



Kubernetes Networking



CLARUSWAY®
WAY TO REINVENT YOURSELF

Students, write your response!

Pear Deck Interactive Slide
Do not remove this bar



Table of Contents

- ▶ Cluster Networking
- ▶ Services
- ▶ Service Types
- ▶ Labels and loose coupling



1

Cluster Networking



Cluster Networking

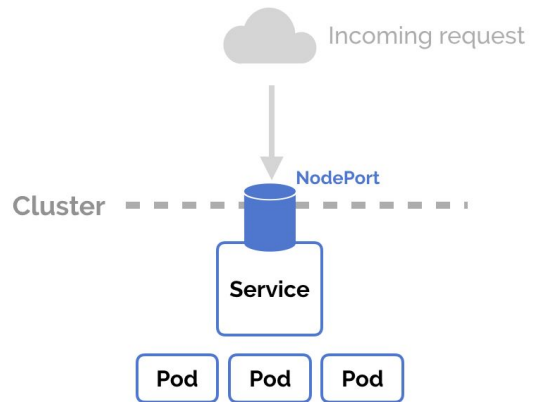
There are 4 distinct networking problems to address:

1. container-to-container communications:
This is solved by Pods and localhost communications.
2. Pod-to-Pod communications:
Each Kubernetes Pod gets its own IP address.
3. Pod-to-Service communications: this is covered by services.
4. External-to-Service communications: this is covered by services.



2

Services



Services

FRONTEND

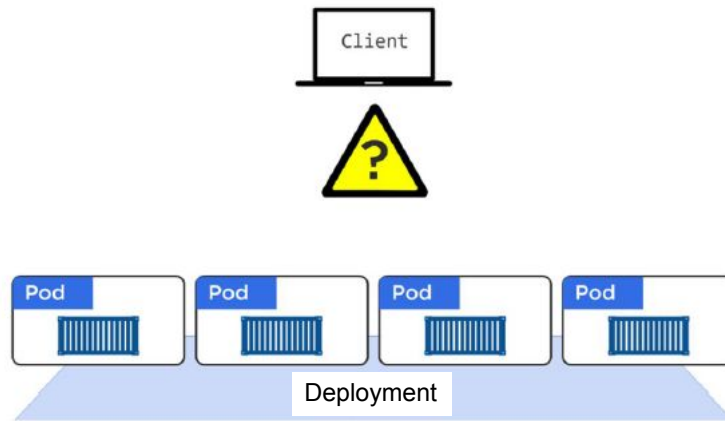
BACKEND

DATABASE



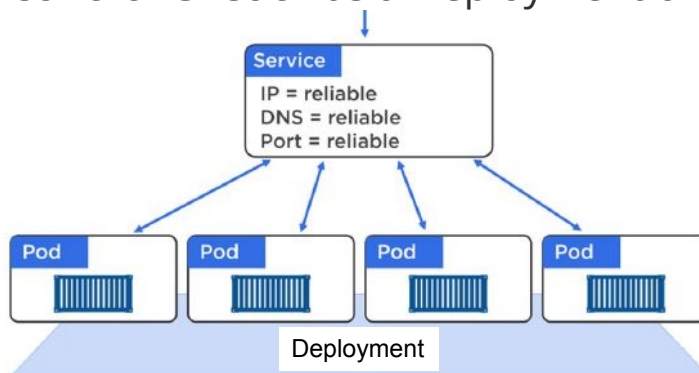
Services

Each Kubernetes Pod gets its own IP address. But Kubernetes **Pods** are mortal. They are born and when they die, they are not resurrected. If you use a Deployment to run your app, it can create and destroy Pods dynamically. So, Pod IPs are unreliable.



Services

A **Service** offers a single **DNS entry** for a containerized application managed by the Kubernetes cluster, regardless of the number of replicas, by providing a common **load balancing** access point to a set of pods logically grouped and managed by a **controller** such as a Deployment or ReplicaSet.



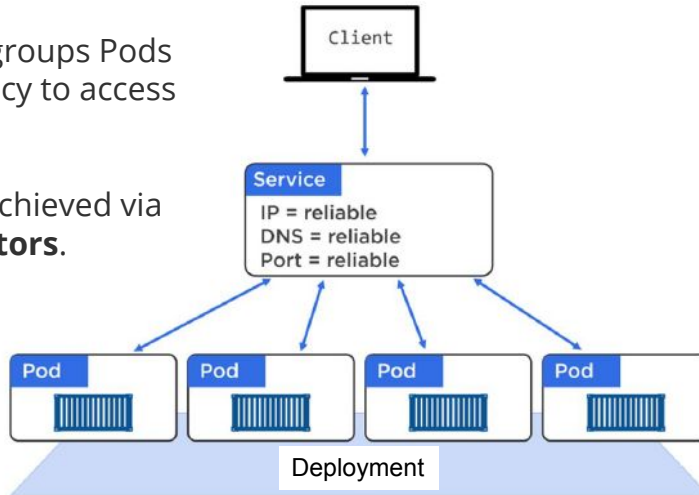


Services

The **Service** is associated with the Pods, and provides them with a stable IP, DNS and port. It also **loadbalances** requests across the Pods.

Service logically groups Pods and defines a policy to access them.

This grouping is achieved via **Labels** and **Selectors**.



