1. What are the main components of the Kubernetes?

Kubernetes Architecture

Master controls the cluster, and the nodes in it. It ensures the execution only happens in nodes and coordinates the act.

Nodes host the containers; in-fact these Containers are grouped logically to form Pods. Each node can run multiple such Pods, which are a group of containers, that interact with each other, for a deployment.

Replication Controller is Master's resource to ensure that the requested no. of pods are always running on nodes.

- **Replica Set:** replica sets are created by deployment, these deployments contains declaration of containers which you want to run in cluster. like image/tag, env variable, data volumes, Kubernetes has several components in its architecture.

Service is an object on Master that provides load balancing across a replicated group of Pods.

- **minion/node/worker node:** is the node on which all the services run. You can have many minions running at one point in time. Each minion will host one or more POD.

- **POD:** are Mortal & is the smallest unit of deployment in K8S object mode or is like hosting a service. Each POD then contains the Docker containers(xyz). Each POD can host a different set of Docker containers. The proxy is then used to control the exposing of these services to the outside world. You cannot create your own PODs, they are created by replicaset.

etcd − This component is a highly available key-value store that is used for storing shared configuration and service discovery. Here the various applications will be able to connect to the services via the discovery service.
Flannel − This is a back-end network which is required for the containers.
kube-apiserver − This is an API which can be used to orchestrate the Docker containers.
kube-controller-manager − This is used to control the Kubernetes services.

kube-scheduler − This is used to schedule the containers on hosts.
Kubelet − This is used to control the launching of containers via manifest files from worker host. (which talks with K8S cluster).
kube-proxy − This is used to provide network proxy services to the outside world.

Kubernetes Terminology

1. **Nodes**:

   Hosts that run Kubernetes applications

2. **Containers**:

   Units of packaging

3. **Pods**:

   Units of deployment which is collection of containers

4. **Replication Controller**:

   Ensures availability and scalability

5. **Labels**:

   Key-value pairs for identification

6. **Services**:

   Collection of pods exposed as an endpoint

Kubernetes Pod definition

- A group of one or more containers is called a Pod.
- Containers in a Pod are deployed together, and are started, stopped, and replicated as a group.
- each Pod has only 1 IP, irrespective of number of containers.
- all container in a Pod shares IP, cgroups, namespaces, localhost adapter, volumes
- every pod can interact directly with other pod via Pod N/W (Inter-Pod communication)
- A Pod is the smallest and simplest Kubernetes object.
- It is the unit of deployment in Kubernetes, which represents a single instance of the application.
- A Pod is a logical collection of one or more containers, which:
- Pods Are scheduled together on the same host
- Pods Share the same network namespace
- Pods Mount the same external storage (Volumes)
- Pods life is short in nature, and they do not have the capability to self-heal by themselves. That is why we use them with controllers, which can handle a Pod's replication, fault tolerance, self-heal, etc.
- Examples of controllers are Deployments, Replica Sets, Replication Controllers, etc. We attach the Pod's specification to other objects using Pod Templates
- A Pod is a unit of deployment in Kubernetes that provides a set of abstractions and services for applications running on the Kubernetes cluster. It can include one or multiple containers (e.g., Docker, Rkt) that share storage and network resources, Linux cgroups and namespaces, are co-scheduled/colocated, and share the same life cycle.

  Pod manifest file:

```
$ vi hello-pod.yml
apiVersion: v1
kind: Pod
metadata:
  name: hello-pod
  labels:
    zones: prod
```

```
    version: v1
spec:
  containers:
  - name: hello-ctr
    image: nigelpoulton/pluralsight-docker-ci:latest
    ports:
    - containerPort: 8080
```

```
$ kubectl create -f hello-pod.yml
pod/hello-pod created
```

To access our hello-pod , we need to expose the pods though a service:

kubectl expose

When a pod is created, without a service, we cannot access to the app running within container in the pod. The most obvious way is to create a service for the pod either via Load Balancer or NodePort.

```
$ kubectl expose pod hello-pod --type=NodePort --target-port=80 -o yaml
```

```
$ kubectl describe pod hello-pod | grep -i ip
IP:               10.244.0.83
```

```
$ curl http://localhost:30779
http://10.244.0.83:30779/
```

```
$ kubectl get svc hello-pod -o wide
NAME     TYPE     CLUSTER-IP      EXTERNAL-IP   PORT(S)      AGE   SELECTOR

hello-pode NodePort 10.107.83.213 <none> 8080:30779/TCP 14m   version=v1,zones=prod
```

types of Kubernetes Health check:
Readiness probes and liveness probes

1. **Readiness=when Pod is ready to accept the traffic.(health)**
2. **Liveness=when to Restart the pod (ping)**

Replication Controller (**rc**)

- RC is a controller that is part of the Master Node's Controller Manager.
- It makes sure the specified number of replicas for a Pod is running at any given point in time.
- If there are more Pods than the desired count, the Replication Controller would kill the extra Pods, and, if there are less Pods, then the Replication Controller would create more Pods to match the desired count.
- Generally, we don't deploy a Pod independently, as it would not be able to re-start itself, if something goes wrong. We always use controllers like Replication Controller to create and manage Pods.

- A Replication Controller is a structure that enables you to easily create multiple pods, then make sure that that number of pods always exists. If a pod does crash, the Replication Controller replaces it.
- Replication Controllers also provide other benefits, such as the ability to scale the number of pods, and to update or delete multiple pods with a single command.

```
$ vi soaktestrc.yml
apiVersion: v1
kind: ReplicationController
metadata:
  name: soaktestrc
spec:
  replicas: 3
  selector:
    app: soaktestrc
  template:
    metadata:
      name: soaktestrc
      labels:
        app: soaktestrc
    spec:
      containers:
      - name: soaktestrc
        image: nickchase/soaktest
        ports:
        - containerPort: 80
```

```
# kubectl create -f soaktestrc.yml
replicationcontroller "soaktestrc" created
```

```
# kubectl get pods
NAME              READY     STATUS    RESTARTS   AGE
soaktestrc-cws05   1/1       Running   0          3m
soaktestrc-g5snq   1/1       Running   0          3m
soaktestrc-ro2bl   1/1       Running   0          3m
```

Let's "expose" the pods:

```
$ kubectl expose ReplicationController  new-nginx- --type=NodePort --target-port=80 -o ya
ml
```

```
$ kubectl describe pod soaktestrc | grep -i ip
IP:                10.244.0.83
```

```
$ curl http://localhost:30779
http://10.244.0.83:30779/
```

```
$ kubectl get svc soaktestrc -o wide
NAME     TYPE     CLUSTER-IP      EXTERNAL-IP   PORT(S)       AGE   SELECTOR

soaktestrc NodePort 10.107.83.213 <none> 8080:30779/TCP 14m   app:soaktestrc
```

Deployment lifecycle

There are three stages a deployment can be in its lifecycle.

- Progressing
- Complete
- Failed

K8S Deployments:

- The smallest unit of deployment, a *Pod*, runs containers. Each Pod has its own IP address and shares a PID namespace, network, and host name.
- You can define a *deployment* to create a Replica Set or to remove deployments and adopt all their resources with new deployments.
- When you revise a deployment, a Replica Set is created that describes the state that you want. During a rollout, the deployment controller changes the actual state to the state that you want at a controlled rate.
- Each deployment revision can also be rolled back. Deployments can also be scaled.

```
kubectl create deployment new-nginx --image=nginx:latest

[node1 ~]$ kubectl get deployment new-nginx
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
new-nginx  1         1         1            0           2m
```

Let's "expose" the pods:

```
$ kubectl expose deployment new-nginx- --type=NodePort --target-port=80 -o yaml
```

```
$ kubectl describe pod new-nginx | grep -i ip
IP:               10.244.0.83
```

```
$ curl http://localhost:30779
http://10.244.0.83:30779/
```

```
$ kubectl get svc soaktestrc -o wide
NAME    TYPE    CLUSTER-IP     EXTERNAL-IP   PORT(S)      AGE   SELECTOR

New-nginx NodePort 10.107.83.213 <none> 8080:30779/TCP 14m   app:nginx
```

- A **ReplicaSet** (rs) is the next-generation ReplicationController.
- Replica Sets support both equality- and set-based Selectors, whereas Replication Controllers only support equality-based Selectors.

services are REST objects in K8s, service stands in front of Pod so that outside world can interact to Pods via it. service never change mean its IP, DNS, Ports are reliable, unlike Pods which are unreliable in nature

now since pods are mortal and they come and go, so Its Endpoint which maintains the list of available pods dynamically.

Service use Labels to identify the Pods and do the things on them.

Our service definition looks like this:

```
apiVersion: v1

kind: Service

metadata:
  name: myapp-service

spec:
  type: NodePort
  ports:
```

```
  - targetPort: 80
    port: 80
    nodePort: 30050
  selector:
    app: myapp
    type: front-end
```

```
$ kubectl create -f service-definition.yml
service/myapp-service created
```

DaemonSets manage groups of replicated Pods
DaemonSets works on one-Pod-per-node model

Headless service

We can create **headless service** (here, my-service-4) which is **a service that does not get a ClusterIP** and is created when we specify a Service with **ClusterIP** set to **None**. This is often used when a deployer wants to decouple their service from Kubernetes and use an alternative method of service discovery and load balancing.

```
$ kubectl expose deployment nginx-deployment --cluster-ip=None --name=my-service-4
service/my-service-4 exposed

$ kubectl get svc
NAME         TYPE         CLUSTER-IP      EXTERNAL-IP   PORT(S)       AGE
kubernetes   ClusterIP    10.96.0.1       <none>        443/TCP       86d
my-service   NodePort     10.108.176.127  <none>        80:32373/TCP  2d
my-service-2 LoadBalancer 10.108.139.202  <pending>     80:30531/TCP  2d
my-service-3 ClusterIP    10.109.254.145  <none>        80/TCP        2d
my-service-4 ClusterIP    None            <none>        80/TCP        9s
```

We may choose to use the **headless service** combined with running the application as a StatefulSet.

```
##djangoservice.yaml
apiVersion: v1
kind: Service
metadata:
  name: djangoapp-server
  labels:
    app: djangoapp
spec:
  clusterIP: None
  ports:
  - port: 8080
    name: server
selector:
  app: djangoapp
```

```
$ kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
djangoapp-0   1/1   Running     0        31m
djangoapp-1   1/1   Running     0        31m
djangoapp-2   1/1   Running     0        31m
```

- In the above Scenario our djangoapp-server Service is deployed in the existing kubernetes cluster but you can see that clusterIP is specified as **None**, which specifies to the kube-proxy that Service is the **"Headless-Service"** .
- Now coming back to **"What is the use to create a Headless Service"?**
  - The main benefit of using a **headless service** is to be able to reach each Pod directly.
  - If this was a standard service, then the service would act as a **loadbalancer or proxy** and you would access your workload object just using the service name djangoapp-server.
  - In the above example,
    - With headless service, the Pod djangoapp-0 could use djangoapp-1.djangoapp-server to talk to djangoapp-1directly.
    - Syntax is: **<StatefulSet-Name>-<Ordinal-Index>.<ServiceName>**
  - To ensure stable network ID you need to define a headless service for **stateful applications.**

For the above example,

Standard service - you will get the clusterIP value:

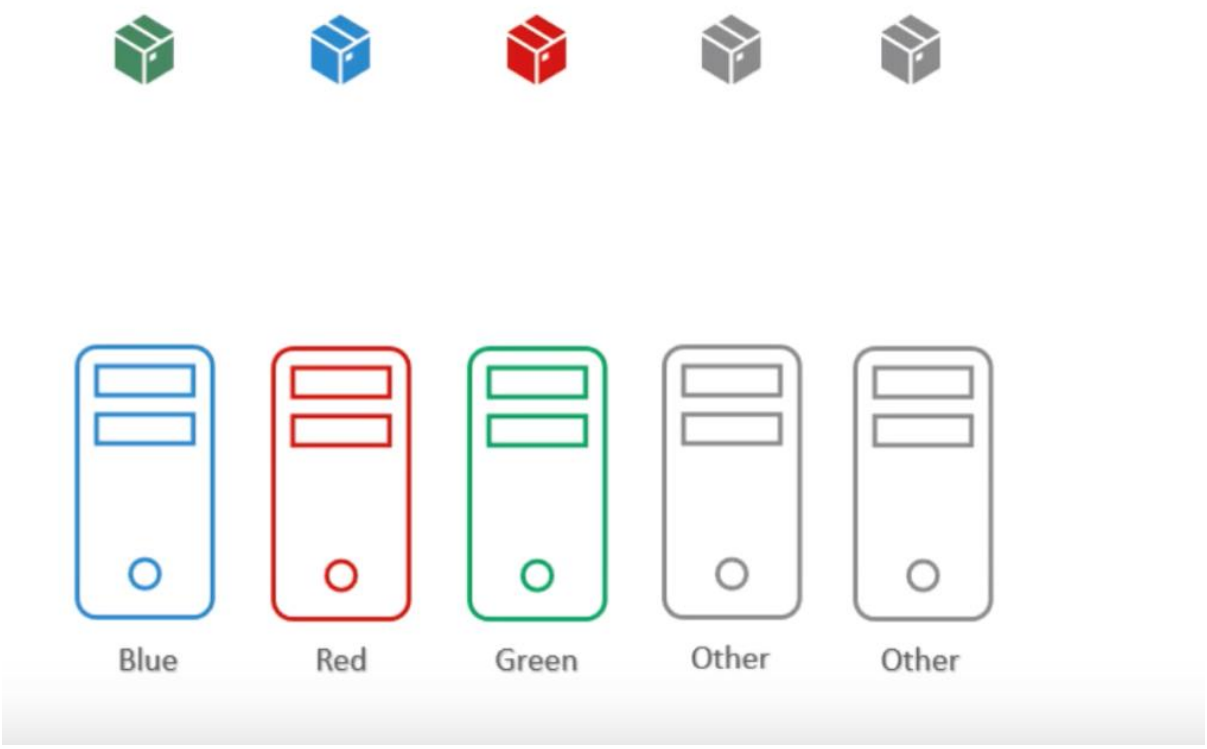
```
$ kubectl exec djangoapp-0 -- nslookup djangoapp
Server: 10.0.0.12
Address: 10.0.0.12#51

Name: djangoapp.default.svc.cluster.local
Address: 10.0.0.210
```

Headless service - you will get the IP of each Pod:

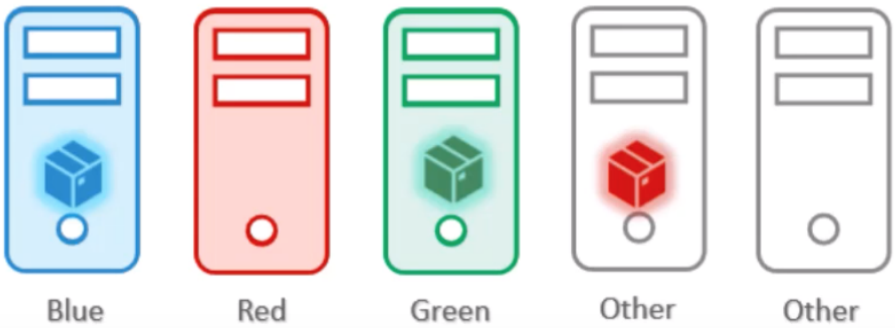
```
$ kubectl exec djangoapp-0 -- nslookup djangoapp
Server: 10.0.0.12
Address: 10.0.0.12#51

Name: djangoapp.default.svc.cluster.local
Address: 172.17.0.1
Name: djangoapp.default.svc.cluster.local
Address: 172.17.0.2
Name: djangoapp.default.svc.cluster.local
Address: 172.17.0.3
```

There are some scenarios where this is not feasible as the pods start one by one and only the latest pod could discover other pods.

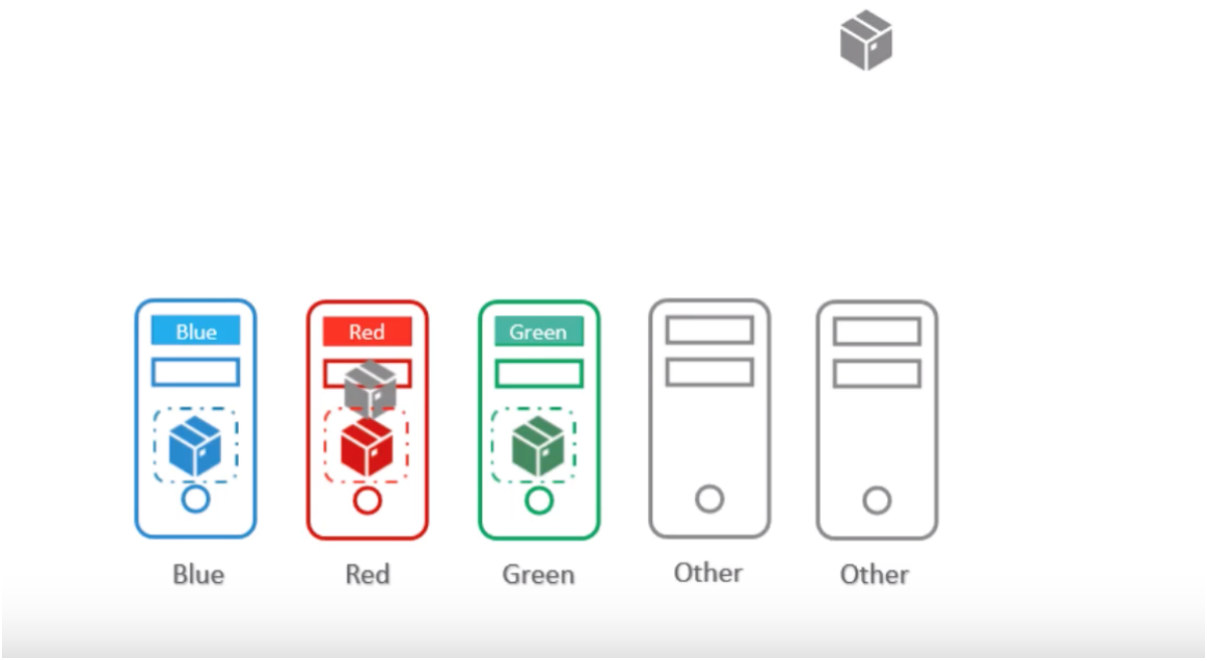
Taints and tolerations, pod and node affinities demystified



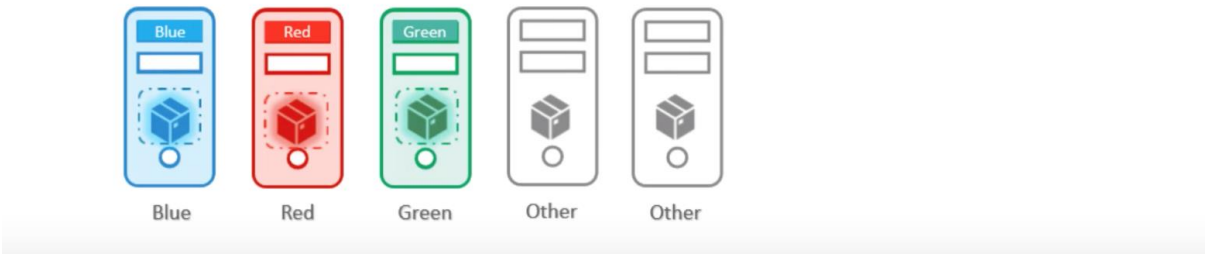
Taints and Tolerations



Node Affinity



Taints/Tolerations and Node Affinity



Enterprises often use multi-tenant and heterogenous clusters to deploy their applications to Kubernetes. These applications usually have needs which require special scheduling constraints. Pods may require nodes with special hardware, isolation, or colocation with other pods running in the system.

Taints and tolerations

- This Kubernetes feature allows users to mark a node (taint the node) so that no pods can be scheduled to it, unless a pod explicitly tolerates the taint.
- Using this Kubernetes feature we can create nodes that are reserved (dedicated) for specific pods. E.g. pods which require that most of the resources of the node be available to them in order to operate flawlessly should be scheduled to nodes that are reserved for them.
- In practice tainted nodes will be more like pseudo-reserved nodes, since taints and tolerations won't exclude undesired pods in certain circumstances:

- system pods are created with toleration settings that tolerate all taints thus can be scheduled onto any node. This is by design, as system pods are required by the Kubernetes infrastructure (e.g. kube-proxy) or by the Cloud Provider in case of managed Kubernetes (e.g. on EKS the aws-node system pod).
- users can't be stopped from deploying pods that tolerate "wrong" taint thus, beside system pods, pods other than desired ones may still run on the reserved nodes

The format of a taint is <key>=<value>:<effect>. The <effect> instructs the Kubernetes scheduler what should happen to pods that don't tolerate this taint. We can distinguish between two different effects:

- **NoSchedule** - instructs Kubernetes scheduler not to schedule any new pods to the node unless the pod tolerates the taint.
- **NoExecute** - instructs Kubernetes scheduler to evict pods already running on the node that don't tolerate the taint.

Taints in Kubernetes

- Taints *allow* a Kubernetes node to *repel* a set of pods.
- In other words, **if you want to deploy your pods everywhere except some specific nodes you just need to *taint* that node.**

Let's take a look at the kubeadm master node for example:

```
$ kubectl describe node master1 | grep -i taint
Taints:             node-role.kubernetes.io/master:NoSchedule
```

- As you can see, this node has a taint `node-role.kubernetes.io/master:NoSchedule`.
- The taint has the key `node-role.kubernetes.io/master`, value `nil` (which is not shown), and taint effect `NoSchedule`.
- So lets' talk about taint effects in more details.

Taint Effects

Each taint has one of the following effects:

- **NoSchedule** - this means that no pod will be able to schedule onto node unless it has a matching toleration.

- **PreferNoSchedule** - this is a “preference” or “soft” version of NoSchedule – the system will try to avoid placing a pod that does not tolerate the taint on the node, but it is not required.
- **NoExecute** - the pod will be evicted from the node (if it is already running on the node), and will not be scheduled onto the node (if it is not yet running on the node).

How to Taint the Node

Actually, it’s pretty easy - imagine, we have a node named `node1`, and we want to add a tainting effect to it:

```
$ kubectl taint nodes node1  key:=NoSchedule

$ kubectl describe node node1 | grep -i taint
Taints:             node-role.kubernetes.io/master:NoSchedule
```

How to Remove Taint from the Node

To remove the taint from the node run:

```
$ kubectl taint nodes  key:NoSchedule-
node "node1" untainted

$ kubectl describe node node1 | grep -i taint
Taints:             <none>
```

Tolerations

In order to schedule to the **“tainted” node** pod should have some special tolerations, let’s take a look on system pods in kubeadm, for example, etcd pod:

```
$ kubectl describe pod etcd-node1 -n kube-system | grep  -i toleration
Tolerations:       :NoExecute
```

As you can see it has toleration to `:NoExecute` taint, let’s see where this pod has been deployed:

```
$ kubectl get pod etcd-node1 -n kube-system -o wide
NAME                          READY     STATUS    RESTARTS   AGE       IP
NODE
etcd-node1                    1/1       Running   0          22h       192.168.1.212   node1
```

- The pod was indeed deployed on the “tainted” node because it “tolerates” `NoSchedule` effect.
- Now let’s have some practice: let’s create our own taints and try to deploy the pods on the “tainted” nodes with and without tolerations.

