Oracle® Cloud Native Environment

Service Mesh for Release 1.2

F40903-03
November 2021

Oracle Legal Notices

Table of Contents

Preface

This document contains information about the Istio module for Oracle Cloud Native Environment to set up a service mesh. It describes the differences from the upstream version, and includes information on installing and using the Istio module.

Document generated on: 2021-11-17 (revision: 1158)

Audience

This document is written for system administrators and developers who want to use the Istio module to create a service mesh in Oracle Cloud Native Environment. It is assumed that readers have a general understanding of the Oracle Linux operating system, container concepts and service mesh concepts.

Related Documents

The latest version of this document and other documentation for this product are available at:

https://docs.oracle.com/en/operating-systems/olcne/

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
https://www.oracle.com/corporate/accessibility/.

Access to Oracle Support for Accessibility

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit
https://www.oracle.com/corporate/accessibility/learning-support.html#support-tab.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Chapter 1 Introduction to the Service Mesh

This chapter provides introductory information about the Istio module for Oracle Cloud Native Environment, which is used to set up a service mesh in Oracle Cloud Native Environment.

1.1 What is a Service Mesh?

A *service mesh* is a configurable, low-latency infrastructure layer that controls the interaction between a network of microservices. A service mesh makes sure communication among containerized application infrastructure services is fast, reliable, and secure. The service mesh provides critical capabilities including service discovery, load balancing, encryption, observability, traceability, and authentication and authorization.

A service mesh provides the ability to monitor the microservices in the Kubernetes cluster. Istio can support most of the popular current deployment patterns for deploying microservices. This is transparent to a developer.

1.2 What is Istio?

Istio is a type of service mesh designed to manage the interaction and operation of services in a microservices architecture. Istio is an open source project that coordinates communication between services, providing service discovery, load balancing, security, recovery, telemetry, and policy enforcement capabilities. Istio uses a *sidecar* service mesh model. This means that network communication proxy capabilities are implemented in a separate container for every service or application container that is deployed. Envoy is the product that implements this proxy capability and these special containers run alongside every other container. The Istio sidecar service mesh frees developers from having to program these types of capabilities into application code and makes development and enhancement of applications in a microservice architecture much more efficient and rapid.

Istio's control plane provides an abstraction layer over the underlying cluster management platform, Kubernetes.

Istio contains the following components:

- **Envoy**: Sidecar proxies per microservice to handle ingress/egress traffic between services in the cluster and from a service to external services. The proxies form a secure microservice mesh providing a rich set of functions like discovery, rich layer-7 routing, policy enforcement and telemetry recording/reporting functions.
- **Istiod**: A component that incorporates Pilot, Mixer and Citadel into a single component. It is responsible for service discovery, configuration and certificate management.

Oracle Cloud Native Environment Release 1.1 installs an older version of Istio. The older Istio version does not include istiod. Instead, it includes Pilot, Mixer and Citadel. Oracle Cloud Native Environment Release 1.2 installs Istio Release 1.9.8 , which consolidates Pilot, Mixer and Citadel into a single component: istiod.

For more information on the Istio deployment architecture, see the upstream documentation at:

https://istio.io/latest/docs/ops/deployment/architecture/

1.3 About the Istio Module

The Istio module is based on a stable release of the upstream Istio project. Differences between Oracle versions of the software and upstream releases are limited to Oracle provided configuration profiles and patches for specific bugs.

For upstream Istio documentation, see https://istio.io/docs/.

For more information about Istio, see https://istio.io/.

1.4 Istio Module Components

The upstream Istio installation has a number of configuration profiles you can choose from. The Istio module components are based on the upstream installation configuration profiles, and includes components curated for Oracle Cloud Native Environment. You can see the upstream installation configuration profiles at:

https://istio.io/docs/setup/additional-setup/config-profiles/

The core Istio components installed with their corresponding container name prefix are:

- Egress gateway (`istio-egressgateway`)
- Ingress gateway (`istio-ingressgateway`)
- Istiod (`istiod`)

Note

Oracle Cloud Native Environment Release 1.1 includes a container named Pilot (`istio-pilot`) and not istiod (`istiod`) .

