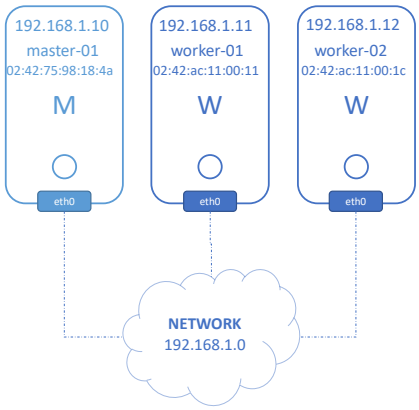
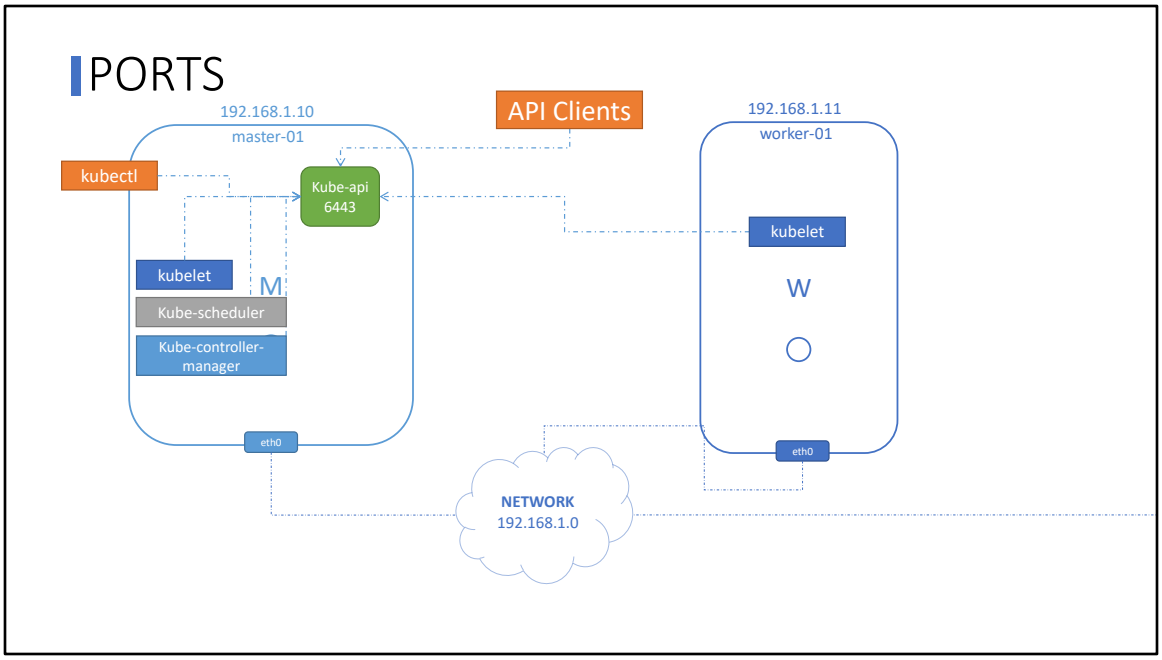

KodeKloud

In this lecture we look at the networking configurations required on the master and worker nodes.

IP & FQDN

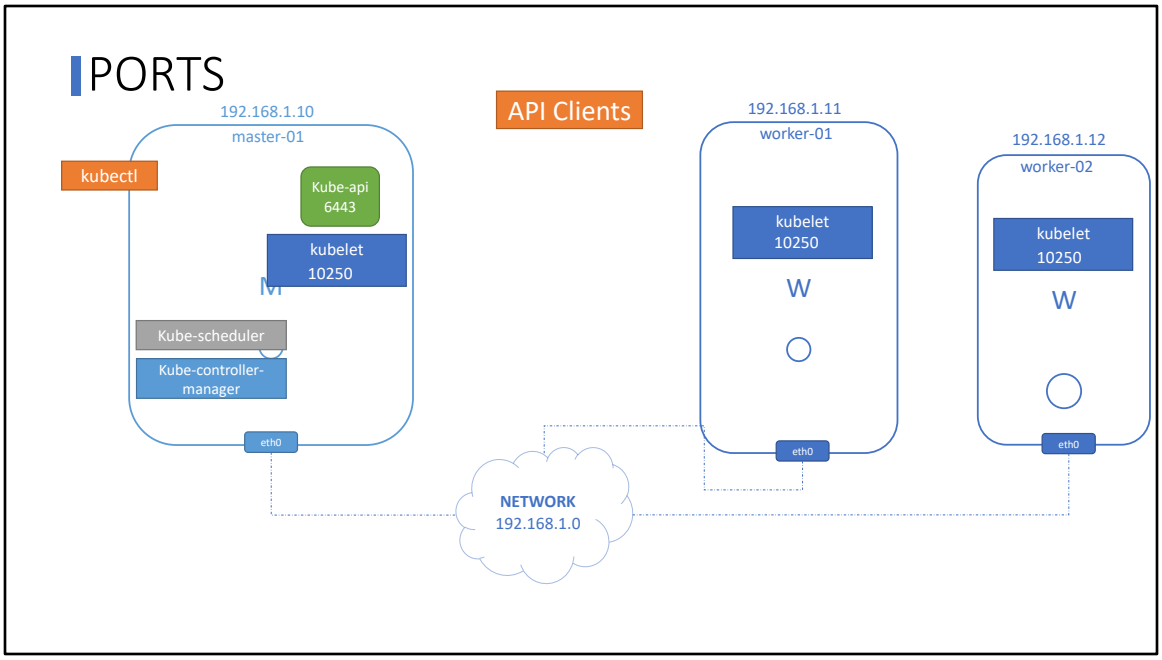


The kubernetes cluster consists of master and worker nodes. Each node must have at least 1 interface connected to a network. Each interface must have an address configured. The hosts must have a unique hostname set. As well as a unique MAC address. You should note this especially if you created the VMs by cloning from existing ones.

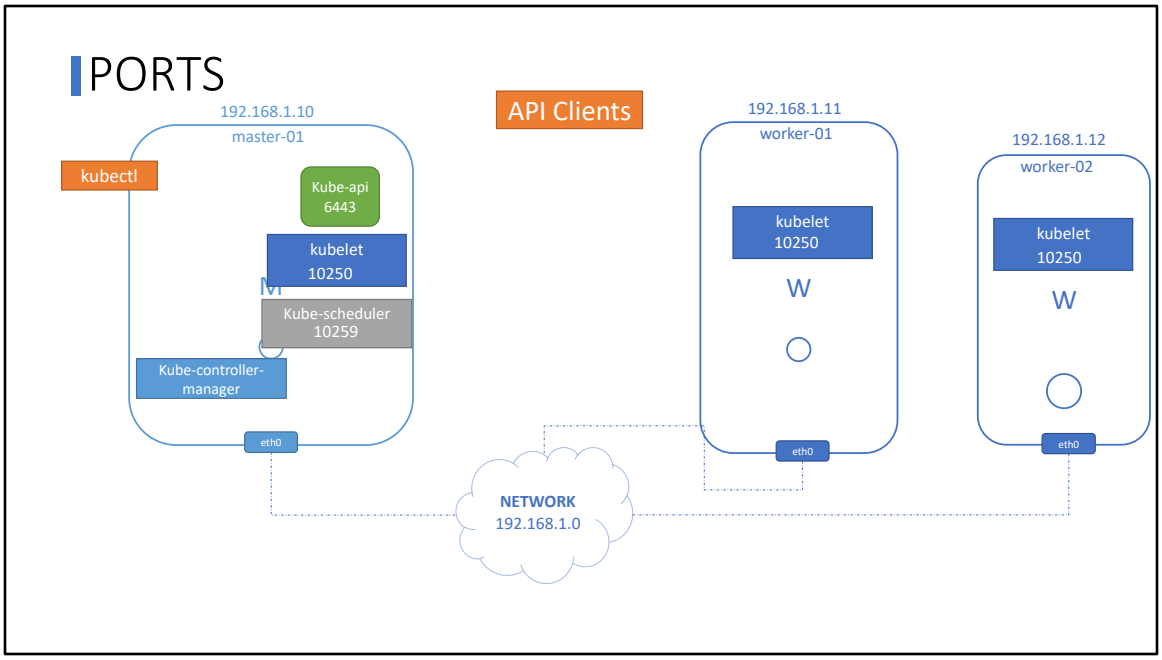


There are some ports that needs to be opened. These are used by the various components in the control plane.

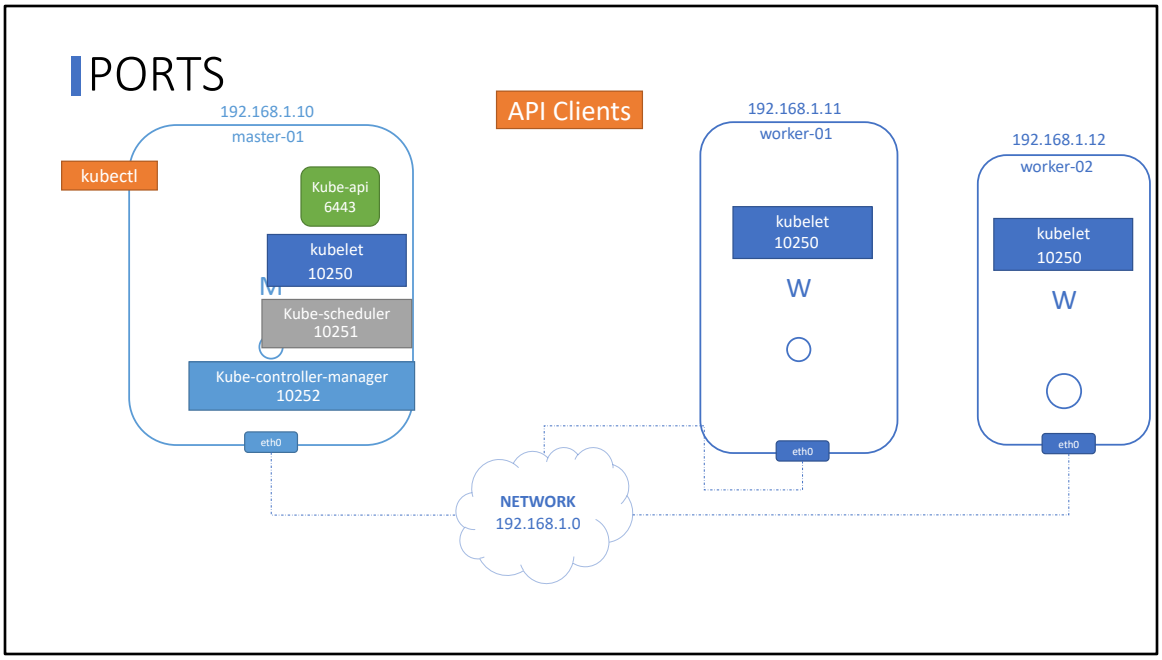
The master should accept connections on 6443 for the API server. The worker nodes, Kubectl tool, external users, and all other control plane components access the kube-api server via this port.



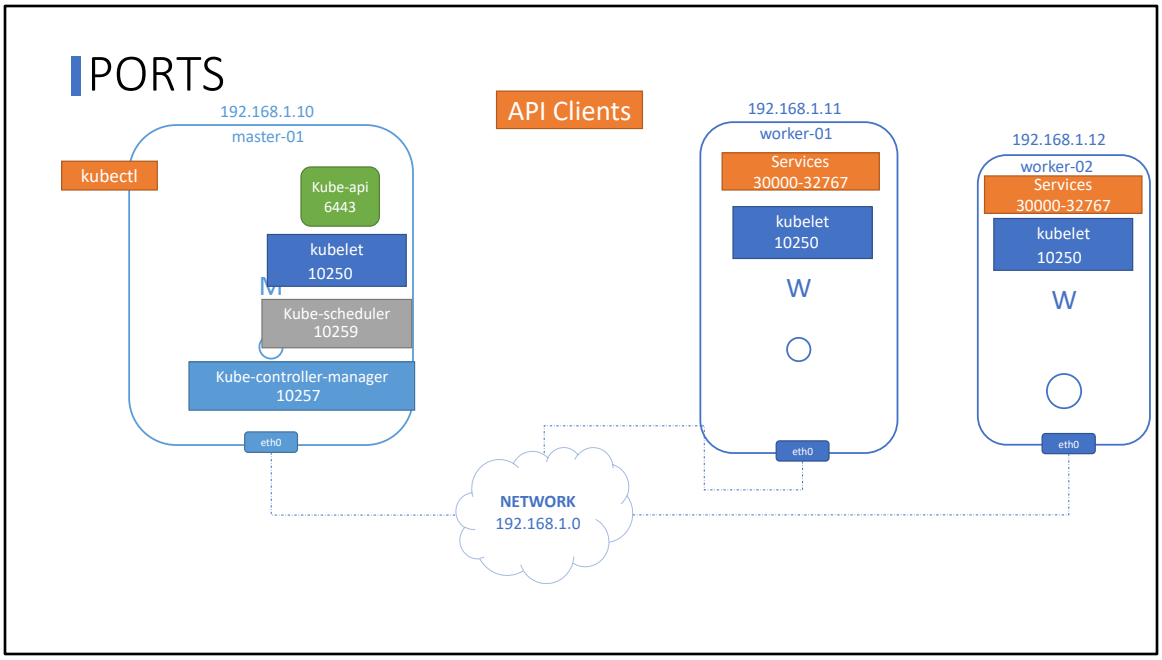
The kubelets on the master and worker nodes listen on 10250. Yes, in case we didn't discuss this, the kubelet's can be present on the master node as well.



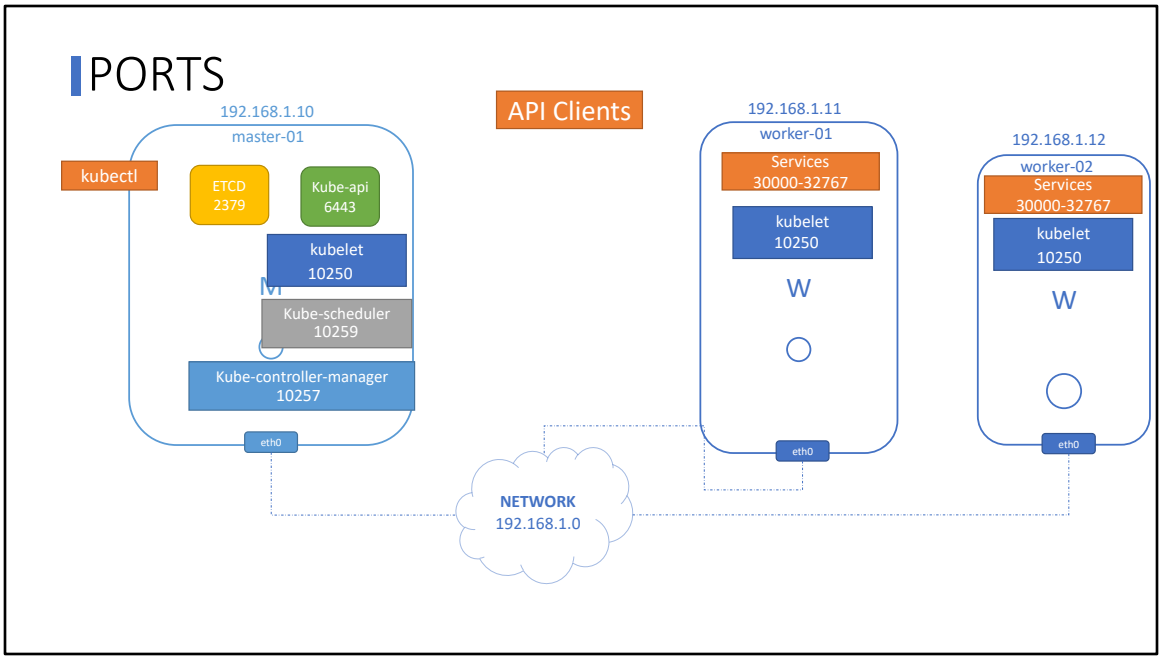
The kube-scheduler requires port 10251 to be open.



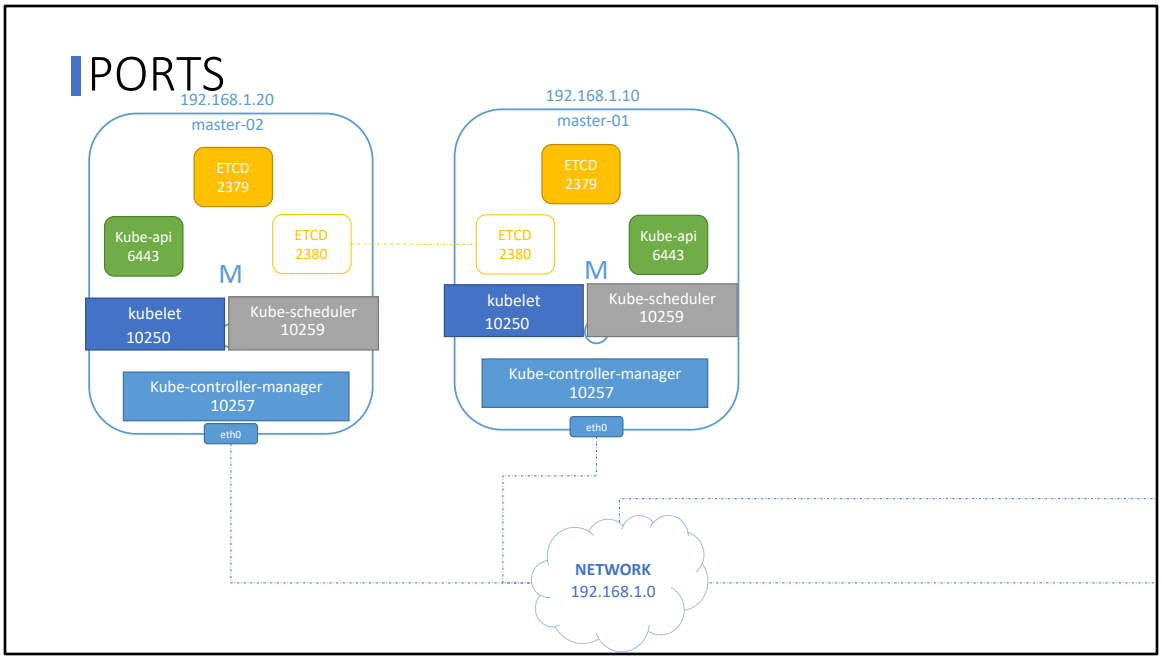
The kube-controller-manager requires port 10252 to be open.



The worker nodes expose services for external access on ports 30000 to 32767. So those should be open as well.



Finally, the ETCD server listens on port 2379.



If you have multiple master nodes, all of these ports need to be open on those as well. And you also need an additional port 2380 open so the ETCD clients can communicate with each other.

Documentation

Check required ports

Master node(s)

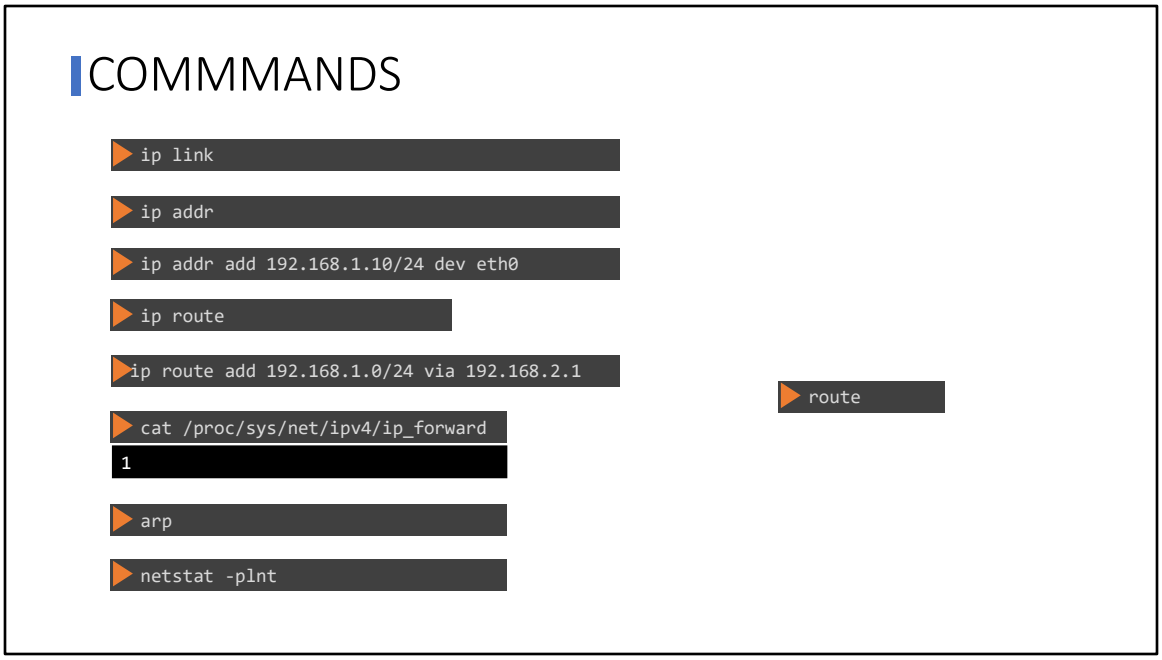
Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	6443*	Kubernetes API server	All
TCP	Inbound	2379-2380	etcd server client API	kube-apiserver, etcd
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	10251	kube-scheduler	Self
TCP	Inbound	10252	kube-controller-manager	Self

Worker node(s)

Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	30000-32767	NodePort Services**	All

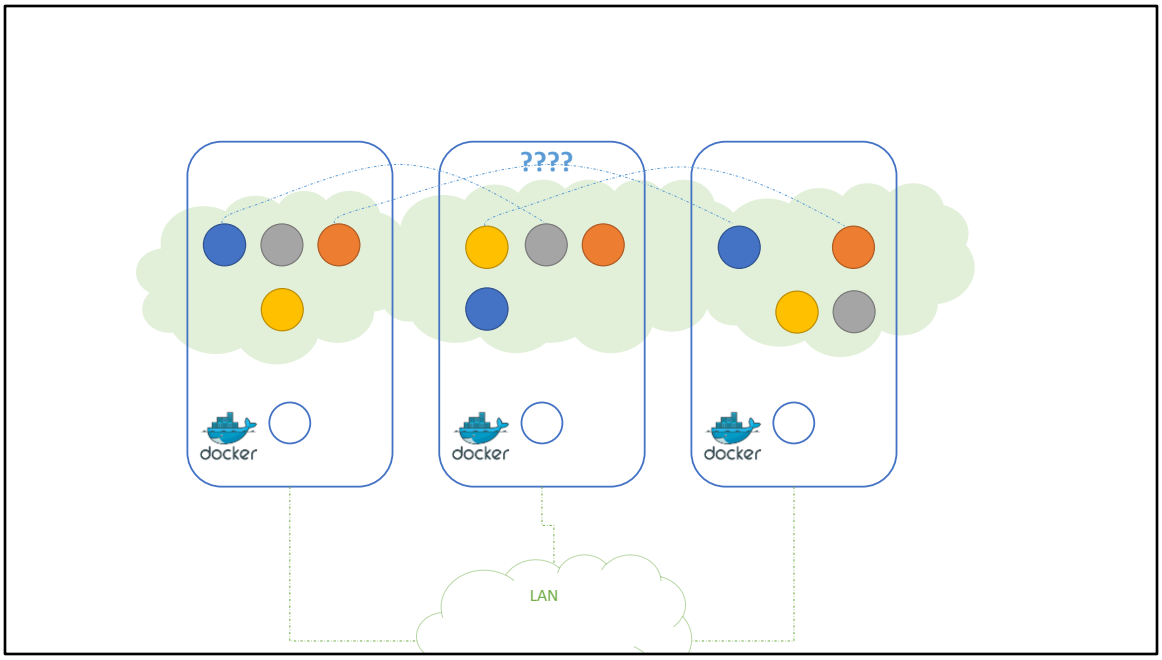
https://kubernetes.io/docs/setup/independent/install-

The list of ports to be opened are also available in the kubernetes documentation. So consider these when you setup networking for your nodes, in your firewalls, or ip table rules or network security group in a cloud environment such as GCP or Azure or AWS. And if things are not working this is one place to look for while you are investigating.



