

HCLSoftware

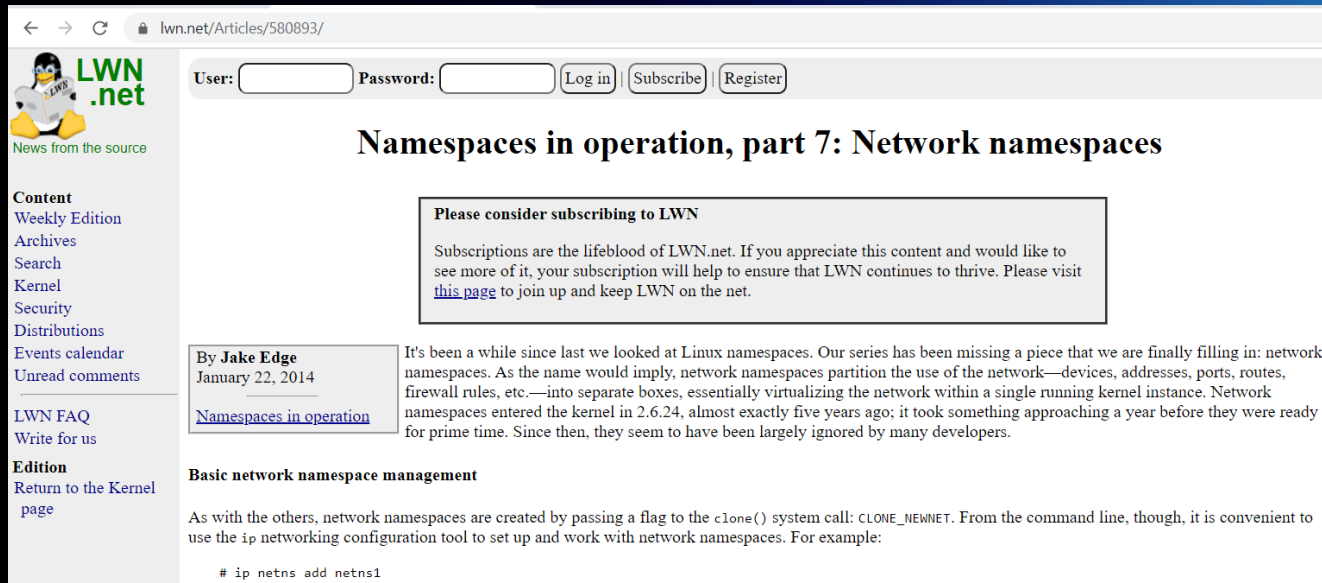
Software Containerization Lesson 3

l.ziosi@vu.nl

Agenda


- Networking basic concepts
 - OSI model
 - Switch
 - Router
 - Default Gateway
 - Ip Forwarding
 - Network Namespaces
 - Using Virtual Ethernet Interfaces to connect two namespaces
 - Using Linux Bridge to connect multiple namespaces
- Networking in Kubernetes
 - CNI
 - Services
 - ClusterIP
 - NodePort
 - LoadBalancer
 - CoreDNS
 - Ingress
 - Ingress controller
 - Ingress resources
 - Network policies
 - Calico
 - Calico on Microk8s

Networking basic concepts



The screenshot shows a web browser displaying the LWN.net website. The address bar shows the URL `lwn.net/Articles/580893/`. The page title is "Namespaces in operation, part 7: Network namespaces". The author is Jake Edge, and the date is January 22, 2014. The article is titled "Namespaces in operation". The content discusses network namespaces and their use in Linux. A sidebar on the left contains links to "Content", "Weekly Edition", "Archives", "Search", "Kernel", "Security", "Distributions", "Events calendar", and "Unread comments". A "Login" button is visible in the top right corner.

← → ↻ 🔒 lwn.net/Articles/580893/

 **LWN.net**
News from the source

User: Password: [Log in](#) [Subscribe](#) [Register](#)

Namespaces in operation, part 7: Network namespaces

Please consider subscribing to LWN

Subscriptions are the lifeblood of LWN.net. If you appreciate this content and would like to see more of it, your subscription will help to ensure that LWN continues to thrive. Please visit [this page](#) to join up and keep LWN on the net.

By Jake Edge
January 22, 2014

[Namespaces in operation](#)

It's been a while since last we looked at Linux namespaces. Our series has been missing a piece that we are finally filling in: network namespaces. As the name would imply, network namespaces partition the use of the network—devices, addresses, ports, routes, firewall rules, etc.—into separate boxes, essentially virtualizing the network within a single running kernel instance. Network namespaces entered the kernel in 2.6.24, almost exactly five years ago; it took something approaching a year before they were ready for prime time. Since then, they seem to have been largely ignored by many developers.

Basic network namespace management

As with the others, network namespaces are created by passing a flag to the `clone()` system call: `CLONE_NEWNET`. From the command line, though, it is convenient to use the `ip` networking configuration tool to set up and work with network namespaces. For example:

```
# ip netns add netns1
```

Networking Basic concepts: OSI model

OSI Layer	Description
1. Physical	Transmission of raw bit streams over physical medium
2. Link	Node to node data transfer, may use Point to Point Protocol and its derivatives.
3. Network	Transfer packets among nodes. Internet Protocol is used to relate datagrams across network boundaries.
4. Transport	Transmission Control Protocol (TCP) and User Datagram protocol (UDP). TCP provides reliable, ordered, and error-checked delivery of a stream of bytes between applications running on hosts communicating via an IP network.
5. Session	Network sockets are created by network APIs and use handles that are a type of file descriptor on Unix-like systems. They establish a long lasting communication channel.
6. Presentation	Purposes: character encoding, compression, decompression, encryption, decryption. Examples protocols: MIME, SSL, TLS
7. Application	High level APIs: file access, resource sharing. Example protocol: WebSockets

OSI: Open System Interconnection Model

Networking: Switches (Layer 2)

Switches connect devices within the same network and typically operate at the data link layer (2).

You can see the available network interfaces on your host with the `ip link` command:

```
lara@kube-master-gui:~$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
   link/ether 08:00:27:3c:1d:45 brd ff:ff:ff:ff:ff:ff
```

With the `ip addr` command you can see the IP addresses associated with every network interface:

```
lara@kube-master-gui:~$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
   link/ether 08:00:27:3c:1d:45 brd ff:ff:ff:ff:ff:ff
   inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute enp0s3
       valid_lft 57659sec preferred_lft 57659sec
   inet6 fe80::97cb:42be:abd5:d2f8/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

Networking: Routers

Routers operate at the 3-rd level in the ISO model and are capable of routing packets across different networks.

In packet switching networks, routing directs packets from source towards destination through intermediate nodes.

The `route` or `ip route` commands show the routing table that the kernel uses to direct packets.

In the example below, you can see that the network interface `enp0s3` directs all traffic with destination in the `10.0.2.0` subnet without using any Gateway. This is local traffic within the own network.

```
lara@kube-master-gui:~$ route
Kernel IP routing table
Destination      Gateway         Genmask         Flags Metric Ref    Use Iface
default          _gateway       0.0.0.0         UG    100    0      0 enp0s3
10.0.2.0         0.0.0.0        255.255.255.0   U     100    0      0 enp0s3
```

You can use the command:

```
ip route add <target_subnet_in_cidr_notation> via <ip_address>
```

to add a new route that allows the local network to reach the target subnet stepping through a node that is locally reachable.

Ip forwarding

A single computer may have different network interfaces, physically connected to different networks.

If the traffic received on one interface is forwarded by default to another interface, this effectively connects the two networks to one another.

If you want to use the device to forward the traffic, you have to check the file:

```
/proc/sys/net/ipv4/ip_forward
```

If it contains the value 0 it means that no traffic is forwarded.

If you change the value to 1 then all traffic is forwarded. There are two ways to see the current value:

```
lara@kubernetes-master-gui:~$ more /proc/sys/net/ipv4/ip_forward
1
lara@kubernetes-master-gui:~$ sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 1
```

To persist the change across reboots, use the following command to change the value:

```
lara@kubernetes-master-gui:~$ sudo sysctl -w net.ipv4.ip_forward=0
[sudo] password for lara:
net.ipv4.ip_forward = 0
lara@kubernetes-master-gui:~$ sudo sysctl -w net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
```


Network namespaces

To create and list network namespaces, you can use the `ip netns` commands:

```
lara@kube-master-gui:~$ sudo ip netns add ns1
[sudo] password for lara:
```

```
lara@kubernetes-master:~$ sudo ip netns add ns2
lara@kubernetes-master:~$ ip netns
ns2
ns1
```

To see the network interfaces inside these namespaces, you can use the `ip link` command, with the parameter `-n` followed by the namespace name. Note that there is only the loopback interface. The interfaces of the host are not visible. The network namespace is isolated from the host and from other namespaces.

```
lara@kube-master-gui:~$ sudo ip -n ns1 link
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
lara@kube-master-gui:~$ sudo ip -n ns2 link
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```


Connecting network namespaces

You can create a “virtual cable” between two network namespaces to connect them. First, you establish a link between two virtual ethernet interfaces (veth). Then you attach each virtual Ethernet interface to one network namespace.

```
lara@kube-master-gui:~$ sudo ip link add veth-ns1 type veth peer name veth-ns2
lara@kube-master-gui:~$ sudo ip link set veth-ns1 netns ns1
lara@kube-master-gui:~$ sudo ip link set veth-ns2 netns ns2
```

You can then set an IP address (from one of the private IP address ranges) to each virtual Ethernet interface.

```
lara@kube-master-gui:~$ sudo ip -n ns1 addr add 192.168.1.1 dev veth-ns1
lara@kube-master-gui:~$ sudo ip -n ns2 addr add 192.168.1.2 dev veth-ns2
lara@kube-master-gui:~$ sudo ip -n ns1 link set veth-ns1 up
lara@kube-master-gui:~$ sudo ip -n ns2 link set veth-ns2 up
```

Verify the virtual ethernet interfaces

To verify the new virtual Ethernet interfaces, you can use the `ip addr` command specifying the desired namespace:

```
lara@kubernetes-master-gui:~$ sudo ip -n ns1 addr
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
41: veth-ns1@if40: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether ee:13:29:33:9e:4c brd ff:ff:ff:ff:ff:ff link-netns ns2
    inet 192.168.1.1/32 scope global veth-ns1
        valid_lft forever preferred_lft forever
    inet6 fe80::ec13:29ff:fe33:9e4c/64 scope link
        valid_lft forever preferred_lft forever
lara@kubernetes-master-gui:~$ sudo ip -n ns2 addr
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
40: veth-ns2@if41: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 62:92:b2:f6:91:4e brd ff:ff:ff:ff:ff:ff link-netns ns1
    inet 192.168.1.2/32 scope global veth-ns2
        valid_lft forever preferred_lft forever
    inet6 fe80::6092:b2ff:fef6:914e/64 scope link
        valid_lft forever preferred_lft forever
```

However, at this point, you cannot yet ping the ip address of veth-ns2 from the namespace ns1 and viceversa. The reason is explained [here](#).