CLARUS

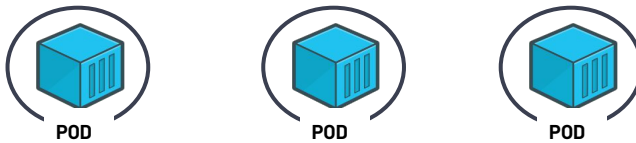
WAY TO REINVENT YOURSELF

9



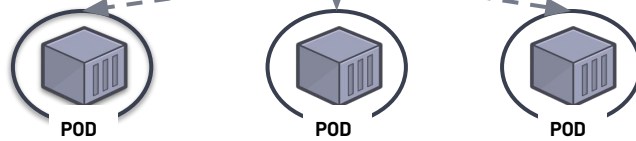
Services

FRONTEND



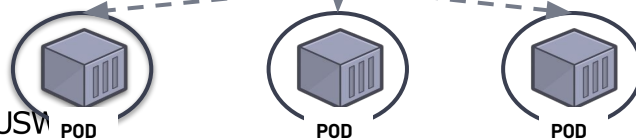
BACKEND-SERVICE

BACKEND



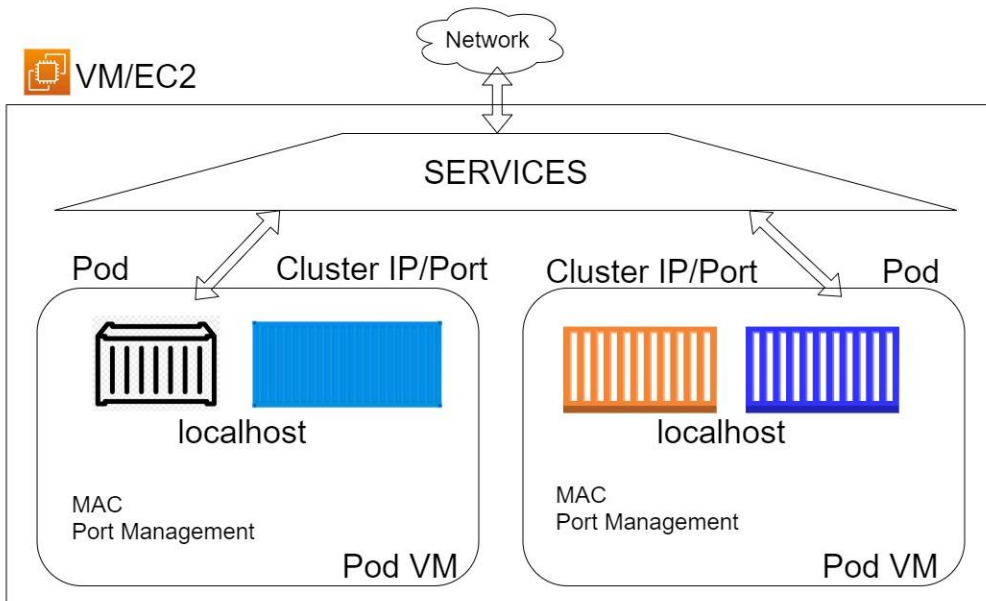
DB-SERVICE

DATABASE



CLARUS

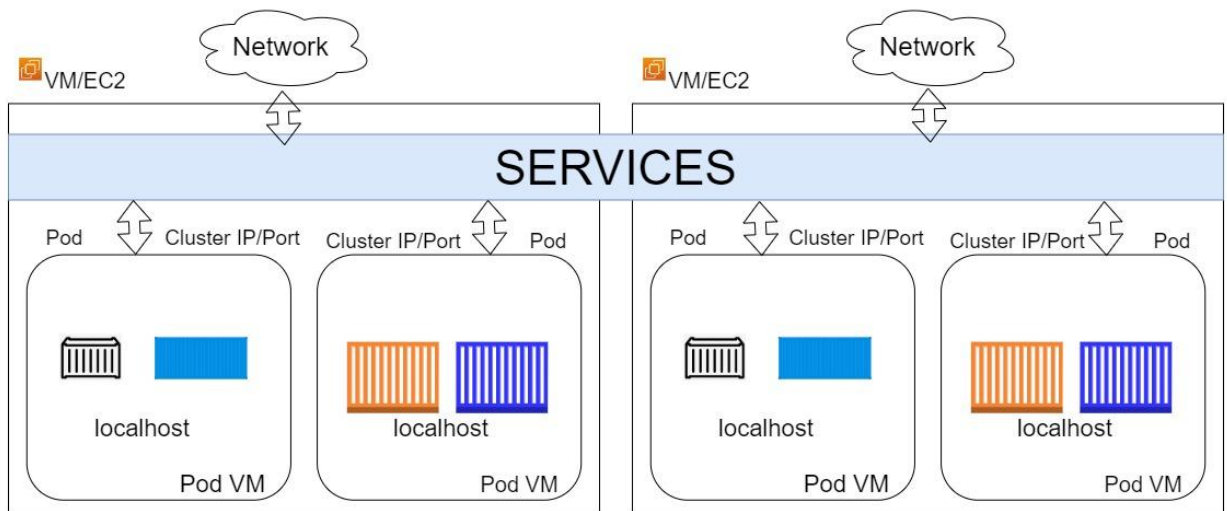
WAY TO REINVENT YOU



CLARUS K8s Network

WAY TO REINVENT YOURSELF

11



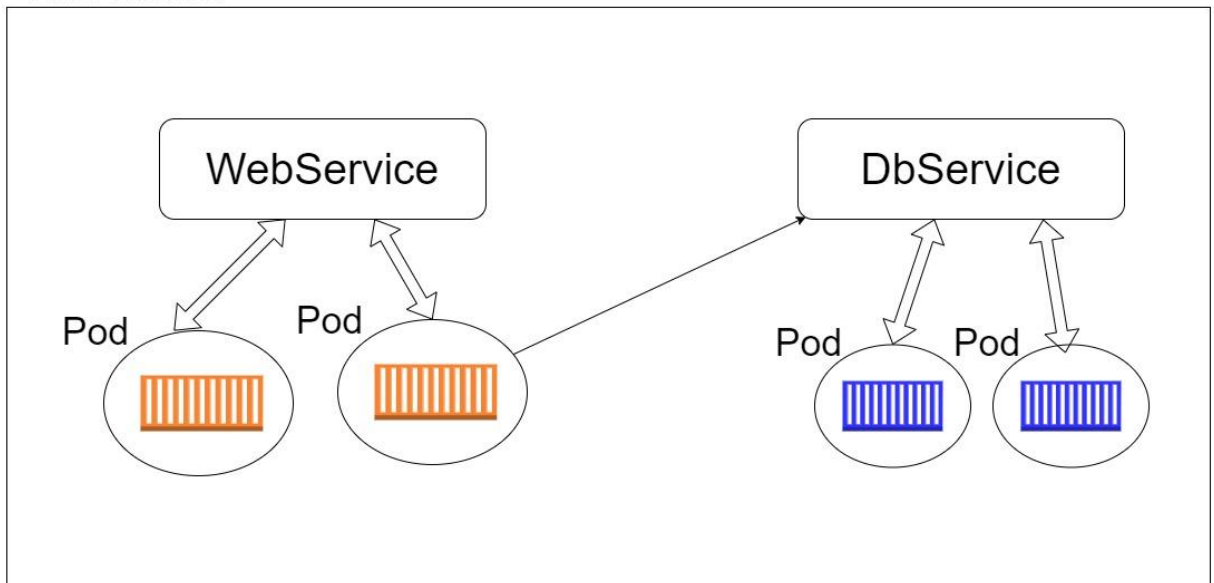
K8s Network

CLARUSWAY©

WAY TO REINVENT YOURSELF

12

K8s Cluster



kube-proxy

- Each cluster node runs a daemon called **kube-proxy**, that watches the API server on the master node for the addition, updates, and removal of Services and endpoints.
- **kube-proxy** is responsible for **implementing the Service configuration** on behalf of an administrator or developer, in order to enable traffic **routing** to an exposed application running in Pods.
- For each new Service, on each node, **kube-proxy** configures **iptables** rules to capture the traffic for its **ClusterIP** and forwards it to one of the Service's endpoints.
- Therefore any node can receive the external traffic and then route it internally in the cluster based on the **iptables** rules.
- When the Service is removed, **kube-proxy** removes the corresponding **iptables** rules on all nodes as well.



► Service Discovery