Head over to the practice session and explore the networking setup in the existing environment. Keep these commands handy while you look for information. We will start with simple exercises where you will explore an existing kubernetes cluster and view information about the interfaces, Ips, hostnames, ports etc. These will help you familiarize with the environment and look for information in the future sections. Going forward we will get into more challenging exercises. For now let's start slow.


13
POD Networking
Concepts



So far we have setup several kubernetes master and worker nodes and configured networking between them so they are all on a network that can reach each other. We also made sure the firewall and network security groups are configured correctly to allow for the kubernetes control plane components to reach each other. Assume that we have also setup all the kubernetes control plane components such as the kube-api server, the etcd servers, kubelets etc. And we are finally ready to deploy our applications.

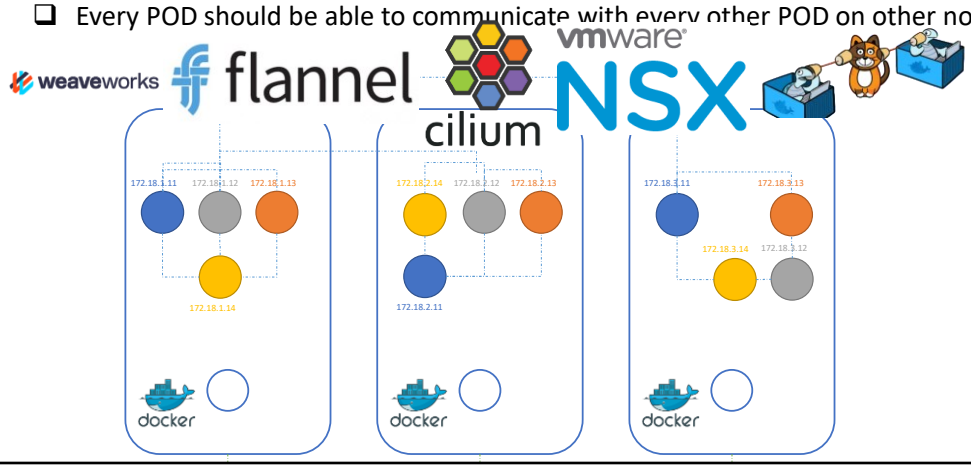
But before we can do that there is something that we must address. We talked about the Network that connects the nodes together. But there is also another layer of networking that is crucial to the cluster's functioning. <c> And that is the networking at the POD layer. Our kubernetes cluster is soon <c> going to have a large number of PODs and services running on it. <c> How are these PODs addressed, how do they communicate with each other, how do you access the services running on these PODs internally from within the cluster, as well as externally from outside the cluster. These are challenges that kubernetes expects you to solve.

As of today, Kubernetes does not come with a built-in solution for this. It expects you to implement a networking solution that solves these challenges. However,

Kubernetes have laid out, clearly, the requirements for POD networking. Let’s take a look at what they are.

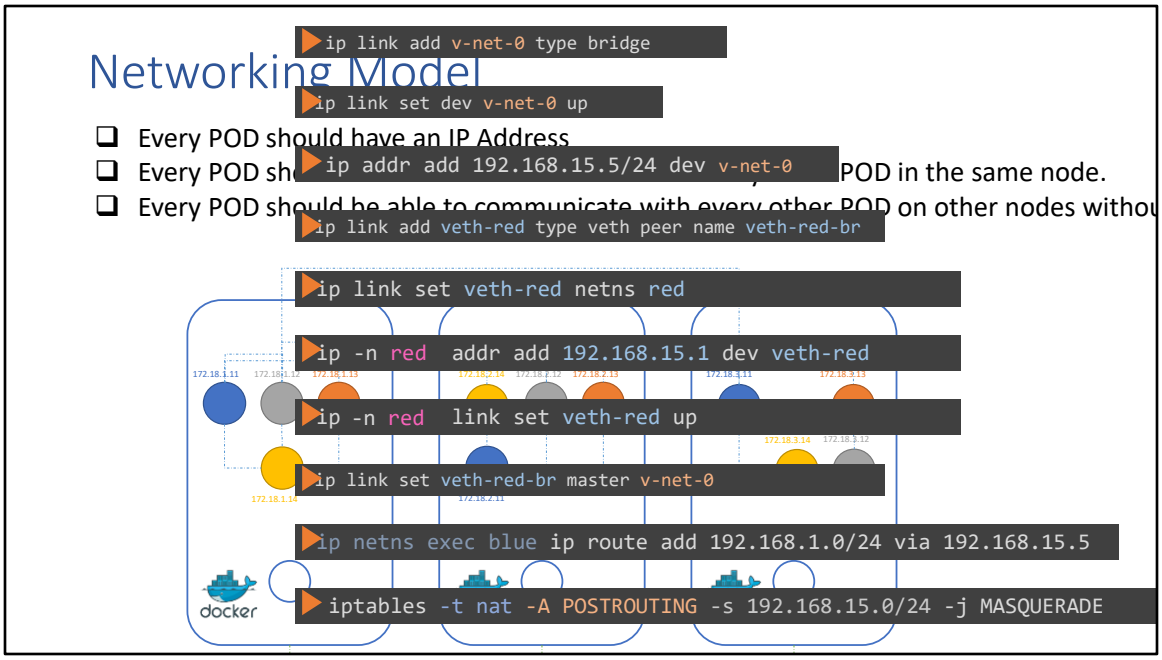
Networking Model

- ❑ Every POD should have an IP Address
- ❑ Every POD should be able to communicate with every other POD in the same node.
- ❑ Every POD should be able to communicate with every other POD on other nodes without

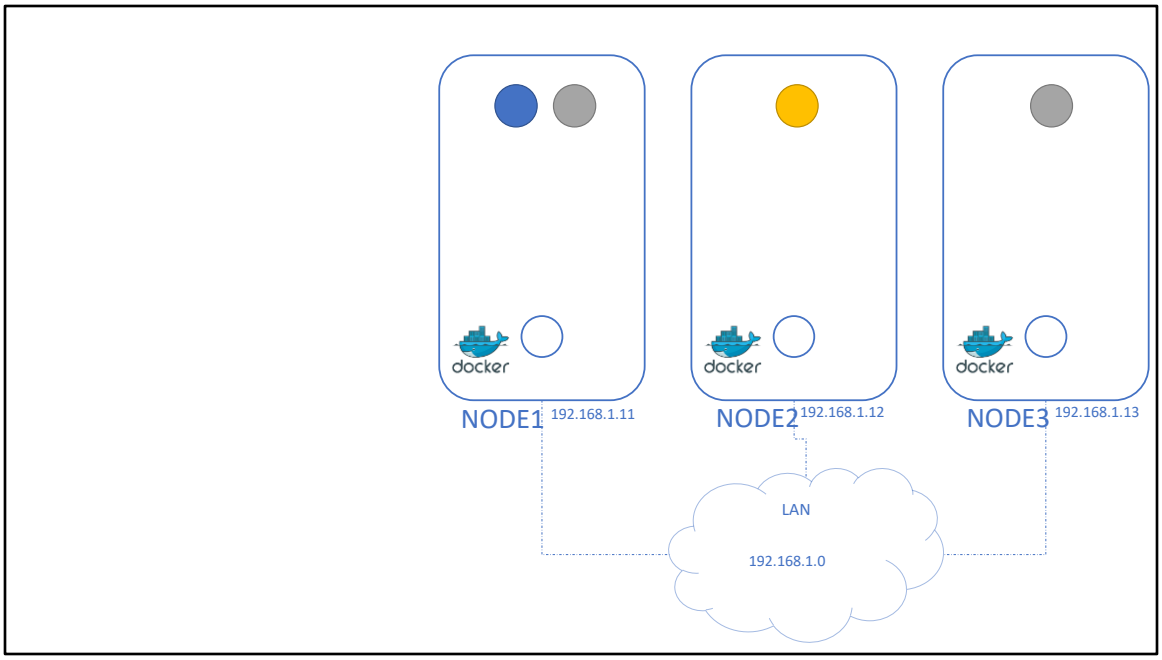


Kubernetes expects every POD to get <c> its own unique IP address. And that every POD should be able to reach every other POD within the same node using that IP address. And every pod should be able to reach every other pod on other nodes as well. Using the same IP address. It doesn't care what IP address that is, and what range or subnet it belongs to. As long as you can implement a solution that takes care of automatically assigning IP addresses and establish connectivity between the PODs in a node as well as PODs on different nodes, you are good. Without having to configure any NAT rules. <elaborate if necessary>

So how do you implement a model that solves these requirements? <c> Now there are many networking solutions available out there that does these.



But we have already learned about networking concepts, routing, IP Address management, namespaces and CNI. <c> So let's try to use that knowledge to solve this by ourselves first. This will help in understanding how other solutions work better. I know there is a bit of repetition. But I am trying to relate the same concept and approach all the way from plain network namespaces on Linux all the way to kubernetes.



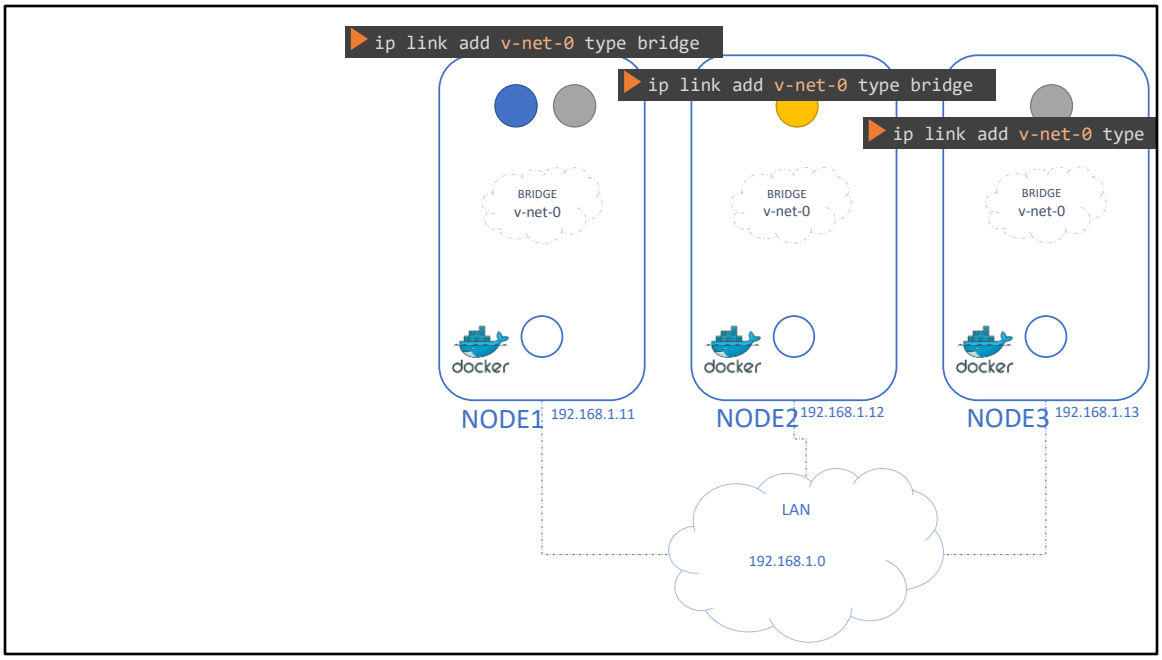
So we have a 3 node cluster. It doesn't matter which one is master or worker. They all run pods either for management or workload purposes. As far as networking is concerned we are going to consider all of them as same.

So first, let's plan what we are going to do.

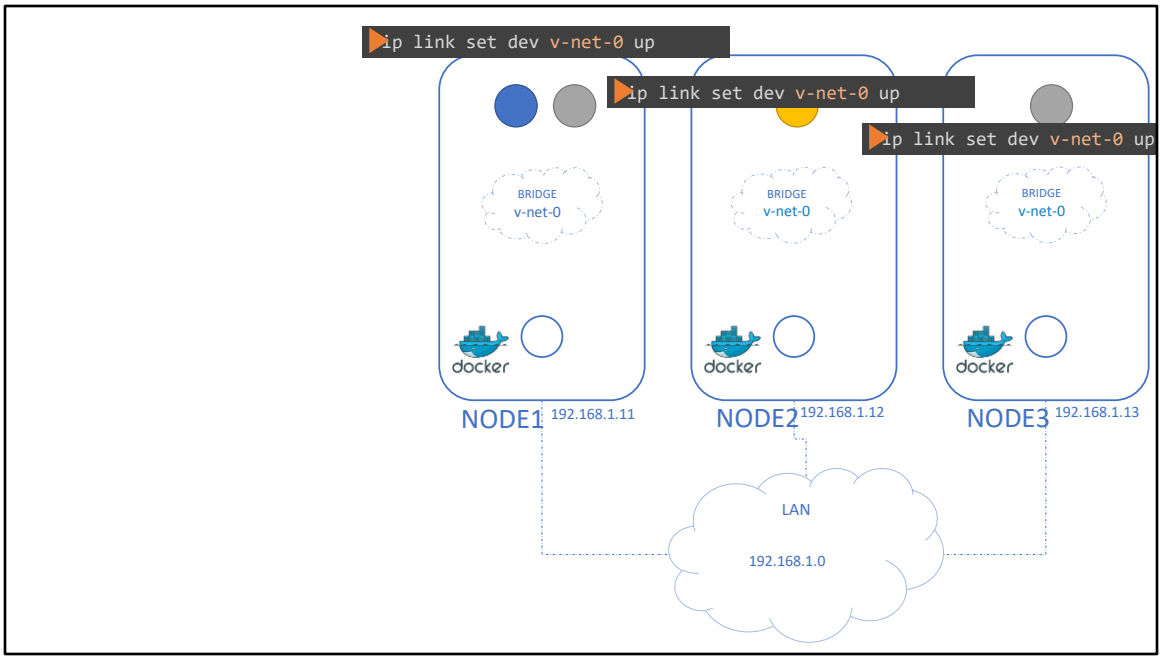
The nodes are part of an external network and has IP addresses in the 192.168.1 series. Node1 is assigned 11, node2 is 12 and node3 is 13.

Next step, When containers are <c> created kubernetes creates network namespaces for them. To enable communication between them, we attach these namespaces to a network. But what network?

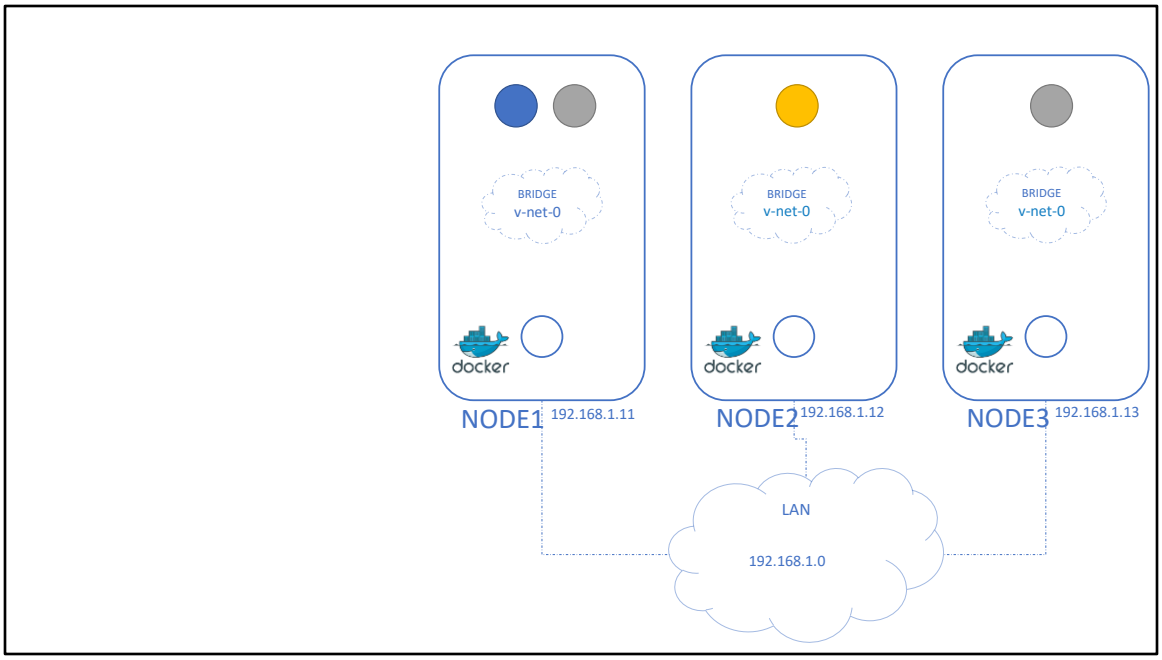
We've learned about bridge networks that can be created within nodes to attach namespaces.



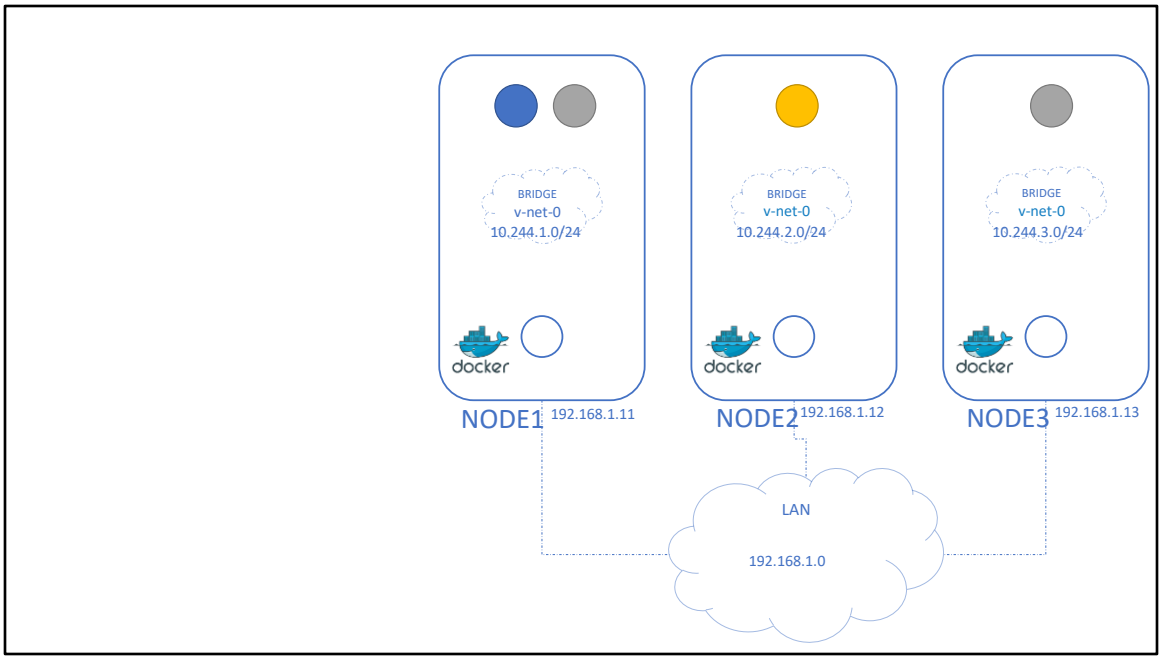
So we create bridge network on each node.



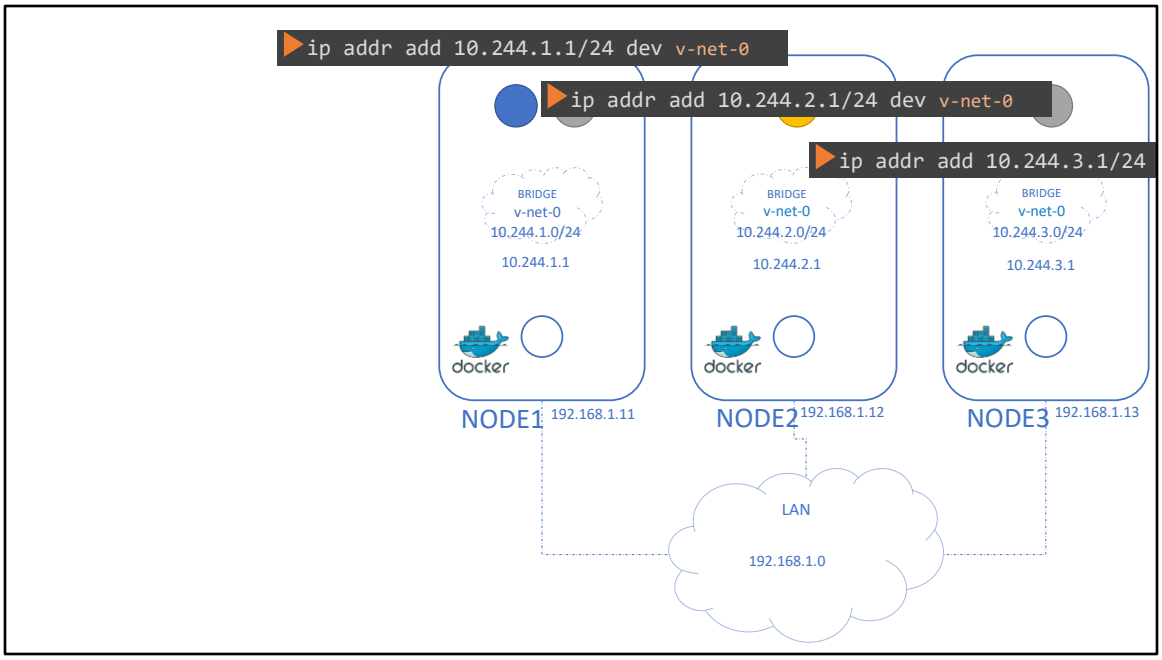
And then bring them up…



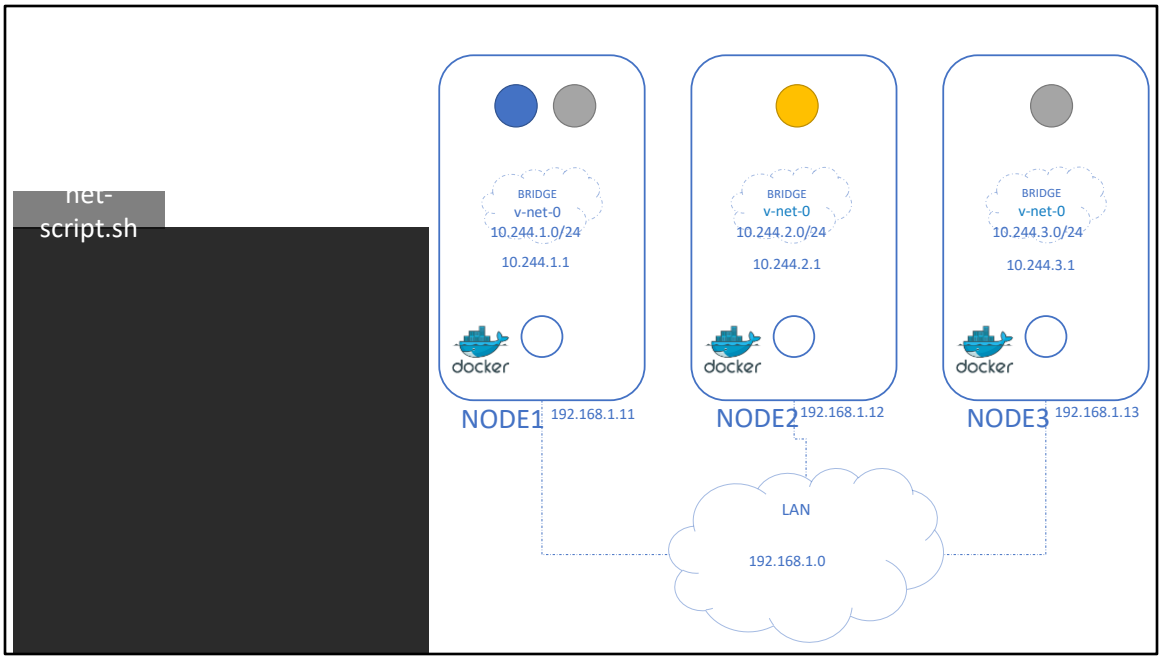
And then bring them up… [pause 2]



It's time to assign an IP address to the bridge interfaces or networks. But what IP address. We decide that each bridge network will be on its own subnet. Chose any private IP address range. Say 10.244.1, 10.244.2 and 10.244.3.