Setting up routes so that the two namespaces can ping each other

In this simple example of direct connection of two Virtual Ethernet interfaces, you must specify the route explicitly to allow ns1 to ping the IP address of veth-ns2 in ns2 and viceversa:

```
lara@kubernetes-master-gui:~$ sudo ip -n ns1 addr add 192.168.1.1 peer 192.168.1.2 dev veth-ns1
[sudo] password for lara:
lara@kubernetes-master-gui:~$ sudo ip -n ns2 addr add 192.168.1.2 peer 192.168.1.1 dev veth-ns2
lara@kubernetes-master-gui:~$ sudo ip netns exec ns1 ping 192.168.1.2
PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data:
64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=0.064 ms
64 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=0.095 ms
64 bytes from 192.168.1.2: icmp_seq=3 ttl=64 time=0.048 ms
^C
--- 192.168.1.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2071ms
rtt min/avg/max/mdev = 0.048/0.069/0.095/0.019 ms
lara@kubernetes-master-gui:~$ sudo ip netns exec ns2 ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data:
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.021 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=64 time=0.027 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=64 time=0.047 ms
^C
--- 192.168.1.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2056ms
rtt min/avg/max/mdev = 0.021/0.031/0.047/0.011 ms
```

ARP = Address
Resolution protocol

```
lara@kubernetes-master-gui:~$ sudo ip netns exec ns1 arp
Address          HWtype  HWaddress      Flags Mask    Iface
192.168.1.2      ether   c2:46:93:42:7e:7d C              veth-ns1
lara@kubernetes-master-gui:~$ sudo ip netns exec ns2 arp
Address          HWtype  HWaddress      Flags Mask    Iface
192.168.1.1      ether   fe:67:90:90:3d:c3 C              veth-ns2
```

arp on the host does
not list these routes

Using Linux Bridge to connect multiple net namespaces

This example is based on the following blog:

<https://ops.tips/blog/using-network-namespaces-and-bridge-to-isolate-servers/>

If you have many network namespaces it's impractical to create all the direct connections between couples of namespaces as in the previous example. It's simpler to connect all the namespaces to an interface that works like a virtual switch. The Linux Bridge provides such an interface. You can create it with:

```
sudo ip link add my-bridge type bridge
sudo ip link set dev my-bridge up
```

Create additional namespaces with the commands:

```
sudo ip netns add ns3
sudo ip netns add ns4
```

Remove the link created previously between ns1 and ns2 (removing one end is enough as the other is then removed automatically):

```
sudo ip -n ns1 link del veth-ns1
```

Create links from all namespaces to the bridge

Now create links from each namespace to the bridge with the following commands:

```
sudo ip link add veth-ns1 type veth peer name veth-ns1-bridge
sudo ip link add veth-ns2 type veth peer name veth-ns2-bridge
sudo ip link add veth-ns3 type veth peer name veth-ns3-bridge
sudo ip link add veth-ns4 type veth peer name veth-ns4-bridge
```

Set the two ends of the links to be in the relevant namespaces with the commands (for each namespace):

```
sudo ip link set veth-ns1 netns ns1
sudo ip link set veth-ns1-bridge master my-bridge
```

```
sudo ip link set veth-ns2 netns ns2
sudo ip link set veth-ns2-bridge master my-bridge
```

```
sudo ip link set veth-ns3 netns ns3
sudo ip link set veth-ns3-bridge master my-bridge
```

```
sudo ip link set veth-ns4 netns ns4
sudo ip link set veth-ns4-bridge master my-bridge
```

Configure IP addresses and netmasks

You can verify that the virtual Ethernet interfaces in the namespaces don't have any IP address. Instead of adding an ip address with a /32 netmask as before, add an ip address with /24 netmask (255.255.255.000).

Repeat this for each of the namespaces using increasing and unique IP addresses.

```
l10n@kali:~/k8s$ sudo ip netns exec ns1 ip address show
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
50: veth-ns1@if49: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state LOWERLAYERDOWN group
default qlen 1000
    link/ether e6:34:ec:62:23:29 brd ff:ff:ff:ff:ff:ff link-netnsid 0
lara@kubernetes-gui:~$ sudo ip netns exec ns1 ip addr add 192.168.1.1/24 dev veth-ns1
lara@kubernetes-gui:~$ sudo ip netns exec ns1 ip address show
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
50: veth-ns1@if49: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state LOWERLAYERDOWN group
default qlen 1000
    link/ether e6:34:ec:62:23:29 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.1.1/24 scope global veth-ns1
        valid_lft forever preferred_lft forever
```

Set the ip address of the bridge interface as another, not used, internal IP address:

```
sudo ip addr add 192.168.1.10/24 brd + dev my-bridge
```

This sets the address of the `my-bridge` interface (bridge device) to 192.168.1.10/24 and also sets the broadcast address to 192.168.1.255 (the `+` symbol sets the host bits to 255).

Activate all the network interfaces

The Virtual Ethernet Interfaces on the bridge side reside in the default network namespace so you activate them with:

```
sudo ip link set veth-ns1-bridge up
sudo ip link set veth-ns2-bridge up
sudo ip link set veth-ns3-bridge up
sudo ip link set veth-ns4-bridge up
```

The Virtual Ethernet Interfaces in the namespaces are activated with:

```
sudo ip netns exec ns1 ip link set veth-ns1 up
sudo ip netns exec ns2 ip link set veth-ns2 up
sudo ip netns exec ns3 ip link set veth-ns3 up
sudo ip netns exec ns4 ip link set veth-ns4 up
```

Verify the status of the bridge:

```
lara@kube-master-gui:~$ bridge link show my-bridge
49: veth-ns1-bridge@if50: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master my-bridge state forwarding priority 32 cost 2
51: veth-ns2-bridge@if52: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master my-bridge state forwarding priority 32 cost 2
53: veth-ns3-bridge@if54: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master my-bridge state forwarding priority 32 cost 2
55: veth-ns4-bridge@if56: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master my-bridge state forwarding priority 32 cost 2
```


Verify connectivity from the host through the bridge to the namespaces

Now you can verify that the host can connect to the IP addresses assigned to the virtual Ethernet interfaces of the various network namespaces, using the normal ping command:

```
lara@kube-master-gui:~$ ping 192.168.1.1 -c 1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.068 ms

--- 192.168.1.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.068/0.068/0.068/0.000 ms
lara@kube-master-gui:~$ ping 192.168.1.4 -c 1
PING 192.168.1.4 (192.168.1.4) 56(84) bytes of data.
64 bytes from 192.168.1.4: icmp_seq=1 ttl=64 time=0.070 ms

--- 192.168.1.4 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.070/0.070/0.070/0.000 ms
lara@kube-master-gui:~$
```

Verify connectivity from one namespace to another

To verify that one namespace can connect to the IP addresses assigned to the virtual Ethernet interfaces of a different namespaces, use a command like:

```
sudo ip netns exec ns1 ping 192.168.1.3 -c 1
```

```
lara@kube-master-gui:~$ sudo ip netns exec ns1 ping 192.168.1.3 -c 1
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=0.061 ms

--- 192.168.1.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.061/0.061/0.061/0.000 ms
lara@kube-master-gui:~$ sudo ip netns exec ns2 ping 192.168.1.4 -c 1
PING 192.168.1.4 (192.168.1.4) 56(84) bytes of data.
64 bytes from 192.168.1.4: icmp_seq=1 ttl=64 time=0.090 ms

--- 192.168.1.4 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.090/0.090/0.090/0.000 ms
```

Verify the routes in the different namespaces

You can use the `ip route` command inside each namespace to verify which routing information is available:

```
lara@kube-master-gui:~$ sudo ip netns exec ns1 ip route
192.168.1.0/24 dev veth-ns1 proto kernel scope link src 192.168.1.1
lara@kube-master-gui:~$ sudo ip netns exec ns2 ip route
192.168.1.0/24 dev veth-ns2 proto kernel scope link src 192.168.1.2
lara@kube-master-gui:~$ sudo ip netns exec ns3 ip route
192.168.1.0/24 dev veth-ns3 proto kernel scope link src 192.168.1.3
lara@kube-master-gui:~$ sudo ip netns exec ns4 ip route
192.168.1.0/24 dev veth-ns4 proto kernel scope link src 192.168.1.4
```

This shows that only the addresses in the local subnet 192.168.1.0/24 can be reached.

In fact, the Google DNS server 8.8.8.8 cannot be reached from ns1:

```
lara@kube-master-gui:~$ sudo ip netns exec ns1 ping 8.8.8.8 -c 1
ping: connect: Network is unreachable
```

To fix the issue we add a default network route that goes via the bridge interface. This allows us to resolve the route to 8.8.8.8 but we still cannot get the response back from the remote server:

```
lara@kube-master-gui:~$ sudo ip netns exec ns1 ip route add default via 192.168.1.10
lara@kube-master-gui:~$ sudo ip netns exec ns1 ping 8.8.8.8 -c 1
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

Modify iptable rules to allow for replies from remote server (SNAT)

The iptables command controls the firewall rules on a linux machine.

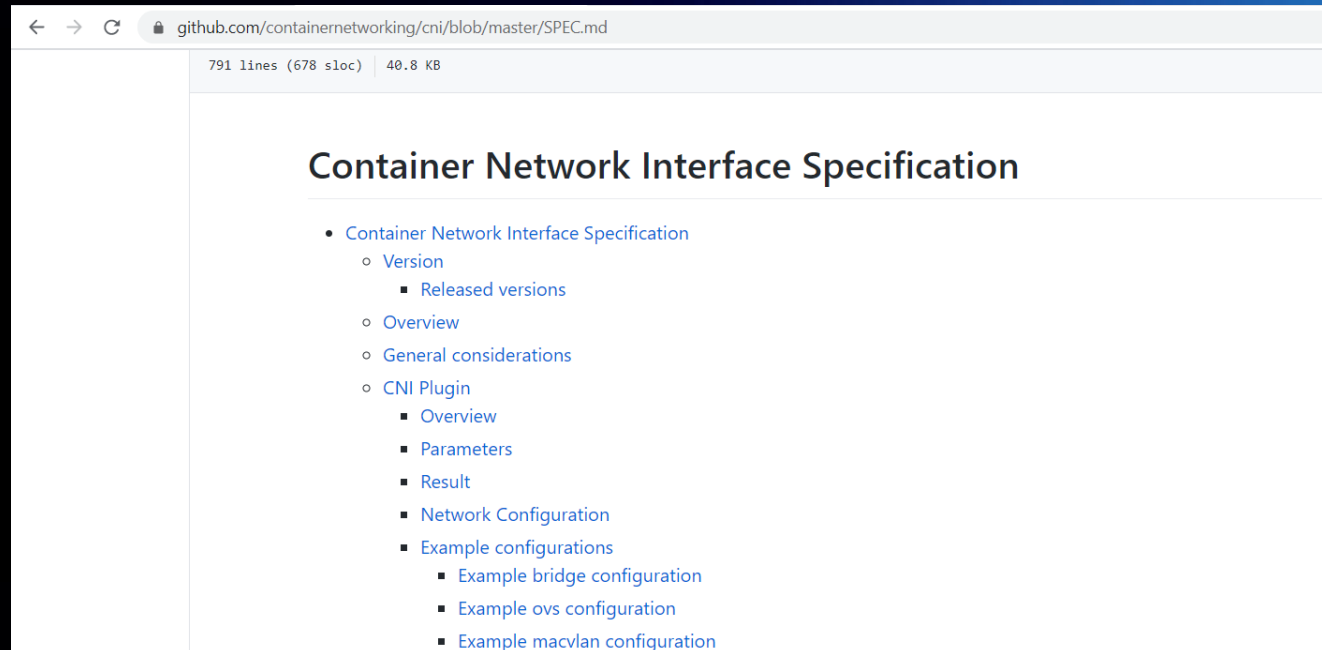
Here we use iptables with the following parameters to configure Source Network Address Translation (SNAT):

- `-t nat` select table "nat" for configuration of NAT rules.
- `-A POSTROUTING` Append a rule to the POSTROUTING chain (-A stands for "append").
- `-s 192.168.1.0/24` this rule is valid for packets that have the specified source address and netmask
- `-j MASQUERADE` the action that should take place is to 'masquerade' packets, i.e. replacing the sender's address by the router's address.

