Certified Kubernetes Administrator

Services, Load Balancing, and Networking

Learning Objectives

By the end of this lesson, you will be able to:

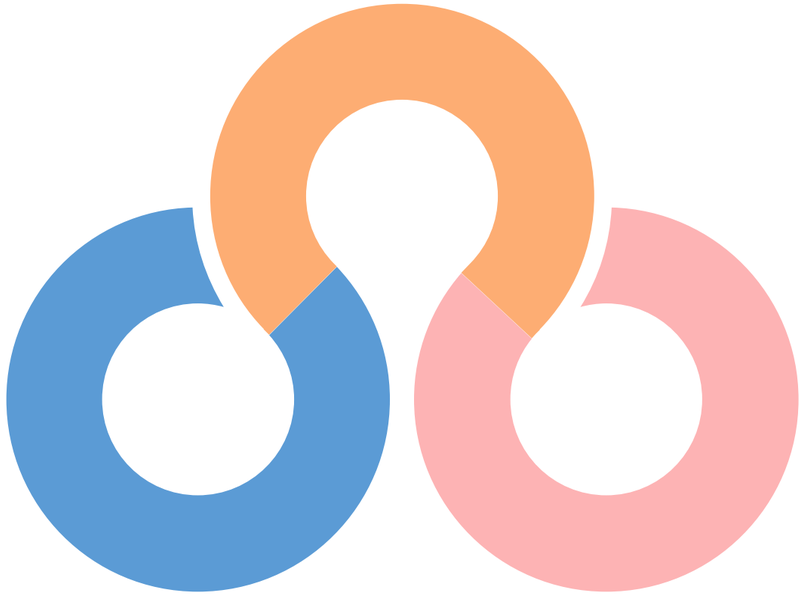
- State the concerns that Kubernetes networking addresses
- Describe topology and DNS
- Present an overview of EndpointSlices, Ingress, and ingress controllers
- Define network policies and IPv4_IPv6 dual-stack
- List the ways in which cluster networking is implemented

Overview

Services

Kubernetes Service is a logical abstraction for a deployed group of Pods in a cluster.

Service enables the group of Pods.



Each Pod will be assigned a name and a unique IP address.

IP address of the Pod cannot be changed when a Service is running on it.

Networking

Kubernetes networking addresses four concerns.

Loopback is used by the Containers within a Pod for communication.

Communication between Pods happens through cluster networking.

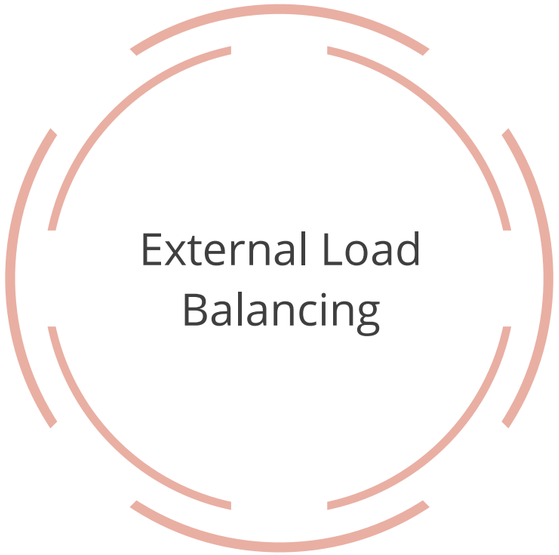
An application running in a Pod can be made reachable from outside the cluster using the Service resource.

Services can be published only for consumption within the cluster.

Load Balancing

Load Balancing helps in distributing network traffic or client's request to multiple servers.

Load balancing can be:



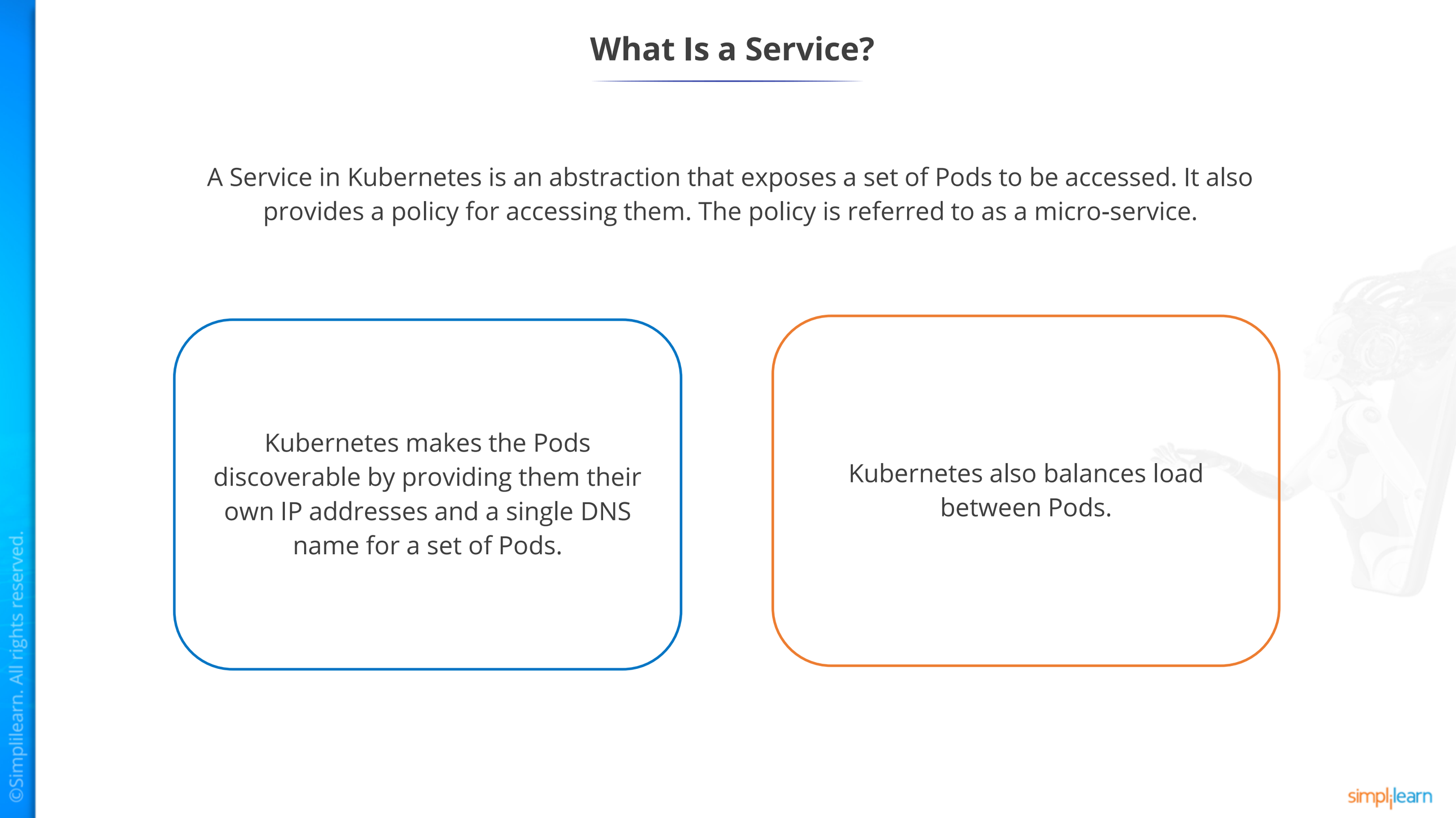
Services

What Is a Service?

A Service in Kubernetes is an abstraction that exposes a set of Pods to be accessed. It also provides a policy for accessing them. The policy is referred to as a micro-service.

Kubernetes makes the Pods discoverable by providing them their own IP addresses and a single DNS name for a set of Pods.

Kubernetes also balances load between Pods.



Define a Service

A Service is a REST object. To create a new instance of the Service, the Service definition can be POSTed to the API server.

The YAML file shown below exposes a set of Pods that listen on TCP port 9376 and contain a label app, **MyApp.** This specification creates a new Service object, named **my-service,** which targets TCP port 9376 on any Pod with the app, **MyApp** label.

Demo

```
apiVersion: v1
Kind:    service
Metadata:
   name:  my-service
spec:
   selector:
      app:   MyApp
   ports:
      - protocol: TCP
        port: 80
        targetPort: 9376
```

Service

1

The default protocol for Service is TCP. However, UDP, SCTP, HTTP, and PROXY are also supported.

2

Kubernetes supports multiple port definitions on a Service object.

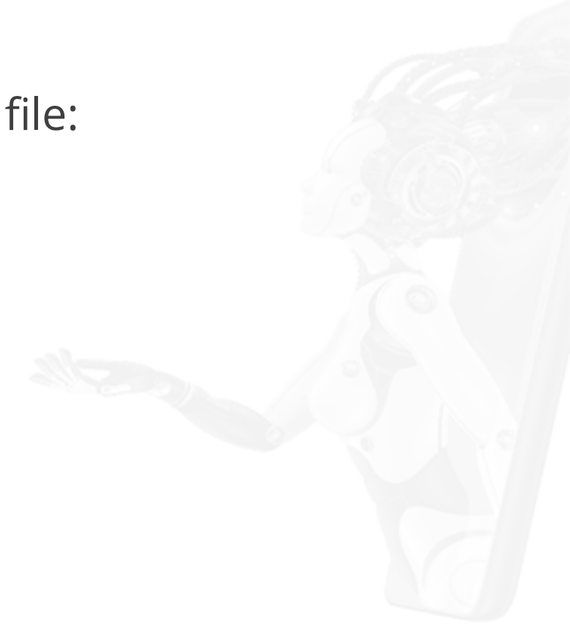
Service Without Selectors

Services mostly abstract access to Kubernetes Pods though they can also abstract other kinds of backends

To define a Service without a Pod Selector, add the following in the YAML file:

Demo

```
apiVersion: v1
Kind:  service
Metadata:
   name:  my-service
spec:
     ports:
      - protocol: TCP
        port: 80
        targetPort: 9376
```



Service without Selectors

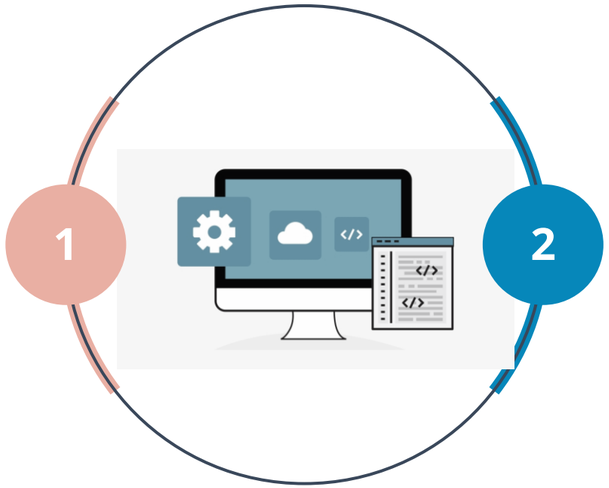
A Service can be mapped manually to the network address and the port where it is running by adding an Endpoints object.

Demo

```
apiVersion: v1
Kind:  Endpoints
Metadata:
   name:  my-service
sybsets:
   - addresses:
        -ip: 192.0.2.42
     ports:
         - port: 9376
```

Application Protocol

The appProtocol field provides a way to specify an Application Protocol for each Service port.



It follows standard Kubernetes Label Syntax.

Values should either be IANA standard service names or domain prefixed names.

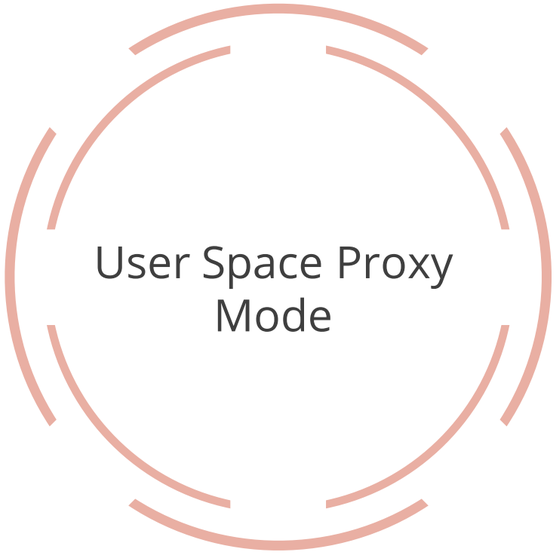
Virtual IPs and Service Proxies

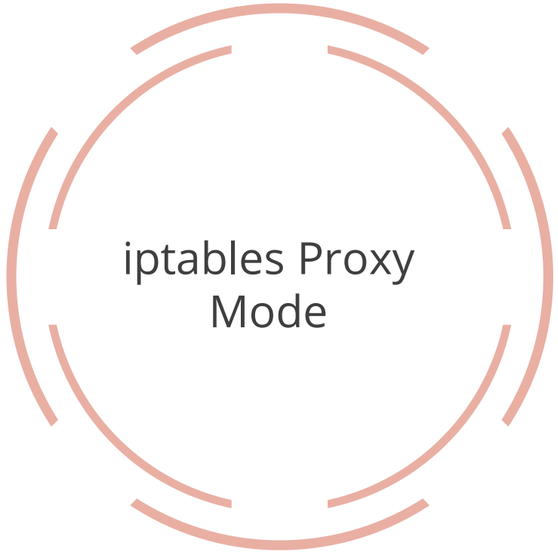
1. Every Node in a Kubernetes cluster runs a kube-proxy.

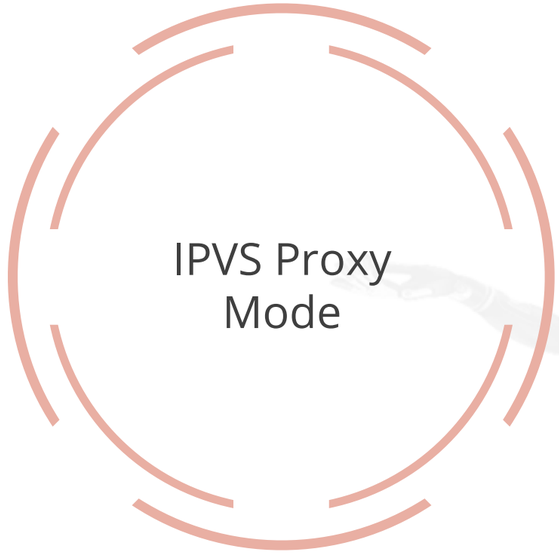
2. kube-proxy is responsible for implementing a form of virtual IP for Services of type other than External Name.

