# Service

Ow

# What is Service?

- An abstract way to expose an application running on a set of pods as network service.
- **Problem statement:** Kubernetes Pods are created and destroyed to match the desired state of your cluster. Pods are nonpermanent resources. 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?

# Service Resources

- In Kubernetes, a Service is an abstraction which defines a logical set of Pods and a policy by which to access them (sometimes this pattern is called a micro-service).
- The set of Pods targeted by a Service is usually determined by a selector.
- Kubernetes api-server will be queried for service discovery, they provide the updated end-points.

# Defining a service

- A Service in Kubernetes is a REST object, similar to pod
- Service request post to the API Server to create a new instance
- The name of a Service object must be a valid RFC 1035 label name.
- This means the name must:
- contain at most 63 characters
- contain only lowercase alphanumeric characters or '-'
- start with an alphabetic character
- end with an alphanumeric character

# Example

- For example, suppose you have a set of Pods where each listens on TCP port 9376 and contains a label app.kubernetes.io/name=MyApp

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

..

- This specification creates a new Service object named "my-service", which targets TCP port 9376 on any Pod with the app.kubernetes.io/name=MyApp label.
- Kubernetes assigns this Service an IP address (sometimes called the "cluster IP"), which is used by the Service proxies
- The controller for the Service selector continuously scans for Pods that match its selector, and then POSTs any updates to an Endpoint object also named "my-service".

# lab

- Nginx-svc and nginx-pod yaml in repo
- Create svc using
  - kubectl create –f <svc.yaml>
  - kubectl get svc
  - kubectl get ep
- Create pod using kubectl command
- kubectl  get ep

# Service without selectors

- We can services without selector. In this case used corresponding endpoint. 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.

# 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 Endpoints object is not created automatically. You can manually map the Service to the network address and port where it's running, by adding an Endpoints object manually:

```yaml
apiVersion: v1
kind: Endpoints
metadata:
  # the name here should match the name of the Service
  name: my-service
subsets:
  - addresses:
      - ip: 192.0.2.42
    ports:
      - port: 9376
```

# Virtual ips and service proxies

- Every node in a Kubernetes cluster runs a kube-proxy. kube-proxy is responsible for implementing a form of virtual IP for Services
- The kube-proxy's configuration is done via a ConfigMap, and the ConfigMap for kube-proxy effectively deprecates the behaviour for almost all of the flags for the kube-proxy.
- Kubectl get pods –namespace=kube-sytem
- *kubectl get pod kube-proxy-vkm79 -n kube-system -o yaml*
- *kubectl exec kube-proxy-vkm79 -c kube-proxy -n kube-system -- cat /var/lib/kube-proxy/config.conf*

```
apiVersion: kubeproxy.config.k8s.io/v1alpha1
bindAddress: 0.0.0.0
bindAddressHardFail: false
clientConnection:
  acceptContentTypes: ""
  burst: 0
  contentType: ""
  kubeconfig: /var/lib/kube-proxy/kubeconfig.conf
  qps: 0
clusterCIDR: 10.244.0.0/16
configSyncPeriod: 0s
```

# Proxy mode

- Mainly two types proxy modes
  - Iptables
  - IPVS
- Iptables mode In this mode, kube-proxy watches the Kubernetes control plane for the addition and removal of Service and Endpoint objects, Using iptables to handle traffic has a lower system overhead, because traffic is handled by Linux netfilter without the need to switch between userspace and the kernel space.
- You can use Pod readiness probes to verify that backend Pods are working OK, so that kube-proxy in iptables mode only sees backends that test out as healthy

# IPVS

- In ipvs mode, kube-proxy watches Kubernetes Services and Endpoints, calls netlink interface to create IPVS rules accordingly and synchronizes IPVS rules with Kubernetes Services and Endpoints periodically.
- To run kube-proxy in IPVS mode, you must make IPVS available on the node before starting kube-proxy.
- When kube-proxy starts in IPVS proxy mode, it verifies whether IPVS kernel modules are available. If the IPVS kernel modules are not detected, then kube-proxy falls back to running in iptables proxy mode.

# 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

# Core-DNS

- CoreDNS, watches the Kubernetes API for new Services and creates a set of DNS records for each one
- 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.

# SRV records

- 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.

# 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 (.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.
- For headless Services that define selectors, the endpoints controller creates Endpoints records in the API, and modifies the DNS configuration to return A records (IP addresses) that point directly to the Pods backing the Service.

# Type of services

- 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 ServiceType.
- NodePort: Exposes the Service on each Node's IP at a static port (the NodePort). A ClusterIP Service, to which the NodePort Service routes, is automatically created. You'll be able to contact the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort>.
- LoadBalancer: Exposes the Service externally using a cloud provider's load balancer. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.
- ExternalName: Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up.

# 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.
- 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.

# Example and lab

- Note that this Service is visible as <NodeIP>:spec.ports[*].nodePort and .spec.clusterIP:spec.ports[*].port.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app.kubernetes.io/name: MyApp
  ports:
      # By default and for convenience, the `targetPort` is se
    - port: 80
      targetPort: 80
      # Optional field
      # By default and for convenience, the Kubernetes control
      nodePort: 30007
```