```
lara@kube-master-gui:~$ sudo iptables \
> -t nat \
> -A POSTROUTING \
> -s 192.168.1.0/24 \
> -j MASQUERADE
lara@kube-master-gui:~$ sudo ip netns exec ns1 ping 8.8.8.8 -c 1
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=116 time=7.42 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 7.415/7.415/7.415/0.000 ms
```

Kubernetes Networking



The Kubernetes Network model

The Kubernetes network model imposes constraints on third party network implementation:

- Every Pod has its own IP address. You do not need to create links between Pods manually.
- Containers within a pod share the pod IP address and can communicate with each other like processes inside the same VM, referring to each other via localhost.
- Pods on a node can communicate with all pods on all nodes without NAT.
- Agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node.
- For those platforms that support Pods running in the host network (e.g. Linux, where pods can be configured to share the host IP address):
 - Pods in the host network of a node can communicate with all pods on all nodes without NAT.

Depending on the Networking solution you choose, you may be able to impose restrictions on the above all-open communication model, using **Kubernetes Network Policies**.

The implementation of the network model is not part of Kubernetes itself. You must install a third party networking solutions for Kubernetes. For a list of solutions see:

<https://kubernetes.io/docs/concepts/cluster-administration/networking/>

Microk8s installs Calico by default. Calico supports Kubernetes Network Policies.

Container Network Interface (CNI) specification

The Container Network Interface specification formalizes the responsibilities of the **Container Runtime** versus **Network Plugins**. It defines *container* as a synonym for Linux network namespace and *network* as a group of entities that are uniquely addressable that can communicate amongst each other. According to the specification:

The **Container Runtime** is responsible for:

1. Creating a new network namespace for the container before invoking any plugins.
2. Determining which networks this container should belong to, and for each network, which plugins must be executed.
3. Adding the container to each network by executing the corresponding plugins for each network sequentially.
4. Executing the plugins in reverse order (relative to the order in which they were executed to add the container) to disconnect the container from the networks, upon completion of the container lifecycle

A **CNI plugin** is responsible for:

1. Inserting a network interface into the container network namespace (e.g. one end of a veth pair)
2. Attaching the other end of the veth into a bridge on the host
3. Assigning the IP to the interface
4. Setting up the routes consistent with the **IP Address Management** section by invoking appropriate **IPAM plugin**.

How can Pods communicate outside of the cluster?

Pods can be routable or non routable.

Not routable

- Pod IP address is not known outside of the cluster
- Connection from Pod to outside of the Cluster requires **SNAT (Source Network Address Translation)** to change the source IP address from the IP address of the Pod to the IP address of the Node hosting the Pod.
- Return packets are mapped automatically from the Node IP to the Pod IP
- No connections from outside the cluster to the Pod are possible (unless you create **Kubernetes Services** or **Kubernetes Ingress**)

Routable

- Pod IP address is known outside of the cluster, so the Pod IP must be unique in the broader network

Kubernetes Networking

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

Why do we need Kubernetes Services?

- **Each pod has its own IP address**
- **Pods are not permanent, so the IP addresses at which we access an application may change over time**
 - Pods that are part of a workload resource can be created and destroyed dynamically by the controller. For example, a crashed or deleted pod will be recreated by the ReplicaSet to maintain the desired number of replicas. A Horizontal Pod Autoscaler can create and delete pods automatically based on conditions. A manual scaling action can also create or destroy pods.
 - If a Pod is re-created after a deletion or crash, it will generally get a new IP address.
- **How can applications in pods refer to applications in other pods if their IP addresses are not permanent?**
 - In a typical web application, the web frontend needs to access the backend (API). But if the pod that serves the API unexpectedly fails and is automatically recreated, how can the frontend know how to access the new pod?
- **Services solve this problem by:**
 - Defining a logical set of pods
 - With a Selector (filter based on labels)
 - Without a Selector
 - Specifying how to access these pods

Service Types (1/2)

- **ClusterIP:**
 - Exposes the Service on a cluster-internal IP address.
 - The Service is only reachable from within the cluster.
 - This is the default ServiceType.
 - It gets created automatically when you create a NodePort or LoadBalancer Service
- **NodePort:**
 - Exposes the Service on **each** Node's IP at a static port (the NodePort) chosen from a **range (30000-32767)**.
 - The NodePort automatically creates a ClusterIP service
 - The NodePort Service routes to the ClusterIP service.
 - Clients can reach the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort> FOR EACH of the NODES of the CLUSTER (both MASTER and WORKER nodes).

Service Types (2/2)

- **LoadBalancer:**
 - Exposes the Service externally using an external load balancer (not included in K8s).
 - It automatically creates a ClusterIP and NodePort
 - On a public Cloud, the LoadBalancer implementation is provided by the IaaS.
 - On bare metal, you can use for example MetalLB: <https://metallb.universe.tf/>
 - The LoadBalancer Service routes to the NodePort and ClusterIP Services it created.
 - Clients can reach the LoadBalancer service at the LoadBalancer (external) IP.
 - If no Load Balancer is installed, and you try to create Service of type LoadBalancer, the ExternalIP remains in PENDING state forever

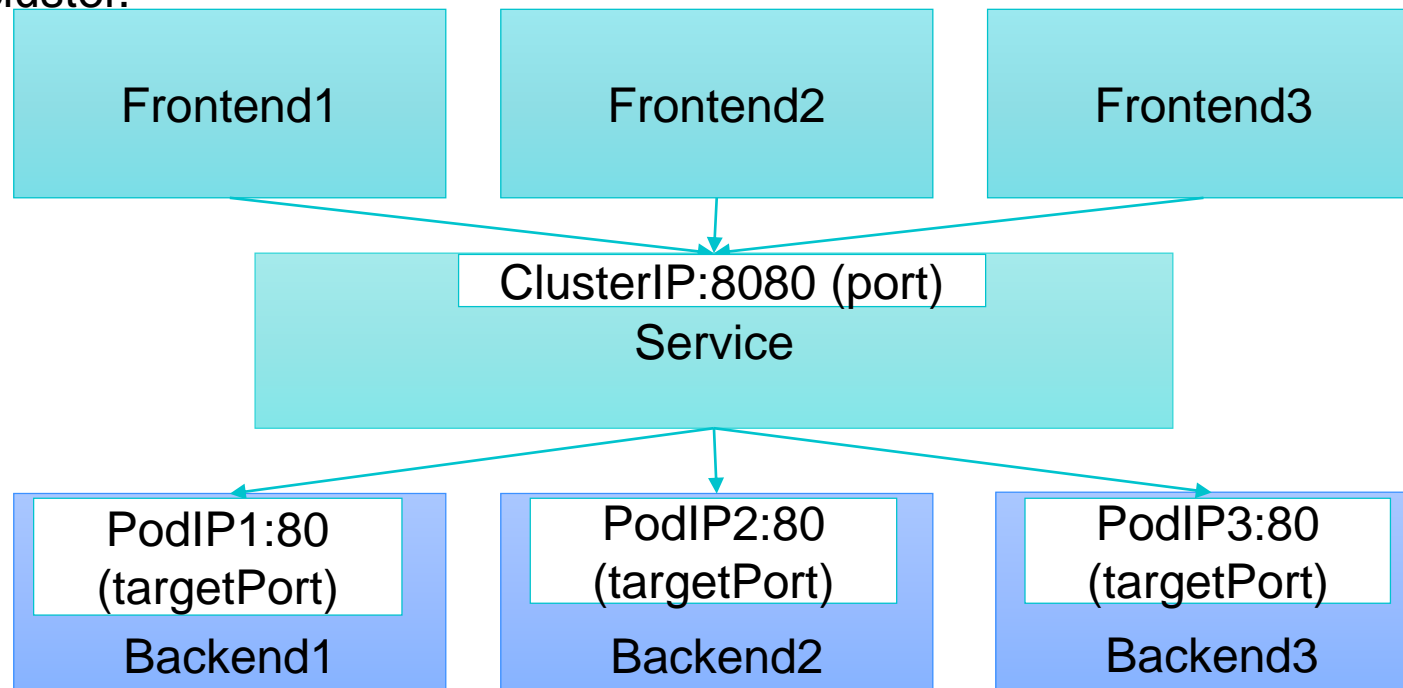
ClusterIP

A Service of type ClusterIP defines an IP address visible only inside the cluster, and a port it listens on. Requests made to this port and ClusterIP are forwarded to the targetPort of the Pods that the service identifies as targets.

A typical use case is to make backend pods accessible by frontend pods, without using the Pod IPs that are subject to change over time.

A ClusterIP load balances the incoming requests to all the target pods.

A ClusterIP cannot be used to expose a web site to external users that run their browser on a machine external to the cluster.



ClusterIP definition file example (using selector)

Assume that you have created a Deployment with a few Pods that provide a REST API. These pods have the label `app` with the value: `fancy-api`.

The REST API listens on port 80 inside the Pods.

Now you want to allow another set of pods, which provide the web front-end, to invoke the REST API.

You need to avoid that the Web front-end refers directly to any of the IP addresses of the REST API pods because they could change and because which one should you choose?

So you introduce a service of type ClusterIP called `fancy-api.service`. The ClusterIP exposes port 8081 and forwards any calls on that port to the `targetPort` 80 of any pods that have the label `app` with the value of `fancy-api`.

Note that the supported protocols are: "SCTP", "TCP", "UDP"

```
apiVersion: v1
kind: Service
metadata:
  name: fancy-api-service
spec:
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 8081
      targetPort: 80
  selector:
    app: fancy-api
```


Create a ClusterIP

To create a ClusterIP, you simply create the definition file and then apply the definition with:

```
kubectl apply -f my-service-definition.yaml
```

To view the existing service you use the commands:

```
kubectl get services
```

or

```
kubectl get service <service-name>
```

Or

```
Kubectl get service <service-name> -o wide
```

This shows the IP address that was assigned to service, the port over which it listens and the selector with which it finds the target pods:

```
lara@kube-master-gui:~/k8s_services$ kubectl get services
NAME                TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
kubernetes           ClusterIP     10.152.183.1    <none>           443/TCP          10d
inventory-api-service ClusterIP     10.152.183.253  <none>           8081/TCP          97m
lara@kube-master-gui:~/k8s_services$ kubectl get service inventory-api-service
NAME                TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
inventory-api-service ClusterIP     10.152.183.253  <none>           8081/TCP          97m
lara@kube-master-gui:~/k8s_services$ kubectl get service inventory-api-service -o wide
NAME                TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE      SELECTOR
inventory-api-service ClusterIP     10.152.183.253  <none>           8081/TCP          98m      app=inventory-api
```

Endpoints

When you create a service using a selector, Kubernetes can identify the target pods and their IP addresses.

It adds automatically an Endpoint object as in the following example:

```
lara@k8s-master-gui:~/k8s_services$ kubectl get endpoints
```

NAME	ENDPOINTS	AGE
kubernetes	10.0.2.15:16443	10d
inventory-api-service	10.1.96.149:5000,10.1.96.151:5000,10.1.96.155:5000	112m

```
lara@k8s-master-gui:~/k8s_services$ kubectl get endpoints inventory-api-  
service -o yaml  
apiVersion: v1  
kind: Endpoints  
...  
  name: inventory-api-service  
  namespace: default  
...  
subsets:  
- addresses:  
  - ip: 10.1.96.149  
    nodeName: kube-master-gui  
    targetRef:  
      kind: Pod  
      name: inventory-api-deployment-76745bddd7-km2c6  
      namespace: default  
      resourceVersion: "681543"  
      uid: 0b2a814b-7355-499c-a91d-89db2276ab44  
  - ip: 10.1.96.151  
...  
ports:  
  - port: 5000  
    protocol: TCP
```

The Endpoint shows the IP addresses and targetPorts of all the pods that the selector identified.

It also records the name and unique id of the pods as well as the ports.

Services without selector

You can also **create a Service without a Pod selector**.

This makes sense, for example, if you need to target a traditional database installation (not running inside the Kubernetes Cluster), or a pod in a different cluster.

In this case, Kubernetes cannot create the Endpoint Automatically as it has no knowledge of the target system.

You have to provide the Endpoint definition file yourself, and you have to supply the IP address(es) and port(s) at which the external application can be reached.

