

# 5

## *Services: enabling clients to discover and talk to pods*

---

### **This chapter covers**

- Creating Service resources to expose a group of pods at a single address
- Discovering services in the cluster
- Exposing services to external clients
- Connecting to external services from inside the cluster
- Controlling whether a pod is ready to be part of the service or not
- Troubleshooting services

You've learned about pods and how to deploy them through ReplicaSets and similar resources to ensure they keep running. Although certain pods can do their work independently of an external stimulus, many applications these days are meant to respond to external requests. For example, in the case of microservices, pods will usually respond to HTTP requests coming either from other pods inside the cluster or from clients outside the cluster.

Pods need a way of finding other pods if they want to consume the services they provide. Unlike in the non-Kubernetes world, where a sysadmin would configure

each client app by specifying the exact IP address or hostname of the server providing the service in the client's configuration files, doing the same in Kubernetes wouldn't work, because

- *Pods are ephemeral*—They may come and go at any time, whether it's because a pod is removed from a node to make room for other pods, because someone scaled down the number of pods, or because a cluster node has failed.
- *Kubernetes assigns an IP address to a pod after the pod has been scheduled to a node and before it's started*—Clients thus can't know the IP address of the server pod up front.
- *Horizontal scaling means multiple pods may provide the same service*—Each of those pods has its own IP address. Clients shouldn't care how many pods are backing the service and what their IPs are. They shouldn't have to keep a list of all the individual IPs of pods. Instead, all those pods should be accessible through a single IP address.

To solve these problems, Kubernetes also provides another resource type—Services—that we'll discuss in this chapter.

## 5.1 Introducing services

A Kubernetes Service is a resource you create to make a single, constant point of entry to a group of pods providing the same service. Each service has an IP address and port that never change while the service exists. Clients can open connections to that IP and port, and those connections are then routed to one of the pods backing that service. This way, clients of a service don't need to know the location of individual pods providing the service, allowing those pods to be moved around the cluster at any time.

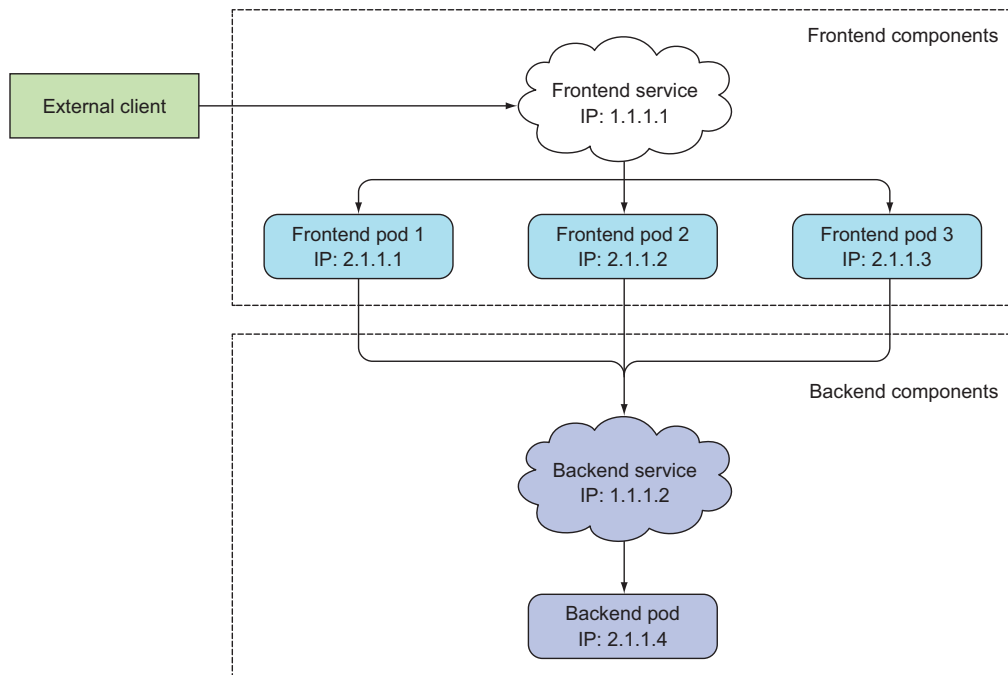
### EXPLAINING SERVICES WITH AN EXAMPLE

Let's revisit the example where you have a frontend web server and a backend database server. There may be multiple pods that all act as the frontend, but there may only be a single backend database pod. You need to solve two problems to make the system function:

- External clients need to connect to the frontend pods without caring if there's only a single web server or hundreds.
- The frontend pods need to connect to the backend database. Because the database runs inside a pod, it may be moved around the cluster over time, causing its IP address to change. You don't want to reconfigure the frontend pods every time the backend database is moved.

