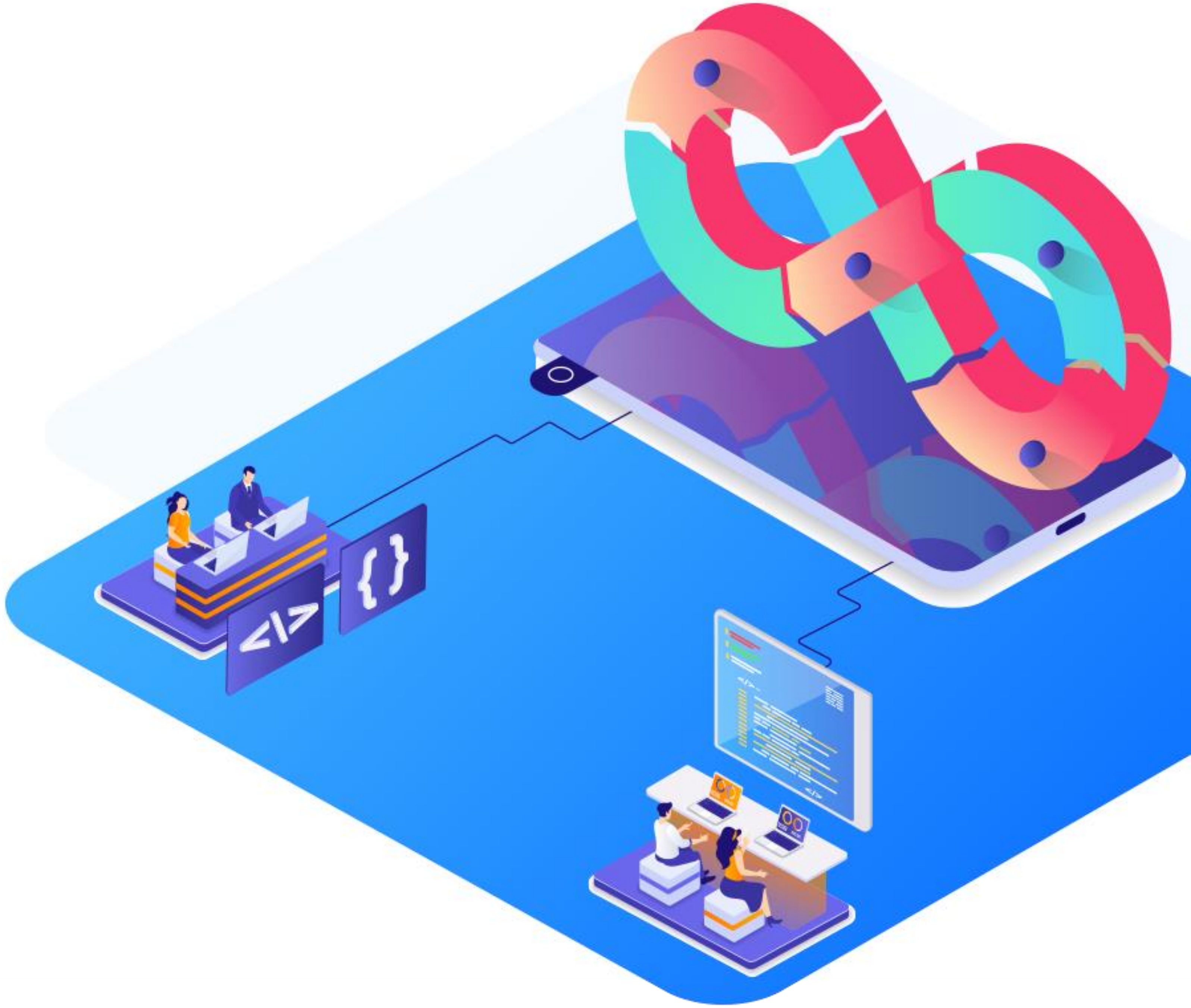
Container Orchestration Using Kubernetes



Services, Load Balancing, and Networking

Learning Objectives

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

- Outline networking solutions with Kubernetes to manage containerized applications
- Analyze the topology and DNS to make Kubernetes services discoverable
- Work with EndpointSlices and Ingress controllers to enable object tracking
- Create and apply an IPv4/IPv6 dual-stack network policy for the autonomous transition of addresses
- List how cluster networking is implemented to run containers across multiple machines

Overview of Services

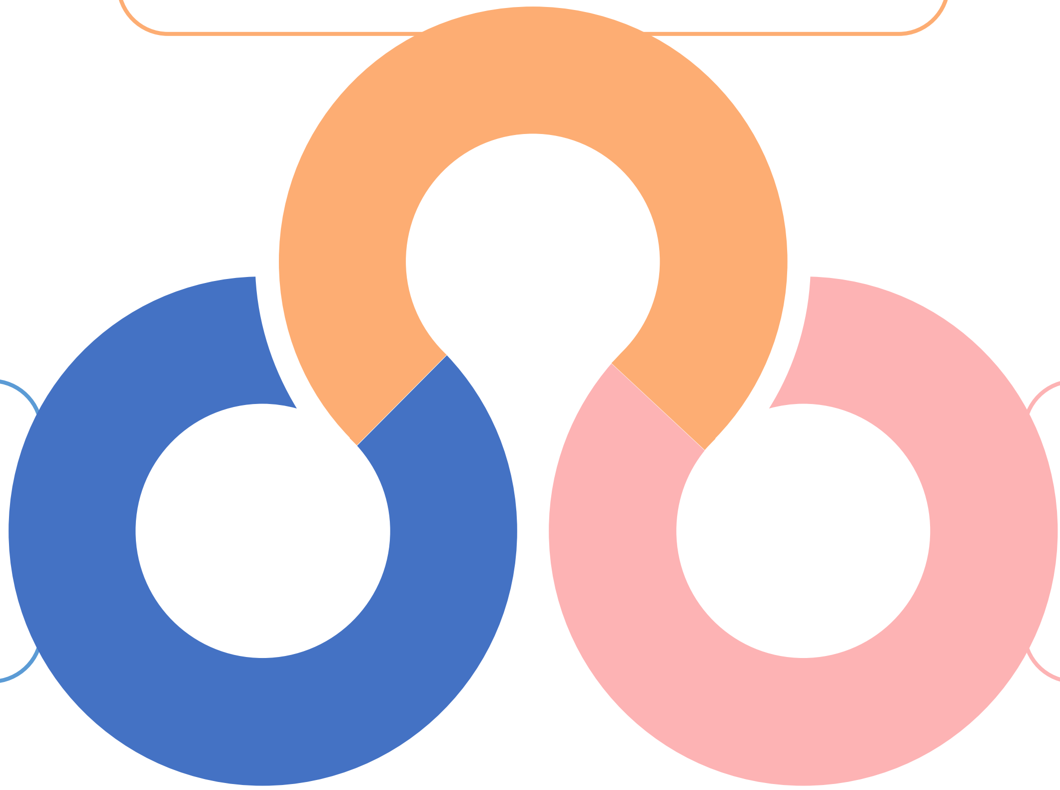
Services

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

A Service enables the group of pods.

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

The IP address of the pod cannot be changed when a service is running on it.



Networking

The Kubernetes networking addresses four concerns:

Loopback is used by the containers within a pod for communication.

Communication between pods happens through cluster networking.

The 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 the balanced distribution of network traffic or client requests across multiple servers. It can be:

Internal load balancing

External load balancing

Cluster Networking

Overview of Networking

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 and all the pods on a node

Implementing Kubernetes Network Model

There are several 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

Implementing Kubernetes Network Model

There are several ways in which this network model can be implemented:

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

Quick Check

You have a Kubernetes cluster with multiple pods running different applications. A new microservice is deployed, and it needs to communicate with another pod in a different namespace. Which of the following best describes how Kubernetes handles network communication between these pods?

A. Pods in different namespaces cannot communicate without a network plugin.

B. Kubernetes automatically enables pod-to-pod communication across namespaces without additional configuration.

C. Pods can only communicate across namespaces using NodePort or LoadBalancer services.

D. Network Policies are required for cross-namespace communication between pods.

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 microservice.

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

Kubernetes also balances the load between pods.

Define a Service

A service is an object whose definition can be created, viewed, or modified using the Kubernetes API. Usually, a tool such as kubectl is used to make these API calls.

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

This specification creates a new service object called **my-service,** which targets TCP port **9376** on any pod with the app labeled **MyApp**.

Define a Service

A service can map any incoming port to a **targetPort**. By default, the **targetPort** is set to the same value as the port field.

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 although they can also abstract other kinds of backends.