```
apiVersion: v1
kind: Endpoints
metadata:
  name: my-service
subsets:
  - addresses:
    - ip: 192.0.2.42
    ports:
      - port: 9376
```

Accessing a Service via Environment Variables

There are two ways for a client pod to access a service:

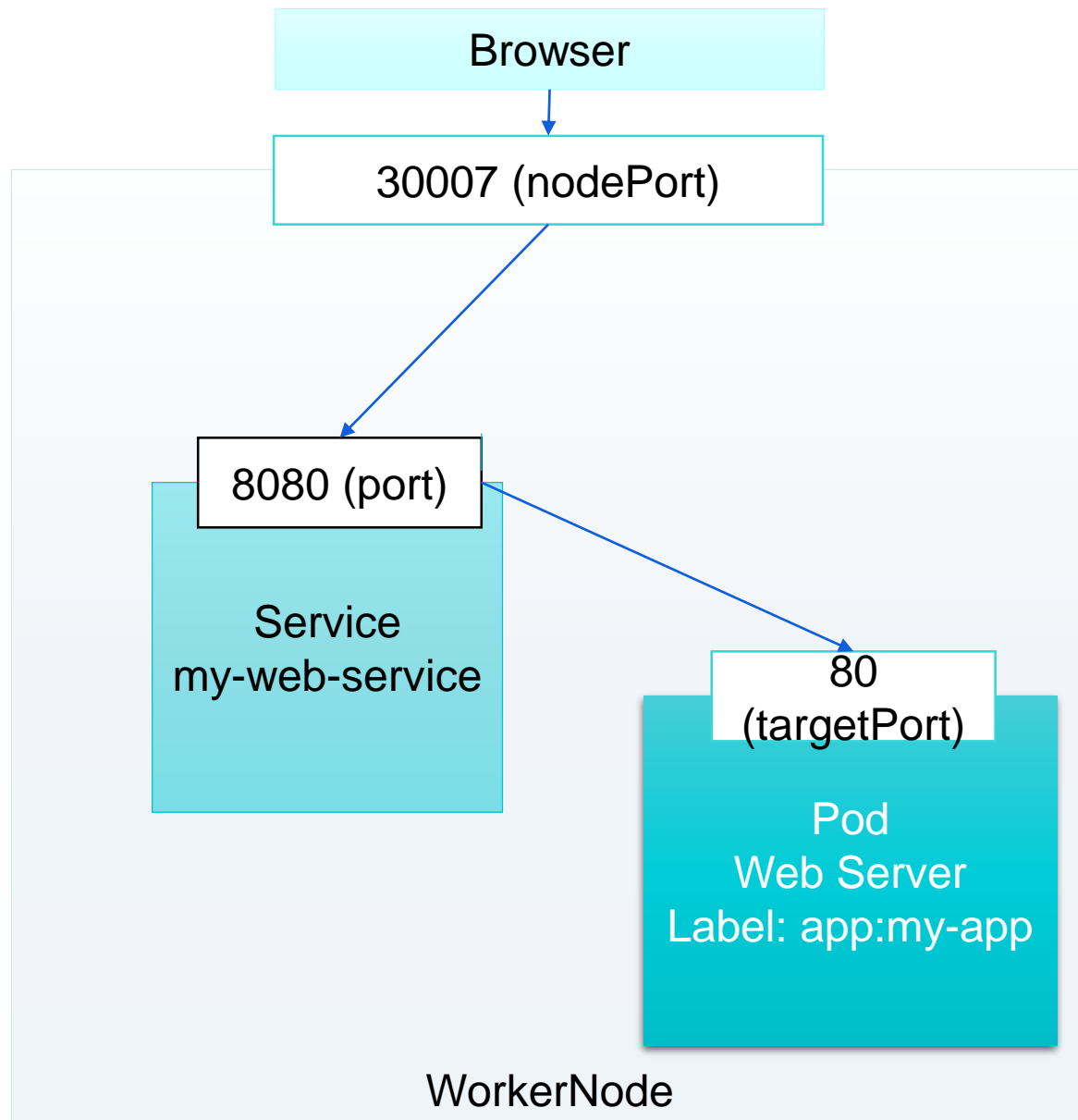
- Using environment variables
- Using CoreDNS

For now let's focus on environment variables. As long as the client pod is created AFTER the service, Kubernetes injects environment variables that allow the client pod to identify the service hostname and port:

```
lara@kube-master-gui:~/k8s_services$ kubectl run service-client-pod --image alpine -it
If you don't see a command prompt, try pressing enter.
/ # printenv | grep SERVICE
INVENTORY_API_SERVICE_PORT_8081_TCP_ADDR=10.152.183.253
KUBERNETES_SERVICE_PORT=443
INVENTORY_API_SERVICE_PORT_8081_TCP_PORT=8081
INVENTORY_API_SERVICE_PORT_8081_TCP_PROTO=tcp
INVENTORY_API_SERVICE_SERVICE_PORT=8081
INVENTORY_API_SERVICE_PORT=tcp://10.152.183.253:8081
INVENTORY_API_SERVICE_PORT_8081_TCP=tcp://10.152.183.253:8081
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_SERVICE_HOST=10.152.183.1
INVENTORY_API_SERVICE_SERVICE_HOST=10.152.183.253
/ #
```

<https://kubernetes.io/docs/concepts/services-networking/connect-applications-service/>

Services: NodePort (using selector)



The browser accesses NodeIP:NodePort on ANY node IP

```
apiVersion: v1
kind: Service
metadata:
  name: my-web-service
spec:
  type: NodePort
  selector:
    app: my-app
  ports:
    # By convention `targetPort`
    # has same value as `port`
    - port: 8080
      targetPort: 80
    # Optional field
    # By default k8s allocates a port
    # from 30000-32767
    nodePort: 30007
```

NodePort – multiple pods

Scenario 1: multiple pods in the same node

If there are multiple pods with the same selector label values in the same node, the NodePort Service will act as a load balancer and it will distribute the incoming traffic over all those pods.

The end user needs to connect to the IP address of the single node, using the port specified as the nodePort variable in the Service definition.

Scenario 2: multiple pods in multiple nodes

If multiple pods with the same selector label are spread among multiple worker nodes, the NodePort service will listen on the same nodePort on EACH node IP address.

The client application can send the request to any of the node IP addresses, using always the same nodePort, and it will be directed to any of the pods that the Service selector matches.

NodePort is the only practical way to expose a service to external users unless you have a real Load Balancer available.

Create a NodePort

To create a NodePort, you simply create the definition file and then apply the definition with:

```
kubectl apply -f my-service-definition.yaml
```

To view the existing service you use the commands:

```
kubectl get services
```

or

```
kubectl get service <service-name>
```

Or

```
Kubectl get service <service-name> -o wide
```

This shows the IP address that was assigned to service, the port over which it listens and the selector with which it finds the target pods:

```
lara@kube-master-gui:~/k8s_services$ kubectl get services -o wide
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE	SELECTOR
kubernetes	ClusterIP	10.152.183.1	<none>	443/TCP	10d	<none>
inventory-api-service	ClusterIP	10.152.183.253	<none>	8081/TCP	104m	app=inventory-api
inventory-ui-service	NodePort	10.152.183.137	<none>	8080:30007/TCP	14s	app=inventory-ui

Assuming the master node runs on 10.0.2.15, you can access the NodePort at 10.0.2.15:30007. If there are other nodes in the cluster, you can also use one of the other nodes IP addresses.

LoadBalancer

The Kubernetes project provides an interface for an object of type LoadBalancer, however it does not provide any implementation.

If you use microk8s, it comes with an add-on called metallb that provides a snap packaged installation of the MetalLB Load Balancer. You can enable it as follows:

```
lara@kube-master-gui:~/k8s_services$ microk8s enable metallb
Enabling MetalLB
Enter each IP address range delimited by comma (e.g. '10.64.140.43-10.64.140.49,192.168.0.105-192.168.0.111'): 10.50.100.5-10.50.100.25
Applying Metallb manifest
namespace/metallb-system created
secret/memberlist created
podsecuritypolicy.policy/controller created
podsecuritypolicy.policy/speaker created
serviceaccount/controller created
serviceaccount/speaker created
clusterrole.rbac.authorization.k8s.io/metallb-system:controller created
clusterrole.rbac.authorization.k8s.io/metallb-system:speaker created
role.rbac.authorization.k8s.io/config-watcher created
role.rbac.authorization.k8s.io/pod-lister created
clusterrolebinding.rbac.authorization.k8s.io/metallb-system:controller created
clusterrolebinding.rbac.authorization.k8s.io/metallb-system:speaker created
rolebinding.rbac.authorization.k8s.io/config-watcher created
rolebinding.rbac.authorization.k8s.io/pod-lister created
daemonset.apps/speaker created
deployment.apps/controller created
configmap/config created
MetalLB is enabled
```

Note that when you enable MetalLB it asks you for a range of IP addresses that it can use to associate with services. You need to provide a range of free IP addresses and it typically makes sense to provide public IPs if you want the services to be accessible over the Internet. In this example I used 25 private addresses in the range: 10.50.100.5 to 10.50.100.25

LoadBalancer service

To create a LoadBalancer service, you need to create a yaml file that creates an object of kind Service and type LoadBalancer.

You then specify a selector allows the Load Balancer to identify the pods to target.

You specify the targetPort as the port inside the container where the service runs (in this case, port 5000 is the default port where the Flask application listens).

You specify port as the port over which to access the Load Balancer service.