Next we set the IP address for the bridge interface… [pause 1]

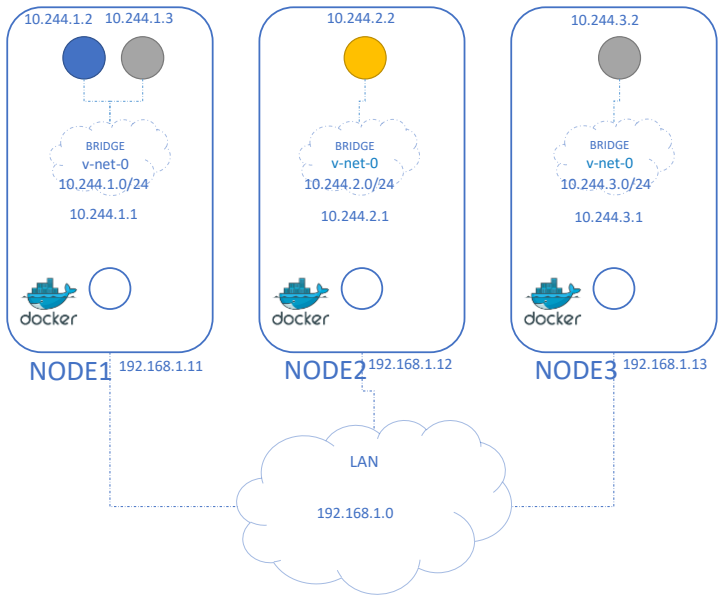


So we have built our base. The remaining steps are to be performed for each container and everytime a new container is created. So we write script for it. Now you don't have to know any kind of complicated scripting. Its just a file that has all commands we will be using. And we can run this multiple times for each container going forward.

To attach a container to the network, we need a pipe or a virtual network cable.

net-script.sh

```
# Create veth pair
ip link add ……

# Attach veth pair
ip link set ……
ip link set ……

# Assign IP Address
ip -n <namespace> addr add ……
ip -n <namespace> route add ……

# Bring Up Interface
ip -n <namespace> link set ……
```



So we have built our base and the configurations on the nodes are complete. The remaining steps are to be performed for each container and should be done every time a new container is created. So we write script for it. Now you don't have to know any kind of complicated scripting. Its just a file that has all commands we will be using to attach a container to a network. And we can run this multiple times for each container going forward.

To attach a container to the network, <c> we need a pipe or a virtual network cable. <c> We create that using the ip link add command. Don't focus on the options, as they are similar to what we saw our previous lectures. Assume that they vary depending on the inputs. We then attach one end to the container and another to the bridge using the ip link set command. <c> We then assign IP address using the ip addr command and add a route to default gateway. But what IP do we add? We either manage that ourselves or store that information in some kind of database. For now we will assume it is 10.244.1.2 which is a free IP in the subnet. We discuss about IPAM in more detail later.

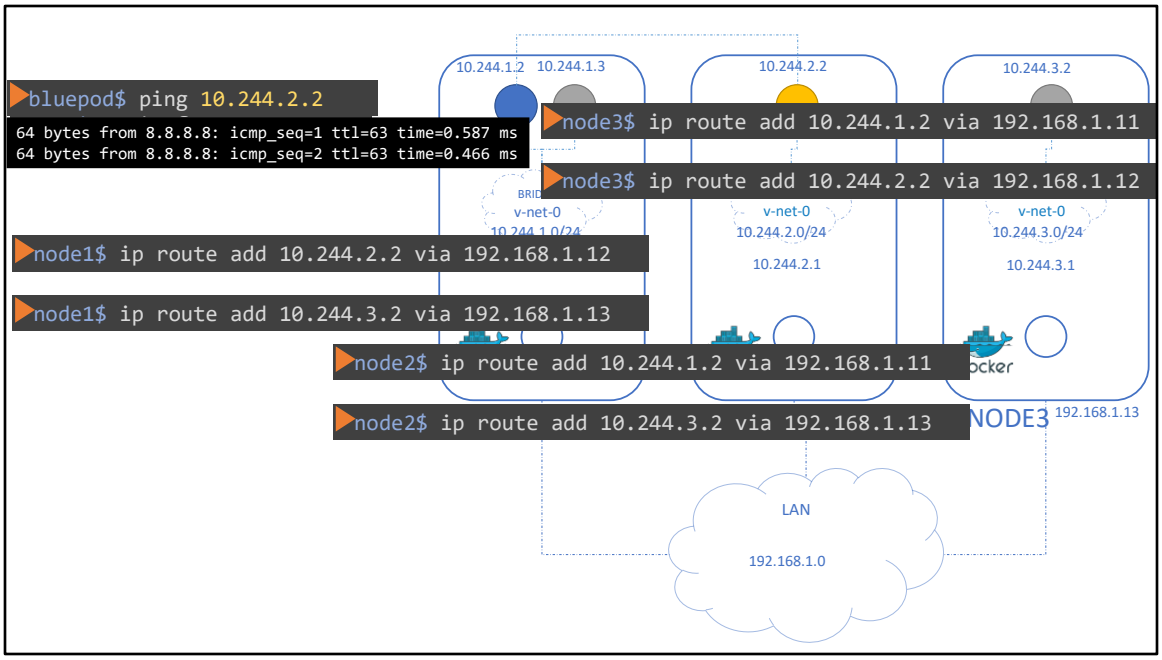
<C> Finally, we bring up the interface.

We then run the same script this time for the second container <c> with its information and gets that container connected to the network. The two containers can now communicate with each other.

<c> We copy the script the other nodes and run the script on them to assign IP address and connect those containers to their own internal networks.

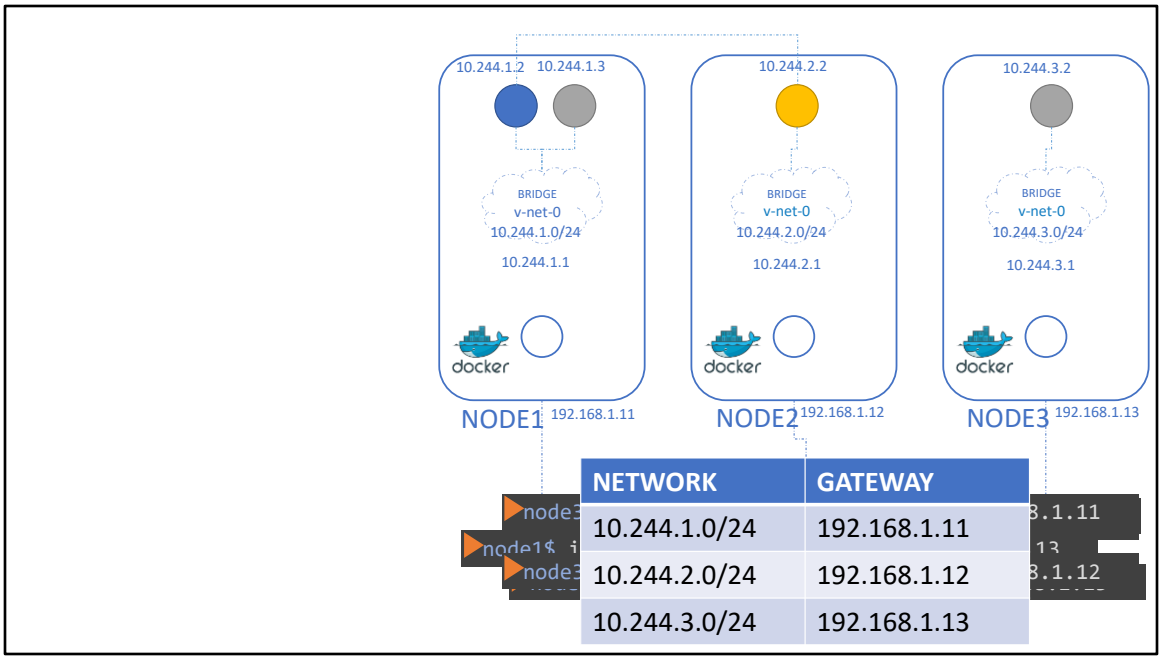
So we have solved the first part of the challenge. The PODs all get their own unique IP address and are abale to communicate with each other on their own nodes.

The next part is to enable them to reach other PODs on other nodes.

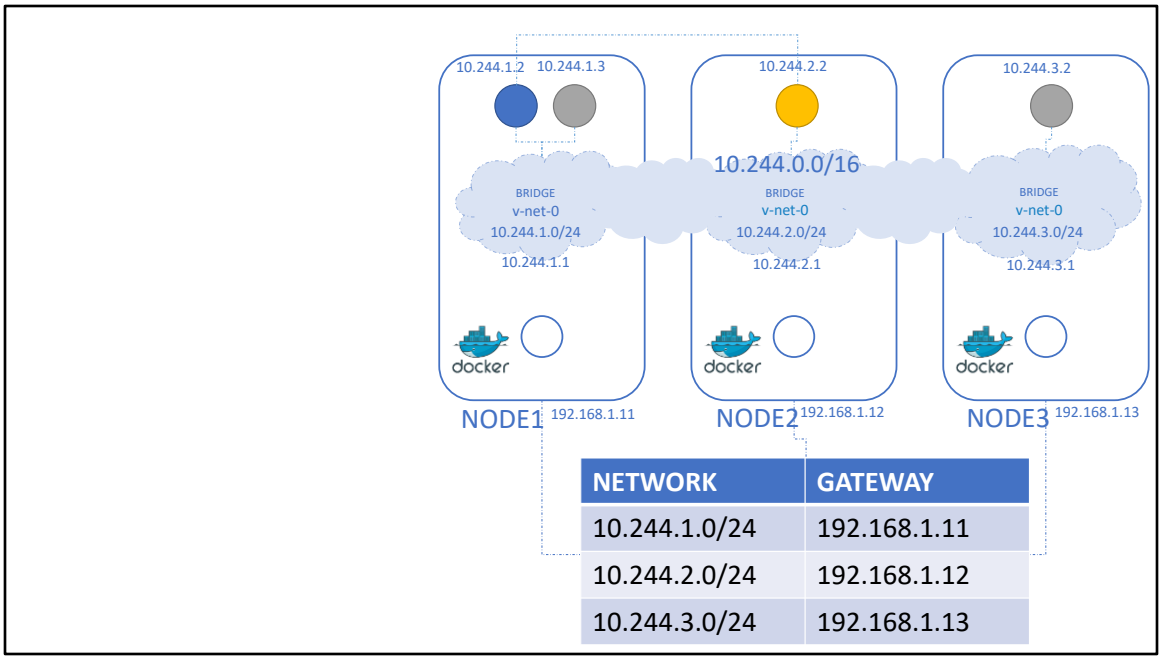


Say for example the pod at 10.244.1.2 on Node1 wants to ping pod 10.244.2.2 on Node2. <c> As of now the first has no idea where the address 10.244.2.2 is, because it is on a different network than its own. So it routes to Node1's IP as it is set to be the default gateway. Node1 doesn't know either since 10.244.2.2 is a private network on Node2. Add a route to Node1s routing table to route traffic to 10.244.2.2 via seconds nodes IP 192.168.1.12. Once the route is added the bluepod is able to ping across.

Similarly we configure route on all hosts to all other hosts with information regarding the respective networks within them. Now this works fine in this simple setup. But this will require a lot more configuration as and when your underlying network architecture gets complicated. With that, the individual virtual networks we created with the address 10.244.1.0/24 …

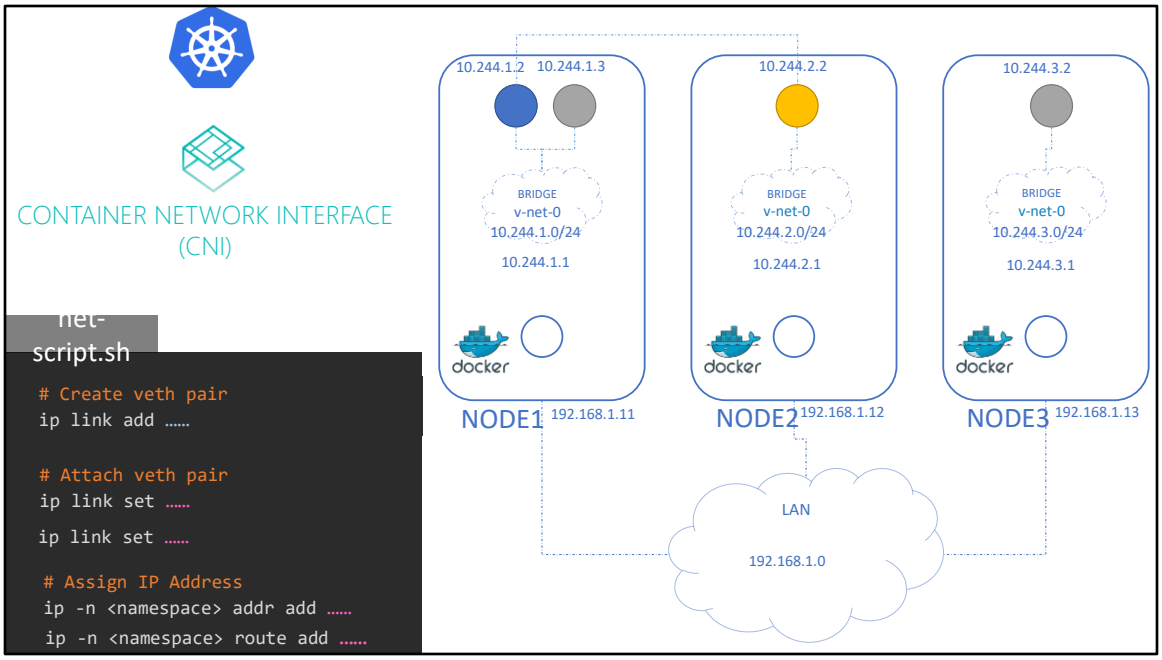


Instead of having to configure routes on each server, a better solution is to do that on a router if you have one in your network. And point all hosts to use that as the default gateway. That way you can easily manage routes to all networks in the routing table on the router.

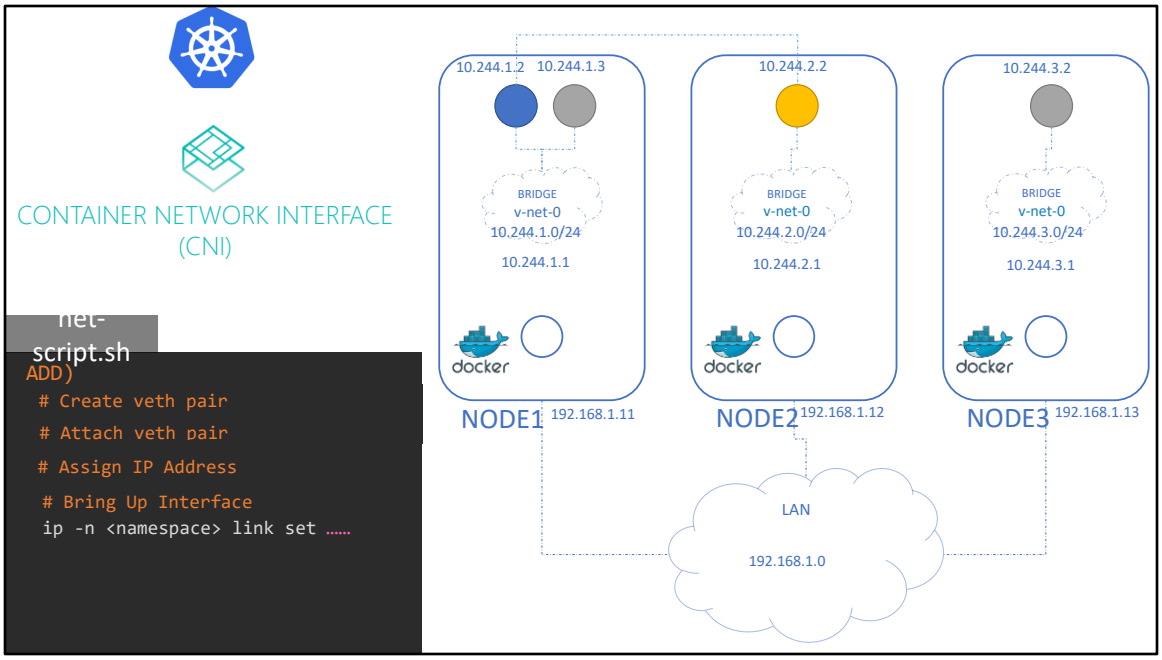


NETWORK	GATEWAY
10.244.1.0/24	192.168.1.11
10.244.2.0/24	192.168.1.12
10.244.3.0/24	192.168.1.13

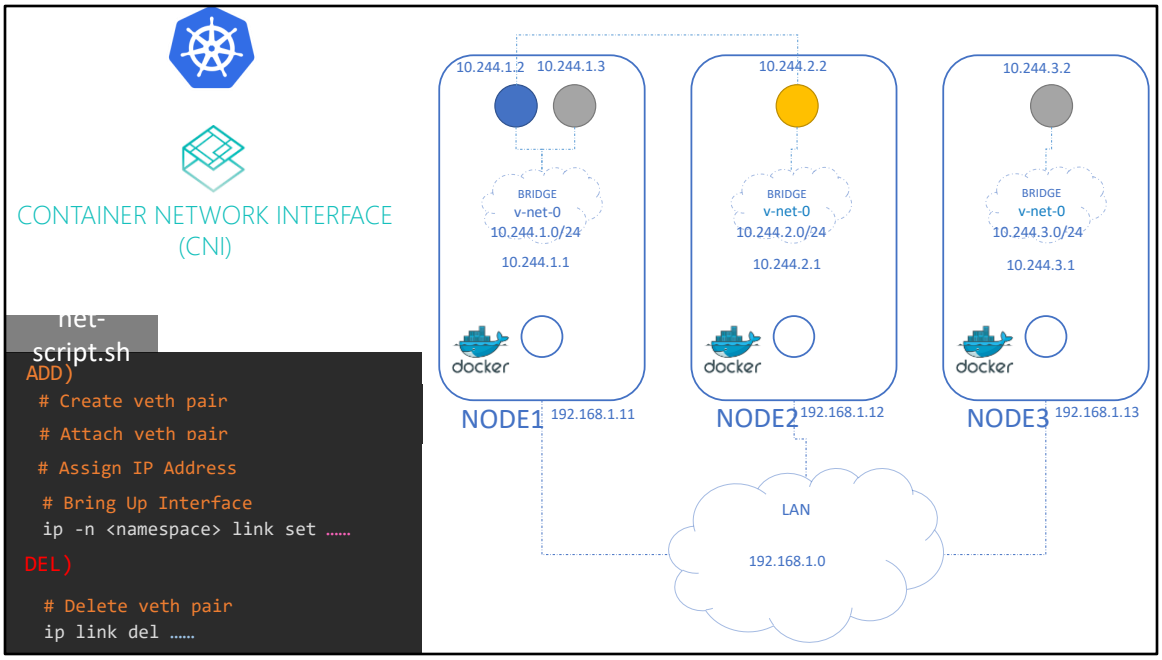
…now form a single large network with the address 10.244.0.0/16.



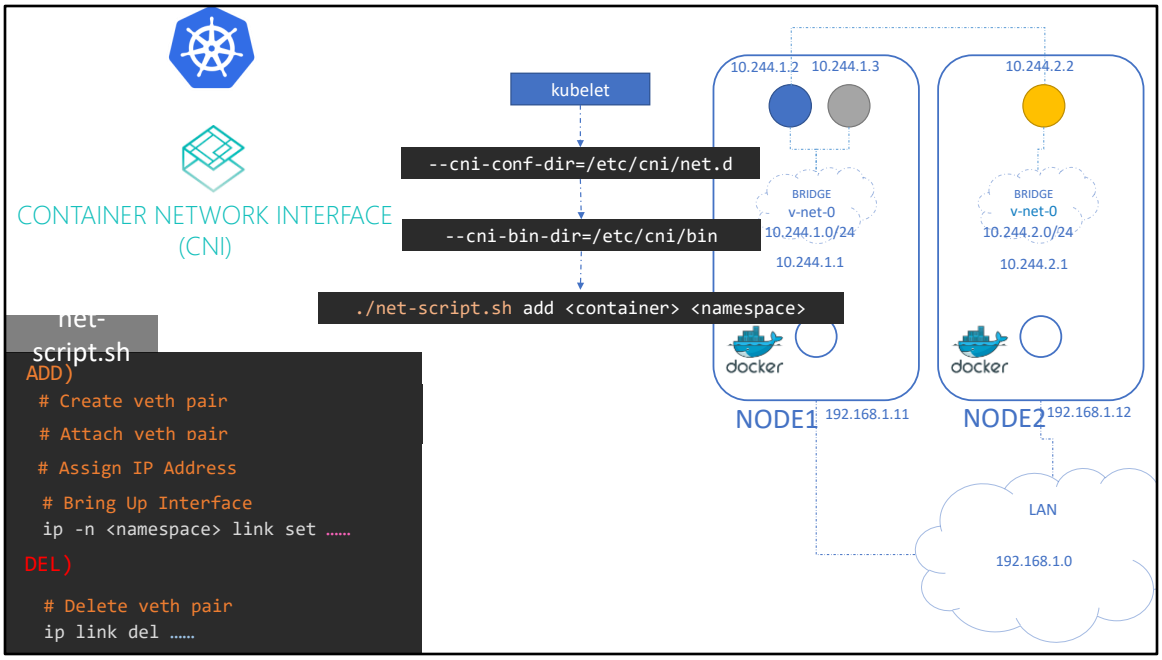
It's time to tie everything together. We performed a number of manual steps to get the environment ready with the bridge networks and routing tables. We then wrote a script that can be run for each container that performs the necessary steps required to connect each container to the network. And we executed the script manually. Ofcourse we don't want to do that, as in large environments 1000s of PODs are created every minute. So how do we run the script automatically when a pod is created on kubernetes? That's where CNI comes in acting as the middlemen. CNI tells kubernetes that this is how you should call a script as soon as you create a container. And CNI tells us this is how your script should be.



So we need to modify the script a little bit to meet CNI's standards. It should have an ADD section that will take care of adding a container to the network..