Proxy Modes

Three types of proxy modes are supported in Kubernetes:

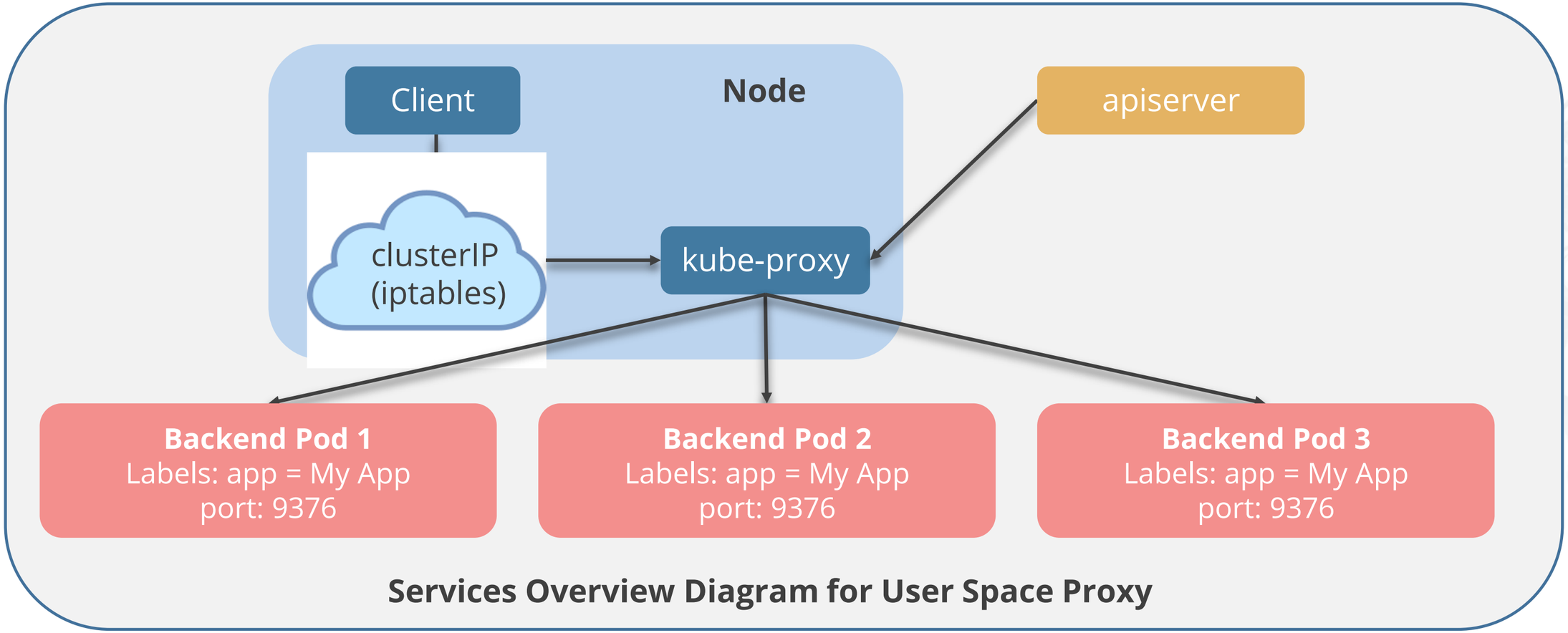






User Space Proxy Mode

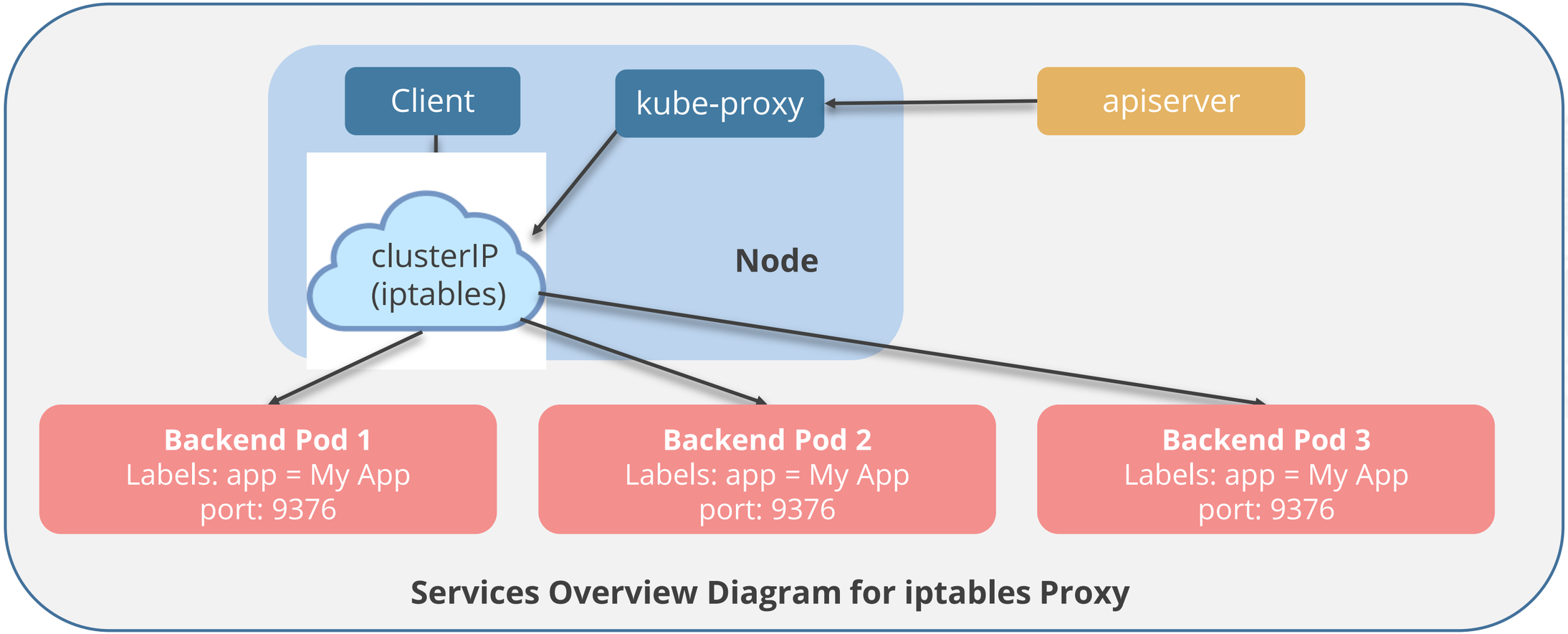
In this mode, kube-proxy watches the Kubernetes Control Plane for the addition and removal of Service and Endpoint objects. It chooses a backend via a round-robin algorithm.



Services Overview Diagram for User Space Proxy

iptables Proxy Mode

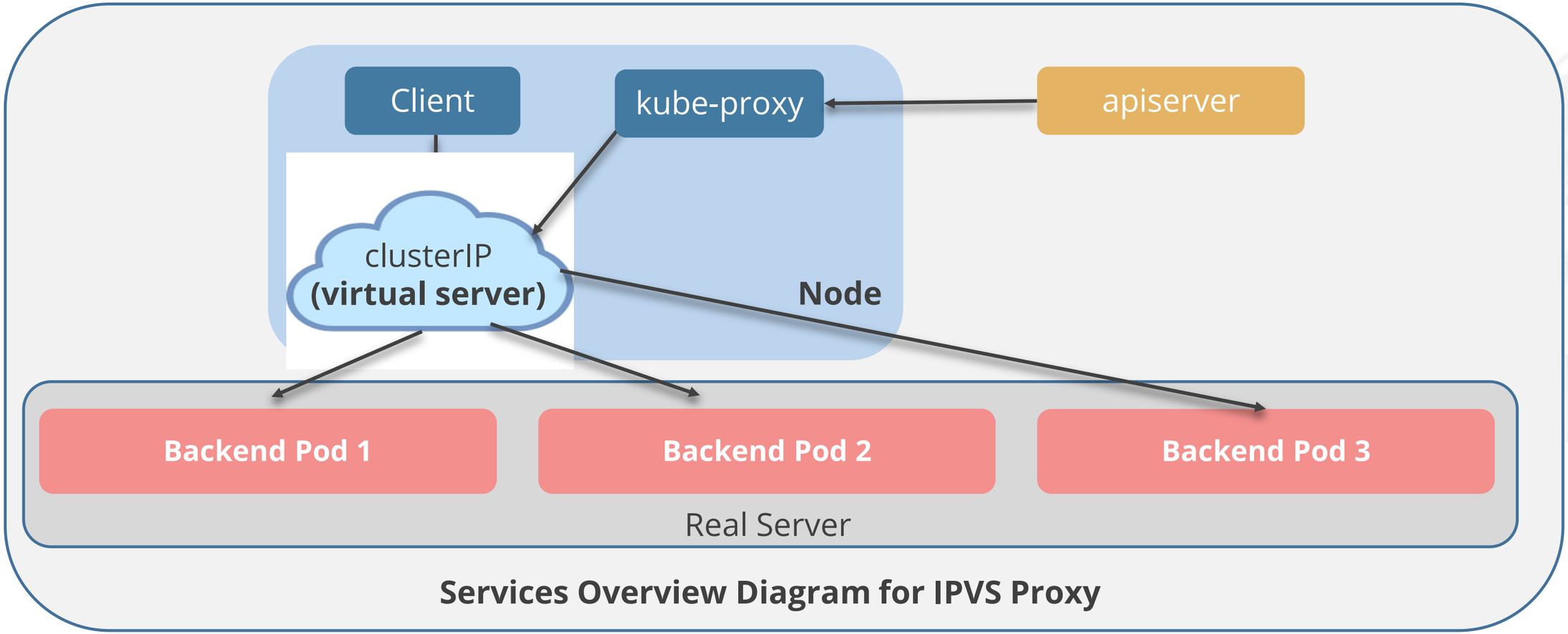
In this mode, kube-proxy watches the Kubernetes Control Plane for the addition and removal of Service and Endpoint objects. By default, in the iptables mode, kube-proxy chooses a backend at random.



Services Overview Diagram for iptables Proxy

IPVS Proxy Mode

In IPVS proxy mode, kube-proxy watches Kubernetes Services and Endpoints and calls net link interface to create IPVS rules accordingly. It synchronizes IPVS rules with Kubernetes Services and Endpoints periodically.



Services Overview Diagram for IPVS Proxy

Multi-Port Services

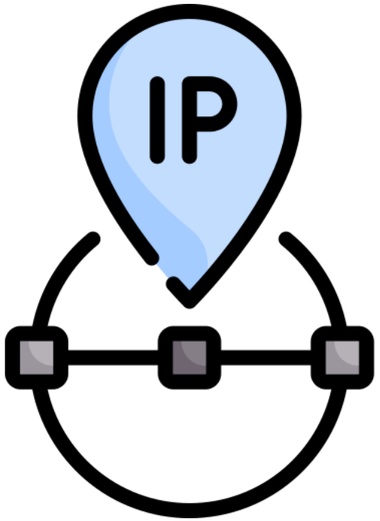
Kubernetes helps to configure multiple port definitions on a Service object. The configuration below shows the use of the **port** attribute:

Demo

```
apiVersion: v1
Kind:  service
Metadata:
   name:  my-service
spec:
  selector:
    app: MyApp
   ports:
      - name: http
        protocol: TCP
        port: 80
      - name: http
        protocol: TCP
        port: 443
        targetPort: 9377
```

Choose IP Addresses

Cluster IP address can be specified as part of a Service creation request.

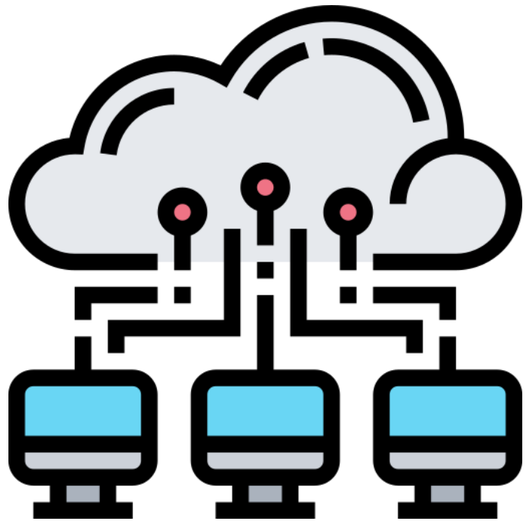


Note

The chosen IP address must be a valid IPv4 or IPv6 address from within the service-cluster-ip-range CIDR range that is configured for the API server.

Discover Services

Services can be discovered using environment variables and DNS.



Environment Variables

The Kubelet adds a set of environment variables for each active Service when a Pod is run on a Node. The example below shows the environment variables for the Service redis-master, which exposes TCP port 6379 and has been allocated cluster IP address 10.0.0.11.

```
REDIS_MASTER_SERVICE_HOST = 10.0.0.11
REDIS_MASTER_SERVICE_PORT= 6379
REDIS_MASTER_PORT= tcp://10.0.011:6379
REDIS_MASTER_PORT_6379_TCP = tcp://10.0.011:6379
REDIS_MASTER_PORT_6379_TCP_PROTO= tcp
REDIS_MASTER_PORT_6370_TCP_PORT= 6379
REDIS MASTER_PORT_6379_TCP_ADDP= 10.0.0.11
```

DNS

A cluster-aware DNS server, such as CoreDNS, watches the Kubernetes API for new Services and creates a set of DNS records for each. The admin must almost always set up a DNS service for the Kubernetes cluster using an add-on. The Kubernetes DNS server is the only way to access ExternalName Services.

Example:

For a Service called **my-service** in a Kubernetes namespace **my-ns**, the Control Plane and the DNS Service together create a DNS record for **my-service.my-ns**.

Pods in the **my-ns** namespace should be able to find the service by doing a name lookup for my-service.

Pods in other namespaces must qualify the name as **my-service.my-ns**.

Headless Service

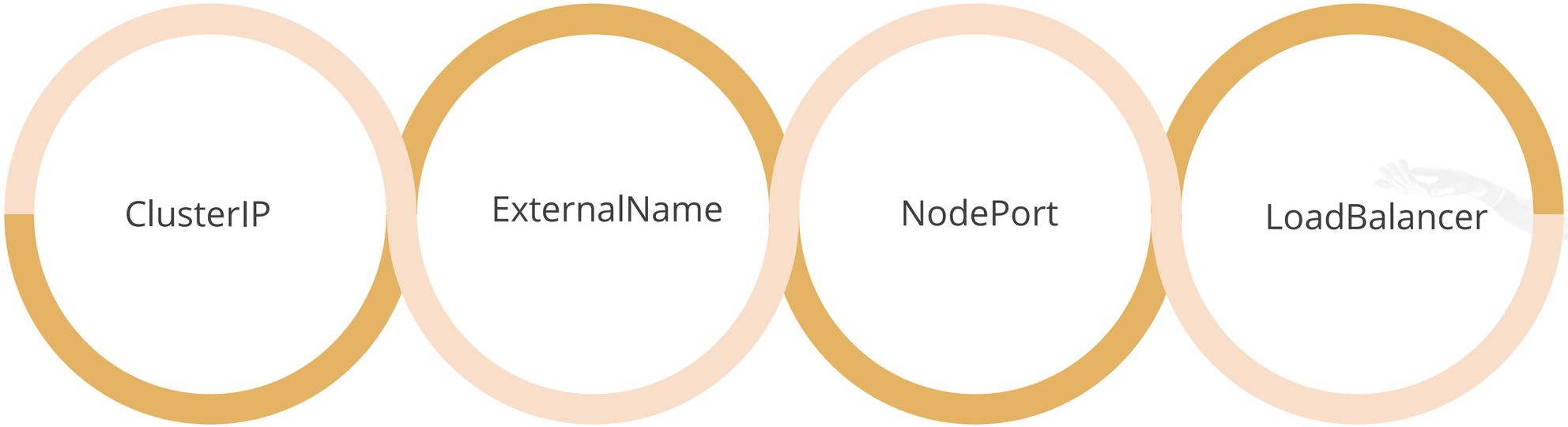
A Headless Service is a service that has no Service IP. They are used to interface with other service discovery mechanisms, without being tied to the implementation of Kubernetes.

Headless Services may be classified into two categories:

Publishing Services (ServiceTypes)

Kubernetes ServiceTypes help to specify the required kind of service.

Type values and their behaviors include:



Type NodePort

The **NodePort** Service Type allows setting up a custom Load Balancing solution for configuring environments that are not fully supported by Kubernetes. The **NodePort** type can be configured as shown below:

Demo

```
    Apiversion: v1
    Kind:  service
    Metadata:
       name:  my-service
    spec:
       type: NodePort
       selector:
        app: MyApp
       ports:
           # By default, and for convenience , the 'targetPort' is
set to the same value as the ' port' field.
         - port: 80
           targetPort: 80
           # Optional field
           # By default, and for convenience, the kubernetes control
plane will allocate a port from a range ( default: 30000-32767)
            NodePort:  30007
```

Type LoadBalancer

The LoadBalancer type provisions Load Balancers for cloud providers that support external Load Balancers. Load Balancers are created asynchronously. **status.loadBalancer** field contains the provisioned balancer's information.

Demo

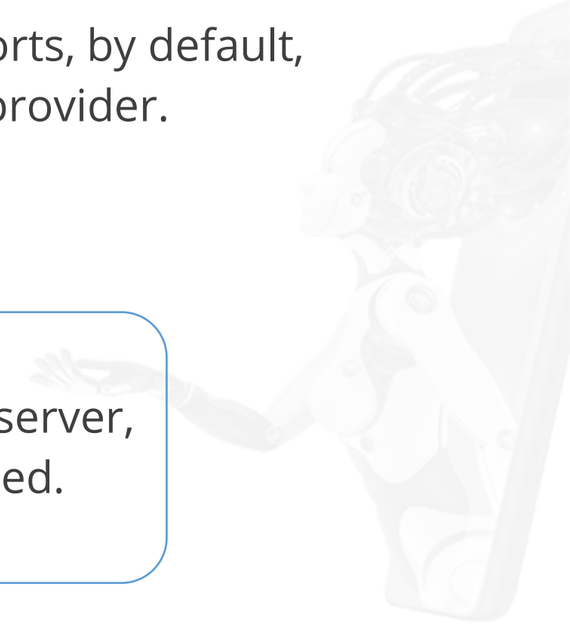
```
apiVersion: v1
Kind:  service
Metadata:
   name:  my-service
spec:
  selector:
    app: MyApp
  ports:
      - protocol: TCP
        port: 80
        targetPort: 9376
  clusterIP: 10.0.171.239
  type: LoadBalancer
status:
     LoadBalancer: 9377
        ingress:
           -ip: 192.0.2.127
```

To direct traffic from the external Load Balancer to the backend Pods, use the configuration shown.

Load Balancers with Mixed Protocol Types

For **LoadBalancer** type of Services, when there is more than one port defined, all ports, by default, must have the same protocol. The protocol must be supported by the cloud provider.

If the feature gate **MixedProtocolLBService** is enabled for the kube-apiserver, it can use different protocols when there is more than one port defined.



Disable Load Balancer NodePort Allocation

Starting in v1.20, Node port allocation or a **Service Type=LoadBalancer** can be optionally disabled. This can be done by setting the field **spec.allocateLoadBalancerNodePorts** to false. It can only be used for Load Balancer implementations that route traffic directly to Pods.

By default, **spec.allocateLoadBalancerNodePorts** is true and type LoadBalancer Services will continue to allocate Node ports.

If **spec.allocateLoadBalancerNodePorts** is set to false on an existing Service with allocated Node ports, these Node ports will not be de-allocated automatically.