- Kubernetes has an add-on for **DNS**, which creates a DNS record for each Service and its format is:

`web-svc.my-namespace.svc.cluster.local`

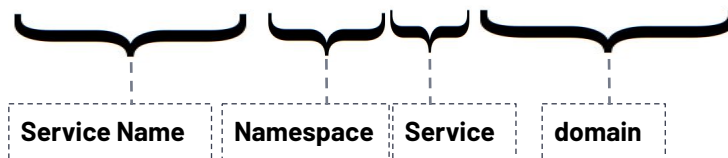
- Services within the same Namespace find other Services just by their names.
- If we add a Service **redis-master** in **my-ns** Namespace, all Pods in the same **my-ns** Namespace lookup the Service just by its name, **redis-master**.



► Service Discovery

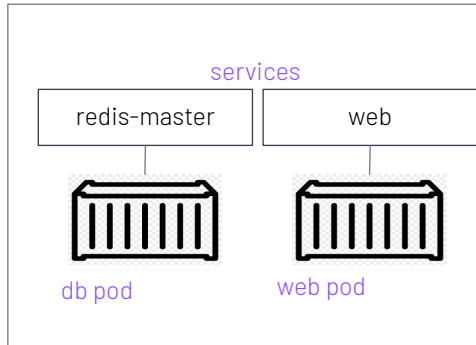
- Pods from other Namespaces, such as **test-ns**, lookup the same Service by adding the respective Namespace as a suffix, such as **redis-master.my-ns** or providing the **FQDN** of the service as below.

`redis-master.my-ns.svc.cluster.local`

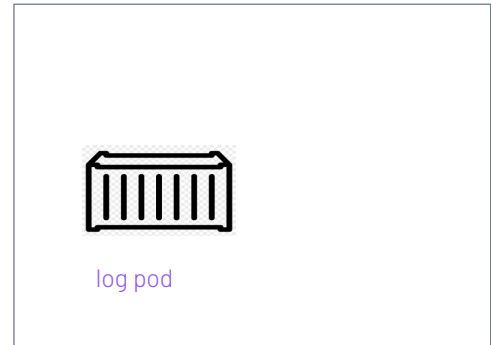




my-ns



test-ns



To connect to the "db pod":

From "web pod" -> hostname: redis-master:port

From "log pod" -> hostname: redis-master.my-ns:port

From "log pod" -> hostname: redis-master.my-ns.svc.cluster.local:port



3

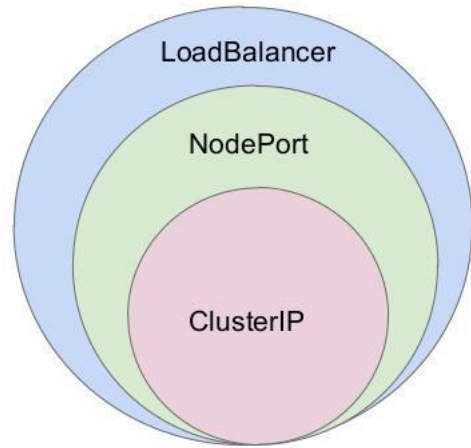
Service Types



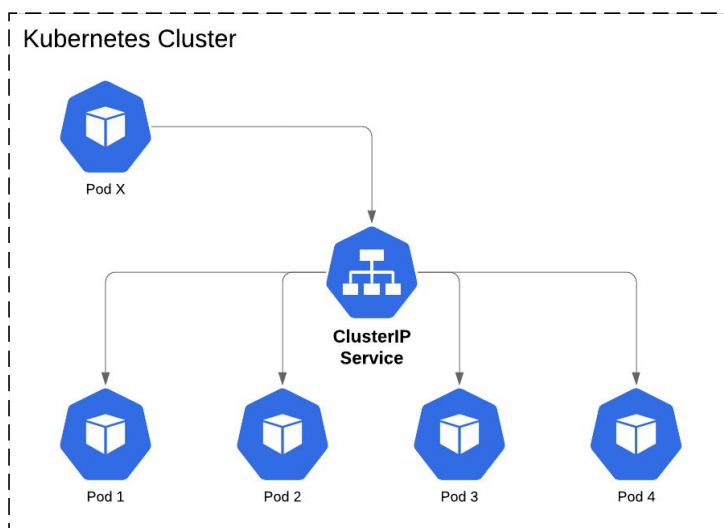
Service Types

There are 4 major service types:

- ClusterIP (default)
- NodePort
- LoadBalancer
- ExternalName



Service Types/ClusterIP



ClusterIP:

Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default ServiceType.