Practice

We have a four nodes cluster:

```
$ kubectl get nodes
NAME                 STATUS    ROLES     AGE       VERSION
master01             Ready     master    17d       v1.9.7
master02             Ready     master    20d       v1.9.7
node1                Ready     <none>    20d       v1.9.7
node2                Ready     <none>    20d       v1.9.7
```

- Imagine that you want **node1** to be available preferably for **testing POC pods**.
- This can be easily done with taints, so let's create one.

```
$ kubectl taint nodes node1 node-type=testing:NoSchedule
node "node1" tainted

$ kubectl describe node node1 | grep -i taint
Taints:             node-type=testing:NoSchedule
```

- So three nodes are tainted, because two masters in kubeadm cluster are tainted with `NoSchedule` effect by default and we just tainted the node1
- so, basically, all **pods without tolerations** should be deployed to node2.
- Let's test our theory:

```
$ kubectl run test --image alpine --replicas 3 -- sleep 999

kubectl get po -o wide
NAME                    READY     STATUS    RESTARTS   AGE       IP          NODE
test-5478d8b69f-2mhd7   1/1       Running   0          9s        10.47.0.9   node2
test-5478d8b69f-8lcgv   1/1       Running   0          9s        10.47.0.10  node2
test-5478d8b69f-r8q4m   1/1       Running   0          9s        10.47.0.11  node2
```

Indeed, all pods being scheduled to node2! Now let's add some tolerations to the next Kubernetes deployemnt, so we can schedule pods to node1.

```
$ cat <<EOF | kubectl create -f -
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: testing
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: testing
    spec:
      containers:
      - args:
        - sleep
        - "999"
        image: alpine
        name: main
      tolerations:
      - key: node-type
        operator: Equal
        value: testing
        effect: NoSchedule
EOF
deployment "testing" created
```

The most important part here is:

```
...
    tolerations:
    - key: node-type
      operator: Equal
      value: testing
      effect: NoSchedule
...
```

- As you can see, it tolerates the node with the key: `node-type`, operator: `=`, value: `testing` and effect `NoSchedule`.
- So the pods from this deployment can be schduled to our tainted node. `

```
$ kubectl get po -o wide
NAME                       READY    STATUS    RESTARTS   AGE       IP          NODE
test-5478d8b69f-2mhd7      1/1      Running   0          14m       10.47.0.9   node2
test-5478d8b69f-8lcgv      1/1      Running   0          14m       10.47.0.10  node2
test-5478d8b69f-r8q4m      1/1      Running   0          14m       10.47.0.11  node2
testing-788d87fd58-9j8lx   1/1      Running   0          10s       10.44.0.6   node2
testing-788d87fd58-ts5zt   1/1      Running   0          10s       10.44.0.3   node1
testing-788d87fd58-vzgd7   1/1      Running   0          10s       10.47.0.13  node1
```

- As you can see, two of three testing pods were deployed on node1, but why one of them was deployed on node1? It's because of the way how Scheduler in Kubernetes works - it wants to avoid a SPOF if it possible.
- You can easily prevent that from happening via adding another taint to a non-testing node, like `node-type=non-testing:NoSchedule`

Usecases

Taints and tolerations can help you to create the dedicated nodes only for some special set of pods (like in kubeadm master node example).

Similar to this you can restrict pods to run on some node with a special hardware.

Node affinity

- To get pods to be scheduled to specific nodes Kubernetes
- provides nodeSelectors and nodeAffinity. As nodeAffinity encompasses what can be achieved with nodeSelectors, nodeSelectors will be deprecated in Kubernetes thus we discuss nodeAffinity here.

With node affinity we can tell Kubernetes which nodes to schedule to a pod using the labels on each node.

LET'S SEE HOW NODE AFFINITY WORKS

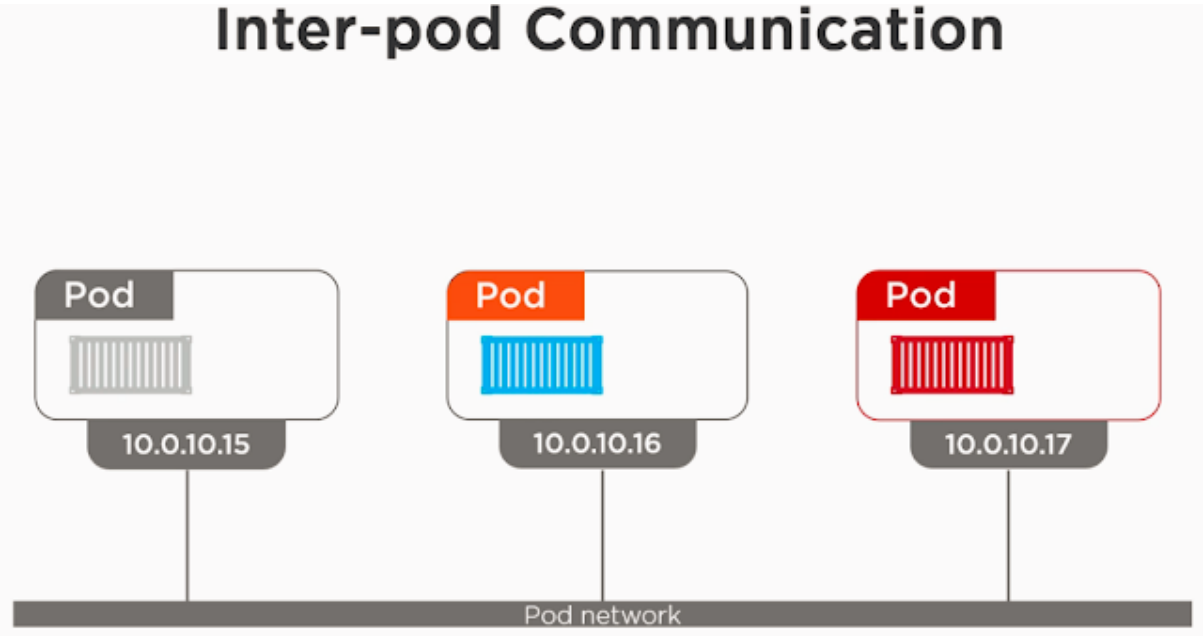
Since node affinity identifies the nodes on which to place pods via labels, we first need to add a label to our node.

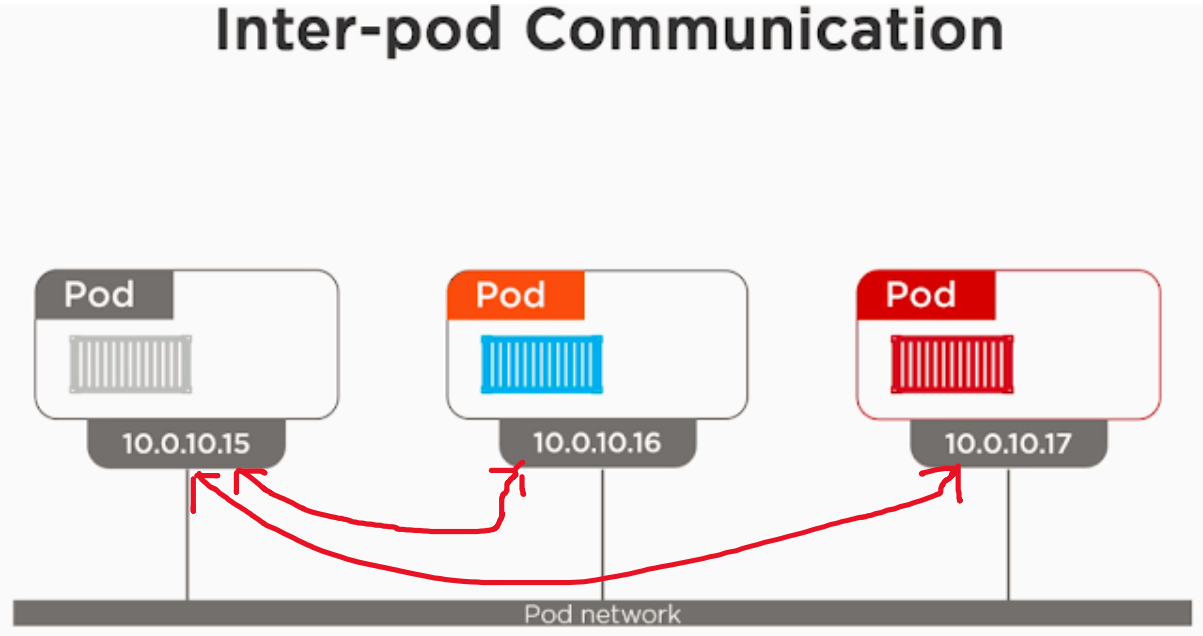
```
$ kubectl edit node node2

labels:
```

```
    ...
    test-node-affinity: test
...
```

Pod affinity and anti-affinity

- Pod affinity and anti-affinity allows placing pods to nodes as a function of the labels of other pods.
- These Kubernetes features are useful in scenarios like: an application that consists of multiple services, some of which may require that they be co-located on the same node for performance reasons; replicas of critical services shouldn't be placed onto the same node to avoid loss in the event of node failure.





kubectl create -f nginx-deployment.yaml

master $ kubectl get pods -o wide

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE

nginx-deployment-67594d6bf6-bxfqc 1/1 Running 0 26s 10.40.0.3 node01 <none>

nginx-deployment-67594d6bf6-dqcv8 1/1 Running 0 26s 10.40.0.4 node01 <none>

nginx-deployment-67594d6bf6-sbrtj 1/1 Running 0 26s 10.40.0.5 node01 <none>

communication between pod1 and pod2

master $ kubectl exec -it nginx-deployment-67594d6bf6-bxfqc bash

root@nginx-deployment-67594d6bf6-bxfqc:/# ping 10.40.0.4

PING 10.40.0.4 (10.40.0.4): 48 data bytes

56 bytes from 10.40.0.4: icmp_seq=0 ttl=64 time=0.151 ms

56 bytes from 10.40.0.4: icmp_seq=1 ttl=64 time=0.086 ms

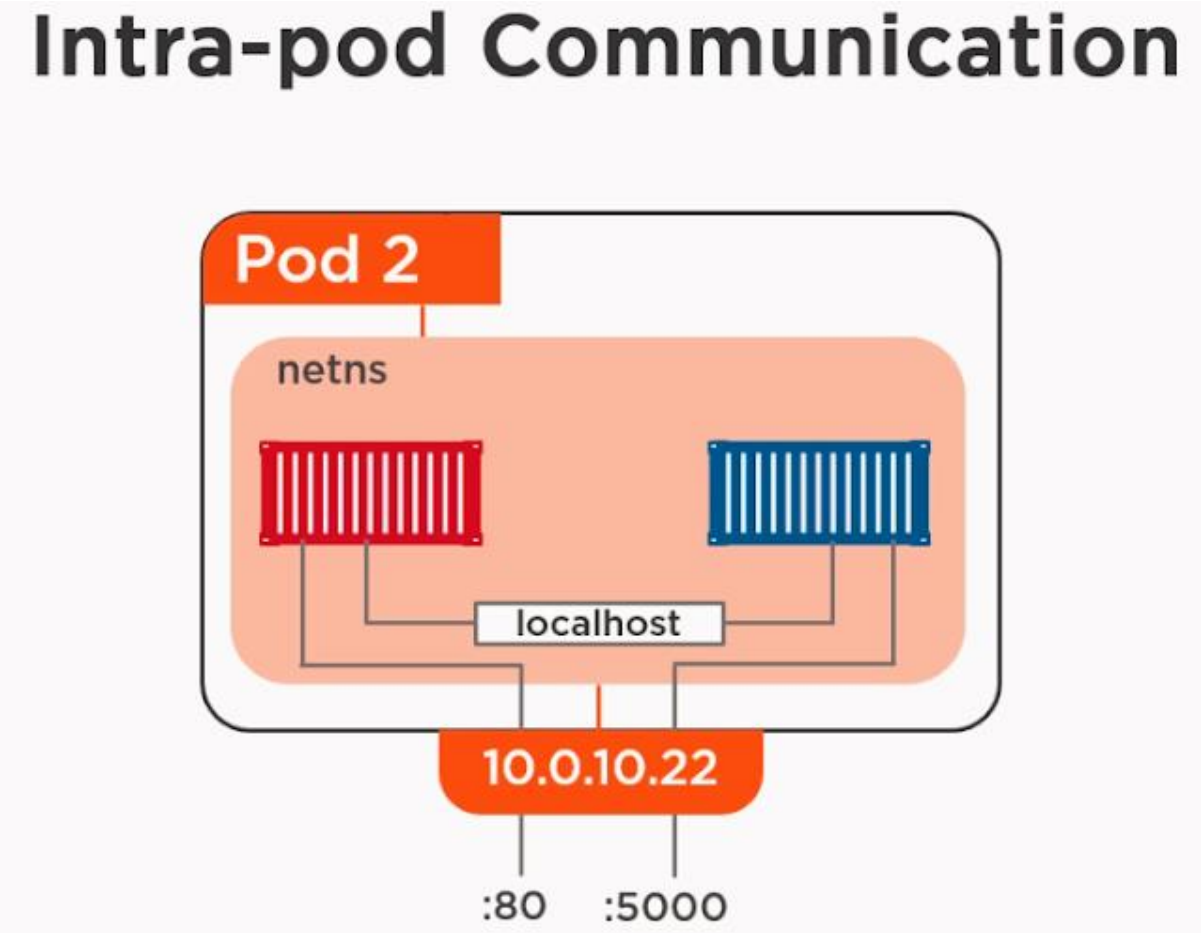
56 bytes from 10.40.0.4: icmp_seq=2 ttl=64 time=0.085 ms