After you apply the file and get the description of the load balancer you can see the External IP address which is in the range you have provided during the installation (10.50.100.5 in this case)

```
apiVersion: v1
kind: Service
metadata:
  name: inventory-api-lb
spec:
  selector:
    app: inventory-api
  ports:
    - port: 8090
      targetPort: 5000
  type: LoadBalancer
```

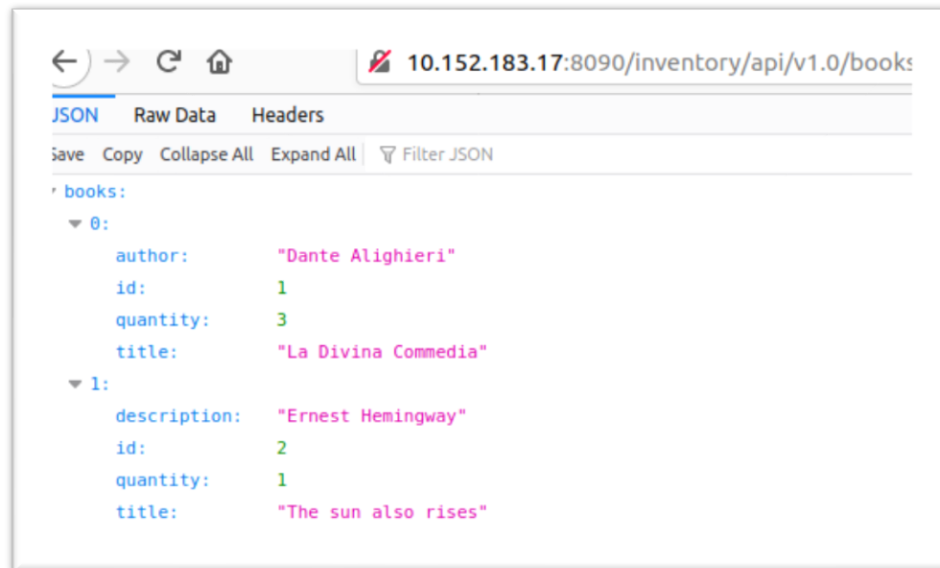
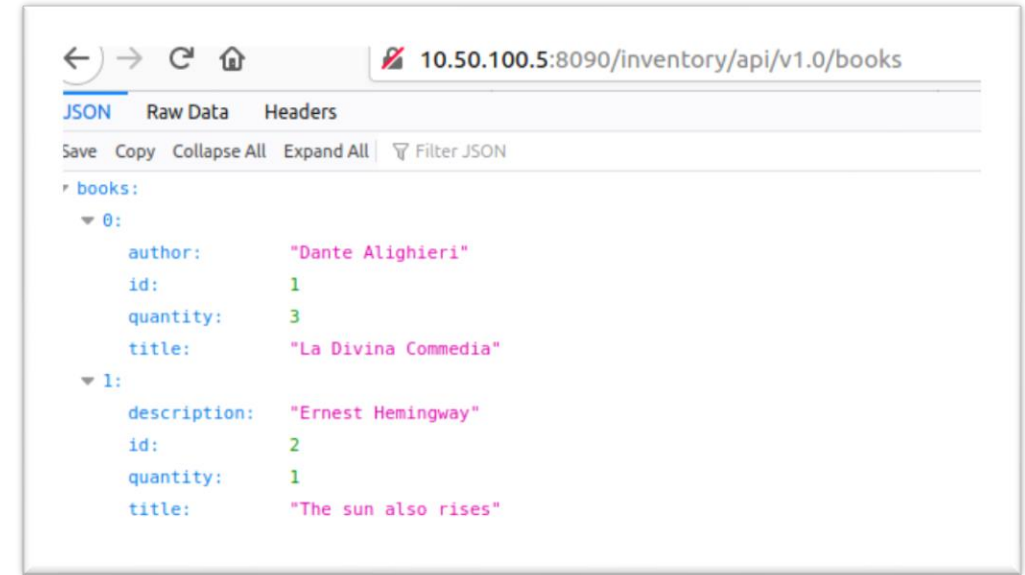
```
lara@kube-master-gui:~/k8s_services$ kubectl apply -f inventory-api-lb.yaml
service/inventory-api-lb created
lara@kube-master-gui:~/k8s_services$ kubectl get service -o wide
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE	SELECTOR
kubernetes	ClusterIP	10.152.183.1	<none>	443/TCP	10d	<none>
inventory-api-lb	LoadBalancer	10.152.183.17	10.50.100.5	8090:32576/TCP	14s	app=inventory-api

Accessing the Load Balancer Service

The Load Balancer created 3 services:

- ClusterIP: 10.152.183.17 with port 8090
This is only visible inside the cluster
- NodePort: localhost with port 32576.
If you had multiple nodes, this port would be accessible on the IP address of each of the nodes.
- LoadBalancer: 10.50.100.5 with port 8090
This IP address is in the range provided to MetalLB.
In a real-life scenario, this would be a public IP address.



kube_proxy

- Kubernetes Services are implemented by kube_proxy. kube_proxy can work in 4 different ways:
 - **userspace**: The earliest load balancing scheme, it listens to a port in user space, all services are forwarded to this port through iptables, and then load balanced to the actual Pod inside it. The main problem of this method is low efficiency and obvious performance bottleneck. This mode is already deprecated.
 - **iptables**: This implements service load balancing in the form of iptables rules (iptables is a Linux firewall). The main problem of this mechanism is that too many iptables rules are generated when there are too many services. Non-incremental updates will introduce a certain delay, and there are obvious performance problems in large-scale cases.
 - **ipvs**: IP Virtual Server (IPVS) is a Linux connection (L4) load balancer. In order to solve the performance problem of iptables mode, v1.11 adds ipvs mode, adopts incremental update, and can ensure that the connection remains continuous during service update open. (Using ipvs requires you need to pre-load kernel modules nf_conntrack_ipv4, ip_vs, ip_vs_rr, ip_vs_wrr, ip_vs_sh etc. on each Node)
 - **winuserspace**: Same as userspace, but only works on Windows nodes.

From: <https://medium.com/geekculture/k8s-kube-proxy-13a2a6e80364>

kube_proxy daemonset

```
$ kubectl describe daemonset kube-proxy -n kube-system
Name: kube-proxy
Selector: k8s-app=kube-proxy
Node-Selector: kubernetes.io/os=linux
Labels: k8s-app=kube-proxy
Annotations: deprecated.daemonset.template.generation: 1
Desired Number of Nodes Scheduled: 1
Current Number of Nodes Scheduled: 1
Number of Nodes Scheduled with Up-to-date Pods: 1
Number of Nodes Scheduled with Available Pods: 1
Number of Nodes Misscheduled: 0
Pods Status: 1 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels: k8s-app=kube-proxy
  Service Account: kube-proxy
  Containers:
    kube-proxy:
      Image: k8s.gcr.io/kube-proxy:v1.20.2
      Port: <none>
      Host Port: <none>
      Command:
        /usr/local/bin/kube-proxy
        --config=/var/lib/kube-proxy/config.conf
        --hostname-override=$(NODE_NAME)
  Environment:
    NODE_NAME: (v1:spec.nodeName)
  Mounts:
    /lib/modules from lib-modules (ro)
    /run/xtables.lock from xtables-lock (rw)
    /var/lib/kube-proxy from kube-proxy (rw)
```

```
Volumes:
  kube-proxy:
    Type: ConfigMap (a volume populated by a ConfigMap)
    Name: kube-proxy
    Optional: false
  xtables-lock:
    Type: HostPath (bare host directory volume)
    Path: /run/xtables.lock
    HostPathType: FileOrCreate
  lib-modules:
    Type: HostPath (bare host directory volume)
    Path: /lib/modules
    HostPathType:
      Priority Class Name: system-node-critical
Events:
  Type      Reason              Age    From              Message
  ----      -
  Normal    SuccessfulCreate    6m41s  daemonset-controller  Created
pod: kube-proxy-cnwdf
```

kube_proxy configuration

- The configuration is stored in a config map called kube-proxy in namespace kube-system

```
$ kubectl get cm kube-proxy -n kube-system -o yaml
apiVersion: v1
data:
  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
    conntrack:
      maxPerCore: null
      min: null
      tcpCloseWaitTimeout: null
      tcpEstablishedTimeout: null
    detectLocalMode: ""
    enableProfiling: false
    healthzBindAddress: ""
    hostnameOverride: ""
```

```
iptables:
  masqueradeAll: false
  masqueradeBit: null
  minSyncPeriod: 0s
  syncPeriod: 0s
ipvs:
  excludeCIDRs: null
  minSyncPeriod: 0s
  scheduler: ""
  strictARP: false
  syncPeriod: 0s
  tcpFinTimeout: 0s
  tcpTimeout: 0s
  udpTimeout: 0s
kind: KubeProxyConfiguration
metricsBindAddress: 0.0.0.0:10249
mode: ""
```

mode can be used to specify iptables or ipvs

From: <https://medium.com/geekculture/k8s-kube-proxy-13a2a6e80364>

iptables on Katacoda

Microk8s repackages kube-proxy in a custom way, but minikube shows the standard configuration.

The screenshot shows the iptable rules on Katacoda before starting minikube.

In the following 2 slides, see the changes to the rules after starting minikube.

Kubernetes Bootcamp Terminal

```
$ iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source                destination

Chain FORWARD (policy DROP)
target     prot opt source                destination
DOCKER-USER all  --  anywhere              anywhere
DOCKER-ISOLATION-STAGE-1 all  --  anywhere              anywhere
ACCEPT     all  --  anywhere              anywhere             ctstate RELATED,ESTABLISHED
DOCKER     all  --  anywhere              anywhere
ACCEPT     all  --  anywhere              anywhere
ACCEPT     all  --  anywhere              anywhere

Chain OUTPUT (policy ACCEPT)
target     prot opt source                destination

Chain DOCKER (1 references)
target     prot opt source                destination

Chain DOCKER-ISOLATION-STAGE-1 (1 references)
target     prot opt source                destination
DOCKER-ISOLATION-STAGE-2 all  --  anywhere              anywhere
RETURN     all  --  anywhere              anywhere

Chain DOCKER-ISOLATION-STAGE-2 (1 references)
target     prot opt source                destination
DROP       all  --  anywhere              anywhere
RETURN     all  --  anywhere              anywhere

Chain DOCKER-USER (1 references)
target     prot opt source                destination
RETURN     all  --  anywhere              anywhere
$
```



```

$ iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source                               destination
KUBE-EXTERNAL-SERVICES all -- anywhere                           anywhere        ctstate NEW /* kubernetes externally-visible service portals */
KUBE-FIREWALL all -- anywhere                           anywhere

Chain FORWARD (policy DROP)
target     prot opt source                               destination
KUBE-FORWARD all -- anywhere                           anywhere        /* kubernetes forwarding rules */
KUBE-SERVICES all -- anywhere                           anywhere        ctstate NEW /* kubernetes service portals */
KUBE-EXTERNAL-SERVICES all -- anywhere                           anywhere        ctstate NEW /* kubernetes externally-visible service portals */
DOCKER-USER all -- anywhere                           anywhere
DOCKER-ISOLATION-STAGE-1 all -- anywhere                           anywhere
ACCEPT     all -- anywhere                           anywhere        ctstate RELATED,ESTABLISHED
DOCKER     all -- anywhere                           anywhere
ACCEPT     all -- anywhere                           anywhere
ACCEPT     all -- anywhere                           anywhere

Chain OUTPUT (policy ACCEPT)
target     prot opt source                               destination
KUBE-SERVICES all -- anywhere                           anywhere        ctstate NEW /* kubernetes service portals */
KUBE-FIREWALL all -- anywhere                           anywhere

Chain DOCKER (1 references)
target     prot opt source                               destination

Chain DOCKER-ISOLATION-STAGE-1 (1 references)
target     prot opt source                               destination
DOCKER-ISOLATION-STAGE-2 all -- anywhere                           anywhere
RETURN     all -- anywhere                           anywhere

Chain DOCKER-ISOLATION-STAGE-2 (1 references)
target     prot opt source                               destination
DROP       all -- anywhere                           anywhere
RETURN     all -- anywhere                           anywhere

```


iptables on Katacoda – after starting minikube

```
Chain DOCKER-USER (1 references)
target      prot opt source                destination
RETURN      all  --  anywhere              anywhere

Chain KUBE-EXTERNAL-SERVICES (2 references)
target      prot opt source                destination

Chain KUBE-FIREWALL (2 references)
target      prot opt source                destination
DROP        all  --  anywhere              anywhere          /* kubernetes firewall for dropping marked packets */ mark match 0x8000/0x8000
DROP        all  --  !127.0.0.0/8          127.0.0.0/8          /* block incoming localnet connections */ ! ctstate RELATED,ESTABLISHED,DNAT

Chain KUBE-FORWARD (1 references)
target      prot opt source                destination
DROP        all  --  anywhere              anywhere              ctstate INVALID
ACCEPT      all  --  anywhere              anywhere              /* kubernetes forwarding rules */ mark match 0x4000/0x4000
ACCEPT      all  --  anywhere              anywhere              /* kubernetes forwarding conntrack pod source rule */ ctstate RELATED,ESTABLISHED
ACCEPT      all  --  anywhere              anywhere              /* kubernetes forwarding conntrack pod destination rule */ ctstate RELATED,ESTABLISHED
HED