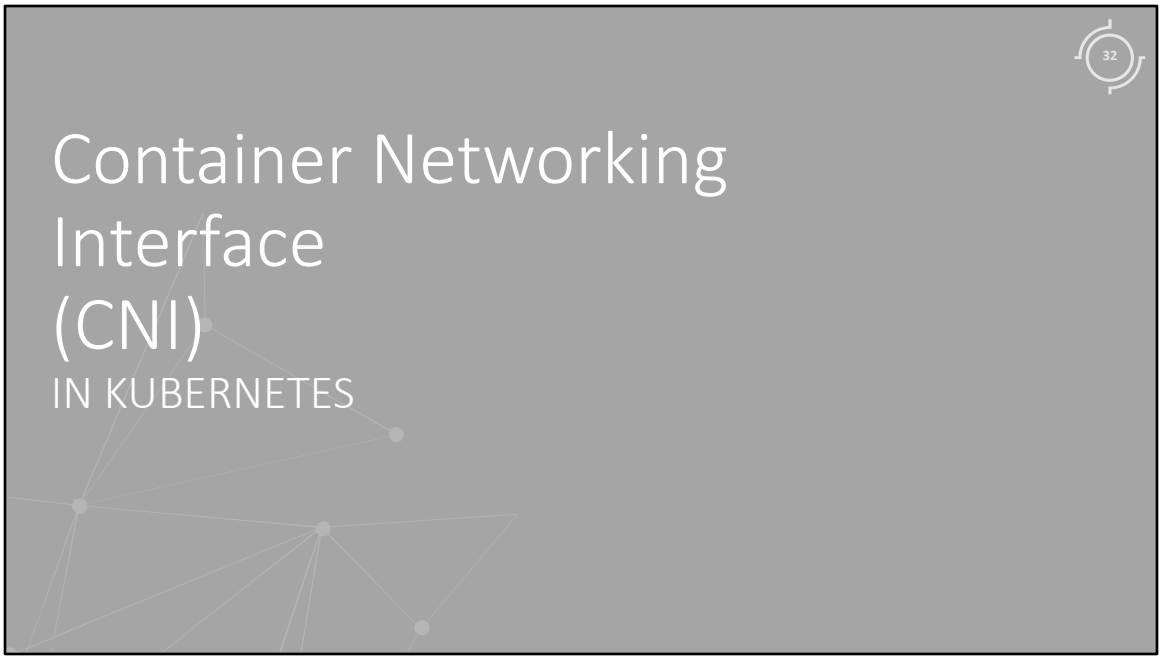


And a delete section that will take care of deleting container interfaces from the network and freeing the IP address etc. So our script is ready.



The kubelet on each node is responsible for creating containers. Whenever a container is created, the <c> kubelet looks at the CNI configuration passed as a command line argument when it was run, and identifies our scripts name. <c> It then looks in the cni bin directory to find our script and then executes the script with the add command and the name and namespace ID of the container. And then our scripts takes care of the rest. We will look at how and where the CNI is configured in kubernetes in the next lecture along with practice tests.

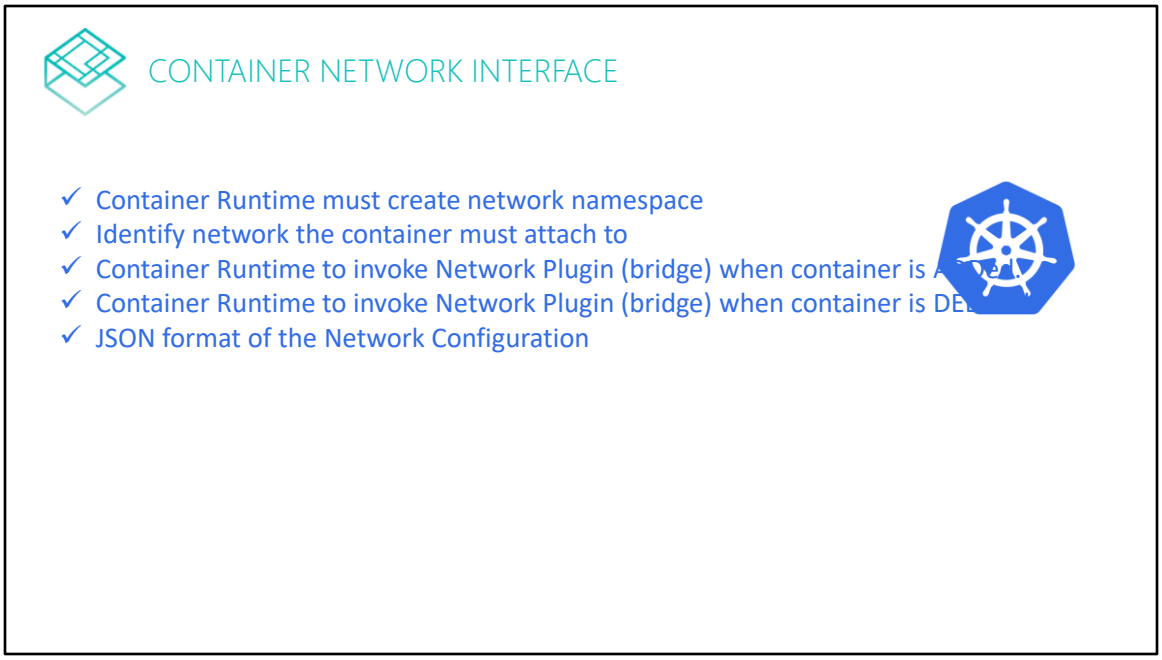
For now that's it from the POD Networking concepts lecture. Hopefully that should give you enough knowledge on inspecting networking within PODs in a kubernetes cluster. We will see how others do the same thing that we did in the upcoming lectures.


32
Container Networking Interface (CNI)
IN KUBERNETES

Pre-Requisites

- ✓ Network Namespaces in Linux
- ✓ Networking in Docker
- ✓ Why and what is Container Network Interface (CNI)?
- ✓ CNI Plugins

In the pre-requisite lectures we started all the way from the absolute basics of Network Namespaces, then we saw how it is done in Docker, we then discussed why you need standards for networking containers and how the Container Network Interface came to be and then we saw a list of supported plugins available with CNI. In this lecture we will see how kubernetes is configured to use these network plugins.



As we discussed in the pre-requisite lecture CNI defines the responsibilities of container runtime. As per CNI, container runtimes, in our case Kubernetes, is responsible for creating container network namespaces, identifying and attaching those namespaces to the right network by calling the right network plugin. So where do we specify the CNI plugins for Kubernetes to use?

Configuring CNI

kubelet.service

```
ExecStart=/usr/local/bin/kubelet \\
  --config=/var/lib/kubelet/kubelet-config.yaml \\
  --container-runtime=remote \\
  --container-runtime-endpoint=unix:///var/run/containerd/containerd.sock \\
  --image-pull-progress-deadline=2m \\
  --kubeconfig=/var/lib/kubelet/kubeconfig \\
  --network-plugin=cni \\
  --cni-bin-dir=/opt/cni/bin \\
  --cni-conf-dir=/etc/cni/net.d \\
  --register-node=true \\
  --v=2
```

The CNI plugin must be invoked by the component within Kubernetes that is responsible for creating containers. Because component must then invoke the appropriate network plugin after the container is created. The CNI plugin is configured in the kubelet service on each node in the cluster. If you look at the kubelet service file, you will see an option called network-plugin set to CNI.

View kubelet options

```
ps -aux | grep kubelet
root      2095  1.8  2.4 960676 98788 ?        Ssl  02:32   0:36 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf --config=/var/lib/kubelet/config.yaml --cgroup-driver=cgroupfs --cni-bin-dir=/opt/cni/bin --cni-conf-dir=/etc/cni/net.d --network-plugin=cni
```

```
ls /opt/cni/bin
bridge  dhcp  flannel  host-local  ipvlan  loopback  macvlan  portmap  ptp  sample  tuning  vlan  weave-ipam  weave-net  weave-plugin-2.2.1
```

```
ls /etc/cni/net.d
10-bridge.conf
```

You can see the same information on viewing the running kubelet service. You can see the network-plugin set to CNI and a few other options related to CNI, such as the CNI bin directory and CNI Config directory. The CNI bin directory has all the supported CNI plugins as executables. Such as the bridge, dhcp, flannel etc.

The CNI config directory has a set of configuration files. This is where kubelet looks to find out which plugin needs to be used. In this case it finds the bridge configuration file. If there are multiple files here, it will chose the one in alphabetical order.

View kubelet options

```
ls /etc/cni/net.d
10-bridge.conf
```

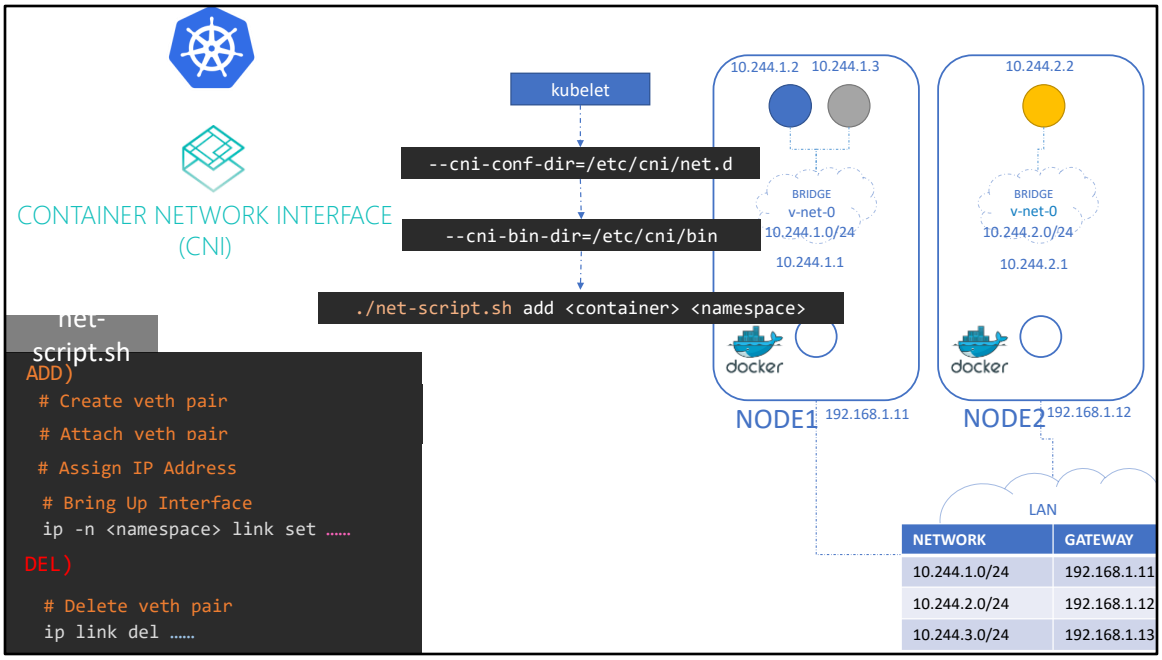
```
cat /etc/cni/net.d/10-bridge.conf
```

```
{
    "cniVersion": "0.2.0",
    "name": "mynet",
    "type": "bridge",
    "bridge": "cni0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.22.0.0/16",
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
    }
}
```

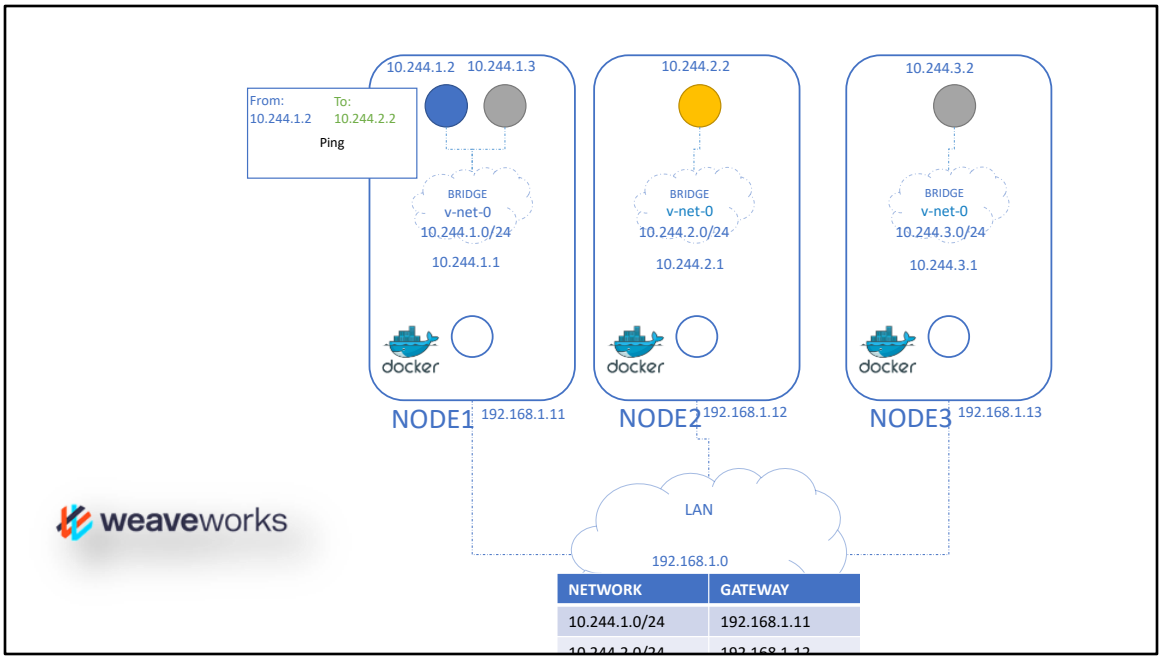
If you look at the bridge conf file, it looks like this. This is a format defined by the CNI standard for a plugin configuration file. It's name is mynet, type is bridge. It also has a set of other configurations which we can relate to the concepts we discussed in the pre-requisite lectures on bridging, routing and Masquerading in NAT. The isGateway defines whether the bridge interface should get an IP address assigned so it can act as a gateway. The ipMasquerade defines if a NAT rule should be added for IP masquerading. The IPAM section defines IPAM configuration. This is where you specify the subnet or range of IP addresses that will be assigned to PODs and any necessary routes. The type host-local indicates that the IP addresses are managed locally on this host. Unlike a DHCP server maintaining it remotely. The type can also be set to DHCP to configure an external DHCP server.

Well that's it for this lecture. Head over to the practice exercises and practice working with CNI in Kubernetes.

WeaveWorks weave CNI plugin. In the previous practice test, we saw how it is configured. Now, we will see more details about how it works.

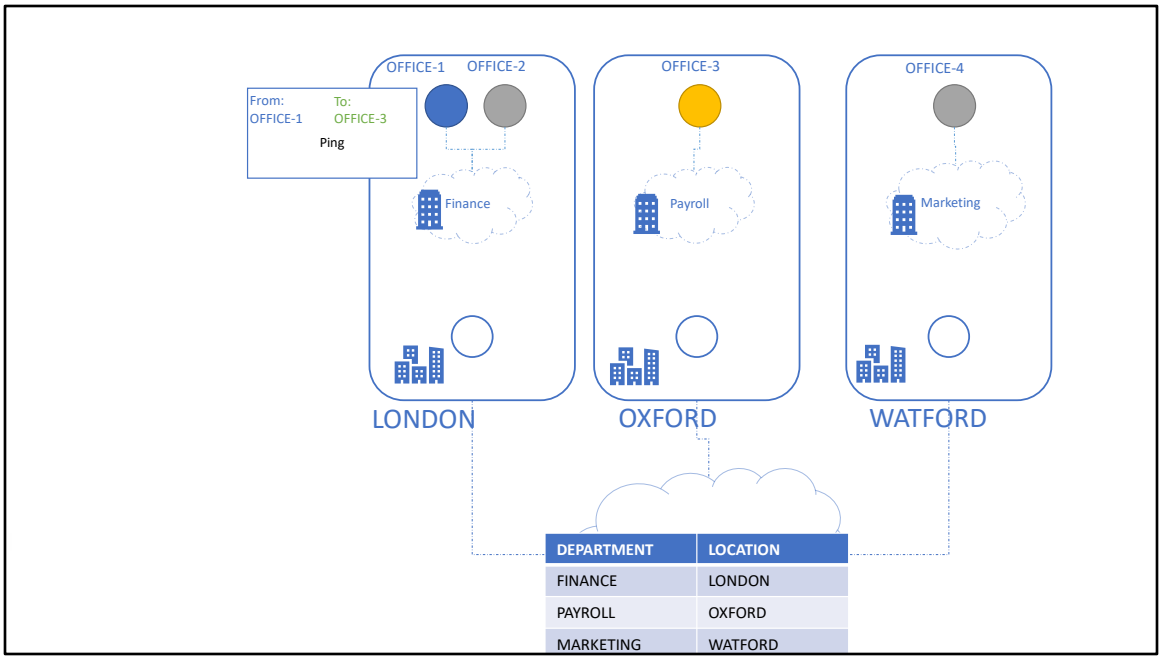


We will start where we left off in the POD Networking Concepts section. We had our own custom CNI script that we built and integrated into kubelet through CNI.



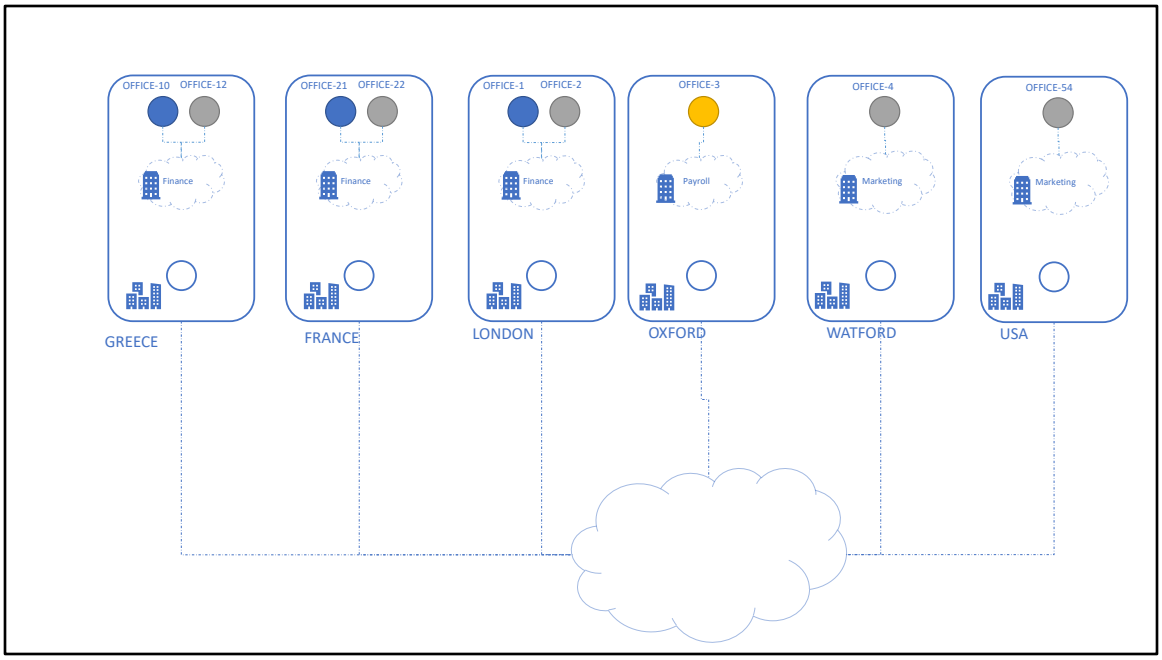
In the previous lecture we saw how, instead of our custom script, we can integrate the weave plugin. Let us now see how the weave solution works, as it is important to understand at least one solution well. You should then be able to relate this to other solutions as well.

So the networking solution we setup manually had a routing table which mapped what networks are on what hosts. <c> So when a packet is sent from one pod to the other, it goes out to the network, to the router and finds its way to the node that hosts that pod. Now that works for a small environment and in a simple network. But in larger environments with 100s of nodes in a cluster and 100s of PODs on each node, this is not practical. The routing table may not support so many entries. That is where you need to get creative and look for other solutions.

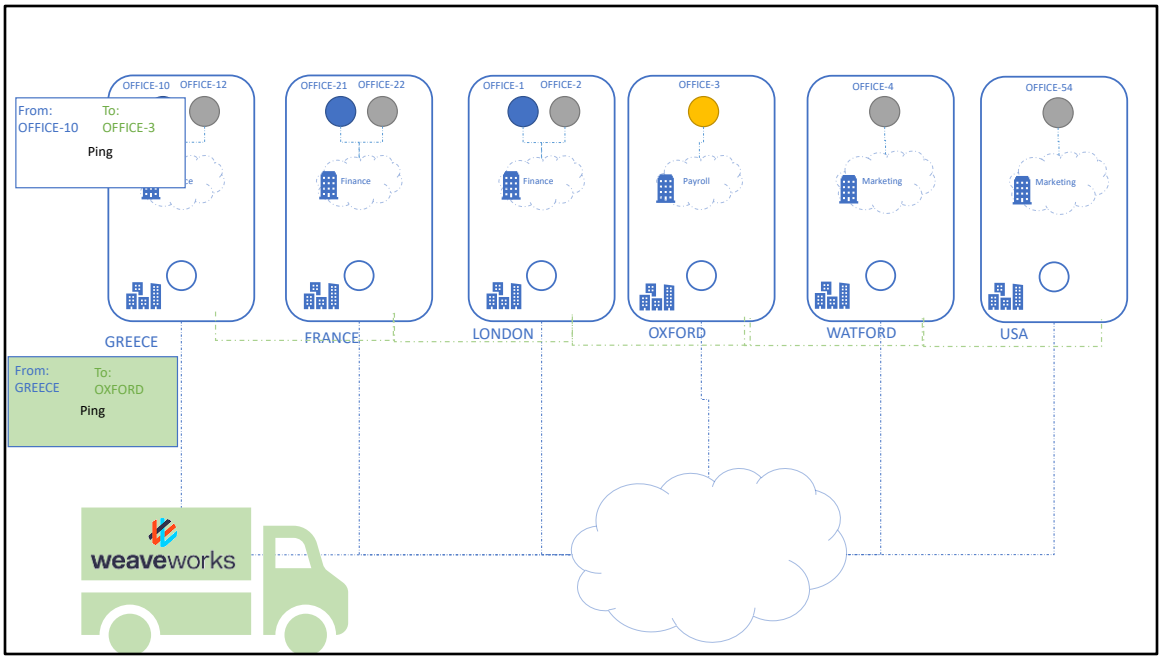


Think of the kubernetes cluster as our company. And the nodes as different office sites. With each site, we have different departments and within each department we have offices. Someone is office-1 wants to send a packet to office-3 and hands it over to the office boy. All he knows is it needs to go to office 3 and he doesn't care who or how it is transported.

The office boy takes the package, gets in his car, looks up address of the target office in GPS, uses directions on the street and finds his way to the destination site. Delivers the package to Payroll department who in turn forwards the package to office3. This works just fine for now.

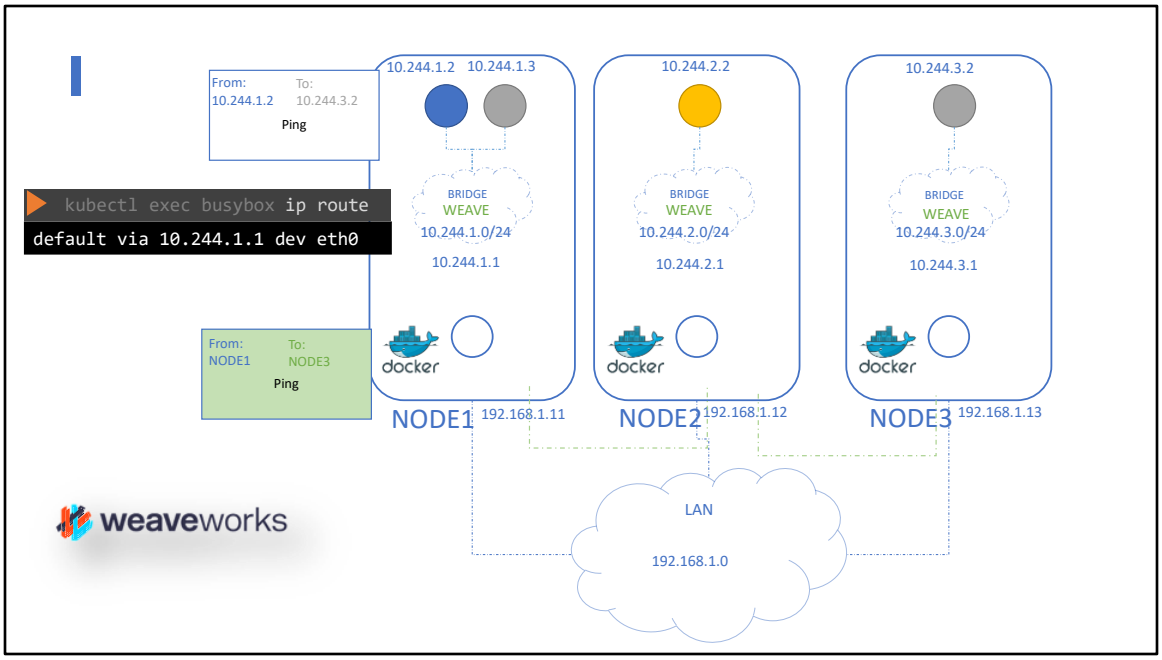


We soon expand to different regions and countries and this process no longer works. It's hard for the office boy to keep track of routes to these large number of offices across different countries and of course he can't drive to these offices by himself. That's where we decide to outsource all mailing and shipping activities to a company who does it best.