To define a service without a pod selector, add the following in the YAML file:

```
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 endpoint object. The following is an example:

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

Application Protocol

The Kubernetes Services expose one or more ports. A port exposed by an application can serve a specific application protocol such as HTTP, TCP, gRPC, and so on.

1. The **appProtocol** field provides a way to specify an application protocol for each service port.

2. It follows standard Kubernetes label syntax.

The values should either be IANA standard service names or domain-prefixed names.

Virtual IPs

When users create a service, a new virtual IP (also known as a **clusterIP**) is created on their behalf.

1. Every node in a Kubernetes cluster runs a **kube-proxy**.
2. **kube-proxy** is responsible for implementing a type of virtual IP for services other than **ExternalName**.

Multi-Port Services

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

```
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
```

IP Addresses Specification

A cluster IP address can be specified as a 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.

Environment Variables

The kubelet adds a set of environment variables for each active service when a pod is running on a node.

```
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
```

The example above 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**.

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 administrator should 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.

DNS

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 can be classified into two categories:

- Services with selectors
- Services without selectors

Publishing Services (ServiceTypes)

Kubernetes **ServiceTypes** help to specify the required kind of service. The default is ClusterIP.

Type values and their behaviors include:

ClusterIP

ExternalName

NodePort

LoadBalancer

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:

```
apiVersion: v1
kind:  service
metadata:
   name:  my-service
spec:
   type: NodePort
   selector:
    app: MyApp
   ports:
       # By default, the ‘targetPort’ is set to the same value as the ‘ port’ field.
      - port: 80
        targetPort: 80
        # By default,the kubernetes control plane will allocate a port from a range (30000-32767)
        NodePort:  30007
```

Type LoadBalancer

On cloud providers support external load balancers by setting the type field to **LoadBalancer** which provisions a load balancer for the service.

```yaml
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 as shown.

The **.status.loadBalancer** field contains information about the provisioned balancer.

Load Balancers with Mixed Protocol Types

When multiple ports are configured for load balancer services, all ports must have the same protocol by default.

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 multiple ports are specified.

Disable Load Balancer NodePort Allocation

The node port allocation for a service of **Type=LoadBalancer** can be optionally disabled. This can be done by setting the field **spec.allocateLoadBalancerNodePorts** to false.

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, those node ports will not be de-allocated automatically.

The **ServiceLBNodePortControl** feature gate must be enabled to use this field.

Quick Check

You are managing a Kubernetes cluster and need to expose a service that allows clients to discover the IP addresses of backend pods directly, without load balancing or assigning a cluster IP. Which Kubernetes service type would be most appropriate for this requirement?

A. ClusterIP

B. NodePort

C. LoadBalancer

D. Headless service

Connecting Applications with Services

Kubernetes Model for Connecting Containers

After configuring a continuously running and replicated application, this application can be exposed to a network.

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

2. Kubernetes gives every pod its own cluster-private IP address. As a result, there is no need to make explicit linkages between pods or map container ports to 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 follows:

```
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

The command to verify the nodes on which the pod is running is as follows:

```
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 that defines a logical set of pods running in a cluster with the same functionality. The configuration below helps to create a service for **nginx** replica:

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

Command:
kubectl expose deployment/my-nginx

Result:
service/my-nginx exposed
```

```
#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: TCP
       selector:
         run: my-nginx
```

Create a Service

This specification will create a service that targets TCP port 80 on any pod with the run my-nginx label. It is exposed in an abstracted service port.

```
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/TCP     21s
```

Create a Service

A 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 automatically gets added to the endpoints.

Create a Service

Check the endpoints and note that the IPs are the same as the pods created in the first step:

```
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>
```

```
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.

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

Result:
KUBERNETES_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.

```
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
- Nginx server configured to use the certificates
- A secret that makes the certificates accessible to pods

Secure a Service

The following commands are used to create a Nginx HTTPS service useful in verifying proof of **concept**, **keys**, **secrets**, **configmap**, and **end-to-end HTTPS service** creation in Kubernetes.

```
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

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
```

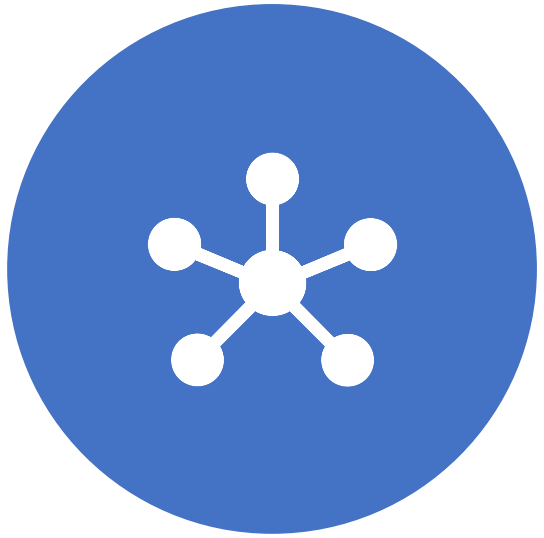
Secure a Service

Here is an example of how to create a ConfigMap:

```
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

LoadBalancers

Expose a Service

The YAML configuration given below creates a Nginx HTTPS replica to serve the traffic on the internet if the node has a public IP:

```
kubectl get svc my-nginx -o yaml | grep nodePort -C 5
  uid: 07191fb3-f61a-11e5-8ae5-42010af00002
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
```

```
kubectl get nodes -o yaml | grep ExternalIP -C 1
    - address: 104.197.41.11
      type: ExternalIP
    allocatable:
--
    - address: 23.251.152.56
      type: ExternalIP
    allocatable:
...

$ curl https://<EXTERNAL-IP>:<NODE-PORT> -k
...
<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 or 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>
```

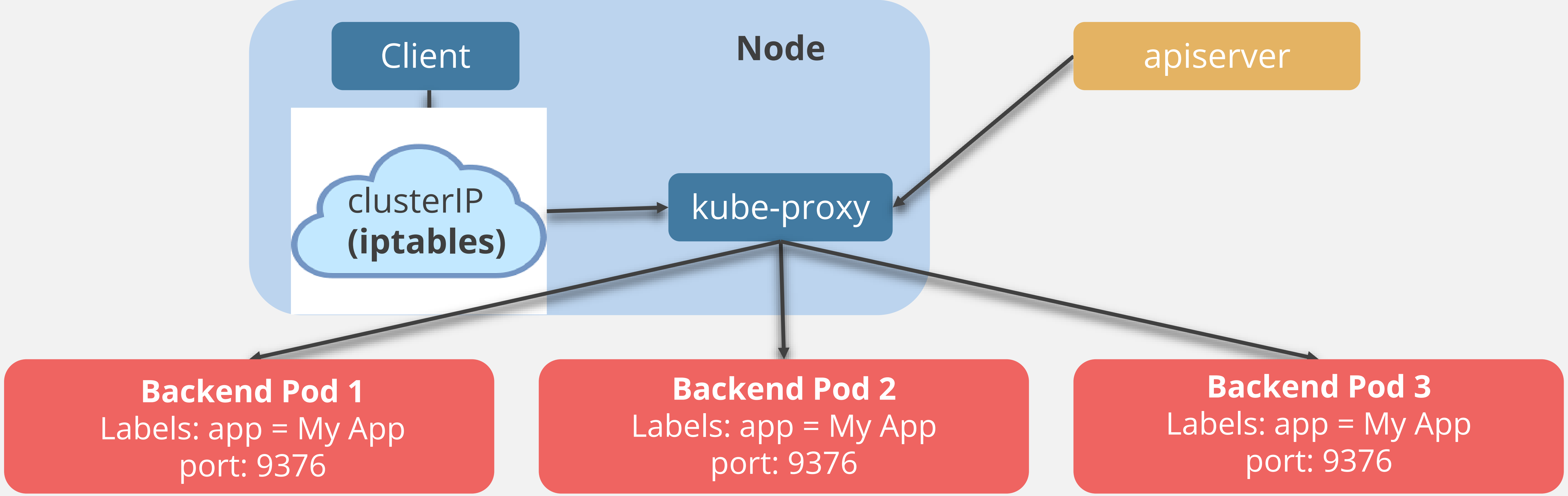
Proxy Modes

Three types of proxy modes supported in Kubernetes are:

- User Space Proxy Mode
- iptables Proxy Mode
- IPVS Proxy Mode

User Space Proxy Mode

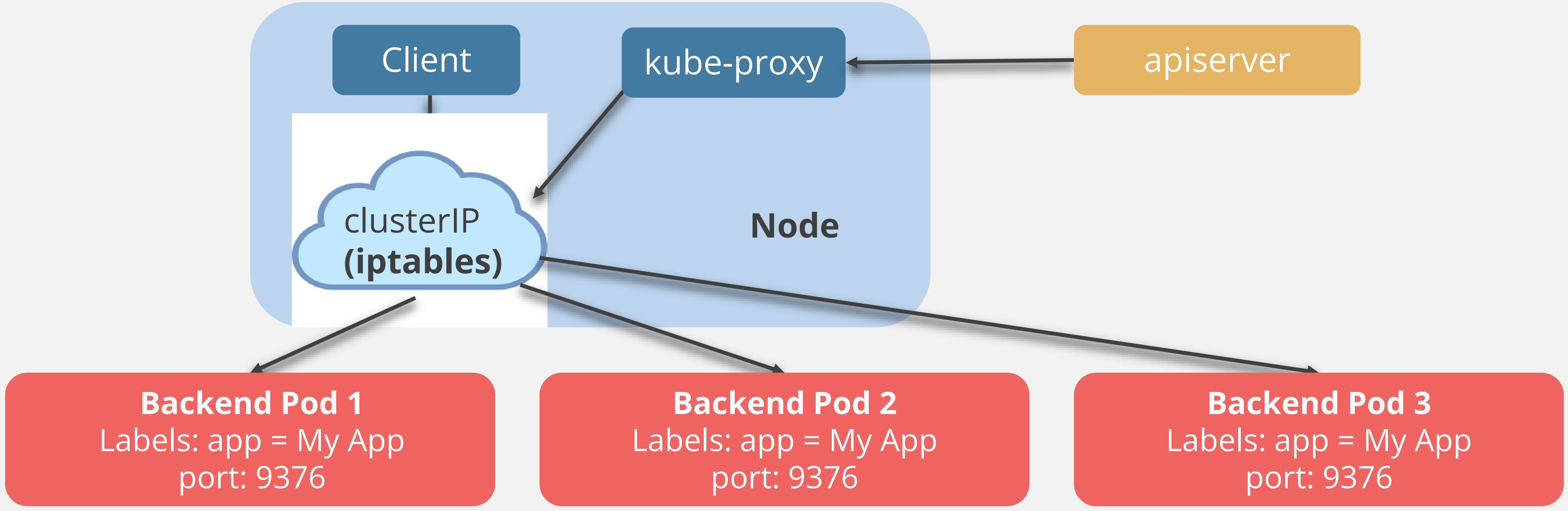
In this (legacy) mode, kube-proxy watches the Kubernetes control plane for the addition and removal of service and endpoint objects. By default, kube-proxy in userspace mode chooses a backend via a round-robin algorithm.



Services Overview Diagram for User Space Proxy

iptables Proxy Mode

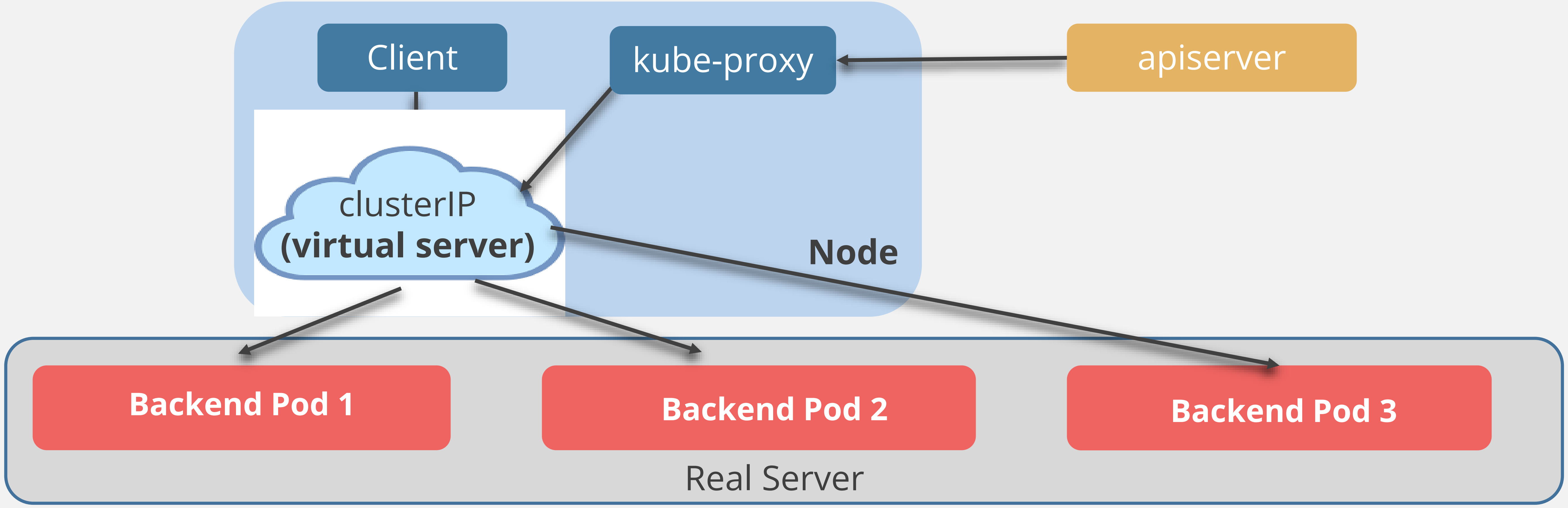
In this mode, kube-proxy watches the Kubernetes control plane for the addition and removal of service and endpoint objects. By default, kube-proxy in iptables mode chooses a backend at random.



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

Assisted Practice

Deploying a Multi-Port Service Pod **Duration: 10 Min.**

Problem statement:

You have been asked to deploy a Kubernetes pod using a multi-port service for accessing the deployment through multiple ports.

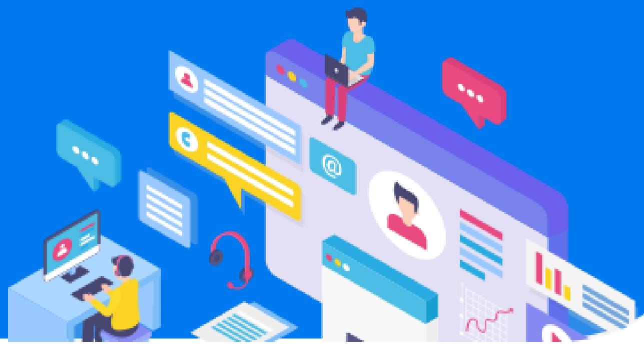
Outcome:

By the end of this demo, you will be able to deploy a Kubernetes pod that effectively utilizes a multi-port service for accessing the deployment through multiple ports.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines



Steps to be followed:

1. Create a deployment
2. Define a service
3. Access the deployment from multiple ports

Quick Check

You are deploying a microservices-based application in Kubernetes where different containers need to communicate with each other. Which Kubernetes resource would you use to connect a set of pods that provide the same functionality, ensuring consistent communication within the cluster?

A. ConfigMap

B. PersistentVolume

C. Service

D. Deployment

Topology

Topology-Aware Traffic Routing

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.

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.

The traffic will be directed to the node whose first label value matches the originating node's value.

A second label will be considered when there is no backend for the service on a matching node.

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

Constraints

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

Quick Check

You are deploying a multi-region application and want to ensure that traffic from clients in a specific region is routed to pods running in the same region to reduce latency. Which feature of Kubernetes would you use to implement this?

A. NodeSelector

B. Topology-aware traffic routing

C. Horizontal pod autoscaler

D. ServiceAccount

DNS for Services and Pods

Introduction to DNS

Kubernetes creates DNS records for services and pods.

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.

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.

DNS Records

The following objects get DNS records:

- Services
- Pods

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 the 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:

- Default
- ClusterFirst
- ClusterFirstWithHostNet
- None

Pod's DNS Policy

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

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

Pod's DNS Config

Pod's DNS config allows users to have more control over 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 of a pod with custom DNS settings:

```yaml
apiVersion: v1
Kind: Pod
metadata:
    namespace: default
    name: dns-example
spec:
  containers:
      - name: test
        image: nginx
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
          - 1.2.3.4
    searches:
        - ns1.svc.cluater-domain.example
        - my.dns.search.suffix
    options:
        - name: ndots
          value: "2"
        - name: edns0
```

Assisted Practice

Configuring DNS for Kubernetes Services and Pods **Duration: 10 Min.**

Problem statement:

You have been asked to configure the Domain Name System (DNS) for Kubernetes services and pods to ensure proper network resolution and connectivity.

Outcome:

By the end of this demo, you will be able to configure the Domain Name System (DNS) for Kubernetes services and pods to ensure proper network resolution and connectivity.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Determine the default DNS in the cluster
2. Execute DNS query
3. Configure the DNS policy
4. Create a custom DNS configuration

ASSISTED PRACTICE

Quick Check

You have deployed a set of pods in a Kubernetes cluster, and you need to ensure they can resolve each other's DNS names to communicate efficiently. How does Kubernetes provide DNS resolution for services and pods?