ServiceLBNodePortControl feature gate must be enabled to use this field.

Topology

Topology-Aware Traffic Routing

Service Topology enables a Service to route traffic, based on the Node topology of the cluster.

By default, traffic sent to a **ClusterIP** or **NodePort** Service can be routed to any backend address for the Service.

The label matching between the source and destination lets the cluster operator designate sets of Nodes that are closer and farther from one another.

Usage of Service Topology

Service traffic routing can be controlled if the ServiceTopology feature of the cluster is enabled. This can be done by specifying the **topologyKeys** field on the service spec, a preference-order list of Node labels.

Traffic will be directed to the Node whose first label value matches the originating Node's value.

The second label will be considered if there is no backend for the Service on a matching Node. It will move until no labels remain.

Topology constraints will not be applied if topologyKeys is empty or unspecified.

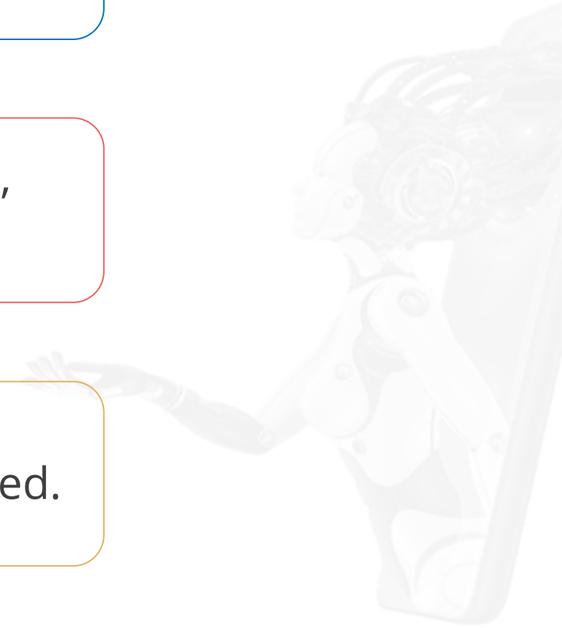
Constraints

1. Service topology is not compatible with **externalTrafficPolicy=Local**, and therefore, a Service cannot use both these features.

2. Valid Topology keys are currently limited to kubernetes.io/hostname, topology.kubernetes.io/zone, and topology.kubernetes.io/region.

3. Topology keys must be valid label keys; at most, 16 keys may be specified.

4. If the **catch-all value *** is used, it must be the last value in the Topology keys.



Understanding Service Topology

Problem Statement: Understand the working of service topology in Kubernetes.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to demonstrate topology in Kubernetes:

1. Route through local Endpoints only
2. Route through local Endpoints as a preference

Service Catalog

Overview

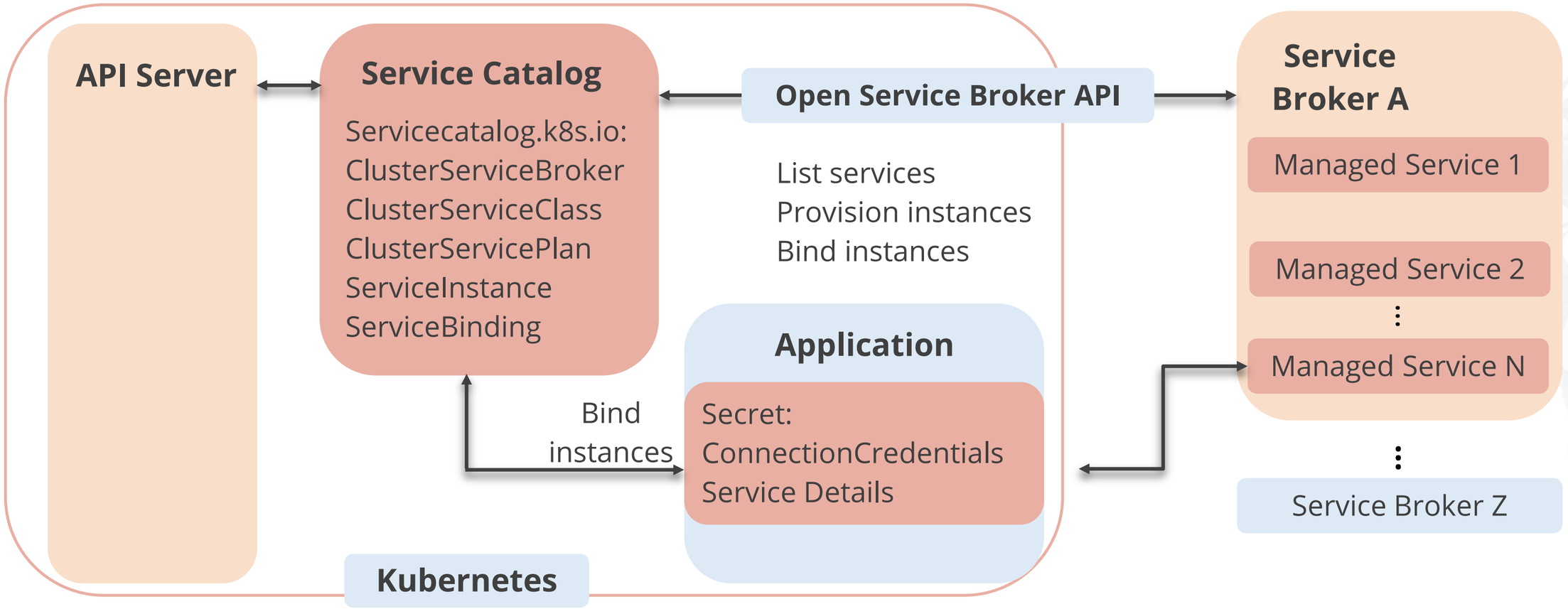
Service Catalog is an extension API that enables applications running in Kubernetes clusters to easily use externally managed software offerings.

Service Catalog helps to list, provision, and bind with external Managed Services from Service Brokers by abstracting service creation and management.

It acts as an intermediary for the Kubernetes API Server to negotiate the initial provisioning and retrieve the credentials necessary for the application to use a Managed Service.

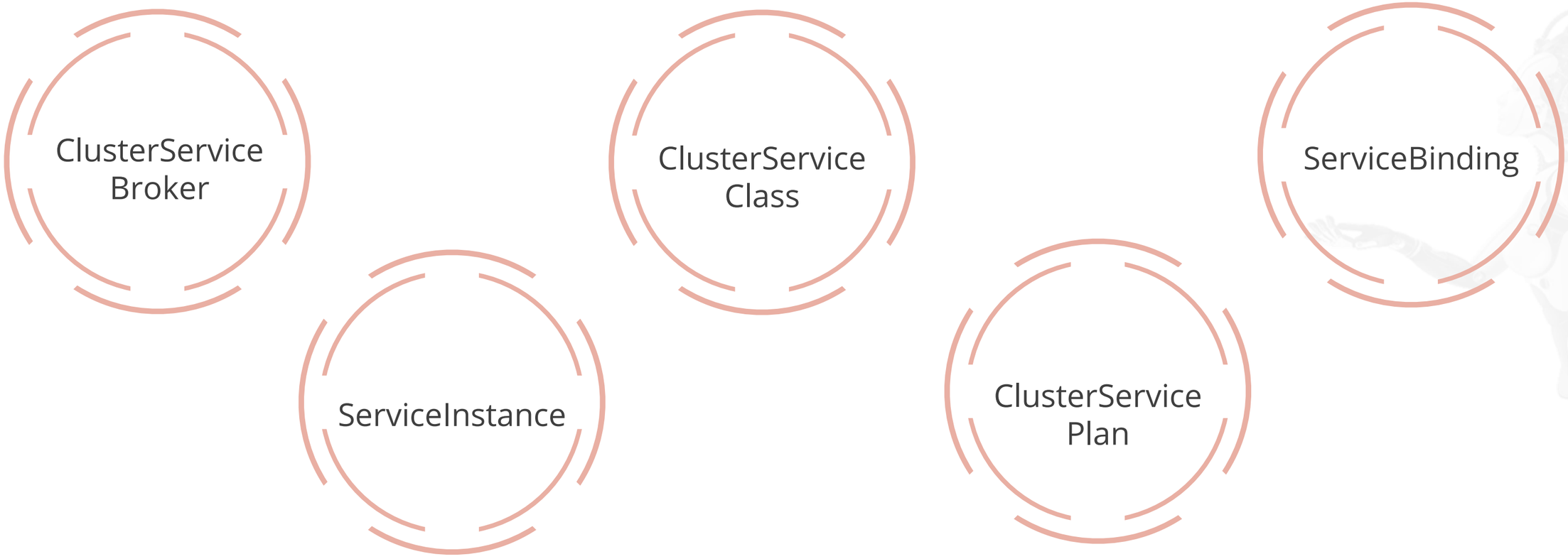
Architecture

Service Catalog uses the Open Service Broker API to communicate with Service Brokers.



API Resources

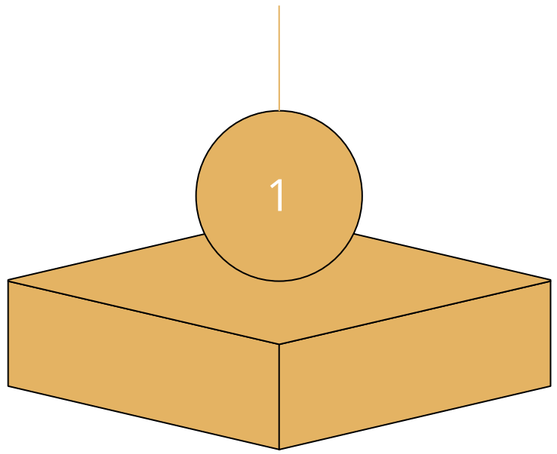
Service Catalog installs the servicecatalog.k8s.io API and provides the following Kubernetes resources:



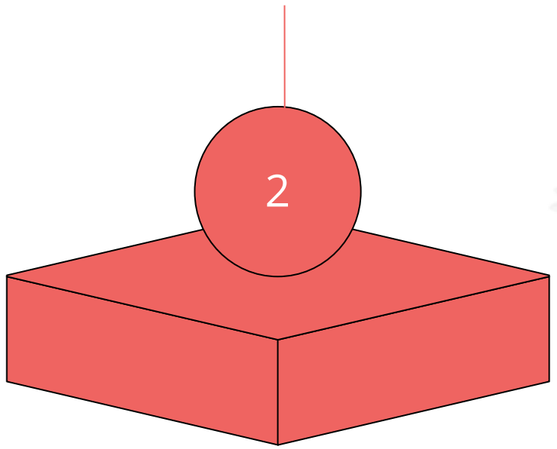
Authentication

Service Catalog supports two methods of authentication:

Basic (username/password)

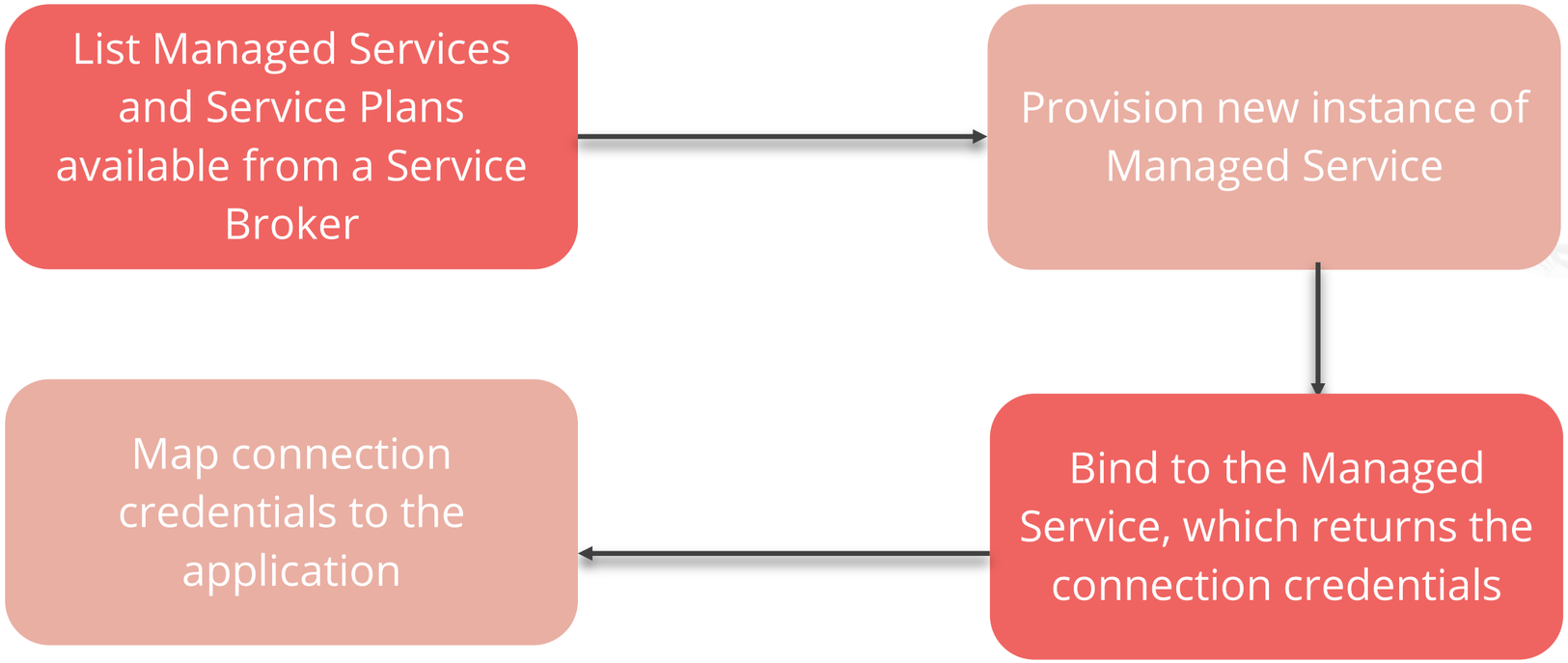


OAuth 2.0 Bearer Token



Usage

Service Catalog API Resources are used to provision Managed Services and make them available within a Kubernetes cluster, employing a four-step process:



List Managed Services and Service Plans

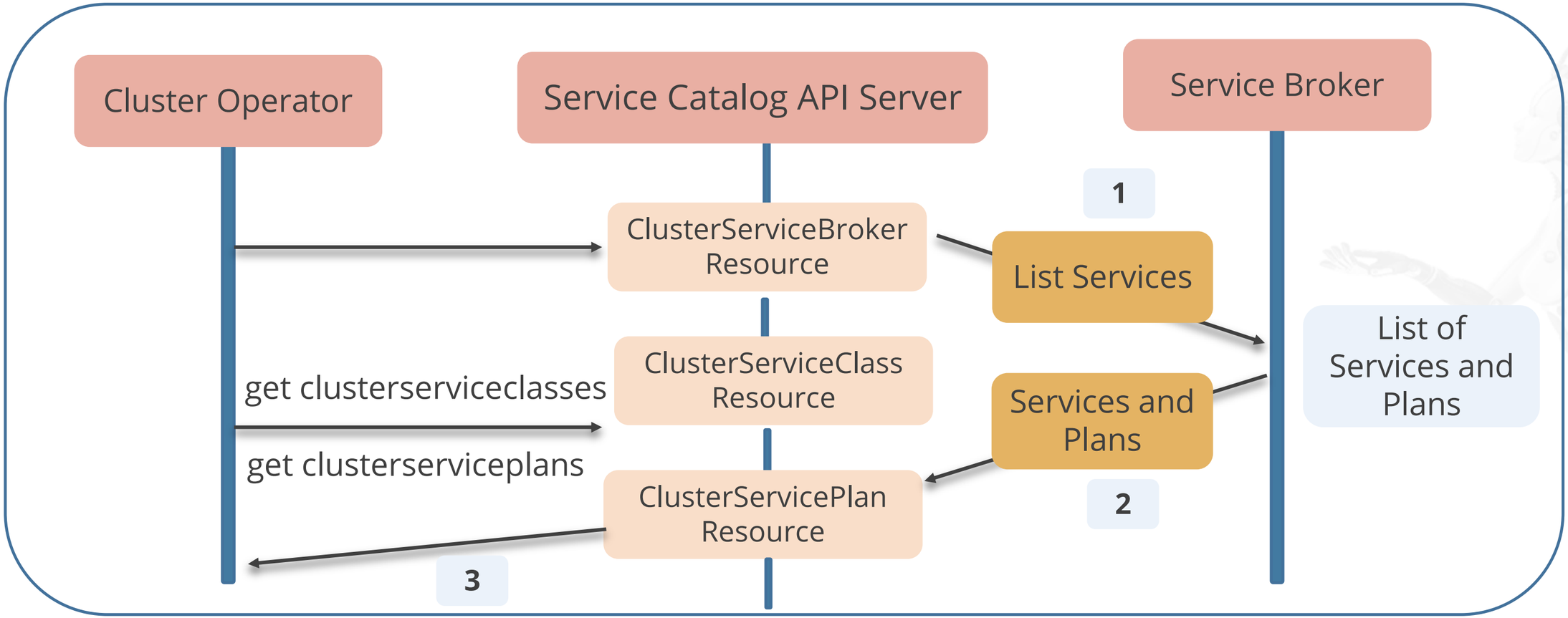
A ClusterServiceBroker resource is created within the servicecatalog.k8s.io group. This resource contains the URL and connection details necessary to access a Service Broker Endpoint. Shown below is an example of a ClusterServiceBroker resource:

