Kubernetes Networking

Students, write your response!

Pear Deck Interactive Slide
Do not remove this bar

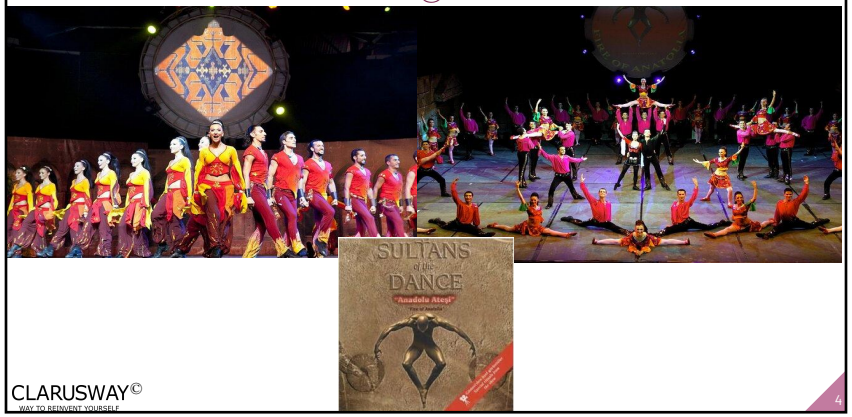
CLARUSWAY©
WAY TO REINVENT YOURSELF

Table of Contents

1

Cluster Networking

Cluster Networking



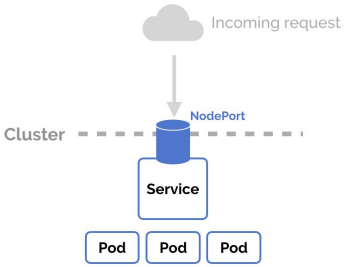
Cluster Networking

There are 4 distinct networking problems to address:

1. container-to-container communications:
2. Pod-to-Pod communications:
3. Pod-to-Service communications: this is covered by services.
4. External-to-Service communications: this is covered by services.

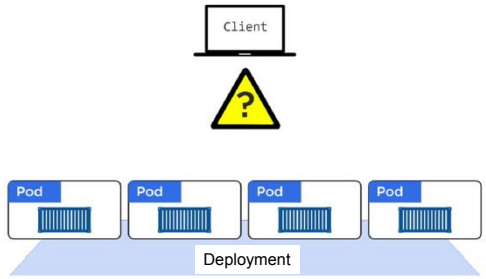
2

Services



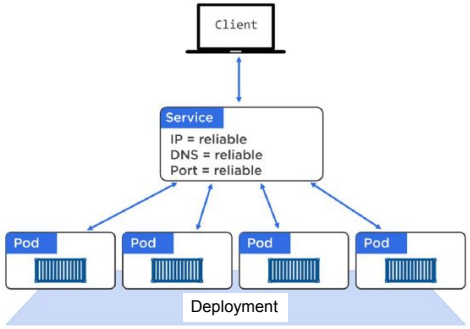
Services

Pods are not reliable



Services

A **Service** offers a single **DNS entry** for a containerized application managed by the Kubernetes cluster