The add-on components installed with the Istio module are:

- Grafana (`grafana`)
- Prometheus (`prometheus`)

Chapter 2 Setting up a Service Mesh

This chapter discusses how to install the Istio module to set up a service mesh, and the components deployed when you do this.

The high level overview of setting up a service mesh is:

- **Oracle Cloud Native Environment**: Set up an environment in which to deploy the modules.
- **Kubernetes module**: You can install a service mesh into an existing Kubernetes cluster, or deploy the cluster at the same time.
- **Helm module**: Helm is a tool to manage Kubernetes packages and is used to install applications and resources into Kubernetes clusters. Helm interacts with the Kubernetes API server to install, upgrade, query and remove Kubernetes resources.
- **Istio module**: The Istio module deploys the required containers to deploy a service mesh, including the Istio ingress and egress gateways, Prometheus (a time-series metric collection database), and the cluster visualization tool Grafana.

When you deploy the Istio module, an embedded instance of Prometheus is also deployed. Prometheus is used to monitor and gather metrics about the Kubernetes cluster.

Another embedded component deployed with the Istio module is Grafana. Grafana is a monitoring and visualization tool for time-series data stored in a database which in this case is Prometheus. Grafana enables you to visually query and monitor the network traffic and services in your Kubernetes cluster. Grafana includes browser-based dashboards to visualize the cluster metrics gathered from Prometheus. For information on using Grafana, see *Monitoring and Visualization*.

Note

Prometheus and Grafana are automatically configured to provide standard metrics and dashboards for the Istio module. Persisting custom or manual configuration of these components is not possible.

You can enter all the information required to create a service mesh in a number of ways. The examples in this chapter show the following methods, although there are other combinations you can use:

- Create all modules in one `olcnectl module create` command, including the Kubernetes module. The example in Section 2.1, “Deploying a Service Mesh (Simple Method)” shows this method.
- Create all modules using multiple `olcnectl module create` commands. The example in Section 2.2, “Deploying a Service Mesh (Advanced Method)” shows this method.
- Create all modules using the `olcnectl module create` command by being prompted for values, interactively. The example in Section 2.3, “Deploying a Service Mesh (Interactive Method)” shows this method.