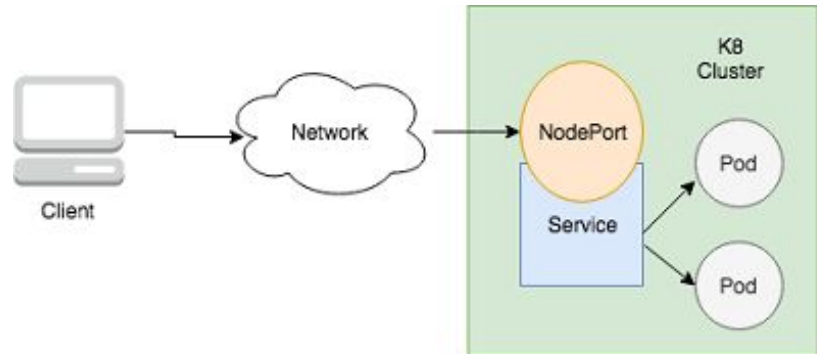
Good for service of database & back-end apps.



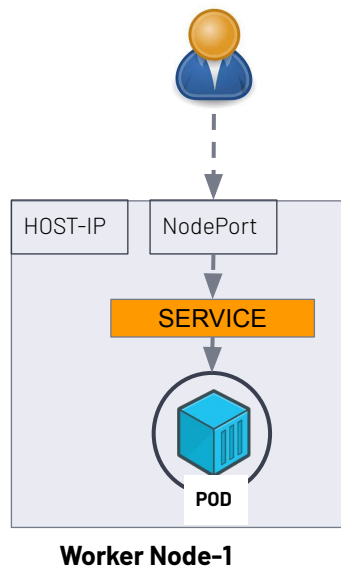
Service Types/NodePort

NodePort: Exposes the Service on each Node's IP at a static port (the NodePort). A ClusterIP Service, to which the NodePort Service **routes**, is automatically created. Port can either be **statically** defined, or **dynamically** taken from a range between 30000-32767.

With the **NodePort** *ServiceType*, in addition to a ClusterIP, a high-port is mapped to the respective Service, from all the worker nodes.

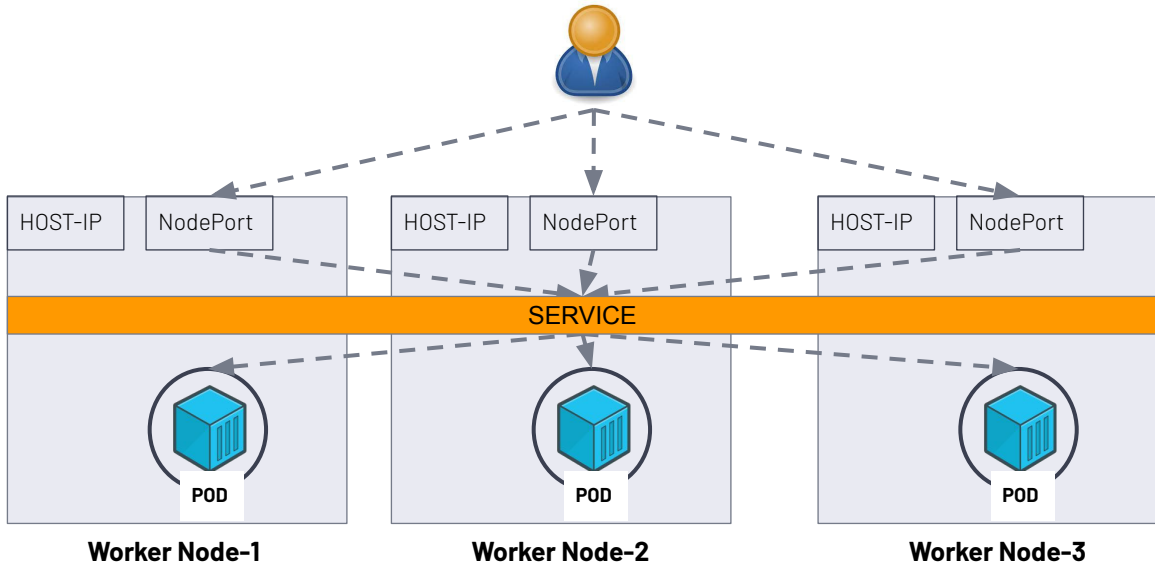


Service Types/NodePort





Service Types/NodePort



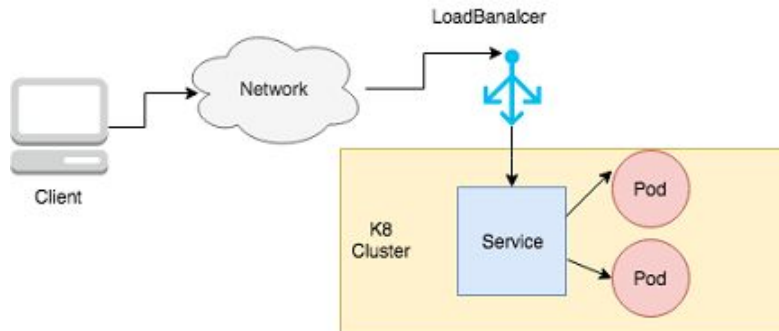
Service Types/NodePort

- The **NodePort** ServiceType is useful when we want to make our Services accessible from the external world.
- The end-user connects to any worker node on the specified high-port, which proxies the request internally to the ClusterIP of the Service, then the request is forwarded to the applications running inside the cluster.
- Let's not forget that the Service is load balancing such requests, and only forwards the request to one of the Pods running the desired application.



