Process of creating a service in Kubernetes:

❶ Define the service: The first step in creating a service is to define the service using a YAML file or through the Kubernetes API. The YAML file specifies details such as the name of the service, the selector used to identify the pods that the service should route traffic to, and the type of service (ClusterIP, NodePort, or LoadBalancer).

❷ Submit the service definition: Once the service definition is created, it can be submitted to the Kubernetes API server

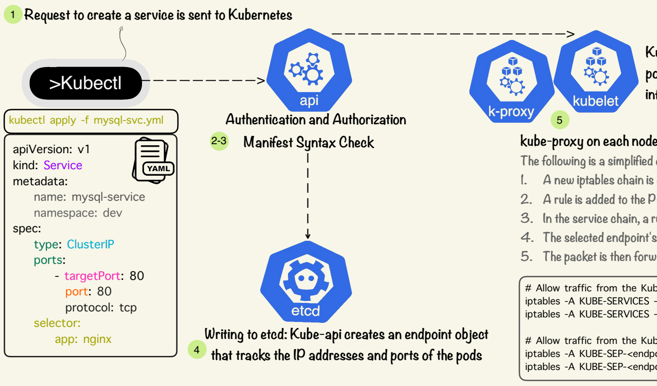
❸ API Server validates the service definition: The Kubernetes API server receives the service definition and validates it to ensure that it is well-formed and contains all the required information.

❹ Service is created: Once the service is created, Kubernetes creates an endpoint object that tracks the IP addresses and ports of the pods that the service should route traffic to. This information is stored in etcd

❺ iptables rules are created: Once the endpoint object is created, kube-proxy creates iptables rules on each node in the cluster to route traffic to the pods that are part of the service. These iptables rules are used to ensure that traffic is routed to the correct pod, and that traffic is load balanced across multiple pods if more than one pod matches the selector.

❻ Access the service: The service is now accessible within the cluster using its name or DNS name, and can be used to route traffic to the pods that are part of the service.

❼ Monitor the service: Once the service is running, Kubernetes monitors its health and takes action if any issues arise. For example, if a pod fails, Kubernetes will automatically remove it from the list of endpoints for the service.


1 Request to create a service is sent to Kubernetes
>Kubectl
api
Authentication and Authorization
2-3 Manifest Syntax Check
etcd
4 Writing to etcd: Kube-api creates an endpoint object that tracks the IP addresses and ports of the pods
k-proxy
kubelet
5
YAML

```
kubectl apply -f mysql-svc.yml
```

```
apiVersion: v1
kind: Service
metadata:
    name: mysql-service
    namespace: dev
spec:
    type: ClusterIP
    ports:
        - targetPort: 80
          port: 80
          protocol: tcp
    selector:
        app: nginx
```

Kubelet interacts with kube-proxy to ensure that the iptables rules are properly configured and connected to the appropriate pods. Kubelet provides kube-proxy with information about the pods that are running on the node, and kube-proxy uses this information to update the iptables rules as needed.

kube-proxy on each node in the cluster automatically creates iptables rules to forward traffic to the service endpoints.

The following is a simplified overview of the iptables rules created by kube-proxy:

1. A new iptables chain is created with the name of the service (e.g. "my-service").
2. A rule is added to the PREROUTING chain to match incoming traffic that is destined for the service's cluster IP address (e.g. 10.0.0.1) and jump to the service chain.
3. In the service chain, a rule is added to select one of the service's endpoints using a load balancing algorithm (e.g. round-robin).
4. The selected endpoint's IP address is rewritten as the destination IP address of the packet.
5. The packet is then forwarded to the selected endpoint.

iptables

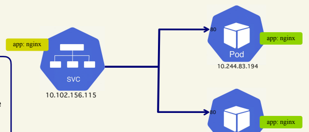
```
# Allow traffic from the Kubernetes Service IP address and port to the Kubernetes Endpoints
iptables -A KUBE-SERVICES -d <ClusterIP>/32 -p tcp -m comment --comment "mysql-service/mysql: cluster IP" -m tcp --dport <Port> -j KUBE-MARK-MASQ
iptables -A KUBE-SERVICES -d <ClusterIP>/32 -p tcp -m comment --comment "mysql-service/mysql" -m tcp --dport <Port> -j KUBE-SVC-<service-uid>

# Allow traffic from the Kubernetes Endpoints to the Pods
iptables -A KUBE-SEP-<endpoint-uid> -s <PodIP> -m comment --comment "mysql-service/mysql" -j KUBE-MARK-MASQ
iptables -A KUBE-SEP-<endpoint-uid> -s <PodIP> -m comment --comment "mysql-service/mysql" -j DNAT --to-destination <PodIP>:<Port>
```