Once the shipping company is engaged, the first thing that they do is place their agents in each of our companies sites. <c> These agents are responsible for managing all shipping activities between sites. <c> They also keep talking to each other and are well connected. So they all know about each others sites, the departments in them and the offices in them.

And so, when a package is sent from say, <c> office-10 to office-3, the shipping agent in that site intercepts the package, and looks at the target office name, and he knows exactly in which site and department that office is in through his little internal network with his peers on the other sites. <c> He then places this package into his own new package with the destination address set the target sites location and <c> then sends the package through. Once the package arrives at the destination, it is again intercepted by the agent on that site, <c> opens the packet retrieves the original packet and delivers it to the right department. [pause 2]



Back to our world, when the weave CNI plugin is deployed on a cluster, it deploys an agent or service on each node. They communicate with each other to exchange information regarding the nodes and networks and PODs within them. Each agent or peer stores a topology of the entire setup, that way they know the pods and their IPs on the other nodes.

<c> Weave creates its own bridge on the nodes and names it weave. Then assigns IP address to each network. The Ips shown here are examples only. In the upcoming practice test you will figure out the exact range of IP addresses weave assigns on each node. We will talk about IPAM and how IP addresses are handed out to PODs and containers in the next lecture.

<c> Remember that a single POD may be attached to multiple bridge networks. For example you could have a POD attach to the weave bridge as well as docker bridge created by docker. What path a packet takes to reach destination depends on the route configured on the container. <c> Weave makes sure that PODs get the correct route configured to reach the agent. And the agent then takes care of other PODs.

<c> now when a packet is sent from one pod to another on another node, <c> weave

intercepts the packet and identifies that its on a separate network. It then encapsulates this packet into a new one with new source and destination and <c> sends it across the network. Once on the other side, the other weave agent retrieves the packet, <c> decapsulates and routes the packet to the right POD.

Deploy Weave

```
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"
serviceaccount/weave-net created
clusterrole.rbac.authorization.k8s.io/weave-net created
clusterrolebinding.rbac.authorization.k8s.io/weave-net created
role.rbac.authorization.k8s.io/weave-net created
rolebinding.rbac.authorization.k8s.io/weave-net created
daemonset.extensions/weave-net created
```

So how do we deploy weave on a kubernetes cluster? Weave and weave peers can be deployed as services or daemons on each node in the cluster manually or if kubernetes is setup already then an easier way to do that is to deploy it as PODs in the cluster. <

Once the base kubernetes system is ready with nodes and networking configured correctly between the nodes and the basic control plan components are deployed, weave can be deployed in the cluster with a single <c> kubectl apply command. This deploys all the necessary components required for weave in the cluster. Most importantly the weave peers are deployed as a <c> daemonset. A daemonset ensures that one pod of the given kind is deployed on all nodes in the cluster. This works perfectly for the weave peers.

Weave Peers

```
kubectl get pods –n kube-system
NAME                              READY     STATUS    RESTARTS   AGE       IP             NODE
NOMINATED NODE
coredns-78fcdf6894-99khw          1/1       Running   0          19m       10.44.0.2      master
<none>
coredns-78fcdf6894-p7dpj          1/1       Running   0          19m       10.44.0.1      master
<none>
etcd-master                       1/1       Running   0          18m       172.17.0.11    master
<none>
kube-apiserver-master             1/1       Running   0          18m       172.17.0.11    master
<none>
kube-scheduler-master             1/1       Running   0          17m       172.17.0.11    master
<none>
```

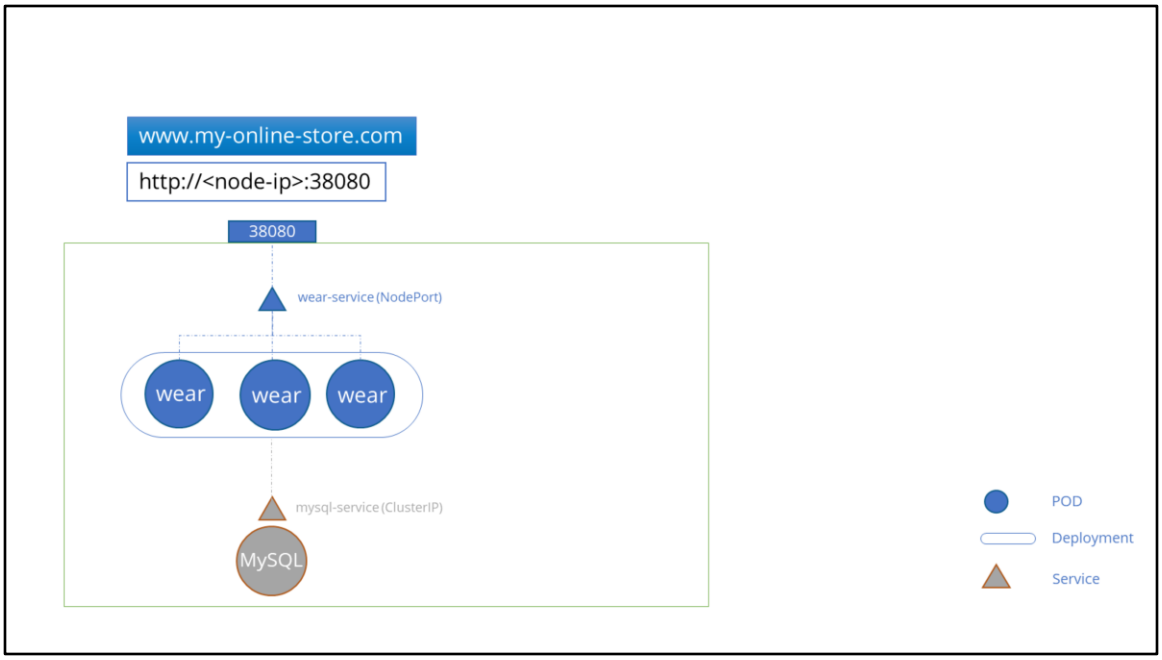
```
kubectl logs weave-net-5gcmb weave –n kube-system
INFO: 2019/03/03 03:41:08.643858 Command line options: map[status-addr:0.0.0.0:6782 http-addr:127.0.0.1:6784 ipalloc-range:10.32.0.0/12 name:9e:96:c8:09:bf:c4 nickname:node02 conn-limit:30
datapath:datapath db-prefix:/weavedb/weave-net host-root:/host port:6783 docker-api: expect-npc:true ipalloc-init:consensus=4 no-dns:true]
INFO: 2019/03/03 03:41:08.643980 weave  2.2.1
INFO: 2019/03/03 03:41:08.751508 Bridge type is bridged_fastdp
INFO: 2019/03/03 03:41:08.751526 Communication between peers is unencrypted.
INFO: 2019/03/03 03:41:08.753583 Our name is 9e:96:c8:09:bf:c4(node02)
INFO: 2019/03/03 03:41:08.753615 Launch detected - using supplied peer list: [172.17.0.11 172.17.0.23 172.17.0.30 172.17.0.52]
INFO: 2019/03/03 03:41:08.753632 Checking for pre-existing addresses on weave bridge
INFO: 2019/03/03 03:41:08.756183 [allocator 9e:96:c8:09:bf:c4] No valid persisted data
INFO: 2019/03/03 03:41:08.761033 [allocator 9e:96:c8:09:bf:c4] Initialising via deferred consensus
INFO: 2019/03/03 03:41:08.761091 Sniffing traffic on datapath (via ODP)
INFO: 2019/03/03 03:41:08.761659 ->[172.17.0.23:6783] attempting connection
INFO: 2019/03/03 03:41:08.817477 overlay_switch ->[8a:31:f6:b1:38:3f(node03)] using fastdp
INFO: 2019/03/03 03:41:08.819493 sleeve ->[172.17.0.52:6783|8a:31:f6:b1:38:3f(node03)]: Effective MTU verified at 1438
INFO: 2019/03/03 03:41:09.107287 Weave version 2.5.1 is available; please update at https://github.com/weaveworks/weave/releases/download/v2.5.1/weave
INFO: 2019/03/03 03:41:09.284907 Discovered remote MAC 8a:dd:b5:14:8f:a3 at 8a:dd:b5:14:8f:a3(node01)
INFO: 2019/03/03 03:41:09.331952 Discovered remote MAC 8a:31:f6:b1:38:3f at 8a:31:f6:b1:38:3f(node03)
INFO: 2019/03/03 03:41:09.355976 Discovered remote MAC 8a:a5:9c:d2:86:1f at 8a:31:f6:b1:38:3f(node03)
```

<c> If you deployed your cluster with the kubeadm tool and weave plugin, you can see the weave peers as pods deployed on each node.

<c> For troubleshooting purpose, view the logs using the kubectl logs command.

So let’s get started

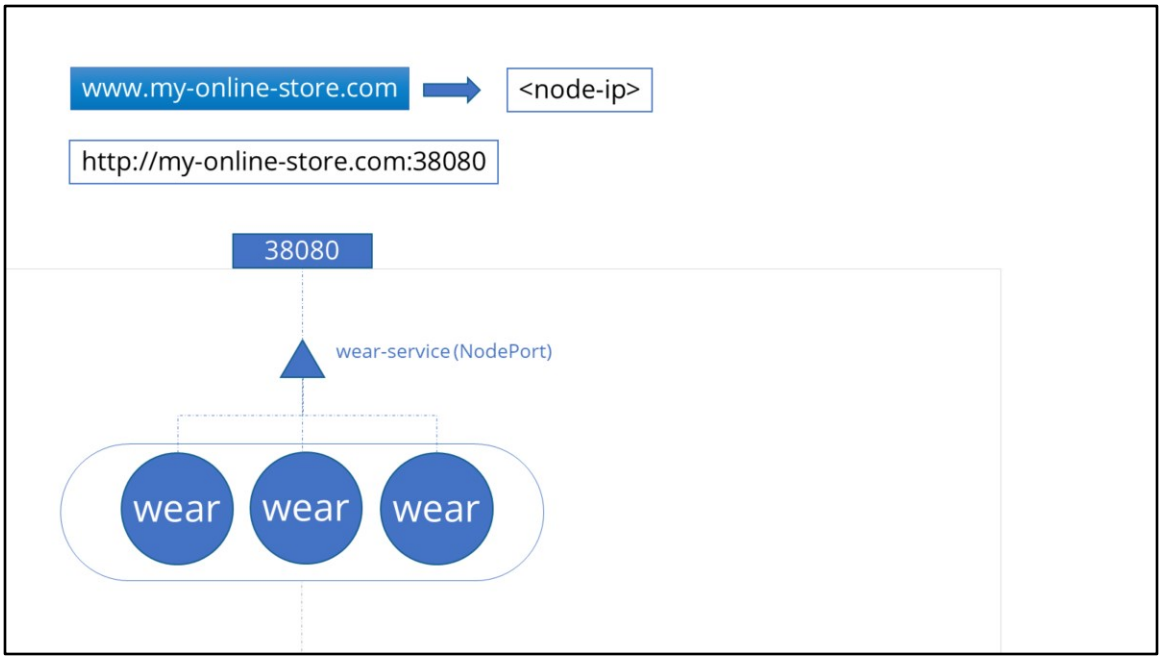
One of the common questions that students reach out about usually is regarding services and ingress. What's the difference between the two and when to use what. So we are going to briefly revisit services and work our way towards ingress. We will start with a simple scenario. You are deploying an application on Kubernetes for a company that has an online store selling products. Your application would be available at say my-online-store.com.



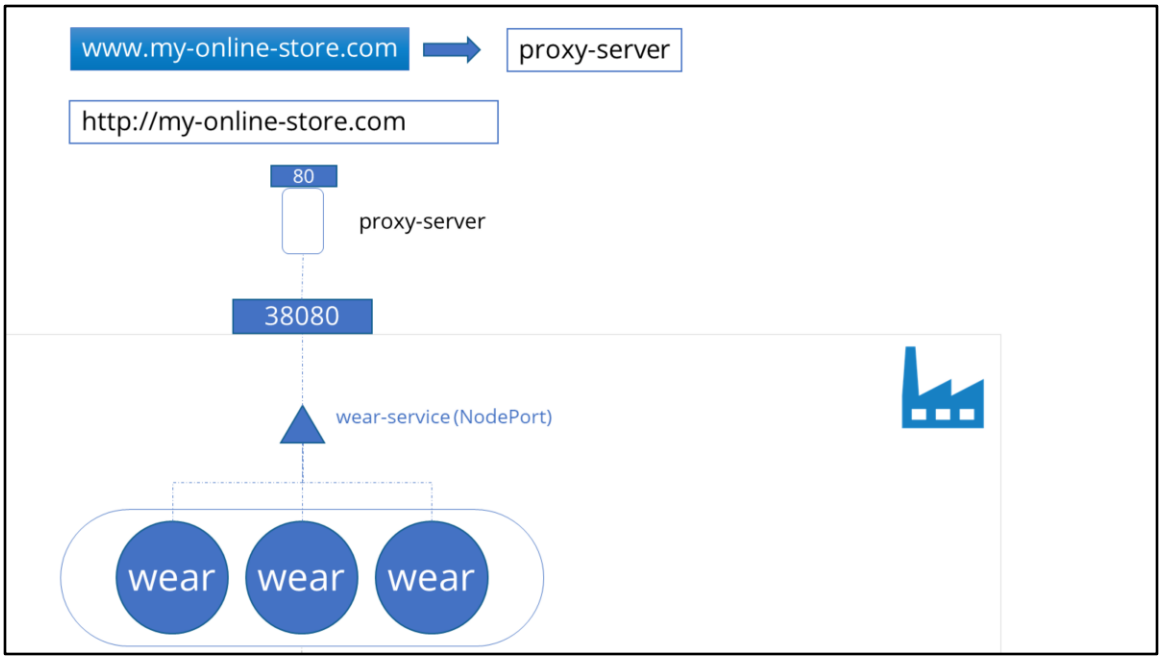
You build the application into a Docker Image and deploy it on the kubernetes cluster as a POD in a Deployment. Your application needs a database so you deploy a MySQL database as a POD and create a service of type ClusterIP called mysql-service to make it accessible to your application. Your application is now working. To make the application accessible to the outside world, you create another service, this time of type NodePort and make your application available on a high-port on the nodes in the cluster. In this example a port 38080 is allocated for the service. The users can now access your application using the URL: http, colon, slash slash IP of any of your nodes followed by port 38080. That setup works and users are able to access the application.

Whenever traffic increases, we increase the number of replicas of the POD to handle the additional traffic, and the service takes care of splitting traffic between the PODs.

<pause>
However, if you have deployed a production grade application before, you know that there are many more things involved in addition to simply splitting the traffic between the PODs.

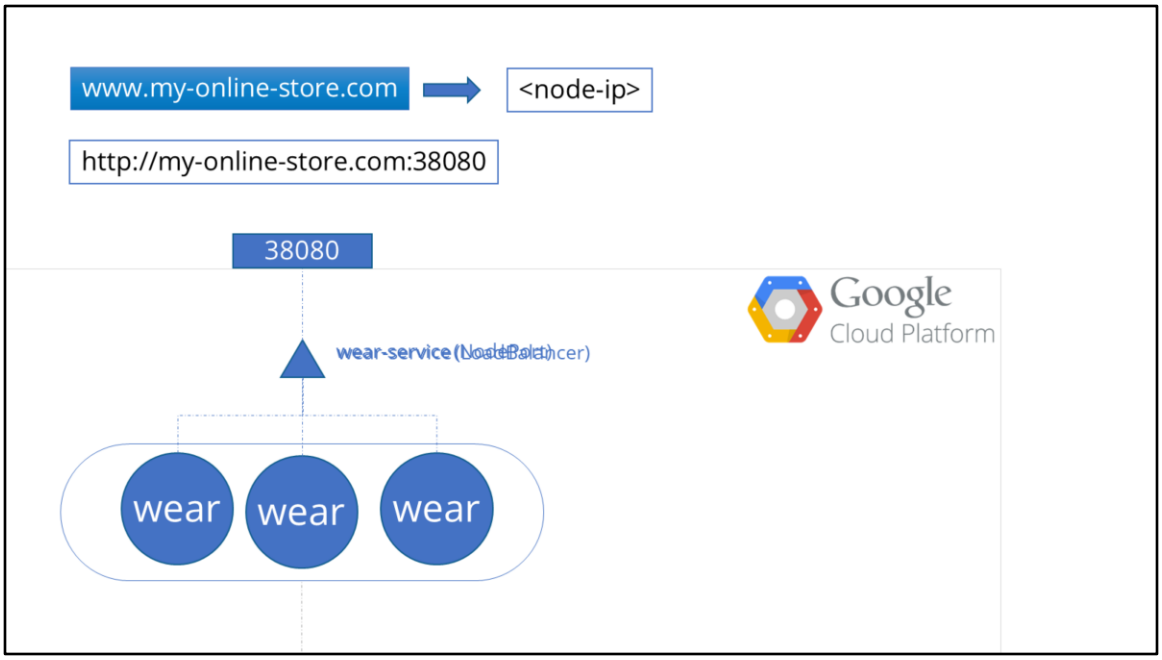


For example, we do not want the users to have to type in IP address everytime. So you configure your DNS server to point to the IP of the nodes. Your users can now access your application using the URL my-online-store.com and port 38080.



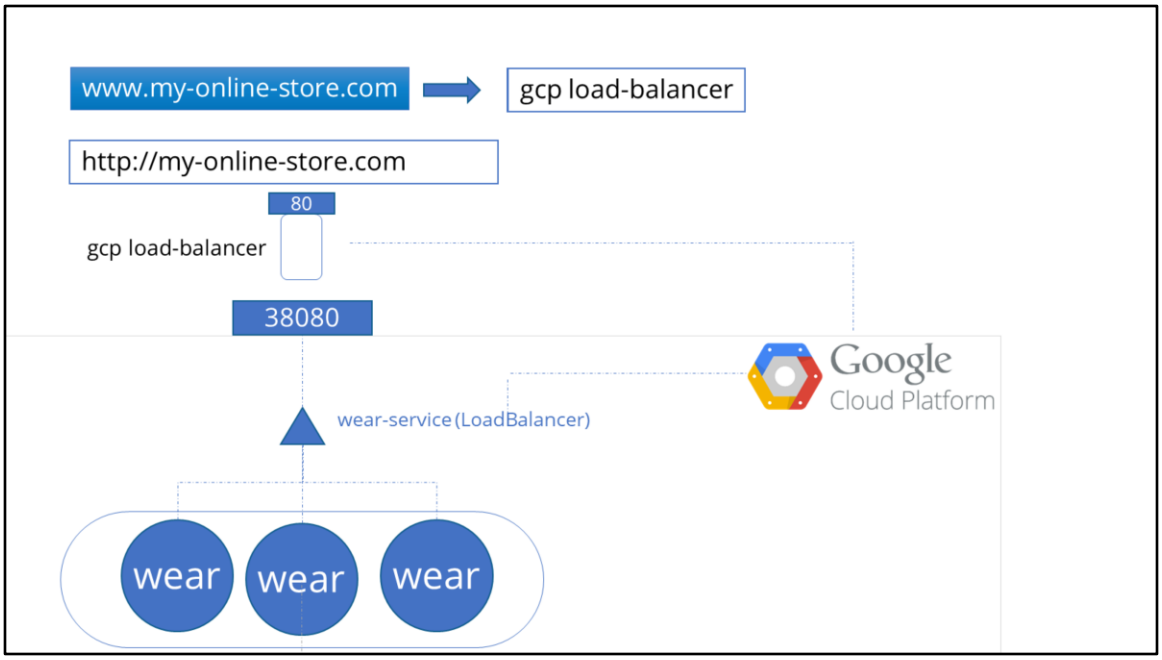
Now, you don't want your users to have to remember port number either. However, Service NodePorts can only allocate high numbered ports which are greater than 30,000. So you then bring in an additional layer between the DNS server and your cluster, like a proxy server, that proxies requests on port 80 to port 38080 on your nodes. You then point your DNS to this server, and <click> users can now access your application by simply visiting my-online-store.com.

<click> Now, this is if your application is hosted on-prem in your datacenter.

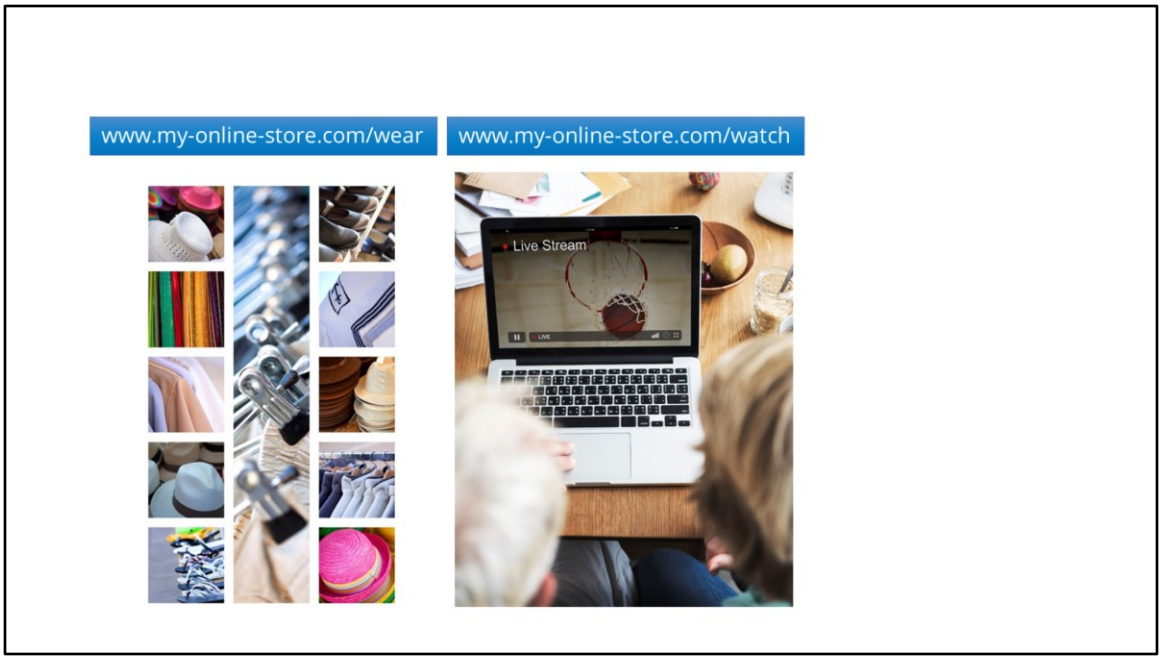


Let's take a step back and see what you could do it you were on a public cloud environment like Google Cloud Platform.