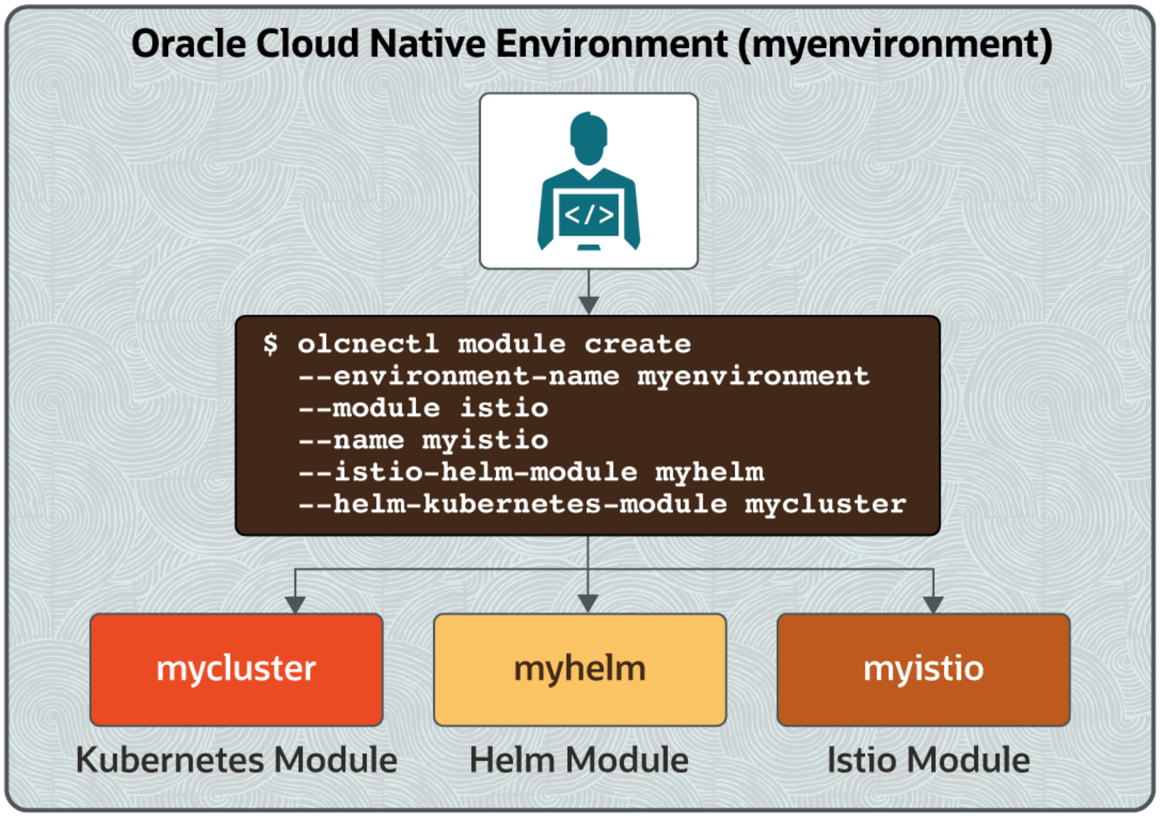
In reality, the method you use to deploy the modules depends on your own environment and preference, and whether you want to deploy into an existing Kubernetes cluster, create a Kubernetes cluster at the same time as deploying the service mesh, or take it step by step in some other way. There are many variations and possibilities you can use to do the deployment of a service mesh, or any module deployment for that matter.

2.1 Deploying a Service Mesh (Simple Method)

You can deploy all the modules required to create a service mesh and a Kubernetes cluster using a single `olcnectl module create` command. This method might be useful if you want to deploy a service mesh at the same time as deploying a Kubernetes cluster. If you have an existing deployment of the Kubernetes module, you can specify that instance when deploying a service mesh.

Figure 2.1, "Example Deployment" shows the modules deployed in this example. The name of each module in the examples is shown in the boxes. The examples create an Oracle Cloud Native Environment named `myenvironment`, into which a Kubernetes module named `mycluster` is deployed, and then into which a Helm module named `myhelm` is deployed, and finally, an Istio module named `myistio`.

Figure 2.1 Example Deployment



```
$ olcnectl module create
  --environment-name myenvironment
  --module istio
  --name myistio
  --istio-helm-module myhelm
  --helm-kubernetes-module mycluster
```

For the full list of the options available when creating modules, see the `olcnectl module create` command in *Platform Command-Line Interface*.

To deploy a service mesh:

1. If you do not already have an environment set up, create one into which the modules can be deployed. For information on setting up an environment, see *Container Orchestration* . The name of the environment in this example is `myenvironment`.

2. If you do not already have a Kubernetes module set up or deployed, set one up. For information on adding a Kubernetes module to an environment, see *Container Orchestration*. The name of the Kubernetes module in this example is `mycluster`.

3. Create the service mesh by adding the required modules to the Kubernetes module in the environment using the `olcnectl module create` command. This example uses a Kubernetes module named `mycluster`, into which it deploys a Helm module named `myhelm`, and finally, it creates an Istio module named `myistio`.

   ```
   olcnectl module create \
   --environment-name myenvironment \
   --module istio \
   --name myistio \
   --helm-kubernetes-module mycluster \
   --istio-helm-module myhelm
   ```

   The `--module` option sets the module type to create, which is `istio`. You define the name of the Istio module using the `--name` option, which in this case is `myistio`.

   As the Istio module requires Kubernetes and Helm, you must also provide the options for those modules.