communication between pod1 and pod3

master $ kubectl exec -it nginx-deployment-67594d6bf6-dqcv8 bash

root@nginx-deployment-67594d6bf6-dqcv8:/# ping 10.40.0.5

```
PING 10.40.0.5 (10.40.0.5): 48 data bytes
56 bytes from 10.40.0.5: icmp_seq=0 ttl=64 time=0.161 ms
56 bytes from 10.40.0.5: icmp_seq=1 ttl=64 time=0.082 ms
56 bytes from 10.40.0.5: icmp_seq=2 ttl=64 time=0.060 ms
56 bytes from 10.40.0.5: icmp_seq=3 ttl=64 time=0.084 ms
56 bytes from 10.40.0.5: icmp_seq=4 ttl=64 time=0.071 ms
^C--- 10.40.0.5 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.060/0.092/0.161/0.036 ms
```




```
vi productlib-pod.yaml

apiVersion: v1

kind: Pod

metadata:
```

```
  name: productlib-pod
spec:
  restartPolicy: Never
  containers:
    - name: app-container
      image: nimmis/apache-php7

    - name: cypress-container
      image: rabbitmq
```

```
kubectl apply -f productlib-pod.yaml

Wait until packages in Ubuntu are installed:
$ kubectl logs productlib-pod app-container
$ kubectl logs productlib-pod cypress-container

$ kubectl exec -ti productlib-pod -c app-container -- bash
root@productlib-pod:/# ping cypress-container
kubectl exec -ti productlib-pod -c cypress-container -- bash
root@productlib-pod:/# ping app-container
master $ kubectl get pod -o wide | grep productlib-pod
productlib-pod   2/2     Running   0          10m   10.32.0.4   master   <none>           <none>
master $ curl http://10.32.0.4
```

2. What is Kubernetes Load Balancing?

Load Balancing is one of the most common and the standard ways of exposing the services.

There are two types of load balancing in Kubernetes and they are:

1. Internal load balancer – This type of balancer automatically balances loads and allocates the pods with the required configuration.
2. External Load Balancer – This type of balancer directs the traffic from the external loads to backend pods.

3. What are the different types of services being provided by Kubernetes?

The followings are the different types of services being provided by the Kubernetes:

- Cluster IP
- Node Port
- Load Balancer
- External name

ClusterIP: This is the default service type which exposes the service on a cluster-internal IP by making the service only reachable within the cluster.

- A ClusterIP service is the **default Kubernetes service**. It gives you a service **inside** your cluster that other apps inside your cluster can access.
- **There is no external access.**

When would you use this?

There are a few scenarios where you would use the Kubernetes proxy to access your services.

1. Debugging your services, or connecting to them directly from your laptop for some reason
2. Allowing internal traffic, displaying internal dashboards, etc.

Because this method requires you to run kubectl as an authenticated user, you should NOT use this to expose your service to the internet or use it for production services.

NodePort: This exposes the service on each Node's IP at a static port. Since, a **ClusterIP** service, to which the NodePort service will route, is automatically created. We can contact the NodePort service outside the cluster.

- A NodePort service is the most primitive way to get external traffic directly to your service.

NodePort, as the name implies, opens a specific port on all the Nodes (the VMs), and any traffic that is sent to this port is forwarded to the service

When would you use this?

There are many downsides to this method:

1. You can only have once service per port
2. You can only use ports 30,000–32,767
3. If your Node/VM IP address change, you need to deal with that

For these reasons, I don't recommend using this method in production to directly expose your service. If you are running a service that doesn't have to be always available, or you are very cost sensitive, this method will work for you. A good example of such an application is a demo app or something temporary.

ExternalName: This service type maps the service to the contents of the **externalName** field by returning a **CNAME** record with its value.

So, guys that was all about services. Now, you might be wondering how do external services connect to these networks right?

Well, that's by none other than **Ingress Network**.

Ingress Network

Unlike all the above examples, Ingress is actually NOT a type of service. Instead, it sits in front of multiple services and act as a "smart router" or entry point into your cluster.

You can do a lot of different things with an Ingress, and there are many types of Ingress controllers that have different capabilities.

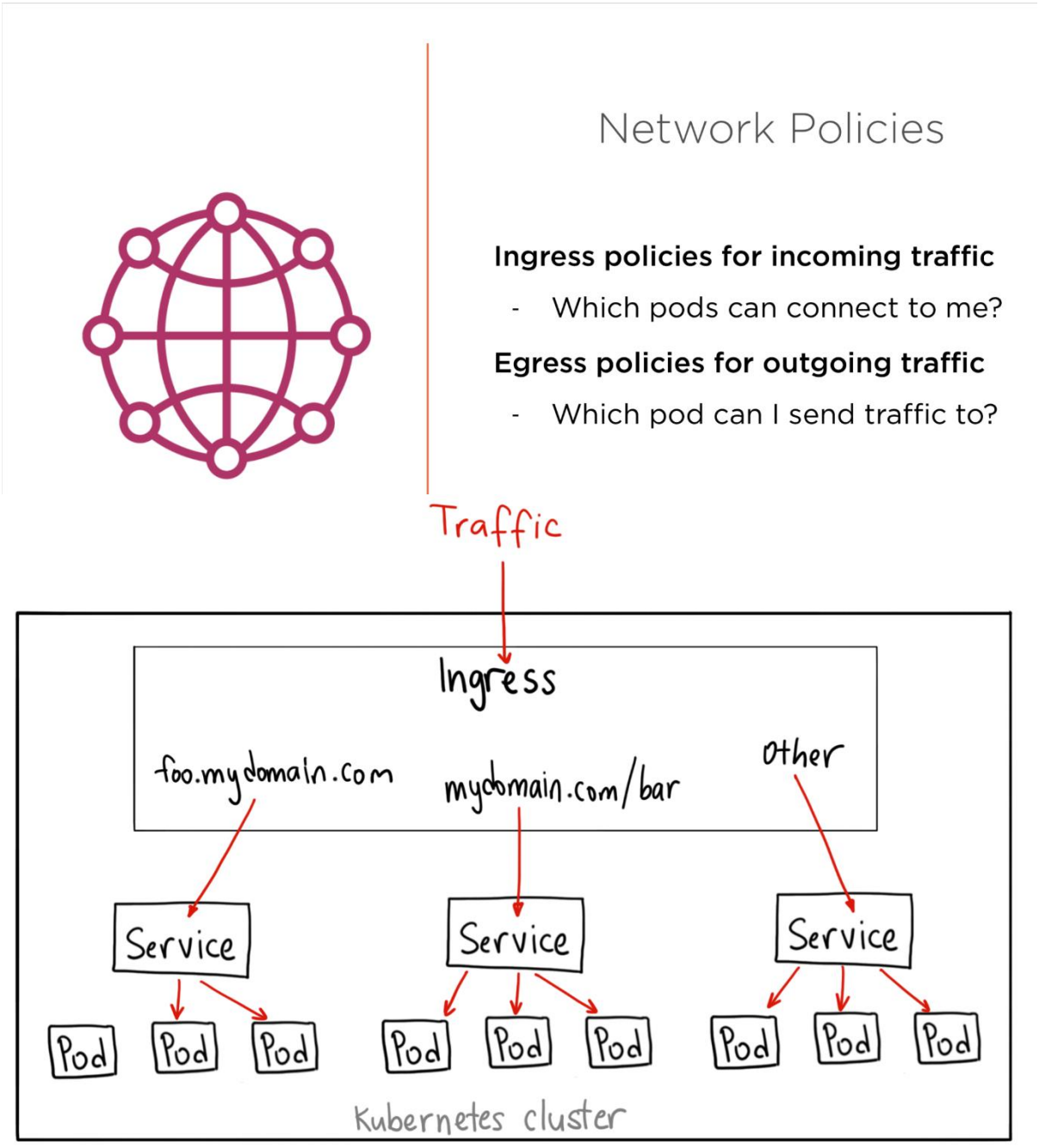
The default GKE ingress controller will spin up a HTTP(S) Load Balancer for you. This will let you do both path based and subdomain based routing to backend services. For example, you can send everything on foo.yourdomain.com to the foo service, and everything under the yourdomain.com/bar/ path to the bar service.

When would you use this?

- Ingress is probably the most powerful way to expose your services, but can also be the most complicated.
- There are many types of Ingress controllers, from the Google Cloud Load Balancer, Nginx, Contour, Istio, and more.

- There are also plugins for Ingress controllers, like the [cert-manager](), that can automatically provision SSL certificates for your services.
- Ingress is the most useful if you want to expose multiple services under the same IP address, and these services all use the same L7 protocol (typically HTTP).
- You only pay for one load balancer if you are using the native GCP integration, and because Ingress is “smart” you can get a lot of features out of the box (like SSL, Auth, Routing, etc)

Ingress Network




```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
```

```
  name: my-ingress
spec:
  backend:
    serviceName: other
    servicePort: 8080
  rules:
  - host: foo.mydomain.com
    http:
      paths:
      - backend:
          serviceName: foo
          servicePort: 8080
  - host: mydomain.com
    http:
      paths:
      - path: /bar/*
        backend:
          serviceName: bar
          servicePort: 8080
```

Unlike all the above examples, Ingress is actually NOT a type of service. Instead, it sits in front of multiple services and act as a “smart router” or entry point into your cluster.

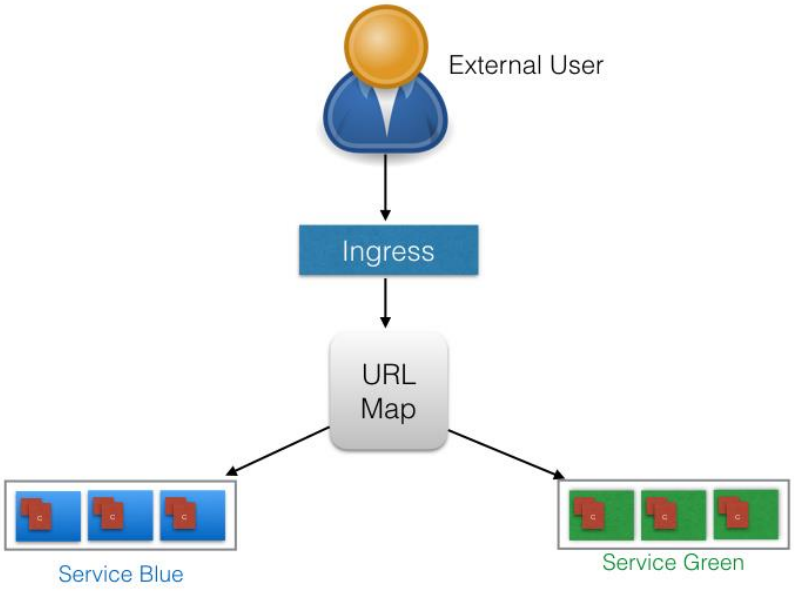
You can do a lot of different things with an Ingress, and there are many types of Ingress controllers that have different capabilities.

