



Kubernetes Service&Network APIs: ClusterIP, NodePort, LoadBalancer, Ingress, GatewayAPI

Check GitHub for helpful DevOps tools:

Michael Robotics
Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats information overload by adhering to the set of principles: simplify, prioritize, and execute.

 <https://github.com/MichaelRobotics>



Ask Personal AI Document assistant to learn interactively (FASTER)!

1

Download PDF

2

Go to website

3

Browse file

4

Chat with Document

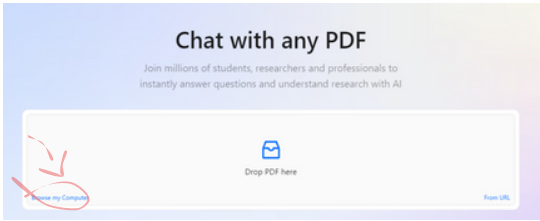
1

<https://github.com/MichaelRobotics/DevOpsTools/blob/main/KubernetesService.pdf>

2

Click there to go to ChatPdf website

3



4

Ask questions about document!

Completly new to Linux and Networking?

Essential for this PDF is a thorough knowledge of networking. I highly recommend the HTB platform's networking module, which offers extensive information to help build a comprehensive understanding.

HTB - Your Cyber Performance Center

We provide a human-first platform creating and maintaining high performing cybersecurity individuals and organizations.

 <https://www.hackthebox.com/>



What is Kubernetes?

Kubernetes is an open-source platform that automates the deployment, scaling, and management of containerized applications. It helps manage clusters of nodes running containers, ensuring efficient and reliable operation.

How Kubernetes clusters are made?

Kubernetes clusters consist of a control plane and multiple worker nodes. The control plane manages cluster operations, while worker nodes run the actual container workloads.

Why and When use Kubernetes

Kubernetes is ideal for deploying scalable, resilient, and automated containerized applications. It is used when managing multiple containers across different environments is necessary.

Example: Running a microservices-based e-commerce platform that scales up during peak hours.

System Requirements

- RAM: 2 GB per node (1 GB can work for testing but may lead to limited performance)
- 10 GB free storage
- Ubuntu

Kubernetes: Main components & packages

- **kube-apiserver:** Central management component that exposes the Kubernetes API; acts as the front-end for the cluster.
- **etcd:** Distributed key-value store for storing all cluster data, ensuring data consistency across nodes.
- **kube-scheduler:** Assigns pods to available nodes based on resource requirements and policies.
- **kube-controller-manager:** Manages core controllers that handle various functions like node status, replication, and endpoints.
- **kubelet:** Agent that runs on each node, responsible for managing pods and their containers.
- **kube-proxy:** Manages networking on each node, ensuring communication between pods and services within the cluster.

Kubernetes Service&Network APIs: Intro

1) Pods vs VM

In Kubernetes, pods are lightweight, isolated environments, often compared to VMs. Each pod has its own IP address, but these addresses are temporary since pods are created and destroyed dynamically. This makes it impractical to use a pod's IP as a fixed communication endpoint.

2) Services and network API's

Kubernetes solves this with Services, which provide a stable endpoint for accessing pods. Services ensure consistent communication by abstracting away the changing IPs of pods. They also offer features like load balancing and DNS-based service discovery, simplifying networking in dynamic environments.

3) Types:

ClusterIP: Exposes the Service internally within the cluster. It is the default type and provides a stable IP for communication between internal components.

NodePort: Makes the Service accessible outside the cluster by mapping it to a static port on each cluster node.

LoadBalancer: Integrates with cloud provider load balancers to expose the Service externally with a public IP.

Ingress: Manages external HTTP/S access to Services, providing routing, SSL termination, and domain-based traffic management.

Gateway API: A Kubernetes framework for managing HTTP/S, TCP, and UDP traffic with Gateways, GatewayClasses, and Routes for granular control, enhancing traffic management and service mesh integration.

ExternalName: Maps a Service to an external DNS name, allowing access to resources outside the cluster.