By creating a service for the frontend pods and configuring it to be accessible from outside the cluster, you expose a single, constant IP address through which external clients can connect to the pods. Similarly, by also creating a service for the backend pod, you create a stable address for the backend pod. The service address doesn't

change even if the pod's IP address changes. Additionally, by creating the service, you also enable the frontend pods to easily find the backend service by its name through either environment variables or DNS. All the components of your system (the two services, the two sets of pods backing those services, and the interdependencies between them) are shown in figure 5.1.



**Figure 5.1** Both internal and external clients usually connect to pods through services.

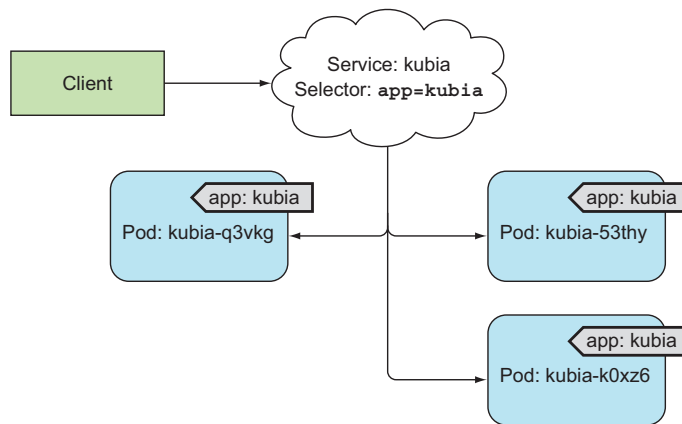
You now understand the basic idea behind services. Now, let's dig deeper by first seeing how they can be created.

### 5.1.1 Creating services

As you've seen, a service can be backed by more than one pod. Connections to the service are load-balanced across all the backing pods. But how exactly do you define which pods are part of the service and which aren't?

You probably remember label selectors and how they're used in ReplicationControllers and other pod controllers to specify which pods belong to the same set. The same mechanism is used by services in the same way, as you can see in figure 5.2.

In the previous chapter, you created a ReplicationController which then ran three instances of the pod containing the Node.js app. Create the ReplicationController again and verify three pod instances are up and running. After that, you'll create a Service for those three pods.



**Figure 5.2** Label selectors determine which pods belong to the Service.

### CREATING A SERVICE THROUGH KUBECTL EXPOSE

The easiest way to create a service is through `kubectl expose`, which you’ve already used in chapter 2 to expose the ReplicationController you created earlier. The `expose` command created a Service resource with the same pod selector as the one used by the ReplicationController, thereby exposing all its pods through a single IP address and port.

Now, instead of using the `expose` command, you’ll create a service manually by posting a YAML to the Kubernetes API server.

### CREATING A SERVICE THROUGH A YAML DESCRIPTOR

Create a file called `kubernia-svc.yaml` with the following listing’s contents.

#### Listing 5.1 A definition of a service: `kubernia-svc.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: kubernia
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: kubernia
```

The port this service will be available on

The container port the service will forward to

All pods with the `app=kubernia` label will be part of this service.

You’re defining a service called `kubernia`, which will accept connections on port 80 and route each connection to port 8080 of one of the pods matching the `app=kubernia` label selector.

Go ahead and create the service by posting the file using `kubectl create`.

**EXAMINING YOUR NEW SERVICE**

After posting the YAML, you can list all Service resources in your namespace and see that an internal cluster IP has been assigned to your service:

```
$ kubectl get svc
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	10.111.240.1	<none>	443/TCP	30d
kubia	10.111.249.153	<none>	80/TCP	6m

Here's your service.

The list shows that the IP address assigned to the service is 10.111.249.153. Because this is the cluster IP, it's only accessible from inside the cluster. The primary purpose of services is exposing groups of pods to other pods in the cluster, but you'll usually also want to expose services externally. You'll see how to do that later. For now, let's use your service from inside the cluster and see what it does.

**TESTING YOUR SERVICE FROM WITHIN THE CLUSTER**

You can send requests to your service from within the cluster in a few ways:

- The obvious way is to create a pod that will send the request to the service's cluster IP and log the response. You can then examine the pod's log to see what the service's response was.
- You can ssh into one of the Kubernetes nodes and use the `curl` command.
- You can execute the `curl` command inside one of your existing pods through the `kubectl exec` command.