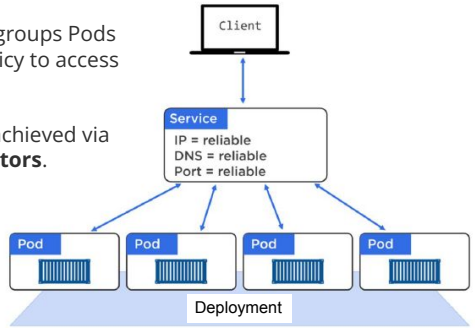


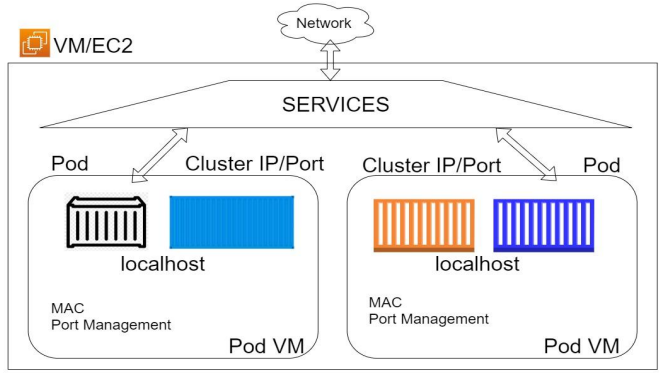
Services

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

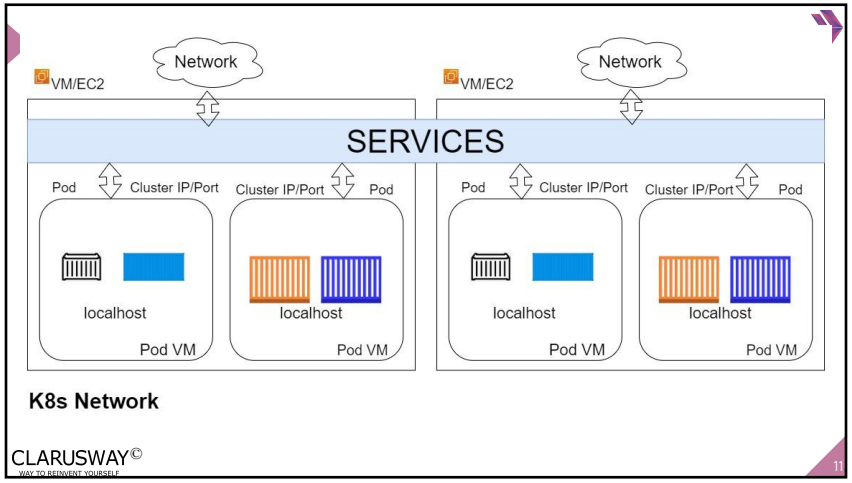
Service logically groups Pods and defines a policy to access them.

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

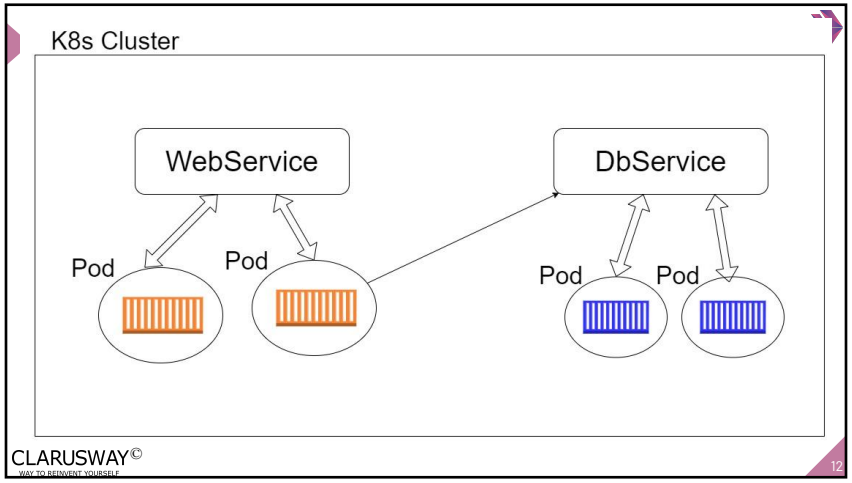




K8s Network

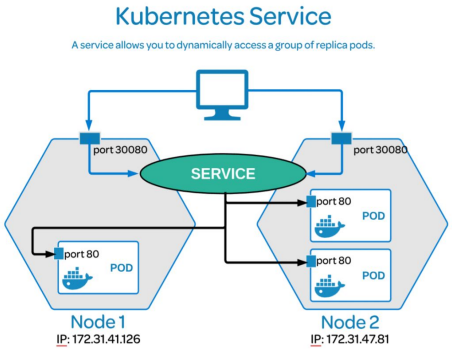

Network
VM/EC2
Network
VM/EC2
SERVICES
Pod
Cluster IP/Port
Cluster IP/Port
Pod
Pod
Cluster IP/Port
Cluster IP/Port
Pod
localhost
localhost
localhost
localhost
Pod VM
Pod VM
Pod VM
Pod VM

K8s Network


K8s Cluster
WebService
DbService
Pod
Pod
Pod
Pod

Services

Kubernetes **Services** enable communication between various components **within** and **outside** of the application. Kubernetes Services helps us connect applications together with other applications or users.