Download and enter project repo

```
git clone https://github.com/vfarcic/kubernetes-demo  
cd kubernetes-demo
```

Deploy application:

```
kubectl create namespace a-team  
kubectl --namespace a-team apply --filename deployment/base.yaml  
kubectl create namespace b-team
```

Kubernetes Service&Network APIs: ClusterIP

1) ClusterIP structure

ClusterIP restricts Service communication to Pods within the same cluster, preventing external traffic access. This limitation ensures internal-only connectivity for matching Pods.

show ClusterIP service yaml:

```
cat service/base.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: silly-demo
  name: silly-demo
spec:
  ports:
    - name: http
      port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    app.kubernetes.io/name: silly-demo
  type: ClusterIP
```

It uses TCP on pod port 8080 to forward requests to targetPort 8080 on container, assuming a container process is listening there. The Service selector targets Pods with the name: silly-demo label in the same Namespace, forwarding requests to them.

2) ClusterIP service discovery

apply manifest:

```
kubectl --namespace a-team apply --filename service/base.yaml
```

To communicate to those Pods from inside the same cluster We just need to know the name of the Service and, sometimes, also the Namespace where it's running. Create another pod in same namespace as service:

```
kubectl --namespace a-team run curl \
  --image curlimages/curl:8.7.1 --stdin --tty --rm \
  -- sh
```

Now curl service on port on which app is exposed, at endpoint related to application functionality:

```
curl http://silly-demo:8080/fibonacci?number=10
curl http://silly-demo:8080/fibonacci?number=5
curl http://silly-demo:8080/fibonacci?number=15
curl http://silly-demo:8080/fibonacci?number=20
```

now check if pods were reached, by checking its logs. First, check pods names:

```
kubectl --namespace a-team get pods
```

```
controlplane $ kubectl --namespace a-team get pods
NAME                                READY   STATUS    RESTARTS   AGE
silly-demo-5b764b57cc-bttns        1/1     Running   0           62s
silly-demo-5b764b57cc-s74nl        1/1     Running   0           62s
```

Then check each pod logs:

```
kubectl --namespace a-team logs silly-demo-5b764b57cc-bttns | grep fibonacci
```

```
controlplane $ kubectl --namespace a-team logs silly-demo-5b764b57cc-bttns | grep fibonacci
[GIN-debug] GET    /fibonacci          --> main.fibonacciHandler (1 handlers)
2024/12/27 21:41:01 DEBUG Handling request URI="/fibonacci?number=5"
2024/12/27 21:41:01 DEBUG Handling request URI="/fibonacci?number=15"
2024/12/27 21:41:03 DEBUG Handling request URI="/fibonacci?number=20"
```

```
kubectl --namespace a-team logs silly-demo-5b764b57cc-s74nl | grep fibonacci
```

```
controlplane $ kubectl --namespace a-team logs silly-demo-5b764b57cc-s74nl | grep fibonacci
[GIN-debug] GET    /fibonacci          --> main.fibonacciHandler (1 handlers)
```

Each Service has an internal DNS entry matching its name, so the requester only needs to know the Service name. The Service tracks the IPs of all associated Pods and distributes requests evenly using a round-robin algorithm. For example, if there are 3 Pods and 300 requests, each Pod receives roughly 100 requests. Not even distribution from current example is caused of small amount of sent requests.

Now try the same, but from different namespace

```
kubectl --namespace b-team run curl \
  --image curlimages/curl:8.7.1 --stdin --tty --rm \
  -- sh
```

send request:

```
curl http://silly-demo:8080
```


Output is as follows:

```
controlplane $ kubectl --namespace b-team run curl \
>   --image curlimages/curl:8.7.1 --stdin --tty --rm \
>   -- sh
If you don't see a command prompt, try pressing enter.
~ $ curl http://silly-demo:8080
curl: (6) Could not resolve host: silly-demo
```

pods try to reach other pods in same namespace (b-team). To change namespace, form URL where silly-demo is subdomain of namespace a-team.

```
curl http://silly-demo.a-team:8080
```

Now it should work, check output.

```
~ $ curl http://silly-demo.a-team:8080
This is a silly demo
```

Kubernetes Service&Network APIs: NodePort

1) NodePort structure

The Service type is NodePort, making it accessible outside the cluster. A port is opened on all nodes, allowing requests to reach any Pod associated with the Service.

show NodePort service yaml:

```
cat service/node-port.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: silly-demo
  name: silly-demo
spec:
  ports:
    - name: http
      port: 8080
      protocol: TCP
      targetPort: 8080
      nodePort: 30000
  selector:
    app.kubernetes.io/name: silly-demo
  type: NodePort
```

By default, Kubernetes assigns a random port, but we can specify a port (e.g., nodePort: 30000). However, specifying a fixed port can lead to conflicts if another Service uses the same port. It's generally better to let Kubernetes choose a random port to avoid issues. For simplicity, we're using 30000 here, but it's not recommended in practice.