   The `--helm-kubernetes-module` option sets the name of the Kubernetes module to use. The Kubernetes module should already be set up or deployed. If you have an existing Kubernetes module deployed, you can specify the name of the module using this option. If no Kubernetes module is deployed with the name you provide, a new module is deployed which allows you to deploy Kubernetes at the same time as a service mesh.

   The `--istio-helm-module` option sets the name of the Helm module to deploy.

   If you do not include all the required options when adding the modules you are prompted to provide them.

   Note

   The Istio module and Grafana also require an instance of Prometheus. When you deploy an Istio module, an embedded instance of Prometheus is created and deployed. You do not need to provide any information for the embedded Prometheus instance.

4. If you are deploying a new Kubernetes module, validate whether the module can be deployed to the nodes using the `olcnectl module validate` command. You do not need to perform this step if you have an existing Kubernetes module deployed to the nodes. For example:

   ```
   olcnectl module validate \
   --environment-name myenvironment \
   --name mycluster
   ```

5. If you are deploying a new Kubernetes module, use the `olcnectl module install` command to install it on the nodes. For example:

   ```
   olcnectl module install \
   --environment-name myenvironment \
   --name mycluster
   ```

6. Use the `olcnectl module validate` command to validate the Helm module can be deployed to the nodes. For example:

   ```
   olcnectl module validate \
   --environment-name myenvironment \
   --name myhelm
   ```

7. Use the `olcnectl module install` command to install the Helm module. For example:

   ```
   olcnectl module install \
   --environment-name myenvironment \
   --name myhelm
   ```

   The Helm software packages are installed on the control plane nodes, and the Helm module is deployed into the Kubernetes cluster.

8. Use the `olcnectl module validate` command to validate the Istio module can be deployed to the nodes. For example:

   ```
   olcnectl module validate \
   --environment-name myenvironment \
   --name myistio
   ```

9. Use the `olcnectl module install` command to install the Istio module. For example:

   ```
   olcnectl module install \
   --environment-name myenvironment \
   --name myistio
   ```

   The Istio software packages are installed on the control plane nodes, and the Istio module is deployed into the Kubernetes cluster.

2.2 Deploying a Service Mesh (Advanced Method)

You can deploy each module required to create a service mesh in a Kubernetes cluster individually. There are more steps in this method, but you have explicit control of how each module is created and deployed to your cluster. This example assumes a Kubernetes module named `mycluster` is already deployed into an environment named `myenvironment`.

To deploy a service mesh:

1. Use the `olcnectl module create` command to create a Helm module and add it to a Kubernetes module. For example, to create a Helm module named `myhelm` and add it to the Kubernetes module named `mycluster`:

   ```
   olcnectl module create \
   --environment-name myenvironment \
   --module helm \
   --name myhelm \
   --helm-kubernetes-module mycluster
   ```

   The `--module` option sets the module type to create, which is `helm`. You define the name of the Helm module using the `--name` option, which in this case is `myhelm`.

   The `--helm-kubernetes-module` option sets the name of the Kubernetes module into which Helm should be installed.

2. Use the `olcnectl module validate` command to validate the Helm module can be deployed to the nodes. For example, to validate the Helm module named `myhelm` in the environment named `myenvironment`:

   ```
   olcnectl module validate \
   --environment-name myenvironment \
   --name myhelm
   ```

3. Use the `olcnectl module install` command to deploy the Helm module. For example, to deploy the Helm module named `myhelm` in the environment named `myenvironment`:

   ```
   olcnectl module install \
   --environment-name myenvironment \
   --name myhelm
   ```

4. Use the `olcnectl module create` command to create an Istio module and associate it with the Helm module. For example, to create an Istio module named `myistio` and associate it with the Helm module named `myhelm`:

   ```
   olcnectl module create \
   --environment-name myenvironment \
   --module istio \
   --name myistio \
   --istio-helm-module myhelm
   ```

   The `--module` option sets the module type to create, which is `istio`. You define the name of the Istio module using the `--name` option, which in this case is `myistio`.