The default GKE ingress controller will spin up a HTTP(S) Load Balancer for you. This will let you do both path based and subdomain based routing to backend services. For example, you can send everything on foo.yourdomain.com to the foo service, and everything under the yourdomain.com/bar/ path to the bar service.

When would you use this?

- Ingress is probably the most powerful way to expose your services, but can also be the most complicated.
- There are many types of Ingress controllers, from the Google Cloud Load Balancer, Nginx, Contour, Istio, and more.
- There are also plugins for Ingress controllers, like the cert-manager, that can automatically provision SSL certificates for your services.
- Ingress is the most useful if you want to expose multiple services under the same IP address, and these services all use the same L7 protocol (typically HTTP).
- You only pay for one load balancer if you are using the native GCP integration, and because Ingress is “smart” you can get a lot of features out of the box (like SSL, Auth, Routing, etc)

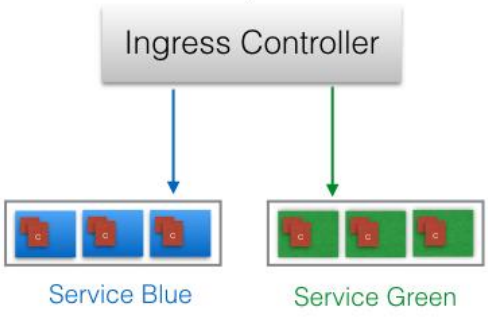
Ingress



Ingress URL Mapping

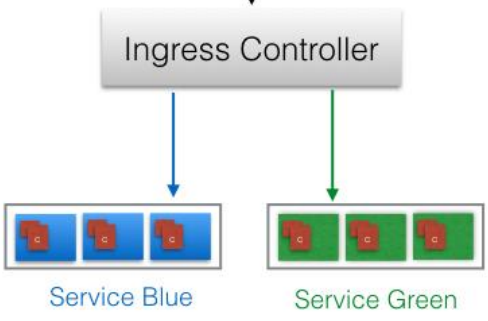
Fan Out

myweb.com/blue
myweb.com/green



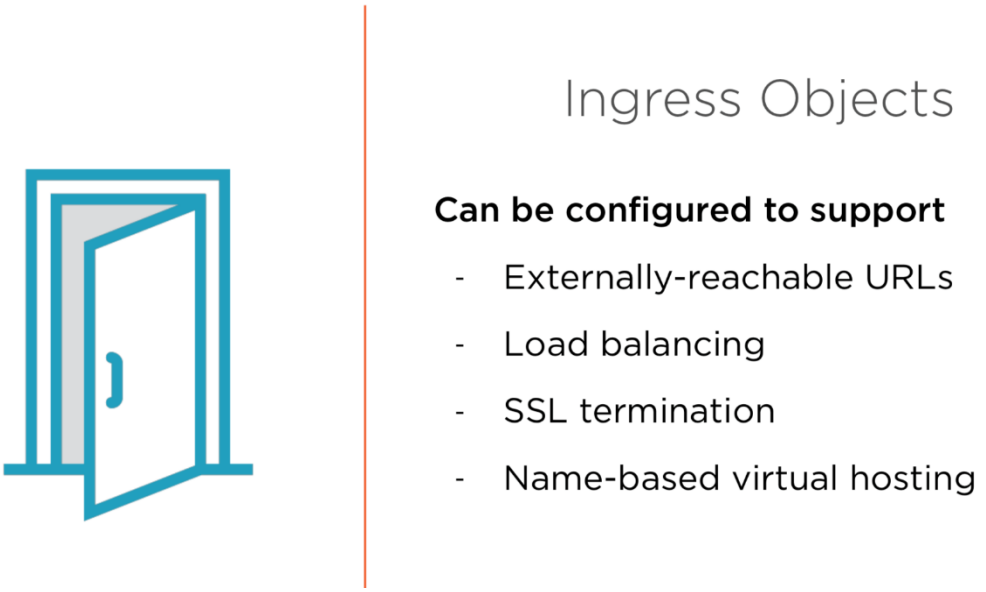
Virtual Hosting

blue.myweb.com
green.myweb.com



Ingress is split into two main parts – Ingress resources and ingress controller

1. **Ingress Resources**
   Ingress Resources defines how you want the requests to the services to be routed. It contains the main routing rules.



2. **Ingress controller**
   what ingress controller does is, it reads the ingress resource's information and process the data accordingly. So basically, ingress resources contain the rules to route the traffic and ingress controller routes the traffic.Traffic routing is done by an ingress controller.

   There are many types of Ingress controllers, from the Google Cloud Load Balancer, Nginx, Contour, Istio, and more.

4. What do you mean by Kubelet?

Kubelet is a type of primary node agents that especially runs on each node. Kubelet only works on the descriptions that the containers provide to the Podspec. Kubelet also makes sure that the container described in Podspec is healthy and running.

5. What are the uses of Google Kubernetes Engine?

The followings are the uses of the Google Kubernetes Engine:

- Create or resize Docker container clusters
- Creates container pods, replication controller, jobs, services or load balancer
- Resize application controllers
- Update and upgrade container clusters
- Debug container clusters.

6. What is GKE in Kubernetes?

Firstly GKE stands for Google Kubernetes Engine. GKE is a management and an orchestration system that is used for Docker container and all the container clusters that basically run within the Google's public cloud services. Google Kubernetes engine is based on Kubernetes.

7. What are the features of Minikube?

The followings are the main features of the Minikube:

- DNS
- Nodeports
- Configure maps and secrets
- Dashboards
- Enabling CNI
- Ingress
- Container runtime: Docker, rkt, CRI – O and containerd

8. What is minikube?

Minikube is a type of tool that makes the Kubernetes easy to run locally. Minikube basically runs on the single nodes Kubernetes cluster that is inside the virtual machine on your laptop. This is also used by the developers who are trying to develop by using Kubernetes day to day.

9. What is heapster in Kubernetes?

Heapster is a type of cluster-wide aggregator that helps in the process of monitoring and event data. Heapster helps to enable the container cluster monitoring and performance analysis for Kubernetes.

10. What are the initial namespaces from which the Kubernetes starts?

The followings are the three initial namespaces from which the Kubernetes starts:

- Default
- Kube – system
- Kube – public

11. What are namespaces in Kubernetes?

Kubernetes is especially intended for the use of the environments with many other users that are being spread across multiple teams or projects. Namespaces are the way to divide the cluster resources between the multiple users.

12. What are pods in Kubernetes?

A Kubernetes pod is a group of containers that are being deployed in the same host. Pods have the capacity to operate one level higher than the individual containers. This is because pods have the group of containers that work together to produce an artefact or to process a set of work.

13. What does the nodes status contains?

The followings are the main components that the node status:

- Address
- Condition
- Capacity
- Info

14. What are nodes in kubernetes?

A node is a type of work machine in Kubernetes that was previously known as a minion. A node can be a type of virtual machine or the physical machine. It always depends upon the clusters. Each of the nodes provides the services that are necessary to run pods, and it is also managed by the master components.

15. What do you understand by Kubernetes?

Kubernetes is basically a type of an open – source container. Kubernetes has the potential to hold the container deployment, scaling and descaling of the container and load balancing. Kubernetes was being developed in the year of 2014. It is also used to manage the Linux containers across the privates, hybrid and cloud environments.

16. What are the difference between Kubernetes and Docker Swarm?

The followings are the main difference between Kubernetes and docker and they are:

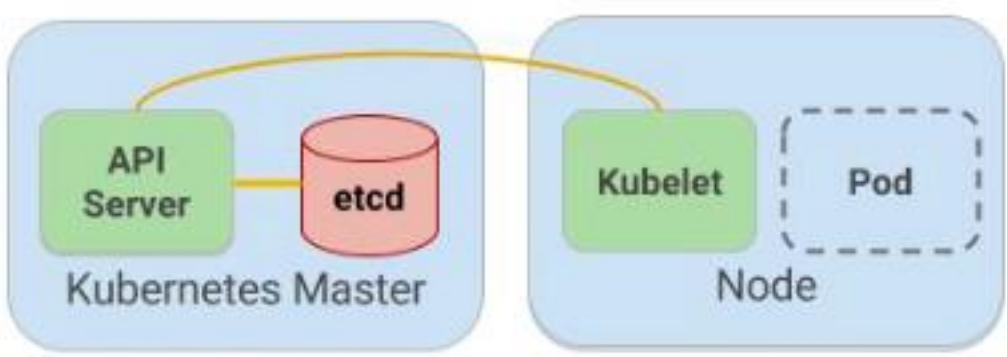
- The installation structure of the Kubernetes is very complicated but if it is once installed then the cluster is robust. On the other hand, the Docker swarm installation process is very simple but the cluster is not at all robust.
- Kubernetes can do the process of the auto scaling but the Docker swarm cannot do the process of the auto scaling.
- Kubernetes is highly scalable and also scales fast. But the Docker swarm scales are 5x faster than Kubernetes and is also highly scalable.

KUBERNETES - SECRETS

Secrets

Secrets are secure objects which store sensitive data, such as passwords, tokens, and ssh keys.

Storing sensitive data in Secrets is more secure than plaintext ConfigMaps or in Pod specifications.



Using Secrets gives us control over how sensitive data is used, and reduces the risk of exposing the data to unauthorized users.

Kubernetes keeps those info within a single central place (etcd), which may cause security issues as described Securing Kubernetes secrets : How to efficiently secure access to etcd and protect your secrets.

Creating a Secret using Kubectl

Suppose that some pods need to access a database. The username and password that the pods should use is in the files **username.txt** and **password.txt** on our local machine:

```
# Create files needed for rest of example.
$ echo -n 'admin' > ./username.txt
$ echo -n '1f2d1e2e67df' > ./password.txt
```

The kubectl create secret command packages these files into a Secret and creates the object on the Apiserver:

```
$ kubectl create secret generic db-user-pass \
--from-file=./username.txt --from-file=./password.txt
secret/db-user-pass created
```

We can check that the secret was created like this:

```
$ kubectl get secrets
NAME                  TYPE                  DATA   AGE
db-user-pass          Opaque                2      93s
```

Creating a Secret Manually

We can also create a Secret in a file first, in json or yaml format, and then create that object. The Secret contains two maps: **data** and **stringData**.

The **data** field is used to store arbitrary data, encoded using **base64**. The **stringData** field is provided for convenience, and allows us to provide secret data as unencoded strings.

For example, to store two strings in a **Secret** using the **data** field, convert them to base64 as follows:

```
$ echo -n 'admin' | base64
YWRtaW4=

$ echo -n '1f2d1e2e67df' | base64
MWYyZDFlMmU2N2Rm
```

Write a Secret (**mysecret.yaml**) that looks like this:

```
apiVersion: v1
```

```
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

Using Secrets as Files (Volumes) from a Pod

This is an example of a pod (**mysecret-pod.yaml**) that mounts a secret in a volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
```

Each secret we want to use needs to be referred to in **.spec.volumes**.