Chain KUBE-KUBELET-CANARY (0 references)
target      prot opt source                destination

Chain KUBE-PROXY-CANARY (0 references)
target      prot opt source                destination

Chain KUBE-SERVICES (2 references)
target      prot opt source                destination
```

Inspecting a NodePort service and checking the iptables rules

```
root@kube-master-gui:/home/lara# microk8s kubectl describe svc rest-api-chart-1611572865
Name: rest-api-chart-1611572865
Namespace: default
Labels: app.kubernetes.io/instance=rest-api-chart-1611572865
        app.kubernetes.io/managed-by=Helm
        app.kubernetes.io/name=rest-api-chart
        app.kubernetes.io/version=v1
        helm.sh/chart=rest-api-chart-0.1.0
Annotations: <none>
Selector: app.kubernetes.io/instance=rest-api-chart-1611572865,app.kubernetes.io/name=rest-api-chart
Type: NodePort
IP Family Policy: SingleStack
IP Families: IPv4
IP: 10.152.183.80
IPs: 10.152.183.80
Port: http 80/TCP
TargetPort: 5000/TCP
NodePort: http 31662/TCP
Endpoints:
Session Affinity: None
External Traffic Policy: Cluster
Events: <none>
```

```
root@kube-master-gui:/home/lara# microk8s kubectl get endpoints rest-api-chart-1611572865
NAME                                ENDPOINTS                                AGE
rest-api-chart-1611572865          10.1.96.167:5000,10.1.96.183:5000       720d
```

Inspecting the KUBE_SERVICES iptables chain

```
sudo iptables -t nat -L KUBE-SERVICES -n | column -t
```

```
root@kube-master-gui:/home/lara# sudo iptables -t nat -L KUBE-SERVICES -n | column -t
Chain          KUBE-SERVICES (2 references)
target         prot      opt      source      destination
KUBE-MARK-MASQ  tcp      --      !10.1.0.0/16 10.152.183.247 /* istio-system/istio-telemetry:http-monitoring cluster IP */ tcp dpt:15014
KUBE-SVC-7NKQV7KRSMGKZMKF tcp      --      0.0.0.0/0    10.152.183.247 /* istio-system/istio-telemetry:http-monitoring cluster IP */ tcp dpt:15014
KUBE-MARK-MASQ  tcp      --      !10.1.0.0/16 10.152.183.1  /* default/kubernetes:https cluster IP */ tcp dpt:443
KUBE-SVC-NPX46M4PTMTKRN6Y tcp      --      0.0.0.0/0    10.152.183.1  /* default/kubernetes:https cluster IP */ tcp dpt:443
KUBE-MARK-MASQ  tcp      --      !10.1.0.0/16 10.152.183.192 /* kube-system/dashboard-metrics-scraper cluster IP */ tcp dpt:8000
KUBE-SVC-4GCQP7GTYLI53KTV tcp      --      0.0.0.0/0    10.152.183.192 /* kube-system/dashboard-metrics-scraper cluster IP */ tcp dpt:8000
KUBE-MARK-MASQ  tcp      --      !10.1.0.0/16 10.152.183.247 /* istio-system/istio-telemetry:prometheus cluster IP */ tcp dpt:42422
KUBE-SVC-SWAUWSHBU250T033 tcp      --      0.0.0.0/0    10.152.183.247 /* istio-system/istio-telemetry:prometheus cluster IP */ tcp dpt:42422
KUBE-MARK-MASQ  tcp      --      !10.1.0.0/16 10.152.183.191 /* kube-system/metrics-server cluster IP */ tcp dpt:443
KUBE-SVC-QMWWTXBG7KFJQKLO tcp      --      0.0.0.0/0    10.152.183.191 /* kube-system/metrics-server cluster IP */ tcp dpt:443
KUBE-MARK-MASQ  tcp      --      !10.1.0.0/16 10.152.183.247 /* istio-system/istio-telemetry:grpc-mixer cluster IP */ tcp dpt:9091
KUBE-SVC-LTOKVKL3D46WIGR3 tcp      --      0.0.0.0/0    10.152.183.247 /* istio-system/istio-telemetry:grpc-mixer cluster IP */ tcp dpt:9091
KUBE-MARK-MASQ  tcp      --      !10.1.0.0/16 10.152.183.247 /* istio-system/istio-telemetry:grpc-mixer-mtls cluster IP */ tcp dpt:15004
KUBE-SVC-POFVSRMRNLJ5KKAQ tcp      --      0.0.0.0/0    10.152.183.247 /* istio-system/istio-telemetry:grpc-mixer-mtls cluster IP */ tcp dpt:15004
KUBE-NODEPORTS all      --      0.0.0.0/0    0.0.0.0/0     /* kubernetes service nodeports; NOTE: this must be the last rule in this chain */
chain */ ADDRTYPE match dst-type LOCAL
```

```
KUBE-NODEPORTS      all      --      0.0.0.0/0    0.0.0.0/0     /* kubernetes
service nodeports; NOTE: this must be the last rule in this chain */ ADDRTYPE match dst-type LOCAL
```

Inspecting the KUBE_NODEPORTS iptables chain

```
sudo iptables -t nat -L <chain_name> -n | column -t
```

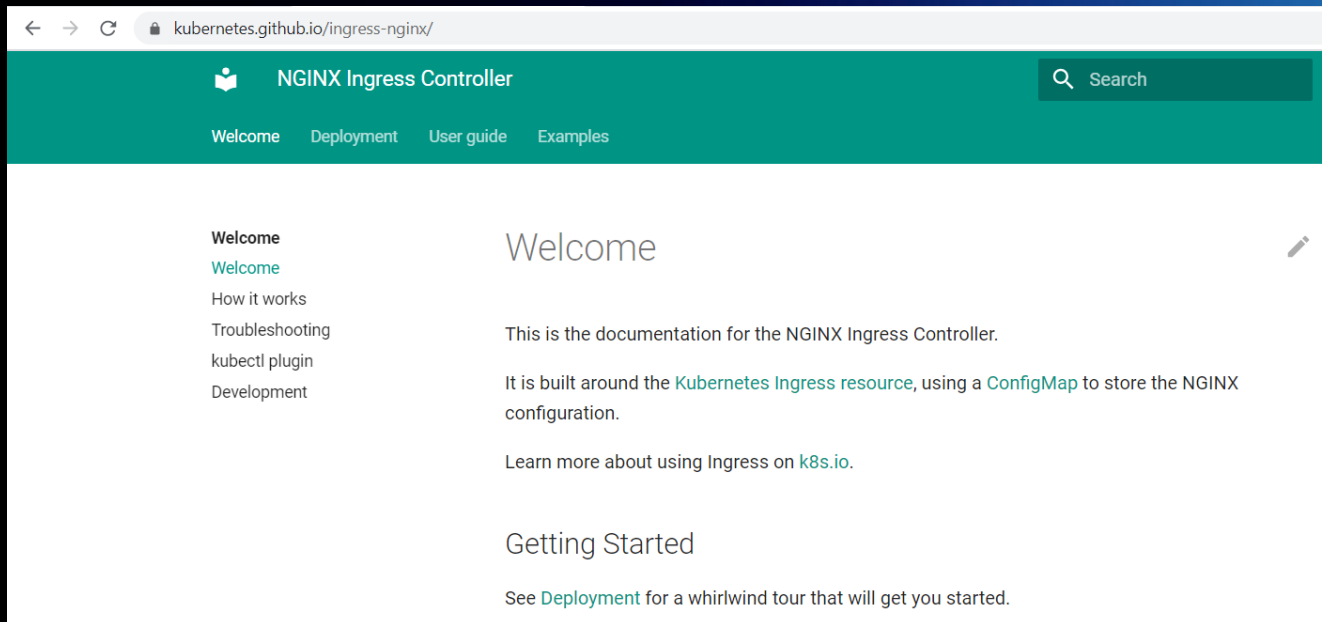
```
root@kube-master-gui:/home/lara# sudo iptables -t nat -L KUBE-NODEPORTS -n | column -t
Chain          KUBE-NODEPORTS (1 references)
target         prot      opt      source      destination
KUBE-MARK-MASQ  tcp      --      0.0.0.0/0   0.0.0.0/0   /* default/rest-api-chart-1611572865:http */ tcp dpt:31662
KUBE-SVC-BTVA6ANVIINMRVTN tcp      --      0.0.0.0/0   0.0.0.0/0   /* default/rest-api-chart-1611572865:http */ tcp dpt:31662
```

Note the destination port is 31662 which is the NodePort

```
root@kube-master-gui:/home/lara# sudo iptables -t nat -L KUBE-MARK-MASQ -n | column -t
Chain          KUBE-MARK-MASQ (21 references)
target prot      opt      source      destination
MARK all      --      0.0.0.0/0   0.0.0.0/0   MARK or 0x4000
root@kube-master-gui:/home/lara# sudo iptables -t nat -L KUBE-SVC-BTVA6ANVIINMRVTN -n | column -t
Chain          KUBE-SVC-BTVA6ANVIINMRVTN (2 references)
target         prot      opt      source      destination
KUBE-SEP-GLFEPICM2UBXEVL2 all      --      0.0.0.0/0   0.0.0.0/0   /* default/rest-api-chart-1611572865:http */
root@kube-master-gui:/home/lara# sudo iptables -t nat -L KUBE-SEP-GLFEPICM2UBXEVL2 -n | column -t
Chain          KUBE-SEP-GLFEPICM2UBXEVL2 (1 references)
target         prot      opt      source      destination
KUBE-MARK-MASQ all      --      10.1.96.183 0.0.0.0/0   /* default/rest-api-chart-1611572865:http */
DNAT           tcp      --      0.0.0.0/0   0.0.0.0/0   /* default/rest-api-chart-1611572865:http */ tcp to:10.1.96.183:5000
```

Note that the source is 10.1.96.183, which is the IP address of one of the two pods in the deployment DNAT (Destination Network Address Translation) redirects the packages to the Pod IP 10.1.96.183 on the TargetPort 5000

Kubernetes Networking Ingress



Ingress

Ingress is a Layer 7 load balancer configurable declaratively inside the Kubernetes cluster.

- Layer 4 Load Balancers operate at the transport layer using the Transmission Control Protocol (TCP). They forward packets after inspecting the first few packets in the stream.
- Layer 7 Load Balancers operate at the application layer using the HTTP protocol. They can take routing decision based on the full content of the message.

You may want to use Ingress when you have multiple services that you want to expose externally.

Typically, you might want to establish routes such that:

<https://my-server.my-domain/serviceA> points to a Service that exposes ApplicationA

<https://my-server.my-domain/serviceB> points to a Service that exposes ApplicationB

Or:

<https://serviceA.my-server.my-domain/> points to a Service that exposes ApplicationA

<https://serviceB.my-server.my-domain/> points to a Service that exposes ApplicationB

To use Ingress, you need to perform two steps:

1. Install an Ingress Controller (it is not part of Kubernetes cluster by default)
2. Create Ingress Resources (these provide the routes that configure the controller behavior)

Ingress Controller

There are various Ingress controllers that you could use:

- Maintained by the Kubernetes development team:
 - NGINX Ingress controller – based on NGINX
 - Google HTTPS Load Balancer (part of Google Compute Engine) if you use Google Kubernetes Engine
- Maintained by others:
 - HAProxy Ingress – based on HAProxy
 - Traefik kubernetes Ingress controller – based on Traefik
 - Istio is a Service Mesh that exposes its own Istio Gateway, but it can also use the Kubernetes Ingress Resource.
 - Microk8s Ingress Addon enables and NGINX Ingress Controller on Microk8s

Enabling Ingress in microk8s

Enabling ingress in microk8s is done with the command:

```
microk8s enable ingress
```

```
lara@k8s-master-gui:~/k8s_services$ microk8s enable ingress
Enabling Ingress
ingressclass.networking.k8s.io/public created
namespace/ingress created
serviceaccount/nginx-ingress-microk8s-serviceaccount created
clusterrole.rbac.authorization.k8s.io/nginx-ingress-microk8s-clusterrole created
role.rbac.authorization.k8s.io/nginx-ingress-microk8s-role created
clusterrolebinding.rbac.authorization.k8s.io/nginx-ingress-microk8s created
rolebinding.rbac.authorization.k8s.io/nginx-ingress-microk8s created
configmap/nginx-load-balancer-microk8s-conf created
configmap/nginx-ingress-tcp-microk8s-conf created
configmap/nginx-ingress-udp-microk8s-conf created
daemonset.apps/nginx-ingress-microk8s-controller created
Ingress is enabled
```

This creates a number of resources among which a namespace, a pod and a daemonset:

```
lara@k8s-master-gui:~/k8s_services$ kubectl get daemonset -n ingress
NAME                                DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
nginx-ingress-microk8s-controller    1         1         1       1             1           <none>          89s

lara@k8s-master-gui:~/k8s_services$ kubectl get pods -n ingress
NAME                                READY   STATUS    RESTARTS   AGE
nginx-ingress-microk8s-controller-hkwj7  1/1     Running   0          98s
```


Ingress Resources

You can now create an ingress resource that exposes the existing service (in this case, the LoadBalancer service) at the root of the my-inventory.com URL on port 80 (http).

The path refers to the LoadBalancer service by name (you don't need to specify the IP address)
It needs to specify the port that the LoadBalancer service listens on (8090).

For this to work locally without a proper DNS server setup, you need to add the following to /etc/hosts:

127.0.0.1 my-inventory.com

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: api-inventory-ingress
spec:
  rules:
    - host: my-inventory.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: inventory-api-lb
                port:
                  number: 8090
```

Testing the Ingress

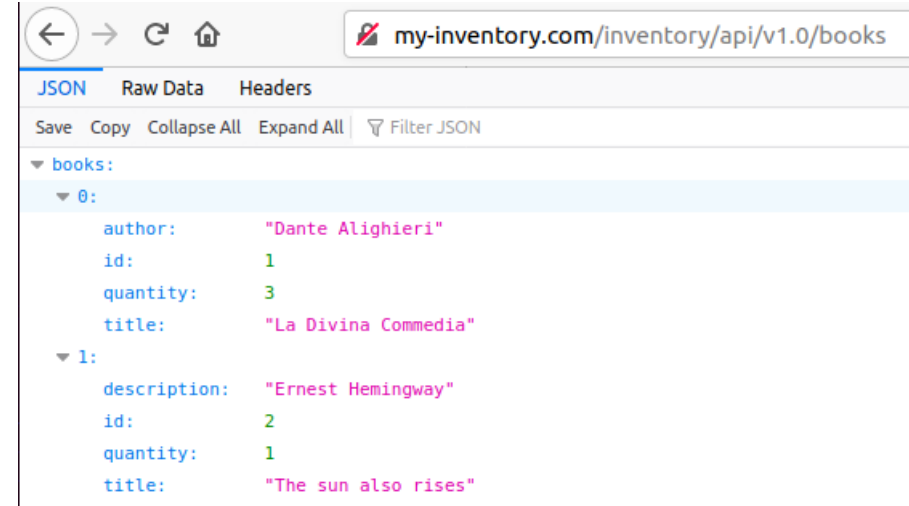
You can now test the Ingress by accessing:

<http://my-inventory.com/inventory/api/v1.0/books>

If you describe the ingress you can see the Rules with each:

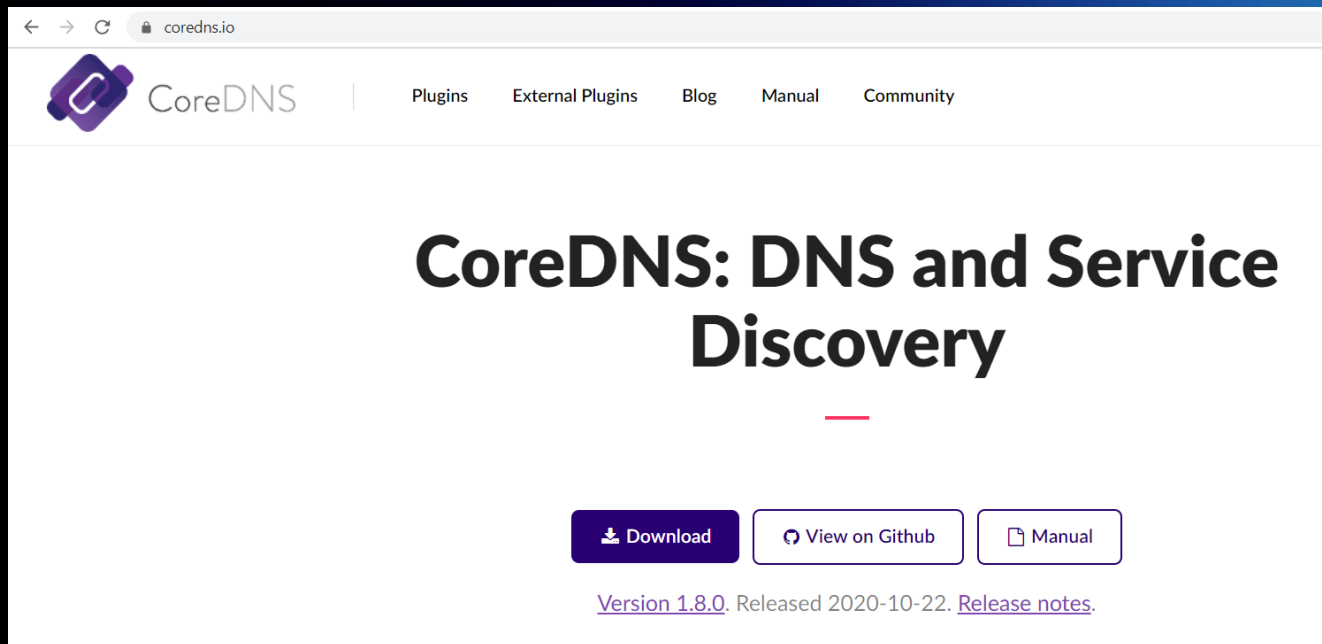
Host, Path, Backends

(You can ignore the error about the missing default backend).



```
lara@kube-master-gui:~/k8s_services$ kubectl describe ingress api-inventory-ingress
Name:          api-inventory-ingress
Namespace:     default
Address:       127.0.0.1
Default backend: default-http-backend:80 (<error: endpoints "default-http-backend" not found>)
Rules:
  Host        Path  Backends
  ----        -
  my-inventory.com  /    inventory-api-lb:8090 (10.1.96.149:5000,10.1.96.151:5000,10.1.96.155:5000)
Annotations:  <none>
Events:
  Type    Reason      Age    From                      Message
  ----    -
  Normal  CREATE      7m5s   nginx-ingress-controller  Ingress default/api-inventory-ingress
  Normal  UPDATE      6m14s  nginx-ingress-controller  Ingress default/api-inventory-ingress
```

Kubernetes Networking CoreDNS



CoreDNS: basic files related to DNS

CoreDNS is a graduated CNCF project that provides a nameserver for Kubernetes (replacing kube-dns).

In the simplest form, a Unix/Linux machine resolves the name of another machine to its IP address by looking up entries in the local file:

```
/etc/hosts  
ip-address1 name1  
ip-address2 name2
```

A Domain Name System (DNS) nameserver contains such entries from many machines and the nameserver can be referenced by a Unix/Client by listing its address in the file:

```
/etc/resolv.conf  
search mydomain.com  
nameserver 8.8.8.8
```

(The search line lists domains that should be added to the hostname to search for a Fully Qualified Domain Name)

CoreDNS: basic commands related to DNS

The nslookup command looks up the ip address corresponding to a domain name by interrogating the configured nameserver

```
lara@kube-master-gui:~$ more /etc/resolv.conf
# This file is managed by man:systemd-resolved(8). Do not edit.
#
# This is a dynamic resolv.conf file for connecting local clients to the
# internal DNS stub resolver of systemd-resolved. This file lists all
# configured search domains.
#
# Run "resolvectl status" to see details about the uplink DNS servers
# currently in use.
#
# Third party programs must not access this file directly, but only through the
# symlink at /etc/resolv.conf. To manage man:resolv.conf(5) in a different way,
# replace this symlink by a static file or a different symlink.
#
# See man:systemd-resolved.service(8) for details about the supported modes of
# operation for /etc/resolv.conf.

nameserver 127.0.0.53
options edns0
search home
```

The dig command allows you to learn more information about the DNS entry for a particular server, such as the record type:

```
lara@kube-master-gui:~$ nslookup google.com
Server:      127.0.0.53
Address:     127.0.0.53#53

Non-authoritative answer:
Name:   google.com
Address: 172.217.20.110
Name:   google.com
Address: 2a00:1450:400e:80e::200e

lara@kube-master-gui:~$ nslookup vu.nl
Server:      127.0.0.53
Address:     127.0.0.53#53

Non-authoritative answer:
Name:   vu.nl
Address: 37.60.194.64
Name:   vu.nl
Address: 2001:4d60:12::64
```

```
lara@kube-master-gui:~$ dig google.com

; <<>> DiG 9.16.1-Ubuntu <<>> google.com
;; global options: +cmd
;; Got answer:
;; ->HEADER<- opcode: QUERY, status: NOERROR, id: 41480
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
;; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;google.com.                IN      A

;; ANSWER SECTION:
google.com.                 173     IN      A      172.217.20.78

;; Query time: 12 msec
;; SERVER: 127.0.0.53#53(127.0.0.53)
;; WHEN: do jan 07 19:16:05 CET 2021
;; MSG SIZE rcvd: 55
```

CoreDNS: installation on microk8s

In microk8s, CoreDNS can be enabled as an add-on with the command:

```
microk8s enable dns
```

```
lara@kube-master-gui:~$ microk8s status
microk8s is running
high-availability: no
  datastore master nodes: 127.0.0.1:19001
  datastore standby nodes: none
addons:
  enabled:
    dns                # CoreDNS
    fluentd            # Elasticsearch-Fluentd-Kibana logging and monitoring
    ha-cluster         # Configure high availability on the current node
    storage            # Storage class; allocates storage from host directory
```

CoreDNS runs as a pod in the kube-system namespace:

```
lara@kube-master-gui:~$ kubectl get pods -n kube-system
NAME                                READY   STATUS    RESTARTS   AGE
hostpath-provisioner-5c65fadb4f-j824g 1/1     Running   8          4d3h
elasticsearch-logging-0               1/1     Running   6          3d21h
coredns-86f78bb79c-l6fk8             1/1     Running   9          4d3h
calico-node-vzfll                    1/1     Running   40         7d3h
calico-kube-controllers-847c8c99d-qmx9q 1/1     Running   38         7d3h
fluentd-es-v3.0.2-r8n46              1/1     Running   16         3d21h
kibana-logging-7cf6dc4687-scljs       1/1     Running   14         3d21h
```

CoreDNS is exposed as a service in the kube-system namespace. It listens on ports 53/UDP, 53/TCP:

```
lara@kube-master-gui:~$ kubectl get svc -n kube-system
NAME            TYPE        CLUSTER-IP    EXTERNAL-IP  PORT(S)                                AGE
kube-dns        ClusterIP   10.152.183.10 <none>       53/UDP,53/TCP,9153/TCP                4d4h
elasticsearch-logging ClusterIP   10.152.183.202 <none>       9200/TCP                               3d21h
kibana-logging  ClusterIP   10.152.183.46  <none>       5601/TCP                               3d21h
```

CoreDNS: configmap for Corefile

CoreDNS stores its configuration in a file called **Corefile**.

