Service

Expose an application running in your cluster behind a single outward-facing endpoint, even when the workload is split across multiple backends.

In Kubernetes, a Service is a method for exposing a network application that is running as one or more Pods in your cluster.

A key aim of Services in Kubernetes is that you don't need to modify your existing application to use an unfamiliar service discovery mechanism. You can run code in Pods, whether this is a code designed for a cloud-native world, or an older app you've containerized. You use a Service to make that set of Pods available on the network so that clients can interact with it.

If you use a Deployment to run your app, that Deployment can create and destroy Pods dynamically. From one moment to the next, you don't know how many of those Pods are working and healthy; you might not even know what those healthy Pods are named. Kubernetes Pods are created and destroyed to match the desired state of your cluster. Pods are ephemeral resources (you should not expect that an individual Pod is reliable and durable).

Each Pod gets its own IP address (Kubernetes expects network plugins to ensure this). For a given Deployment in your cluster, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later.

This leads to a problem: if some set of Pods (call them "backends") provides functionality to other Pods (call them "frontends") inside your cluster, how do the frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload?

Enter *Services*.

Services in Kubernetes

The Service API, part of Kubernetes, is an abstraction to help you expose groups of Pods over a network. Each Service object defines a logical set of endpoints (usually these endpoints are Pods) along with a policy about how to make those pods accessible.

For example, consider a stateless image-processing backend which is running with 3 replicas. Those replicas are fungible—frontends do not care which backend they use. While the actual Pods that compose the backend set may change, the frontend clients should not need to be aware of that, nor should they need to keep track of the set of backends themselves.

The Service abstraction enables this decoupling.

The set of Pods targeted by a Service is usually determined by a selector that you define. To learn about other ways to define Service endpoints, see Services *without* selectors.

If your workload speaks HTTP, you might choose to use an Ingress to control how web traffic reaches that workload. Ingress is not a Service type, but it acts as the entry point for your cluster. An Ingress lets you consolidate your routing rules into a single resource, so that you can expose multiple components of your workload, running separately in your cluster, behind a single listener.

The Gateway API for Kubernetes provides extra capabilities beyond Ingress and Service. You can add Gateway to your cluster - it is a family of extension APIs, implemented using CustomResourceDefinitions - and then use these to configure access to network services that are running in your cluster.

Cloud-native service discovery

If you're able to use Kubernetes APIs for service discovery in your application, you can query the API server for matching EndpointSlices. Kubernetes updates the EndpointSlices for a Service whenever the set of Pods in a Service changes.

For non-native applications, Kubernetes offers ways to place a network port or load balancer in between your application and the backend Pods.

Either way, your workload can use these service discovery mechanisms to find the target it wants to connect to.

Defining a Service

A Service is an object (the same way that a Pod or a ConfigMap is an object). You can create, view or modify Service definitions using the Kubernetes API. Usually you use a tool such as `kubectl` to make those API calls for you.

For example, suppose you have a set of Pods that each listen on TCP port 9376 and are labelled as `app.kubernetes.io/name=MyApp`. You can define a Service to publish that TCP listener:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Applying this manifest creates a new Service named "my-service" with the default ClusterIP service type. The Service targets TCP port 9376 on any Pod with the `app.kubernetes.io/name: MyApp` label.

Kubernetes assigns this Service an IP address (the *cluster IP*), that is used by the virtual IP address mechanism. For more details on that mechanism, read Virtual IPs and Service Proxies.

The controller for that Service continuously scans for Pods that match its selector, and then makes any necessary updates to the set of EndpointSlices for the Service.

The name of a Service object must be a valid RFC 1035 label name.

> **Note:** A Service can map *any* incoming `port` to a `targetPort`. By default and for convenience, the `targetPort` is set to the same value as the `port` field.

