Example output

   ```
   NAME                          READY   STATUS    RESTARTS   AGE
   details-v1-65cfcf56f9-gm6v7      1/1     Running   0          4m55s
   productpage-v1-d5789fdfb-8x6bk   1/1     Running   0          4m53s
   ratings-v1-7c9bd4b87f-6v7hg      1/1     Running   0          4m55s
   reviews-v1-6584ddcf65-6wqtw      1/1     Running   0          4m54s
   reviews-v2-6f85cb9b7c-w9l8s      1/1     Running   0          4m54s
   reviews-v3-6f5b775685-mg5n6      1/1     Running   0          4m54s
   ```

3. To apply the injection label to the **bookinfo** namespace, run the following command at the CLI:

   ```
   $ oc label namespace bookinfo istio-injection=enabled
   namespace/bookinfo labeled
   ```

4. To ensure sidecar injection is applied, redeploy the existing workloads in the **bookinfo** namespace. Use the following command to perform a rolling update of all workloads:

   ```
   $ oc -n bookinfo rollout restart deployments
   ```

Verification

1. Verify the rollout by checking that the new pods display **2/2** containers as **READY**, confirming successful sidecar injection by running the following command:

```
$ oc get pods -n bookinfo
```

You should see output similar to the following example:

Example output

```
NAME                        READY   STATUS    RESTARTS   AGE
details-v1-7745f84ff-bpf8f        2/2     Running   0          55s
productpage-v1-54f48db985-gd5q9   2/2     Running   0          55s
ratings-v1-5d645c985f-xsw7p       2/2     Running   0          55s
reviews-v1-bd5f54b8c-zns4v        2/2     Running   0          55s
reviews-v2-5d7b9dbf97-wbpjr       2/2     Running   0          55s
reviews-v3-5fccc48c8c-bjktn       2/2     Running   0          55s
```

3.3.2. Exclude a workload from the mesh

You can exclude specific workloads from sidecar injection within a namespace where injection is enabled for all workloads.

NOTE

This example is for demonstration purposes only. The bookinfo application requires all workloads to be part of the mesh for proper functionality.

Procedure

1. Open the application's **Deployment** resource in an editor. In this case, exclude the **ratings-v1** service.

2. Modify the **spec.template.metadata.labels** section of your **Deployment** resource to include the label **sidecar.istio.io/inject: false** to disable sidecar injection.

```
kind: Deployment
apiVersion: apps/v1
metadata:
name: ratings-v1
namespace: bookinfo
labels:
  app: ratings
  version: v1
spec:
  template:
    metadata:
      labels:
        sidecar.istio.io/inject: 'false'
```

NOTE

Adding the label to the top-level **labels** section of the **Deployment** does not affect sidecar injection.

Updating the deployment triggers a rollout, creating a new ReplicaSet with updated pod(s).

Verification

1. Verify that the updated pod(s) do not contain a sidecar container and show **1/1** containers as **Running** by running the following command:

```
$ oc get pods -n bookinfo
```

You should see output similar to the following example:

Example output

```
NAME                             READY   STATUS    RESTARTS   AGE
details-v1-6bc7b69776-7f6wz       2/2     Running   0          29m
productpage-v1-54f48db985-gd5q9   2/2     Running   0          29m
ratings-v1-5d645c985f-xsw7p       1/1     Running   0          7s
reviews-v1-bd5f54b8c-zns4v        2/2     Running   0          29m
reviews-v2-5d7b9dbf97-wbpjr       2/2     Running   0          29m
reviews-v3-5fccc48c8c-bjktn       2/2     Running   0          29m
```

3.3.3. Enabling sidecar injection with pod labels

This approach allows you to include individual workloads for sidecar injection instead of applying it to all workloads within a namespace, making it ideal for scenarios where only a few workloads need to be part of a service mesh. This example also demonstrates the use of a revision label for sidecar injection, where the **Istio** resource is created with the name **my-mesh**. A unique **Istio** resource name is required when multiple Istio control planes are present in the same cluster or during a revision-based control plane upgrade.

Procedure

1. Verify the revision name of the Istio control plane by running the following command:

```
$ oc get istiorevisions
```

You should see output similar to the following example:

Example output

```
NAME      TYPE    READY   STATUS    IN USE   VERSION   AGE
my-mesh   Local   True    Healthy   False    v1.23.0   47s
```

Since the revision name is **my-mesh**, use the revision label **istio.io/rev=my-mesh** to enable sidecar injection.

2. Verify that workloads already running show **1/1** containers as **READY**, indicating that the pods are running without sidecars by running the following command:

```
$ oc get pods -n bookinfo
```

You should see output similar to the following example:

Example output

```
NAME                           READY   STATUS    RESTARTS   AGE
details-v1-65cfcf56f9-gm6v7      1/1     Running   0          4m55s
productpage-v1-d5789fdfb-8x6bk   1/1     Running   0          4m53s
ratings-v1-7c9bd4b87f-6v7hg      1/1     Running   0          4m55s
reviews-v1-6584ddcf65-6wqtw      1/1     Running   0          4m54s
reviews-v2-6f85cb9b7c-w9l8s      1/1     Running   0          4m54s
reviews-v3-6f5b775685-mg5n6      1/1     Running   0          4m54s
```

3. Open the application's **Deployment** resource in an editor. In this case, update the **ratings-v1** service.

4. Update the **spec.template.metadata.labels** section of your **Deployment** to include the appropriate pod injection or revision label. In this case, **istio.io/rev: my-mesh**:

```
kind: Deployment
apiVersion: apps/v1
metadata:
name: ratings-v1
namespace: bookinfo
labels:
  app: ratings
  version: v1
spec:
  template:
    metadata:
      labels:
        istio.io/rev: my-mesh
```

NOTE

Adding the label to the top-level **labels** section of the **Deployment** resource does not impact sidecar injection.

Updating the deployment triggers a rollout, creating a new ReplicaSet with the updated pod(s).

Verification

1. Verify that only the ratings-v1 pod now shows **2/2** containers **READY**, indicating that the sidecar has been successfully injected by running the following command:

```
$ oc get pods -n bookinfo
```

You should see output similar to the following example:

Example output

```
NAME                           READY   STATUS    RESTARTS   AGE
details-v1-559cd49f6c-b89hw      1/1     Running   0          42m
productpage-v1-5f48cdcb85-8ppz5   1/1     Running   0          42m
ratings-v1-848bf79888-krdch      2/2     Running   0          9s
reviews-v1-6b7444ffbd-7m5wp      1/1     Running   0          42m
reviews-v2-67876d7b7-9nmw5       1/1     Running   0          42m
reviews-v3-84b55b667c-x5t8s      1/1     Running   0          42m
```

2. Repeat for other workloads that you wish to include in the mesh.

3.4. ENABLING SIDECAR INJECTION WITH NAMESPACE LABELS AND AN ISTIOREVISIONTAG RESOURCE

To use the **istio-injection=enabled** label when your revision name is not **default**, you must create an **IstioRevisionTag** resource with the name **default** that references your **Istio** resource.

Prerequisites

- You have installed the Red Hat OpenShift Service Mesh Operator, created an **Istio** resource, and the Operator has deployed Istio.
- You have created the **IstioCNI** resource, and the Operator has deployed the necessary **IstioCNI** pods.
- You have created the namespaces that are to be part of the mesh, and they are discoverable by the Istio control plane.
- Optional: You have deployed the workloads to be included in the mesh. In the following examples, the Bookinfo has been deployed to the **bookinfo** namespace, but sidecar injection (step 5 in "Deploying the Bookinfo application" procedure) has not been configured. For more information, see "Deploying the Bookinfo application".

Procedure

1. Find the name of your **Istio** resource by running the following command:

   ```
   $ oc get istio
   ```

   Example output

   ```
   NAME     REVISIONS   READY   IN USE   ACTIVE REVISION   STATUS    VERSION
   AGE
   default   1          1       1        default-v1-24-3   Healthy   v1.24.3   11s
   ```

   In this example, the **Istio** resource has the name **default**, but the underlying revision is called **default-v1-24-3**.

2. Create the **IstioRevisionTag** resource in a YAML file:

   Example IstioRevisionTag resource YAML file

   ```
   apiVersion: sailoperator.io/v1
   kind: IstioRevisionTag
   metadata:
     name: default
   spec:
     targetRef:
       kind: Istio
       name: default
   ```

3. Apply the **IstioRevisionTag** resource by running the following command:

```
$ oc apply -f istioRevisionTag.yaml
```

4. Verify that the **IstioRevisionTag** resource has been created successfully by running the following command:

```
$ oc get istiorevisiontags.sailoperator.io
```

Example output

```
NAME      STATUS    IN USE   REVISION          AGE
default   Healthy   True     default-v1-24-3   4m23s
```

In this example, the new tag is referencing your active revision, **default-v1-24-3**. Now you can use the **istio-injection=enabled** label as if your revision was called **default**.

5. Confirm that the pods are running without sidecars by running the following command. Any workloads that are already running in the desired namespace should show **1/1** containers in the **READY** column.

```
$ oc get pods -n bookinfo
```

Example output

```
NAME                             READY   STATUS    RESTARTS   AGE
details-v1-65cfcf56f9-gm6v7      1/1     Running   0          4m55s
productpage-v1-d5789fdfb-8x6bk   1/1     Running   0          4m53s
ratings-v1-7c9bd4b87f-6v7hg      1/1     Running   0          4m55s
reviews-v1-6584ddcf65-6wqtw      1/1     Running   0          4m54s
reviews-v2-6f85cb9b7c-w9l8s      1/1     Running   0          4m54s
reviews-v3-6f5b775685-mg5n6      1/1     Running   0          4m54s
```

6. Apply the injection label to the **bookinfo** namespace by running the following command:

```
$ oc label namespace bookinfo istio-injection=enabled \
namespace/bookinfo labeled
```

7. To ensure sidecar injection is applied, redeploy the workloads in the **bookinfo** namespace by running the following command:

```
$ oc -n bookinfo rollout restart deployments
```

Verification

1. Verify the rollout by running the following command and confirming that the new pods display **2/2** containers in the **READY** column:

```
$ oc get pods -n bookinfo
```

Example output

```
NAME                             READY   STATUS    RESTARTS   AGE
```

```
details-v1-7745f84ff-bpf8f       2/2     Running   0          55s
productpage-v1-54f48db985-gd5q9   2/2     Running   0          55s
ratings-v1-5d645c985f-xsw7p      2/2     Running   0          55s
reviews-v1-bd5f54b8c-zns4v       2/2     Running   0          55s
reviews-v2-5d7b9dbf97-wbpjr      2/2     Running   0          55s
reviews-v3-5fccc48c8c-bjktn      2/2     Running   0          55s
```

3.5. ADDITIONAL RESOURCES

- About admission controllers (Kubernetes documentation)
- Istio sidecar injection problems (Istio documentation)
- Deploying the Bookinfo application
- Scoping the mesh with discovery selectors
- IstioRevisionTag resource

CHAPTER 4. OPENSHIFT SERVICE MESH AND CERT-MANAGER

The cert-manager tool is a solution for X.509 certificate management on Kubernetes. It delivers a unified API to integrate applications with private or public key infrastructure (PKI), such as Vault, Google Cloud Certificate Authority Service, Let's Encrypt, and other providers.