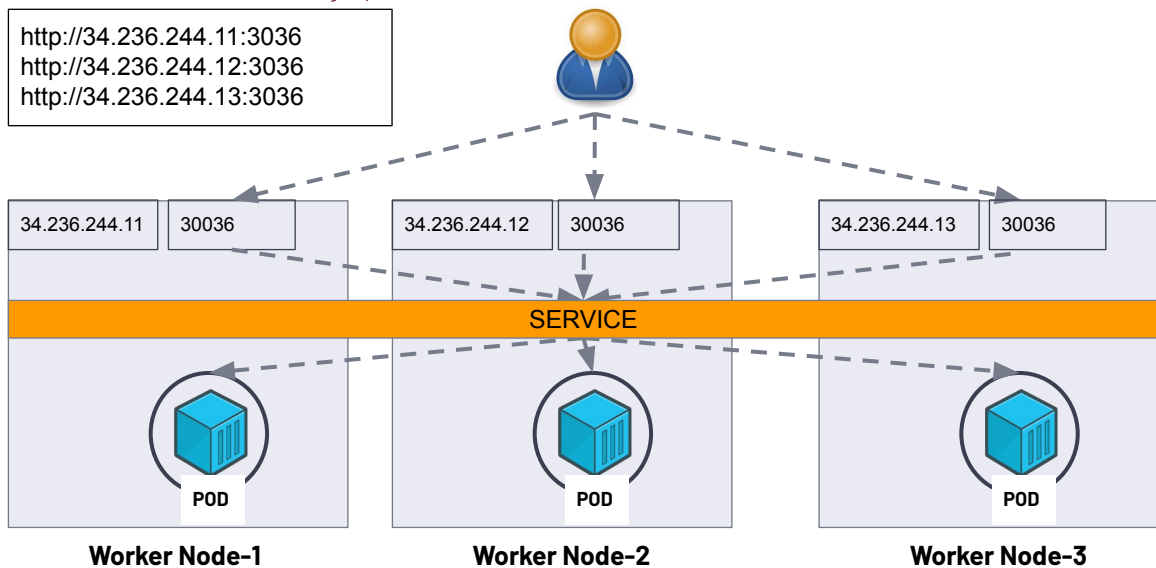
Service Types/LoadBalancer

LoadBalancer: Exposes the Service externally using a cloud provider's load balancer. NodePort and ClusterIP are automatically created, and the external load balancer will route to them.



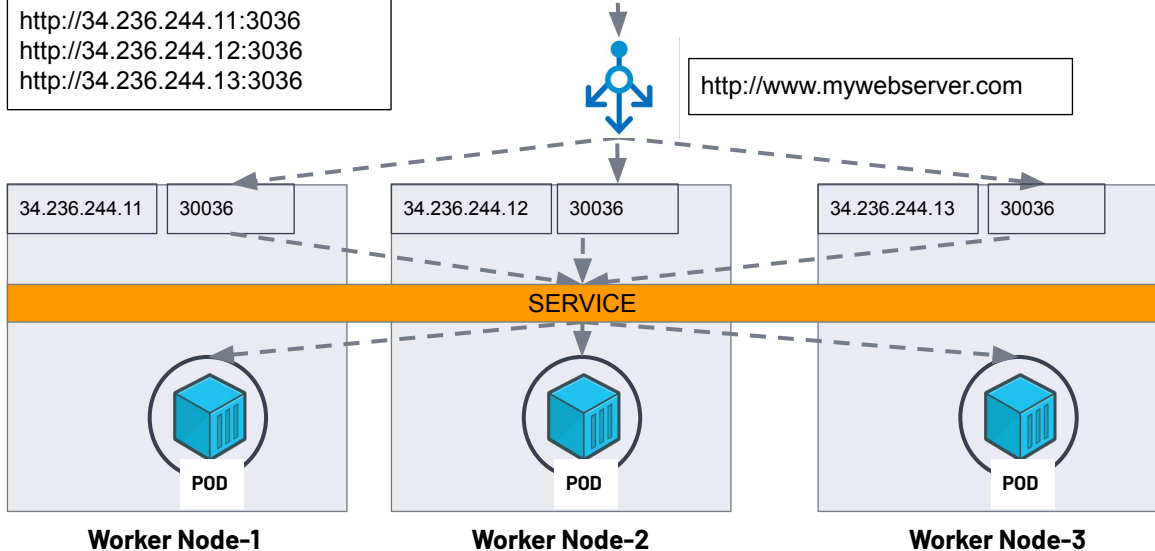
Service Types/LoadBalancer

`http://34.236.244.11:3036`
`http://34.236.244.12:3036`
`http://34.236.244.13:3036`





Service Types



Service Types/LoadBalancer

LoadBalancer:

- The **LoadBalancer** *ServiceType* will only work if the underlying infrastructure supports the automatic creation of Load Balancers and have the respective support in Kubernetes, as is the case with the Google Cloud Platform and AWS.
- If no such feature is configured, the **LoadBalancer IP** address field is **not populated**, it remains in **Pending** state, but the **Service will still work as a typical NodePort type Service**.



▶ Service Types/ExternalName

ExternalName: Maps the Service to the contents of the externalName field (e.g. example.com), by returning a CNAME record with its value.

```
apiVersion: v1
kind: Service
metadata:
  name: example-prod
spec:
  type: ExternalName
spec:
  externalName: example.com
```



▶ Service Types/ExternalName

ExternalName is a special *ServiceType*, that has no Selectors and does not define any endpoints.

When accessed within the cluster, it returns a **CNAME** record of an externally configured Service.

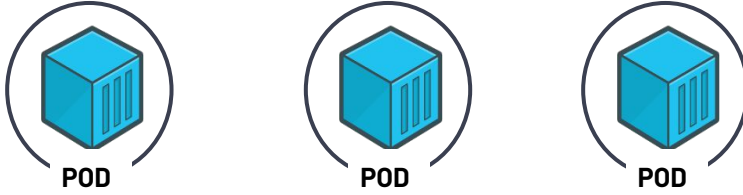
The primary use case of this *ServiceType* is to make externally configured Services like **my-database.example.com** available to applications inside the cluster.

If the externally defined Service resides within the same Namespace, using just the name **my-database** would make it available to other applications and Services within that same Namespace.