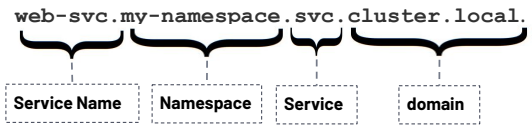


kube-proxy

- Each cluster node runs a daemon called **kube-proxy**
- `kube-proxy` is responsible for ***implementing the Service configuration*** on behalf of an administrator or developer
- 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.
- When the Service is removed, `kube-proxy` removes the corresponding **iptables** rules on all nodes as well.

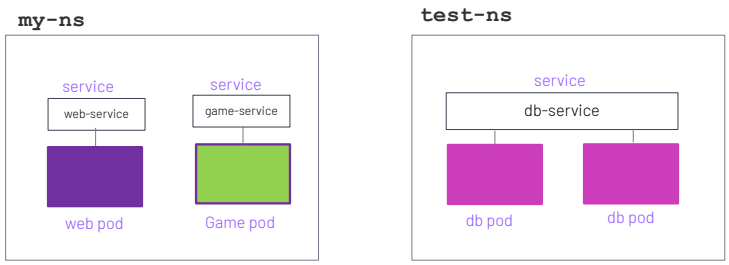
Service Discovery

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



- 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`**.

FQDN: fully qualified domain name



To connect to the "Game pod" and "db pod":

From " web pod" -> " Game pod" —> hostname: game-sevice.my-ns:port
game-service:port

From " web pod" -> "db pod —–> hostname: db-service.test-ns.svc.cluster.local:port

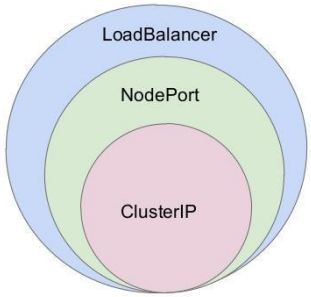
3

Service Types

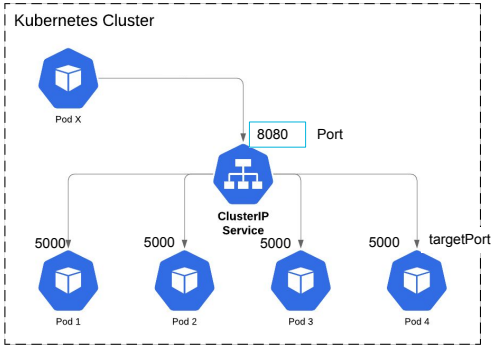
Service Types

There are 4 major service types:

- ClusterIP (default)
- NodePort
- LoadBalancer
- ExternalName



Service Types

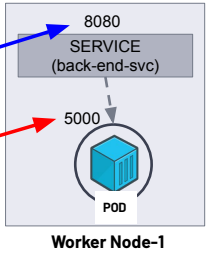


ClusterIP:
Expose traffic internally

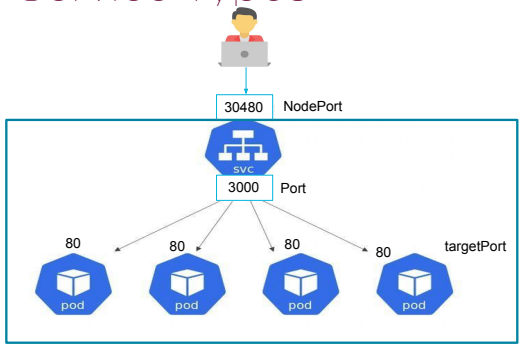
Example Usecase:

Good for service of database & back-end apps.


```
apiVersion: v1
kind: Service
metadata:
 name: back-end-svc
 labels:
   app: back-end
spec:
 type: ClusterIP (default)
 selector:
   app: back-end
 ports:
 - port: 8080
   protocol: TCP
   targetPort: 5000
```



Service Types

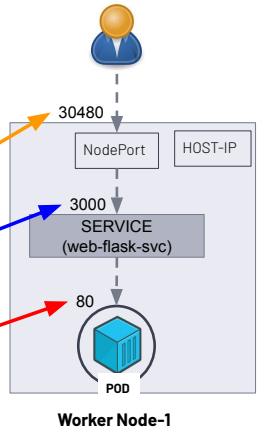


NodePort:
Exposes traffic to the outside.