Port definitions

Port definitions in Pods have names, and you can reference these names in the `targetPort` attribute of a Service. For example, we can bind the `targetPort` of the Service to the Pod port in the following way:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app.kubernetes.io/name: proxy
spec:
  containers:
  - name: nginx
    image: nginx:stable
    ports:
      - containerPort: 80
        name: http-web-svc

---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app.kubernetes.io/name: proxy
  ports:
  - name: name-of-service-port
    protocol: TCP
    port: 80
    targetPort: http-web-svc
```

This works even if there is a mixture of Pods in the Service using a single configured name, with the same network protocol available via different port numbers. This offers a lot of flexibility for deploying and evolving your Services. For example, you can change the port numbers that Pods expose in the next version of your backend software, without breaking clients.

The default protocol for Services is TCP; you can also use any other supported protocol.

Because many Services need to expose more than one port, Kubernetes supports multiple port definitions for a single Service. Each port definition can have the same `protocol`, or a different one.

Services without selectors

Services most commonly abstract access to Kubernetes Pods thanks to the selector, but when used with a corresponding set of EndpointSlices objects and without a selector, the Service can abstract other kinds of backends, including ones that run outside the cluster.

For example:

- You want to have an external database cluster in production, but in your test environment you use your own databases.
- You want to point your Service to a Service in a different Namespace or on another cluster.
- You are migrating a workload to Kubernetes. While evaluating the approach, you run only a portion of your backends in Kubernetes.

In any of these scenarios you can define a Service *without* specifying a selector to match Pods. For example:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Because this Service has no selector, the corresponding EndpointSlice (and legacy Endpoints) objects are not created automatically. You can map the Service to the network address and port where it's running, by adding an EndpointSlice object manually. For example:

```yaml
apiVersion: discovery.k8s.io/v1
kind: EndpointSlice
metadata:
  name: my-service-1 # by convention, use the name of the Service
                     # as a prefix for the name of the EndpointSlice
  labels:
    # You should set the "kubernetes.io/service-name" label.
    # Set its value to match the name of the Service
    kubernetes.io/service-name: my-service
addressType: IPv4
ports:
  - name: '' # empty because port 9376 is not assigned as a well-known
             # port (by IANA)
    appProtocol: http
    protocol: TCP
    port: 9376
endpoints:
  - addresses:
      - "10.4.5.6"
  - addresses:
      - "10.1.2.3"
```

Custom EndpointSlices

When you create an EndpointSlice object for a Service, you can use any name for the EndpointSlice. Each EndpointSlice in a namespace must have a unique name. You link an EndpointSlice to a Service by setting the `kubernetes.io/service-name` label on that EndpointSlice.

> **Note:**
> The endpoint IPs *must not* be: loopback (127.0.0.0/8 for IPv4, ::1/128 for IPv6), or link-local (169.254.0.0/16 and 224.0.0.0/24 for IPv4, fe80::/64 for IPv6).
>
> The endpoint IP addresses cannot be the cluster IPs of other Kubernetes Services, because kube-proxy doesn't support virtual IPs as a destination.

For an EndpointSlice that you create yourself, or in your own code, you should also pick a value to use for the label `endpointslice.kubernetes.io/managed-by`. If you create your own controller code to manage EndpointSlices, consider using a value similar to `"my-domain.example/name-of-controller"`. If you are using a third party tool, use the name of the tool in all-lowercase and change spaces and other punctuation to dashes (`-`). If people are directly using a tool such as `kubectl` to manage EndpointSlices, use a name that describes this manual management, such as `"staff"` or `"cluster-admins"`. You should avoid using the reserved value `"controller"`, which identifies EndpointSlices managed by Kubernetes' own control plane.

Accessing a Service without a selector

Accessing a Service without a selector works the same as if it had a selector. In the example for a Service without a selector, traffic is routed to one of the two endpoints defined in the EndpointSlice manifest: a TCP connection to 10.1.2.3 or 10.4.5.6, on port 9376.

> **Note:** The Kubernetes API server does not allow proxying to endpoints that are not mapped to pods. Actions such as `kubectl proxy <service-name>` where the service has no selector will fail due to this constraint. This prevents the Kubernetes API server from being used as a proxy to endpoints the caller may not be authorized to access.

An `ExternalName` Service is a special case of Service that does not have selectors and uses DNS names instead. For more information, see the ExternalName section.

EndpointSlices

FEATURE STATE: `Kubernetes v1.21 [stable]`

EndpointSlices are objects that represent a subset (a *slice*) of the backing network endpoints for a Service.

Your Kubernetes cluster tracks how many endpoints each EndpointSlice represents. If there are so many endpoints for a Service that a threshold is reached, then Kubernetes adds another empty EndpointSlice and stores new endpoint information there. By default, Kubernetes makes a new EndpointSlice once the existing EndpointSlices all contain at least 100 endpoints. Kubernetes does not make the new EndpointSlice until an extra endpoint needs to be added.

See EndpointSlices for more information about this API.

Endpoints

In the Kubernetes API, an Endpoints (the resource kind is plural) defines a list of network endpoints, typically referenced by a Service to define which Pods the traffic can be sent to.

The EndpointSlice API is the recommended replacement for Endpoints.

Over-capacity endpoints

Kubernetes limits the number of endpoints that can fit in a single Endpoints object. When there are over 1000 backing endpoints for a Service, Kubernetes truncates the data in the Endpoints object. Because a Service can be linked with more than one EndpointSlice, the 1000 backing endpoint limit only affects the legacy Endpoints API.

In that case, Kubernetes selects at most 1000 possible backend endpoints to store into the Endpoints object, and sets an annotation on the Endpoints: `endpoints.kubernetes.io/over-capacity: truncated` . The control plane also removes that annotation if the number of backend Pods drops below 1000.

Traffic is still sent to backends, but any load balancing mechanism that relies on the legacy Endpoints API only sends traffic to at most 1000 of the available backing endpoints.

The same API limit means that you cannot manually update an Endpoints to have more than 1000 endpoints.

Application protocol

FEATURE STATE: `Kubernetes v1.20 [stable]`

The `appProtocol` field provides a way to specify an application protocol for each Service port. This is used as a hint for implementations to offer richer behavior for protocols that they understand. The value of this field is mirrored by the corresponding Endpoints and EndpointSlice objects.

This field follows standard Kubernetes label syntax. Valid values are one of:

- IANA standard service names.
- Implementation-defined prefixed names such as `mycompany.com/my-custom-protocol`.
- Kubernetes-defined prefixed names:

Protocol	Description
`kubernetes.io/h2c`	HTTP/2 over cleartext as described in RFC 7540

Multi-port Services

For some Services, you need to expose more than one port. Kubernetes lets you configure multiple port definitions on a Service object. When using multiple ports for a Service, you must give all of your ports names so that these are unambiguous. For example:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 9376
    - name: https
      protocol: TCP
      port: 443
      targetPort: 9377
```

Note:
As with Kubernetes names in general, names for ports must only contain lowercase alphanumeric characters and `-`. Port names must also start and end with an alphanumeric character.

For example, the names `123-abc` and `web` are valid, but `123_abc` and `-web` are not.

Service type

For some parts of your application (for example, frontends) you may want to expose a Service onto an external IP address, one that's accessible from outside of your cluster.

Kubernetes Service types allow you to specify what kind of Service you want.

The available `type` values and their behaviors are:

`ClusterIP`

Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default that is used if you don't explicitly specify a `type` for a Service. You can expose the Service to the public internet using an Ingress or a Gateway.

`NodePort`

Exposes the Service on each Node's IP at a static port (the `NodePort`). To make the node port available, Kubernetes sets up a cluster IP address, the same as if you had requested a Service of `type: ClusterIP`.

`LoadBalancer`

Exposes the Service externally using an external load balancer. Kubernetes does not directly offer a load balancing component; you must provide one, or you can integrate your Kubernetes cluster with a cloud provider.

`ExternalName`

Maps the Service to the contents of the `externalName` field (for example, to the hostname `api.foo.bar.example`). The mapping configures your cluster's DNS server to return a `CNAME` record with that external hostname value. No proxying of any kind is set up.