Let's go for the last option, so you also learn how to run commands in existing pods.

**REMOTELY EXECUTING COMMANDS IN RUNNING CONTAINERS**

The `kubectl exec` command allows you to remotely run arbitrary commands inside an existing container of a pod. This comes in handy when you want to examine the contents, state, and/or environment of a container. List the pods with the `kubectl get pods` command and choose one as your target for the `exec` command (in the following example, I've chosen the `kubia-7nog1` pod as the target). You'll also need to obtain the cluster IP of your service (using `kubectl get svc`, for example). When running the following commands yourself, be sure to replace the pod name and the service IP with your own:

```
$ kubectl exec kubia-7nog1 -- curl -s http://10.111.249.153
You've hit kubia-gzwli
```

If you've used `ssh` to execute commands on a remote system before, you'll recognize that `kubectl exec` isn't much different.

### Why the double dash?

The double dash (--) in the command signals the end of command options for `kubectl`. Everything after the double dash is the command that should be executed inside the pod. Using the double dash isn't necessary if the command has no arguments that start with a dash. But in your case, if you don't use the double dash there, the `-s` option would be interpreted as an option for `kubectl exec` and would result in the following strange and highly misleading error:

```
$ kubectl exec kubia-7nog1 curl -s http://10.111.249.153
The connection to the server 10.111.249.153 was refused - did you
specify the right host or port?
```

This has nothing to do with your service refusing the connection. It's because `kubectl` is not able to connect to an API server at 10.111.249.153 (the `-s` option is used to tell `kubectl` to connect to a different API server than the default).

Let's go over what transpired when you ran the command. Figure 5.3 shows the sequence of events. You instructed Kubernetes to execute the `curl` command inside the container of one of your pods. `Curl` sent an HTTP request to the service IP, which is backed by three pods. The Kubernetes service proxy intercepted the connection, selected a random pod among the three pods, and forwarded the request to it. `Node.js` running inside that pod then handled the request and returned an HTTP response containing the pod's name. `Curl` then printed the response to the standard output, which was intercepted and printed to its standard output on your local machine by `kubectl`.

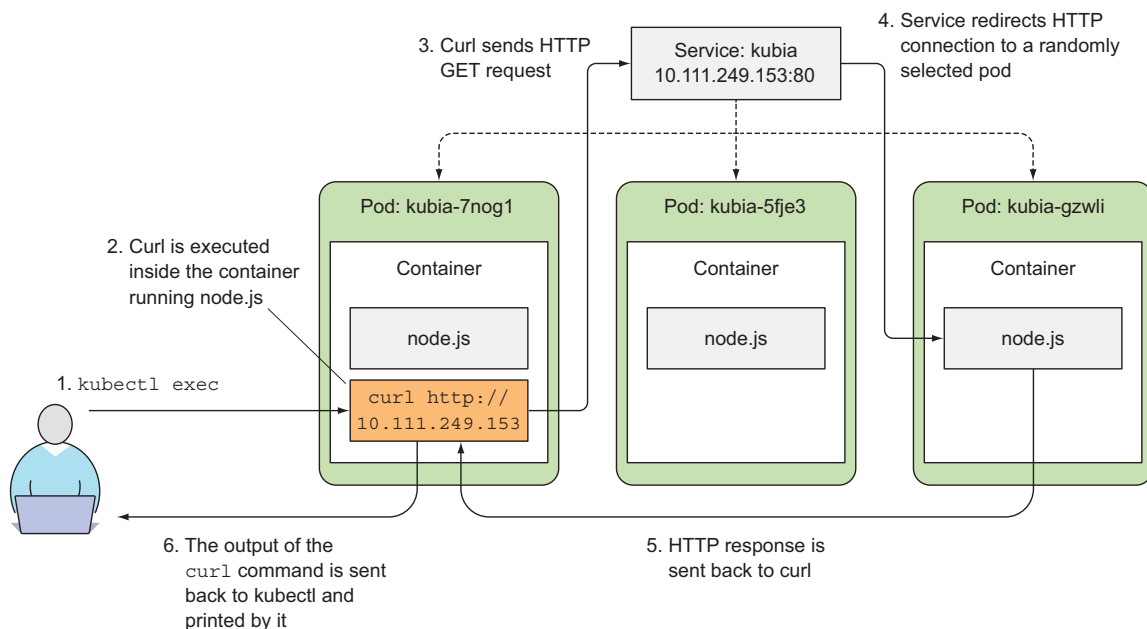


Figure 5.3 Using `kubectl exec` to test out a connection to the service by running `curl` in one of the pods

In the previous example, you executed the `curl` command as a separate process, but inside the pod's main container. This isn't much different from the actual main process in the container talking to the service.