Demo

```
    apiVersion: servicecatalog.k8s.io/v1beta1
    Kind:  ClusterServiceBroker
    Metadata:
       name:  cloud-broker
    spec:
#Points to the endpoint of a service broker. ( This example is not a
working URL.)
        URL:
https://servicebroker.somecloudprovider.com/v1a1pha1/projects/service-
catalog/brokers/default
      ####
      #Additional values can be added here, which may be used to
communicae
#with the service broker, such as bearer token info or a caBundle for
TLS
            ####
```

List Managed Services and Service Plans

Here is a sequence diagram illustrating the steps involved in listing Managed Services and Plans available from a Service Broker:



Provision a New Instance

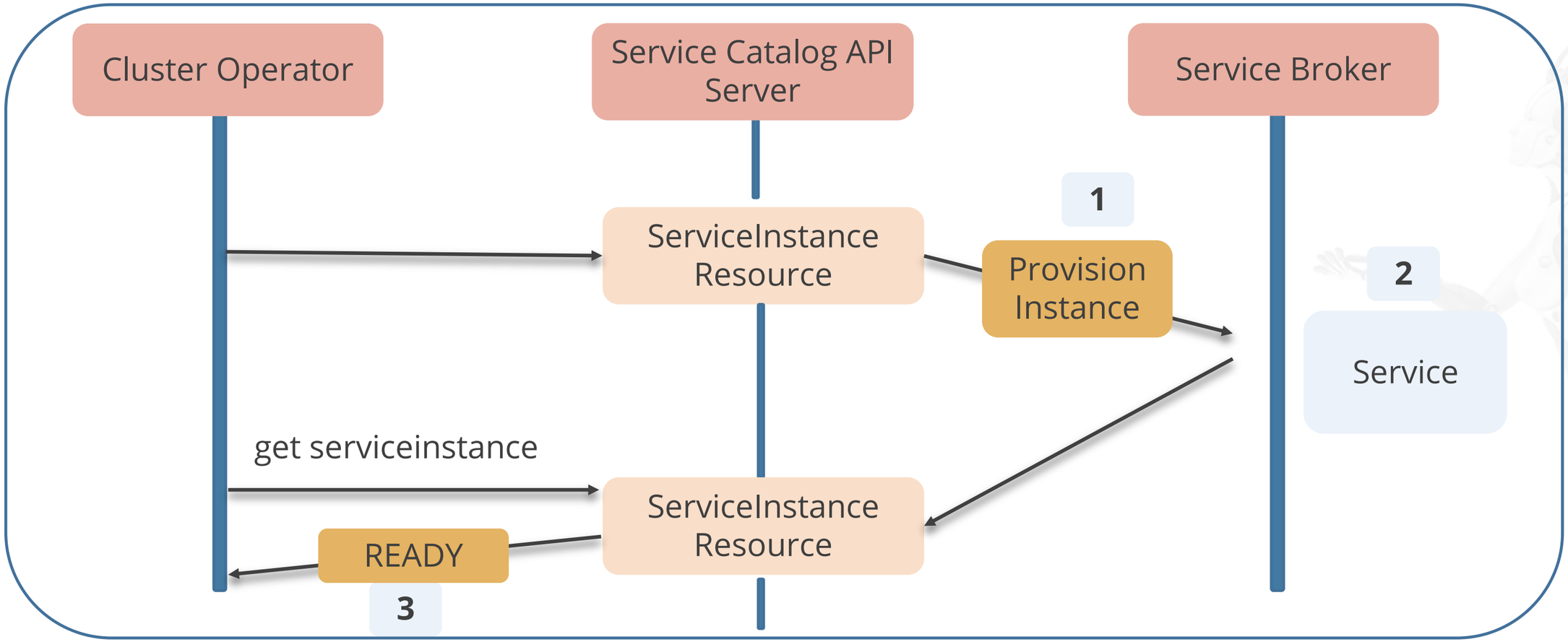
Cluster operator can initiate the provisioning of a new instance by creating a ServiceInstance resource. Given below is an example:

Demo

```
apiVersion: servicecatalog.k8s.io/v1beta1
Kind:  ServiceInstance
Metadata:
   name:  cloud-queue-instance
   namespace: cloud-apps
spec:
   # Reference one of the previously returned services
   clusterServiceClassExternalName: cloud-provider-name
   clusterServicePlanExtenalName: service-plan-name
   ####
   # Additional parameters can be added here
   # which may be used by the service broker
   ####
```

Provision a New Instance

The sequence diagram below illustrates the steps involved in provisioning a new instance of Managed Service.



Bind to a Managed Service

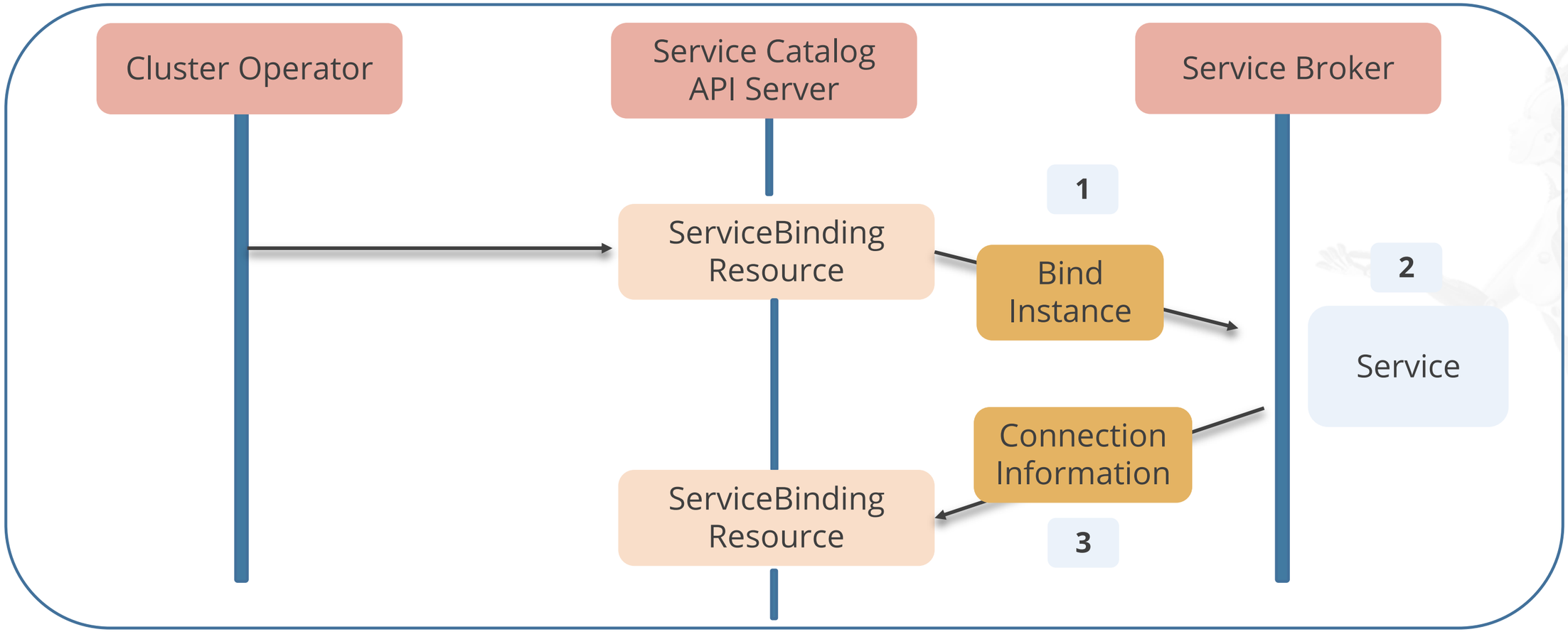
A cluster operator must bind to the Managed Service to get the connection credentials and service account details necessary for the application to use the service. An example of a **ServiceBinding** resource is shown below:

Demo

```
    apiVersion: servicecatalog.k8s.io/v1beta1
    Kind:  ServiceBinding
    Metadata:
       name:  cloud-queue-binding
       namespace: cloud-apps
    spec:
       instanceRef:
          name: cloud-provider-name
       ####
       # Additional information can be added here, such as a secretNmae or
       # service account parameters, which may be used by the service
broker.
       ####
```

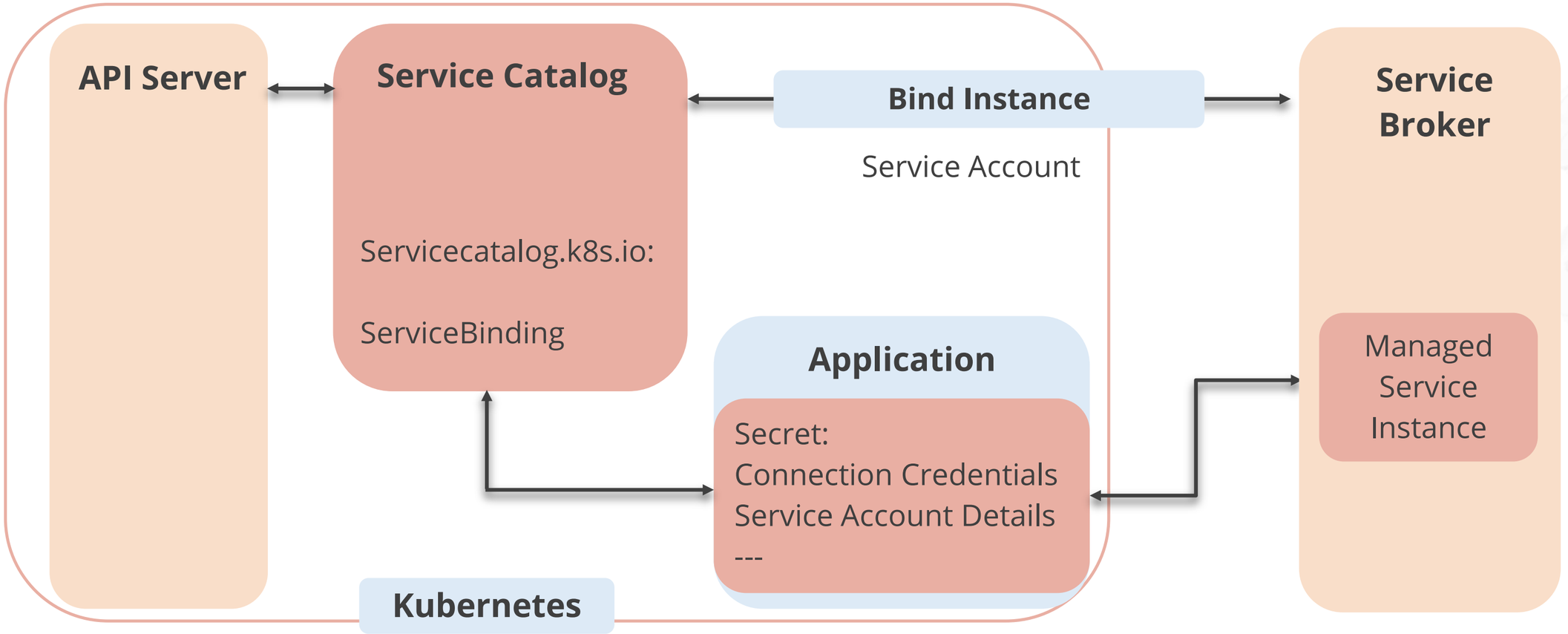
Bind to a Managed Service

The sequence diagram given below illustrates the steps involved in binding to a Managed Service instance.



Map the Connection Credentials

The final step involves mapping the connection credentials and service-specific information into the application.



Pod Configuration File

Declarative Pod configuration can be used to perform mapping. The example here shows how to map service account credentials into the application.

Demo

```
...
   spec:
     volumes:
       - name:  provoder-cloud-key
         secret:
           secretNmae: sa-key
     containers:
...
       volumeMounts:
        - name: provider-cloud-key
          mountpath: /var/secrets/provider
       env:
        - name: PROVIDER_APPLICATION_CREDENTIALS
          value: "/var/secrets/provider/key.json"
```

Pod Configuration File

The example below describes how to map secret values into application environment variables:

Demo

```
    ...
          env:
           - name: “TOPIC”
             valueFrom:
                 secretKeyRef:
                    name: provider-queue-credentials
                    key: topic
”
```

Understanding Service Catalog

Problem Statement: Understand the working of Service Catalog in Kubernetes.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to demonstrate Service Catalog in Kubernetes:

1. Install service catalog through helm
2. Add service catalog helm repository
3. Check whether it is installed
4. Check whether RBAC is enabled
5. Install service catalog into the cluster

TECHNOLOGY

DNS for Services and Pods

simplilearn

Introduction

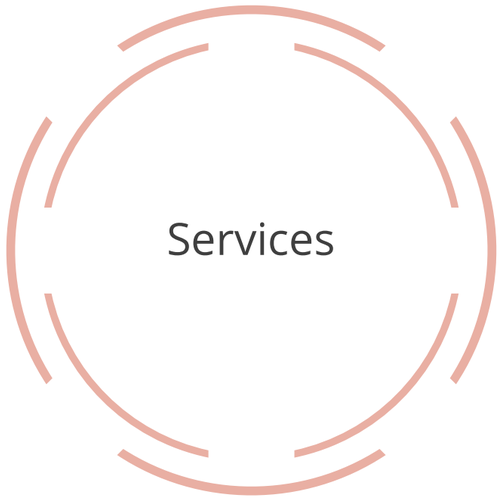
Kubernetes creates DNS records for Services and Pods. Kubernetes DNS schedules a DNS Pod and Service on the cluster and configures the Kubelets to tell individual Containers to use the DNS Service's IP to resolve DNS names.

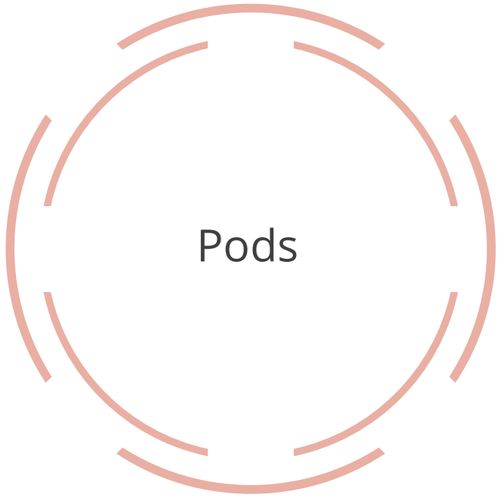
A DNS query may return different results based on the namespace of the Pod making it.

DNS queries may be expanded using the Pod's /etc/resolv.conf file.

DNS Records

The following objects get DNS records:





Services

Snapshots can be provisioned in two ways:

1

Normal Services are assigned a **DNS A or AAAA record**, depending on the IP family of the service, for a name of the form my-svc.my-namespace.svc.cluster-domain.example.

2

SRV Records are created for named ports that are part of normal or Headless Services.

Pods

A Pod has the following DNS resolution:

Pod-ip-address.my-namespace.Pod.cluster-domain.example

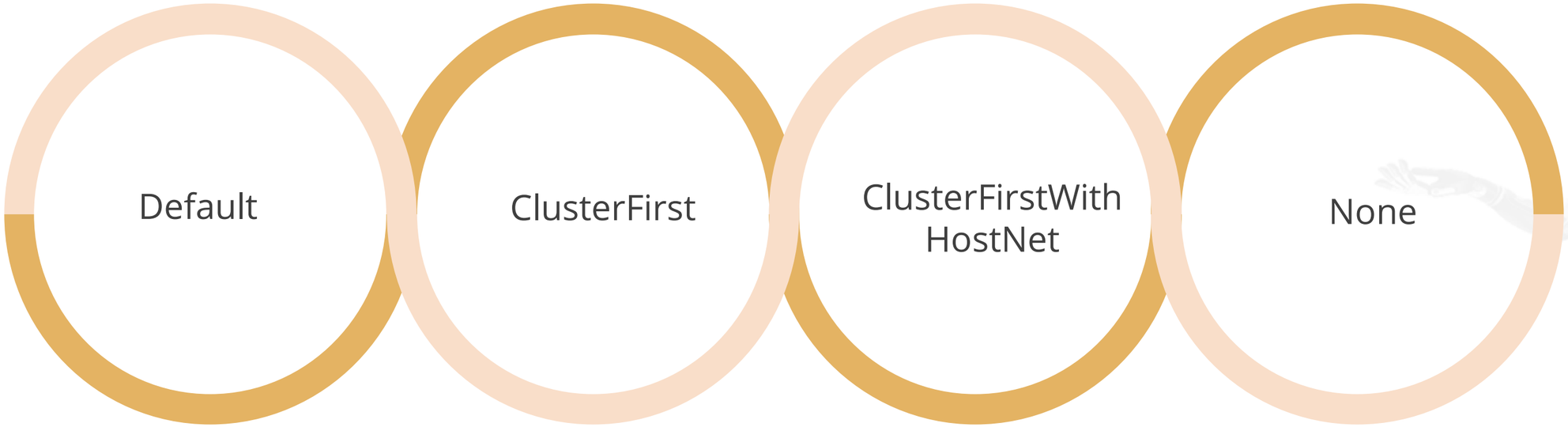
The Pod spec has an optional hostname field, which can be used to specify the Pod's hostname.

