Services

AUTOPILOT (/KUBERNETES-ENGINE/DOCS/CONCEPTS/AUTOPILOT-OVERVIEW)

STANDARD (/KUBERNETES-ENGINE/DOCS/CONCEPTS/TYPES-OF-CLUSTERS)

This page describes Kubernetes Services (https://kubernetes.io/docs/concepts/services-networking/service/) and their use in Google Kubernetes Engine (GKE). There are different types of Services, which you can use to group a set of Pod (https://kubernetes.io/docs/concepts/workloads/pods/) endpoints into a single resource. To learn how to create a Service, see Exposing applications using services (/kubernetes-engine/docs/how-to/exposing-apps).

What is a Kubernetes Service?

The idea of a Service is to group a set of Pod endpoints into a single resource. You can configure various ways to access the grouping. By default, you get a stable cluster IP address that clients inside the cluster can use to contact Pods in the Service. A client sends a request to the stable IP address, and the request is routed to one of the Pods in the Service.

A Service identifies its member Pods with a selector. For a Pod to be a member of the Service, the Pod must have all of the labels specified in the selector. A label (https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/) is an arbitrary key/value pair that is attached to an object.

The following Service manifest has a selector that specifies two labels. The `selector` field says any Pod that has both the `app: metrics` label and the `department:engineering` label is a member of this Service.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: metrics
    department: engineering
  ports:
  ...
```

Why use a Kubernetes Service?

In a Kubernetes cluster, each Pod has an internal IP address. But the Pods in a Deployment come and go, and their IP addresses change. So it doesn't make sense to use Pod IP addresses directly. With a Service, you get a stable IP address that lasts for the life of the Service, even as the IP addresses of the member Pods change.

A Service also provides load balancing. Clients call a single, stable IP address, and their requests are balanced across the Pods that are members of the Service.

Types of Kubernetes Services

There are five types of Services:

- **ClusterIP (default):** Internal clients send requests to a stable internal IP address.
- **NodePort:** Clients send requests to the IP address of a node on one or more `nodePort` values that are specified by the Service.
- **LoadBalancer:** Clients send requests to the IP address of a network load balancer.
- **ExternalName:** Internal clients use the DNS name of a Service as an alias for an external DNS name.
- **Headless:** You can use a headless service (https://kubernetes.io/docs/concepts/services-networking/service/#headless-services) when you want a Pod grouping, but don't need a stable IP address.

The `NodePort` type is an extension of the `ClusterIP` type. So a Service of type `NodePort` has a cluster IP address.

The `LoadBalancer` type is an extension of the `NodePort` type. So a Service of type `LoadBalancer` has a cluster IP address and one or more `nodePort` values.

Services of type ClusterIP

When you create a Service of type `ClusterIP`, Kubernetes creates a stable IP address that is accessible from nodes in the cluster.

Here is a manifest for a Service of type ClusterIP:

```
apiVersion: v1
kind: Service
metadata:
  name: my-cip-service
spec:
  selector:
    app: metrics
    department: sales
  type: ClusterIP
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

You can create the Service (/kubernetes-engine/docs/how-to/exposing-apps) by using `kubectl apply -f [MANIFEST_FILE]`. After you create the Service, you can use `kubectl get service` to see the stable IP address:

```
NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
my-cip-service   ClusterIP   10.11.247.213   none          80/TCP
```

Clients in the cluster call the Service by using the cluster IP address and the TCP port specified in the `port` field of the Service manifest. The request is forwarded to one of the member Pods on the TCP port specified in the `targetPort` field. For the preceding example, a client calls the Service at `10.11.247.213` on TCP port 80. The request is forwarded to one of the member Pods on TCP port 8080. The member Pod must have a container that is listening on TCP port 8080. If there is no container listening on port 8080, clients will see a message like "Failed to connect" or "This site can't be reached".

Service of type NodePort

When you create a Service of type `NodePort`, Kubernetes gives you a `nodePort` value. Then the Service is accessible by using the IP address of any node along with the `nodePort` value.

Here is a manifest for a Service of type `NodePort`:

```
apiVersion: v1
kind: Service
metadata:
  name: my-np-service
spec:
  selector:
    app: products
    department: sales
  type: NodePort
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

After you create the Service, you can use `kubectl get service -o yaml` to view its specification and see the `nodePort` value.