   The `--istio-helm-module` option sets the name of the Helm module to use to deploy the Istio module. In this case, this is the Helm module named `myhelm`, which is already deployed.

   Note

   The Istio module also requires an instance of Prometheus. When you deploy an Istio module, an embedded instance of Prometheus is created and deployed. You do not need to provide any information for the embedded Prometheus instance.

5. Use the `olcnectl module validate` command to validate the Istio module can be deployed to the nodes. For example, to validate the Istio module named `myistio` in the environment named `myenvironment`:

   ```
   olcnectl module validate \
   --environment-name myenvironment \
   --name myistio
   ```

6. Use the `olcnectl module install` command to deploy the Istio module. For example, to deploy the Istio module named `myistio` in the environment named `myenvironment`:

   ```
   olcnectl module install \
   --environment-name myenvironment \
   --name myistio
   ```

   The Istio software packages are installed on the control plane nodes, and the Istio module is deployed into the Kubernetes cluster.

2.3 Deploying a Service Mesh (Interactive Method)

You can also deploy each module required to create a service mesh in a Kubernetes cluster interactively, being prompted for each required value. As with the other deployment methods, you need to first have an Oracle Cloud Native Environment set up into which you can deploy the service mesh modules.

This example shows you the output when using the `olcnectl module create` command interactively to create the modules required for a service mesh. This example deploys into an existing Kubernetes cluster, using the same module names as the other examples in this chapter.

To deploy a service mesh:

1. Use the `olcnectl module create` command to create the Istio module, and be prompted for all required values:

   ```
   olcnectl module create

   ? Please enter a value for environment-name: myenvironment
   ? Enter the module name: istio
   ? Enter the name of the instance of the istio module myistio
   ? Please select an option for istio-helm-module: New Entry
   ? Please enter a value for istio-helm-module: myhelm
   ? Please select an option for helm-kubernetes-module: mycluster
   Modules created successfully.
   Modules created successfully.
   ```

2. Use the `olcnectl module validate` command to validate the Helm module can be deployed to the nodes. For example, to validate the Helm module named `myhelm` in the environment named `myenvironment`:

   ```
   olcnectl module validate \
   --environment-name myenvironment \
   --name myhelm
   ```

3. Use the `olcnectl module install` command to deploy the Helm module. For example, to deploy the Helm module named `myhelm` in the environment named `myenvironment`:

   ```
   olcnectl module install \
   --environment-name myenvironment \
   --name myhelm
   ```

4. Use the `olcnectl module validate` command to validate the Istio module can be deployed to the nodes. For example, to validate the Istio module named `myistio` in the environment named `myenvironment`:

   ```
   olcnectl module validate \
   --environment-name myenvironment \
   --name myistio
   ```

5. Use the `olcnectl module install` command to deploy the Istio module. For example, to deploy the Istio module named `myistio` in the environment named `myenvironment`:

   ```
   olcnectl module install \
   --environment-name myenvironment \
   --name myistio
   ```

2.4 Verifying the Istio Module Deployment

You can verify the Istio module is deployed and the required containers are running in the `istio-system` namespace. To verify the containers are deployed, you need to use the `kubectl` command. For information on setting up the `kubectl` command, see *Container Orchestration*.

To verify the required containers are running, list the containers running in the `istio-system` namespace. You should see similar results to those shown here:

```
kubectl get deployment -n istio-system

NAME                      READY    UP-TO-DATE    AVAILABLE    AGE
grafana                   2/2      2             2            2m44s
istio-egressgateway       2/2      2             2            2m48s
```

```
istio-ingressgateway   2/2     2            2           2m48s
istiod                 2/2     2            2           3m2s
prometheus             2/2     2            2           2m44s
```