A. By creating a NodePort for each pod

B. By configuring a DNS entry for each pod and service automatically

C. By using an Ingress controller to resolve pod names

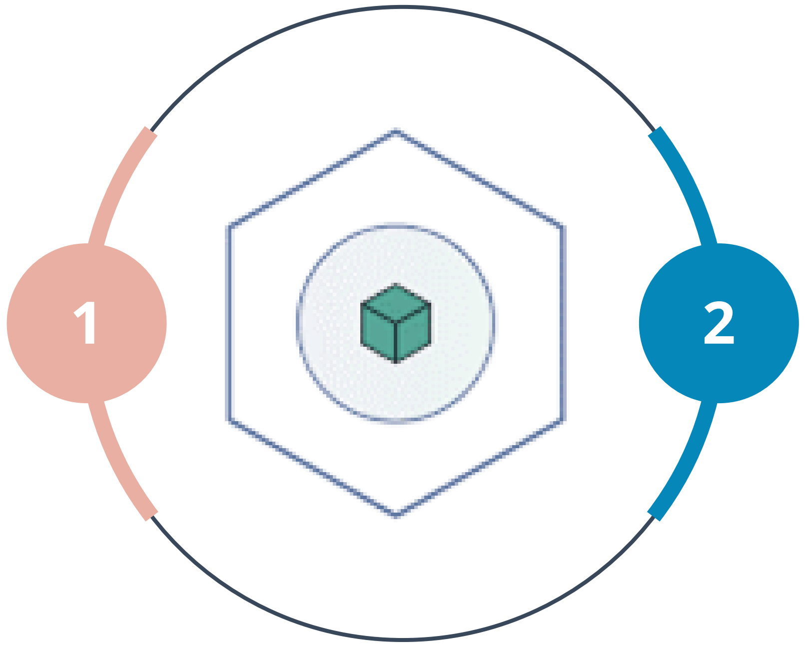
D. By assigning a unique external IP to each pod

EndpointSlices

EndpointSlices

EndpointSlices provides a simple way to track network endpoints within a Kubernetes cluster.

They 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:

```
apiVersion: discovery.k8s.io/v1
kind: EndpointSlice
metadata:
   name: example-abc
   labels:
     kubernetes>io/service-name: example
   addressType: IPv4
   ports:
     - name: http
       protocol: TCP
       port: 80
   endpoints :
     - addresses:
         - "10.1.2.3"
     conditions:
       ready: true
     host name : pod-1
     nodeName: node-1
     zone: us-west2-a
```

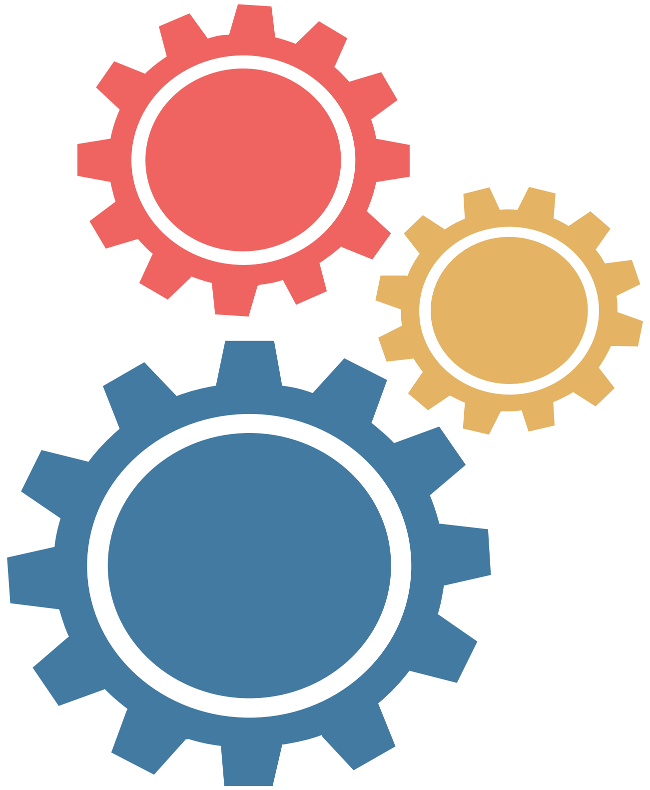
Address Types and Conditions

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



Serving

A condition that does not account for terminating states

Terminating

A condition that indicates whether an endpoint is terminating

Topology Information, Management, and Ownership

Topology enables a service to route traffic based on the node topology of the cluster.

Information

Topology information includes the location of the endpoint and information about the corresponding node and zone.

Management

EndpointSlice objects are often created and managed by the control plane (particularly, the EndpointSlice controller).

Ownership

EndpointSlices are owned by the service that the Endpointslice object tracks endpoints for.

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 a **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. Iterate through EndpointSlices that have been modified in the first step and fill them up with any new endpoints needed
3. If there are new endpoints left to add, try to fit them into a previously unchanged slice or create new ones or both

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.

Assisted Practice

Configuring EndpointSlice **Duration: 10 Min.**

Problem statement:
You have been asked to configure EndpointSlice to track network endpoints within a cluster.

Outcome:
By the end of this demo, you will be able to configure EndpointSlice to efficiently track network endpoints within a Kubernetes cluster.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a deployment and identify its EndpointSlice
2. Create a YAML file for custom EndpointSlice configuration
3. Create a resource for the custom EndpointSlice configuration

ASSISTED PRACTICE

Quick Check

You are scaling a Kubernetes Service to hundreds of pods and want to improve the scalability and performance of service discovery. Which feature would you use to group network endpoints more efficiently?

A. Ingress

B. NetworkPolicies

C. EndpointSlices

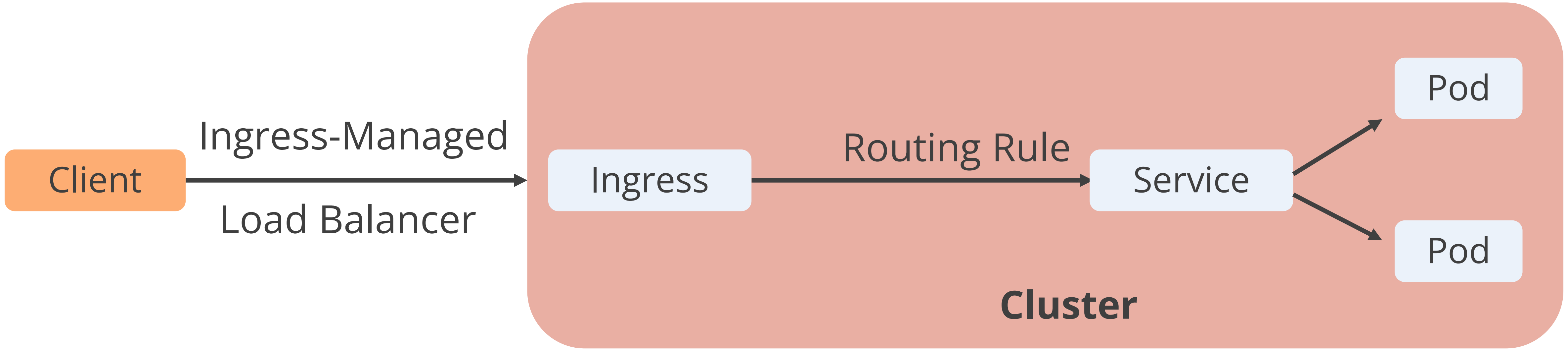
D. Horizontal pod autoscaler

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, load balance traffic, terminate SSL/TLS, and offer name-based virtual hosting.

Terminologies Used in Ingress

Ingress uses the following terminologies:

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

Ingress Resource

An Ingress needs **apiVersion**, **kind**, **metadata** and **spec** fields.

Example:

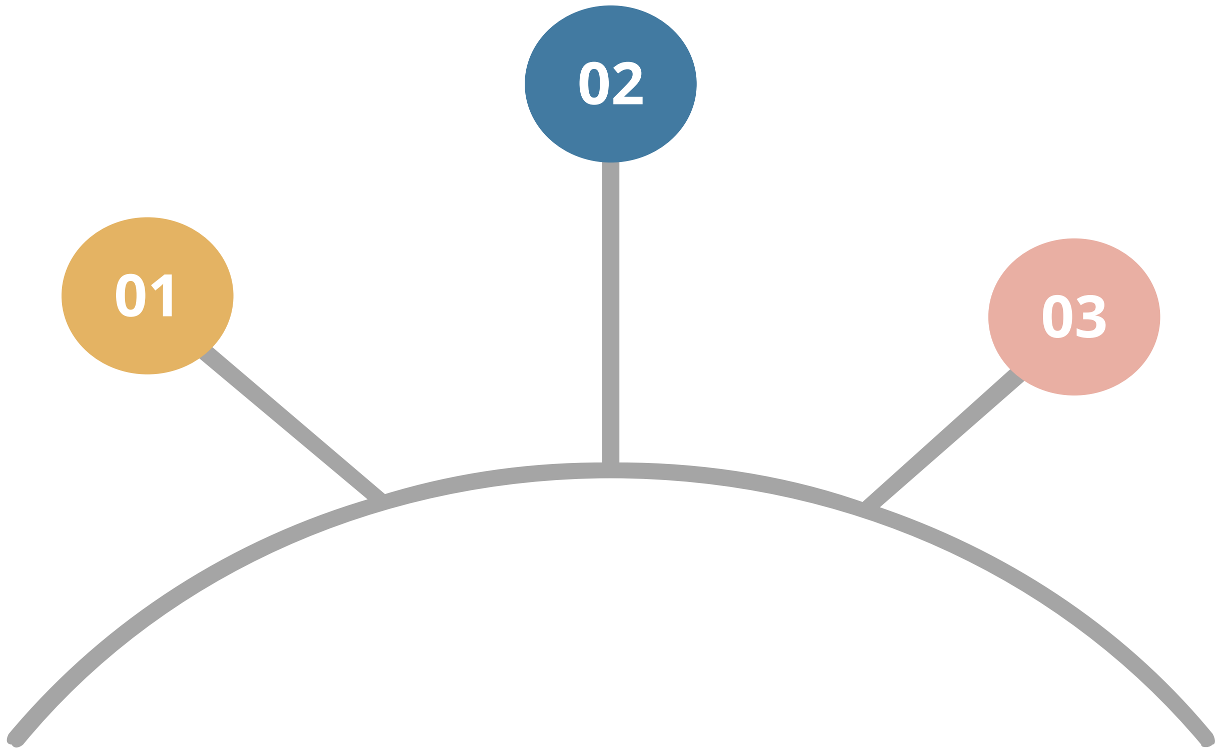
```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx-example
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: test
            port:
              number: 80
```