Create the pod and check if our volume for the secret is properly mounted:

```
$ kubectl create -f mysecret-pod.yaml
pod/mypod created

$ kubectl get po
NAME                READY   STATUS    RESTARTS   AGE
mypod               1/1     Running   0          26s

$ kubectl exec mypod ls /etc/foo
password
username

$ kubectl exec mypod cat /etc/foo/username
admin
$ kubectl exec mypod cat /etc/foo/password
1f2d1e2e67df
```

We can see inside the container that mounts a secret volume, the secret keys appear as files and the secret values are base-64 decoded and stored inside these files. The program in a container is responsible for reading the secrets from the files.

Using Secrets as Environment Variables

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
```

```
    - name: SECRET_USERNAME
      valueFrom:
        secretKeyRef:
          name: mysecret
          key: username
    - name: SECRET_PASSWORD
      valueFrom:
        secretKeyRef:
          name: mysecret
          key: password
  restartPolicy: Never
```

Create a pod and verify:

```
$ kubectl create -f secret-env-pod.yaml
pod/secret-env-pod created

$ kubectl get po
NAME          READY   STATUS    RESTARTS   AGE
secret-env-pod   1/1     Running   0          27s

$ kubectl exec secret-env-pod env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=secret-env-pod
SECRET_USERNAME=admin
SECRET_PASSWORD=1f2d1e2e67df
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
```

...

Labels

Labels are key-value pairs which are attached to pods, replication controller and services. They are used as identifying attributes for objects such as pods and replication controller. They can be added to an object at creation time and can be added or modified at the run time.

With **Label Selectors,** we can select a subset of objects. Kubernetes supports two types of Selectors:

- **Equality-Based Selectors**
  Equality-Based Selectors allow filtering of objects based on label keys and values. With this type of Selectors, we can use the =, ==, or **!=**operators. For example, with **env==dev** we are selecting the objects where the **env** label is set to **dev**.

Equality Based Requirement
Equality based requirement will match for the specified label and filter the resources. The supported operators are =, ==, != .

Let us say I have following pods with the labels.

```
adpspl-mac36:~ vbirada$ kubectl get pod --show-labels
NAME                READY     STATUS    RESTARTS   AGE       LABELS
example-pod         1/1       Running   0          17h
env=prod,owner=Ashutosh,status=online,tier=backend
example-pod1        1/1       Running   0          21m
env=prod,owner=Shovan,status=offline,tier=frontend
example-pod2        1/1       Running   0          8m
env=dev,owner=Abhishek,status=online,tier=backend
example-pod3        1/1       Running   0          7m
env=dev,owner=Abhishek,status=online,tier=frontend
```

Now, I want to see all the pods with online status:

```
adpspl-mac36:~ vbirada$ kubectl get pods  -l status=online
NAME          READY     STATUS    RESTARTS   AGE
example-pod   1/1       Running   0          17h
example-pod2  1/1       Running   0          9m
example-pod3  1/1       Running   0          9m
```

Similarly, go through following commands

```
adpspl-mac36:~ vbirada$ kubectl get pods  -l status!=online
NAME                READY    STATUS    RESTARTS   AGE
example-pod1          1/1      Running   0          25m
example-pod4          1/1      Running   0          11m
adpspl-mac36:~ vbirada$ kubectl get pods  -l status==offline
NAME          READY    STATUS    RESTARTS   AGE
example-pod1   1/1      Running   0          26m
example-pod4   1/1      Running   0          11m
adpspl-mac36:~ vbirada$ kubectl get pods  -l status==offline,status=online
No resources found.
adpspl-mac36:~ vbirada$  kubectl get pods  -l status==offline,env=prod
NAME          READY    STATUS    RESTARTS   AGE
example-pod1   1/1      Running   0          28m
adpspl-mac36:~ vbirada$ kubectl get pods  -l owner=Abhishek
NAME          READY    STATUS    RESTARTS   AGE
example-pod2   1/1      Running   0          15m
example-pod3   1/1      Running   0          14m
```

In above commands, labels separated by comma is a kind of **AND** satisfy operation.
Similarly, you can try other combination using the operators (= , !=, ==)and play!

- **Set-Based Selectors**

  Set-Based Selectors allow filtering of objects based on a set of values. With this type of Selectors, we can use the **in**, **notin**, and **exist** operators. For example, with **env in (dev,qa)**, we are selecting objects where the **env** label is set to **dev** or **qa**.

  Set Based Requirement

  Label selectors also support set based requirements. In other words, label selectors can be use to specify a set of resources.
  The supported operators here are in , notin and exists .

  Let us walk through kubectl commands for filtering resources using set based requirements.

```
adpspl-mac36:~ vbirada$ kubectl get pod -l 'env in (prod)'
NAME          READY    STATUS    RESTARTS   AGE
example-pod    1/1      Running   0          18h
example-pod1   1/1      Running   0          41m
adpspl-mac36:~ vbirada$  kubectl get pod -l 'env in (prod,dev)'
NAME          READY    STATUS    RESTARTS   AGE
```

```
example-pod    1/1     Running   0          18h
example-pod1   1/1     Running   0          41m
example-pod2   1/1     Running   0          27m
example-pod3   1/1     Running   0          27m
```

Here env in (prod,dev) the comma operator acts as a **OR** operator. That is it will list pods which are in prod or dev.

```
adpspl-mac36:~ vbirada$ kubectl get pod -l 'env in (prod),tier in (backend)'
NAME          READY     STATUS    RESTARTS   AGE
example-pod   1/1       Running   0          18h
adpspl-mac36:~ vbirada$  kubectl get pod -l 'env in (qa),tier in (frontend)'
No resources found.
```

Here the comma operator separating env in (qa) and tier in (frontend)will act as an AND operator.

To understand the exists operator let us add label region=central to example-pod and example-pod1 and region=northern to example-pod2 .

```
adpspl-mac36:~ vbirada$ kubectl get  pod --show-labels
NAME                READY     STATUS    RESTARTS   AGE       LABELS
example-pod         1/1       Running   0          18h
env=prod,owner=Ashutosh,region=central,status=online,tier=backend
example-pod1        1/1       Running   0          54m
env=prod,owner=Shovan,region=central,status=offline,tier=frontend
example-pod2        1/1       Running   0          40m
env=dev,owner=Abhishek,region=northern,status=online,tier=backend
example-pod3        1/1       Running   0          40m
env=dev,owner=Abhishek,status=online,tier=frontend
example-pod4        1/1       Running   0          40m
env=qa,owner=Atul,status=offline,tier=backend
```

Now, I want to view pods that is not in central region:

```
adpspl-mac36:~ vbirada$  kubectl get pods -l 'region notin (central)'
NAME                READY     STATUS    RESTARTS   AGE
example-pod2        1/1       Running   0          42m
example-pod3        1/1       Running   0          42m
example-pod4        1/1       Running   0          41m
```

You can realise here that example-pod2 is having a region key with value northern and hence appears in result. But one point to note is that other two pods in result is not having any region field and will satisfy the condition to appear in result.

If we want that pods having region key should only be the set of resources over which filtering should be done we can restrict via the existsoperator.We do not specifically write exists like we do write in and notinin command.

```
adpspl-mac36:~ vbirada$ kubectl get pods -l 'region,region notin (central)'
NAME          READY   STATUS    RESTARTS   AGE
example-pod2  1/1     Running   0          46m
```

Similarly, you can play by using various combinations in set based requirements too for selecting a set of pods.

Kubernetes – Persistent Volumes

Volumes – The Theory

In the Kubernetes world, persistent storage is broken down into two kinds of objects. A Persistent Volume (PV) and a Persistent Volume Claim (PVC). First, lets tackle a Persistent Volume.

Persistent Volumes

Persistent Volumes are simply a piece of storage in your cluster. Similar to how you have a disk resource in a server, a persistent volume provides storage resources for objects in the cluster. At the most simple terms you can think of a PV as a disk drive. It should be noted that this storage resource exists independently from any pods that may consume it. Meaning, that if the pod dies, the storage should remain intact assuming the claim policies are correct. Persistent Volumes are provisioned in two ways, Statically or Dynamically.

Static Volumes – A static PV simply means that some k8s administrator provisioned a persistent volume in the cluster and it's ready to be consumed by other resources.

Dynamic Volumes – In some circumstances a pod could require a persistent volume that doesn't exist. In those cases it is possible to have k8s provision the volume as needed if storage classes were configured to demonstrate where the dynamic PVs should be built. This post will focus on static volumes for now.

Persistent Volume Claims

Pods that need access to persistent storage, obtain that access through the use of a Persistent Volume Claim. A PVC, binds a persistent volume to a pod that requested it.

When a pod wants access to a persistent disk, it will request access to the claim which will specify the size , access mode and/or storage classes that it will need from a Persistent Volume. Indirectly the pods get access to the PV, but only through the use of a PVC.

Claim Policies

We also reference claim policies earlier. A Persistent Volume can have several different claim policies associated with it including:

Retain – When the claim is deleted, the volume remains.

Recycle – When the claim is deleted the volume remains but in a state where the data can be manually recovered.

Delete – The persistent volume is deleted when the claim is deleted.

The claim policy (associated at the PV and not the PVC) is responsible for what happens to the data on when the claim has been deleted.

How do you package Kubernetes applications?

Helm is a package manager which allows users to package, configure, and deploy applications and services to the Kubernetes cluster.

helm init # when you execute this command client is going to create a deployment in the cluster and that deployment will install the tiller, the server side of Helm

The packages we install through client are called charts. They are bundles of templatized manifests. All the templating work is done by the Tiller

helm search redis ***# searches for a specific application***

helm install stable/redis ***# installs the application***

helm ls ***# list the applications***

What is the difference between config map and secret? (Differentiate the answers as with examples)

Config maps ideally stores application configuration in a plain text format whereas Secrets store sensitive data like password in an encrypted format. Both config maps and secrets can be used as volume and mounted inside a pod through a pod definition file.

Config map:

kubectl create configmap myconfigmap

--from-literal=env=dev

Secret:

echo -n ‘admin’ > ./username.txt

echo -n ‘abcd1234’ ./password.txt

kubectl create secret generic mysecret --from-file=./username.txt --from-file=./password.txt

If a node is tainted, is there a way to still schedule the pods to that node?

When a node is tainted, the pods don't get scheduled by default, however, if we have to still schedule a pod to a tainted node we can start applying tolerations to the pod spec.

Apply a taint to a node:

kubectl taint nodes node1 key=value:NoSchedule

Apply toleration to a pod:

spec:

tolerations:

- key: "key"

operator: "Equal"

value: "value"

effect: "NoSchedule"

How to monitor that a Pod is always running?

We can introduce probes. A liveness probe with a Pod is ideal in this scenario.

A liveness probe always checks if an application in a pod is running, if this check fails the container gets restarted. This is ideal in many scenarios where the container is running but somehow the application inside a container crashes.