2) NodePort service discovery

apply manifest:

```
kubectl --namespace a-team apply \
  --filename service/node-port.yaml
```

If your cluster have node with specified ExternalIP, curl it with port 3000 to reach app endpoint:

```
kubectl get nodes \
  --output jsonpath="{.items[0].status.addresses}" | jq .
```

output:

```
[
  {
    "address": "10.142.0.14",
    "type": "InternalIP"
  },
  {
    "address": "35.231.232.6",
    "type": "ExternalIP"
  },
  [...]
]
```

```
curl "http://35.231.232.6:30000"
```

output:

```
This is a silly demo
```

If your cluster doesn't have ExternalIP setup, get nodes name and curl each node DNS at port 30000

```
controlplane $ kubectl get nodes
NAME           STATUS    ROLES          AGE   VERSION
controlplane   Ready     control-plane   21d   v1.31.0
node01         Ready     <none>          21d   v1.31.0
controlplane $ curl "http://controlplane:30000"
This is a silly demo
controlplane $ curl "http://node01:30000"
This is a silly demo
controlplane $
```

Service discovery works correctly. App can be reached from each Node at port 30000

A NodePort Service exposes a port (e.g., 30000) on every node, forwarding requests to one of its associated Pods, which could run on any node. This Service still supports internal communication like a ClusterIP Service.

Note: In managed Kubernetes environments (e.g., GKE, EKS, AKS), you might not need to use NodePort as they offer better alternatives. On-prem try MetalLB.

remove applied NodePort service:

```
kubectl --namespace a-team delete \
--filename service/node-port.yaml
```

Kubernetes Service&Network APIs: LoadBalancer

1) LoadBalancer structure

The problem with NodePort services is their reliance on sending requests to specific nodes, which fails if a node goes down due to upgrades or unexpected events.

A better solution is a proxy or load balancer that dynamically updates the list of healthy nodes and forwards requests accordingly. Building such a system ourselves is complex, but Kubernetes simplifies this with built-in support—just change the Service type to LoadBalancer.

```
cat service/load-balancer.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: silly-demo
  name: silly-demo
spec:
  ports:
    - name: http
      port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    app.kubernetes.io/name: silly-demo
  type: LoadBalancer
```

LoadBalancer Services build on NodePort Services by adding external load balancer support. Like NodePort, they open ports on all nodes, with the nodePort auto-generated if unspecified. Additionally, they create and configure external load balancers to forward traffic to the exposed ports.

2) LoadBalancer service discovery

apply manifest:

```
kubectl --namespace a-team apply \
--filename service/load-balancer.yaml
```

retrieve all the Services from that Namespace.

```
kubectl --namespace a-team get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
silly-demo	LoadBalancer	10.95.50.22	<pending>	8080:32342/TCP	6s

The Service mapped a node port (e.g., 32342) to the container port, as expected. However, the EXTERNAL-IP is <pending> because Cloud provider (if cluster is created on the AKS GKE or EKS) is creating a load balancer. Once ready, it will listen on port 8080, forward requests to the node port, and the Service will route them to the Pods.

If you created On-prem cluster, External-ip will stay in pending state (unless you have something like MetalLB setup). If on cloud, wait a moment and ExternalIP should show up:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
silly-demo	LoadBalancer	10.95.50.22	34.75.165.181	8080:30797/TCP	117s

copy the external IP assigned to load balancer and use it to send a request.

```
curl "http://34.75.165.181:8080"
```

Output

```
This is a silly demo
```

We received a response from one of the Pods without needing to target a specific node. Here's what happened:

We created a LoadBalancer Service, which instructed the provider to set up an external load balancer. This load balancer keeps an updated list of healthy worker nodes, removing any that go down. It was automatically configured to accept requests on port 8080 and forward them to an auto-generated port on the nodes, which the Service then routed to the appropriate Pods.

However, relying solely on LoadBalancer Services would require one load balancer per application, making DNS management chaotic. To streamline this, we need Ingress controllers.

Kubernetes Service&Network APIs: Ingress

1) Ingress intro

All your services will be ClusterIP, accessible only within the cluster, with no need for NodePort or load balancers. Unlike Services, Ingress isn't built into Kubernetes; it provides the specification but not the implementation. Install Traefik.