Endpoint

When a Service is created, the Service controller queries the Kubernetes API server to get a list of all Pods that match the Service's label selector. It then creates an Endpoint object that includes the IP addresses and ports of these Pods, and associates the Endpoint with the Service. The Kubernetes networking layer uses this Endpoint information to route traffic to the appropriate Pods that make up the Service.

```
apiVersion: v1
kind: Endpoints
metadata:
  name: my-web-service
subsets:
  - addresses:
      - ip: 10.244.1.194
      - ip: 10.244.1.195
    ports:
      - name: http
        port: 80
        protocol: TCP
```



Endpoints are automatically created and managed by Kubernetes when you create a Service, and they are updated dynamically as Pods are added or removed from the Service

DNS

Kubernetes has a built-in DNS component that provides naming and discovery between pods running on the cluster. It assigns DNS records (A records, SRV records, etc) for each pod/service automatically. The DNS name follows a specific format, such as <service-name>.<namespace>.svc.cluster.local for accessing a Service or <pod-name>.<service-name>.<namespace>.svc.cluster.local for accessing a specific Pod associated with a headless Service.

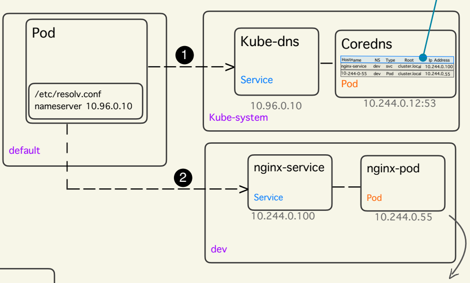
If a Pod located in the "default" namespace needs to communicate with a service named "nginx-service" residing in the "dev" namespace, it can do so by using the URL "http://nginx-service.dev.svc.cluster.local".

```
<pod-name>.<service-name>.<namespace>.svc.cluster.local
```

In Kubernetes, FQDN stands for Fully Qualified Domain Name. It is a complete domain name that specifies the exact location of a resource within the DNS hierarchy. By using FQDNs, Kubernetes simplifies the process of resource discovery, network routing, and namespace isolation within the cluster

CoreDNS will have the following DNS records for dev namespace

```
nginx-service.dev.svc.cluster.local          10.244.0.100
10-244-0-100.dev.pod.cluster.local           10.244.0.55
```



Hostname	NS	Type	Root	Ip Address
nginx-service	dev	svc	cluster.local	10.244.0.100
10-244-0-55	dev	Pod	cluster.local	10.244.0.55

The default DNS provider in Kubernetes is CoreDNS, which runs as pods/containers inside the cluster. CoreDNS retrieves pod/service information from the Kubernetes API to update its DNS records.

Notice:Kubernetes does not automatically create DNS records for Pod names directly. This is because Pod IPs keep changing whenever Pods are recreated or rescheduled.Instead, stable DNS records are maintained at the Service level in Kubernetes. Services have unchanging virtual IPs that act as stable endpoints

Pods get DNS resolution indirectly via records in the Pod DNS subdomain:
Each Pod gets a DNS record in the format :
<pod-ip-address>.<namespace>.pod.cluster.local

Pod dns policy

Pod's DNS settings can be configured based on the dnsPolicy field in a Pod specification. This dnsPolicy field accepts three possible values:

ClusterFirst: Any DNS query that does not match the configured search domains for the Pod are forwarded to the upstream nameserver. This is the default policy if dnsPolicy is not specified.

None: Allows a Pod to ignore DNS settings from the Kubernetes environment. All DNS settings are supposed to be provided using the dnsConfig field in the Pod Spec.

Default: Use the DNS settings of the node that the Pod is running on. This means it will use the same DNS as the node that the Pod runs on.

Please note that the Pod's DNS config allows you to customize the DNS parameters of a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mypod
      image: myimage
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 1.2.3.4
    searches:
      - ns1.svc.cluster-domain.example
      - my.dns.search.suffix
    options:
      - name: ndots
        value: "2"
```

In this example, the Pod mypod uses a custom DNS resolver (1.2.3.4) and a custom search list (ns1.svc.cluster-domain.example and my.dns.search.suffix). The option ndots:2 means that if a DNS query name contains less than 3 dots, then the search list mechanism will be used. For example, a query for mypod will be first tried as mypod.ns1.svc.cluster-domain.example and if that fails, as mypod.my.dns.search.suffix.