IMPORTANT

The cert-manager tool must be installed before you create and install your **Istio** resource.

The cert-manager tool ensures the certificates are valid and up-to-date by attempting to renew certificates at a configured time before they expire.

4.1. ABOUT INTEGRATING SERVICE MESH WITH CERT-MANAGER AND ISTIO-CSR

The cert-manager tool provides integration with Istio through an external agent called **istio-csr**. The **istio-csr** agent handles certificate signing requests (CSR) from Istio proxies and the **controlplane** in the following ways:

1. Verifying the identity of the workload.
2. Creating a CSR through cert-manager for the workload.

The cert-manager tool then creates a CSR to the configured CA Issuer, which signs the certificate.

NOTE

Red Hat provides support for integrating with **istio-csr** and cert-manager. Red Hat does not provide direct support for the **istio-csr** or the community cert-manager components. The use of community cert-manager shown here is for demonstration purposes only.

Prerequisites

- One of these versions of cert-manager:
  - Red Hat cert-manager Operator 1.10 or later
  - community cert-manager Operator 1.11 or later
  - cert-manager 1.11 or later
- Red Hat OpenShift Service Mesh 3.0 or later
- An **IstioCNI** instance is running in the cluster
- Istio CLI (**istioctl**) tool is installed
- **jq** is installed
- Helm is installed

4.2. INSTALLING CERT-MANAGER

You can integrate cert-manager with OpenShift Service Mesh by deploying **istio-csr** and then creating an **Istio** resource that uses the **istio-csr** agent to process workload and control plane certificate signing requests. This example creates a self-signed **Issuer**, but any other **Issuer** can be used instead.

IMPORTANT

You must install cert-manager before installing your **Istio** resource.

Procedure

1. Create the **istio-system** namespace by running the following command:

```
$ oc create namespace istio-system
```

2. Create the root issuer by creating an **Issuer** object in a YAML file.

   a. Create an **Issuer** object similar to the following example:

Example issuer.yaml file

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned
  namespace: istio-system
spec:
  selfSigned: {}
---
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
    name: istio-ca
    namespace: istio-system
spec:
  isCA: true
  duration: 87600h # 10 years
  secretName: istio-ca
  commonName: istio-ca
  privateKey:
    algorithm: ECDSA
    size: 256
  subject:
    organizations:
      - cluster.local
      - cert-manager
  issuerRef:
    name: selfsigned
    kind: Issuer
    group: cert-manager.io
---
apiVersion: cert-manager.io/v1
kind: Issuer
```

```
metadata:
  name: istio-ca
  namespace: istio-system
spec:
  ca:
    secretName: istio-ca
---
```

b. Create the objects by running the following command:

```
$ oc apply -f issuer.yaml
```

c. Wait for the **istio-ca** certificate to contain the "Ready" status condition by running the following command:

```
$ oc wait --for=condition=Ready certificates/istio-ca -n istio-system
```

3. Copy the **istio-ca** certificate to the **cert-manager** namespace so it can be used by istio-csr:

    a. Copy the secret to a local file by running the following command:

    ```
    $ oc get -n istio-system secret istio-ca -o jsonpath='{.data.tls\.crt}' | base64 -d > ca.pem
    ```

    b. Create a secret from the local certificate file in the **cert-manager** namespace by running the following command:

    ```
    $ oc create secret generic -n cert-manager istio-root-ca --from-file=ca.pem=ca.pem
    ```

Next steps

To install **istio-csr**, you must follow the **istio-csr** installation instructions for the type of update strategy you want. By default, **spec.updateStrategy** is set to **InPlace** when you create and install your **Istio** resource. You create and install your **Istio** resource after you install **istio-csr**.

- Installing the istio-csr agent by using the in place update strategy
- Installing the istio-csr agent by using the revision based update strategy

4.2.1. Installing the istio-csr agent by using the in place update strategy

Istio resources use the in place update strategy by default. Follow this procedure if you plan to leave **spec.updateStrategy** as **InPlace** when you create and install your **Istio** resource.

Procedure

1. Add the Jetstack charts repository to your local Helm repository by running the following command:

    ```
    $ helm repo add jetstack https://charts.jetstack.io --force-update
    ```

2. Install the **istio-csr** chart by running the following command:

    ```
    $ helm upgrade cert-manager-istio-csr jetstack/cert-manager-istio-csr \
    ```

```
  --install \
  --namespace cert-manager \
  --wait \
  --set "app.tls.rootCAFile=/var/run/secrets/istio-csr/ca.pem" \
  --set "volumeMounts[0].name=root-ca" \
  --set "volumeMounts[0].mountPath=/var/run/secrets/istio-csr" \
  --set "volumes[0].name=root-ca" \
  --set "volumes[0].secret.secretName=istio-root-ca" \
  --set "app.istio.namespace=istio-system"
```

Next steps

- Installing your Istio resource

4.2.2. Installing the istio-csr agent by using the revision based update strategy

Istio resources use the in place update strategy by default. Follow this procedure if you plan to change **spec.updateStrategy** to **RevisionBased** when you create and install your **Istio** resource.

Procedure

1. Specify all the Istio revisions to your **istio-csr** deployment. See "istio-csr deployment".

2. Add the Jetstack charts to your local Helm repository by running the following command:

```
$ helm repo add jetstack https://charts.jetstack.io --force-update
```

3. Install the **istio-csr** chart with your revision name by running the following command:

```
$ helm upgrade cert-manager-istio-csr jetstack/cert-manager-istio-csr \
  --install \
  --namespace cert-manager \
  --wait \
  --set "app.tls.rootCAFile=/var/run/secrets/istio-csr/ca.pem" \
  --set "volumeMounts[0].name=root-ca" \
  --set "volumeMounts[0].mountPath=/var/run/secrets/istio-csr" \
  --set "volumes[0].name=root-ca" \
  --set "volumes[0].secret.secretName=istio-root-ca" \
  --set "app.istio.namespace=istio-system" \
  --set "app.istio.revisions={default-v1-24-3}"
```

NOTE

Revision names use the following format, **<istio-name>-v<major_version>-<minor_version>-<patch_version>**. For example: **default-v1-24-3**.

Additional resources

- istio-csr deployment

Next steps

- Installing your Istio resource

4.2.3. Installing your Istio resource

After you have installed **istio-csr** by following the procedure for either an in place or revision based update strategy, you can install the **Istio** resource.

You need to disable Istio's built in CA server and tell istiod to use the **istio-csr** CA server. The **istio-csr** CA server issues certificates for both istiod and user workloads.

Procedure

1. Create the **Istio** object as shown in the following example:

   Example istio.yaml object

   ```
   apiVersion: sailoperator.io/v1
   kind: Istio
   metadata:
     name: default
   spec:
     version: v1.24.3
     namespace: istio-system
     values:
       global:
         caAddress: cert-manager-istio-csr.cert-manager.svc:443
       pilot:
         env:
           ENABLE_CA_SERVER: "false"
   ```

   NOTE

   If you installed your CSR agent with a revision based update strategy, then you need to add the following to your **Istio** object YAML:

   ```
   kind: Istio
   metadata:
     name: default
   spec:
     updateStrategy:
       type: RevisionBased
   ```

2. Create the **Istio** resource by running the following command:

   ```
   $ oc apply -f istio.yaml
   ```

3. Wait for the **Istio** object to become ready by running the following command:

   ```
   $ oc wait --for=condition=Ready istios/default -n istio-system
   ```

4.2.4. Verifying cert-manager installation

You can use the sample **httpbin** service and **sleep** application to check communication between the workloads. You can also check the workload certificate of the proxy to verify that the cert-manager tool is installed correctly.

Procedure

1. Create the **sample** namespace by running the following command:

   ```
   $ oc new-project sample
   ```

2. Find your active Istio revision by running the following command:

   ```
   $ oc get istios default -o jsonpath='{.status.activeRevisionName}'
   ```

3. Add the injection label for your active revision to the **sample** namespace by running the following command:

   ```
   $ oc label namespace sample istio.io/rev=<your-active-revision-name> --overwrite=true
   ```

4. Deploy the sample **httpbin** service by running the following command:

   ```
   $ oc apply -n sample -f https://raw.githubusercontent.com/openshift-service-
   mesh/istio/refs/heads/master/samples/httpbin/httpbin.yaml
   ```

5. Deploy the sample **sleep** application by running the following command:

   ```
   $ oc apply -n sample -f https://raw.githubusercontent.com/openshift-service-
   mesh/istio/refs/heads/master/samples/sleep/sleep.yaml
   ```

6. Wait for both applications to become ready by running the following command:

   ```
   $ oc rollout status -n sample deployment httpbin sleep
   ```

7. Verify that **sleep** application can access the **httpbin** service by running the following command:

   ```
   $ oc exec "$(oc get pod -l app=sleep -n sample \
       -o jsonpath={.items..metadata.name})" -c sleep -n sample -- \
       curl http://httpbin.sample:8000/ip -s -o /dev/null \
       -w "%{http_code}\n"
   ```

   ### Example of a successful output

   ```
   200
   ```

8. Run the following command to print the workload certificate for the **httpbin** service and verify the output:

   ```
   $ istioctl proxy-config secret -n sample $(oc get pods -n sample -o
   jsonpath='{.items..metadata.name}' --selector app=httpbin) -o json | jq -r
   '.dynamicActiveSecrets[0].secret.tlsCertificate.certificateChain.inlineBytes' | base64 --decode
   | openssl x509 -text -noout
   ```

Example output

```
...
Issuer: O = cert-manager + O = cluster.local, CN = istio-ca
...
X509v3 Subject Alternative Name:
  URI:spiffe://cluster.local/ns/sample/sa/httpbin
```

4.3. UPDATING ISTIO-CSR AGENTS WITH REVISION-BASED UPDATE STRATEGIES

If you deployed your Istio resource using the revision based update strategy, you must pass all revisions each time you update your control plane. You must perform the update in the following order:

1. Update the **istio-csr** deployment with the new revision.
2. Update the value of **Istio.spec.version** parameter/field.

Example update for RevisionBased control plane

In this example, the **controlplane** is being updated from **v1.24.0** to **1.24.1.**

1. Update the **istio-csr** deployment with the new revision by running the following command:

   ```
   $ helm upgrade cert-manager-istio-csr jetstack/cert-manager-istio-csr \
   --wait \
     --reuse-values \
     --set "app.istio.revisions={<old_revision>,<new_revision>}"
   ```

   where:

   old_revision

   Specifies the old revision in the **<istio-name>-v<major_version>-<minor_version>-<patch_version>** format. For example: **default-v1-24-0**.

   new_revision

   Specifies the new revision in the **<istio-name>-v<major_version>-<minor_version>-<patch_version>** format. For example: **default-v1-24-1**.

2. Update the **istio.spec.version** in the **Istio** object similar to the following example:

   #### Example istio.yaml file

   ```
   apiVersion: sailoperator.io/v1
   kind: Istio
   metadata:
     name: default
   spec:
     version: <new_revision> 1
   ```

   1 Update to the new revision prefixed with the letter *v*, such as **v1.24.1**

3. Remove the old revision from your **istio-csr** deployment by running the following command:

```
helm upgrade cert-manager-istio-csr jetstack/cert-manager-istio-csr \
  --install \
  --namespace cert-manager \
  --wait \
  --reuse-values \
  --set "app.istio.revisions={default-v1-24-1}"
```

CHAPTER 5. MULTI-CLUSTER TOPOLOGIES

Multi-Cluster topologies are useful for organizations with distributed systems or environments seeking enhanced scalability, fault tolerance, and regional redundancy.

5.1. ABOUT MULTI-CLUSTER MESH TOPOLOGIES

In a multi-cluster mesh topology, you install and manage a single Istio mesh across multiple OpenShift Container Platform clusters, enabling communication and service discovery between the services. Two factors determine the multi-cluster mesh topology: control plane topology and network topology. There are two options for each topology. Therefore, there are four possible multi-cluster mesh topology configurations.

- Multi-Primary Single Network: Combines the multi-primary control plane topology and the single network network topology models.
- Multi-Primary Multi-Network: Combines the multi-primary control plane topology and the multi-network network topology models.
- Primary-Remote Single Network: Combines the primary-remote control plane topology and the single network network topology models.
- Primary-Remote Multi-Network: Combines the primary-remote control plane topology and the multi-network network topology models.

5.1.1. Control plane topology models

A multi-cluster mesh must use one of the following control plane topologies:

- Multi-Primary: In this configuration, a control plane resides on every cluster. Each control plane observes the API servers in all of the other clusters for services and endpoints.
- Primary-Remote: In this configuration, the control plane resides only on one cluster, called the primary cluster. No control plane runs on any of the other clusters, called remote clusters. The control plane on the primary cluster discovers services and endpoints and configures the sidecar proxies for the workloads in all clusters.

5.1.2. Network topology models

A multi-cluster mesh must use one of the following network topologies:

- Single Network: All clusters reside on the same network and there is direct connectivity between the services in all the clusters. There is no need to use gateways for communication between the services across cluster boundaries.
- Multi-Network: Clusters reside on different networks and there is no direct connectivity between services. Gateways must be used to enable communication across network boundaries.

5.2. MULTI-CLUSTER CONFIGURATION OVERVIEW

To configure a multi-cluster topology you must perform the following actions:

- Install the OpenShift Service Mesh Operator for each cluster.
- Create or have access to root and intermediate certificates for each cluster.

- Apply the security certificates for each cluster.
- Install Istio for each cluster.

5.2.1. Creating certificates for a multi-cluster topology

Create the root and intermediate certificate authority (CA) certificates for two clusters.

Prerequisites

- You have OpenSSL installed locally.

Procedure

1. Create the root CA certificate:
    a. Create a key for the root certificate by running the following command:

    ```
    $ openssl genrsa -out root-key.pem 4096
    ```

    b. Create an OpenSSL configuration certificate file named **root-ca.conf** for the root CA certificates:

    #### Example root certificate configuration file

    ```
    encrypt_key = no
    prompt = no
    utf8 = yes
    default_md = sha256
    default_bits = 4096
    req_extensions = req_ext
    x509_extensions = req_ext
    distinguished_name = req_dn
    [ req_ext ]
    subjectKeyIdentifier = hash
    basicConstraints = critical, CA:true
    keyUsage = critical, digitalSignature, nonRepudiation, keyEncipherment, keyCertSign
    [ req_dn ]
    O = Istio
    CN = Root CA
    ```

    c. Create the certificate signing request by running the following command:

    ```
    $ openssl req -sha256 -new -key root-key.pem \
      -config root-ca.conf \
      -out root-cert.csr
    ```

    d. Create a shared root certificate by running the following command:

    ```
    $ openssl x509 -req -sha256 -days 3650 \
      -signkey root-key.pem \
      -extensions req_ext -extfile root-ca.conf \
      -in root-cert.csr \
      -out root-cert.pem
    ```

2. Create the intermediate CA certificate for the East cluster:

   a. Create a directory named **east** by running the following command:

   ```
   $ mkdir east
   ```

   b. Create a key for the intermediate certificate for the East cluster by running the following command:

   ```
   $ openssl genrsa -out east/ca-key.pem 4096
   ```

   c. Create an OpenSSL configuration file named **intermediate.conf** in the **east**/ directory for the intermediate certificate of the East cluster. Copy the following example file and save it locally:

   Example configuration file

   ```
   [ req ]
   encrypt_key = no
   prompt = no
   utf8 = yes
   default_md = sha256
   default_bits = 4096
   req_extensions = req_ext
   x509_extensions = req_ext
   distinguished_name = req_dn
   [ req_ext ]
   subjectKeyIdentifier = hash
   basicConstraints = critical, CA:true, pathlen:0
   keyUsage = critical, digitalSignature, nonRepudiation, keyEncipherment, keyCertSign
   subjectAltName=@san
   [ san ]
   DNS.1 = istiod.istio-system.svc
   [ req_dn ]
   O = Istio
   CN = Intermediate CA
   L = east
   ```

   d. Create a certificate signing request by running the following command:

   ```
   $ openssl req -new -config east/intermediate.conf \
     -key east/ca-key.pem \
     -out east/cluster-ca.csr
   ```

   e. Create the intermediate CA certificate for the East cluster by running the following command:

   ```
   $ openssl x509 -req -sha256 -days 3650 \
     -CA root-cert.pem \
     -CAkey root-key.pem -CAcreateserial \
     -extensions req_ext -extfile east/intermediate.conf \
     -in east/cluster-ca.csr \
     -out east/ca-cert.pem
   ```

f. Create a certificate chain from the intermediate and root CA certificate for the east cluster by running the following command:

```
$ cat east/ca-cert.pem root-cert.pem > east/cert-chain.pem && cp root-cert.pem east
```

3. Create the intermediate CA certificate for the West cluster:

   a. Create a directory named **west** by running the following command:

   ```
   $ mkdir west
   ```

   b. Create a key for the intermediate certificate for the West cluster by running the following command:

   ```
   $ openssl genrsa -out west/ca-key.pem 4096
   ```

   c. Create an OpenSSL configuration file named **intermediate.conf** in the **west/** directory for for the intermediate certificate of the West cluster. Copy the following example file and save it locally:

   Example configuration file

   ```
   [ req ]
   encrypt_key = no
   prompt = no
   utf8 = yes
   default_md = sha256
   default_bits = 4096
   req_extensions = req_ext
   x509_extensions = req_ext
   distinguished_name = req_dn
   [ req_ext ]
   subjectKeyIdentifier = hash
   basicConstraints = critical, CA:true, pathlen:0
   keyUsage = critical, digitalSignature, nonRepudiation, keyEncipherment, keyCertSign
   subjectAltName=@san
   [ san ]
   DNS.1 = istiod.istio-system.svc
   [ req_dn ]
   O = Istio
   CN = Intermediate CA
   L = west
   ```

   d. Create a certificate signing request by running the following command:

   ```
   $ openssl req -new -config west/intermediate.conf \
     -key west/ca-key.pem \
     -out west/cluster-ca.csr
   ```

   e. Create the certificate by running the following command:

   ```
   $ openssl x509 -req -sha256 -days 3650 \
     -CA root-cert.pem \
     -CAkey root-key.pem -CAcreateserial \
   ```

```
-extensions req_ext -extfile west/intermediate.conf \
-in west/cluster-ca.csr \
-out west/ca-cert.pem
```

f. Create the certificate chain by running the following command:

```
$ cat west/ca-cert.pem root-cert.pem > west/cert-chain.pem && cp root-cert.pem west
```

5.2.2. Applying certificates to a multi-cluster topology

Apply root and intermediate certificate authority (CA) certificates to the clusters in a multi-cluster topology.

NOTE

In this procedure, **CLUSTER1** is the East cluster and **CLUSTER2** is the West cluster.

Prerequisites

- You have access to two OpenShift Container Platform clusters with external load balancer support.
- You have created the root CA certificate and intermediate CA certificates for each cluster or someone has made them available for you.

Procedure

1. Apply the certificates to the East cluster of the multi-cluster topology:

   a. Log in to East cluster by running the following command:

   ```
   $ oc login -u https://<east_cluster_api_server_url>
   ```

   b. Set up the environment variable that contains the **oc** command context for the East cluster by running the following command:

   ```
   $ export CTX_CLUSTER1=$(oc config current-context)
   ```

   c. Create a project called **istio-system** by running the following command:

   ```
   $ oc get project istio-system --context "${CTX_CLUSTER1}" || oc new-project istio-
   system --context "${CTX_CLUSTER1}"
   ```

   d. Configure Istio to use **network1** as the default network for the pods on the East cluster by running the following command:

   ```
   $ oc --context "${CTX_CLUSTER1}" label namespace istio-system
   topology.istio.io/network=network1
   ```

   e. Create the CA certificates, certificate chain, and the private key for Istio on the East cluster by running the following command:

   ```
   $ oc get secret -n istio-system --context "${CTX_CLUSTER1}" cacerts || oc create secret
   ```

```
generic cacerts -n istio-system --context "${CTX_CLUSTER1}" \
  --from-file=east/ca-cert.pem \
  --from-file=east/ca-key.pem \
  --from-file=east/root-cert.pem \
  --from-file=east/cert-chain.pem
```

> **NOTE**
>
> If you followed the instructions in "Creating certificates for a multi-cluster mesh", your certificates will reside in the **east/** directory. If your certificates reside in a different directory, modify the syntax accordingly.

2. Apply the certificates to the West cluster of the multi-cluster topology:

   a. Log in to the West cluster by running the following command:

   ```
   $ oc login -u https://<west_cluster_api_server_url>
   ```

   b. Set up the environment variable that contains the **oc** command context for the West cluster by running the following command:

   ```
   $ export CTX_CLUSTER2=$(oc config current-context)
   ```

   c. Create a project called **istio-system** by running the following command:

   ```
   $ oc get project istio-system --context "${CTX_CLUSTER2}" || oc new-project istio-
   system --context "${CTX_CLUSTER2}"
   ```

   d. Configure Istio to use **network2** as the default network for the pods on the West cluster by running the following command:

   ```
   $ oc --context "${CTX_CLUSTER2}" label namespace istio-system
   topology.istio.io/network=network2
   ```

   e. Create the CA certificate secret for Istio on the West cluster by running the following command:

   ```
   $ oc get secret -n istio-system --context "${CTX_CLUSTER2}" cacerts || oc create secret
   generic cacerts -n istio-system --context "${CTX_CLUSTER2}" \
     --from-file=west/ca-cert.pem \
     --from-file=west/ca-key.pem \
     --from-file=west/root-cert.pem \
     --from-file=west/cert-chain.pem
   ```

   > **NOTE**
   >
   > If you followed the instructions in "Creating certificates for a multi-cluster mesh", your certificates will reside in the **west/** directory. If the certificates reside in a different directory, modify the syntax accordingly.

Next steps

Install Istio on all the clusters comprising the mesh topology.

5.3. INSTALLING A MULTI-PRIMARY MULTI-NETWORK MESH

Install Istio in the multi-primary multi-network topology on two OpenShift Container Platform clusters.

NOTE

In this procedure, **CLUSTER1** is the East cluster and **CLUSTER2** is the West cluster.

You can adapt these instructions for a mesh spanning more than two clusters.

Prerequisites

- You have installed the OpenShift Service Mesh 3 Operator on all of the clusters that comprise the mesh.
- You have completed "Creating certificates for a multi-cluster mesh".
- You have completed "Applying certificates to a multi-cluster topology".
- You have created an Istio Container Network Interface (CNI) resource.
- You have **istioctl** installed on the laptop you can use to run these instructions.

Procedure

1. Create an **ISTIO_VERSION** environment variable that defines the Istio version to install by running the following command:

```
$ export ISTIO_VERSION=1.24.3
```

2. Install Istio on the East cluster:

   a. Create an **Istio** resource on the East cluster by running the following command:

```
$ cat <<EOF | oc --context "${CTX_CLUSTER1}" apply -f -
apiVersion: sailoperator.io/v1
kind: Istio
metadata:
  name: default
spec:
  version: v${ISTIO_VERSION}
  namespace: istio-system
  values:
    global:
      meshID: mesh1
      multiCluster:
        clusterName: cluster1
      network: network1
EOF
```

   b. Wait for the control plane to return the **Ready** status condition by running the following command:

```
$ oc --context "${CTX_CLUSTER1}" wait --for condition=Ready istio/default --
timeout=3m
```

c. Create an East-West gateway on the East cluster by running the following command:

```
$ oc --context "${CTX_CLUSTER1}" apply -f https://raw.githubusercontent.com/istio-
ecosystem/sail-operator/main/docs/deployment-models/resources/east-west-gateway-
net1.yaml
```

d. Expose the services through the gateway by running the following command:

```
$ oc --context "${CTX_CLUSTER1}" apply -n istio-system -f
https://raw.githubusercontent.com/istio-ecosystem/sail-operator/main/docs/deployment-
models/resources/expose-services.yaml
```

3. Install Istio on the West cluster:

a. Create an **Istio** resource on the West cluster by running the following command:

```
$ cat <<EOF | oc --context "${CTX_CLUSTER2}" apply -f -
apiVersion: sailoperator.io/v1
kind: Istio
metadata:
  name: default
spec:
  version: v${ISTIO_VERSION}
  namespace: istio-system
  values:
    global:
      meshID: mesh1
      multiCluster:
        clusterName: cluster2
      network: network2
EOF
```

b. Wait for the control plane to return the **Ready** status condition by running the following command:

```
$ oc --context "${CTX_CLUSTER2}" wait --for condition=Ready istio/default --
timeout=3m
```

c. Create an East-West gateway on the West cluster by running the following command:

```
$ oc --context "${CTX_CLUSTER2}" apply -f https://raw.githubusercontent.com/istio-
ecosystem/sail-operator/main/docs/deployment-models/resources/east-west-gateway-
net2.yaml
```

d. Expose the services through the gateway by running the following command:

```
$ oc --context "${CTX_CLUSTER2}" apply -n istio-system -f
https://raw.githubusercontent.com/istio-ecosystem/sail-operator/main/docs/deployment-
models/resources/expose-services.yaml
```

4. Create the **istio-reader-service-account** service account for the East cluster by running the following command:

   ```
   $ oc --context="${CTX_CLUSTER1}" create serviceaccount istio-reader-service-account -n
   istio-system
   ```

5. Create the **istio-reader-service-account** service account for the West cluster by running the following command:

   ```
   $ oc --context="${CTX_CLUSTER2}" create serviceaccount istio-reader-service-account -n
   istio-system
   ```

6. Add the **cluster-reader** role to the East cluster by running the following command:

   ```
   $ oc --context="${CTX_CLUSTER1}" adm policy add-cluster-role-to-user cluster-reader -z
   istio-reader-service-account -n istio-system
   ```

7. Add the **cluster-reader** role to the West cluster by running the following command:

   ```
   $ oc --context="${CTX_CLUSTER2}" adm policy add-cluster-role-to-user cluster-reader -z
   istio-reader-service-account -n istio-system
   ```

8. Install a remote secret on the East cluster that provides access to the API server on the West cluster by running the following command:

   ```
   $ istioctl create-remote-secret \
     --context="${CTX_CLUSTER2}" \
     --name=cluster2 \
     --create-service-account=false | \
     oc --context="${CTX_CLUSTER1}" apply -f -
   ```

9. Install a remote secret on the West cluster that provides access to the API server on the East cluster by running the following command:

   ```
   $ istioctl create-remote-secret \
     --context="${CTX_CLUSTER1}" \
     --name=cluster1 \
     --create-service-account=false | \
     oc --context="${CTX_CLUSTER2}" apply -f -
   ```

5.3.1. Verifying a multi-cluster topology

Deploy sample applications and verify traffic on a multi-cluster topology on two OpenShift Container Platform clusters.

NOTE

In this procedure, **CLUSTER1** is the East cluster and **CLUSTER2** is the West cluster.

Prerequisites

- You have installed the OpenShift Service Mesh Operator on all of the clusters that comprise the mesh.
- You have completed "Creating certificates for a multi-cluster mesh".
- You have completed "Applying certificates to a multi-cluster topology".
- You have created an Istio Container Network Interface (CNI) resource.
- You have **istioctl** installed on the laptop you will use to run these instructions.
- You have installed a multi-cluster topology.

Procedure

1. Deploy sample applications on the East cluster:
    a. Create a sample application namespace on the East cluster by running the following command:

    ```
    $ oc --context "${CTX_CLUSTER1}" get project sample || oc --
    context="${CTX_CLUSTER1}" new-project sample
    ```

    b. Label the application namespace to support sidecar injection by running the following command:

    ```
    $ oc --context="${CTX_CLUSTER1}" label namespace sample istio-injection=enabled
    ```

    c. Deploy the **helloworld** application:
        i. Create the **helloworld** service by running the following command:

        ```
        $ oc --context="${CTX_CLUSTER1}" apply \
          -f https://raw.githubusercontent.com/openshift-service-mesh/istio/release-
        1.24/samples/helloworld/helloworld.yaml \
          -l service=helloworld -n sample
        ```

        ii. Create the **helloworld-v1** deployment by running the following command:

        ```
        $ oc --context="${CTX_CLUSTER1}" apply \
          -f https://raw.githubusercontent.com/openshift-service-mesh/istio/release-
        1.24/samples/helloworld/helloworld.yaml \
          -l version=v1 -n sample
        ```

    d. Deploy the **sleep** application by running the following command:

    ```
    $ oc --context="${CTX_CLUSTER1}" apply \
      -f https://raw.githubusercontent.com/openshift-service-mesh/istio/release-
    1.24/samples/sleep/sleep.yaml -n sample
    ```

    e. Wait for the **helloworld** application on the East cluster to return the **Ready** status condition by running the following command:

```
$ oc --context="${CTX_CLUSTER1}" wait --for condition=available -n sample
deployment/helloworld-v1
```

f. Wait for the **sleep** application on the East cluster to return the **Ready** status condition by running the following command:

```
$ oc --context="${CTX_CLUSTER1}" wait --for condition=available -n sample
deployment/sleep
```

2. Deploy the sample applications on the West cluster:

   a. Create a sample application namespace on the West cluster by running the following command:

   ```
   $ oc --context "${CTX_CLUSTER2}" get project sample || oc --
   context="${CTX_CLUSTER2}" new-project sample
   ```

   b. Label the application namespace to support sidecar injection by running the following command:

   ```
   $ oc --context="${CTX_CLUSTER2}" label namespace sample istio-injection=enabled
   ```

   c. Deploy the **helloworld** application:

      i. Create the **helloworld** service by running the following command:

      ```
      $ oc --context="${CTX_CLUSTER2}" apply \
        -f https://raw.githubusercontent.com/openshift-service-mesh/istio/release-
      1.24/samples/helloworld/helloworld.yaml \
        -l service=helloworld -n sample
      ```

      ii. Create the **helloworld-v2** deployment by running the following command:

      ```
      $ oc --context="${CTX_CLUSTER2}" apply \
        -f https://raw.githubusercontent.com/openshift-service-mesh/istio/release-
      1.24/samples/helloworld/helloworld.yaml \
        -l version=v2 -n sample
      ```

   d. Deploy the **sleep** application by running the following command:

   ```
   $ oc --context="${CTX_CLUSTER2}" apply \
     -f https://raw.githubusercontent.com/openshift-service-mesh/istio/release-
   1.24/samples/sleep/sleep.yaml -n sample
   ```

   e. Wait for the **helloworld** application on the West cluster to return the **Ready** status condition by running the following command:

   ```
   $ oc --context="${CTX_CLUSTER2}" wait --for condition=available -n sample
   deployment/helloworld-v2
   ```

   f. Wait for the **sleep** application on the West cluster to return the **Ready** status condition by running the following command:

```
$ oc --context="${CTX_CLUSTER2}" wait --for condition=available -n sample
deployment/sleep
```

Verifying traffic flows between clusters

1. For the East cluster, send 10 requests to the **helloworld** service by running the following command:

   ```
   $ for i in {0..9}; do \
     oc --context="${CTX_CLUSTER1}" exec -n sample deploy/sleep -c sleep -- curl -sS
   helloworld.sample:5000/hello; \
   done
   ```

   Verify that you see responses from both clusters. This means version 1 and version 2 of the service can be seen in the responses.

2. For the West cluster, send 10 requests to the **helloworld** service:

   ```
   $ for i in {0..9}; do \
     oc --context="${CTX_CLUSTER2}" exec -n sample deploy/sleep -c sleep -- curl -sS
   helloworld.sample:5000/hello; \
   done
   ```

   Verify that you see responses from both clusters. This means version 1 and version 2 of the service can be seen in the responses.

5.3.2. Removing a multi-cluster topology from a development environment

After experimenting with the multi-cluster functionality in a development environment, remove the multi-cluster topology from all the clusters.

NOTE

In this procedure, **CLUSTER1** is the East cluster and **CLUSTER2** is the West cluster.

Prerequisites

- You have installed a multi-cluster topology.

Procedure

1. Remove Istio and the sample applications from the East cluster of the development environment by running the following command:

   ```
   $ oc --context="${CTX_CLUSTER1}" delete istio/default ns/istio-system ns/sample ns/istio-
   cni
   ```

2. Remove Istio and the sample applications from the West cluster of development environment by running the following command:

   ```
   $ oc --context="${CTX_CLUSTER2}" delete istio/default ns/istio-system ns/sample ns/istio-
   cni
   ```

5.4. INSTALLING A PRIMARY-REMOTE MULTI-NETWORK MESH

Install Istio in a primary-remote multi-network topology on two OpenShift Container Platform clusters.

NOTE

In this procedure, **CLUSTER1** is the East cluster and **CLUSTER2** is the West cluster. The East cluster is the primary cluster and the West cluster is the remote cluster.

You can adapt these instructions for a mesh spanning more than two clusters.

Prerequisites

- You have installed the OpenShift Service Mesh 3 Operator on all of the clusters that comprise the mesh.
- You have completed "Creating certificates for a multi-cluster mesh".
- You have completed "Applying certificates to a multi-cluster topology".
- You have created an Istio Container Network Interface (CNI) resource.
- You have **istioctl** installed on the laptop you will use to run these instructions.

Procedure

1. Create an **ISTIO_VERSION** environment variable that defines the Istio version to install by running the following command:

```
$ export ISTIO_VERSION=1.24.3
```

2. Install Istio on the East cluster:

   a. Set the default network for the East cluster by running the following command:

```
$ oc --context="${CTX_CLUSTER1}" label namespace istio-system
topology.istio.io/network=network1
```

   b. Create an **Istio** resource on the East cluster by running the following command:

```
$ cat <<EOF | oc --context "${CTX_CLUSTER1}" apply -f -
apiVersion: sailoperator.io/v1
kind: Istio
metadata:
  name: default
spec:
  version: v${ISTIO_VERSION}
  namespace: istio-system
  values:
    global:
      meshID: mesh1
      multiCluster:
        clusterName: cluster1
      network: network1
      externalIstiod: true (1)
EOF
```

(1) This enables the control plane installed on the East cluster to serve as an external control plane for other remote clusters.

c. Wait for the control plane to return the "Ready" status condition by running the following command:

```
$ oc --context "${CTX_CLUSTER1}" wait --for condition=Ready istio/default --
timeout=3m
```

d. Create an East-West gateway on the East cluster by running the following command:

```
$ oc --context "${CTX_CLUSTER1}" apply -f https://raw.githubusercontent.com/istio-
ecosystem/sail-operator/main/docs/deployment-models/resources/east-west-gateway-
net1.yaml
```

e. Expose the control plane through the gateway so that services in the West cluster can access the control plane by running the following command:

```
$ oc --context "${CTX_CLUSTER1}" apply -n istio-system -f
https://raw.githubusercontent.com/istio-ecosystem/sail-operator/main/docs/deployment-
models/resources/expose-istiod.yaml
```

f. Expose the application services through the gateway by running the following command:

```
$ oc --context "${CTX_CLUSTER1}" apply -n istio-system -f
https://raw.githubusercontent.com/istio-ecosystem/sail-operator/main/docs/deployment-
models/resources/expose-services.yaml
```

3. Install Istio on the West cluster:

a. Save the IP address of the East-West gateway running in the East cluster by running the following command:

```
$ export DISCOVERY_ADDRESS=$(oc --context="${CTX_CLUSTER1}" \
    -n istio-system get svc istio-eastwestgateway \
    -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
```

b. Create an **Istio** resource on the West cluster by running the following command:

```
$ cat <<EOF | oc --context "${CTX_CLUSTER2}" apply -f -
apiVersion: sailoperator.io/v1
kind: Istio
metadata:
  name: default
spec:
  version: v${ISTIO_VERSION}
  namespace: istio-system
  profile: remote
  values:
    istiodRemote:
      injectionPath: /inject/cluster/cluster2/net/network2
    global:
      remotePilotAddress: ${DISCOVERY_ADDRESS}
EOF
```

c. Annotate the **istio-system** namespace in the West cluster so that it is managed by the control plane in the East cluster by running the following command:

```
$ oc --context="${CTX_CLUSTER2}" annotate namespace istio-system
topology.istio.io/controlPlaneClusters=cluster1
```

d. Set the default network for the West cluster by running the following command:

```
$ oc --context="${CTX_CLUSTER2}" label namespace istio-system
topology.istio.io/network=network2
```

e. Install a remote secret on the East cluster that provides access to the API server on the West cluster by running the following command:

```
$ istioctl create-remote-secret \
  --context="${CTX_CLUSTER2}" \
  --name=cluster2 | \
  oc --context="${CTX_CLUSTER1}" apply -f -
```

f. Wait for the **Istio** resource to return the "Ready" status condition by running the following command:

```
$ oc --context "${CTX_CLUSTER2}" wait --for condition=Ready istio/default --
timeout=3m
```

g. Create an East-West gateway on the West cluster by running the following command:

```
$ oc --context "${CTX_CLUSTER2}" apply -f https://raw.githubusercontent.com/istio-
ecosystem/sail-operator/main/docs/deployment-models/resources/east-west-gateway-
net2.yaml
```

NOTE

Since the West cluster is installed with a remote profile, exposing the application services on the East cluster exposes them on the East-West gateways of both clusters.

5.5. INSTALLING KIALI IN A MULTI-CLUSTER MESH

Install Kiali in a multi-cluster mesh configuration on two OpenShift Container Platform clusters.

NOTE

In this procedure, **CLUSTER1** is the East cluster and **CLUSTER2** is the West cluster.

You can adapt these instructions for a mesh spanning more than two clusters.

Prerequisites

- You have installed the latest Kiali Operator on each cluster.
- Istio installed in a multi-cluster configuration on each cluster.
- You have **istioctl** installed on the laptop you can use to run these instructions.
- You are logged in to the OpenShift Container Platform web console as a user with the **cluster-admin** role.
- You have configured a metrics store so that Kiali can query metrics from all the clusters. Kiali queries metrics and traces from their respective endpoints.

Procedure

1. Install Kiali on the East cluster:

    a. Create a YAML file named **kiali.yaml** that creates a namespace for the Kiali deployment.

    ### Example configuration

    ```
    apiVersion: kiali.io/v1alpha1
    kind: Kiali
    metadata:
      name: kiali
      namespace: istio-system
    spec:
      version: default
      external_services:
        prometheus:
          auth:
            type: bearer
            use_kiali_token: true
          thanos_proxy:
            enabled: true
          url: https://thanos-querier.openshift-monitoring.svc.cluster.local:9091
    ```

    NOTE

    The endpoint for this example uses OpenShift Monitoring to configure metrics. For more information, see "Configuring OpenShift Monitoring with Kiali".

    b. Apply the YAML file on the East cluster by running the following command:

    ```
    $ oc --context cluster1 apply -f kiali.yaml
    ```

    ### Example output

    ```
    kiali-istio-system.apps.example.com
    ```

2. Ensure that the Kiali custom resource (CR) is ready by running the following command:

```
$ oc wait --context cluster1 --for=condition=Successful kialis/kiali -n istio-system --
timeout=3m
```

Example output

```
kiali.kiali.io/kiali condition met
```

3. Display your Kiali Route hostname.

```
$ oc --context cluster1 get route kiali -n istio-system -o jsonpath='{.spec.host}'
```

4. Create a Kiali CR on the West cluster.

Example configuration

```
apiVersion: kiali.io/v1alpha1
kind: Kiali
metadata:
  name: kiali
  namespace: istio-system
spec:
  version: default
  auth:
    openshift:
      redirect_uris:
        # Replace kiali-route-hostname with the hostname from the previous step.
        - "https://{kiali-route-hostname}/api/auth/callback/cluster2"
  deployment:
    remote_cluster_resources_only: true
```

The Kiali Operator creates the resources necessary for the Kiali server on the East cluster to connect to the West cluster. The Kiali server is not installed on the West cluster.

5. Apply the YAML file on the West cluster by running the following command:

```
$ oc --context cluster2 apply -f kiali-remote.yaml
```

6. Ensure that the Kiali CR is ready by running the following command:

```
$ oc wait --context cluster2 --for=condition=Successful kialis/kiali -n istio-system --
timeout=3m
```

7. Create a remote cluster secret so that Kiali installation in the East cluster can access the West cluster.

   a. Create a long lived API token bound to the kiali-service-account in the West cluster. Kiali uses this token to authenticate to the West cluster.

   Example configuration

```
apiVersion: v1
kind: Secret
metadata:
    name: "kiali-service-account"
    namespace: "istio-system"
    annotations:
      kubernetes.io/service-account.name: "kiali-service-account"
  type: kubernetes.io/service-account-token
```

b. Apply the YAML file on the West cluster by running the following command:

```
$ oc --context cluster2 apply -f kiali-svc-account-token.yaml
```

c. Create a **kubeconfig** file and save it as a secret in the namespace on the East cluster where the Kiali deployment resides.
To simplify this process, use the **kiali-prepare-remote-cluster.sh** script to generate the **kubeconfig** file by running the following **curl** command:

```
$ curl -L -o kiali-prepare-remote-cluster.sh
https://raw.githubusercontent.com/kiali/kiali/master/hack/istio/multicluster/kiali-prepare-
remote-cluster.sh
```

d. Modify the script to make it executeable by running the following command:

```
chmod +x kiali-prepare-remote-cluster.sh
```

e. Execute the script so that it passes the East and West cluster contexts to the **kubeconfig** file by running the following command:

```
$ ./kiali-prepare-remote-cluster.sh --kiali-cluster-context cluster1 --remote-cluster-context
cluster2 --view-only false --kiali-resource-name kiali-service-account --remote-cluster-
namespace istio-system --process-kiali-secret true --process-remote-resources false --
remote-cluster-name cluster2
```

NOTE

Use the **--help** option to display additional details about how to use the script.

8. Trigger the reconciliation loop so that the Kiali Operator registers the remote secret that the CR contains by running the following command:

```
$ oc --context cluster1 annotate kiali kiali -n istio-system --overwrite
kiali.io/reconcile="$(date)"
```

9. Wait for Kiali resource to become ready by running the following command:

```
oc --context cluster1 wait --for=condition=Successful --timeout=2m kialis/kiali -n istio-system
```

10. Wait for Kiali server to become ready by running the following command:

```
oc --context cluster1 rollout status deployments/kiali -n istio-system
```

11. Log in to Kiali.

a. When you first access Kiali, log in to the cluster that contains the Kiali deployment. In this example, access the **East** cluster.

b. Display the hostname of the Kiali route by running the following command:

```
oc --context cluster1 get route kiali -n istio-system -o jsonpath='{.spec.host}'
```

c. Navigate to the Kiali URL in your browser: https://<your-kiali-route-hostname>.

12. Log in to the West cluster through Kiali.
In order to see other clusters in the Kiali UI, you must first login as a user to those clusters through Kiali.

a. Click on the user profile dropdown in the top right hand menu.

b. Select **Login to West** You are redirected to an OpenShift login page and prompted for credentials for the West cluster.

13. Verify that Kiali shows information from both clusters.

a. Click **Overview** and verify that you can see namespaces from both clusters.

b. Click **Navigate** and verify that you see both clusters on the mesh graph.

Additional resources

- Using Kiali Operator provided by Red Hat

Next steps

- Verifying a multi-cluster topology
- Removing a multi-cluster topology from a development environment

CHAPTER 6. DEPLOYING MULTIPLE SERVICE MESHES ON A SINGLE CLUSTER

You can use the Red Hat OpenShift Service Mesh to operate multiple service meshes in a single cluster, with each mesh managed by a separate control plane. Using discovery selectors and revisions prevents conflicts between control planes.

6.1. PREREQUISITES

- You have installed the OpenShift Service Mesh operator.
- You have created an Istio Container Network Interface (CNI) resource.

  NOTE

  You can run the following command to check for existing **Istio** instances:

  ```
  $ oc get istios
  ```

- You have installed the **istioctl** binary on your localhost.

6.2. ABOUT DEPLOYING MULTIPLE CONTROL PLANES

To configure a cluster to host two control planes, set up separate Istio resources with unique names in independent Istio system namespaces. Assign a unique revision name to each Istio resource to identify the control planes, workloads, or namespaces it manages. Apply these revision names using injection or **istio.io/rev** labels to specify which control plane injects the sidecar proxy into application pods.

Each **Istio** resource must also configure discovery selectors to specify which namespaces the Istio control plane observes. Only namespaces with labels that match the configured discovery selectors can join the mesh. Additionally, discovery selectors determine which control plane creates the **istio-ca-root-cert** config map in each namespace, which is used to encrypt traffic between services with mutual TLS within each mesh.

When adding an additional Istio control plane to a cluster with an existing control plane, ensure that the existing **Istio** instance has discovery selectors configured to avoid overlapping with the new control plane.

NOTE

Only one **IstioCNI** resource is shared by all control planes in a cluster, and you must update this resource independent of other cluster resources.

6.3. USING MULTIPLE CONTROL PLANES ON A SINGLE CLUSTER

You can use discovery selectors to limit the visibility of an Istio control plane to specific namespaces in a cluster. By combining discovery selectors with control plane revisions, you can deploy multiple control planes in a single cluster, ensuring that each control plane manages only its assigned namespaces. This approach avoids conflicts between control planes and enables soft multi-tenancy for service meshes.

6.4. DEPLOYING MULTIPLE CONTROL PLANES

You can have extended support for more than two control planes. The maximum number of service meshes in a single cluster depends on the available cluster resources.

6.4.1. Deploying the first control plane

You deploy the first control plane by creating its assigned namespace.

Procedure

1. Create the namespace for the first Istio control plane called **istio-system-1** by running the following command:

```
$ oc new-project istio-system-1
```

2. Add the following label to the first namespace, which is used with the Istio **discoverySelectors** field by running the following command:

```
$ oc label namespace istio-system-1 istio-discovery=mesh-1
```

3. Create a YAML file named **istio-1.yaml** with the name **mesh-1** and the **discoverySelector** as **mesh-1**:

Example configuration

```
kind: Istio
apiVersion: sailoperator.io/v1
metadata:
  name: mesh-1
spec:
  namespace: istio-system-1
  values:
    meshConfig:
      discoverySelectors:
        - matchLabels:
            istio-discovery: mesh-1
# ...
```

4. Create the first **Istio** resource by running the following command:

```
$ oc apply -f istio-1.yaml
```

5. To restrict workloads in **mesh-1** from communicating freely with decrypted traffic between meshes, deploy a **PeerAuthentication** resource to enforce mutual TLS (mTLS) traffic within the **mesh-1** data plane. Apply the **PeerAuthentication** resource in the **istio-system-1** namespace by using a configuration file, such as **peer-auth-1.yaml**:

```
$ oc apply -f peer-auth-1.yaml
```

Example configuration

```
apiVersion: security.istio.io/v1
kind: PeerAuthentication
```

```
metadata:
  name: "mesh-1-peerauth"
  namespace: "istio-system-1"
spec:
  mtls:
    mode: STRICT
```

6.4.2. Deploying the second control plane

After deploying the first control plane, you can deploy the second control plane by creating its assigned namespace.

Procedure

1. Create a namespace for the second Istio control plane called **istio-system-2** by running the following command:

   ```
   $ oc new-project istio-system-2
   ```

2. Add the following label to the second namespace, which is used with the Istio **discoverySelectors** field by running the following command:

   ```
   $ oc label namespace istio-system-2 istio-discovery=mesh-2
   ```

3. Create a YAML file named **istio-2.yaml**:

   Example configuration

   ```
   kind: Istio
   apiVersion: sailoperator.io/v1
   metadata:
     name: mesh-2
   spec:
     namespace: istio-system-2
     values:
       meshConfig:
         discoverySelectors:
           - matchLabels:
               istio-discovery: mesh-2
   # ...
   ```

4. Create the second **Istio** resource by running the following command:

   ```
   $ oc apply -f istio-2.yaml
   ```

5. Deploy a policy for workloads in the **istio-system-2** namespace to only accept mutual TLS traffic **peer-auth-2.yaml** by running the following command:

   ```
   $ oc apply -f peer-auth-2.yaml
   ```

   Example configuration

```
apiVersion: security.istio.io/v1
kind: PeerAuthentication
metadata:
  name: "mesh-2-peerauth"
  namespace: "istio-system-2"
spec:
  mtls:
    mode: STRICT
```

6.4.3. Verifying multiple control planes

Verify that both of the Istio control planes are deployed and running properly. You can validate that the **istiod** pod is successfully running in each Istio system namespace.

1. Verify that the workloads are assigned to the control plane in **istio-system-1** by running the following command:

   ```
   $ oc get pods -n istio-system-1
   ```

   Example output

   ```
   NAME                           READY   STATUS    RESTARTS   AGE
   istiod-mesh-1-b69646b6f-kxrwk   1/1     Running   0          4m14s
   ```

2. Verify that the workloads are assigned to the control plane in **istio-system-2** by running the following command:

   ```
   $ oc get pods -n istio-system-2
   ```

   Example output

   ```
   NAME                           READY   STATUS    RESTARTS   AGE
   istiod-mesh-2-8666fdfc6-mqp45   1/1     Running   0          118s
   ```

6.5. DEPLOY APPLICATION WORKLOADS IN EACH MESH

To deploy application workloads, assign each workload to a separate namespace.

Procedure

1. Create an application namespace called **app-ns-1** by running the following command:

   ```
   $ oc create namespace app-ns-1
   ```

2. To ensure that the namespace is discovered by the first control plane, add the **istio-discovery=mesh-1** label by running the following command:

   ```
   $ oc label namespace app-ns-1 istio-discovery=mesh-1
   ```

3. To enable sidecar injection into all the pods by default while ensuring that pods in this namespace are mapped to the first control plane, add the **istio.io/rev=mesh-1** label to the namespace by running the following command:

```
$ oc label namespace app-ns-1 istio.io/rev=mesh-1
```

4. Optional: You can verify the **mesh-1** revision name by running the following command:

```
$ oc get istiorevisions
```

5. Deploy the **sleep** and **httpbin** applications by running the following command:

```
$ oc apply -n app-ns-1 \
  -f https://raw.githubusercontent.com/openshift-service-mesh/istio/release-
1.24/samples/sleep/sleep.yaml \
  -f https://raw.githubusercontent.com/openshift-service-mesh/istio/release-
1.24/samples/httpbin/httpbin.yaml
```

6. Wait for the **httpbin** and **sleep** pods to run with sidecars injected by running the following command:

```
$ oc get pods -n app-ns-1
```

Example output

```
NAME                      READY   STATUS    RESTARTS   AGE
httpbin-7f56dc944b-kpw2x   2/2     Running   0          2m26s
sleep-5577c64d7c-b5wd2     2/2     Running   0          91m
```

7. Create a second application namespace called **app-ns-2** by running the following command:

```
$ oc create namespace app-ns-2
```

8. Create a third application namespace called **app-ns-3** by running the following command:

```
$ oc create namespace app-ns-3
```

9. Add the label **istio-discovery=mesh-2** to both namespaces and the revision label **mesh-2** to match the discovery selector of the second control plane by running the following command:

```
$ oc label namespace app-ns-2 app-ns-3 istio-discovery=mesh-2 istio.io/rev=mesh-2
```

10. Deploy the **sleep** and **httpbin** applications to the **app-ns-2** namespace by running the following command:

```
$ oc apply -n app-ns-2 \
  -f https://raw.githubusercontent.com/openshift-service-mesh/istio/release-
1.24/samples/sleep/sleep.yaml \
  -f https://raw.githubusercontent.com/openshift-service-mesh/istio/release-
1.24/samples/httpbin/httpbin.yaml
```

11. Deploy the **sleep** and **httpbin** applications to the **app-ns-3** namespace by running the following command:

```
$ oc apply -n app-ns-3 \
  -f https://raw.githubusercontent.com/openshift-service-mesh/istio/release-
1.24/samples/sleep/sleep.yaml \
  -f https://raw.githubusercontent.com/openshift-service-mesh/istio/release-
1.24/samples/httpbin/httpbin.yaml
```

12. Optional: Use the following command to wait for a deployment to be available:

```
$ oc wait deployments -n app-ns-2 --all --for condition=Available
```

Verification

1. Verify that each application workload is managed by its assigned control plane by using the **istioctl ps** command after deploying the applications:

   a. Verify that the workloads are assigned to the control plane in **istio-system-1** by running the following command:

   ```
   $ istioctl ps -i istio-system-1
   ```

   ### Example output

   ```
   NAME                          CLUSTER      CDS           LDS           EDS           RDS
   ECDS       ISTIOD                       VERSION
   httpbin-7f56dc944b-vwfm5.app-ns-1    Kubernetes    SYNCED (11m)    SYNCED (11m)
   SYNCED (11m)    SYNCED (11m)    IGNORED    istiod-mesh-1-b69646b6f-kxrwk
   1.23.0
   sleep-5577c64d7c-d675f.app-ns-1      Kubernetes    SYNCED (11m)    SYNCED (11m)
   SYNCED (11m)    SYNCED (11m)    IGNORED    istiod-mesh-1-b69646b6f-kxrwk
   1.23.0
   ```

   b. Verify that the workloads are assigned to the control plane in **istio-system-2** by running the following command:

   ```
   $ istioctl ps -i istio-system-2
   ```

   ### Example output

   ```
   NAME                          CLUSTER      CDS            LDS             EDS
   RDS             ECDS       ISTIOD                      VERSION
   httpbin-7f56dc944b-54gjs.app-ns-3    Kubernetes    SYNCED (3m59s)    SYNCED
   (3m59s)    SYNCED (3m59s)    SYNCED (3m59s)    IGNORED    istiod-mesh-2-
   8666fdfc6-mqp45    1.23.0
   httpbin-7f56dc944b-gnh72.app-ns-2    Kubernetes    SYNCED (4m1s)     SYNCED
   (4m1s)     SYNCED (3m59s)    SYNCED (4m1s)     IGNORED    istiod-mesh-2-
   8666fdfc6-mqp45    1.23.0
   sleep-5577c64d7c-k9mxz.app-ns-2      Kubernetes    SYNCED (4m1s)     SYNCED
   (4m1s)     SYNCED (3m59s)    SYNCED (4m1s)     IGNORED    istiod-mesh-2-
   8666fdfc6-mqp45    1.23.0
   ```