The `type` field in the Service API is designed as nested functionality - each level adds to the previous. However there is an exception to this nested design. You can define a `LoadBalancer` Service by disabling the load balancer `NodePort` allocation.

`type: ClusterIP`

This default Service type assigns an IP address from a pool of IP addresses that your cluster has reserved for that purpose.

Several of the other types for Service build on the `ClusterIP` type as a foundation.

If you define a Service that has the `.spec.clusterIP` set to `"None"` then Kubernetes does not assign an IP address. See headless Services for more information.

Choosing your own IP address

You can specify your own cluster IP address as part of a `Service` creation request. To do this, set the `.spec.clusterIP` field. For example, if you already have an existing DNS entry that you wish to reuse, or legacy systems that are configured for a specific IP address and difficult to re-configure.

The IP address that you choose must be a valid IPv4 or IPv6 address from within the `service-cluster-ip-range` CIDR range that is configured for the API server. If you try to create a Service with an invalid `clusterIP` address value, the API server will return a 422 HTTP status code to indicate that there's a problem.

Read avoiding collisions to learn how Kubernetes helps reduce the risk and impact of two different Services both trying to use the same IP address.

`type: NodePort`

If you set the `type` field to `NodePort`, the Kubernetes control plane allocates a port from a range specified by `--service-node-port-range` flag (default: 30000-32767). Each node proxies that port (the same port number on every Node) into your Service. Your Service reports the allocated port in its `.spec.ports[*].nodePort` field.

Using a NodePort gives you the freedom to set up your own load balancing solution, to configure environments that are not fully supported by Kubernetes, or even to expose one or more nodes' IP addresses directly.

For a node port Service, Kubernetes additionally allocates a port (TCP, UDP or SCTP to match the protocol of the Service). Every node in the cluster configures itself to listen on that assigned port and to forward traffic to one of the ready endpoints associated with that Service. You'll be able to contact the `type: NodePort` Service, from outside the cluster, by connecting to any node using the appropriate protocol (for example: TCP), and the appropriate port (as assigned to that Service).

Choosing your own port

If you want a specific port number, you can specify a value in the `nodePort` field. The control plane will either allocate you that port or report that the API transaction failed. This means that you need to take care of possible port collisions yourself. You also have to use a valid port number, one that's inside the range configured for NodePort use.

Here is an example manifest for a Service of `type: NodePort` that specifies a NodePort value (30007, in this example):

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - port: 80
      # By default and for convenience, the `targetPort` is set to
      # the same value as the `port` field.
      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
```

Reserve Nodeport Ranges to avoid collisions when port assigning

FEATURE STATE: `Kubernetes v1.28 [beta]`

The policy for assigning ports to NodePort services applies to both the auto-assignment and the manual assignment scenarios. When a user wants to create a NodePort service that uses a specific port, the target port may conflict with another port that has already been assigned. In this case, you can enable the feature gate `ServiceNodePortStaticSubrange`, which allows you to use a different port allocation strategy for NodePort Services. The port range for NodePort services is divided into two bands. Dynamic port assignment uses the upper band by default, and it may use the lower band once the upper band has been exhausted. Users can then allocate from the lower band with a lower risk of port collision.

Custom IP address configuration for `type: NodePort` Services

You can set up nodes in your cluster to use a particular IP address for serving node port services. You might want to do this if each node is connected to multiple networks (for example: one network for application traffic, and another network for traffic between nodes and the control plane).

If you want to specify particular IP address(es) to proxy the port, you can set the `--nodeport-addresses` flag for kube-proxy or the equivalent `nodePortAddresses` field of the kube-proxy configuration file to particular IP block(s).

This flag takes a comma-delimited list of IP blocks (e.g. `10.0.0.0/8`, `192.0.2.0/25`) to specify IP address ranges that kube-proxy should consider as local to this node.

For example, if you start kube-proxy with the `--nodeport-addresses=127.0.0.0/8` flag, kube-proxy only selects the loopback interface for NodePort Services. The default for `--nodeport-addresses` is an empty list. This means that kube-proxy should consider all available network interfaces for NodePort. (That's also compatible with earlier Kubernetes releases.)

> **Note:** This Service is visible as `<NodeIP>:spec.ports[*].nodePort` and `.spec.clusterIP:spec.ports[*].port`. If the `--nodeport-addresses` flag for kube-proxy or the equivalent field in the kube-proxy configuration file is set, `<NodeIP>` would be a filtered node IP address (or possibly IP addresses).

`type: LoadBalancer`

On cloud providers which support external load balancers, setting the `type` field to `LoadBalancer` provisions a load balancer for your Service. The actual creation of the load balancer happens asynchronously, and information about the provisioned balancer is published in the Service's `.status.loadBalancer` field. For example:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  clusterIP: 10.0.171.239
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
    - ip: 192.0.2.127
```

