# Accessing K8S pods

This is some note, nothing serious. **DO NOT** cite from this article.

Download this note as a PDF

# 1. Forwarding a Local Network Port to a Port in The Pod

When you want to talk to a specific pod without going through a service (for debugging or other reasons), Kubernetes allows you to configure port forwarding to the pod.

This is done through the `kubectl port-forward` command. The following command will forward your machine's local port `8888` to port `8080` of your e.g `kubia-manual` pod.

**Example:**

```
$ kubectl port-forward kubia-manual 8888:8080
```

**Output:**

```
... Forwarding from 127.0.0.1:8888 -> 8080
... Forwarding from [::1]:8888 -> 8080
```

## 1.1. Connecting to The Pod Through the Port Forwarder

In a different terminal, you can now use curl to send an HTTP request to your pod through the `kubectl port-forward` proxy running on `localhost:8888`.

**Example:**

```
$ curl localhost:8888
```

**Output:**

```
You've hit kubia-manual
```

# 2. Service Object

Each pod gets its own IP address, but this address is internal to the cluster and isn't accessible from outside of it. To make the pod accessible from the outside, you'll expose it through a Service object. You'll create a special service of type `LoadBalancer`, because if you create a regular service (a `ClusterIP` service), like the pod, it would also only be accessible from inside the cluster. By creating a `LoadBalancer` type service, an external load balancer will be created and you can connect to the pod through the load balancer's public IP.

# 2.1. Creating a Service Object

To create the service, you’ll tell Kubernetes to expose the ReplicationController you created:

### Using YAML file

#### Manifest:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  ports:
  - port: 80 # service's port
    targetPort: 8080 # the forward-to port by service
  selector: # all pods labeled `kubia` will follow/select this service
    app: kubia
```

#### Apply the service:

```
$ kubectl create -f kubia-srv.yaml
```

### Using kubectl CLI options

#### Template:

```
$ kubectl expose rc <rep-controller-name> --type=LoadBalancer --name <lb-name>
```

#### Expose:

```
$ kubectl expose rc kubia --type=LoadBalancer --name kubia-http
```

#### Output:

```
service "kubia-http" exposed
```

#### Remotely Executing Commands in Running Containers:

- You’ll also need to obtain the cluster IP of your service (using `kubectl get svc`, for example)

```
$ kubectl exec kubia-7nog1 -- curl -s http://10.111.249.153
```

#### Output:

```
You've hit kubia-gzwli
```

# 2.2. Session Affinity on the Service

If you execute the same command a few more times, you should hit a different pod with every invocation, because the service proxy normally forwards each connection to a randomly selected backing pod, even if the connections are coming from the same client.

If, on the other hand, you want all requests made by a certain client to be redirected to the same pod every time, you can set the service's `sessionAffinity` property to `ClientIP` (instead of `None`, which is the default), as shown in the following listing.

### Service with `ClientIP` Session Affinity Manifest

```
apiVersion: v1
kind: Service
spec:
  sessionAffinity: ClientIP
  ...
```

- Kubernetes supports only two types of service session affinity: `None` and `ClientIP`.
- Kubernetes services don't operate at the HTTP level. Services deal with TCP and UDP packets and don't care about the payload they carry. Because cookies are a construct of the HTTP protocol, services don't know about them, which explains why session affinity cannot be based on cookies.

# 2.3. Exposing Multiple Ports in the Same Service

### Manifest

```
apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  ports:
  - name: http
    port: 80
    targetPort: 8080
  - name: https
    port: 443
    targetPort: 8443
  selector:
    app: kubia
```

# 2.4. Using Named Ports

You can give a name to each pod's port and refer to it by name in the service spec.

### Specifying port names in a pod definition Manifest:

```
kind: Pod
spec:
  containers:
  - name: kubia
    ports:
    - name: http
      containerPort: 8080
    - name: https
```

```
        containerPort: 8443
```

#### Referring to named ports in a service Manifest:

```
apiVersion: v1
kind: Service
spec:
  ports:
  - name: http
    port: 80
    targetPort: http
  - name: https
    port: 443
    targetPort: https
```

# 3. Connecting to services living outside the cluster

Instead of having the service redirect connections to pods in the cluster, you want it to redirect to external IP(s) and port(s).

This allows you to take advantage of both service load balancing and service discovery. Client pods running in the cluster can connect to the external service like they connect to internal services.

## 3.1. Service Endpoints

Services don't link to pods directly. Instead, a resource sits in between—the Endpoints resource. You may have already noticed endpoints if you used the `kubectl describe` command on your service.

#### Full details of a service:

```
$ kubectl describe svc kubia
```

#### Output:

```
Name:               kubia
Namespace:          default
Labels:             <none>
Selector:           app=kubia
Type:               ClusterIP
IP:                 10.111.249.153
Port:               <unset> 80/TCP
Endpoints:          10.108.1.4:8080,10.108.2.5:8080,10.108.2.6:8080
Session             Affinity: None
No events.
```

An Endpoints resource (yes, plural) is a list of IP addresses and ports exposing a service. The Endpoints resource is like any other Kubernetes resource, so you can display its basic info with kubectl get.

```
$ kubectl get endpoints kubia
```

**Output:**

```
NAME     ENDPOINTS                                             AGE
kubia    10.108.1.4:8080,10.108.2.5:8080,10.108.2.6:8080      1h
```

### Manually Configuring Service Endpoints

- having the service's endpoints decoupled from the service allows them to be configured and updated manually.
- If you create a service without a pod selector, Kubernetes won't even create the Endpoints resource
  - after all, without a selector, it can't know which pods to include in the service
- To create a service with manually managed endpoints, you need to create both a Service and an Endpoints resource

**A service without a pod selector:** `external-service.yaml`

```yaml
apiVersion: v1
kind: Service
metadata:
  name: external-service # must match the endpoints name
spec:
  ports:
  - port: 80
```

- Endpoints are a separate resource and not an attribute of a service
- Because you created the service without a selector, the corresponding Endpoints resource hasn't been created automatically

**A manually created Endpoints resource:** `external-service-endpoints.yaml`

```yaml
apiVersion: v1
kind: Endpoints
metadata:
  name: external-service # must match the service name
subsets:
  - addresses:
    - ip: 11.11.11.11
    - ip: 22.22.22.22
    ports:
    - port: 80 # target port of endpoints
```

# 4. Exposing services to external clients

Few ways to make a service accessible externally.

- `NodePort` Service Type
  - Each cluster node opens a port on the node itself (hence the name) and redirects traffic received on that port to the underlying service.
  - The service isn't accessible only at the internal cluster IP and port, but also through a dedicated port

on all nodes.

- `LoadBalancer` Service Type, an extention of `NodePort` type
  - This makes the service accessible through a dedicated load balancer, provisioned from the cloud infrastructure Kubernetes is running on.
  - The load balancer redirects traffic to the node port across all the nodes. Clients connect to the service through the load balancer's IP.
- Create `Ingress` Resource, radically different mechanism for exposing multiple services through a single IP address
  - It operates at the HTTP level (network layer 7) and can thus offer more features than layer 4 services can

## 4.1. Using a NodePort service

By creating a `NodePort` service, you make Kubernetes reserve a port on all its nodes (the same port number is used across all of them) and forward incoming connections to the pods that are part of the service.

This is similar to a regular service (their actual type is `ClusterIP`), but a `NodePort` service can be accessed not only through the service's internal cluster IP, but also through any node's IP and the reserved node port.

This will make more sense when you try interacting with a `NodePort` service.

#### A NodePort service definition: `kubia-svc-nodeport.yaml`

```yaml
apiVersion: v1
kind: Service
metadata:
  name: kubia-nodeport
spec:
  type: NodePort # service type
  ports:
  - port: 80 # service's internal cluster IP port
    targetPort: 8080 # target port of the backing pods
    nodePort: 30123 # service will listen on port 30123, each cluster nodes
  selector:
    app: kubia
```

#### Examine the NodePort Service:

```
$ kubectl get svc kubia-nodeport
```

#### Output:

```
NAME             CLUSTER-IP       EXTERNAL-IP    PORT(S)         AGE
kubia-nodeport   10.111.254.223   <nodes>        80:30123/TCP    2m
```

`EXTERNAL-IP` column shows `<nodes>`, indicating the service is accessible through the IP address of any cluster node. The `PORT(S)` column shows both the internal port of the cluster IP (`80`) and the node port (`30123`). The service is accessible at the following addresses:

- `10.11.254.223:80`
- `<1st node's IP>:30123`

- `<2nd node’s IP>:30123`
- and so on

### Using JSONPath to get the IPs of all your nodes

You can find the IP in the JSON or YAML descriptors of the nodes. But instead of sifting through the relatively large JSON, you can tell `kubectl` to print out only the node IP instead of the whole service definition

```
$ kubectl get nodes -o \
  jsonpath='{.items[*].status.addresses[?(@.type=="ExternalIP")].address}'
```

### Output:

```
130.211.97.55 130.211.99.206
```

Once you know the IPs of your nodes, you can try accessing your service through them.

```
$ curl http://130.211.97.55:30123
```

### Output:

```
You've hit kubia-ym8or
```

```
$ curl http://130.211.99.206:30123
```

### Output:

```
You've hit kubia-xueq1
```

## 4.2. Exposing a service through an external load balancer

Kubernetes clusters running on cloud providers usually support the automatic provision of a load balancer from the cloud infrastructure. All you need to do is set the service’s type to `LoadBalancer` instead of `NodePort`. The load balancer will have its own unique, publicly accessible IP address and will redirect all connections to your service. You can thus access your service through the load balancer’s IP address.

If Kubernetes is running in an environment that doesn’t support `LoadBalancer` services, the load balancer will not be provisioned, but the service will still behave like a `NodePort` service. That’s because a `LoadBalancer` service is an extension of a `NodePort` service. You’ll run this example on Google Kubernetes Engine, which supports `LoadBalancer` services. Minikube doesn’t, at least not as of this writing.

#### Creating a Loadbalancer Service

**A LoadBalancer-type service:** `kubia-svc-loadbalancer.yaml`

```
apiVersion: v1
kind: Service
metadata:
```

```
  name: kubia-loadbalancer
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: kubia
```

- The service type is set to LoadBalancer instead of NodePort. You're not specifying a specific node port, although you could (you're letting Kubernetes choose one instead).

### Connecting to the Service Through the Load Balancer

```
$ kubectl get svc kubia-loadbalancer
```

### Output

```
NAME                 CLUSTER-IP       EXTERNAL-IP      PORT(S)       AGE
kubia-loadbalancer  10.111.241.153   130.211.53.173   80:32143/TCP  1m
```

**Session affinity and web browsers**

Because your service is now exposed externally, you may try accessing it with your web browser. You'll see something that may strike you as odd—the browser will hit the exact same pod every time. Did the service's session affinity change in the meantime? With `kubectl explain`, you can double-check that the service's session affinity is still set to `None`, so why don't different browser requests hit different pods, as is the case when using `curl`?

Let me explain what's happening. The browser is using keep-alive connections and sends all its requests through a single connection, whereas `curl` opens a new connection every time. Services work at the connection level, so when a connection to a service is first opened, a random pod is selected and then all network packets belonging to that connection are all sent to that single pod. Even if session affinity is set to `None`, users will always hit the same pod (until the connection is closed).

## 4.3. Understanding the peculiarities of external connections

You must be aware of several things related to externally originating connections to services.

### Understanding and Preventing Unnecessary Network Hops

When an external client connects to a service through the node port (this also includes cases when it goes through the load balancer first), the randomly chosen pod may or may not be running on the same node that received the connection. An additional network hop is required to reach the pod, but this may not always be desirable.

You can prevent this additional hop by configuring the service to redirect external traffic only to pods running on the node that received the connection. This is done by setting the `externalTrafficPolicy` field in the service's `spec` section