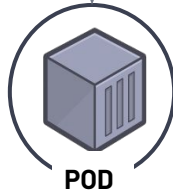
Services/ExternalName

BACKEND



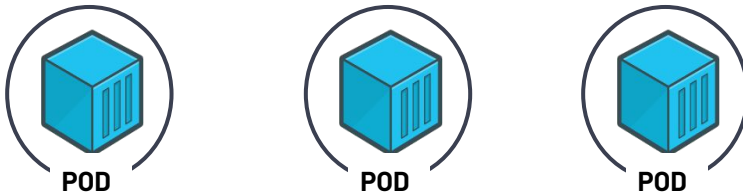
DB-SERVICE (ClusterIP)

DATABASE



Services/ExternalName

BACKEND



DB-SERVICE (ExternalName)

DATABASE

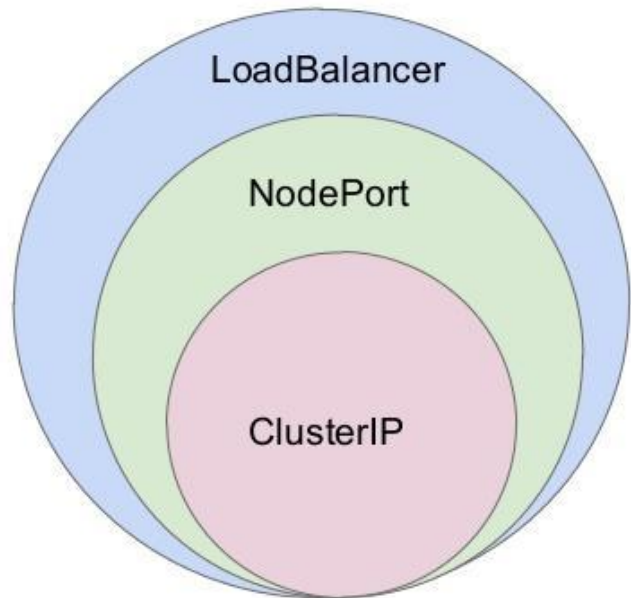


RDS



Service Types

- **LoadBalancer** Service has **NodePort** and **ClusterIp** features.
- **NodePort** Service has **ClusterIp** features.



4

Labels and loose coupling



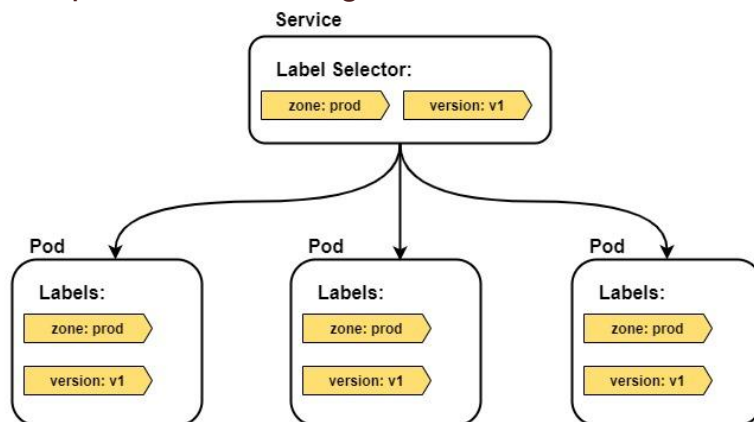
Labels and loose coupling

- Labels and Selectors use a **key/value** pair format.
- Pods and Services are loosely coupled via labels and label selectors.
- For a Service to match a set of Pods, and therefore provide stable networking and load-balance, it only needs to match some of the Pods labels.
- However, for a Pod to match a Service, the Pod must match all of the values in the Service's label selector.



Labels and loose coupling

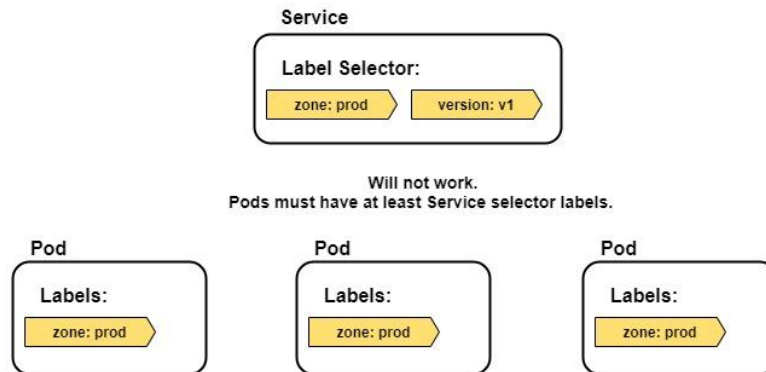
The figure below shows an example where 3 Pods are labeled as **zone=prod** and **version=1**, and the Service has a label selector that matches. This Service provides stable networking to all three Pods. It also provides simple load-balancing.





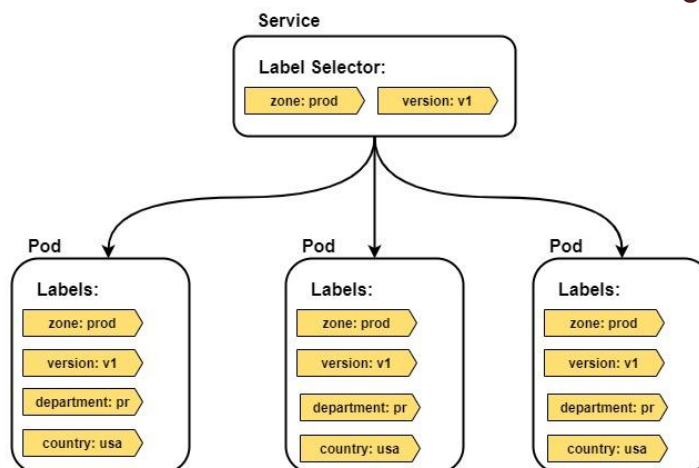
Labels and loose coupling

The figure below shows an example where the Service does not match any of the Pods. This is because the Service is selecting on two labels, but the Pods only have one of them. The logic behind this is a Boolean AND operation.