```
helm upgrade --install traefik traefik \
  --repo https://helm.traefik.io/traefik \
  --namespace traefik --create-namespace --wait
```

look at the Services in the traefik Namespace. I assume your cluster is created on AKS EKS or GKE. If not, just follow along.

```
kubectl --namespace traefik get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
traefik	LoadBalancer	10.95.49.105	35.185.11.184	80:31820/TCP,443:31323/TCP	47s

Traefik created a LoadBalancer service exposing ports 80 (HTTP) and 443 (HTTPS). The external load balancer maps these to random node ports. Normally, we'd map our domain DNS to the EXTERNAL-IP, but instead, we'll store it in an environment variable for now.

```
export EXTERNAL_IP=35.185.11.184
```


2) Ingress specification

see which Ingresses are available

```
kubectl get ingressclasses
```

NAME	CONTROLLER	PARAMETERS	AGE
traefik	traefik.io/ingress-controller	<none>	70s

look at an Ingress definition

```
cat service/ingress.yaml
```

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  labels:
    app.kubernetes.io/name: silly-demo
  name: silly-demo
spec:
  ingressClassName: traefik
  rules:
    - host: silly-demo.35.185.11.184.nip.io
      http:
        paths:
          - backend:
              service:
                name: silly-demo
                port:
                  number: 8080
            path: /
            pathType: ImplementationSpecific
```

This Ingress uses `ingressClassName: traefik` and defines rules to route requests from a `nip.io` subdomain to the `silly-demo` service on port 8080. `Nip.io` simulates domains by forwarding requests to the specified IP, useful for testing without a real domain. Before testing, we'll update the host with the load balancer's IP stored in `EXTERNAL_IP` using the `yq` command.

Install yq

```
wget https://github.com/mikefarah/yq/releases/latest/download/yq_linux_amd64 -O /usr/bin/yq
chmod +x /usr/bin/yq
yq --version
```

modify ingress YAML and place EXTERNAL_IP into host address.

```
yq --inplace \
  ".spec.rules[0].host = \"silly-demo.$EXTERNAL_IP.nip.io\"" \
  service/ingress.yaml
```

apply the Ingress resource

```
kubectl --namespace a-team apply --filename service/ingress.yaml
```

Let's see whether it works by sending a request

```
curl "http://silly-demo.$EXTERNAL_IP.nip.io"
```

The output should be as follows.

```
This is a silly demo
```

Here's what we did:

We set up a ClusterIP Service, accessible only within the cluster, linked to several Pods running an application.

We installed Traefik as an Ingress controller to act as a proxy and implement the Ingress specification.

The Ingress controller created a LoadBalancer service, exposing random ports for HTTP and HTTPS and configuring an external load balancer.

We applied an Ingress resource to configure the proxy to forward traffic from a specific domain to the service.

Requests to that domain are routed through the external load balancer, the Ingress Service, and the proxy, which forwards them to the ClusterIP Service and its Pods.

We can now add more Services and Ingress resources to handle traffic for additional domains.

Note: Ingress is being replaced by Gateway API, so consider skipping it for new projects.

Kubernetes Service&Network APIs: Gateway API

1) Gateway API intro

The Kubernetes Ingress specification has proven too limiting, leading many Ingress controllers to add custom features or abandon the spec entirely. It's widely regarded as one of the most problematic Kubernetes APIs.

Recognizing these limitations, the Kubernetes community developed Gateway API—a new, extensible specification designed to meet current and future needs. Unlike Ingress, it's flexible and future-proof, making it the preferred choice moving forward.

New users are advised to skip Ingress and adopt Gateway API, while existing users should plan to transition.

Google Kubernetes Engine, supports Gateway API out of the box so its great to use it for learning purposes.

Gateway API use gateway classes, to define what kind of loadbalancer to creat. List them:

```
kubectl get gatewayclasses
```

NAME	CONTROLLER	ACCEPTED	AGE
gke-l7-global-external-managed	networking.gke.io/gateway	True	47m
gke-l7-gxlb	networking.gke.io/gateway	True	47m
gke-l7-regional-external-managed	networking.gke.io/gateway	True	47m
gke-l7-riib	networking.gke.io/gateway	True	47m

Show gateway resource:

```
cat service/gateway.yaml
```

```
kind: Gateway
apiVersion: gateway.networking.k8s.io/v1beta1
metadata:
  name: http
spec:
  gatewayClassName: gke-l7-global-external-managed
  listeners:
    - name: http
      protocol: HTTP
      port: 80
```

The Gateway resource defines the Gateway Class (gatewayClassName), protocol, and port to be configured in the external load balancer.

apply