It also has an optional subdomain field that can be used to specify its subdomain.

Pod's DNS Policy

DNS policies can be set on a per-Pod basis.

Kubernetes supports four Pod-Specific DNS Policies:



Pod's DNS Policy

The example below shows a Pod with its DNS policy set to **ClusterFirstWithHostNet** because it has **hostNetwork** set to **true**.

Demo

```
apiVersion: v1
Kind:  Pod
Metadata:
   name:  busybox
   namespace: default
spec:
  containers:
  - image: busybox:1.28
    command:
      - sleep:
      - "3600"
    imaagePullPolicy: IfNotPresent
    name: busybox
  restartPolicy: Always
  hostNetwork: true
  dnsPolicy: ClusterFirstWithHostNet
```

Pod's DNS Config

Pod's DNS Config allows users more control on the DNS settings for a Pod.

A user can specify three properties in the dnsConfig field:

- **nameservers**
- **searches**
- **options**

Pod's DNS Config

Here is an example Pod with custom DNS settings:

Demo

```
apiVersion: v1
Kind:  Pod
Metadata:
   namespace: default
   name:  dns-example
spec:
  containers:
 - name
   image: nginx
  dnsPolicy: “None”
  dnsConfig:
      nameservers:
         - 1.2.3.4
    searches:
       - ns1.svc.cluater-domain.example
       - my.dns.search.suffix
  options: busybox
   -  name: ndots
  valuek: “2”
 - name: edns0
```

Understanding the Configuration of DNS for Services and Pods

Problem Statement: Learn how to configure DNS for Services and Pods.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to demonstrate DNS for Services and Pods in Kubernetes:

1. Configure DNS policy
2. Create DNS custom configuration

TECHNOLOGY

Connecting Applications with Services

Kubernetes Model for Connecting Containers

1. Kubernetes assumes that Pods can communicate with other Pods, regardless of the host on which host they land.

2. Kubernetes gives every Pod its own cluster-private IP address. There is no need to explicitly create links between Pods or map Container ports and host ports.

Expose Pods to a Cluster

When Pods are exposed to a cluster, it makes them accessible from any Node in the cluster. This is done as shown below:

Demo

```
apiVersion: apps/ v1
    Kind:  Deployment
    Metadata:
       name:  my-nginx
    spec:
      selector:
        matchLabels
       run: my-nginx
      replicas: 2
      template:
          metadata:
            labels:
              run: my-nginx
    spec:
      containers:
       -  name: my-nginx
          image: nginx
          ports:
            - containerPort: 80
```

Expose Pods to a Cluster

How to check the Nodes that the Pod is running on:

Demo

```
Command:
kubectl apply -f ./run-my-nginx.yaml
kubectl get Pods -l run=my-nginx -o wide

Result:

NAME                     READY   STATUS    RESTARTS   AGE     IP          NODE
my-nginx-3800858182-jr4a2   1/1     Running   0          13s     10.244.3.4   kubernetes-minion-905m
my-nginx-3800858182-kna2y   1/1     Running   0          13s     10.244.2.5   kubernetes-minion-ljyd
```

Create a Service

A Kubernetes Service refers to the abstraction which defines a logical set of Pods running in a cluster with the same functionality. The configuration below helps create a Service for **nginx** replica:

Demo

```
#Create a Service for your 2 nginx replicas with
#kubectl expose:

Command:
kubectl expose deployment/my-nginx

Result:
service/my-nginx exposed
```

Demo

```
#This is equivalent to kubectl apply -f the following
#YAML:
apiVersion: v1
    Kind:  Service
    Metadata:
       name:  my-nginx
       labels:
       run: my-nginx
    spec:
      ports:
       - port: 80
      protocol: TCO
          selector:
            run: my-nginx
```

Create a Service

This specification will create a Service which targets TCP port 80 on any Pod with the run my-nginx label. It exposes it on an abstracted Service port.

Demo

```
Command:
kubectl get svc my-nginx

Result:
NAME        TYPE         CLUSTER-IP     EXTERNAL-
IP    PORT(S)    AGE
my-
nginx    ClusterIP    10.0.162.149    <none>        80/TC
P     21s
```

Create a Service

Service is backed by a group of Pods. These Pods are exposed through **Endpoints**.

The Service's Selector is evaluated continuously and the results are POSTed to an Endpoints object, also named **my-nginx**.

When a Pod dies, it is automatically removed from the Endpoints and new Pods matching the Service's Selector are automatically added to the Endpoints.

Create a Service

Check the Endpoints. Note that the IPs are the same as the Pods created in the first step.

Demo

```
Command:
kubectl describe svc my-nginx

Result:
Name:                my-nginx
Namespace:           default
Labels:              run=my-nginx
Annotations:         <none>
Selector:            run=my-nginx
Type:                ClusterIP
IP:                  10.0.162.149
Port:                <unset> 80/TCP
Endpoints:           10.244.2.5:80,10.244.3.4:80
Session Affinity:    None
Events:              <none>
```

Demo

```
Command:
kubectl get ep my-nginx

Result:
NAME        ENDPOINTS                       AGE
my-nginx    10.244.2.5:80,10.244.3.4:80     1m
```

Access a Service Using Environment Variables

When a Pod runs on a Node, the kubelet adds a set of environment variables for each active Service.

Demo

```
#Command:
kubectl exec my-nginx-3800858182-jr4a2 -- printenv | grep SERVICE

#Result:
KUBER    ES_SERVICE_HOST= 10.0.0.1
KUBERNETES_SERVICE_PORT= 443
KUBERNETES_SERVICE_PORT_HTTPS= 443
```

Access a Service Using DNS

Kubernetes offers a DNS cluster addon Service that automatically assigns DNS names to other Services.

Demo

```
#Command:
kubectl get services kube-dns --namespace=kube-system

#Result:
NAME        TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
kube-dns    ClusterIP   10.0.0.10     <none>         53/UDP,53/TCP    8m
```

Secure a Service

These are the requirements of a secure communication channel:

- Self-signed certificates for https
- An nginx server configured to use the certificates
- A secret that makes the certificates accessible to Pods

Secure a Service

nginx https example

Demo

```
Command:
make keys KEY=/tmp/nginx.key
CERT=/tmp/nginx.crt
kubectl create secret tls nginxsecret --
key /tmp/nginx.key --cert /tmp/nginx.crt

Result:
secret/nginxsecret created
```

configmap example

Demo

```
Command:
kubectl get secrets

Result:
NAME                  TYPE
                  DATA      AGE
default-token-
il9rc   kubernetes.io/service-account-
token   1         1d
nginxsecret           kubernetes.io/tls
                  2         1m
```

Demo

```
Command:
kubectl create configmap nginxconfigmap -
-from-file=default.conf

Result:
configmap/nginxconfigmap created

Command:
kubectl get configmaps

Result:
NAME              DATA   AGE
nginxconfigmap    1      114s
```

Expose a Service

Kubernetes supports two ways of exposing a service: **NodePorts** and **LoadBalancers.** The YAML configuration given below creates an **nginx HTTPS** replica to serve the traffic on the internet if the Node has a **public IP**:

Demo

```
Kubectl get svc my-nginx  -o yaml   |   grep NodePort.  -c 5
     uid:  07191fb3-f61a-11e5-42010f00002
 spec:
     clusterIP:  10.0.162.149
    ports:
       - name : http
         Nodeport: 31704
        port: 8080
        protocol:  TCP
        targetPort:  80
    - name: https
       Nodeport: 32453
      port:  443
      protocol:  TCP
      targetPort:  443
    selector:
       run:   my-nginx
```

Demo

```
Kubectl get Nodes  - 0  yaml   |   grep ExternalIP  -C
1
       - address:  104.197.41.11
          type:  ExternalIP
          allocable:
-

          -   address :  23.251.152.56
              type:  ExternalIP
              allocatable:
.  .  .

    $  curl https: //, EXTENAL - IP>:<NODE-PORT>  -l
.  .   .
     <h1> Welcome to nginx!</h1>
```

Expose a Service

The IP address in the EXTERNAL-IP column is the one that is available on the public internet.
The CLUSTER-IP is only available inside the cluster/private cloud network.

```
#Command:
kubectl edit svc my-nginx
kubectl get svc my-nginx

#Result:
NAME        TYPE           CLUSTER-IP     EXTERNAL-IP         PORT(S)                 AGE
my-nginx    LoadBalancer   10.0.162.149     xx.xxx.xxx.xxx      8080:30163/TCP      21

curl https://<EXTERNAL-IP> -k
...
<title>Welcome to nginx!</title>
```

Connecting Applications with Services

Problem Statement: Learn how to connect and configure applications with Services in Kubernetes.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to demonstrate connecting applications with Services in Kubernetes:

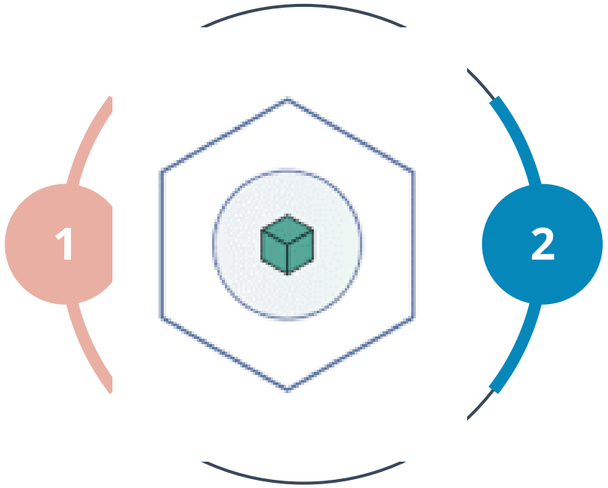
1. Expose Pod to cluster
2. Create Pod
3. Check the Nodes on which the Pod is running
4. Create a Service
5. Check the Service
6. Fetch environment variables

EndpointSlices

Introduction

EndpointSlices provide a simple way to track network Endpoints within a Kubernetes cluster.

EndpointSlices help to mitigate issues and provide an extensible platform for additional features such as Topological Routing.



They offer a more scalable and extensible alternative to Endpoints.

EndpointSlice Resources

An EndpointSlice contains references to a set of network Endpoints. Here is a sample EndpointSlice resource:

Demo

```
apiVersion: discovery.k8s.io/v1
Kind:  EndpointSlice
Metadata:
   name:  example-abc
   labels:
     kubernetes>io/servic-name: example
   addressType: IPv4
   ports:
   - name: http
      protocol: TCP
      port: 80
   endpoints :
   - addressess:
     -"10.1.2.3"
     conditions:
       ready: true
     host name : Pod-1
     NodeName: Node-1
     zone: us-west2-a
```

Address Types and Conditions

EndpointSlices support three address types, namely, IPv4, IPv6, and Fully Qualified Domain Name (FQDN).

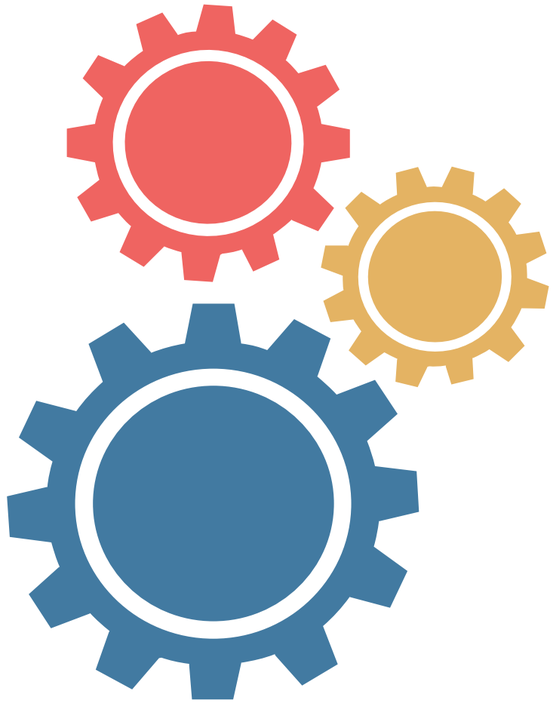
The EndpointSlice API stores three Endpoint conditions:

Ready

A condition that maps to a Pod's Ready condition

Terminating

A condition that indicates whether an Endpoint is terminating

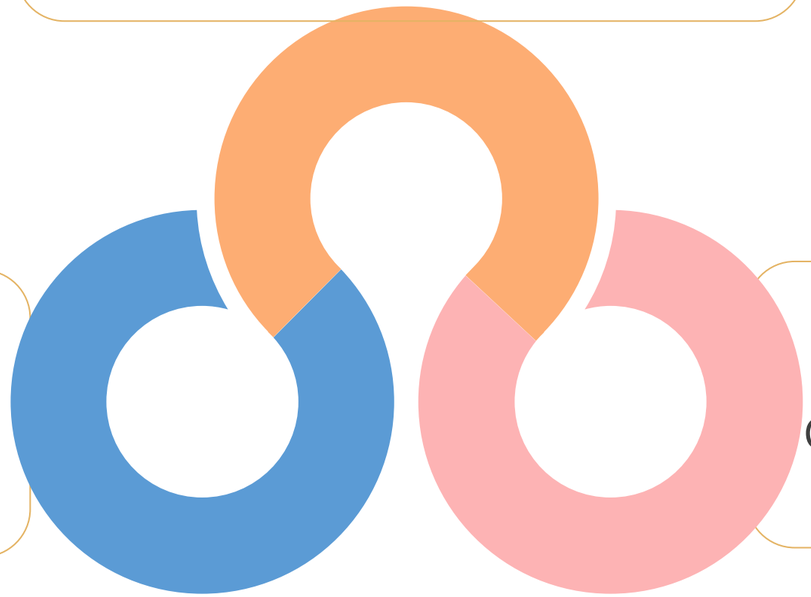


Serving

A condition that does not account for terminating states

Topology Information, Management, and Ownership

Topology information includes the location of the Endpoint and information about the corresponding Node and zone.



EndpointSlices are owned by the Service that the Endpointslice object tracks Endpoints for.

Most often, the Control Plane (specifically the EndpointSlice Controller) creates and manages EndpointSlice objects.

EndpointSlice Mirroring

When applications create custom Endpoint resources, the cluster's Control Plane mirrors most Endpoint resources to corresponding EndpointSlices. This helps to avoid concurrently writing to both Endpoints and EndpointSlice resources.

The Control Plane mirrors Endpoints resources unless:

- The Endpoints resource has an Endpointslice.kubernetes.io/skip-mirror label set to true.
- The Endpoints resource has a control-plane.alpha.kubernetes.io/leader annotation.
- The corresponding Service resource does not exist.
- The corresponding Service resource has a non-nil Selector.

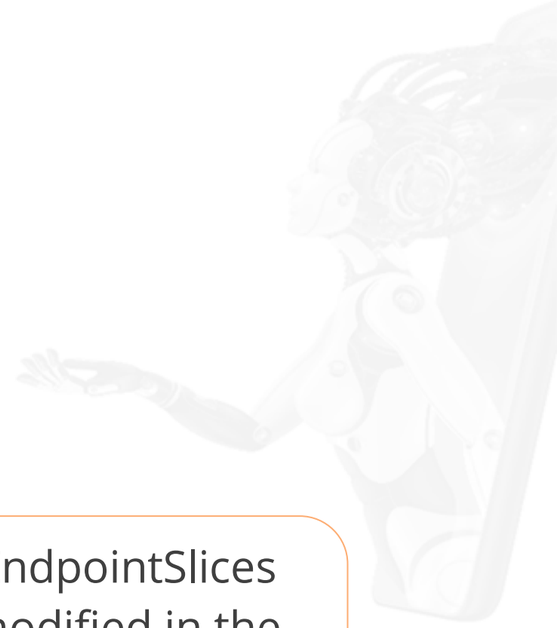
Distribution of EndpointSlices

To fill the EndpointSlices, the Control Plane does the following:

1. Iterate through existing EndpointSlices, remove Endpoints that are no longer desired, and update matching Endpoints that have changed

2. If there are still new Endpoints left to be added, try to fit them into a previously unchanged slice and/or create new ones

3. Iterate through EndpointSlices that have been modified in the first step and fill them up with any new Endpoints needed

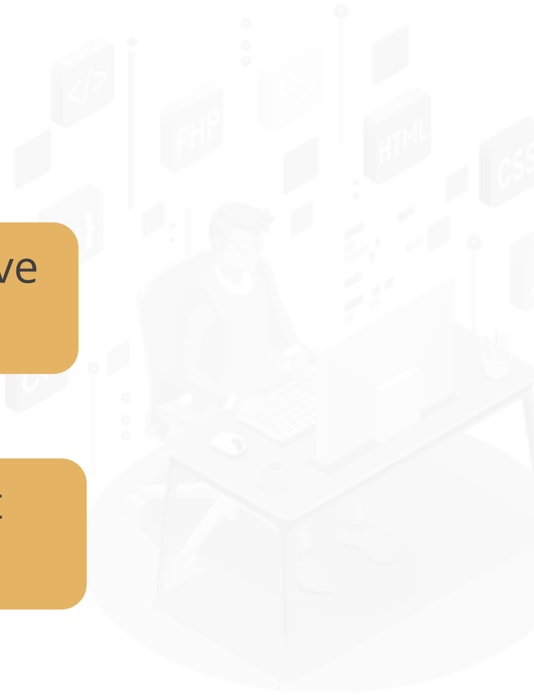


Duplicate Endpoints

Due to the nature of EndpointSlice changes, Endpoints may be represented in more than one EndpointSlice at the same time.