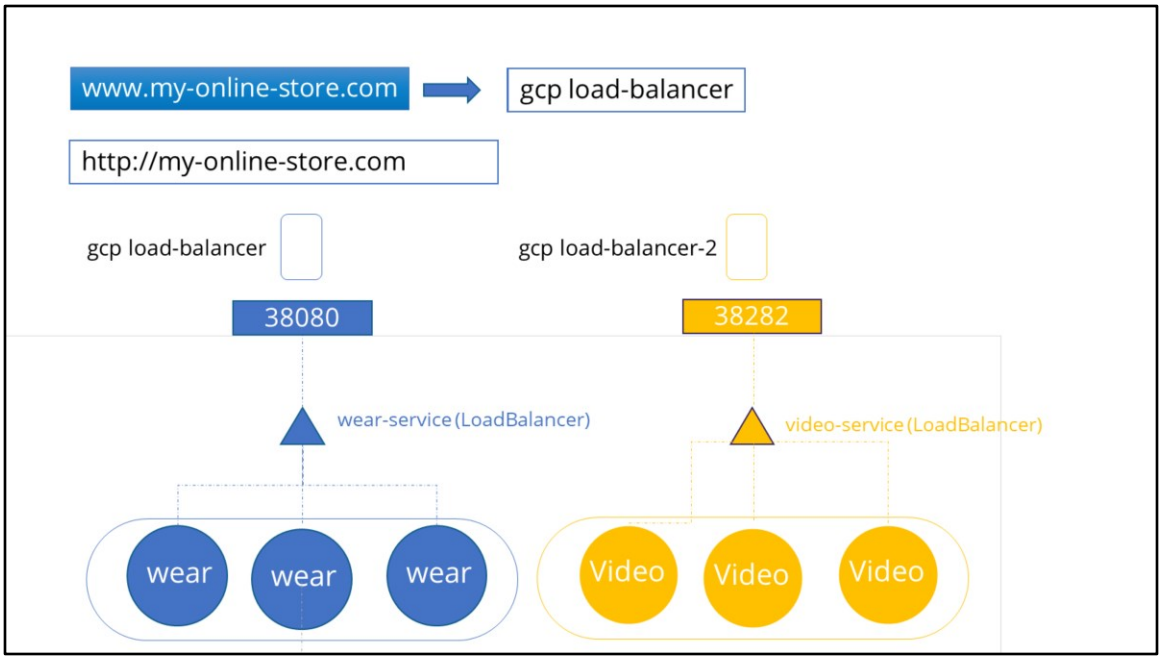
In that case, instead of creating a service of type NodePort for your wear application, you could set it to type LoadBalancer. When you do that Kubernetes would still do everything that it has to do for a NodePort, which is to provision a high port for the service, but in addition to that kubernetes also sends a request to Google Cloud Platform to provision a network load balancer for this service.



On receiving the request, GCP would then automatically deploy a LoadBalancer configured to route traffic to the service ports on all the nodes and return its information to kubernetes. The LoadBalancer has an external IP that can be provided to users to access the application. In this case we set the DNS to point to this IP and users access the application using the URL my-online-store.com. Perfect!!

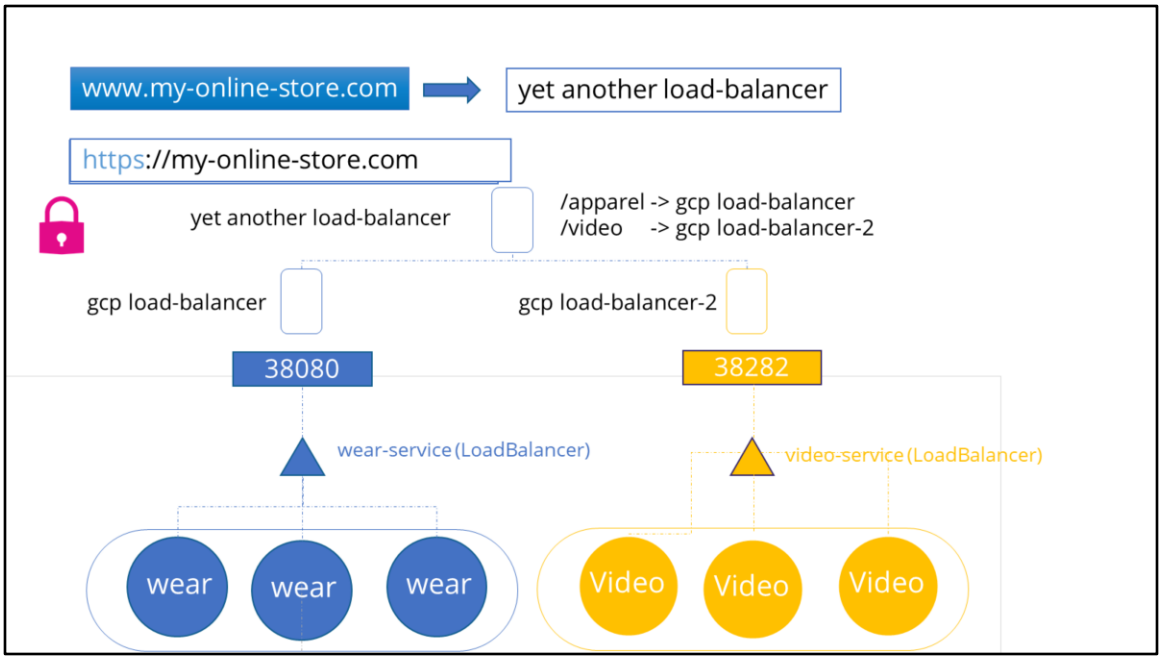


Your companies business grows and you now have new services for your customers. For example, a video streaming service. You want your users to be able to access your new video streaming service by going to my-online-store.com/watch. You'd like to make your old application accessible at my-online.store.com / wear.



Your developers developed the new video streaming application as a completely different application, as it has nothing to do with the existing one. However to share the cluster resources, you deploy the new application as a separate deployment within the same cluster. You create a service called video-service of type LoadBalancer. Kubernetes provisions port 38282 for this service and also provisions a Network LoadBalancer on the cloud. The new load balancer has a new IP. Remember you must pay for each of these load balancers and having many such load balancers can inversely affect your cloud bill.

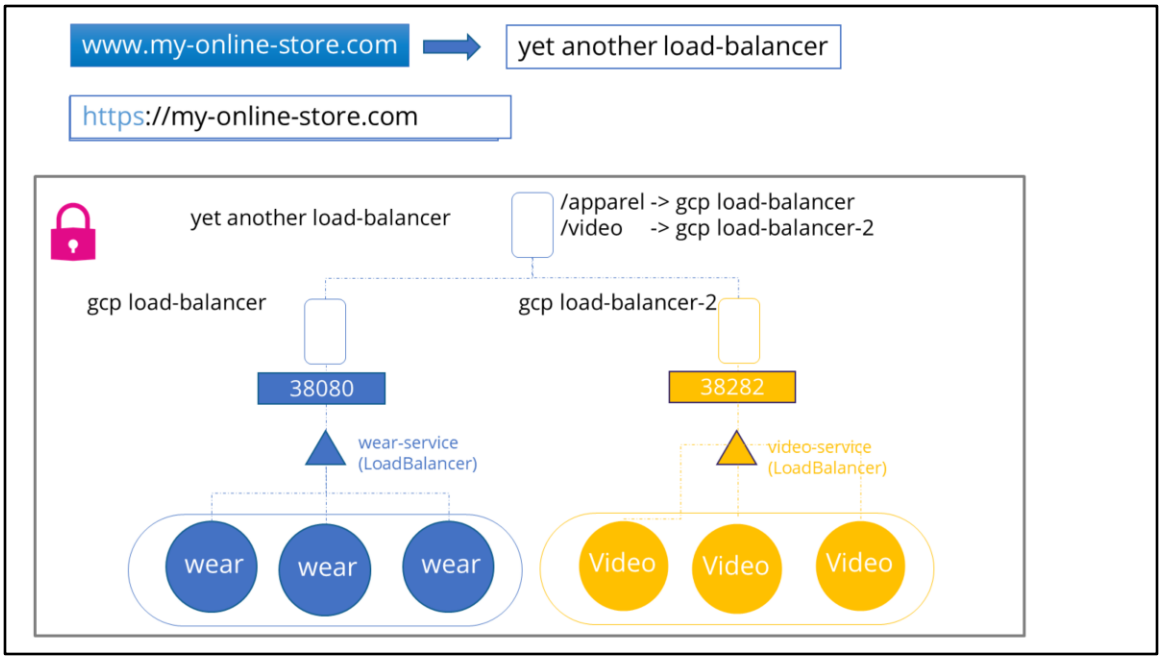
So how do you direct traffic between each of these load balancers based on the URL that the user types in?



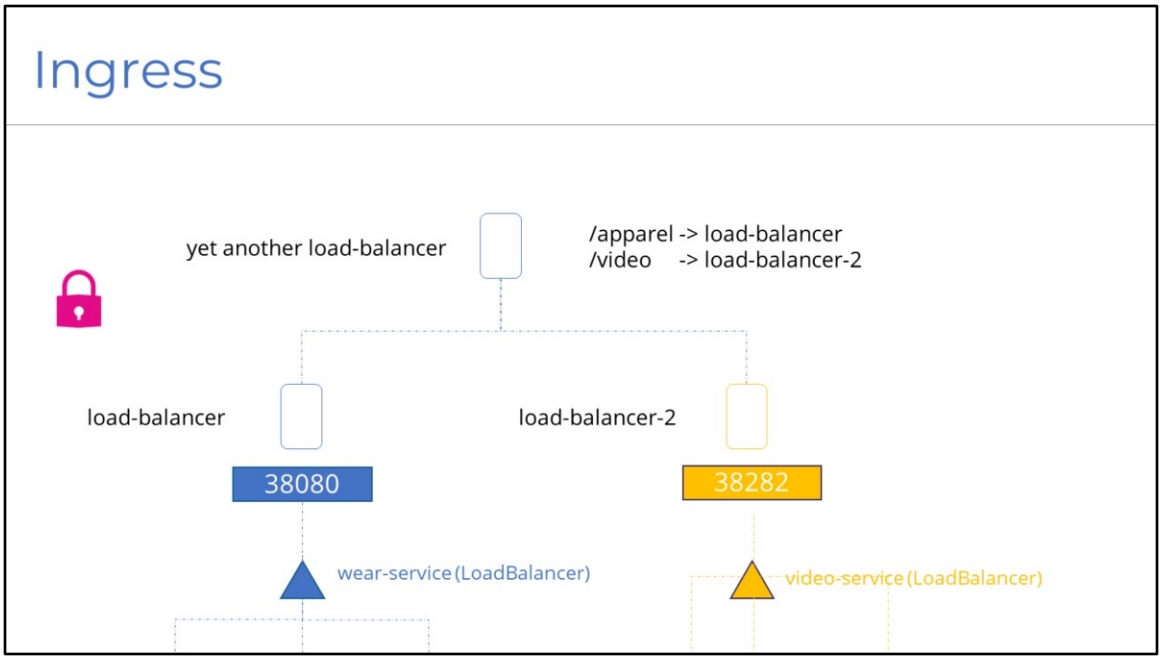
You need yet another proxy or load balancer that can re-direct traffic based on URLs to the different services. Every time you introduce a new service you have to re-configure the load balancer.

And finally you also need to enable SSL for your applications, so your users can access your application using https. Where do you configure that? It can be done at different levels, either at the application level itself, or at the load balancer or proxy server level, but which one? You don't want your developers to implement it in their applications as they would do it in different ways. You want it to be configured in one place with minimal maintenance.

Now that's a lot of different configuration and all of these becomes difficult to manage when your application scales. It requires involving different individuals in different teams. You need to configure your firewall rules for each new service. And its expensive as well, as for each service a new cloud native load balancer needs to be provisioned.

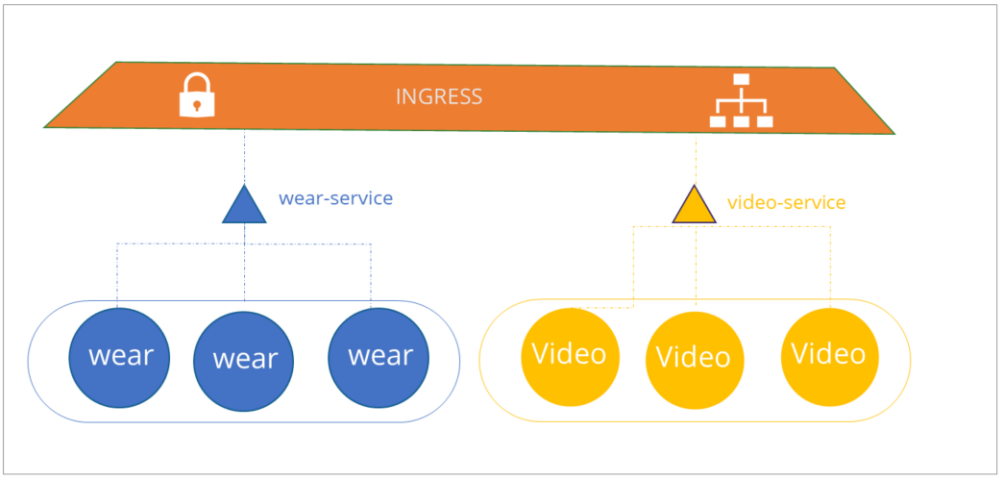


Wouldn't it be nice if you could manage all of that within the Kubernetes cluster, and have all that configuration as just another kubernetes definition file, that lives along with the rest of your application deployment files?

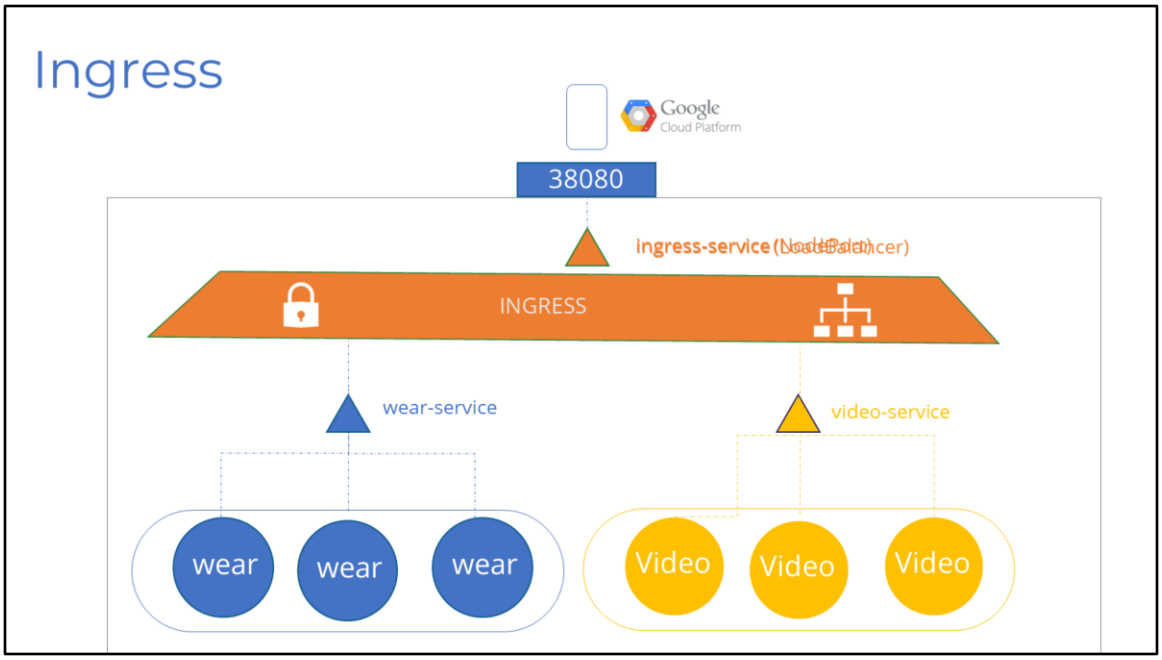


That's where Ingress comes in. Ingress helps your users access your application using a single Externally accessible URL, that you can configure to route to different services within your cluster based on the URL path, at the same time implement SSL security as well.

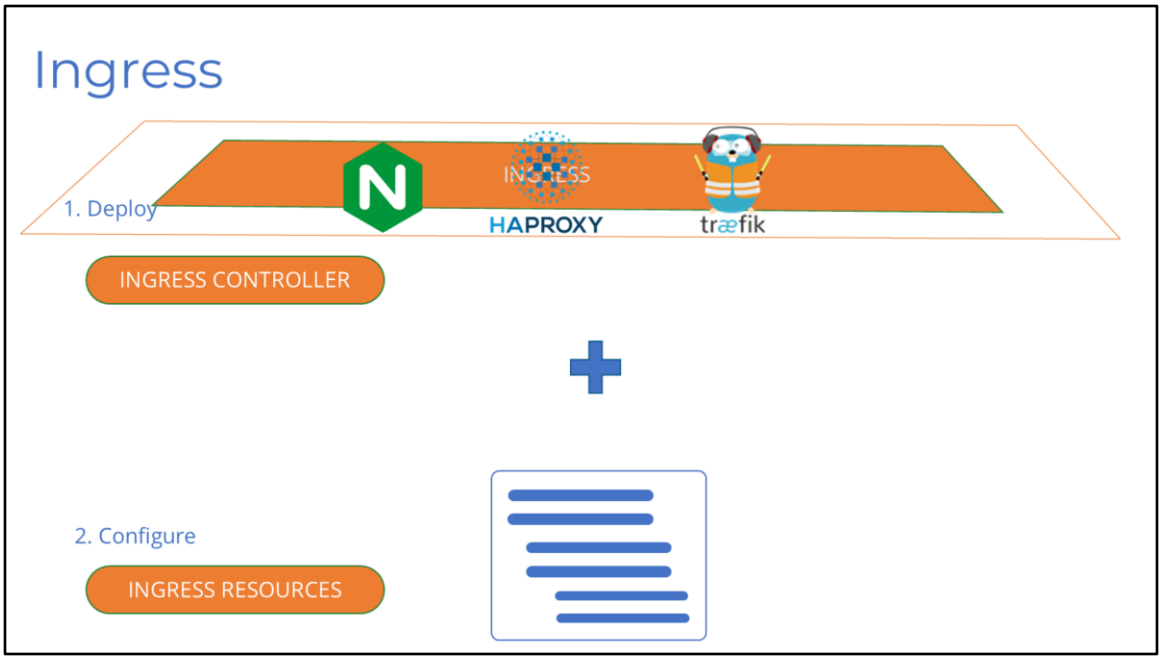
Ingress



Simply put, think of ingress as a layer 7 load balancer built-in to the kubernetes cluster that can be configured using native kubernetes primitives just like any other object in kubernetes.



Now remember, even with Ingress you still need to expose it to make it accessible outside the cluster. So you still have to either publish it as a NodePort or with a Cloud Native LoadBalancer. But that is just a one time configuration. Going forward you are going to perform all your load balancing, Auth, SSL and URL based routing configurations on the Ingress controller.



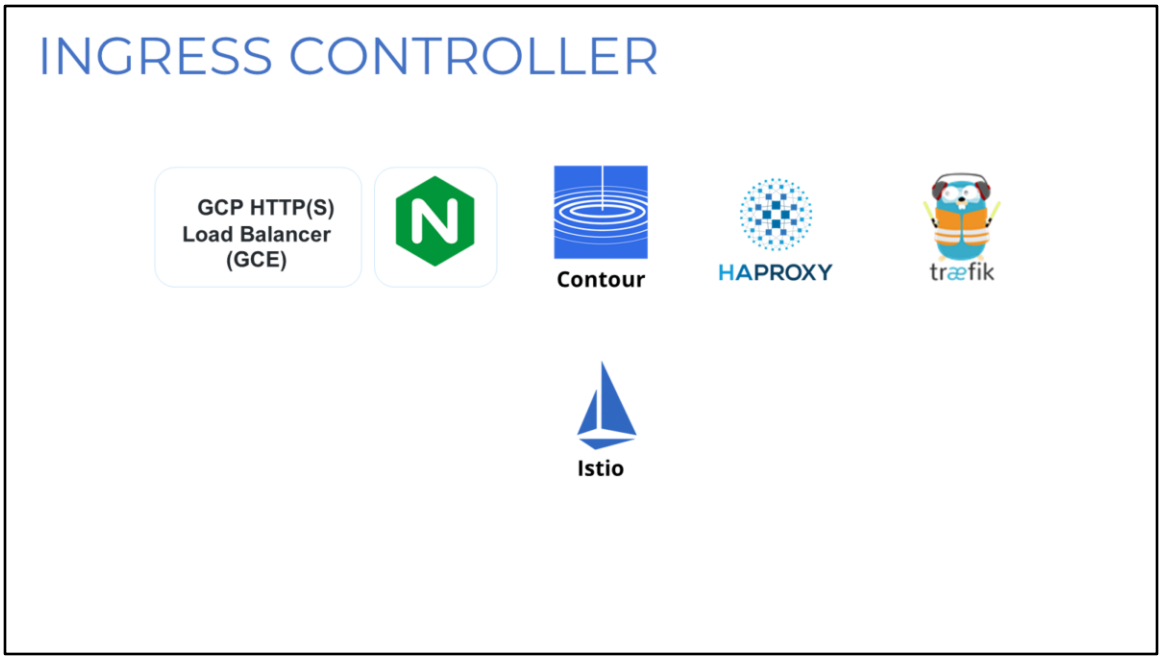
So how does it work? What is it? Where is it? How can you see it? How can you configure it? How does it load balance? How does it implement SSL?

Without ingress, how would YOU do all of these? I would use a reverse-proxy or a load balancing solution like NGINX or HAProxy or Traefik. I would deploy them on my kubernetes cluster and configure them to route traffic to other services. The configuration involves defining URL Routes, configuring SSL certificates etc.

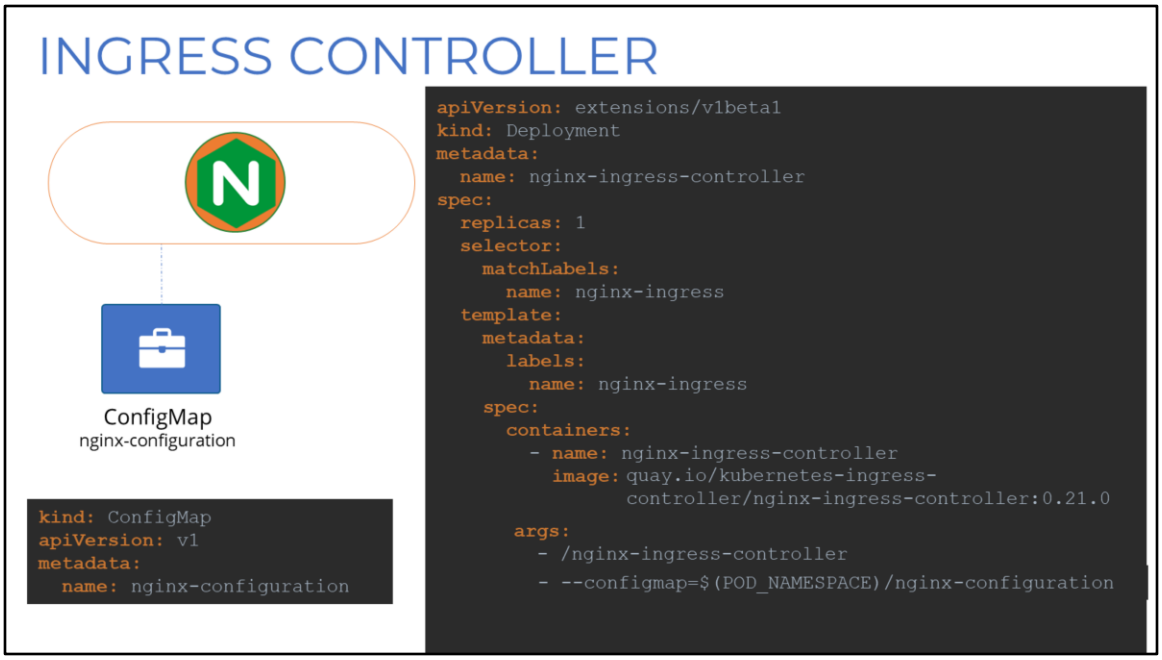
Ingress is implemented by Kubernetes in kind of the same way. You first deploy a supported solution, which happens to be any of these listed here, and then specify a set of rules to configure Ingress. The solution you deploy is called as an Ingress Controller. And the set of rules you configure are called as Ingress Resources. Ingress resources are created using definition files like the ones we used to create PODs, Deployments and services earlier in this course.

Now remember a kubernetes cluster does NOT come with an Ingress Controller by default. If you setup a cluster following the demos in this course, you won't have an ingress controller built-in to it. So if you simply create ingress resources and expect them to work, they wont.