```
spec:
containers:
- name: liveness
image: k8s.gcr.io/liveness
args:
- /server
livenessProbe:
    httpGet:
      path: /healthz
```

Having a Pod with two containers, can I ping each other? like using the container name?

Containers on same pod act as if they are on the same machine. You can ping them using localhost:port itself. Every container in a pod shares the same IP. You can `ping localhost` inside a pod. Two containers in the same pod share an IP and a network namespace and They are both localhost to each other. Discovery works like this: Component A's pods -> Service Of Component B -> Component B's pods and Services have domain names servicename.namespace.svc.cluster.local, the dns search path of pods by default includes that stuff, so a pod in namespace Foo can find a Service bar in same namespace Foo by connecting to `bar`

28. How does Kubernetes provide high availability of applications in a Cluster?

In a Kubernetes cluster, there is a Deployment Controller. This controller monitors the instances created by Kubernetes in a cluster. Once a node or the machine hosting the node goes down, Deployment Controller will replace the node.

It is a self-healing mechanism in Kubernetes to provide high availability of applications.

Therefore in Kubernetes cluster, Kubernetes Deployment Controller is responsible for starting the instances as well as replacing the instances in case of a failure.

26. What is the use of Kubernetes?

We use Kubernetes for automation of large-scale deployment of Containerized applications.

It is an open source system based on concepts similar to Google’s deployment process of millions of containers.

It can be used on cloud, on-premise datacenter and hybrid infrastructure.

In Kubernetes we can create a cluster of servers that are connected to work as a single unit. We can deploy a containerized application to all the servers in a cluster without specifying the machine name.

We have to package applications in such a way that they do not depend on a specific host.

Ingress

Using Kubernetes Ingress:
Kubernetes ingress is not a service but a collection of routing rules that govern how the external users access services running on the Kubernetes cluster. Ingress sits in front of the

cluster and acts as a smart router. This is always implemented using a third party called a proxy. Traffic routing is done by an ingress controller.

what brings us to the needs of Kubernetes Ingress.

Why Kubernetes Ingress?
When you have an application deployed on your cluster, you obviously want to expose it to the internet to get inbound traffic otherwise what is the deployment for, right? Kubernetes Ingress is a built-in load balancing framework for routing external traffic.

There are two main reasons why we use Kubernetes Ingress:

1. When running the cluster and deployments on a cloud platform like AWS or GKE, the load balancing feature is available out of the box and there's no need to define ingress rules. But again, using external load balancers means spending more money and especially when your deployment is a small-scale deployment and you have a tight budget, you might as well use Kubernetes Ingress which is absolutely free and economical.
2. Ingress is an abstraction of layer 7 load balancing and not layer 4. When we talk about layer 7, we refer the layer 7 of the OSI model which is the application layer. It routes the traffic using the actual content of the message. When the load balancer has access to the content of the message, it can obviously make smarter routing decisions compared to counter load balancing techniques which use Layer 4(transport layer).

What is Kubernetes Ingress?
By now you would have understood that Kubernetes Ingress is a collection of routing rules that govern how external users access services running on the Kubernetes cluster.

Functionalities provided by Kubernetes Ingress:

1. Layer 7 load balancing – As discussed above
2. SSL Termination – Secure your connection with SSL termination
3. Name-based virtual hosting – Defines routes to services based on incoming request's hostnames. This allows you to run multiple services on the same IP address.

Let's get further into the details. Ingress is split into two main parts – Ingress resources and ingress controller

1. **Ingress Resources**
   Ingress Resources defines how you want the requests to the services to be routed. It contains the main routing rules.
2. **Ingress controller**
   What ingress controller does is, it reads the ingress resource's information and process the data accordingly. So basically, ingress resources contain the rules to route the traffic and ingress controller routes the traffic.

Routing using ingress is not standardized i.e. different ingress controller have different semantics (different ways of routing).

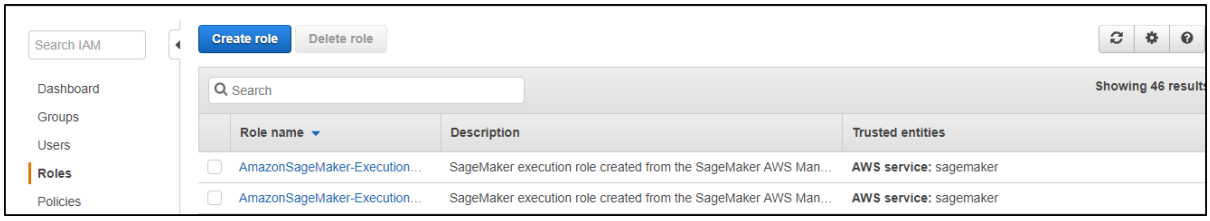
At the end of the day, you need to build your own ingress controller based on your requirements and implementations. Ingress is the most flexible and configurable routing feature available.

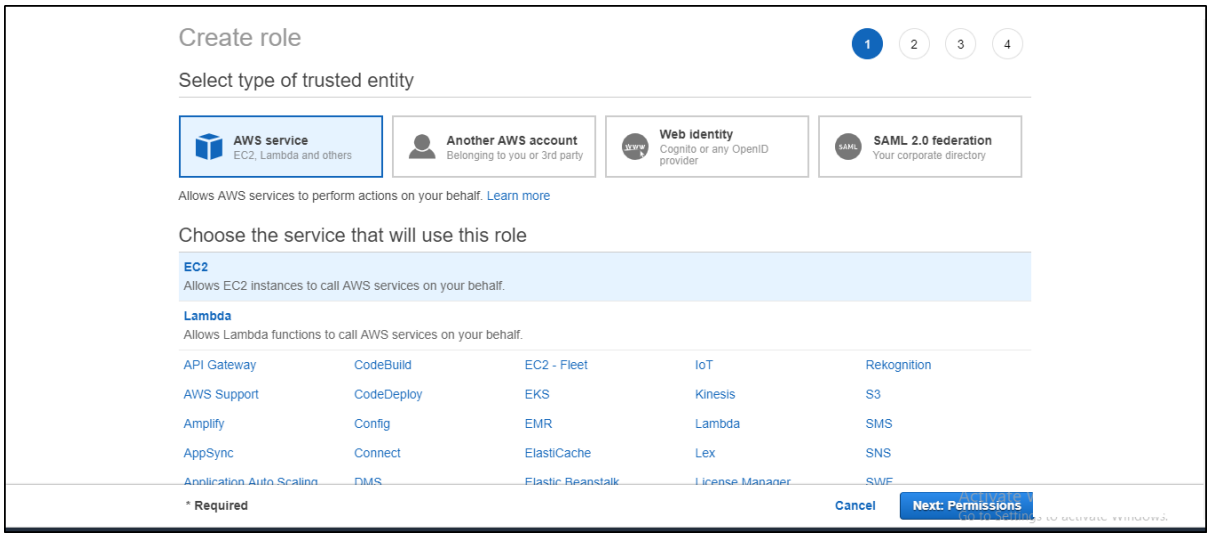
Demo 1: Create a Kubernetes cluster on AWS
The first thing that's needed to deploy any kind of service/application on Kubernetes, is a cluster. Every cluster consists of one master and single or multiple nodes depending on the requirement. In this demo, I'm going to show you how to create a Kubernetes cluster on AWS.

Step 1: Create an instance, name it kubectl. We're going to deploy the cluster using this instance. This instance only has the kubectl tool installed that will interact with the master, it does not have Kubernetes installed on it.

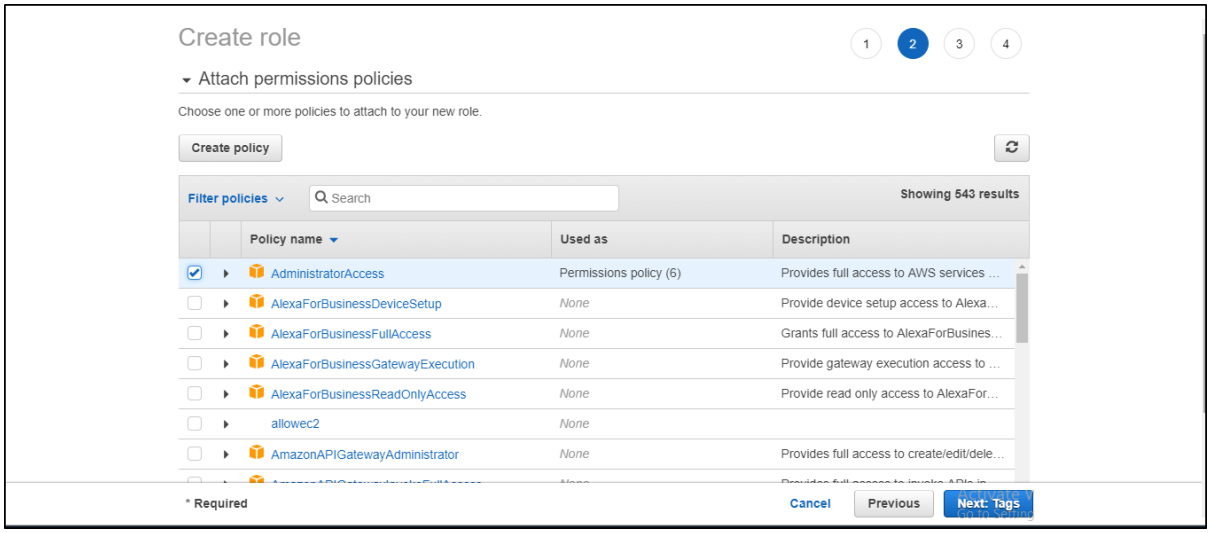
Note: We have three services in AWS that we'll be using here – s3 bucket which stores all the files, EC2 which is used to create an instance and deploy the service and IAM which is used to configure permissions. These three pictures have no clue about each other's existence and hence Roles come into the picture.

Step 2: Create a Role in the IAM section



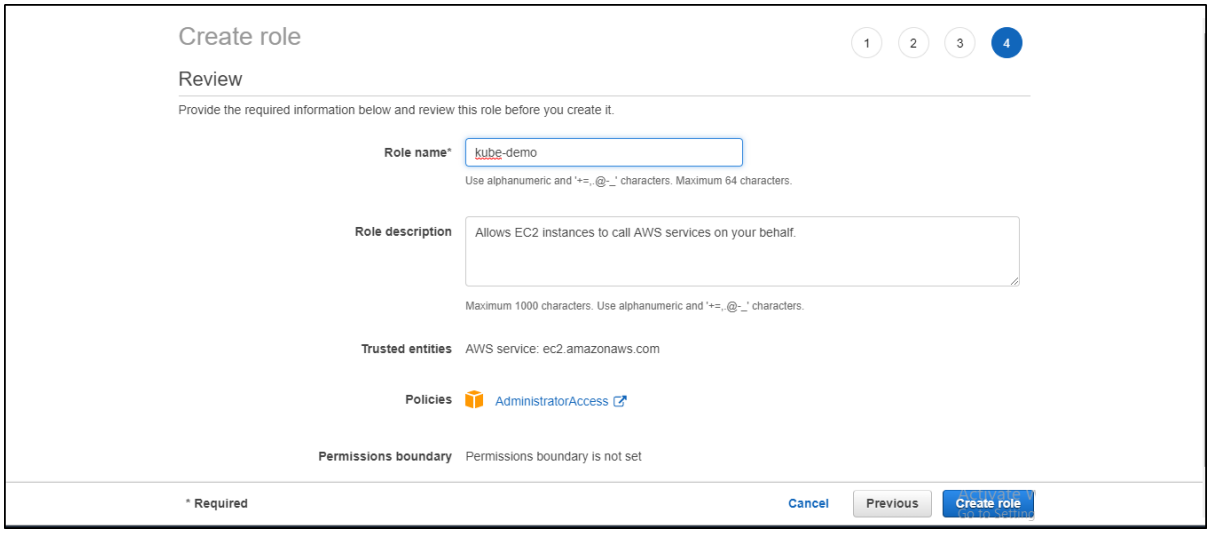


Attach the appropriate policy to your Role (for this example admin access is given)

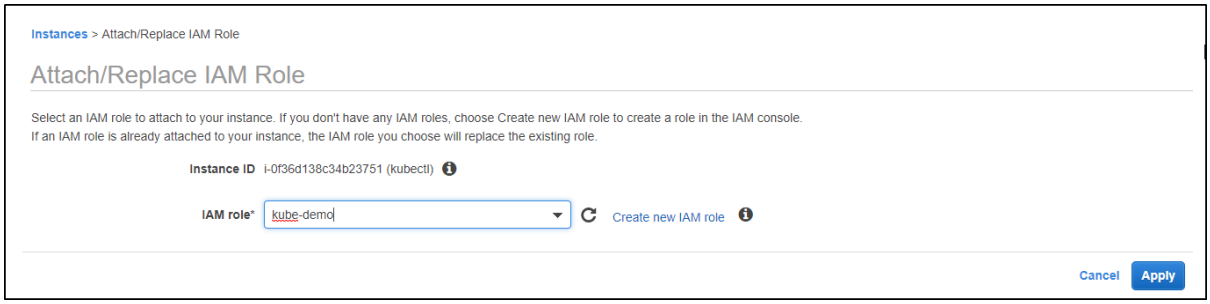


Next, it'll ask you to add tags which are optional. In my case, I haven't attached any tags.

Give your Role a name and review the policies assigned to it and then press **Create role**.



Step 3: Attach the role to the instance. Go to **instance settings -> Attach/Replace IAM role** -> attach the role you've created and then click on **Apply**.



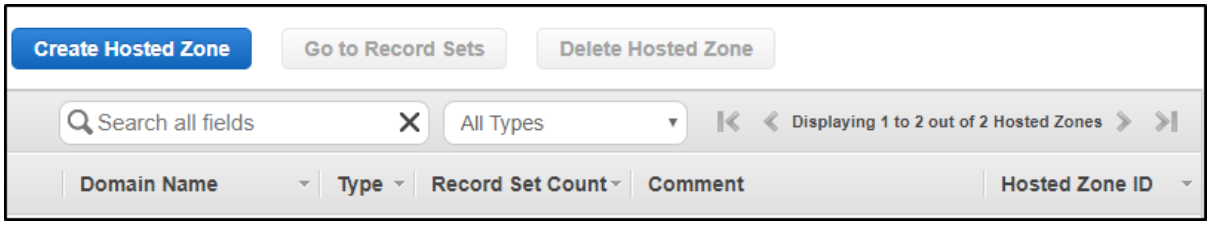
Step 4: Once you've created the instance and attached the role, open the command emulator i.e. cmder or putty and connect to the AWS instance. I'll be using cmder for this demo. Once you've connected to the instance, update the repository and install aws-cli using the following commands:

$ sudo apt-get install
$ sudo apt-get install awscli
Step 5: Install and set up kubectl using the following commands:

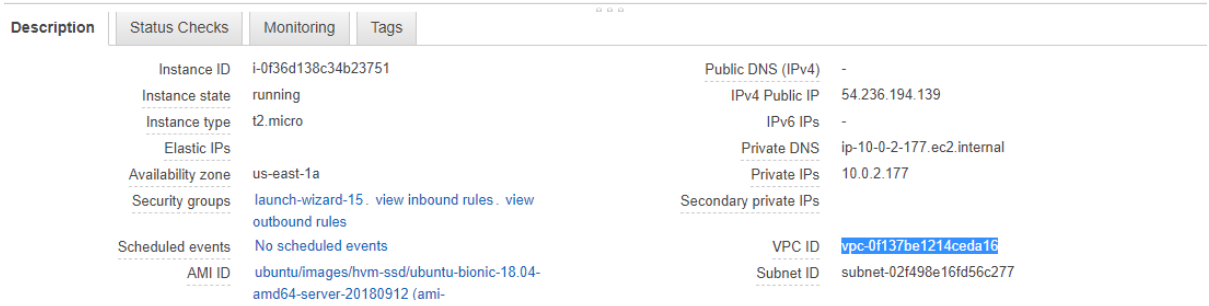
$ sudo apt-get update && sudo apt-get install -y apt-transport-https
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
$ echo "deb http://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee -a /etc/apt/sources.list.d/kubernetes.list
$ sudo apt-get update
$ sudo apt-get install -y kubectl
Step 6: Install Kops on the system using the following commands:

$ wget https://github.com/kubernetes/kops/releases/download/1.10.0/kops-linux-amd64
$ chmod +x kops-linux-amd64
$ mv kops-linux-amd64 /usr/local/bin/kops
Step 7: With Kops installed, you must configure a domain for your cluster to access it from outside. Create a hosted zone for it

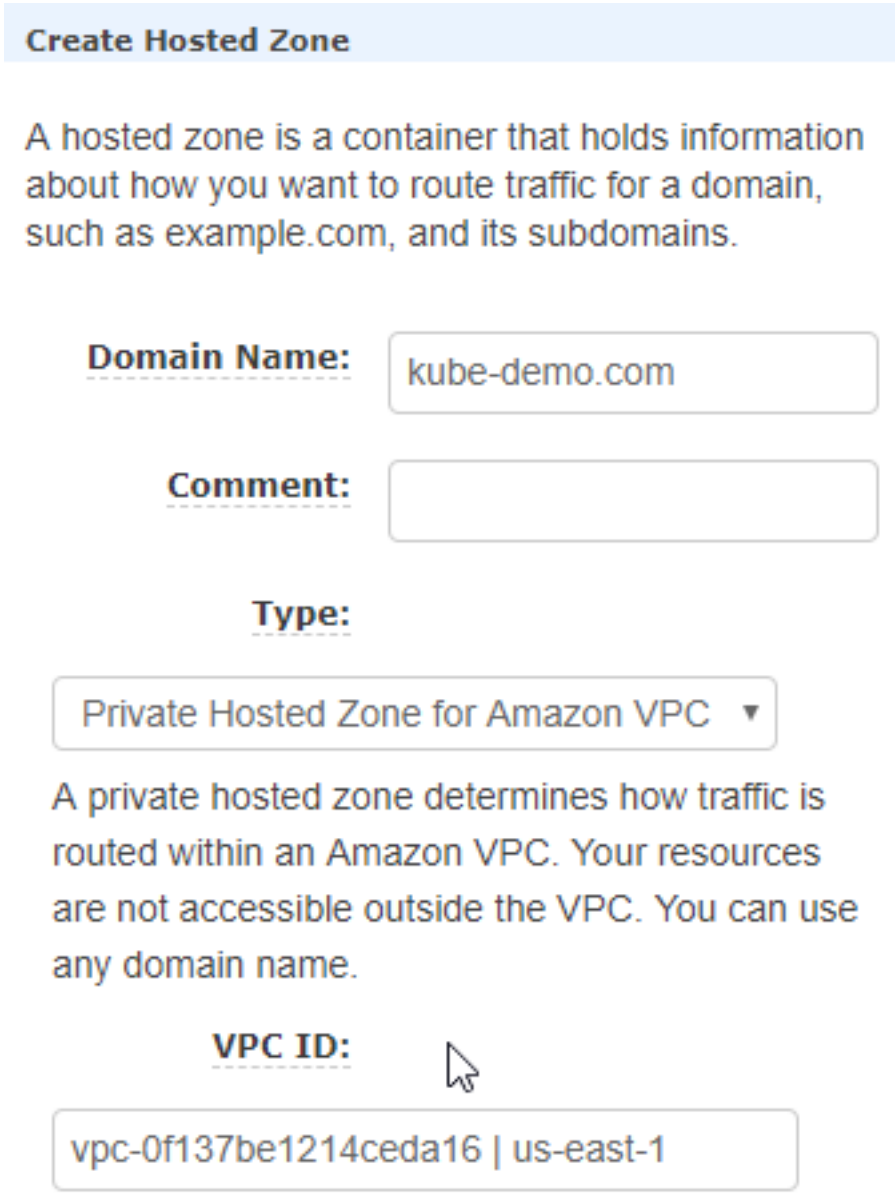
Services-> Route53-> Hosted zones-> Create Hosted Zone



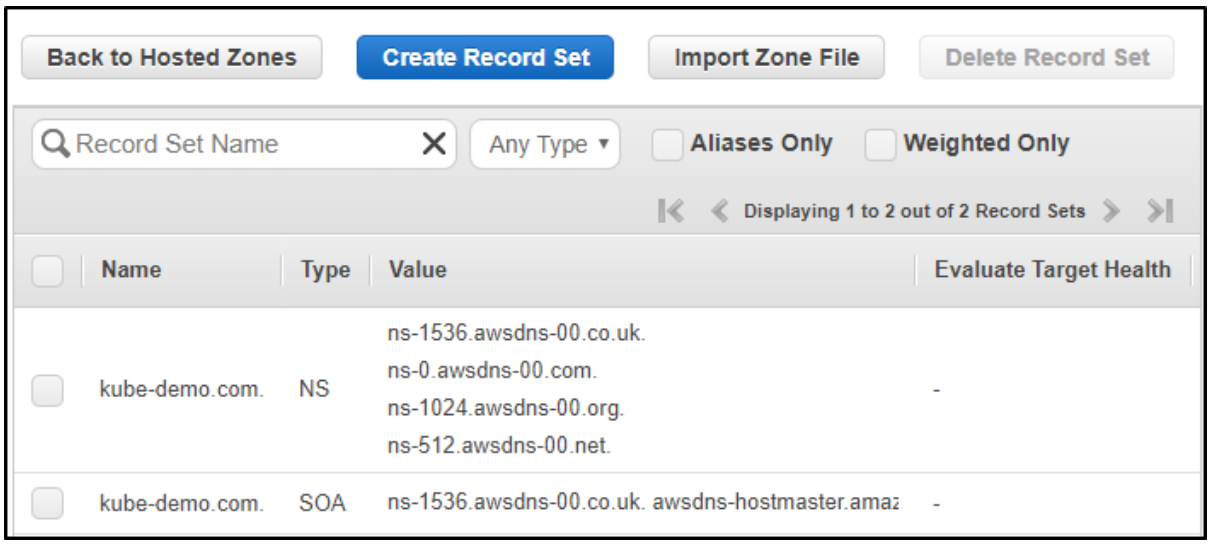
Add a domain name for your cluster, change the type from **Public Hosted Zone** to **Private Hosted Zone for Amazon VPC** and copy your instance **VPC ID** from the instance page to the VPC ID column and add the region you want to create your hosted zone in.



Copy the VPC ID



The above screenshot shows where to add Domain name and VPC ID



You can now see your Hosted Zone is created.

Step 8: Create a bucket as the same name as domain name using the following command:

$ aws s3 mb s3://kube-demo.com