This naturally occurs as changes to different EndpointSlice objects can arrive at the Kubernetes client watch or cache at different times.

Implementations using EndpointSlice must be able to have the Endpoint appear in more than one slice.



Understanding the Configuration of EndpointSlices

Problem Statement: Learn how to configure Endpointslices in Kubernetes.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to demonstrate EndpointSlices in Kubernetes:

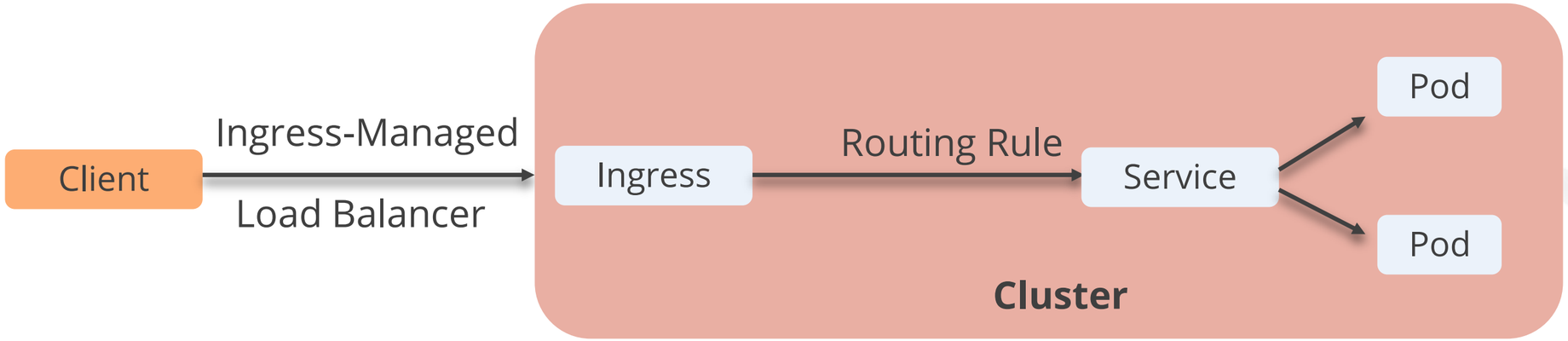
1. Create and configure a simple Endpointslice
2. Check the resources
3. Describe EndpointSlices

Ingress

What Is Ingress?

Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster.

Here is a simple example where an Ingress sends all its traffic to one Service:



An Ingress may be configured to give Services externally reachable URLs and Load Balance traffic, terminate SSL/TLS, and offer name-based virtual hosting.

Terminologies Used in Ingress

- Node
- Cluster
- Edge Router
- Cluster Network
- Service

Ingress Resource

An Ingress needs apiVersion, kind, and metadata fields.
Shown below is an example of a minimal Ingress resource:

Demo

```
apiVersion: networking.k8s.10/v1
Kind:  Ingress
Metadata:
   name:  minimal-ingress
   annotations:
     nginx.ingress.kubernetes.io/reqrite-target:/ addressType: IPv4
     spec:
     rules
     -  http
          paths:
          - path: /testpath
            pathtype: Prefix
               backend:
                service:
                   name: test
                   port : Pod-1
                      Number: 80
```

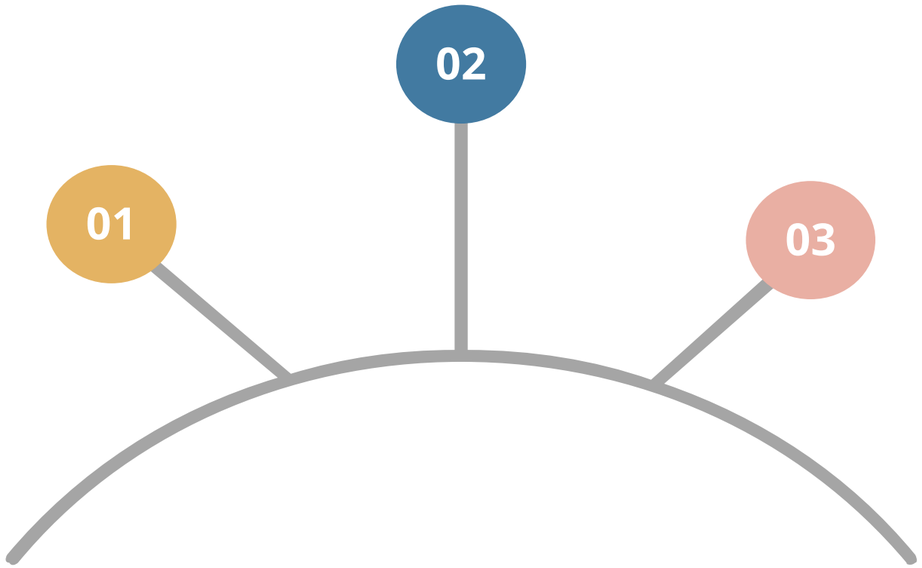
Ingress Rules

Each HTTP rule contains the following information:

An optional host

A backend

A list of paths



Default and Resource Backend

The default backend is conventionally a configuration option of the Ingress Controller and is not specified in the Ingress resources.

Default backend

A Resource backend is an ObjectRef to another Kubernetes resource within the same namespace as the Ingress object.

Resource backend

Resource Backend

A common usage for a Resource backend is to ingress data to an object storage backend with static assets.

Demo

```
apiVersion: networking.k8s.10/v1
Kind:  Ingress
Metadata:
   name:  ingress-resource-backend
   defaultBackend:
     respurce: IPv4
     apiGroup:
     kind:
     name: static-assets
     -  http
          paths:
          - path: /icons
            pathtype: ImplementationSpecific
               backend:
                resource:
                   apigroup: k8s.exaple.com
                   kind : StorageBucket
                   Name: icon-assets
```

Command to View Ingress

To view the created Ingress, use the command shown below:

Demo

```
kubectl describe ingress ingress-resource-backend
```

Path Types

Each path in an Ingress must have a corresponding path type. There are three supported path types:

ImplementationSpecific

Is dependent on the **IngressClass** for matching

Exact

Matches the **URL path exactly,** with case sensitivity

Prefix

Matches based on a **URL path prefix** split by the **/** separator

Hostname Wildcards

Precise matches require that the HTTP host header matches the host field. Wildcard matches require the HTTP host header to be equal to the suffix of the wildcard rule.

Hosts can be precise matches (for example **foo.bar.com**) or a wildcard (for example ***.foo.com**).

Host	Host header	Match
*.foo.com	bar.foo.com	Matches based on shared suffix
*.foo.com	baz.bar.foo.com	No match, wildcard only covers a single DNS label
*.foo.com	foo.com	No match, wildcard only covers a single DNS label

Hostname Wildcards

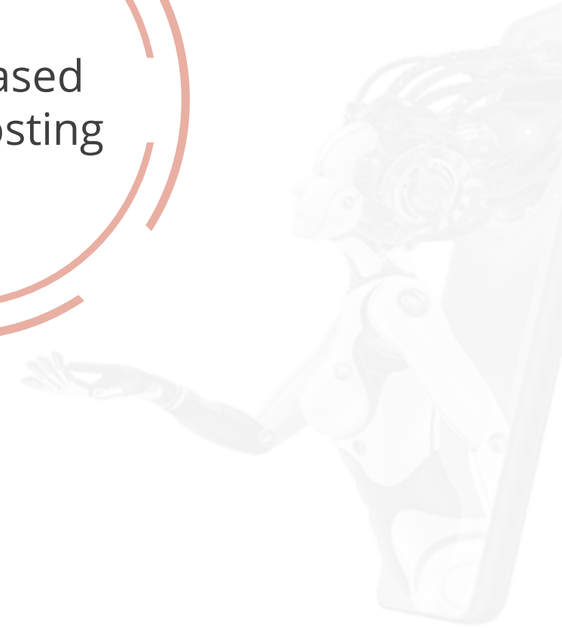
Demo

```
apiVersion: networking.k8s.10/v1
    Kind:  Ingress
    Metadata:
       name:  ingress-wildcard-host
       spec:
         rules:
         - host: “foo.bar.com”
         http:
         paths:
          -  pathtype: Prefix
              path: “/bar”
              backend:
                 service:
                   Name: service1
                   port:
                     number 80
         - host: “*foo.com”
            http:
             paths:
          -    pathtype: Prefix
               path: “/foo”
               backend:
                 service:
                   Name: service2
                   port:
                     number 80
```

An example of the configuration file showing the use of wildcards

Types of Ingress

- Ingress Backed by a Single Service
- Simple Fanout
- Name-Based Virtual Hosting
- TLS
- Load Balancing



Ingress Backed by a Single Service

A single service can be exposed using an Ingress by specifying a default backend with no rules.

Demo

```
apiVersion: networking.k8s.10/v1
Kind:  Ingress
Metadata:
   name:  test-ingress
   spec:
     defaultBackend
     service:
     name: test
     port:
        number: 80
```

Demo

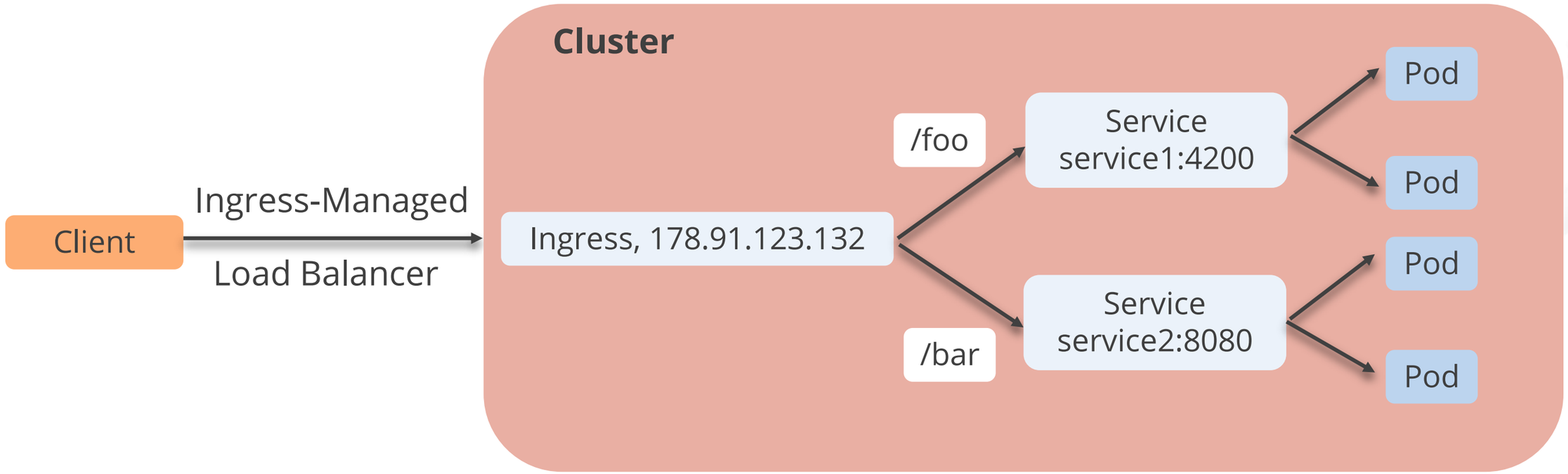
```
#Command view the state of the created Ingress:
kubectl get ingress test-ingress

Result:
NAME            CLASS         HOSTS    ADDRESS
      PORTS    AGE
test-ingress    external-
lb    *       203.0.113.123   80       59s

Where 203.0.113.123 is the IP allocated by the
Ingress controller to satisfy this Ingress.
```

Simple Fanout

A Fanout configuration routes traffic from a single IP address to more than one Service, based on the HTTP URI being requested.



Simple Fanout

It would require an Ingress such as:

Demo

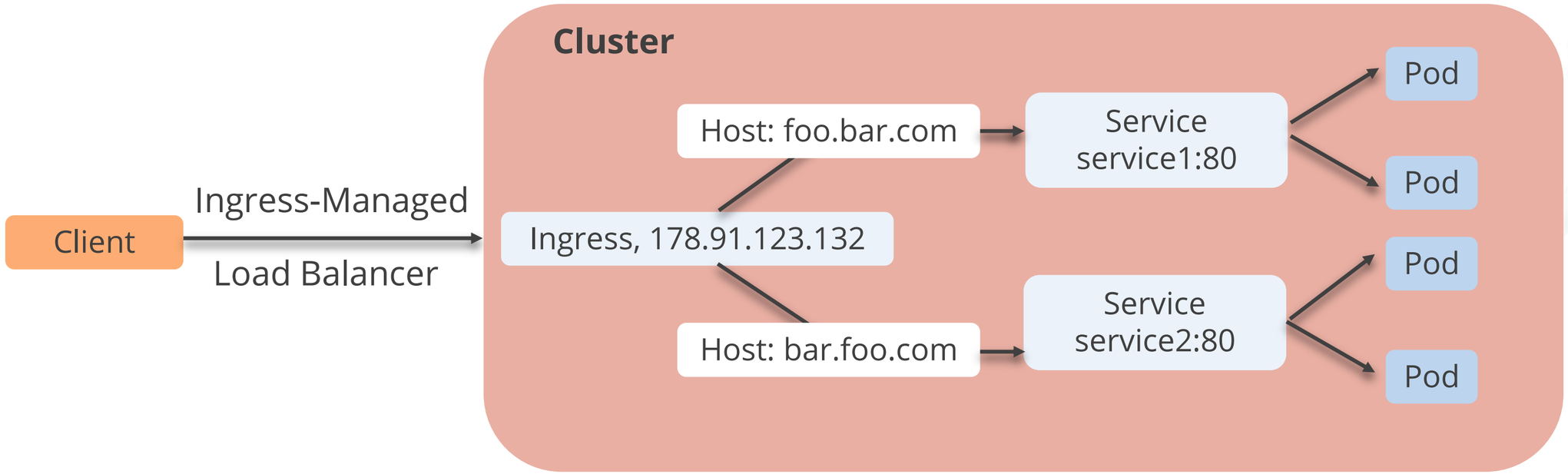
```
apiVersion: networking.k8s.10/v1
    Kind:  Ingress
    Metadata:
       name:  name-virtual-host-ingress
       spec:
         rules:
         - host: foo.bar.com
         http:
          paths:
          -  pathtype: Prefix
              path: "/"
              backend:
                 service:
                   Name: service1
                   port:
                     number 80
         - host: bar.foo.com
            http:
             paths:
          -    pathtype: Prefix
               path: "/"
               backend:
                 service:
                   Name: service2
                   port:
                     number 80
```

Demo

```
#Command to describe Ingress:
kubectl describe ingress simple-fanout-example
Result:
Name:             simple-fanout-example
Namespace:        default
Address:          178.91.123.132
Default backend:  default-http-backend:80 (10.8.2.3:8080)
Rules:
  Host          Path  Backends
  ----          ----  --------
  foo.bar.com
                /foo   service1:4200 (10.8.0.90:4200)
                /bar   service2:8080 (10.8.0.91:8080)
Events:
  Type     Reason  Age                From
Message
  ----     ------  ----               ----                    -
------
  Normal   ADD     22s                loadbalancer-controller
default/test
```

Name-Based Virtual Hosting

Name-based virtual hosts support routing HTTP traffic to multiple host names at the same IP address.