```
spec:
  clusterIP: 10.11.254.114
  externalTrafficPolicy: Cluster
  ports:
  - nodePort: 32675
    port: 80
    protocol: TCP
    targetPort: 8080
```

External clients call the Service by using the external IP address of a node along with the TCP port specified by `nodePort`. The request is forwarded to one of the member Pods on the TCP port specified by the `targetPort` field.

For example, suppose the external IP address of one of the cluster nodes is `203.0.113.2`. Then for the preceding example, the external client calls the Service at `203.0.113.2` on TCP port 32675. The request is forwarded to one of the member Pods on TCP port 8080. The member Pod must have a container listening on TCP port 8080.

The `NodePort` Service type is an extension of the `ClusterIP` Service type. So internal clients have two ways to call the Service:

- Use `clusterIP` and `port`.
- Use a node's IP address and `nodePort`.

For some cluster configurations, the external Application Load Balancer (/load-balancing/docs/https) uses a Service of type `NodePort`. For more information, see Set up an external Application Load Balancer with Ingress (/kubernetes-engine/docs/tutorials/http-balancer).

An external Application Load Balancer is a proxy server, and is fundamentally different from the external passthrough Network Load Balancer (/load-balancing/docs/network) described in this topic under Service of type LoadBalancer (#services_of_type_loadbalancer).

Note: You can specify your own `nodePort` value in the 30000--32767 range. However, it's best to omit the field and let Kubernetes allocate a `nodePort` for you. This avoids collisions between Services.

Services of type LoadBalancer

To learn more about Services of type LoadBalancer, see LoadBalancer Service concepts (/kubernetes-engine/docs/concepts/service-load-balancer).

Service of type ExternalName

A Service of type `ExternalName` provides an internal alias for an external DNS name. Internal clients make requests using the internal DNS name, and the requests are redirected to the external name.

Here is a manifest for a Service of type `ExternalName`:

```
apiVersion: v1
kind: Service
metadata:
  name: my-xn-service
spec:
  type: ExternalName
  externalName: example.com
```

When you create a Service, Kubernetes creates a DNS name that internal clients can use to call the Service. For the preceding example, the DNS name is my-xn-service.default.svc.cluster.local. When an internal client makes a request to my-xn-service.default.svc.cluster.local, the request gets redirected to example.com.

The `ExternalName` Service type is fundamentally different from the other Service types. In fact, a Service of type `ExternalName` does not fit the definition of Service given at the beginning of this topic. A Service of type `ExternalName` is not associated with a

set of Pods, and it does not have a stable IP address. Instead, a Service of type `ExternalName` is a mapping from an internal DNS name to an external DNS name.

Headless Service

A headless Service is a type of Kubernetes Service that does not allocate a cluster IP address. Instead, a headless Service uses DNS to expose the IP addresses of the Pods that are associated with the Service. This allows you to connect directly to the Pods, instead of going through a proxy.

Headless Services are useful for a variety of scenarios, including:

- **Load balancing across pods**: You can use headless Services to load balance across Pods. To implement this, create a Service with a selector that matches the Pods that you want to load balance. The Service will then distribute traffic evenly across all of the Pods that match the selector.
- **Service discovery**: You can use a headless Service to implement Service discovery. To implement this, create a Service with a name and a selector. DNS record for the headless service contains all the IPs of the Pods behind the Service that match the selector. Clients can use these DNS records to find the IP addresses of the Pods that are associated with the Service.
- **Direct Pod access**: Clients can connect directly to the Pods that are associated with a headless Service, which can be useful for Services that require direct access to the underlying Pods, such as load balancers and DNS servers.
- **Flexibility**: Headless services can be used to create a variety of different topologies, such as load balancers, DNS servers, and distributed databases.

If you have special network requirements for your workloads that can not be solved using headless Services with selectors, there is also the possibility of using headless Services without selectors. Headless Services are a useful tool for accessing Services that are not located within the Kubernetes cluster itself, as the control plane does not create EndpointSlice objects, you can read more about it in Service without selectors (https://kubernetes.io/docs/concepts/services-networking/service/#without-selectors)

The following example is a manifest for a Headless Service:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  clusterIP: None
  selector:
    app: nginx
  ports:
  - name: http
    port: 80
    targetPort: 80
```

Once you have created a headless Service, you can find the IP addresses of the Pods that are associated with the Service by querying the DNS. For example, the following command lists the IP addresses of the Pods that are associated with the nginx Service:

Note: This example assumes that the Pods created are tagged with the **`nginx`** label.

```
dig +short nginx.default.svc.cluster.local
```

Another example which uses Kubernetes query expansion::

```
dig +short +search nginx
```

You can create a headless Service with a single command, and headless Services are easy to update and scale.

```
kubectl create service clusterip my-svc --clusterip="None" --dry-run=client -o yaml > [file.yaml]
```

Service abstraction

A Service is an abstraction in the sense that it is not a process that listens on some network interface. Part of the abstraction is implemented in the iptables (https://linux.die.net/man/8/iptables) rules of the cluster nodes. Depending on the type of the Service, other parts of the abstraction are implemented by either an external passthrough Network Load Balancer (/load-balancing/docs/network) or an external Application Load Balancer (/load-balancing/docs/https).