#### CONFIGURING 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.

#### Listing 5.2 A example of a service with `ClientIP` session affinity configured

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

This makes the service proxy redirect all requests originating from the same client IP to the same pod. As an exercise, you can create an additional service with session affinity set to `ClientIP` and try sending requests to it.

Kubernetes supports only two types of service session affinity: `None` and `ClientIP`. You may be surprised it doesn't have a cookie-based session affinity option, but you need to understand that 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.

#### EXPOSING MULTIPLE PORTS IN THE SAME SERVICE

Your service exposes only a single port, but services can also support multiple ports. For example, if your pods listened on two ports—let's say 8080 for HTTP and 8443 for HTTPS—you could use a single service to forward both port 80 and 443 to the pod's ports 8080 and 8443. You don't need to create two different services in such cases. Using a single, multi-port service exposes all the service's ports through a single cluster IP.

**NOTE** When creating a service with multiple ports, you must specify a name for each port.

The spec for a multi-port service is shown in the following listing.

#### Listing 5.3 Specifying multiple ports in a service definition

```
apiVersion: v1
kind: Service
metadata:
  name: kubia
```

```
spec:
  ports:
    - name: http
      port: 80
      targetPort: 8080
    - name: https
      port: 443
      targetPort: 8443
  selector:
    app: kubia
```

**Port 80 is mapped to the pods' port 8080.**

**Port 443 is mapped to pods' port 8443.**

**The label selector always applies to the whole service.**

**NOTE** The label selector applies to the service as a whole—it can't be configured for each port individually. If you want different ports to map to different subsets of pods, you need to create two services.

Because your kubia pods don't listen on multiple ports, creating a multi-port service and a multi-port pod is left as an exercise to you.

### USING NAMED PORTS

In all these examples, you've referred to the target port by its number, but you can also give a name to each pod's port and refer to it by name in the service spec. This makes the service spec slightly clearer, especially if the port numbers aren't well-known.

For example, suppose your pod defines names for its ports as shown in the following listing.

#### Listing 5.4 Specifying port names in a pod definition

```
kind: Pod
spec:
  containers:
    - name: kubia
      ports:
        - name: http
          containerPort: 8080
        - name: https
          containerPort: 8443
```

**Container's port 8080 is called http**

**Port 8443 is called https.**

You can then refer to those ports by name in the service spec, as shown in the following listing.

#### Listing 5.5 Referring to named ports in a service

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

**Port 80 is mapped to the container's port called http.**

**Port 443 is mapped to the container's port, whose name is https.**



But why should you even bother with naming ports? The biggest benefit of doing so is that it enables you to change port numbers later without having to change the service spec. Your pod currently uses port 8080 for http, but what if you later decide you'd like to move that to port 80?

If you're using named ports, all you need to do is change the port number in the pod spec (while keeping the port's name unchanged). As you spin up pods with the new ports, client connections will be forwarded to the appropriate port numbers, depending on the pod receiving the connection (port 8080 on old pods and port 80 on the new ones).

### 5.1.2 *Discovering services*

By creating a service, you now have a single and stable IP address and port that you can hit to access your pods. This address will remain unchanged throughout the whole lifetime of the service. Pods behind this service may come and go, their IPs may change, their number can go up or down, but they'll always be accessible through the service's single and constant IP address.

But how do the client pods know the IP and port of a service? Do you need to create the service first, then manually look up its IP address and pass the IP to the configuration options of the client pod? Not really. Kubernetes also provides ways for client pods to discover a service's IP and port.

#### DISCOVERING SERVICES THROUGH ENVIRONMENT VARIABLES

When a pod is started, Kubernetes initializes a set of environment variables pointing to each service that exists at that moment. If you create the service before creating the client pods, processes in those pods can get the IP address and port of the service by inspecting their environment variables.

Let's see what those environment variables look like by examining the environment of one of your running pods. You've already learned that you can use the `kubectl exec` command to run a command in the pod, but because you created the service only after your pods had been created, the environment variables for the service couldn't have been set yet. You'll need to address that first.