The output above shows Istio deployed into Oracle Cloud Native Environment Release 1.2. If you are using Oracle Cloud Native Environment Release 1.1, your output would look more like the following:

```
kubectl get deployment -n istio-system

NAME                   READY   UP-TO-DATE   AVAILABLE   AGE
grafana                2/2     1            1           5m31s
istio-citadel          2/2     1            1           5m31s
istio-egressgateway    2/2     1            1           5m31s
istio-galley           2/2     1            1           5m31s
istio-ingressgateway   2/2     1            1           5m31s
istio-pilot            2/2     1            1           5m31s
istio-policy           2/2     1            1           5m31s
istio-sidecar-injector 2/2     1            1           5m31s
istio-telemetry        2/2     1            1           5m31s
prometheus             2/2     1            1           5m31s
```

2.5 Removing a Service Mesh

You can remove a deployment of a service mesh and leave the Kubernetes cluster in place. To do this, you remove the Istio module from the environment.

Use the `olcnectl module uninstall` command to remove the Istio module. For example, to uninstall the Istio module named `myistio` in the environment named `myenvironment`:

```
olcnectl module uninstall \
--environment-name myenvironment \
--name myistio
```

The Istio module and embedded Prometheus instance are removed from the environment.

You can confirm the Istio components are removed using the `kubectl` command to query all deployments running in the `istio-system` namespace. You should see there are no deployments returned.

```
kubectl get deployment -n istio-system

No resources found in istio-system namespace.
```

Chapter 3 Using a Service Mesh

Istio automatically populates its service registry with all services you create in the service mesh, so it knows all possible service endpoints. By default, the Envoy proxy sidecars manage traffic by sending requests to each service instance in turn in a round-robin fashion. You can configure the management of this traffic to suit your own application requirements using the Istio traffic management APIs. The APIs are accessed using Kubernetes custom resource definitions (CRDs), which you set up and deploy using YAML files.

The Istio API traffic management features available are:

- **Virtual services**: Configure request routing to services within the service mesh. Each virtual service can contain a series of routing rules, that are evaluated in order.
- **Destination rules**: Configures the destination of routing rules within a virtual service. Destination rules are evaluated and actioned after the virtual service routing rules. For example, routing traffic to a particular version of a service.
- **Gateways**: Configure inbound and outbound traffic for services in the mesh. Gateways are configured as standalone Envoy proxies, running at the edge of the mesh. An ingress and an egress gateway are deployed automatically when you install the Istio module.
- **Service entries**: Configure services outside the service mesh in the Istio service registry. Allows you to manage the traffic to services as if they are in the service mesh. Services in the mesh are automatically added to the service registry, and service entries allow you to bring in outside services.
- **Sidecars**: Configure sidecar proxies to set the ports, protocols and services to which a microservice can connect.

These Istio traffic management APIs are well documented in the upstream documentation at:

https://istio.io/docs/concepts/traffic-management/

3.1 Enabling Proxy Sidecars

Istio enables network communication between services to be abstracted from the services themselves and to instead be handled by proxies. Istio uses a sidecar design, which means that communication proxies run in their own containers alongside every service container.

To enable the use of a service mesh in your Kubernetes applications, you need to enable automatic proxy sidecar injection. This injects proxy sidecar containers into pods you create.

To put automatic sidecar injection into effect, the namespace to be used by an application must be labeled with `istio-injection=enabled`. For example, to enable automatic sidecar injection for the `default` namespace:

```
kubectl label namespace default istio-injection=enabled

namespace/default labeled
```

```
kubectl get namespace -L istio-injection

NAME              STATUS   AGE   ISTIO-INJECTION
default           Active   29h   enabled
istio-system      Active   29h
```

```
kube-node-lease   Active   29h
kube-public       Active   29h
kube-system       Active   29h
```

Any applications deployed into the default namespace have automatic sidecar injection enabled and the sidecar runs alongside the pod. For example, create a simple NGINX deployment:

```
kubectl create deployment --image nginx hello-world

deployment.apps/hello-world created
```

Show the details of the pod to see that an `istio-proxy` container is also deployed with the application:

```
kubectl get pods

NAME                           READY   STATUS    RESTARTS   AGE
hello-world-5fcdb6bc85-wph7h   2/2     Running   0          7m40s
```

```
kubectl describe pods hello-world-5fcdb6bc85-wph7h

...
  Normal  Started    13s   kubelet, worker1.example.com  Started container nginx
  Normal  Started    12s   kubelet, worker1.example.com  Started container istio-proxy
```

3.2 Setting up a Load Balancer for an Ingress Gateway

If you are deploying the Istio module, you may also want to set up a load balancer to handle the Istio ingress gateway traffic. The information in this section shows you how to set up a load balancer to manage access to services from outside the cluster using the Istio ingress gateway.