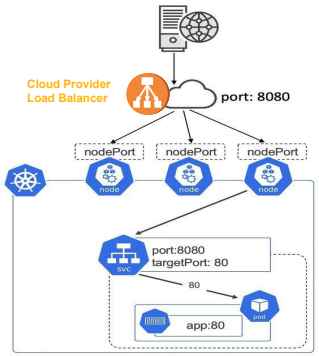
Example Usecase:

when we want to make our Services which has our app or website accessible from the external world.


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



Service Types

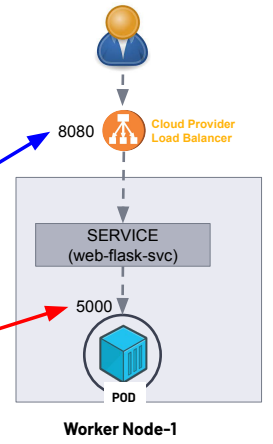


LoadBalancer:
Exposes traffic outside with load balancing feature.

Example Usecase:

when we want to load balancing our Services which has our app or website accessible from the external world.


```
apiVersion: v1
kind: Service
metadata:
 name: web-flask-svc
 labels:
   app: web-flask
spec:
 type: LoadBalancer
 selector:
   app: web-flask
 ports:
 - port: 8080
   protocol: TCP
   targetPort: 5000
```

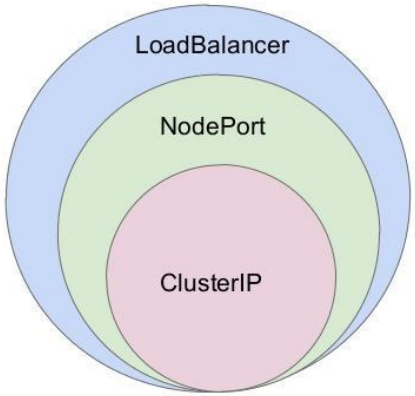


Service Types

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, Azure or 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



Service Types

ExternalName:

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

Example Usecases:

to make externally configured services like;

remote.server.url.com

available to applications inside the cluster.

Service Types

```
apiVersion: v1
kind: Service
metadata:
  labels: io.kompose.service: mysql-server
  name: mysql-server
spec:
  type: ExternalName
  externalName:
serdar.cbanmzptkrzf.us-east-1.rds.amazonaws.com
```

4

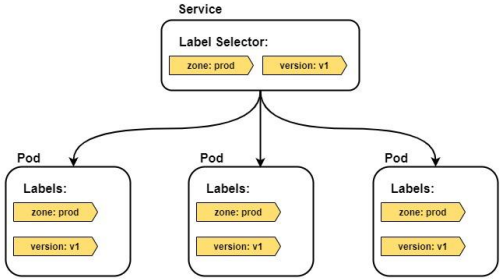
Labels and loose coupling

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.

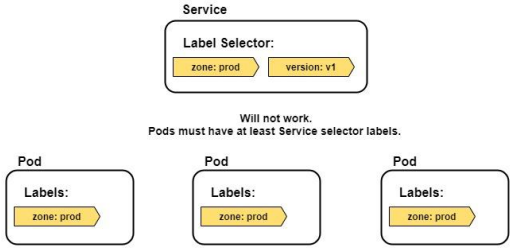
Labels and loose coupling

The figure below shows an example where 3 Pods are labeled as **`zone=prod`** and **`version=v1`**, and the Service has a label selector that matches. This Service provides stable networking to all three Pods. It also provides simple load-balancing.



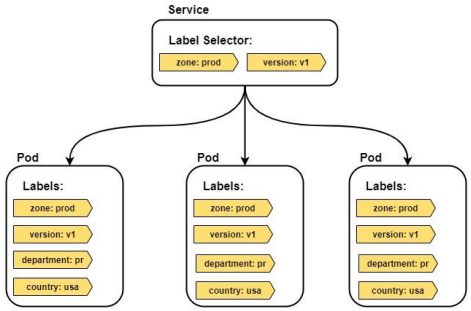
Labels and loose coupling

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.