Ingress Rules

Each HTTP rule contains the following information:

An optional host

A backend



A list of paths

Default and Resource Backend

Default backend

The **defaultBackend** is conventionally a configuration option of the Ingress controller and is not specified in the Ingress resources.

Resource backend

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

Default Backend

An Ingress with no rules sends all traffic to a single default backend and is the backend that should handle requests.

Here are the configurations to be set for backend:

- If no **.spec.rules** are specified, **.spec.defaultBackend** must be specified.
- If **defaultBackend** is not set, the handling of requests that do not match any of the rules will go to the ingress controller.
- If the feature gate **MixedProtocolLBService** is enabled for the **kube-apiserver**, it can use different protocols when multiple ports are specified.

Resource Backend

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

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-resource-backend
spec:
  defaultBackend:
    resource:
      apiGroup: k8s.example.com
      kind: StorageBucket
      name: static-assets
  rules:
    - http:
        paths:
          - path: /icons
            pathType: ImplementationSpecific
            backend:
              resource:
                apiGroup: k8s.example.com
                kind: StorageBucket
                name: icon-assets
```

View Ingress

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

```
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

Matches with the **IngressClass**

Exact

Matches the URL path exactly along 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

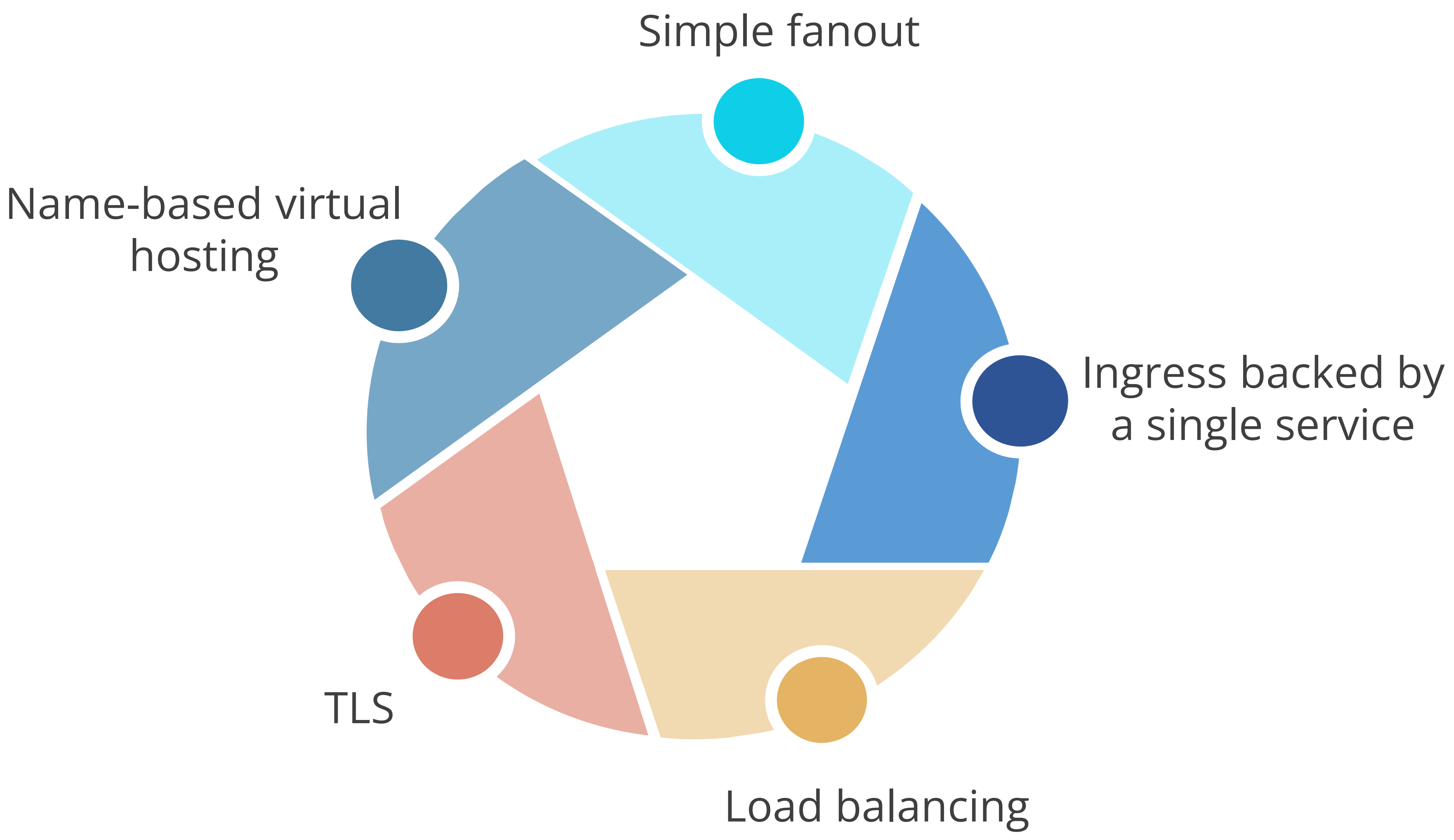
An example of the configuration file showing the use of wildcards:

```
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
```

Types of Ingress

In Kubernetes, there are five different types of Ingress:



Ingress Backed by a Single Service

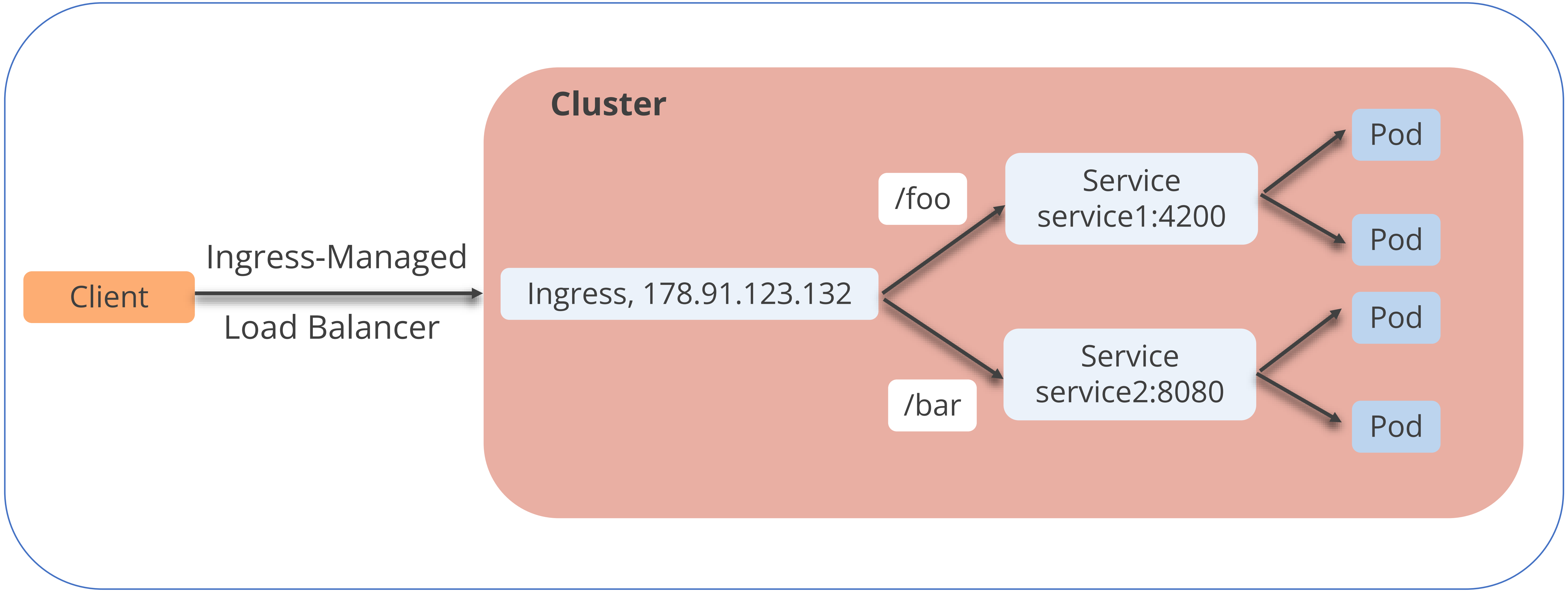
A single service can be exposed using an Ingress by specifying a default backend with no rules.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
spec:
  defaultBackend:
    service:
      name: test
      port:
        number: 80
```