```
kubectl --namespace a-team apply --filename service/gateway.yaml
```

It might take a while until the external load balancer is created and configured so the ADDRESS is empty and it has not yet been PROGRAMMED. So wait a couple of minutes

output the gateways

```
kubectl --namespace a-team get gateways
```

NAME	CLASS	ADDRESS	PROGRAMMED	AGE
http	gke-l7-global-external-managed	34.120.55.14	True	81s

Store LoadBalancer IP in variable:

```
export EXTERNAL_IP=34.120.55.14
```

Now, look at route definition:

```
cat service/route.yaml
```

```
kind: HTTPRoute
apiVersion: gateway.networking.k8s.io/v1beta1
metadata:
  name: silly-demo
  labels:
    app.kubernetes.io/name: silly-demo
spec:
  parentRefs:
    - kind: Gateway
      name: http
  hostnames:
    - silly-demo.34.120.55.14.nip.io
  rules:
    - backendRefs:
        - name: silly-demo
          port: 8080
```

This is similar to an Ingress resource. We specify the Gateway, hostname (silly-demo.*.nip.io), and rules, with one rule forwarding requests to the silly-demo service on port 8080.

Setup Host to use nip.io with load balancer:

```
yq --inplace \
  ".spec.hostnames[0] = \"silly-demo.$EXTERNAL_IP.nip.io\"" \
  service/route.yaml\
```

apply

```
kubectl --namespace a-team apply --filename service/route.yaml
```

check available http routes in cluster

```
kubectl --namespace a-team get httproutes
```

NAME	HOSTNAMES	AGE
silly-demo	["silly-demo.34.120.55.14.nip.io"]	3s

Check if application can be reached through GatewayAPI:

```
curl "http://silly-demo.$EXTERNAL_IP.nip.io"
```

Output should be:

```
This is a silly demo
```

The output may show an error (e.g., fault filter abort) if the route isn't configured yet. Wait a moment and try the curl command again.

Gateway API offers more than Ingress, serving the same purpose but with added capabilities and flexibility for both implementations and users.

common troubleshooting

1) ClusterIP Service Not Accessible

Cause: Service misconfigured or not properly linked to Pods.

Solution: Check service details with `kubectl describe svc <service-name>`. Ensure Pods are running and the selector matches. Verify network policies.

2) LoadBalancer Service Not Getting External IP

Cause: Cloud provider issues or misconfigured LoadBalancer.

Solution: Check service status with `kubectl get svc <service-name>`. Verify cloud provider configuration or try recreating the service. Check events for errors

3) Ingress Not Routing Traffic

Cause: Incorrect Ingress resource, missing or misconfigured Ingress controller.

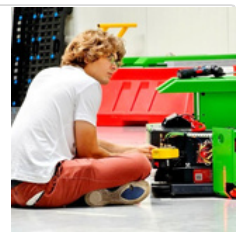
Solution: Check Ingress with `kubectl describe ingress <ingress-name>`. Verify Ingress controller is running. Ensure domain DNS points to the correct external IP or LoadBalancer.

4) Check my Kubernetes Troubleshooting series:

Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

 <https://github.com/MichaelRobotics>




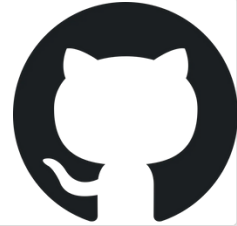
Learn more about Kubernetes

Check Kubernetes and piyushsachdeva - great docs!

Setup a Multi Node Kubernetes Cluster

kubeadm is a tool to bootstrap the Kubernetes cluster

 <https://github.com/piyushsachdeva/CKA-2024/tree/main/Resources/Day27>



Kubernetes Documentation

This section lists the different ways to set up and run Kubernetes

 <https://kubernetes.io/docs/setup/>



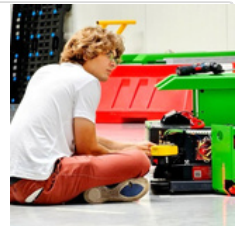
Share, comment, DM and check GitHub for scripts & playbooks created to automate process.

Check my GitHub

Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

<https://github.com/MichaelRobotics>



PS.

If you need a playbook or bash script to manage KVM on a specific Linux distribution, feel free to ask me in the comments or send a direct message!