Name-Based Virtual Hosting

The Ingress shown below tells the backing Load Balancer to route requests based on the Host Header:

Demo

```
apiVersion: networking.k8s.10/v1
Kind:  Ingress
Metadata:
   name:  name-virtual-host-ingress
   spec:
     rules:
     - host: foo.bar.com
     http:
      paths:
      -  pathtype: Prefix
          path: "/"
          backend:
             service:
               Name: service1
               port:
                 number 80
```

Demo

```
- host: bar.foo.com
           http:
            paths:
         -    pathtype: Prefix
              path: "/"
              backend:
                service:
                  Name: service2
                  port:number 80
```

TLS

The tls.crt and tls.key contain the certificate and private key to be used for TLS. An Ingress can be secured using the configuration shown below:

Demo

```
apiVersion: v1
Kind:  Secret
Metadata:
   name:  testsecret-tls
   namespace: default
     data
     tls.crt: base64 encoded cert
     tls.key: base64 encoded key
   type: kubernetes.in/tls
```

TLS

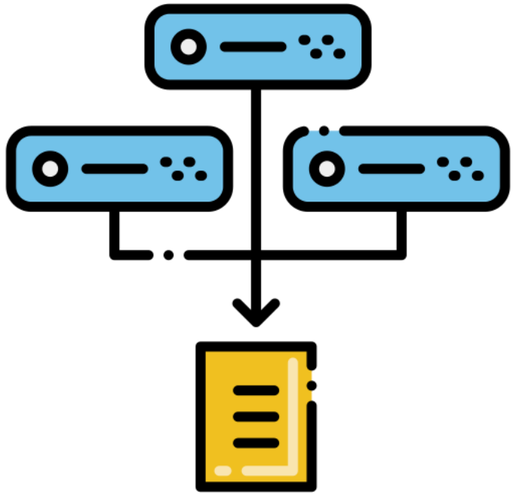
Ensure that the TLS secret created has come from a certificate that contains a Common Name (CN), also known as a Fully Qualified Domain Name (FQDN) for https-example.foo.com.

Demo

```
apiVersion: networking.k8s.10/v1
Kind:  Ingress
Metadata:
   name:  tls-example-ingress
   spec:
     tls:
     - host:
        -https-example.foo.com
       rules:
     - host: bar.foo.com
        http:
         paths:
         -path: /
      -    pathtype: Prefix
           backend:
             service:
               Name: service2
               port:
                 number 80
```

Load Balancing

Load Balancing settings are bootstrapped to an Ingress Controller. These settings can thus apply to all Ingresses.



Ingress does not expose health checks directly. They can be exposed using readiness probes.

Update an Ingress

To update an existing Ingress to be added to a new Host, edit the resource.

Demo

```
Command:
kubectl describe ingress test

Result:
Name:               test
Namespace:          default
Address:            178.91.123.132
```

Demo

```
spec:
 rules:
 - host: foo.bar.com
   https:
    paths:
    - backend
        service:
           Name: service1
           port:
             number 80
         path: /foo
  -    pathtype: Prefix
   - host: foo.bar.com
   https:
    paths:
    - backend
        service:
           Name: service2
           port:
             number 80
         path: /foo
         pathtype: Prefix
```

Understanding the Configuration of Ingress

Problem Statement: Learn how to configure Ingress in Kubernetes.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to demonstrate Ingress in Kubernetes:

1. Create a minimal Ingress configuration
2. Create a resource
3. Check the created Ingress
4. Describe the Ingress

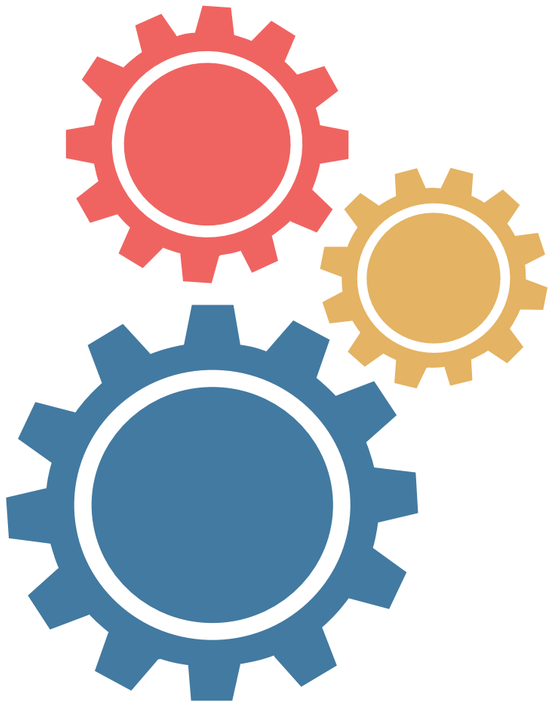
Ingress Controllers

Overview

For an Ingress resource to work, the cluster must have an Ingress Controller running.

Kubernetes as a project supports and maintains three Controllers.

AWS Ingress Controllers



nginx Ingress Controllers

GCE Ingress Controllers

Additional Controllers

Third-party Ingress Controllers include:

1. AKS Application Gateway Ingress Controller for AZure
2. Ambassador API Gateway
3. Apache APISIX Ingress Controller
4. Avi Kubernetes Operator for Load Balancing
5. The Citrix Ingress Controller
6. Contour
7. EnRoute
8. Gloo, an open-source Ingress Controller based on Envoy

Additional Controllers

9. HAProxy Ingress is an ingress
10. The HAProxy Ingress Controller
11. Skipper HTTP Router and Reverse Proxy
12. The Kong Ingress Controller
13. portworxVolume
14. Istio Ingress
15. Voyager
16. The Traefik Kubernetes Ingress Provider

Working with Ingress Controllers

Problem Statement: Understand the working of Ingress Controllers in Kubernetes.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to demonstrate Ingress Controllers in Kubernetes:

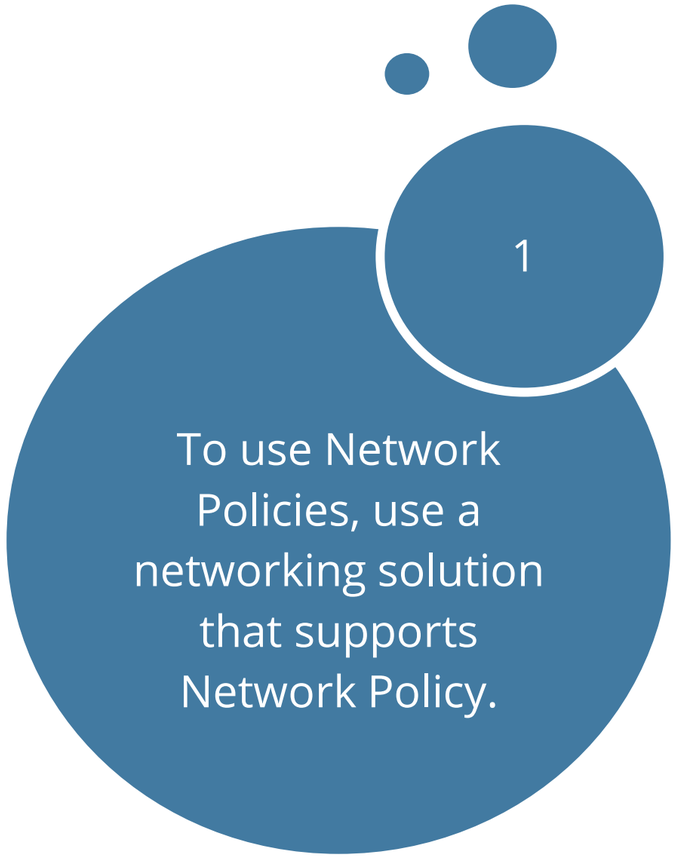
1. Install Ingress Controllers on Bare Metal
2. Verify Installation

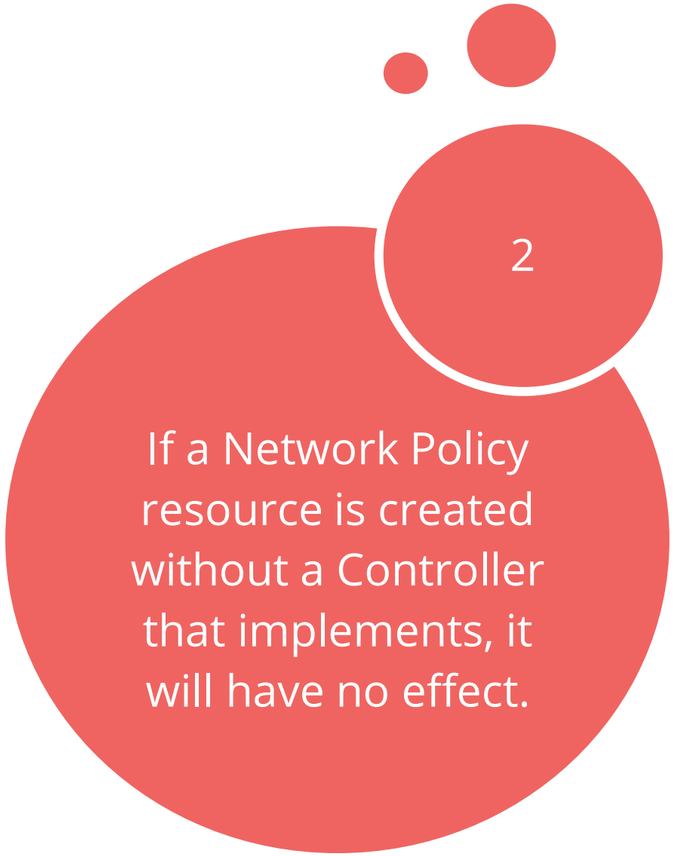
TECHNOLOGY

Network Policies

Network Policies

Network Policies are implemented by the network plugin.



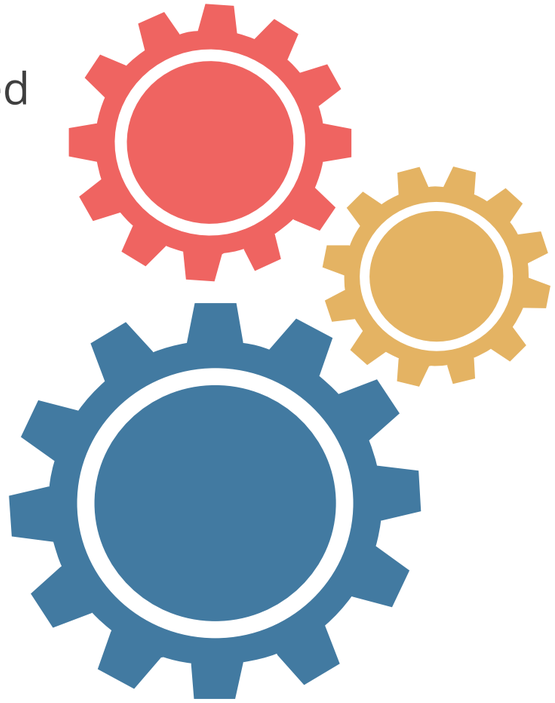


Introduction

Network Policies are application-centric constructs that enable specifying how a Pod is allowed to communicate with various network entities over the network.

The entities that a Pod can communicate with are identified through a combination of three Identifiers.

Other Pods that are allowed



Namespaces that are allowed

IP Blocks

Isolated and Non-isolated Pods

By default, Pods are non-isolated; they accept traffic from any source.

When there is a NetworkPolicy in a namespace selecting a particular Pod, the Pod will reject all connections not allowed by the NetworkPolicy.

If a policy (or policies) select(s) a Pod, the Pod is restricted to what is allowed by the union of the Ingress/Egress rules of the policy (or policies).

For a network flow to happen between two Pods, both the Egress policy on the source Pod and the Ingress policy on the destination Pod should allow the traffic.

Network Policy Resource: Example

Demo

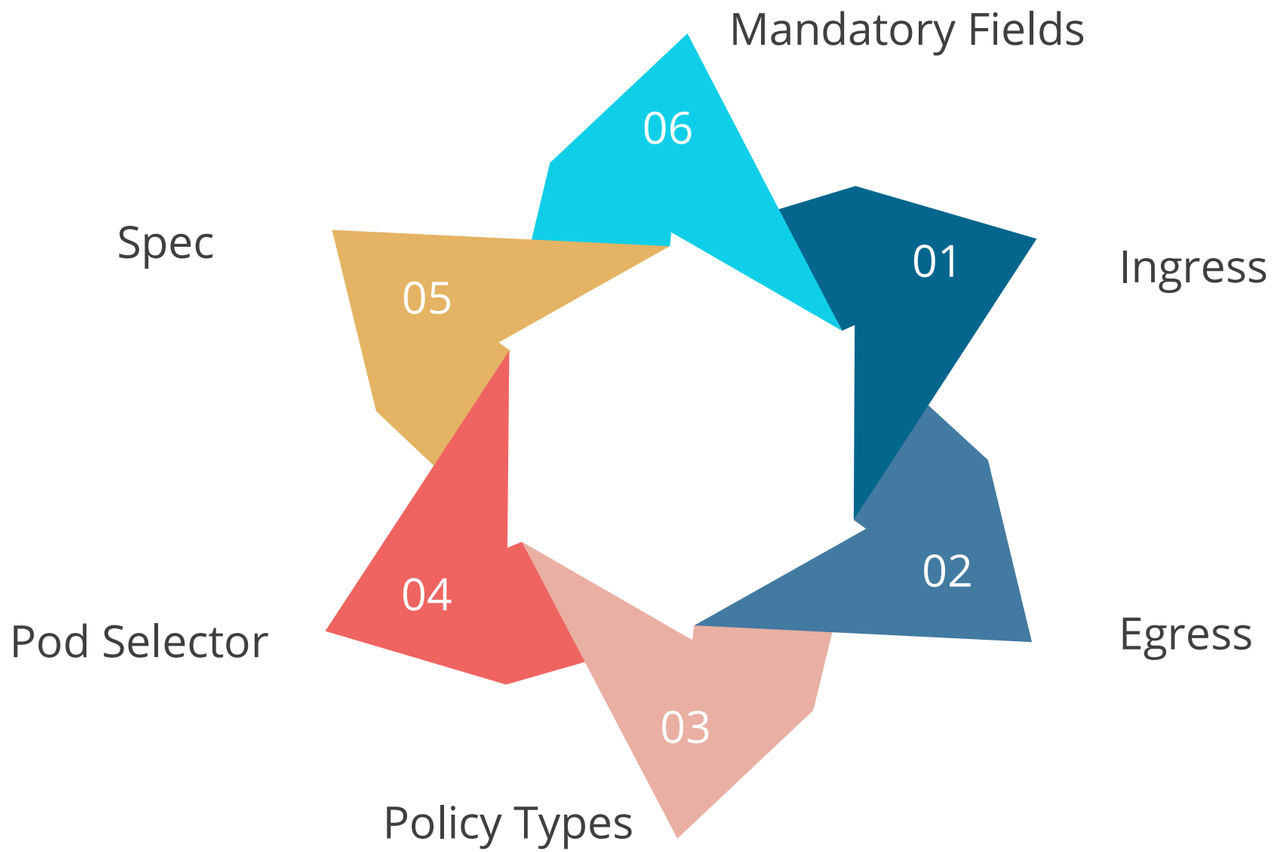
```
apiVersion: networking.k8s.io/v1
Kind:  Ingress
Metadata:
  name:  test-network-policy
  namespacee: default
spec:
  Podselector:
    matchLabels:
     role: db
     policytypes:
      - Ingress
       -Egress
     - ingress:
         -from:
            -ipBlock: 172.17.1.0.0/16
              cidr:
              except
           - 172.17.1.0.0/16
   - namespaceSelector:
```

Demo

```
matchLables:
                project: myproject
          - PodSelector:
             matchLables:
                role:  frontend
                   ports:
                    - protocol: TCP
                       port: 6379
                egress:
                - to:
                 - ipBlock
                     cidr : 10.0.0.0/24
                ports:
                - protocol: TCP
                 - port: 5978
```

Network Policy Resource

The configuration file has six attributes:



Behavior of To and From Selectors

namespaceSelector and podSelector

A single To or From entry, which specifies both namespaceSelector and PodSelector, selects particular Pods within particular namespaces

PodSelector

Selects Pods that should be allowed as Ingress or Egress destinations within the same namespace as the NetworkPolicy

namespaceSelector

Selects namespaces for which all Pods should be allowed as Ingress sources or Egress destinations

ipBlock

Selects cluster-external IP CIDR ranges that can be allowed as Ingress sources or Egress destinations

Behaviour of To and From Selectors

Demo

```
...
    ingress:
    - fromd:
     namespaceSelector:
        matchLables:
         user: alice
         PodSelector:
       matchLabels:
         role: client
...
```

Policy contains a single From element allowing connections from Pods with the label role=client in namespaces with the label user=alice.

Default Policies

By default, if no policies exist in a namespace, then all Ingress and Egress traffic is allowed to and from Pods in that namespace.