```
#Command to 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

A fanout would require an Ingress such as:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: simple-fanout-example
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        pathType: Prefix
        backend:
          service:
            name: service1
            port:
              number: 4200
      - path: /bar
        pathType: Prefix
        backend:
          service:
            name: service2
            port:
              number: 8080
```

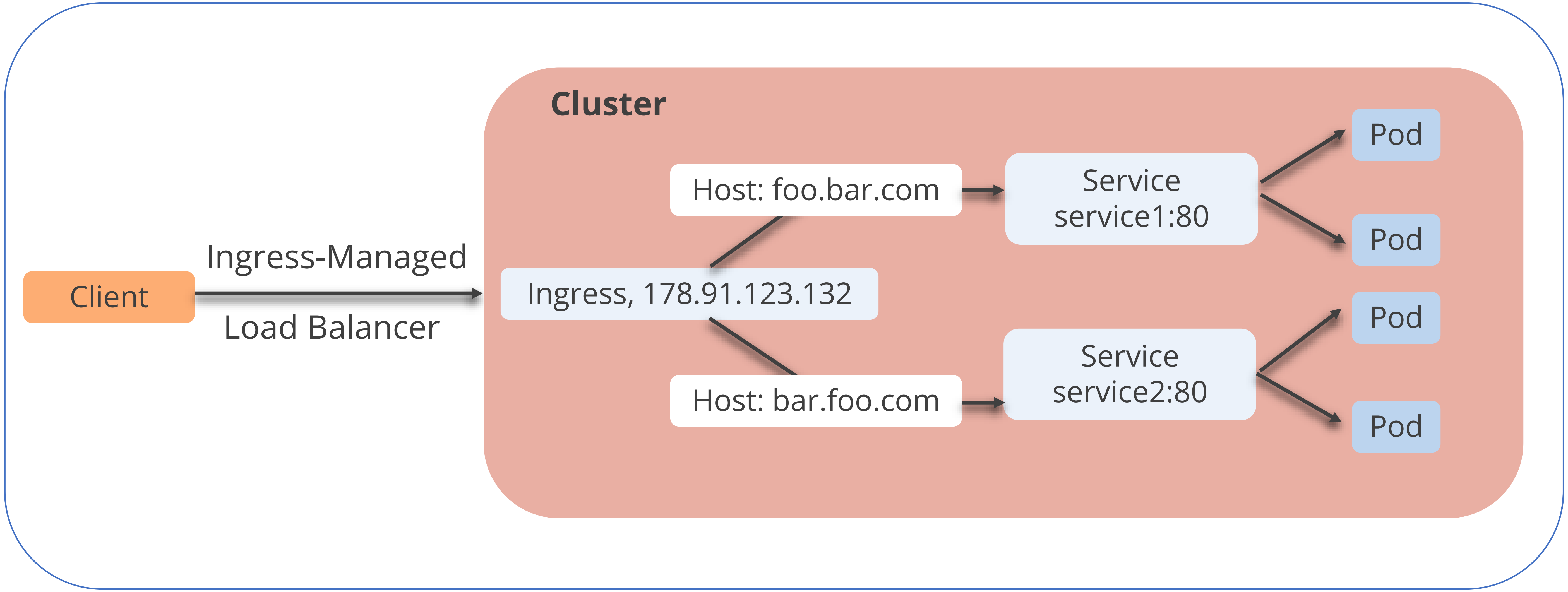
Simple Fanout

The command to describe the simple fanout Ingress is as follows:

```
Command:
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

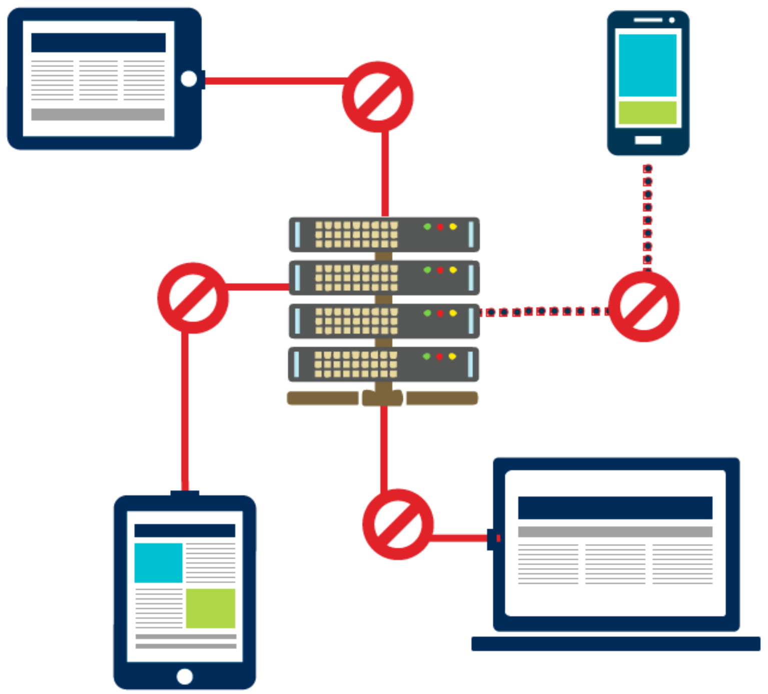
The Ingress shown below tells the backing load balancer to route requests based on the host header:

```
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
```

TLS

Transport Layer Security (TLS) is a cryptographic protocol designed to provide communications security over a computer network.



TLS

An Ingress can be secured using the configuration shown below:

```
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
```

The **tls.crt** and **tls.key** contain the certificate and private key to be used for TLS.

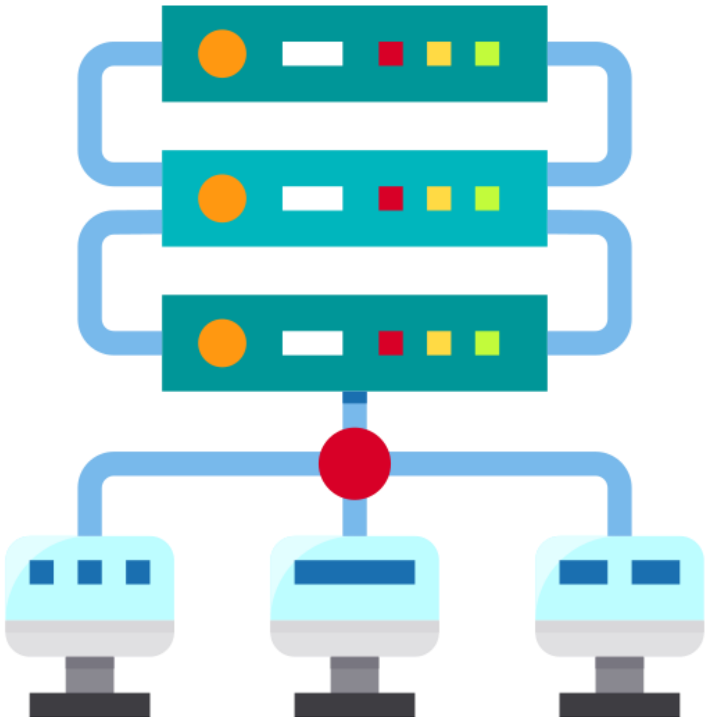
TLS

One must 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.

```yaml
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 as shown:

```
Command:
kubectl describe ingress test

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

```
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
```

Quick Check

You are configuring external access to multiple services running in a Kubernetes cluster using a single IP address. You want to route requests to different services based on URL paths. Which Kubernetes resource should you configure to achieve this?

A. Service

B. Ingress

C. NodePort

D. PersistentVolume

Ingress Controllers

Overview

For an Ingress resource to function, the cluster must have an active Ingress controller running.