```
sleep-5577c64d7c-m9hvm.app-ns-3      Kubernetes     SYNCED (4m1s)      SYNCED
(4m1s)      SYNCED (3m59s)     SYNCED (4m1s)      IGNORED     istiod-mesh-2-
8666fdfc6-mqp45     1.23.0
```

2. Verify that the application connectivity is restricted to workloads within their respective mesh:

   a. Send a request from the **sleep** pod in **app-ns-1** to the **httpbin** service in **app-ns-2** to check that the communication fails by running the following command:

   ```
   $ oc -n app-ns-1 exec deploy/sleep -c sleep -- curl -sIL http://httpbin.app-ns-
   2.svc.cluster.local:8000
   ```

   The **PeerAuthentication** resources created earlier enforce mutual TLS (mTLS) traffic in **STRICT** mode within each mesh. Each mesh uses its own root certificate, managed by the **istio-ca-root-cert** config map, which prevents communication between meshes. The output indicates a communication failure, similar to the following example:

   Example output

   ```
   HTTP/1.1 503 Service Unavailable
   content-length: 95
   content-type: text/plain
   date: Wed, 16 Oct 2024 12:05:37 GMT
   server: envoy
   ```

   b. Confirm that the communication works by sending a request from the **sleep** pod to the **httpbin** service that are present in the **app-ns-2** namespace which is managed by **mesh-2**. Run the following command:

   ```
   $ oc -n app-ns-2 exec deploy/sleep -c sleep -- curl -sIL http://httpbin.app-ns-
   3.svc.cluster.local:8000
   ```

   Example output

   ```
   HTTP/1.1 200 OK
   access-control-allow-credentials: true
   access-control-allow-origin: *
   content-security-policy: default-src 'self'; style-src 'self' 'unsafe-inline'; img-src 'self'
   camo.githubusercontent.com
   content-type: text/html; charset=utf-8
   date: Wed, 16 Oct 2024 12:06:30 GMT
   x-envoy-upstream-service-time: 8
   server: envoy
   transfer-encoding: chunked
   ```

6.6. ADDITIONAL RESOURCES

- Scoping Service Mesh with discoverySelectors

CHAPTER 7. EXTERNAL CONTROL PLANE TOPOLOGY

Use the external control plane topology to isolate the control plane from the data plane on separate clusters.

7.1. ABOUT EXTERNAL CONTROL PLANE TOPOLOGY

The external control plane topology improves security and allows the Service Mesh to be hosted as a service. In this installation configuration one cluster hosts and manages the Istio control plane, and applications are hosted on other clusters.

7.1.1. Installing the control plane and data plane on separate clusters

Install Istio on a control plane cluster and a separate data plane cluster. This installation approach provides increased security.

NOTE

You can adapt these instructions for a mesh spanning more than one data plane cluster. You can also adapt these instructions for multiple meshes with multiple control planes on the same control plane cluster.

Prerequisites

- You have installed the OpenShift Service Mesh Operator on the control plane cluster and the data plane cluster.
- You have **istioctl** installed on the laptop you will use to run these instructions.

Procedure

1. Create an **ISTIO_VERSION** environment variable that defines the Istio version to install on all the clusters by running the following command:

   ```
   $ export ISTIO_VERSION=1.24.3
   ```

2. Create a **REMOTE_CLUSTER_NAME** environment variable that defines the name of the cluster by running the following command:

   ```
   $ export REMOTE_CLUSTER_NAME=cluster1
   ```

3. Set up the environment variable that contains the **oc** command context for the control plane cluster by running the following command:

   ```
   $ export CTX_CONTROL_PLANE_CLUSTER=
   <context_name_of_the_control_plane_cluster>
   ```

4. Set up the environment variable that contains the **oc** command context for the data plane cluster by running the following command:

   ```
   $ export CTX_DATA_PLANE_CLUSTER=<context_name_of_the_data_plane_cluster>
   ```

5. Set up the ingress gateway for the control plane:

    a. Create a project called **istio-system** by running the following command:

    ```
    $ oc get project istio-system --context "${CTX_CONTROL_PLANE_CLUSTER}" || oc
    new-project istio-system --context "${CTX_CONTROL_PLANE_CLUSTER}"
    ```

    b. Create an **Istio** resource on the control plane cluster to manage the ingress gateway by running the following command:

    ```
    $ cat <<EOF | oc --context "${CTX_CONTROL_PLANE_CLUSTER}" apply -f -
    apiVersion: sailoperator.io/v1
    kind: Istio
    metadata:
      name: default
    spec:
      version: v${ISTIO_VERSION}
      namespace: istio-system
      value:
        global:
          network: network1
    EOF
    ```

    c. Create the ingress gateway for the control plane by running the following command:

    ```
    $ oc --context "${CTX_CONTROL_PLANE_CLUSTER}" apply -f
    https://raw.githubusercontent.com/istio-ecosystem/sail-operator/main/docs/deployment-
    models/resources/controlplane-gateway.yaml
    ```

    d. Get the assigned IP address for the ingress gateway by running the following command:

    ```
    $ oc --context "${CTX_CONTROL_PLANE_CLUSTER}" get svc istio-ingressgateway -n
    istio-system -o jsonpath='{.status.loadBalancer.ingress[0].ip}'
    ```

    e. Store the IP address of the ingress gateway in an environment variable by running the following command:

    ```
    $ export EXTERNAL_ISTIOD_ADDR=$(oc -n istio-system --
    context="${CTX_CONTROL_PLANE_CLUSTER}" get svc istio-ingressgateway -o
    jsonpath='{.status.loadBalancer.ingress[0].ip}')
    ```

6. Install Istio on the data plane cluster:

    a. Create a project called **external-istiod** on the data plane cluster by running the following command:

    ```
    $ oc get project external-istiod --context "${CTX_DATA_PLANE_CLUSTER}" || oc new-
    project external-istiod --context "${CTX_DATA_PLANE_CLUSTER}"
    ```

    b. Create an **Istio** resource on the data plane cluster by running the following command:

    ```
    $ cat <<EOF | oc --context "${CTX_DATA_PLANE_CLUSTER}" apply -f -
    apiVersion: sailoperator.io/v1
    kind: Istio
    ```

```
metadata:
  name: external-istiod
spec:
  version: v${ISTIO_VERSION}
  namespace: external-istiod
  profile: remote
  values:
    defaultRevision: external-istiod
    global:
      remotePilotAddress: ${EXTERNAL_ISTIOD_ADDR}
      configCluster: true (1)
    pilot:
      configMap: true
      istiodRemote:
        injectionPath: /inject/cluster/cluster2/net/network1
EOF
```

(1) This setting identifies the data plane cluster as the source of the mesh configuration.

7. Create a project called **istio-cni** on the data plane cluster by running the following command:

```
$ oc get project istio-cni --context "${CTX_DATA_PLANE_CLUSTER}" || oc new-project istio-cni --context "${CTX_DATA_PLANE_CLUSTER}"
```

   a. Create an **IstioCNI** resource on the data plane cluster by running the following command:

```
$ cat <<EOF | oc --context "${CTX_DATA_PLANE_CLUSTER}" apply -f -
apiVersion: sailoperator.io/v1
kind: IstioCNI
metadata:
  name: default
spec:
  version: v${ISTIO_VERSION}
  namespace: istio-cni
EOF
```

8. Set up the external Istio control plane on the control plane cluster:

   a. Create a project called **external-istiod** on the control plane cluster by running the following command:

```
$ oc get project external-istiod --context "${CTX_CONTROL_PLANE_CLUSTER}" || oc new-project external-istiod --context "${CTX_CONTROL_PLANE_CLUSTER}"
```

   b. Create a **ServiceAccount** resource on the control plane cluster by running the following command:

```
$ oc --context="${CTX_CONTROL_PLANE_CLUSTER}" create serviceaccount istiod-service-account -n external-istiod
```

   c. Store the API server address for the data plane cluster in an environment variable by running the following command:

```
$
DATA_PLANE_API_SERVER=https://<hostname_or_IP_address_of_the_API_server_for_t
he_data_plane_cluster>:6443
```

d. Install a remote secret on the control plane cluster that provides access to the API server on the data plane cluster by running the following command:

```
$ istioctl create-remote-secret \
  --context="${CTX_DATA_PLANE_CLUSTER}" \
  --type=config \
  --namespace=external-istiod \
  --service-account=istiod-external-istiod \
  --create-service-account=false \
  --server="${DATA_PLANE_API_SERVER}" | \
  oc --context="${CTX_CONTROL_PLANE_CLUSTER}" apply -f -
```

e. Create an **Istio** resource on the control plane cluster by running the following command:

```
$ cat <<EOF | oc --context "${CTX_CONTROL_PLANE_CLUSTER}" apply -f -
apiVersion: sailoperator.io/v1
kind: Istio
metadata:
  name: external-istiod
spec:
  version: v${ISTIO_VERSION}
  namespace: external-istiod
  profile: empty
  values:
    meshConfig:
      rootNamespace: external-istiod
      defaultConfig:
        discoveryAddress: $EXTERNAL_ISTIOD_ADDR:15012
    pilot:
      enabled: true
      volumes:
        - name: config-volume
          configMap:
            name: istio-external-istiod
        - name: inject-volume
          configMap:
            name: istio-sidecar-injector-external-istiod
      volumeMounts:
        - name: config-volume
          mountPath: /etc/istio/config
        - name: inject-volume
          mountPath: /var/lib/istio/inject
      env:
        INJECTION_WEBHOOK_CONFIG_NAME: "istio-sidecar-injector-external-istiod-
external-istiod"
        VALIDATION_WEBHOOK_CONFIG_NAME: "istio-validator-external-istiod-external-
istiod"
        EXTERNAL_ISTIOD: "true"
        LOCAL_CLUSTER_SECRET_WATCHER: "true"
        CLUSTER_ID: cluster2
        SHARED_MESH_CONFIG: istio
    global:
      caAddress: $EXTERNAL_ISTIOD_ADDR:15012
      configValidation: false
      meshID: mesh1
      multiCluster:
        clusterName: cluster2
      network: network1
EOF
```

f. Create **Gateway** and **VirtualService** resources so that the sidecar proxies on the data plane cluster can access the control plane by running the following command:

```
$ oc --context "${CTX_CONTROL_PLANE_CLUSTER}" apply -f - <<EOF
apiVersion: networking.istio.io/v1
kind: Gateway
metadata:
  name: external-istiod-gw
  namespace: external-istiod
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 15012
        protocol: tls
        name: tls-XDS
      tls:
        mode: PASSTHROUGH
      hosts:
      - "*"
    - port:
        number: 15017
        protocol: tls
        name: tls-WEBHOOK
      tls:
        mode: PASSTHROUGH
      hosts:
      - "*"
---
apiVersion: networking.istio.io/v1
kind: VirtualService
metadata:
  name: external-istiod-vs
  namespace: external-istiod
spec:
    hosts:
    - "*"
    gateways:
    - external-istiod-gw
    tls:
    - match:
      - port: 15012
        sniHosts:
        - "*"
      route:
      - destination:
        host: istiod-external-istiod.external-istiod.svc.cluster.local
        port:
          number: 15012
  - match:
    - port: 15017
      sniHosts:
      - "*"
    route:
    - destination:
        host: istiod-external-istiod.external-istiod.svc.cluster.local
        port:
          number: 443
EOF
```

g. Wait for the **external-istiod Istio** resource on the control plane cluster to return the "Ready" status condition by running the following command:

```
$ oc --context "${CTX_CONTROL_PLANE_CLUSTER}" wait --for condition=Ready
istio/external-istiod --timeout=3m
```

h. Wait for the **Istio** resource on the data plane cluster to return the "Ready" status condition by running the following command:

```
$ oc --context "${CTX_DATA_PLANE_CLUSTER}" wait --for condition=Ready
istio/external-istiod --timeout=3m
```

i. Wait for the **IstioCNI** resource on the data plane cluster to return the "Ready" status condition by running the following command:

```
$ oc --context "${CTX_DATA_PLANE_CLUSTER}" wait --for condition=Ready
istiocni/default --timeout=3m
```

Verification

1. Deploy sample applications on the data plane cluster:

   a. Create a namespace for sample applications on the data plane cluster by running the following command:

   ```
   $ oc --context "${CTX_DATA_PLANE_CLUSTER}" get project sample || oc --
   context="${CTX_DATA_PLANE_CLUSTER}" new-project sample
   ```

   b. Label the namespace for the sample applications to support sidecar injection by running the following command:

   ```
   $ oc --context="${CTX_DATA_PLANE_CLUSTER}" label namespace sample
   istio.io/rev=external-istiod
   ```

   c. Deploy the **helloworld** application:

      i. Create the **helloworld** service by running the following command:

      ```
      $ oc --context="${CTX_DATA_PLANE_CLUSTER}" apply \
        -f
      ```