Traffic from the external load balancer is directed at the backend Pods. The cloud provider decides how it is load balanced.

To implement a Service of `type: LoadBalancer`, Kubernetes typically starts off by making the changes that are equivalent to you requesting a Service of `type: NodePort`. The cloud-controller-manager component then configures the external load balancer to forward traffic to that assigned node port.

You can configure a load balanced Service to omit assigning a node port, provided that the cloud provider implementation supports this.

Some cloud providers allow you to specify the `loadBalancerIP`. In those cases, the load-balancer is created with the user-specified `loadBalancerIP`. If the `loadBalancerIP` field is not specified, the load balancer is set up with an ephemeral IP address. If you specify a `loadBalancerIP` but your cloud provider does not support the feature, the `loadbalancerIP` field that you set is ignored.

> **Note:**
> The `.spec.loadBalancerIP` field for a Service was deprecated in Kubernetes v1.24.
>
> This field was under-specified and its meaning varies across implementations. It also cannot support dual-stack networking. This field may be removed in a future API version.
>
> If you're integrating with a provider that supports specifying the load balancer IP address(es) for a Service via a (provider specific) annotation, you should switch to doing that.
>
> If you are writing code for a load balancer integration with Kubernetes, avoid using this field. You can integrate with Gateway rather than Service, or you can define your own (provider specific) annotations on the Service that specify the equivalent detail.

Load balancers with mixed protocol types

FEATURE STATE: `Kubernetes v1.26 [stable]`

By default, for LoadBalancer type of Services, when there is more than one port defined, all ports must have the same protocol, and the protocol must be one which is supported by the cloud provider.

The feature gate `MixedProtocolLBService` (enabled by default for the kube-apiserver as of v1.24) allows the use of different protocols for LoadBalancer type of Services, when there is more than one port defined.

> **Note:** The set of protocols that can be used for load balanced Services is defined by your cloud provider; they may impose restrictions beyond what the Kubernetes API enforces.

Disabling load balancer NodePort allocation

FEATURE STATE: `Kubernetes v1.24 [stable]`

You can optionally disable node port allocation for a Service of `type: LoadBalancer`, by setting the field `spec.allocateLoadBalancerNodePorts` to `false`. This should only be used for load balancer implementations that route traffic directly to pods as opposed to using node ports. 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. You must explicitly remove the `nodePorts` entry in every Service port to de-allocate those node ports.

Specifying class of load balancer implementation

FEATURE STATE: `Kubernetes v1.24 [stable]`

For a Service with `type` set to `LoadBalancer`, the `.spec.loadBalancerClass` field enables you to use a load balancer implementation other than the cloud provider default.

By default, `.spec.loadBalancerClass` is not set and a `LoadBalancer` type of Service uses the cloud provider's default load balancer implementation if the cluster is configured with a cloud provider using the `--cloud-provider` component flag.

If you specify `.spec.loadBalancerClass`, it is assumed that a load balancer implementation that matches the specified class is watching for Services. Any default load balancer implementation (for example, the one provided by the cloud provider) will ignore Services that have this field set. `spec.loadBalancerClass` can be set on a Service of type `LoadBalancer` only. Once set, it cannot be changed. The value of `spec.loadBalancerClass` must be a label-style identifier, with an optional prefix such as "`internal-vip`" or "`example.com/internal-vip`". Unprefixed names are reserved for end-users.