The load balancer port mapping in this section sets ports for HTTP and HTTPS. That is, the load balancer listens for HTTP traffic on port `80` and redirects it to the Istio ingress gateway NodePort number for `http2`. You query the port number to set for `http2` by entering the following on a control plane node:

```
kubectl describe svc istio-ingressgateway -n istio-system |grep http2

Port:                     http2  80/TCP
NodePort:                 http2  32681/TCP
```

In this example, the NodePort is `32681`. So the load balancer must be configured to listen for HTTP traffic on port `80` and redirect it to the `istio-ingressgateway` service on port `32681`.

For HTTPS traffic, the load balancer listens on port `443` and redirects it to the Istio ingress gateway NodePort number for `https`. To find the port numbers to set for `https`, enter:

```
kubectl describe svc istio-ingressgateway -n istio-system |grep https

Port:                     https  443/TCP
NodePort:                 https  31941/TCP
```

In this example, the NodePort is `31941`. So the load balancer must be configured to listen for HTTPS traffic on port `443` and redirect it to the `istio-ingressgateway` service on port `31941`.

The load balancer should be set up with the following configuration for HTTP traffic:

- The listener listening on TCP port `80`.
- The distribution set to round robin.

- The target set to the TCP port for `http2` on the worker nodes. In this example it is `32681`.
- The health check set to TCP.

For HTTPS traffic:

- The listener listening on TCP port `443`.
- The distribution set to round robin.
- The target set to the TCP port for `https` on the worker nodes. In this example it is `31941`.
- The health check set to TCP.

For more information on setting up your own load balancer, see the *Oracle® Linux 7: Administrator's Guide*, or *Oracle® Linux 8: Setting Up Load Balancing*.

If you are deploying to Oracle Cloud Infrastructure, you can either set up a new load balancer or, if you have one, use the load balancer you set up for the Kubernetes module.

To set up a load balancer on Oracle Cloud Infrastructure for HTTP traffic:

1. Add a backend set to the load balancer using weighted round robin.
2. Add the worker nodes to the backend set. Set the port for the worker nodes to the TCP port for `http2`. In this example it is `32681`.
3. Create a listener for the backend set using TCP port `80`.

To set up a load balancer on Oracle Cloud Infrastructure for HTTPS traffic:

1. Add a backend set to the load balancer using weighted round robin.
2. Add the worker nodes to the backend set. Set the port for the worker nodes to the TCP port for `https`. In this example it is `31941`.
3. Create a listener for the backend set using TCP port `443`.

For more information on setting up a load balancer in Oracle Cloud Infrastructure, see the Oracle Cloud Infrastructure documentation.

3.3 Setting up an Ingress Gateway

An Istio ingress gateway allows you to define entry points into the service mesh through which all incoming traffic flows. A ingress gateway allows you to manage access to services from outside the cluster. You can monitor and set route rules for the traffic entering the cluster.

This section contains a simple example to configure the automatically created ingress gateway to an NGINX web server application. The example assumes you have a load balancer available at `lb.example.com` and is connecting to the `istio-ingressgateway` service on `TCP` port `32681`. The load balancer listener is set to listen on `HTTP` port `80`, which is the port for the NGINX web server application used in the virtual service in this example.