```
https://raw.githubusercontent.com/istio/istio/${ISTIO_VERSION}/samples/helloworld/he
lloworld.yaml \
  -l service=helloworld -n sample
```

ii. Create the **helloworld-v1** deployment by running the following command:

```
$ oc --context="${CTX_DATA_PLANE_CLUSTER}" apply \
  -f
https://raw.githubusercontent.com/istio/istio/${ISTIO_VERSION}/samples/helloworld/he
lloworld.yaml \
  -l version=v1 -n sample
```

d. Deploy the **sleep** application by running the following command:

```
$ oc --context="${CTX_DATA_PLANE_CLUSTER}" apply \
  -f
https://raw.githubusercontent.com/istio/istio/${ISTIO_VERSION}/samples/sleep/sleep.yaml
-n sample
```

e. Verify that the pods on the **sample** namespace have a sidecar injected by running the following command:

```
$ oc --context="${CTX_DATA_PLANE_CLUSTER}" get pods -n sample
```

The terminal should return **2/2** for each pod on the **sample** namespace by running the following command:

Example output

```
NAME                             READY   STATUS    RESTARTS   AGE
helloworld-v1-6d65866976-jb6qc   2/2     Running   0          1m
sleep-5fcd8fd6c8-mg8n2           2/2     Running   0          1m
```

2. Verify that internal traffic can reach the applications on the cluster:

a. Verify a request can be sent to the **helloworld** application through the **sleep** application by running the following command:

```
$ oc exec --context="${CTX_DATA_PLANE_CLUSTER}" -n sample -c sleep
deploy/sleep -- curl -sS helloworld.sample:5000/hello
```

The terminal should return a response from the **helloworld** application:

Example output

```
Hello version: v1, instance: helloworld-v1-6d65866976-jb6qc
```

3. Install an ingress gateway to expose the sample application to external clients:

a. Create the ingress gateway by running the following command:

```
$ oc --context="${CTX_DATA_PLANE_CLUSTER}" apply
-f https://raw.githubusercontent.com/istio-ecosystem/sail-
operator/refs/heads/main/chart/samples/ingress-gateway.yaml -n sample
```

b. Confirm that the ingress gateway is running by running the following command:

```
$ oc get pod -l app=istio-ingressgateway -n sample --
context="${CTX_DATA_PLANE_CLUSTER}"
```

The terminal should return output confirming that the gateway is running:

Example output

```
NAME                                    READY   STATUS    RESTARTS   AGE
istio-ingressgateway-7bcd5c6bbd-kmtl4   1/1     Running   0          8m4s
```

c. Expose the **helloworld** application through the ingress gateway by running the following command:

```
$ oc apply -f
https://raw.githubusercontent.com/istio/istio/refs/heads/master/samples/helloworld/helloworl
d-gateway.yaml -n sample --context="${CTX_DATA_PLANE_CLUSTER}"
```

d. Set the gateway URL environment variable by running the following command:

```
$ export INGRESS_HOST=$(oc -n sample --
context="${CTX_DATA_PLANE_CLUSTER}" get service istio-ingressgateway -o
jsonpath='{.status.loadBalancer.ingress[0].ip}'); \
  export INGRESS_PORT=$(oc -n sample --
context="${CTX_DATA_PLANE_CLUSTER}" get service istio-ingressgateway -o
jsonpath='{.spec.ports[?(@.name=="http2")].port}'); \
  export GATEWAY_URL=$INGRESS_HOST:$INGRESS_PORT
```

4. Verify that external traffic can reach the applications on the mesh:

a. Confirm that the **helloworld** application is accessible through the gateway by running the following command:

```
$ curl -s "http://${GATEWAY_URL}/hello"
```

The **helloworld** application should return a response.

Example output

```
Hello version: v1, instance: helloworld-v1-6d65866976-jb6qc
```

CHAPTER 8. ISTIOCTL TOOL

OpenShift Service Mesh 3 supports **istioctl**, the command line utility for the Istio project that includes many diagnostic and debugging utilities.

8.1. SUPPORT FOR ISTIOCTL

OpenShift Service Mesh 3 supports a selection of Istioctl commands.

Table 8.1. Supported Istioctl commands

Command	Description
admin	Manage the control plane (**istiod**) configuration
analyze	Analyze the Istio configuration and print validation messages
completion	Generate the autocompletion script for the specified shell
create-remote-secret	Create a secret with credentials to allow Istio to access remote Kubernetes API servers
help	Display help about any command
proxy-config, **pc**	Retrieve information about the proxy configuration from Envoy (Kubernetes only)
proxy-status, **ps**	Retrieve the synchronization status of each Envoy in the mesh
remote-clusters	List the remote clusters each **istiod** instance is connected to
validate, **v**	Validate the Istio policy and rules files
version	Print out the build version information
waypoint	Manage the waypoint configuration
ztunnel-config	Update or retrieve the current Ztunnel configuration.

8.2. INSTALLING THE ISTIOCTL TOOL

Install the **istioctl** command-line utility to debug and diagnose Istio service mesh deployments.

Prerequisites

- You have access to the OpenShift Container Platform web console.
- The OpenShift Service Mesh 3 Operator is installed and running.
- You have created at least one **Istio** resource.

Procedure

1. Confirm which version of the **Istio** resource runs on the installation by running the following command:

   ```
   $ oc get istio -ojsonpath="{range .items[*]}{.spec.version}{'\n'}{end}" | sed s/^v// | sort
   ```

   If there are multiple **Istio** resources with different versions, choose the latest version. The latest version is displayed last.

2. In the OpenShift Container Platform web console, click the **Help** icon and select **Command Line Tools**.

3. Click **Download istioctl**. Choose the version and architecture that matches your system.

   - Linux (x86_64, amd64)
   - Linux on ARM (aarch64, arm64)
   - MacOS (x86_64, amd64)
   - MacOS on ARM (aarch64, arm64)
   - Windows (x86_64, amd64)

4. Extract the **istioctl** binary file.

   a. If you are using a Linux operating system, run the following command:

      ```
      $ tar xzf istioctl-<VERSION>-<OS>-<ARCH>.tar.gz
      ```

   b. If you are using an Apple Mac operating system, unpack and extract the archive.

   c. If you are using a Microsoft Windows operating system, use the zip software to extract the archive.

5. Move to the uncompressed directory by running the following command:

   ```
   $ cd istioctl-<VERSION>-<OS>-<ARCH>
   ```

6. Add the **istioctl** client to the path by running the following command:

   ```
   $ export PATH=$PWD:$PATH
   ```

7. Confirm that the **istioctl** client version and the Istio control plane version match or are within one version by running the following command:

   ```
   $ istioctl version
   ```

Sample output:

```
client version: 1.20.0
control plane version: 1.24.3_ossm
data plane version: none
```

CHAPTER 9. ENABLING MUTUAL TRANSPORT LAYER SECURITY

You can use Red Hat OpenShift Service Mesh for your application to customize the communication security between the complex array of microservices. Mutual Transport Layer Security (mTLS) is a protocol that enables two parties to authenticate each other.

9.1. ABOUT MUTUAL TRANSPORT LAYER SECURITY (MTLS)

In OpenShift Service Mesh 3, you use the **Istio** resource instead of the **ServiceMeshControlPlane** resource to configure mTLS settings.

In OpenShift Service Mesh 3, you configure **STRICT** mTLS mode by using the **PeerAuthentication** and **DestinationRule** resources. You set TLS protocol versions through Istio Workload Minimum TLS Version Configuration.

Review the following **Istio** resources and concepts to configure mTLS settings properly:

PeerAuthentication

defines the type of mTLS traffic a sidecar accepts. In **PERMISSIVE** mode, both plaintext and mTLS traffic are accepted. In **STRICT** mode, only mTLS traffic is allowed.

DestinationRule

configures the type of TLS traffic a sidecar sends. In **DISABLE** mode, the sidecar sends plaintext. In **SIMPLE**, **MUTUAL**, and **ISTIO_MUTUAL** modes, the sidecar establishes a TLS connection.

Auto mTLS

ensures that all inter-mesh traffic is encrypted with mTLS by default, regardless of the **PeerAuthentication** mode configuration. **Auto mTLS** is controlled by the global mesh configuration field **enableAutoMtls**, which is enabled by default in OpenShift Service Mesh 2 and 3. The mTLS setting operates entirely between sidecar proxies, requiring no changes to application or service code.

By default, **PeerAuthentication** is set to **PERMISSIVE** mode, allowing sidecars in the Service Mesh to accept both plain-text and mTLS-encrypted traffic.

9.2. ENABLING STRICT MTLS MODE BY USING THE NAMESPACE

You can restrict workloads to accept only encrypted mTLS traffic by enabling the **STRICT** mode in **PeerAuthentication**.

Example PeerAuthentication policy for a namespace

```
apiVersion: security.istio.io/v1
kind: PeerAuthentication
metadata:
  name: default
  namespace: <namespace>
spec:
  mtls:
    mode: STRICT
```

You can enable mTLS for all destination hosts in the **<namespace>** by creating a **DestinationRule** resource with **MUTUAL** or **ISTIO_MUTUAL** mode when **auto mTLS** is disabled and **PeerAuthentication** is set to **STRICT** mode.

Example DestinationRule policy for a namespace

```
apiVersion: networking.istio.io/v1
kind: DestinationRule
metadata:
  name: enable-mtls
  namespace: <namespace>
spec:
  host: "*.<namespace>.svc.cluster.local"
  trafficPolicy:
   tls:
    mode: ISTIO_MUTUAL
```

9.3. ENABLING STRICT MTLS ACROSS THE WHOLE SERVICE MESH

You can configure mTLS across the entire mesh by applying the **PeerAuthentication** policy to the **istiod** namespace, such as **istio-system**. The **istiod** namespace name must match to the **spec.namespace** field of your **Istio** resource.

Example PeerAuthentication policy for the whole mesh

```
apiVersion: security.istio.io/v1
kind: PeerAuthentication
metadata:
  name: default
  namespace: istio-system
spec:
  mtls:
    mode: STRICT
```

Additionally, create a **DestinationRule** resource to disable mTLS for communication with the API server, as it does not have a sidecar. Apply similar **DestinationRule** configurations for other services without sidecars.

Example DestinationRule policy for the whole mesh

```
apiVersion: networking.istio.io/v1
kind: DestinationRule
metadata:
  name: api-server
  namespace: istio-system
spec:
  host: kubernetes.default.svc.cluster.local
  trafficPolicy:
    tls:
      mode: DISABLE
```

9.4. VALIDATING ENCRYPTIONS WITH KIALI

The Kiali console offers several ways to validate whether or not your applications, services, and workloads have mTLS encryption enabled.

The **Services Detail Overview** page displays a **Security** icon on the graph edges where at least one request with mTLS enabled is present. Also note that Kiali displays a lock icon in the **Network** section next to ports that are configured for mTLS.

9.5. ADDITIONAL RESOURCES

- Istio workload minimum TLS version configuration (Istio documentation)
- Understanding TLS configuration (Istio documentation)
- Permissive mode (Istio documentation)