Internal load balancer

In a mixed environment it is sometimes necessary to route traffic from Services inside the same (virtual) network address block.

In a split-horizon DNS environment you would need two Services to be able to route both external and internal traffic to your endpoints.

To set an internal load balancer, add one of the following annotations to your Service depending on the cloud service provider you're using:

Default | GCP | AWS | Azure | IBM Cloud | OpenStack | Baidu Cloud | Tencent Cloud | Alibaba Cloud | OCI

Select one of the tabs.

`type: ExternalName`

Services of type ExternalName map a Service to a DNS name, not to a typical selector such as `my-service` or `cassandra`. You specify these Services with the `spec.externalName` parameter.

This Service definition, for example, maps the `my-service` Service in the `prod` namespace to `my.database.example.com`:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

> **Note:**
> A Service of `type: ExternalName` accepts an IPv4 address string, but treats that string as a DNS name comprised of digits, not as an IP address (the internet does not however allow such names in DNS). Services with external names that resemble IPv4 addresses are not resolved by DNS servers.
>
> If you want to map a Service directly to a specific IP address, consider using headless Services.

When looking up the host `my-service.prod.svc.cluster.local`, the cluster DNS Service returns a `CNAME` record with the value `my.database.example.com`. Accessing `my-service` works in the same way as other Services but with the crucial difference that redirection happens at the DNS level rather than via proxying or forwarding. Should you later decide to move your database into your cluster, you can start its Pods, add appropriate selectors or endpoints, and change the Service's `type`.

> **Caution:**
> You may have trouble using ExternalName for some common protocols, including HTTP and HTTPS. If you use ExternalName then the hostname used by clients inside your cluster is different from the name that the ExternalName references.
>
> For protocols that use hostnames this difference may lead to errors or unexpected responses. HTTP requests will have a `Host:` header that the origin server does not recognize; TLS servers will not be able to provide a certificate matching the hostname that the client connected to.

Headless Services

Sometimes you don't need load-balancing and a single Service IP. In this case, you can create what are termed *headless Services*, by explicitly specifying `"None"` for the cluster IP address (`.spec.clusterIP`).

You can use a headless Service to interface with other service discovery mechanisms, without being tied to Kubernetes' implementation.

For headless Services, a cluster IP is not allocated, kube-proxy does not handle these Services, and there is no load balancing or proxying done by the platform for them. How DNS is automatically configured depends on whether the Service has selectors defined:

With selectors

For headless Services that define selectors, the endpoints controller creates EndpointSlices in the Kubernetes API, and modifies the DNS configuration to return A or AAAA records (IPv4 or IPv6 addresses) that point directly to the Pods backing the Service.

Without selectors

For headless Services that do not define selectors, the control plane does not create EndpointSlice objects. However, the DNS system looks for and configures either:

- DNS CNAME records for `type: ExternalName` Services.
- DNS A / AAAA records for all IP addresses of the Service's ready endpoints, for all Service types other than `ExternalName`.
  - For IPv4 endpoints, the DNS system creates A records.
  - For IPv6 endpoints, the DNS system creates AAAA records.

When you define a headless Service without a selector, the `port` must match the `targetPort`.

Discovering services

For clients running inside your cluster, Kubernetes supports two primary modes of finding a Service: environment variables and DNS.

Environment variables

When a Pod is run on a Node, the kubelet adds a set of environment variables for each active Service. It adds `{SVCNAME}_SERVICE_HOST` and `{SVCNAME}_SERVICE_PORT` variables, where the Service name is upper-cased and dashes are converted to underscores. It also supports variables (see makeLinkVariables) that are compatible with Docker Engine's "*legacy container links*" feature.