THANKS!

Any questions?

You can find me at:

- Email: serdar@clarusway.com
- Slack: @serdar - instructor

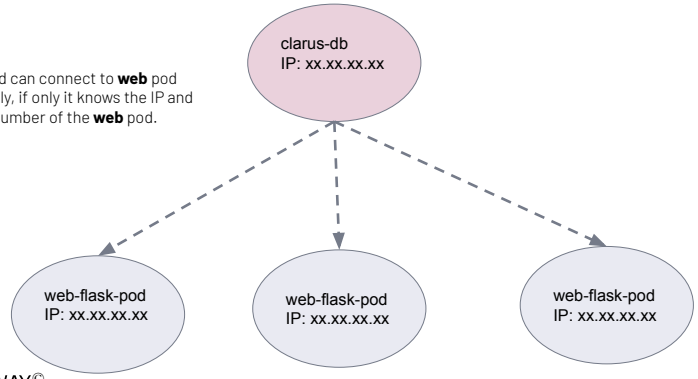
5

Kubernetes hands-on-03

CLARUSWAY©
WAY TO REINVENT YOURSELF

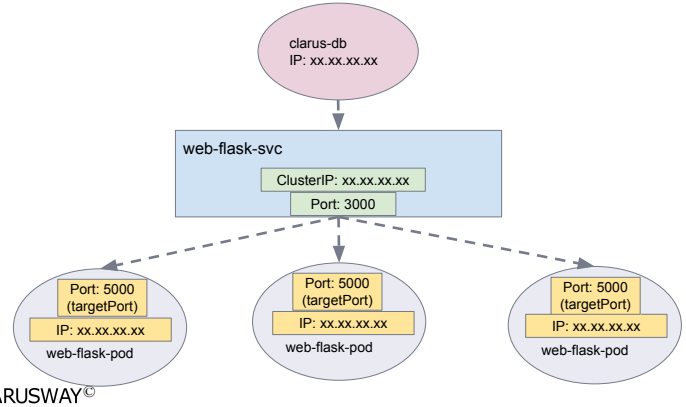
Pod to Pod Connection

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



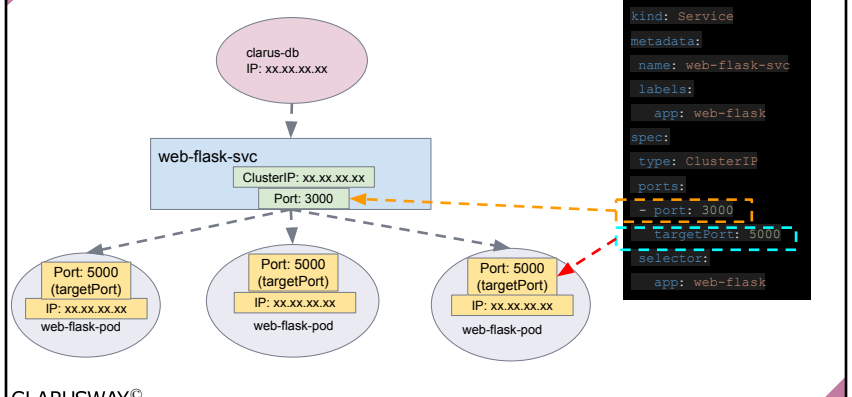
CLARUSWAY©
WAY TO REINVENT YOURSELF

ClusterIP

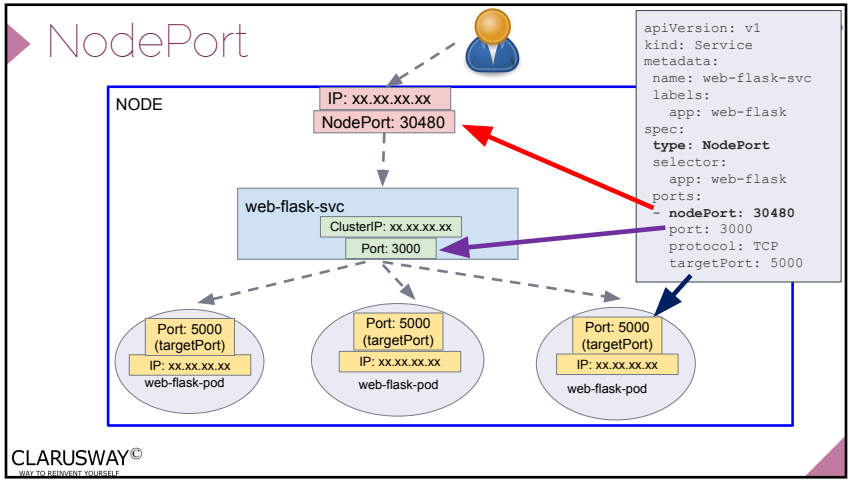


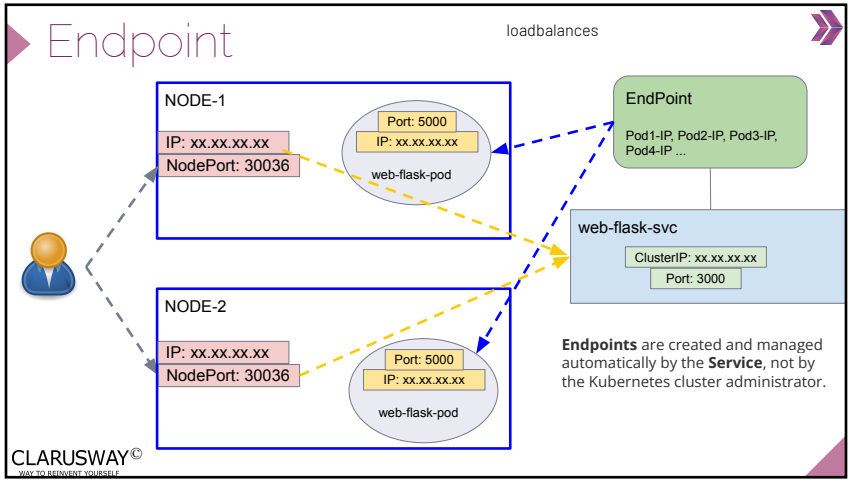
CLARUSWAY©
WAY TO REINVENT YOURSELF

ClusterIP



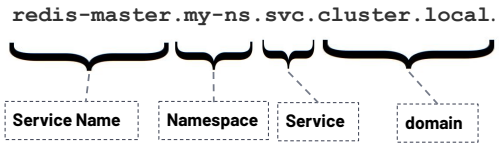
CLARUSWAY©
WAY TO REINVENT YOURSELF


NodePort
NODE
IP: xx.xx.xx.xx
NodePort: 30480
web-flask-svc
ClusterIP: xx.xx.xx.xx
Port: 3000
Port: 5000 (targetPort)
IP: xx.xx.xx.xx
web-flask-pod
Port: 5000 (targetPort)
IP: xx.xx.xx.xx
web-flask-pod
Port: 5000 (targetPort)
IP: xx.xx.xx.xx
web-flask-pod
apiVersion: v1
kind: Service
metadata:
name: web-flask-svc
labels:
app: web-flask
spec:
type: NodePort
selector:
app: web-flask
ports:
- nodePort: 30480
port: 3000
protocol: TCP
targetPort: 5000
CLARUSWAY©
WAY TO REINVENT YOURSELF


Endpoint
loadbalances
NODE-1
IP: xx.xx.xx.xx
NodePort: 30036
Port: 5000
IP: xx.xx.xx.xx
web-flask-pod
NODE-2
IP: xx.xx.xx.xx
NodePort: 30036
Port: 5000
IP: xx.xx.xx.xx
web-flask-pod
EndPoint
Pod1-IP, Pod2-IP, Pod3-IP, Pod4-IP ...
web-flask-svc
ClusterIP: xx.xx.xx.xx
Port: 3000
Endpoints are created and managed automatically by the Service, not by the Kubernetes cluster administrator.
CLARUSWAY©
WAY TO REINVENT YOURSELF