Demo

```
...
    apiVersion: networking.k8s.10/v1
    Kind:  NetworkPolicy
    Metadata:
       name:  default-deny-ingress
       spec:
           Podselector:  {}
           policyTypes:
           - Ingress
```

Default **deny all** Ingress traffic

Demo

```
...
    apiVersion: networking.k8s.10/v1
    Kind:  NetworkPolicy
    Metadata:
       name:  allow-all-ingress
    spec:
       Podselector:  {}
       Ingress:
        - {}
        policyTypes:
        - Ingress
```

Default **allow all** Ingress traffic

Default Policies

Demo

```
...
    apiVersion: networking.k8s.10/v1
    Kind:  NetworkPolicy
    Metadata:
       name:  default-deny-egress
    spec:
       Podselector:  {}
       Ingress:
        - {}
        policyTypes:
        - Ingress
```

Default **deny all** Egress traffic

Demo

```
...
    apiVersion: networking.k8s.10/v1
    Kind:  NetworkPolicy
    Metadata:
       name:  allow-all-egress
    spec:
       Podselector:  {}
       engress:
        - {}
        policyTypes:
        - engress
```

Default **allow all** Egress traffic

Demo

```
...
    apiVersion: networking.k8s.10/v1
    Kind:  NetworkPolicy
    Metadata:
       name:  default-deny-all
    spec:
       Podselector:  {}
        policyTypes:
        - engress
        - Ingress
```

Default **deny all** Ingress and all Egress traffic

Working with Network Policy Resources

Problem Statement: Create and work with network policies in Kubernetes.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to demonstrate Network Policies in Kubernetes:

1. Create a network policy
2. Create a network policy configuration
3. Create the resource
4. Verify the installation

Adding Entries to Pod /etc/hosts With HostAliases

Default Hosts File Content

Adding entries to a Pod's etc or hosts file provides Pod-level override of hostname resolution when DNS and other options are not applicable. These custom entries can be added with the HostAliases field in PodSpec.

Demo

```
Command to start a Nginx Pod that is assigned to a Pod IP is as follows:
kubectl run nginx --image nginx

Result:
Pod/nginx created

Examine a Pod IP - Command:
kubectl get Pods --output=wide

Result:
NAME      READY      STATUS     RESTARTS    AGE     IP            NODE
nginx     1/1        Running    0           13s     10.200.0.4    worker0
```

Default Hosts File Content

This is how the Host File Content will look:

Demo

```
Command:
kubectl exec nginx -- cat /etc/hosts

 Result:
 # Kubernetes-managed hosts file.
 127.0.0.1       localhost
 ::1    localhost ip6-localhost ip6-loopback
 fe00::0         ip6-localnet
 fe00::0         ip6-mcastprefix
 fe00::1         ip6-allNodes
 fe00::2         ip6-allrouters
 10.200.0.4      nginx
```

Additional Entries with HostAliases

In addition to the default boilerplate, more entries can be made to the hosts file. For example, to resolve foo.local, bar.local to 127.0.0.1 and foo.remote, bar.remote to 10.1.2.3, configure HostAliases for a Pod under .spec.hostAliases.

Demo

```
apiVersion: v1
 Kind:  Pod
 Metadata:
    name:  hostaloases-Pod
 spec:
    restartPolicy:  Never
    hostAliases:
     -ip: "127.0.0.1"
      hostnames:
     - "foo.local"
     - "bar.local"
    - ip: "10.1.2.3"
      hostnames:
      - "foo.remote"
      - "bar.remote"
    containers:
      - name: cat-hosts
        image: busybox
        command:
         - cat
        arges
        - " /etc/hosts"
```

Additional Entries with HostAliases

A Pod can be configured by running the following command:

Demo

```
kubectl apply -f https://k8s.io/examples/service/networking/hostaliases-Pod.yaml
Result:
Pod/hostaliases-Pod created
Examine a Pod's details to see its IPv4 address and its status, using the following command:kubectl get Pod --output=wide
Result:
NAME                          READY     STATUS      RESTARTS   AGE       IP              NODE
hostaliases-Pod               0/1       Completed   0          6s        10.200.0.5      worker0
The hosts file content looks like this:
kubectl logs hostaliases-Pod
Result:
# Kubernetes-managed hosts file.
127.0.0.1        localhost
::1      localhost ip6-localhost ip6-loopback
fe00::0  ip6-localnet
fe00::0  ip6-mcastprefix
fe00::1  ip6-allNodes
fe00::2  ip6-allrouters
10.200.0.5       hostaliases-Pod

# Entries added by HostAliases.
127.0.0.1        foo.local        bar.local
10.1.2.3 foo.remote       bar.remote
with the additional entries specified at the bottom.
```

Adding Entries to pod_etc_hosts with HostAliases

Problem Statement: Interpret how to add entries to Pod, etc, or hosts with Host Aliases in Kubernetes.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to demonstrate adding entries to Pod, etc, or hosts with HostAliases in Kubernetes:

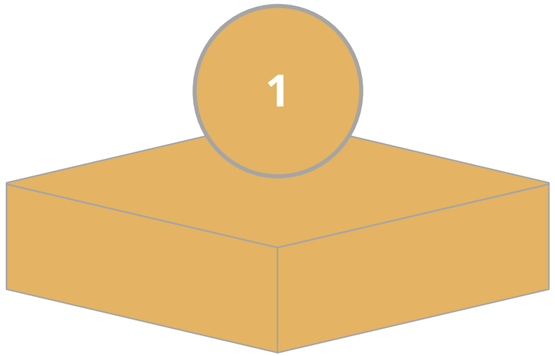
How to add entries to resources with host aliases:

1. Get Default hosts for the Pod webserver that we created earlier
2. Create a configuration with additional entries
3. Create the resource
4. Examine the Pod details
5. Observe the logs

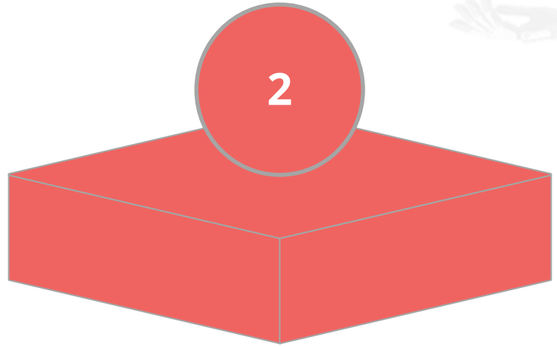
IPv4/IPv6 Dual-Stack

Introduction

IPv4/IPv6 Dual-Stack networking enables the allocation of both IPv4 and IPv6 addresses to Pods and Services.



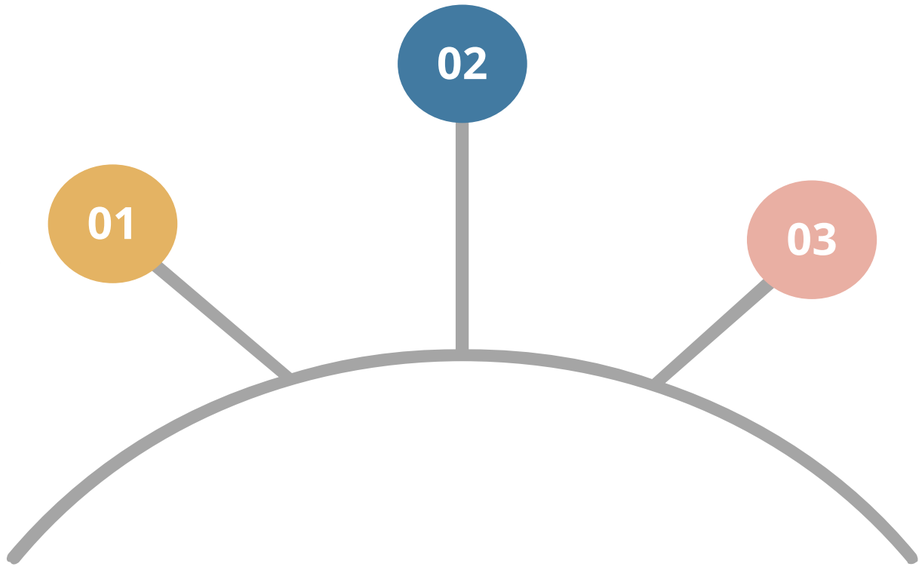
It is enabled by default for the Kubernetes cluster starting in 1.21, allowing simultaneous assignment of both IPv4 and IPv6 addresses.



Supported Features

The IPv4/IPv6 Dual-Stack on the Kubernetes cluster provides the following Services:

IPv4 and IPv6 enabled Services



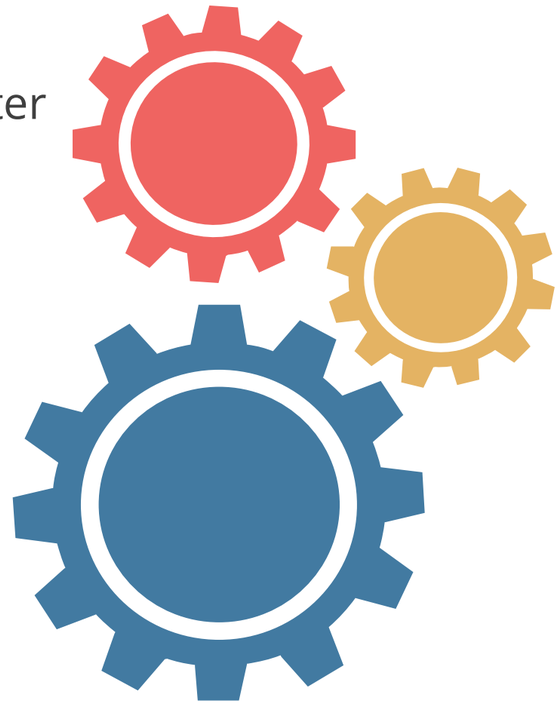
Dual-Stack Pod networking (a single IPv4 and IPv6 address assignment per Pod)

Pod off-cluster Egress routing (for example, the internet) via both IPv4 and IPv6 interfaces

Prerequisites

There are three prerequisites for utilizing IPv4/IPv6 Dual-Stack Kubernetes clusters:

Kubernetes 1.20 or later



Provider support for Dual-Stack networking

A network plugin that supports Dual-Stack Configure IPv4/IPv6 dual-stack

Services

When a Service is defined, it can be optionally configured as dual-stack.

.spec.ipFamilyPolicy field can be used to specify the behavior. It takes one of the following values:

Dual-Stack Options on New Services

This Service specification does not explicitly define .spec.ipFamilyPolicy. Kubernetes **assigns** a **cluster IP** for the **Service** from the first configured service-cluster-ip-range after the Service is created. It sets the **.spec.ipFamilyPolicy** to SingleStack.

Demo

```
apiVersion: v1
Kind:  service
Metadata:
   name:  my-service
   labels:
     app: MyApp
  spec: IPv4
   selector:
    app: MyApp
   ports:
     - protocol: TCP
      port: 80
```

Dual-Stack Options on New Services

The Service specification shown below explicitly defines **PreferDualStack** in .spec.ipFamilyPolicy. .spec.ClusterIPs is the primary field and .spec.ClusterIP is a secondary field.

Demo

```
apiVersion: v1
Kind:  service
Metadata:
   name:  my-service
   labels:
     app: MyApp
  spec:
   ipFamilyPolicy: PreferDualStack
    selector:
    app: MyApp
   ports:
     - protocol: TCP
      port: 80
```

Dual-Stack Options on New Services

The Service specification shown below explicitly defines IPv6 and IPv4 in .spec.ipFamilies and PreferDualStack in **.spec.ipFamilyPolicy**:

Demo

```
apiVersion: v1
Kind:  service
Metadata:
   name:  my-service
   labels:
     app: MyApp
  spec:
   ipFamilyPolicy: PreferDualStack
   ipFamilies:
   - IPpv6
   - IPv4
    selector:
    app: MyApp
   ports:
     - protocol: TCP
      port: 80
```

Dual-Stack Defaults on Existing Services

The examples shown below demonstrate the default behavior of Dual-Stack when it is newly enabled on a cluster with Services:

Demo

```
apiVersion: v1
    Kind:  service
    Metadata:
       name:  my-service
       labels:
         app: MyApp
      spec:
         selector:
           app: MyApp
         ports:
          - protocol: TCP
            port: 80

#Validate this behavior by using kubectl to inspect an
#existing service, using the following command:

kubectl get svc my-service -o yaml
```

Demo

```
apiVersion: v1
   Kind:  service
   Metadata:
      name:  my-service
      labels:
        app: MyApp
     spec:
       clusterIPs: 10.0.197.123
       clusterIPs:
         - 10.0.197.123
       ipFamilies:
         - IPv4
       ipFamilyPolicy: singleStack
       ports:
        - port: 80
          protocol: TCP
          targetPort: 80
       selector:
       app: MyApp
      type: ClusterIP
     status:
       LoadBalancer: {}
```

Dual-Stack Defaults on Existing Services

Upon enabling dual-stack on a cluster, existing Headless Services with Selectors are configured by the Control Plane. **.spec.ipFamilyPolicy** will be set to SingleStack and**.spec.ipFamilies** will be set to the address family of the first service cluster IP range.

Demo

```
apiVersion: v1
Kind:  service
Metadata:
   name:  my-service
   labels:
     app: MyApp
  spec:
     selector:
       app: MyApp
     ports:
      - protocol: TCP
        port: 80
```

Dual-Stack Defaults on Existing Services

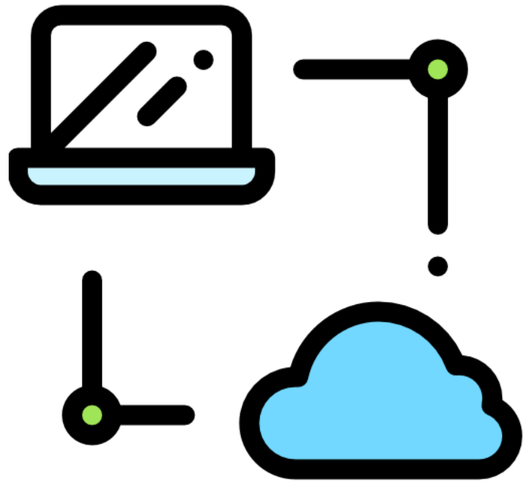
The behavior can be validated by using kubectl to inspect an existing Headless Service with Selectors, using the **kubectl get svc my-service -o yaml** command.

Demo

```
apiVersion: v1
Kind:  service
Metadata:
   labels:
     app: MyApp
  name:  my-service
  spec:
    clusterIP: None
    clusterIPs:
      - None
    ipFamilies:
      - IPv4
   ipFamilyPolicy: singleStack
    ports:
     - port: 80
       protocol: TCP
       targetPort: 80
    selector:
    app: MyApp
```

Egress Traffic

To enable Egress traffic in order to reach off-cluster destinations (for example, the public internet) from a Pod that uses non-publicly routable IPv6 addresses, enable the Pod to use a publicly routed IPv6 address.



Switch Services Between Single-Stack and Dual-Stack

Services can be switched between Dual-Stack and Single-Stack Services and vice versa.

To change a Service from Single-Stack to Dual-Stack, change **.spec.ipFamilyPolicy** from SingleStack to **PreferDualStack** or **RequireDualStack** as desired.

Before:

```
spec:
ipFamilyPoilcy SingleStack
```

After:

```
spec:
ipFamilyPoilcy PreferDualStack
```

Working with IPv4_IPv6_dual_stack

Problem Statement: Understand the working of IPv4/IPv6 dual-stack in Kubernetes.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to demonstrate IPv4/IPv6 dual-stack in Kubernetes:

1. Create a default dual-stack service
2. Create a preferred dual-stack service

Cluster Networking

Overview

Networking is a central part of Kubernetes, but it can be challenging to understand exactly how it is expected to work.

There are four distinct networking problems to address:

1. Highly-coupled Container-to-Container communications
2. Pod-to-Pod communications
3. Pod-to-Service communications
4. External-to-Service communications

Kubernetes Network Model

Every Pod is assigned an IP address. Kubernetes imposes fundamental requirements on any networking implementation.

Kubernetes allows communication between:

1. Pods in the host network of a Node and Pods on all Nodes without NAT

2. Agents on a Node and all the Pods on that Node

Implement the Kubernetes Networking Model

There are a number of ways in which this network model can be implemented:

1. AOS from Apstra
2. AWS VPC CNI for Kubernetes
3. Azure CNI for Kubernetes
4. Cilium
5. CNI-Genie from Huawei
6. cni-ipvlan-vpc-k8s
7. Coil
8. Contiv
9. Contrail/Tungsten Fabric
10. DANM
11. Knitter
12. Kube-OVN
13. Kube-router
14. Multus
15. NSX-T
16. Nuage Networks VCS

Implement the Kubernetes Networking Model

17 OpenVSwitch

18 ACI

19 Flannel

20 Jaguar

21 Romana

22 Weave Net from Weaveworks

23 OVN

24 Antrea

25 Google Compute Engine

26 k-vswitch

27 Big Cloud Fabric from Big Switch Networks

Key Takeaways

- A Kubernetes Service is a logical abstraction for a deployed group of Pods in a cluster.
- Kubernetes supports two primary modes of finding a Service, namely, environment variables and DNS.
- Network Policies are application-centric constructs that specify how a Pod is allowed to communicate with various network entities.
- IPv4/IPv6 Dual-Stack networking supports IPv4 and IPv6 addresses to be allocated to Pods and Services.

Create a Service to Expose an Application Using EndpointSlices

Problem Statement:

You have deployed a containerized application and now you have to expose it by creating a service. You also have to create EndpointSlices and add entries to Pods.

Steps to Perform:

1. Connecting application with services
2. Creating EndpointSlices
3. Adding entries to Pods with host aliases

LESSON-END PROJECT