For example, the Service `redis-primary` which exposes TCP port 6379 and has been allocated cluster IP address 10.0.0.11, produces the following environment variables:

```
REDIS_PRIMARY_SERVICE_HOST=10.0.0.11
REDIS_PRIMARY_SERVICE_PORT=6379
REDIS_PRIMARY_PORT=tcp://10.0.0.11:6379
REDIS_PRIMARY_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_PRIMARY_PORT_6379_TCP_PROTO=tcp
REDIS_PRIMARY_PORT_6379_TCP_PORT=6379
REDIS_PRIMARY_PORT_6379_TCP_ADDR=10.0.0.11
```

Note:
When you have a Pod that needs to access a Service, and you are using the environment variable method to publish the port and cluster IP to the client Pods, you must create the Service *before* the client Pods come into existence. Otherwise, those client Pods won't have their environment variables populated.

If you only use DNS to discover the cluster IP for a Service, you don't need to worry about this ordering issue.

Kubernetes also supports and provides variables that are compatible with Docker Engine's "*legacy container links*" feature. You can read `makeLinkVariables` to see how this is implemented in Kubernetes.

DNS

You can (and almost always should) set up a DNS service for your Kubernetes cluster using an add-on.

A cluster-aware DNS server, such as CoreDNS, watches the Kubernetes API for new Services and creates a set of DNS records for each one. If DNS has been enabled throughout your cluster then all Pods should automatically be able to resolve Services by their DNS name.

For example, if you have a Service called `my-service` in a Kubernetes namespace `my-ns`, the control plane and the DNS Service acting 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` (`my-service.my-ns` would also work).

Pods in other namespaces must qualify the name as `my-service.my-ns`. These names will resolve to the cluster IP assigned for the Service.

Kubernetes also supports DNS SRV (Service) records for named ports. If the `my-service.my-ns` Service has a port named `http` with the protocol set to `TCP`, you can do a DNS SRV query for `_http._tcp.my-service.my-ns` to discover the port number for `http`, as well as the IP address.

The Kubernetes DNS server is the only way to access `ExternalName` Services. You can find more information about `ExternalName` resolution in DNS for Services and Pods.

Virtual IP addressing mechanism

Read Virtual IPs and Service Proxies explains the mechanism Kubernetes provides to expose a Service with a virtual IP address.

Traffic policies

You can set the `.spec.internalTrafficPolicy` and `.spec.externalTrafficPolicy` fields to control how Kubernetes routes traffic to healthy (“ready”) backends.

See Traffic Policies for more details.

Session stickiness

If you want to make sure that connections from a particular client are passed to the same Pod each time, you can configure session affinity based on the client's IP address. Read session affinity to learn more.

External IPs

If there are external IPs that route to one or more cluster nodes, Kubernetes Services can be exposed on those `externalIPs`. When network traffic arrives into the cluster, with the external IP (as destination IP) and the port matching that Service, rules and routes that Kubernetes has configured ensure that the traffic is routed to one of the endpoints for that Service.

When you define a Service, you can specify `externalIPs` for any service type. In the example below, the Service named `"my-service"` can be accessed by clients using TCP, on `"198.51.100.32:80"` (calculated from `.spec.externalIPs[]` and `.spec.ports[].port`).

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 49152
  externalIPs:
    - 198.51.100.32
```

> **Note:** Kubernetes does not manage allocation of `externalIPs`; these are the responsibility of the cluster administrator.

API Object

Service is a top-level resource in the Kubernetes REST API. You can find more details about the Service API object.

What's next

Learn more about Services and how they fit into Kubernetes:

- Follow the Connecting Applications with Services tutorial.
- Read about Ingress, which exposes HTTP and HTTPS routes from outside the cluster to Services within your cluster.
- Read about Gateway, an extension to Kubernetes that provides more flexibility than Ingress.

For more context, read the following:

- Virtual IPs and Service Proxies
- EndpointSlices
- Service API reference
- EndpointSlice API reference
- Endpoint API reference (legacy)

Feedback

Was this page helpful?

Yes No

Last modified October 13, 2023 at 9:12 PM PST: Fix some nits in the service concept page (6ed6bedd93)