```
spec:
  externalTrafficPolicy: Local
  ...
```

#### Being Aware of the non-preservation of the Client’s IP

Usually, when clients inside the cluster connect to a service, the pods backing the service can obtain the client’s IP address. But when the connection is received through a node port, the packets' source IP is changed, because Source Network Address Translation (SNAT) is performed on the packets.

The backing pod can’t see the actual client’s IP, which may be a problem for some applications that need to know the client’s IP. In the case of a web server, for example, this means the access log won’t show the browser’s IP.

The `Local` external traffic policy described in the previous section affects the preservation of the client’s IP, because there’s no additional hop between the node receiving the connection and the node hosting the target pod (SNAT isn’t performed).

# 5. Exposing services externally through an Ingress resource

You must be aware of several things related to externally originating connections to services.

#### Understanding Why Ingresses are Needed

- each LoadBalancer service requires its own load balancer with its own public IP address, whereas an Ingress only requires one, even when providing access to dozens of services
- When a client sends an HTTP request to the Ingress, the host and path in the request determine which service the request is forwarded to
- Ingresses operate at the application layer of the network stack (HTTP) and can provide features such as cookie-based session affinity and the like, which services can’t

#### Understanding that an Ingress Controller is Required

To make Ingress resources work, an Ingress controller needs to be running in the cluster.

## 5.1. Creating an Ingress resource

#### An Ingress resource definition: `kubia-ingress.yaml`

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kubia
spec:
  rules:
  - host: kubia.example.com #  maps the domain name to your service
  http:
    paths:
    - path: /                    # All requests will be sent to port 80
      backend:                   # of the kubia-nodeport service.
        serviceName: kubia-nodeport  #
        servicePort: 80              #
```

This defines an Ingress with a single rule, which makes sure all HTTP requests received by the Ingress controller, in which the host `kubia.example.com` is requested, will be sent to the `kubia-nodeport` service on port `80`.

> Ingress controllers on cloud providers (in GKE, for example) require the Ingress to point to a `NodePort` service. But that's not a requirement of Kubernetes itself.

## 5.2. Accessing the service through the Ingress

To access your service through http://kubia.example.com, you'll need to make sure the domain name resolves to the IP of the Ingress controller

**Obtaining the ip Address of the Ingress:**

```
$ kubectl get ingresses
```

**Output:**

```
NAME     HOSTS               ADDRESS          PORTS      AGE
kubia    kubia.example.com   192.168.99.100   80         29m
```

> When running on cloud providers, the address may take time to appear, because the Ingress controller provisions a load balancer behind the scenes.

- The IP is shown in the `ADDRESS` column.

> Once you know the IP, you can then either configure your DNS servers to resolve kubia.example.com to that IP or you can setup hosts: add the following line to `/etc/hosts` (or `C:\windows\system32\drivers\etc\hosts` on Windows):
>
> `/etc/hosts`
>
> ```
> 192.168.99.100      kubia.example.com
> ```
>
> So you can access the service at http://kubia.example.com using a browser or `curl`

### 5.2.1. Understading how Ingress work

- The client first performed a DNS lookup of kubia.example.com, and the DNS server (or the local operating system) returned the IP of the Ingress controller
- The client then sent an HTTP request to the Ingress controller and specified `kubia.example.com` in the `Host` header
  - From that header, the controller determined which service the client is trying to access, looked up the pod IPs through the Endpoints object associated with the service, and forwarded the client's request to one of the pods.
- The Ingress controller don't forward the request to the service
  - It only use it to select a pod
  - Most, if not all, controllers work like this

## 5.3. Exposing multiple services through the same Ingress

Both `rules` and `paths` are arrays, so they can contain multiple items

Both `rules` and `paths` are arrays, so they can contain multiple items