Labels and loose coupling

This figure shows an example that does work. It doesn't matter that the Pods have additional labels that the Service is not selecting on.





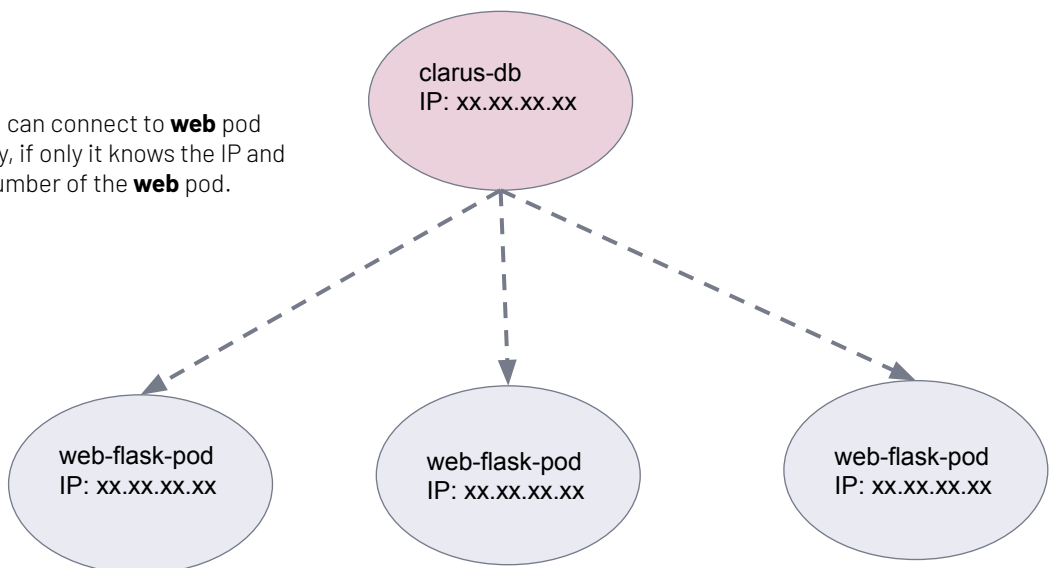
5

Kubernetes hands-on-03



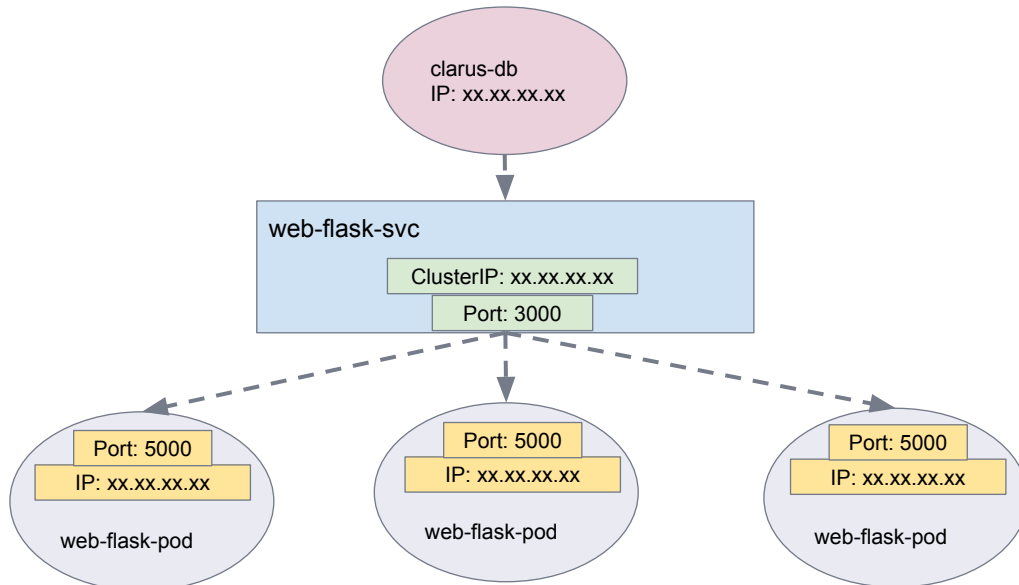
Pod to Pod Connection

db pod can connect to **web** pod directly, if only it knows the IP and port number of the **web** pod.