Kubernetes as a project supports and maintains three controllers:

AWS Ingress Controllers

Nginx Ingress Controllers

GCE Ingress Controllers

Additional Controllers

These are the third-party controllers that give Kubernetes the features it requires:

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. Citrix Ingress Controller
6. Contour
7. EnRoute
8. Gloo, an open-source Ingress Controller based on Envoy

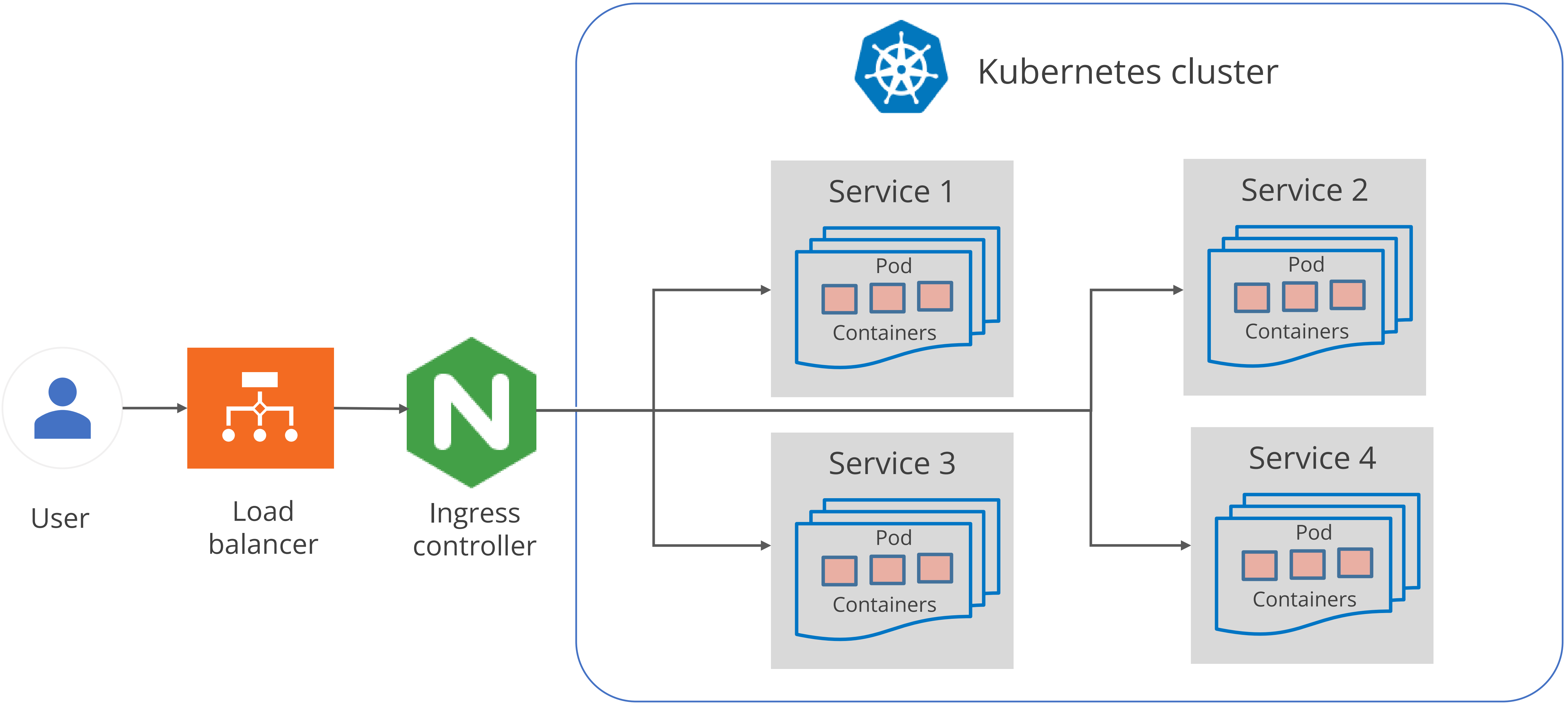
Additional Controllers

These are the third-party controllers that give Kubernetes the features it requires:

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

Using Ingress Controllers and Rules to Manage Network Traffic

The load balancer, as shown below, handles traffic distribution over several clusters, while the clusters contain Ingress controllers to ensure equal distribution to the services.



Using Ingress Controllers and Rules to Manage Network Traffic

Users can deploy any number of **Ingress controllers** using **IngressClass** within a cluster.

- When creating an Ingress, the **ingressClassName** field on the Ingress object must be specified using **.metadata.name**.
- If an Ingress does not have an **IngressClass** specified and the cluster has precisely one **IngressClass** marked as default, Kubernetes applies the cluster's default **IngressClass** to the Ingress.
- By adding the text value **true** to the **ingressclass.kubernetes.io/is-default-class** annotation on the **IngressClass**, it can be declared as default.

Using Ingress Controllers and Rules to Manage Network Traffic

Ingress controller features:

- Accept traffic from outside the Kubernetes platform and distribute it to pods within the platform
- Handle **Ingress** traffic within a cluster for services that need to connect with other services outside of the cluster
- Deploy objects called **Ingress Resources** via the Kubernetes API
- Monitor the pods running in Kubernetes and automatically adjust the loadbalancing rules when pods are added or withdrawn from a service

Assisted Practice

Setting up an Ingress Controller with Transport Layer Security **Duration: 10 Min.**

Problem statement:

You have been asked to implement the transport layer security by deploying an Ingress rule to generate an SSL certificate.

Outcome:

By the end of this demo, you will be able to implement transport layer security by deploying an Ingress rule to generate an SSL certificate effectively.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Deploy Ingress
2. Deploy HTTPD and OpenShift
3. Generate a self-signed SSL certificate and a TLS certificate
4. Verify the Ingress rule

ASSISTED PRACTICE

Network Policies

Network Policies

Network policies are implemented by the network plugin.

1

To use network policies, one must use a networking solution that supports network policy.

2

If a network policy resource is created without a controller that implements, it will have no effect.

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) selects 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

An example of **NetworkPolicy** might look like this:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podselector:
    matchLabels:
      role: db
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - ipBlock:
            cidr: 172.17.0.0/16
            except:
              - 172.17.1.0/24
    ...
```

```yaml
           - namespaceSelector:
               matchLabels:
                 project: myproject
           - podSelector:
               matchLabels:
                 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

There are four kinds of selectors that can be specified in an **ingress from** section or **egress to** section:

namespaceSelector and **podSelector**

A single **to** or **from** entry, which specifies both **namespaceSelector** and **PodSelector**, selects specific **pods** within particular **namespaces**.

podSelector

This selects specific **pods** in the same namespace as the **NetworkPolicy,** which should be allowed as ingress sources or egress destinations.

Behavior of To and From Selectors

There are four kinds of selectors that can be specified in an **ingress from** section or **egress to** section:

namespaceSelector

This selects particular **namespaces** for which all **pods** should be allowed as ingress sources or egress destinations.

ipBlock

This selects particular IP CIDR ranges to allow as ingress sources or egress destinations.

Behavior of To and From Selectors

A single **from** element in **namespaceSelector** and **podSelector** allows connections from pods with the label **role=client** in namespaces with the label **user=alice**.

```
...
    ingress:
    - from:
     namespaceSelector:
        matchLables:
         user: alice
         podSelector:
       matchLabels:
         role: client
...
```

Default Policies

By default, if no policies exist in a namespace, all ingress and egress traffic is allowed to and from pods in that namespace.

```
...
   apiVersion: networking.k8s.10/v1
   kind:  NetworkPolicy
   metadata:
      name:  default-deny-ingress
      spec:
          podselector:  {}
          policyTypes:
          - Ingress
```

Default **deny all** Ingress traffic

```
...
   apiVersion: networking.k8s.10/v1
   kind:  NetworkPolicy
   metadata:
      name:  allow-all-ingress
   spec:
      podselector:  {}
      ingress:
       - {}
       policyTypes:
       - Ingress
```

Default **allow all** Ingress traffic

Default Policies

By default, if no policies exist in a namespace, all ingress and egress traffic is allowed to and from pods in that namespace.

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

Default **deny all** Egress traffic

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

Default **allow all** Egress traffic

```
...
    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

Adding Entries to Pod /etc/hosts with HostAliases

Default Hosts File Content

Adding entries to a pod's **/etc/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:

```
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

Command Examine a Pod IP:
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

The following is an example of the hosts file content:

```
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
```

By default, the **hosts** file only includes IPv4 and IPv6 boilerplates like **localhost** and its own hostname.

Additional Entries with HostAliases

In addition to the default boilerplate, more entries can be made to the **hosts** file.

```
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"
```

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**.

Additional Entries with HostAliases

A pod can be configured by running the following command:

```
Command:
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
```

Additional Entries with HostAliases

The **hosts** file content looks like this:

```
Command:
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.
```

IPv4/IPv6 Dual-Stack

Dual-stacking refers to the ability of the devices to run IPv4 and IPv6 simultaneously.

IPv4/IPv6 dual-stack networking enables the allocation of both IPv4 and IPv6 addresses to pods and Services.

1

IPv4/IPv6 dual-stack networking is enabled by default for the Kubernetes, allowing simultaneous assignment of both IPv4 and IPv6 addresses.