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

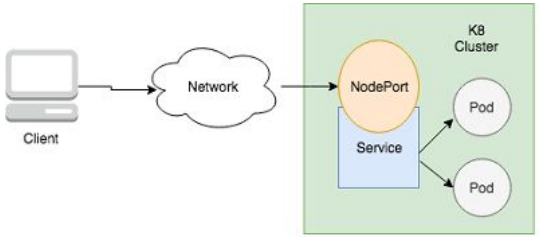


FQDN: fully qualified domain name

Service Types

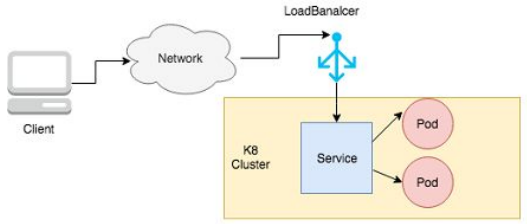
NodePort: Exposes the Service on each Node's IP at a static port (the NodePort). A ClusterIP Service, to which the NodePort Service **routes**, is automatically created. Port can either be **statically** defined, or **dynamically** taken from a range between 30000-32767.

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

LoadBalancer: Exposes the Service externally using a cloud provider's load balancer. The external load balancer routes to the automatically created NodePort and ClusterIP Services.

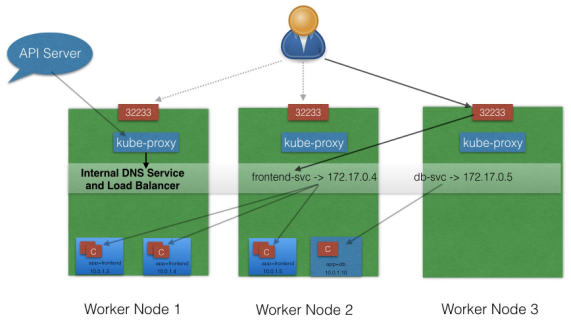


Service Types

NodePort:

db-svc : ClusterIP
frontend-svc : NodePort

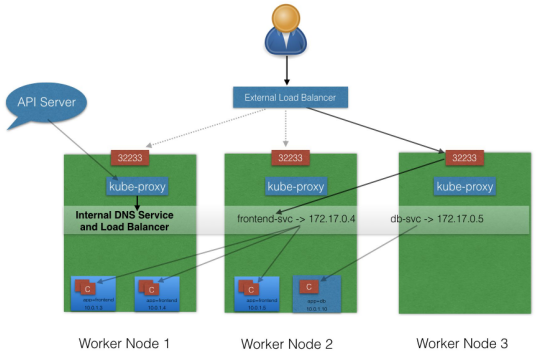
The end-user connects to any worker node on the specified high-port, which proxies the request internally to the ClusterIP of the Service, then the request is forwarded to the applications running inside the cluster. Let's not forget that the Service is load balancing such requests, and only forwards the request to one of the Pods running the desired application. To manage access to multiple application Services from the external world, administrators can configure a reverse proxy - an ingress, and define rules that target specific Services within the cluster.



Service Types

LoadBalancer:

- NodePort and ClusterIP are automatically created, and the external load balancer will route to them
- The Service is exposed at a static port on each worker node
- The Service is exposed externally using the underlying cloud provider's load balancer feature.



Service Types

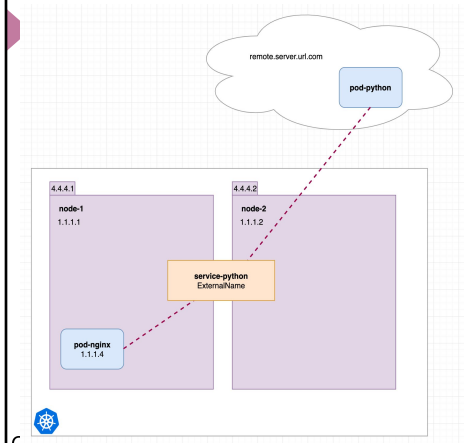
ExternalName is a special *ServiceType*, that has no Selectors and does not define any endpoints.

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.

CNAME : Canonical Name Record or Alias Record




```yaml
kind: Service
apiVersion: v1
metadata:
 name: service-python
spec:
 type: ExternalName
 externalName: remote.server.url.com
 ports:
 - port: 3000
   protocol: TCP
   targetPort: 443
```

Now **pod-nginx** can simply connect to `http://service-python:3000`, just like with a ClusterIP service.

Service Types

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

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

Example Usecases:

When you expose and stick your traffic directly to your subdomain