In Kubernetes, the data of such file is stored in an object called **ConfigMap**.

You can print the contents of the ConfigMap in json or yaml format and you can edit it with `kubectl edit configmap`.

The file shows a number of plugins in the data section.

Pods is responsible for publishing the pod ip addresses.

Forward send requests to the Google DNS.

```
lara@kube-master-gui:~$ kubectl -n kube-system get configmap coredns -o json
{
  "apiVersion": "v1",
  "data": {
    "Corefile": ".:53 {\n
errors\n
health {\n    lameduck 5s\n  }\n
ready\n
log . {\n    class error\n  }\n
kubernetes cluster.local in-addr.arpa ip6.arpa {\n
  pods insecure\n
  fallthrough in-addr.arpa ip6.arpa\n  }\n
prometheus :9153\n
forward . 8.8.8.8 8.8.4.4 \n
cache 30\n
loop\n
reload\n
loadbalance\n}\n"
  },
  "kind": "ConfigMap",
```

CoreDNS: Pod configuration

Inside a pod, the /etc/hosts file contains the pod name and its address.

The /etc/resolv.conf file contains the ip address of the CoreDNS Service, in this case 10.152.183.10.

It searches the domains:

- default.svc.cluster.local
- svc.cluster.local
- cluster.local
- home

The resolution of an external name like google.com is done via the CoreDNS nameserver through the forward plugin.

Note that another pod in the same cluster cannot resolve the pod alpine1 by name.

```
lara@kube-master-gui:~$ kubectl run -it alpine1 --image alpine -- ash
If you don't see a command prompt, try pressing enter.
/ # more /etc/hosts
# Kubernetes-managed hosts file.
127.0.0.1        localhost
::1             localhost ip6-localhost ip6-loopback
fe00::0          ip6-localnet
fe00::0          ip6-mcastprefix
fe00::1          ip6-allnodes
fe00::2          ip6-allrouters
10.1.96.168      alpine1
/ # more /etc/resolv.conf
search default.svc.cluster.local svc.cluster.local cluster.local home
nameserver 10.152.183.10
options ndots:5
/ # nslookup google.com
Server:         10.152.183.10
Address:        10.152.183.10:53

Non-authoritative answer:
Name:   google.com
Address: 216.58.208.110

Non-authoritative answer:
Name:   google.com
Address: 2a00:1450:400e:80c::200e
```


CoreDNS: lookup of services from pods

We saw previously that a client pod created after a service gets environment variables injected that allow it to find the service IP and port.

If CoreDNS is enabled, then the client pod can also make an nslookup.

In the example, attempting to lookup the service name:

inventory-api-service

Results in finding the name:

inventory-api-service.default.svc.cluster.local

mapped to 10.152.183.253 that is indeed the ClusterIP of that service.

```
lara@kube-master-gui:~/k8s_services$ kubectl run service-client-pod --image alpine -it
If you don't see a command prompt, try pressing enter.
/ # printenv | grep SERVICE
INVENTORY_API_SERVICE_PORT_8081_TCP_ADDR=10.152.183.253
KUBERNETES_SERVICE_PORT=443
INVENTORY_API_SERVICE_PORT_8081_TCP_PORT=8081
INVENTORY_API_SERVICE_PORT_8081_TCP_PROTO=tcp
INVENTORY_API_SERVICE_SERVICE_PORT=8081
INVENTORY_API_SERVICE_PORT=tcp://10.152.183.253:8081
INVENTORY_API_SERVICE_PORT_8081_TCP=tcp://10.152.183.253:8081
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_SERVICE_HOST=10.152.183.1
INVENTORY_API_SERVICE_SERVICE_HOST=10.152.183.253
/ # nslookup inventory-api-service
Server:          10.152.183.10
Address:         10.152.183.10:53

** server can't find inventory-api-service.cluster.local: NXDOMAIN

Name:   inventory-api-service.default.svc.cluster.local
Address: 10.152.183.253

** server can't find inventory-api-service.svc.cluster.local: NXDOMAIN
** server can't find inventory-api-service.svc.cluster.local: NXDOMAIN
** server can't find inventory-api-service.cluster.local: NXDOMAIN
** server can't find inventory-api-service.home: NXDOMAIN
** server can't find inventory-api-service.home: NXDOMAIN

/ #
```

```
lara@kube-master-gui:~/k8s_services$ kubectl get service inventory-api-service
NAME                                TYPE        CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE
inventory-api-service               ClusterIP    10.152.183.253 <none>       8081/TCP    152m
```

HCLSoftware

Kubernetes Networking Network policies

Network Policies

Network policies are implemented by a Network Plugin. If you define a Network Policy but use a plugin that does not support Network Policies, the Kubernetes object will be created but will have no effect.

Calico supports Network policies as described here: <https://docs.projectcalico.org/security/adopt-zero-trust>

By default, pods accept traffic from any source in the Cluster.

Pods become isolated by having a NetworkPolicy that selects them.

If you create Network Policies that select a Pod, then the Pod is restricted to what is allowed by the union of those policies' ingress and egress rules. Because of the union, the order of application of the policies does not matter.

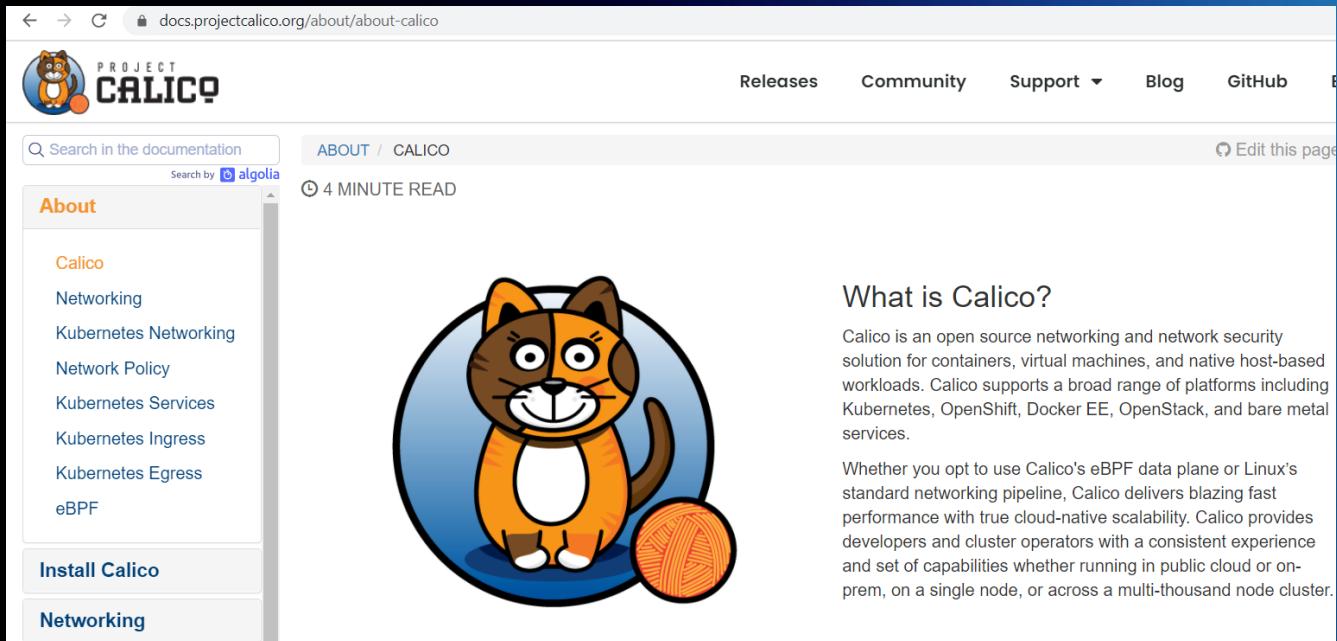
For a network flow between two pods to be allowed, both the egress policy on the source pod and the ingress policy on the destination pod need to allow the traffic.

For examples of network policies, see references below and later lesson.

<https://kubernetes.io/docs/concepts/services-networking/network-policies/>

<https://docs.projectcalico.org/security/kubernetes-network-policy>

Kubernetes Networking Calico



Calico CNI Plugins

There are two types of CNI plugins:

CNI network plugins: responsible for adding or deleting pods to/from the Kubernetes pod network. This includes creating/deleting each pod's network interface and connecting/disconnecting it to the rest of the network implementation.

CNI IPAM plugins: responsible for allocating and releasing IP addresses for pods as they are created or deleted. Depending on the plugin, this may include allocating one or more ranges of IP addresses (CIDRs) to each node, or obtaining IP addresses from an underlying public cloud's network to allocate to pods.

Calico CNI network plugin

The Calico CNI network plugin connects pods to the **host network namespace's L3 routing** using a pair of virtual ethernet devices (veth pair). This L3 architecture avoids the unnecessary complexity and performance overheads of additional **L2 bridges** that feature in many other Kubernetes networking solutions.

Calico CNI IPAM plugin

The Calico CNI IPAM plugin allocates IP addresses for pods out of one or more configurable IP address ranges, dynamically allocating small blocks of IPs per node as required. The result is a more efficient IP address space usage compared to many other CNI IPAM plugins, including the **host local IPAM plugin** which is used in many networking solutions.

<https://docs.projectcalico.org/networking/determine-best-networking>

How Calico is configured on Microk8s

<https://docs.projectcalico.org/getting-started/kubernetes/microk8s>

Policy	IPAM	CNI	Overlay	Routing	Datastore
Calico	Calico	Calico	VXLAN	Calico	Kubernetes

- **Policy= Calico** The Calico plugin implements the full set of Kubernetes network policy features. In addition, Calico supports Calico network policies, providing additional features and capabilities beyond Kubernetes network policies.
- **IPAM = Calico** The IP Address Management plugin determines how Kubernetes assigns IP addresses to Pods. The Calico IPAM plugin dynamically allocates small blocks of IP addresses to nodes as required and supports multiple IP pools, the ability to specify a specific IP address range that a namespace or pod should use, or even the specific IP address a pod should use.
- **CNI = Calico** The Calico CNI plugin connects pods to the host networking using L3 routing, without the need for an L2 bridge.
- **Overlay = VXLAN** An overlay network allows pods to communicate between nodes without the underlying network being aware of the pods or pod IP addresses. VXLAN wraps each original packet in an outer packet that uses node IPs, and hides the pod IPs of the inner packet.
- **Routing = Calico** Calico routing distributes and programs routes for pod traffic between nodes using its data store without the need for BGP (Boundary Gateway Protocol).
- **Datastore = Kubernetes** Calico has two datastore drivers you can choose from:
 - etcd - for direct connection to an etcd cluster
 - Kubernetes - for connection to a Kubernetes API server

Calico networking in detail: reference videos

Kubernetes Services Networking

https://www.youtube.com/watch?v=NFapeJRXos4&list=PLoWxE_5hnZUZMWrEON3wxMBolZvweGeiq&index=4

Kubernetes Ingress Networking

https://www.youtube.com/watch?v=40VfZ_nIFWI&list=PLoWxE_5hnZUZMWrEON3wxMBolZvweGeiq&index=5

HCLSoftware

hcltechsw.com