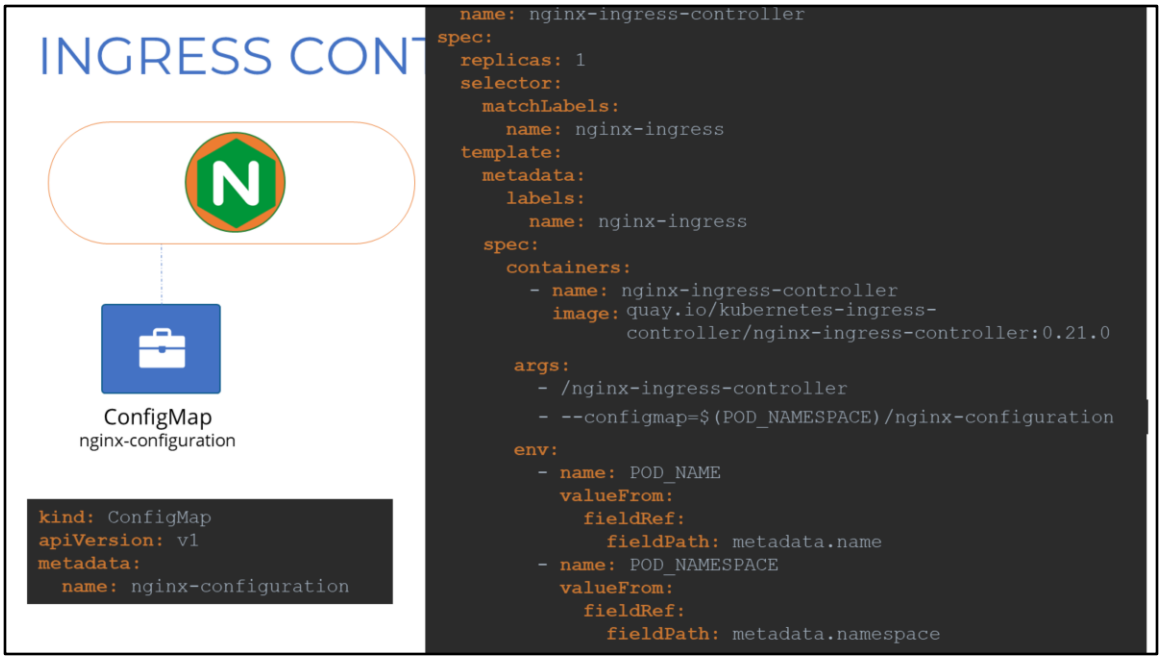
Let’s look at each of these in a bit more detail now.



As I mentioned you do not have an Ingress Controller on Kubernetes by default. So you MUST deploy one. What do you deploy? There are a number of solutions available for Ingress, a few of them being GCE - which is Googles Layer 7 HTTP Load Balancer. NGINX, Contour, HAPROXY, TRAFIK and Istio. <click> Out of this, GCE and NGINX are currently being supported and maintained by the Kubernetes project. And in this lecture we will use NGINX as an example. These Ingress Controllers are not just another load balancer or nginx server. The load balancer components are just a part of it. Ingress controllers have additional intelligence built into them to monitor the kubernetes cluster for new definitions or ingress resources and configure the nginx server accordingly.



An NGINX Controller is deployed as just another deployment in Kubernetes. So we start with a deployment file definition, named nginx-ingress-controller. With 1 replica and a simple pod definition template. We will label it nginx-ingress and the image used is nginx-ingress-controller with the right version. This is a special build of NGINX built specifically to be used as an ingress controller in kubernetes. So it has its own set of requirements. Within the image the nginx program is stored at location /nginx-ingress-controller. So you must pass that as the command to start the nginx-controller-service. If you have worked with NGINX before, you know that it has a set of configuration options such as the path to store the logs, keep-alive threshold, ssl settings, session timeout etc. In order to decouple these configuration data from the nginx-controller image, you must create a ConfigMap object and pass that in. Now remember the ConfigMap object need not have any entries at this point. A blank object will do. But creating one makes it easy for you to modify a configuration setting in the future. You will just have to add it in to this ConfigMap and not have to worry about modifying the nginx configuration files.

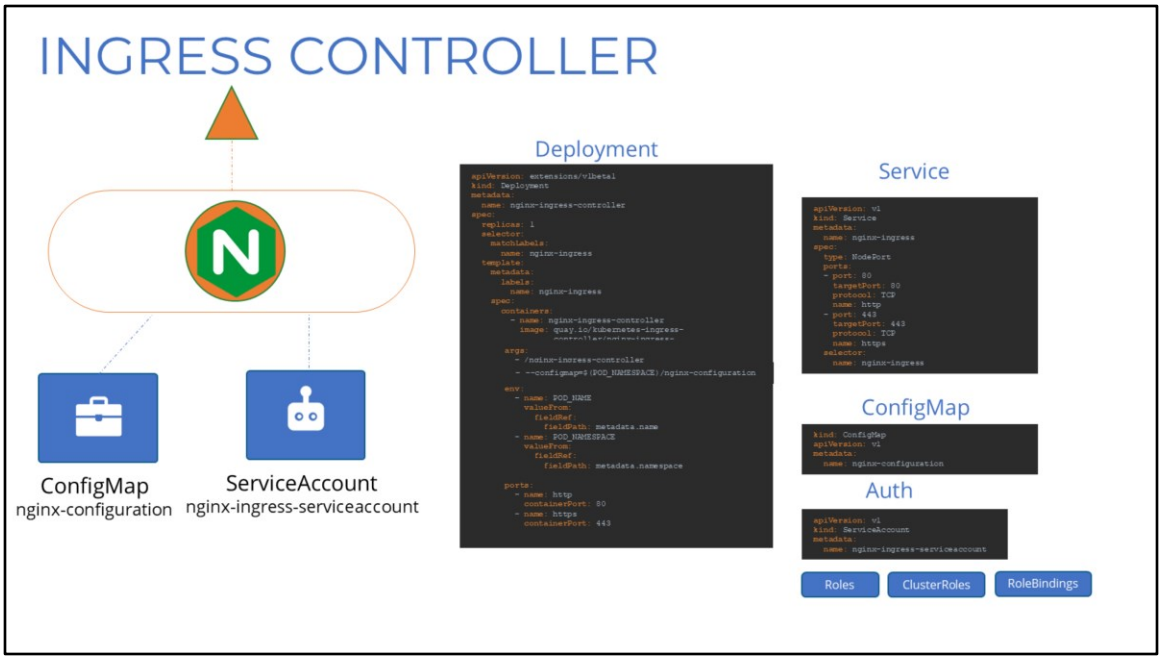


You must also pass in two environment variables that carry the POD's name and namespace it is deployed to. The nginx service requires these to read the configuration data from within the POD.

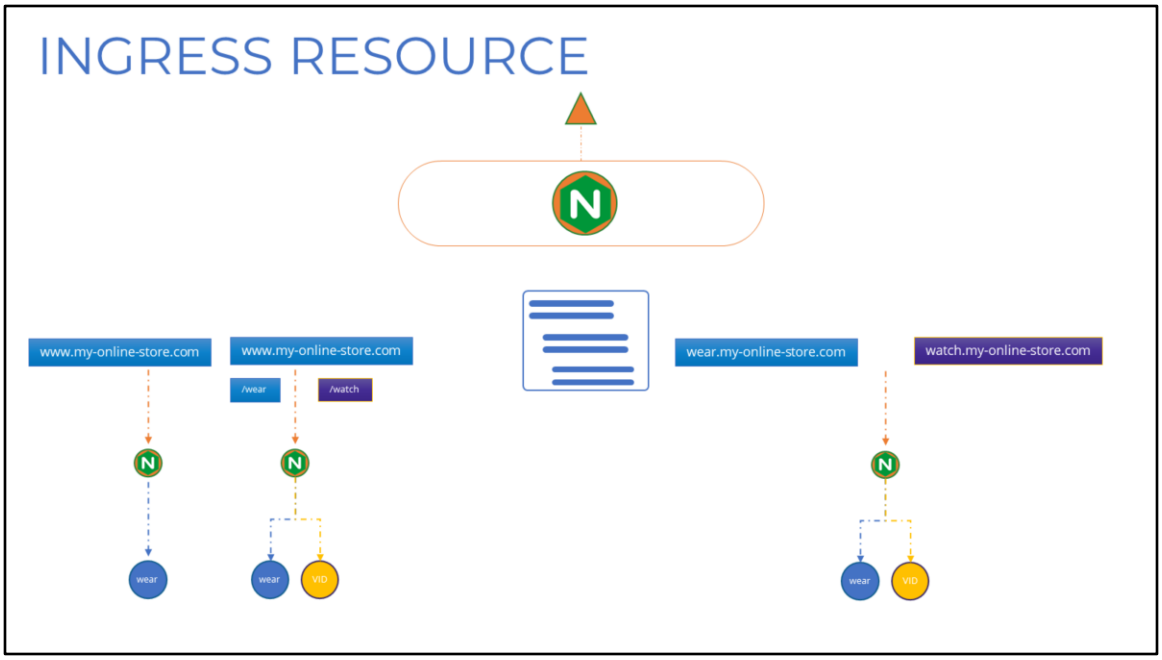
And finally specify the ports used by the ingress controller which happens to be 80 and 443.

We then need a service to expose the ingress controller to the external world. So we create a service of type NodePort with the nginx-ingress label selector to link the service to the deployment.

As mentioned before, the Ingress controllers have additional intelligence built into them to monitor the kubernetes cluster for ingress resources and configure the underlying nginx server when something is changed. But for the ingress controller to do this, it requires a service account with the right set of permissions. For that we create a service account with the correct roles, and role bindings.

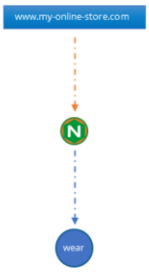


So to summarize, with a deployment of the nginx-ingress image, a service to expose it, a ConfigMap to feed nginx configuration data, and a service account with the right permissions to access all of these objects we should be ready with an ingress controller in its simplest form.



Now on to the next part, of creating Ingress Resources. An Ingress Resource is a set of rules and configurations applied on the ingress controller. You can configure rules to say, simply forward all incoming traffic to a single application, or route traffic to different applications based on the URL. So if user goes to my-online-store.com/wear, then route to one app, or if the user visits the /watch URL then route to the video app. Or you could route user based on the domain name itself. For example, if the user visits wear.my-online-store.com, the route to the wear app or else route to the video app.

INGRESS RESOURCE

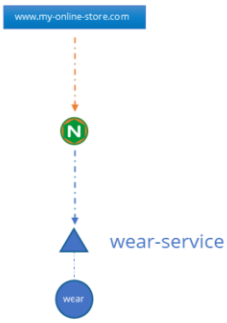


Ingress-wear.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear
spec:
```

Let us look at how to configure these in a bit more detail. The Ingress Resource is created with a Kubernetes Definition file. In this case, ingress-wear.yaml. As with any other object, we have apiVersion, kind, metadata and spec. The apiVersion is extensions/v1beta1, kind is Ingress, we will name it ingress-wear. And under spec we have backend.

INGRESS RESOURCE



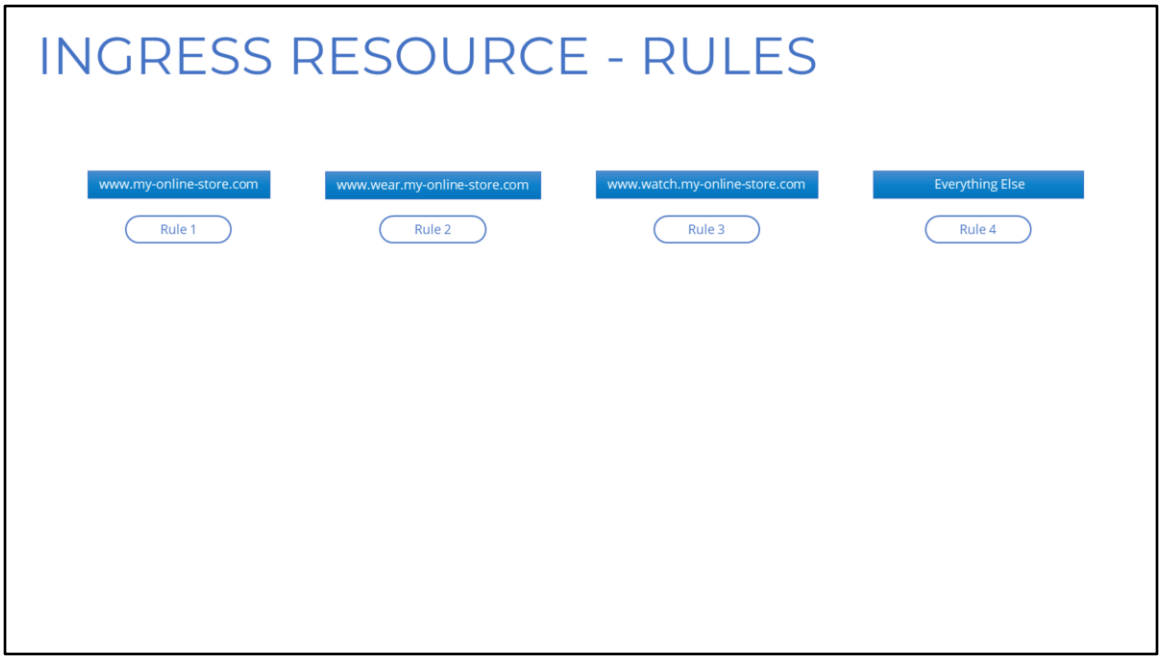
Ingress-wear.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear
spec:
 backend:
     serviceName: wear-service
     servicePort: 80
```

```
kubectl create -f Ingress-wear.yaml
ingress.extensions/ingress-wear created
```

```
kubectl get ingress
NAME           HOSTS     ADDRESS   PORTS
ingress-wear   *         80        2s
```

So the traffic is, of course, routed to the application services and not PODs directly. The Backend section defines where the traffic will be routed to. So if it's a single backend, then you don't really have any rules. You can simply specify the service name and port of the backend wear service. Create the ingress resource by running the kubectl create command. View the created ingress by running the kubectl get ingress command. The new ingress is now created and routes all incoming traffic directly to the wear-service.

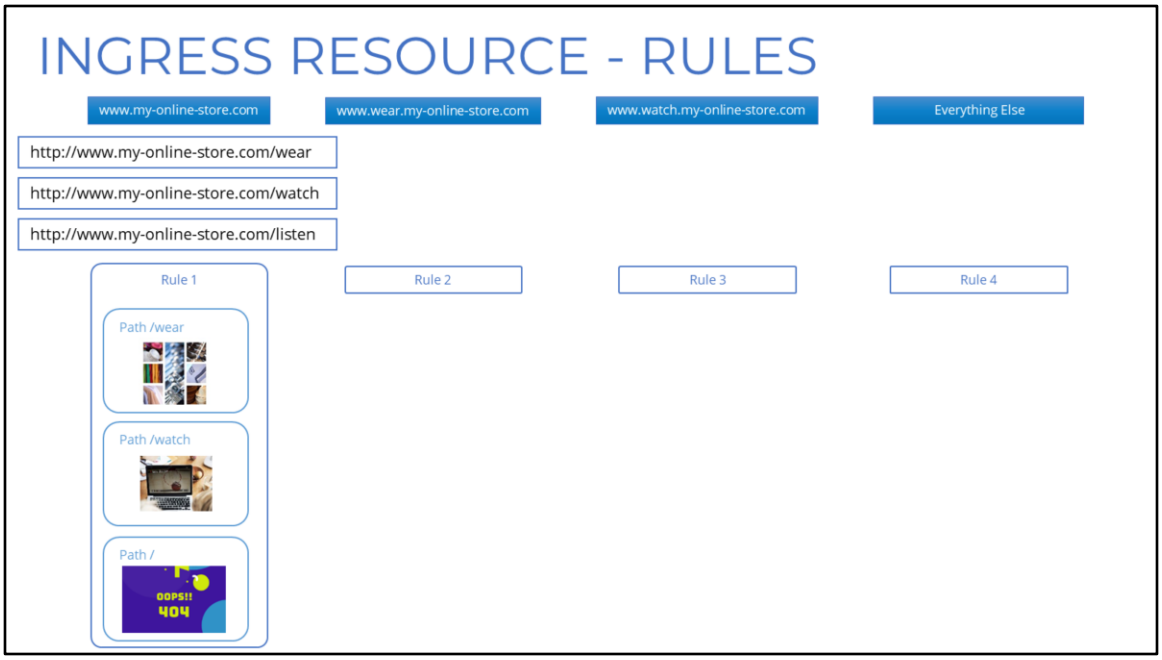


You use rules, when you want to route traffic based on different conditions. For example, you create one rule for traffic originating from each domain or hostname. That means when users reach your cluster using the domain name, my-online-store.com, you can handle that traffic using rule1. When users reach your cluster using domain name wear.my-online-store.com, you can handle that traffic using a separate Rule2. Use Rule3 to handle traffic from watch.my-online-store.com. And say use a 4th rule to handle everything else.

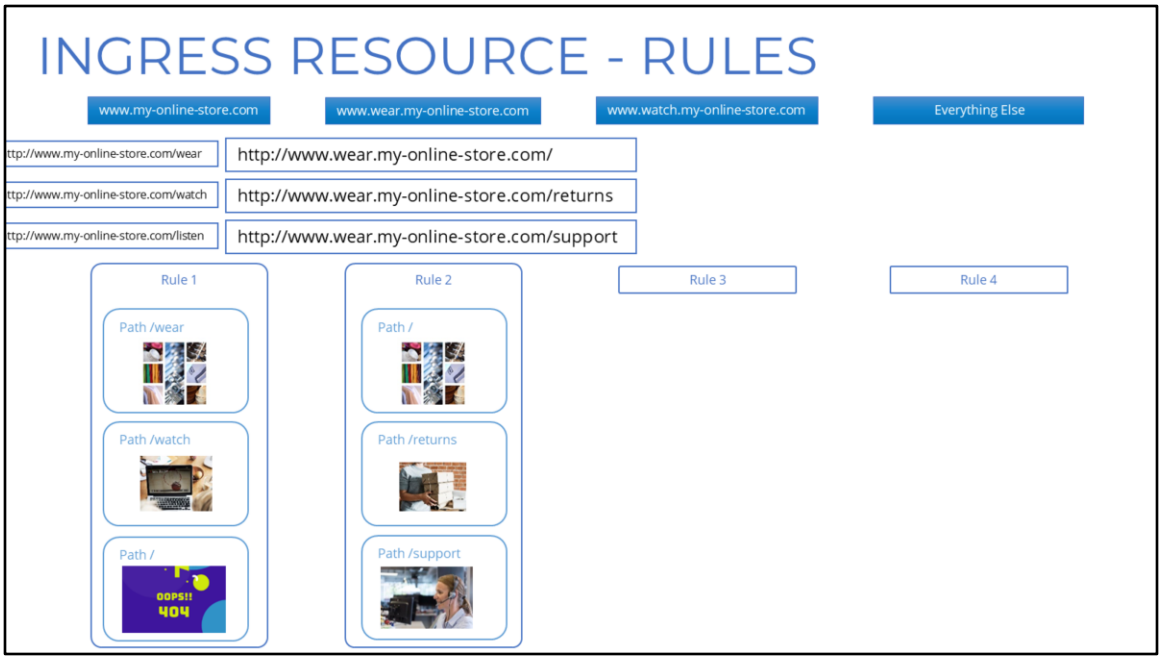
INGRESS RESOURCE - RULES

DNS Name	Forward IP
www.my-online-store.com	10.123.23.12 (INGRESS SERVICE)
www.wear.my-online-store.com	10.123.23.12
www.watch.my-online.store.com	10.123.23.12
www.my-wear-store.com	10.123.23.12
www.my-watch-store.com	10.123.23.12

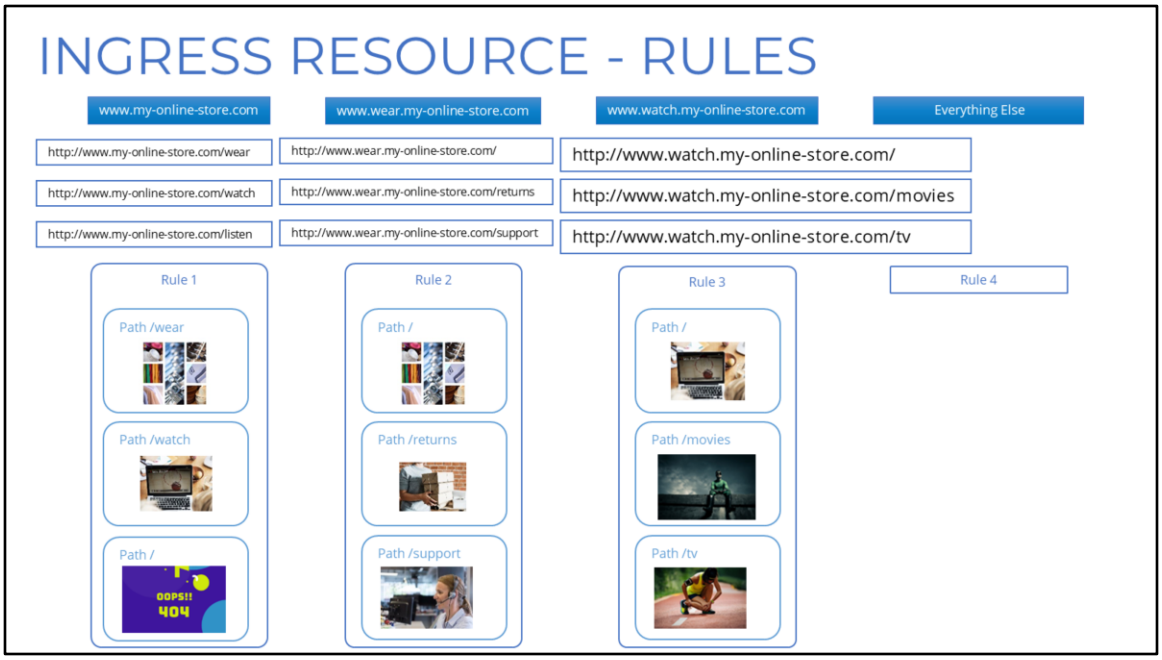
And just in case you didn't know, you could get different domains names to reach your cluster, by adding multiple DNS entries, all pointing to the same Ingress controller service on your kubernetes cluster.



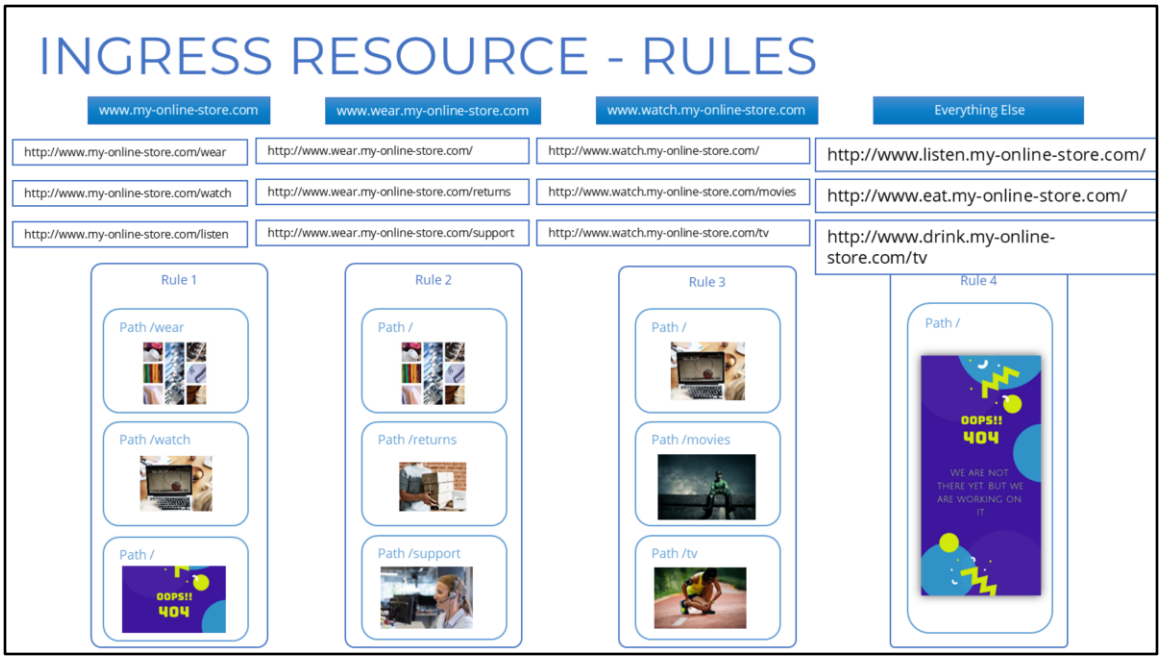
Now within each rule you can handle different paths. For example, within Rule 1 you can handle the wear path to route that traffic to the clothes application. And a watch path to route traffic to the video streaming application. And a third path that routes anything other than the first two to a 404 not found page.