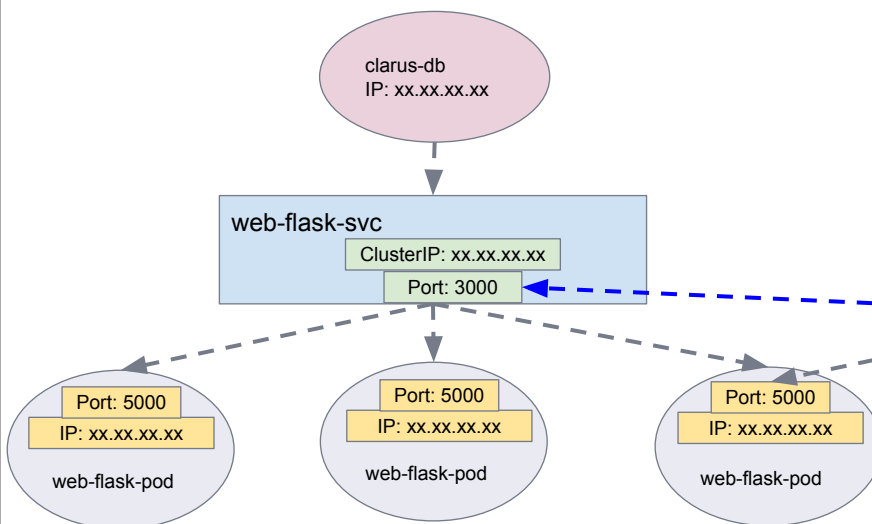




ClusterIP

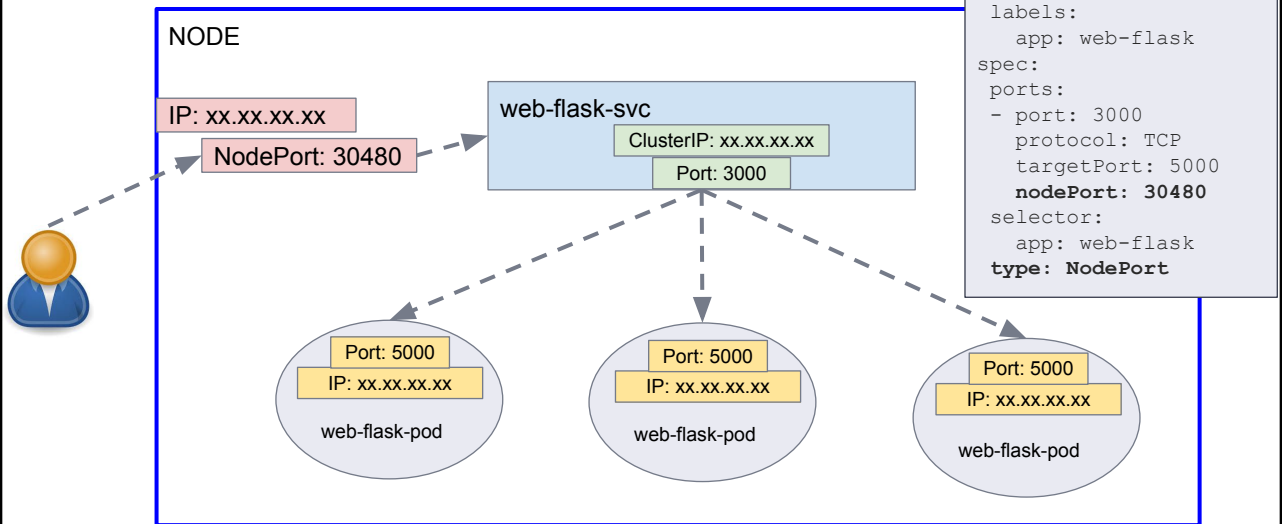


ClusterIP

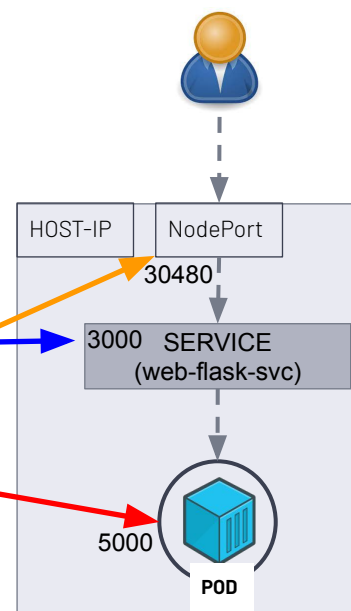


```
apiVersion: v1
kind: Service
metadata:
  name: web-flask-svc
  labels:
    app: web-flask
spec:
  type: ClusterIP
  ports:
    - port: 3000
      targetPort: 5000
  selector:
    app: web-flask
```

NodePort



```
apiVersion: v1
kind: Service
metadata:
  name: web-flask-svc
  labels:
    app: web-flask
spec:
  ports:
    - port: 3000
      protocol: TCP
      targetPort: 5000
      nodePort: 30480
  selector:
    app: web-flask
  type: NodePort
```

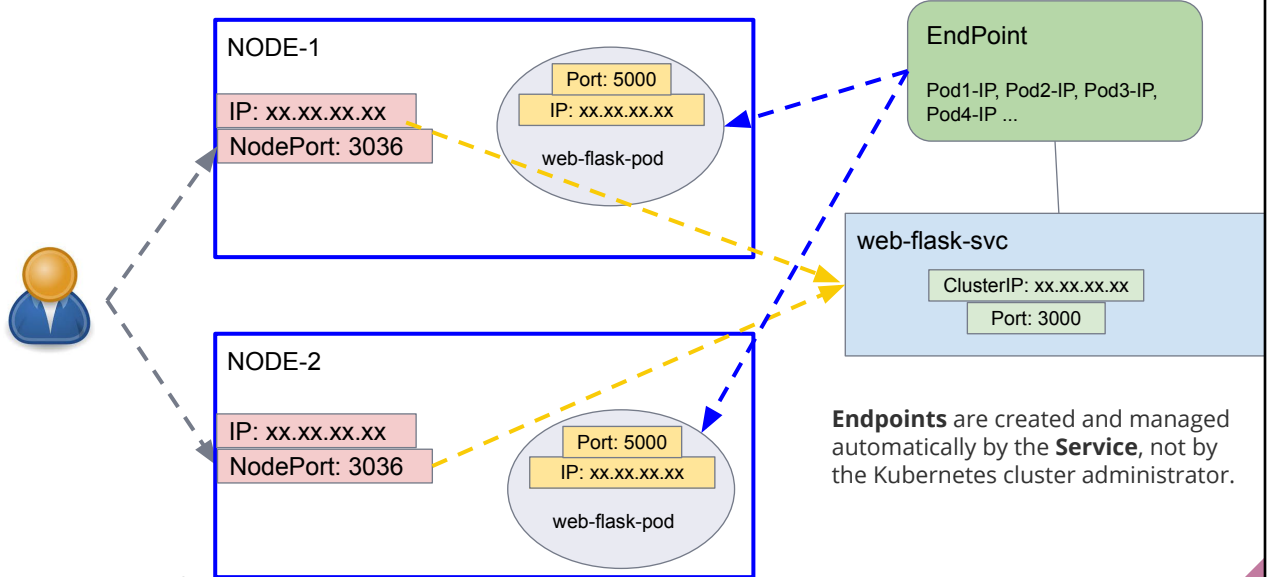


Worker Node-1



Endpoint

loadbalances



CLARUSWAY©
WAY TO REINVENT YOURSELF



THANKS!

Any questions?

You can find me at:

► joe@clarusway.com



CLARUSWAY©
Students, write your response!

Pear Deck Interactive Slide
Do not remove this bar

4
6