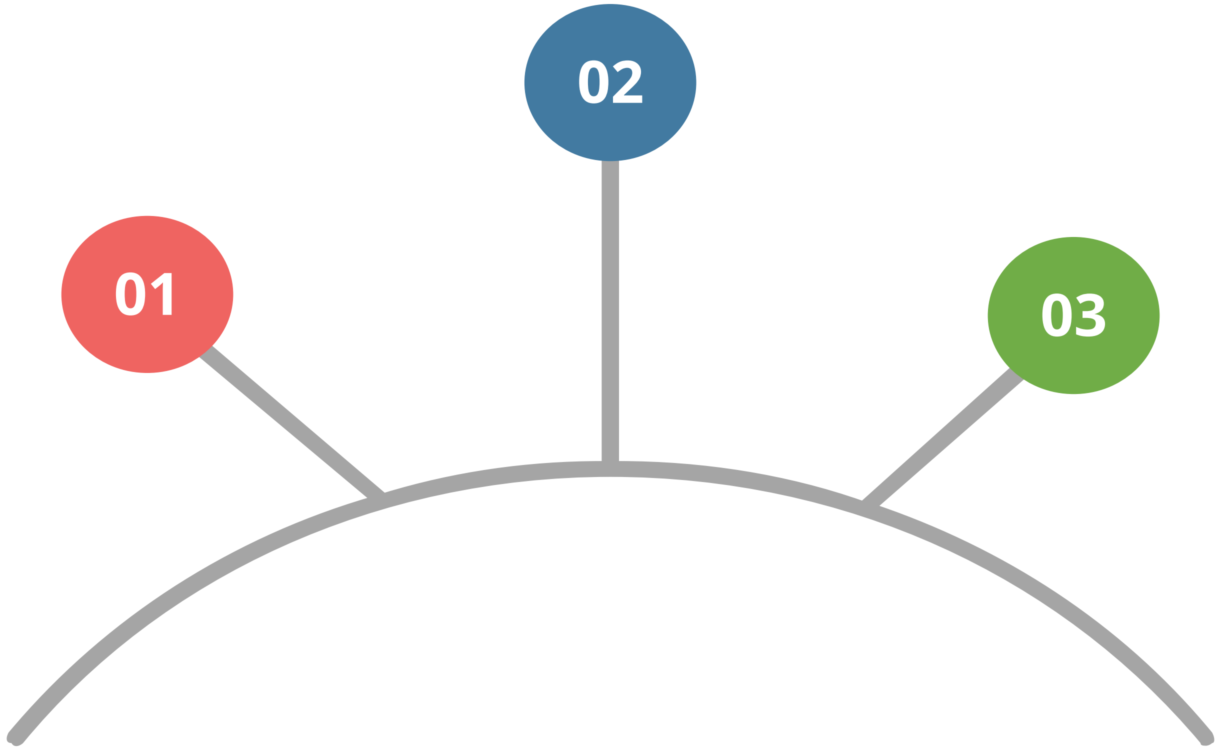
2

IPv4/IPv6 Dual-Stack: 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

IPv4/IPv6 Dual-Stack: Prerequisites

There are three prerequisites for utilizing IPv4/IPv6 dual-stack Kubernetes clusters:

01 Kubernetes v1.20 or later

02 Provider support for dual-stack networking

03 Network plugin that supports dual-stack

IPv4/IPv6 Dual-Stack: Services

When defining a service, it can optionally be configured as dual-stack by using the **.spec.ipFamilyPolicy** field.

It takes one of the following values:

PreferDualStack

RequireDualStack

SingleStack

Dual-Stack Options on New Services

This service specification does not define **.spec.ipFamilyPolicy** explicitly.

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

When the service is created, Kubernetes assigns a cluster IP from the first configured **service-cluster-ip-range** and sets the **.spec.ipFamilyPolicy** to **SingleStack**.

Dual-Stack Options on New Services

The service specification shown below explicitly defines **PreferDualStack** in **.spec.ipFamilyPolicy**:

```
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**:

```
apiVersion: v1
kind:  service
metadata:
   name:  my-service
   labels:
     app: MyApp
  spec:
   ipFamilyPolicy: PreferDualStack
   ipFamilies:
   - IPv6
   - 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:

```yaml
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 the following command:
  kubectl get svc my-service -o yaml
```

```yaml
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 from a pod that utilizes non-publicly routable IPv6 addresses to reach off-cluster destinations (such as the public Internet), users must configure 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.

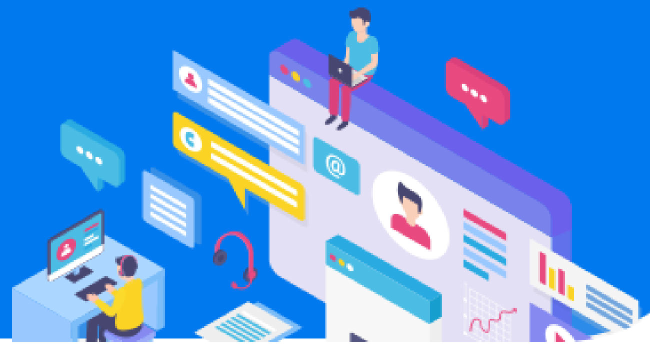
Before:

```
spec:
ipFamilyPoilcy SingleStack
```

After:

```
spec:
ipFamilyPoilcy PreferDualStack
```

Assisted Practice



Blocking All Traffic from an Application **Duration: 10 Min.**

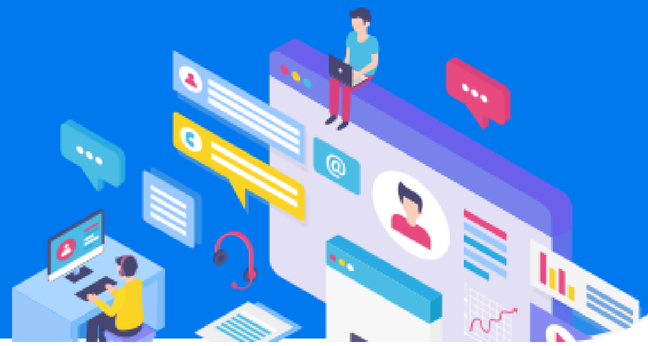
Problem statement:
You have been asked to effectively block all the network traffic from a specific application to ensure enhanced security.

Outcome:
By the end of this demo, you will be able to effectively block all network traffic from a specific application to ensure enhanced security.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines



Steps to be followed:

1. Set up the application pod and policy
2. Verify the network policy

ASSISTED PRACTICE

Assisted Practice

Limiting the Traffic to an Application

Duration: 10 Min.

Problem statement:

You have been asked to limit all network traffic to a specific application within a Kubernetes cluster for controlled access and resource utilization.

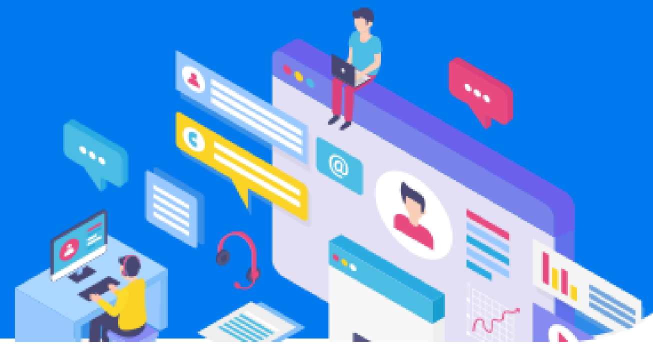
Outcome:

By the end of this demo, you will be able to limit network traffic to a specific application within a Kubernetes cluster for controlled access and optimized resource utilization.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines



Steps to be followed:

1. Launch the API server
2. Configure the YAML file for the network policy
3. Verify the network policy
4. Clear the created resources

ASSISTED PRACTICE

Assisted Practice

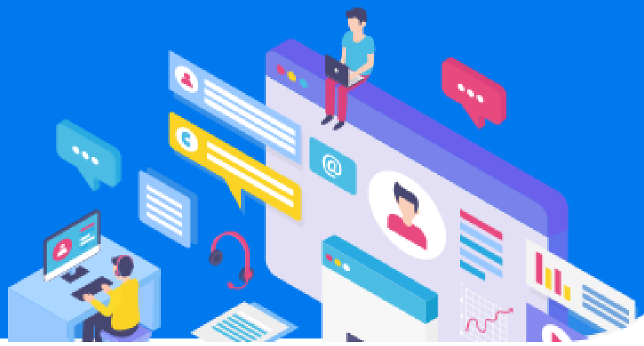
Blocking All the Traffic from Other Namespaces **Duration: 10 Min.**

Problem statement:
You have been asked to implement a Kubernetes network policy for blocking all inter-namespace traffic.

Outcome:
By the end of this demo, you will be able to implement a Kubernetes network policy to block all inter-namespace traffic effectively.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines



Steps to be followed:

1. Launch the web service
2. Configure the YAML file for the network policy
3. Create a new namespace
4. Verify the network policy
5. Clear the created resources

ASSISTED PRACTICE

Key Takeaways

- The 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 allow to specify how a pod is allowed to communicate with various network entities over the network.
- IPv4/IPv6 dual-stack networking supports IPv4 and IPv6 addresses to be allocated to pods and services.