Similarly, the second rule handles all traffic from wear.my-online-store.com. You can have path definition within this rule, to route traffic based on different paths. For example, say you have different applications and services within the apparel section for shopping, or returns, or support, when a user goes to wear.my-online.store.com/, by default they reach the shopping page. But if they go to exchange or support URL, they reach different backend services.

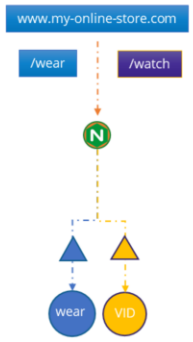


The same goes for Rule 3, where you route traffic to watch.my-online-store.com to the video streaming application. But you can have additional paths in it such as movies or tv.



And finally anything other than the ones listed here will got to the 4th Rule, that would simply show a 404 Not Found Error page. So remember, you have rules at the top for each host or domain name. And within each rule you have different paths to route traffic based on the URL.

INGRESS RESOURCE



Ingress-wear.yaml

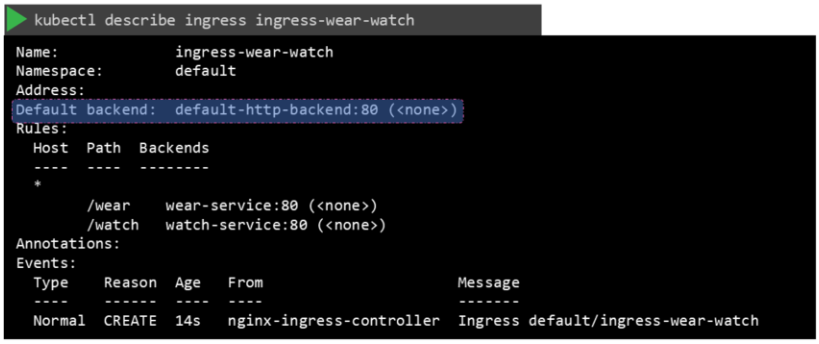
```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear
spec:
  backend:
    serviceName: wear-service
    servicePort: 80
```

Ingress-wear-watch.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
  rules:
  - http:
      paths:
      - path: /wear


      - path: /watch
        backend:
          serviceName: watch-service
          servicePort: 80
```

Now, let's look at how we configure ingress resources in Kubernetes. We will start where we left off. We start with a similar definition file. This time under spec, we start with a set of rules. Now our requirement here is to handle all traffic coming to my-online-store.com and route them based on the URL path. So we just need a single Rule for this, since we are only handling traffic to a single domain name, which is my-online-store.com in this case. Under rules we have one item, which is an http rule in which we specify different paths. So paths is an array of multiple items. One path for each url. Then we move the backend we used in the first example under the first path. The backend specification remains the same, it has a servicename and serviceport. Similarly we create a similar backend entry to the second URL path, for the watch-service to route all traffic coming in through the /watch url to the watch-service. Create the ingress resource using the kubectl create command.

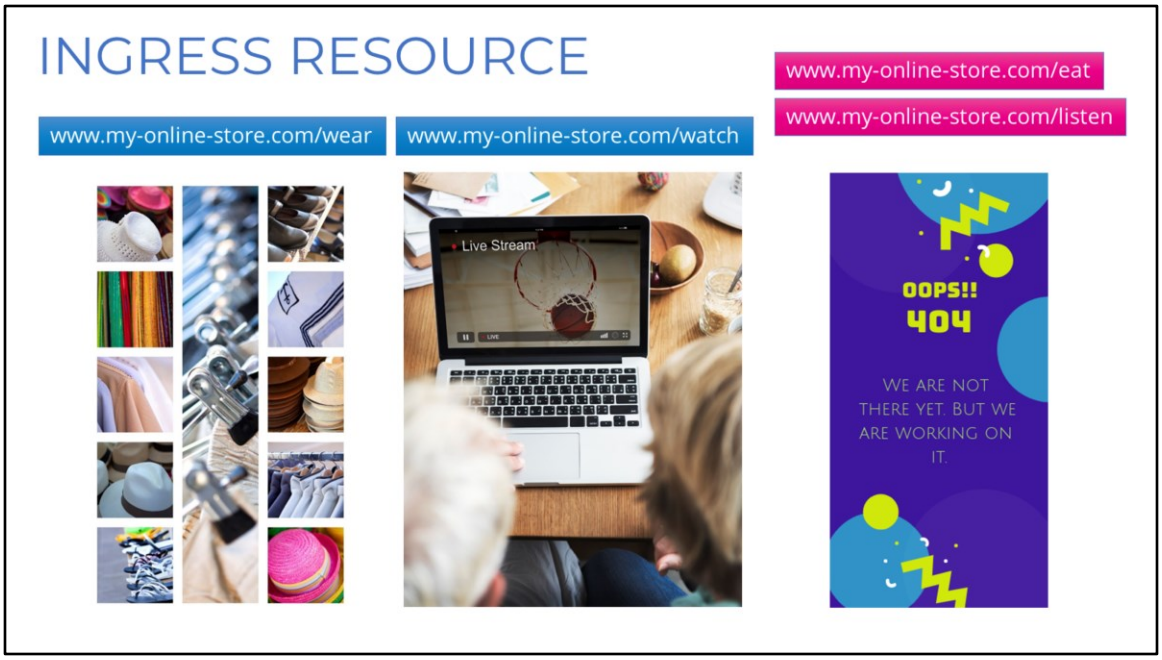
INGRESS RESOURCE



Once created, view additional details about the ingress resource by running the kubectl describe ingress command. You now see two backend URLs under the rules, and the backend service they are pointing to. Just as we created it.

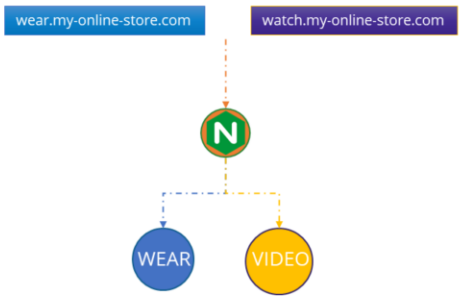
Now, If you look closely in the output of this command, you see that there is something about a Default-backend. Hmmm. What might that be?

If a user tries to access a URL that does not match any of these rules, then the user is directed to the service specified as the default backend. In this case it happens to be a service named default-http-backend. So you must remember to deploy such a service.



Back in your application, say a user visits the URL my-online-store.com/listen or /eat and you don't have an audio streaming or a food delivery service, you might want to show them a nice message. You can do this by configuring a default backend service to display this 404 Not Found error page.

INGRESS RESOURCE



Ingress-wear-watch.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
  rules:
  - host: wear.my-online-store.com
    http:
      paths:
      - backend:
          serviceName: wear-service
          servicePort: 80
  - host: watch.my-online-store.com
    http:
      paths:
      - backend:
          serviceName: watch-service
          servicePort: 80
```

The third type of configuration is using domain names or hostnames. We start by creating a similar definition file for Ingress. Now that we have two domain names, we create two rules. One for each domain. To split traffic by domain name, we use the host field. The host field in each rule matches the specified value with the domain name used in the request URL and routes traffic to the appropriate backend. In this case note that we only have a single backend path for each rule. Which is fine. All traffic from these domain names will be routed to the appropriate backend irrespective of the URL Path used. You can still have multiple path specifications in each of these to handle different URL paths.

INGRESS RESOURCE

Ingress-wear-watch.yaml

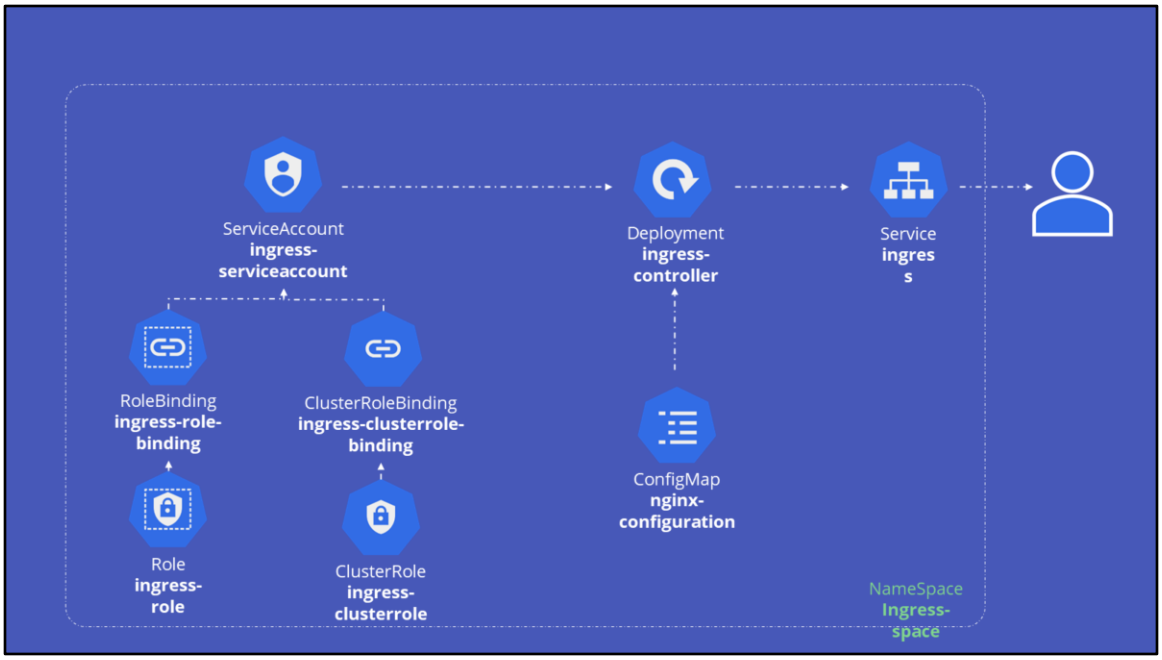
```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
  rules:
  - http:
      paths:
      - path: /wear
        backend:
          serviceName: wear-service
          servicePort: 80
      - path: /watch
        backend:
          serviceName: watch-service
          servicePort: 80
```

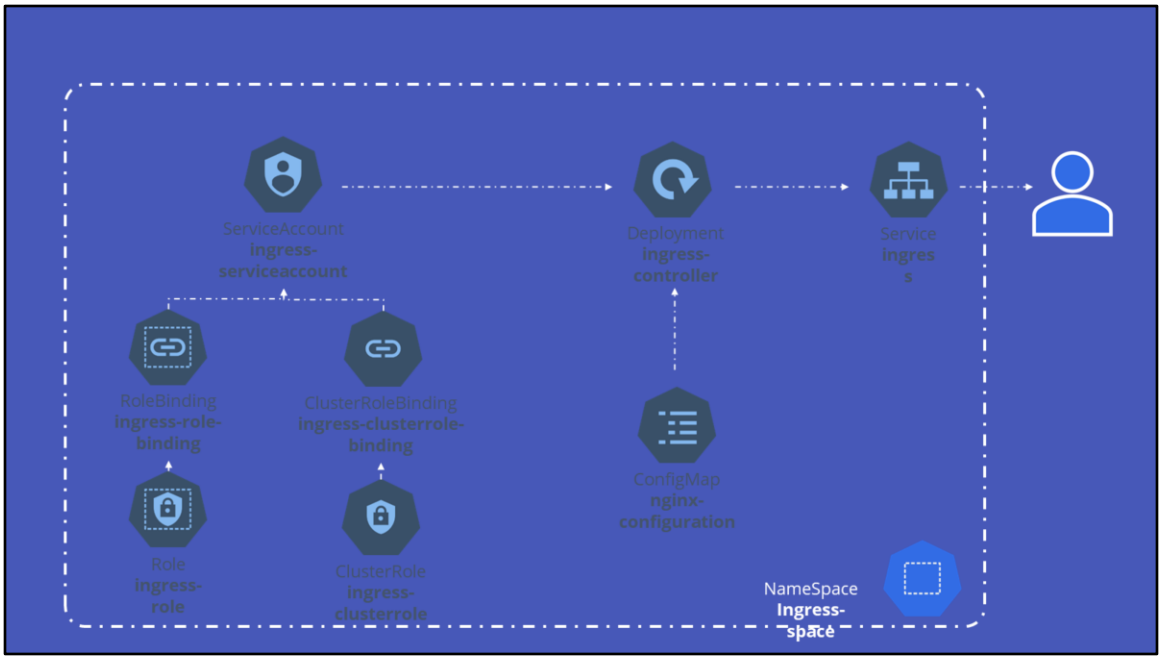
Ingress-wear-watch.yaml

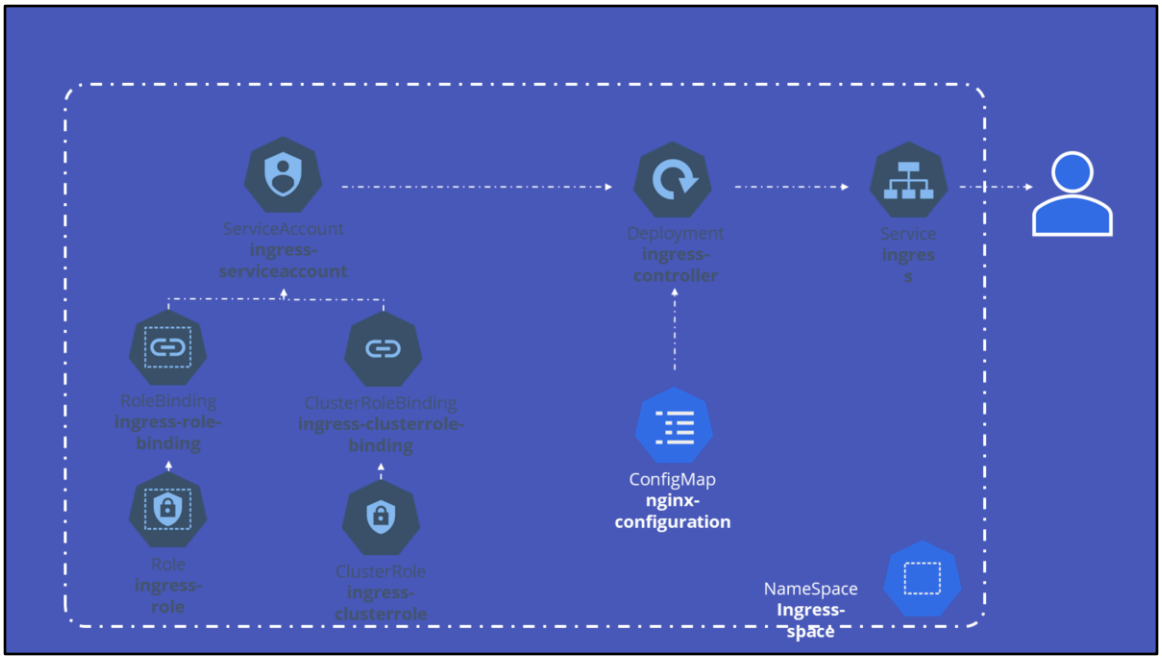
```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:

  rules:
  - host: wear.my-online-store.com
    http:
      paths:
      - backend:
          serviceName: wear-service
          servicePort: 80
  - host: watch.my-online-store.com
    http:
      paths:
      - backend:
          serviceName: watch-service
          servicePort: 80
```

Let's compare the two. Splitting traffic by URL had just one rule and we split the traffic with two paths. To split traffic by hostname, we used two rules and one path specification in each rule.

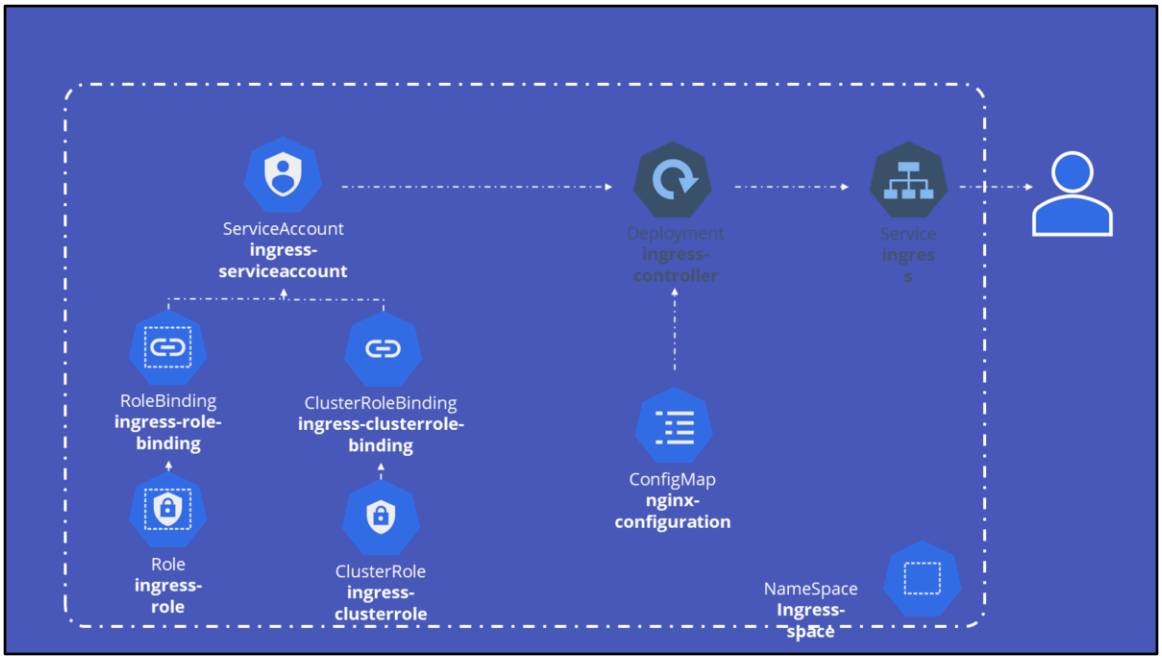
Well that's it for this lecture. Let us now head over to the practice test section and practice working on ingress. There are two types of labs in this section. The first one is where an ingress controller, resources and applications are already deployed and you basically view and walkthrough the environment, gather data and answer questions. Towards the end you would create or modify ingress resources based on the needs. In the second practice test you will be deploying an ingress controller and resources from scratch. Good luck and hope you enjoy the labs. I will see you in the next lecture.


ServiceAccount
ingress-serviceaccount
Deployment
ingress-controller
Service
ingress
RoleBinding
ingress-role-binding
ClusterRoleBinding
ingress-clusterrole-binding
ConfigMap
nginx-configuration
Role
ingress-role
ClusterRole
ingress-clusterrole
NameSpace
Ingress-space

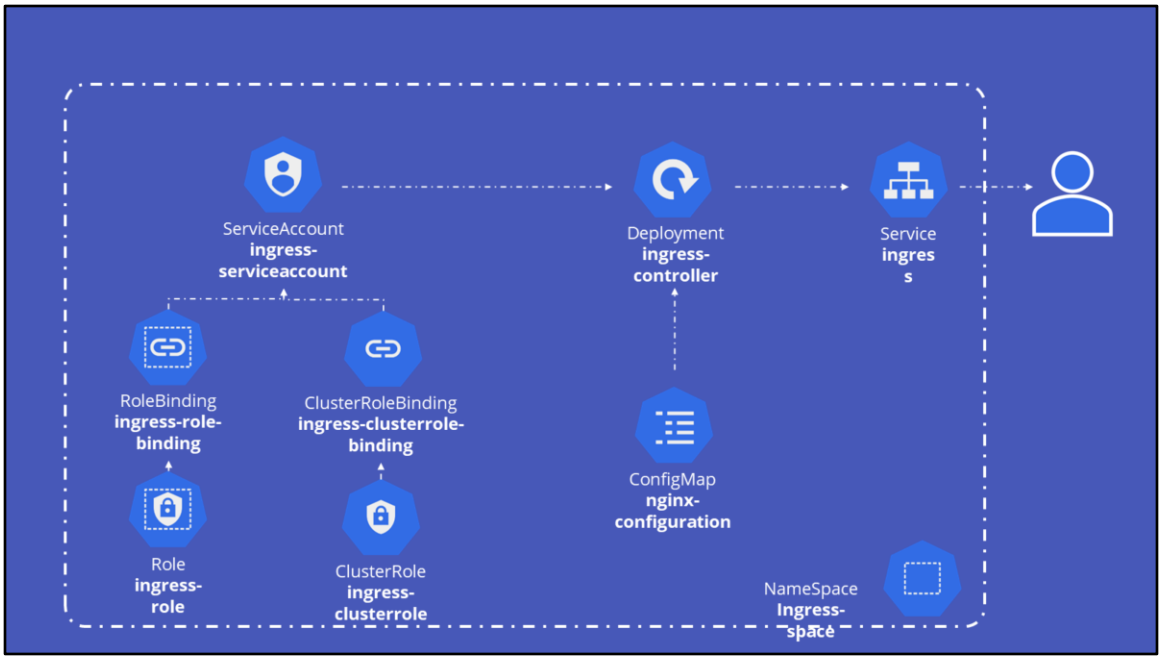

ServiceAccount
ingress-serviceaccount
Deployment
ingress-controller
Service
ingress
RoleBinding
ingress-role-binding
ClusterRoleBinding
ingress-clusterrole-binding
ConfigMap
nginx-configuration
Role
ingress-role
ClusterRole
ingress-clusterrole
NameSpace
Ingress-space


ServiceAccount
ingress-serviceaccount
Deployment
ingress-controller
Service
ingress
RoleBinding
ingress-role-binding
ClusterRoleBinding
ingress-clusterrole-binding
ConfigMap
nginx-configuration
Role
ingress-role
ClusterRole
ingress-clusterrole
NameSpace
Ingress-space


ServiceAccount
ingress-serviceaccount
Deployment
ingress-controller
Service
ingress
RoleBinding
ingress-role-binding
ClusterRoleBinding
ingress-clusterrole-binding
ConfigMap
nginx-configuration
Role
ingress-role
ClusterRole
ingress-clusterrole
NameSpace
Ingress-space


ServiceAccount
ingress-serviceaccount
Deployment
ingress-controller
Service
ingress
RoleBinding
ingress-role-binding
ClusterRoleBinding
ingress-clusterrole-binding
ConfigMap
nginx-configuration
Role
ingress-role
ClusterRole
ingress-clusterrole
NameSpace
Ingress-space


ServiceAccount
ingress-serviceaccount
Deployment
ingress-controller
Service
ingress
RoleBinding
ingress-role-binding
ClusterRoleBinding
ingress-clusterrole-binding
ConfigMap
nginx-configuration
Role
ingress-role
ClusterRole
ingress-clusterrole
NameSpace
Ingress-space


ServiceAccount
ingress-serviceaccount
Deployment
ingress-controller
Service
ingress
RoleBinding
ingress-role-binding
ClusterRoleBinding
ingress-clusterrole-binding
ConfigMap
nginx-configuration
Role
ingress-role
ClusterRole
ingress-clusterrole
NameSpace
Ingress-space

BEST PRACTICES

Create in its own namespace
Use SSL