Arbitrary Service ports

The value of the `port` field in a Service manifest is arbitrary. However, the value of `targetPort` is not arbitrary. Each member Pod must have a container listening on `targetPort`.

Here's a Service, of type `LoadBalancer`, that has a `port` value of 50000:

```
apiVersion: v1
kind: Service
metadata:
  name: my-ap-service
spec:
  clusterIP: 10.11.241.93
  externalTrafficPolicy: Cluster
  ports:
  - nodePort: 30641
    port: 50000
    protocol: TCP
    targetPort: 8080
  selector:
    app: parts
    department: engineering
  sessionAffinity: None
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
    - ip: 203.0.113.200
```

A client calls the Service at `203.0.113.200` on TCP port 50000. The request is forwarded to one of the member Pods on TCP port 8080.

Multiple ports

The `ports` field of a Service is an array of ServicePort (https://v1-25.docs.kubernetes.io/docs/reference/generated/kubernetes-api/v1.25/#serviceport-v1-core) objects. The ServicePort object has these fields:

- `name`
- `protocol`
- `port`
- `targetPort`
- `nodePort`

If you have more than one ServicePort, each ServicePort must have a unique name.

Here is a Service, of type `LoadBalancer`, that has two `ServicePort` objects:

```
apiVersion: v1
kind: Service
metadata:
  name: my-tp-service
spec:
  clusterIP: 10.11.242.196
  externalTrafficPolicy: Cluster
  ports:
  - name: my-first-service-port
    nodePort: 31233
    port: 60000
    protocol: TCP
    targetPort: 50000
  - name: my-second-service-port
    nodePort: 31081
    port: 60001
    protocol: TCP
    targetPort: 8080
  selector:
    app: tests
    department: engineering
  sessionAffinity: None
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
    - ip: 203.0.113.201
```

Note: You can specify a maximum of five ports for a LoadBalancer service.

In the preceding example, if a client calls the Service at `203.0.113.201` on TCP port 60000, the request is forwarded to a member Pod on TCP port 50000. But if a client calls the Service at `203.0.113.201` on TCP port 60001, the request is forwarded to a member Pod on TCP port 8080.

Each member Pod must have a container listening on TCP port 50000 and a container listening on TCP port 8080. This could be a single container with two threads, or two containers running in the same Pod.

Service endpoints

When you create a Service, Kubernetes creates an Endpoints (https://v1-25.docs.kubernetes.io/docs/reference/generated/kubernetes-api/v1.25/#endpoints-v1-core) object that has the same name as your Service. Kubernetes uses the Endpoints object to keep track of which Pods are members of the Service.

Single-stack and dual-stack Services

You can create an IPv6 Service of type `ClusterIP` (/kubernetes-engine/docs/concepts/service#services_of_type_clusterip) or `NodePort` (/kubernetes-engine/docs/concepts/service#service_of_type_nodeport). GKE supports dual-stack Services of type `LoadBalancer` (/kubernetes-engine/docs/concepts/service-load-balancer) during Preview (/products#product-launch-stages) which carries no SLA or technical support.

For each of these Service types, you can define `ipFamilies` and `ipFamilyPolicy` fields as either IPv4, IPv6, or a dual-stack (https://kubernetes.io/docs/concepts/services-networking/dual-stack/#services) Service.

What's next

- Learn more about Kubernetes Services (https://kubernetes.io/docs/concepts/services-networking/service/)
- Expose Applications using Services (/kubernetes-engine/docs/how-to/exposing-apps)
- Learn more about StatefulSets (/kubernetes-engine/docs/concepts/statefulset)
- Learn more about Ingress (/kubernetes-engine/docs/concepts/ingress)

- Complete the Set up an external Application Load Balancer with Ingress (/kubernetes-engine/docs/tutorials/http-balancer) tutorial

Last updated 2023-12-04 UTC.