To set up an ingress gateway:

1. Create the deployment file to create the NGINX webserver application. Create a file named `my-nginx.yml`, containing:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: my-webserver
  name: my-nginx
  namespace: my-namespace
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-webserver
  template:
    metadata:
      labels:
        app: my-webserver
    spec:
      containers:
      - image: nginx
        name: my-nginx
        ports:
        - containerPort: 80
```

2. Create a service for the deployment. Create a file named `my-nginx-service.yml` containing:

```
apiVersion: v1
kind: Service
metadata:
  name: my-http-ingress-service
  namespace: my-namespace
spec:
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: my-webserver
  type: ClusterIP
```

3. Create an ingress gateway for the service. Create a file named `my-nginx-gateway.yml` containing:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: my-nginx-gateway
  namespace: my-namespace
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
      - "mynginx.example.com"
```

4. Create a virtual service for the ingress gateway. Create a file named `my-nginx-virtualservice.yml` containing:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: my-nginx-virtualservice
  namespace: my-namespace
spec:
  hosts:
  - "mynginx.example.com"
  gateways:
  - my-nginx-gateway
  http:
  - match:
    - uri:
        prefix: /
    route:
    - destination:
        port:
          number: 80
        host: my-http-ingress-service
```

5. Set up a namespace for the application named `my-namespace` and enable automatic proxy sidecar injection.

```
kubectl create namespace my-namespace
kubectl label namespaces my-namespace istio-injection=enabled
```

6. Run the deployment, service, ingress gateway and virtual service:

```
kubectl apply -f my-nginx.yml
kubectl apply -f my-nginx-service.yml
kubectl apply -f my-nginx-gateway.yml
kubectl apply -f my-nginx-virtualservice.yml
```

7. You can see the ingress gateway is running using:

```
kubectl get gateways.networking.istio.io -n my-namespace

NAME               AGE
my-nginx-gateway   33s
```

8. You can see the virtual service is running using:

```
kubectl get virtualservices.networking.istio.io -n my-namespace

NAME                       GATEWAYS             HOSTS                    AGE
my-nginx-virtualservice    [my-nginx-gateway]   [mynginx.example.com]    107s
```

9. To confirm the ingress gateway is serving the application to the load balancer, use:

```
curl -I -HHost:mynginx.example.com lb.example.com:80/

HTTP/1.1 200 OK
Date: Fri, 06 Mar 2020 00:39:16 GMT
Content-Type: text/html
Content-Length: 612
Connection: keep-alive
last-modified: Tue, 03 Mar 2020 14:32:47 GMT
etag: "5e5e6a8f-264"
accept-ranges: bytes
x-envoy-upstream-service-time: 15
```

3.4 Setting up an Egress Gateway

The Istio egress gateway allows you to set up access to external HTTP and HTTPS services from applications inside the service mesh. External services are called using the sidecar container.

The Istio egress gateway is deployed automatically. You do not need to manually deploy it. You can confirm the Istio egress gateway service is running using:

```
kubectl get svc istio-egressgateway -n istio-system

NAME                  TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                   AGE
istio-egressgateway   ClusterIP   10.111.233.121   <none>        80/TCP,443/TCP,15443/TCP   9m26s
```

The upstream documentation provides an example to show you how to set up use an Istio egress gateway.

https://istio.io/docs/tasks/traffic-management/egress/egress-gateway/

3.5 Testing Network Resilience

Istio network resilience and testing features allow you to set up and test failure recovery and to inject faults to test resilience. You set up these features dynamically at runtime to improve the reliability of your applications in the service mesh. The network resilience and testing features available in this release are:

- **Timeouts**: The amount of time that a sidecar proxy should wait for replies from a service. You can set up a virtual service to configure specific timeouts for a service. The default timeout for HTTP requests is 15 seconds.
- **Retries**: The number of retries allowed by the sidecar proxy to connect to a service after an initial connection failure. You can set up a virtual service to enable and configure the number of retries for a service. By default, no retries are allowed.
- **Fault injection**: Set up fault injection mechanisms to test failure recovery of applications. You can set up a virtual service to set up and inject faults into a service. You can set delays to mimic network latency or an overloaded upstream service. You can also set aborts to mimic crashes in an upstream service.