```
ubuntu@ip-10-0-2-177:~$ aws s3 mb s3://kube-demo.com
make_bucket: kube-demo.com
ubuntu@ip-10-0-2-177:~$
```

Once you've created the bucket, execute the following command:

$ export KOPS_STATE_STORE=s3://kube-demo.com
Step 9: Before you create the cluster, you'll have to create SSH public key.

$ ssh-keygen
Enter file where you want your key pair to be saved and create a password to access the ssh public key. In this case, I've chosen the default location and used no password.

Step 10: Now that you've created the SSH key, create the cluster using the following command:

$ kops create cluster –cloud=aws –zones=us-east-1a –name=useast1.kube-demo.com –dns-zone=kube-demo.com –-dns private

```
Cluster configuration has been created.

Suggestions:
 * list clusters with: kops get cluster
 * edit this cluster with: kops edit cluster useast1.kube-demo.com
 * edit your node instance group: kops edit ig --name=useast1.kube-demo.com nodes
 * edit your master instance group: kops edit ig --name=useast1.kube-demo.com master-us-east-1a

Finally configure your cluster with: kops update cluster useast1.kube-demo.com --yes

ubuntu@ip-10-0-2-177:~$
```

And then update the cluster

Kubeconfigs

A Kubernetes configuration file, or kubeconfig, is a file that stores “information about clusters, users, namespaces, and authentication mechanisms.” It contains the configuration data needed to connect to and interact with one or more Kubernetes clusters.

You can find more information about kubeconfigs in the Kubernetes documentation:

https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/

Kubeconfigs contain information such as:

• The location of the cluster you want to connect to

• What user you want to authenticate as

• Data needed in order to authenticate, such as tokens or client certificates

You can even define multiple contexts in a kubeconfig file, allowing you to easily switch between multiple clusters.

Why Do We Need Kubeconfigs?

We use kubeconfigs to store the configuration data that will allow the many components of Kubernetes to connect to and interact with the Kubernetes cluster.

How will the kubelet service on one of our worker nodes know how to locate the Kubernetes API and authenticate with it? It will use a kubeconfig!

In the next lesson, we will generate the kubeconfigs that our cluster needs.