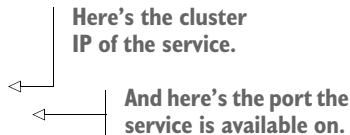
Before you can see environment variables for your service, you first need to delete all the pods and let the ReplicationController create new ones. You may remember you can delete all pods without specifying their names like this:

```
$ kubectl delete po --all
pod "kubia-7nog1" deleted
pod "kubia-bf50t" deleted
pod "kubia-gzwli" deleted
```

Now you can list the new pods (I'm sure you know how to do that) and pick one as your target for the `kubectl exec` command. Once you've selected your target pod, you can list environment variables by running the `env` command inside the container, as shown in the following listing.

**Listing 5.6 Service-related environment variables in a container**

```
$ kubectl exec kubia-3inly env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=kubia-3inly
KUBERNETES_SERVICE_HOST=10.111.240.1
KUBERNETES_SERVICE_PORT=443
...
KUBIA_SERVICE_HOST=10.111.249.153
KUBIA_SERVICE_PORT=80
...
```



Here's the cluster IP of the service.

And here's the port the service is available on.

Two services are defined in your cluster: the `kubernetes` and the `kubia` service (you saw this earlier with the `kubectl get svc` command); consequently, two sets of service-related environment variables are in the list. Among the variables that pertain to the `kubia` service you created at the beginning of the chapter, you'll see the `KUBIA_SERVICE_HOST` and the `KUBIA_SERVICE_PORT` environment variables, which hold the IP address and port of the `kubia` service, respectively.

Turning back to the frontend-backend example we started this chapter with, when you have a frontend pod that requires the use of a backend database server pod, you can expose the backend pod through a service called `backend-database` and then have the frontend pod look up its IP address and port through the environment variables `BACKEND_DATABASE_SERVICE_HOST` and `BACKEND_DATABASE_SERVICE_PORT`.

**NOTE** Dashes in the service name are converted to underscores and all letters are uppercased when the service name is used as the prefix in the environment variable's name.

Environment variables are one way of looking up the IP and port of a service, but isn't this usually the domain of DNS? Why doesn't Kubernetes include a DNS server and allow you to look up service IPs through DNS instead? As it turns out, it does!

**DISCOVERING SERVICES THROUGH DNS**

Remember in chapter 3 when you listed pods in the `kube-system` namespace? One of the pods was called `kube-dns`. The `kube-system` namespace also includes a corresponding service with the same name.

As the name suggests, the pod runs a DNS server, which all other pods running in the cluster are automatically configured to use (Kubernetes does that by modifying each container's `/etc/resolv.conf` file). Any DNS query performed by a process running in a pod will be handled by Kubernetes' own DNS server, which knows all the services running in your system.

**NOTE** Whether a pod uses the internal DNS server or not is configurable through the `dnsPolicy` property in each pod's spec.

Each service gets a DNS entry in the internal DNS server, and client pods that know the name of the service can access it through its fully qualified domain name (FQDN) instead of resorting to environment variables.

**CONNECTING TO THE SERVICE THROUGH ITS FQDN**

To revisit the frontend-backend example, a frontend pod can connect to the backend-database service by opening a connection to the following FQDN:

```
backend-database.default.svc.cluster.local
```

backend-database corresponds to the service name, default stands for the namespace the service is defined in, and svc.cluster.local is a configurable cluster domain suffix used in all cluster local service names.

**NOTE** The client must still know the service's port number. If the service is using a standard port (for example, 80 for HTTP or 5432 for Postgres), that shouldn't be a problem. If not, the client can get the port number from the environment variable.

Connecting to a service can be even simpler than that. You can omit the svc.cluster.local suffix and even the namespace, when the frontend pod is in the same namespace as the database pod. You can thus refer to the service simply as backend-database. That's incredibly simple, right?

Let's try this. You'll try to access the kuberneta service through its FQDN instead of its IP. Again, you'll need to do that inside an existing pod. You already know how to use `kubectl exec` to run a single command in a pod's container, but this time, instead of running the `curl` command directly, you'll run the `bash` shell instead, so you can then run multiple commands in the container. This is similar to what you did in chapter 2 when you entered the container you ran with Docker by using the `docker exec -it bash` command.

**RUNNING A SHELL IN A POD'S CONTAINER**

You can use the `kubectl exec` command to run `bash` (or any other shell) inside a pod's container. This way you're free to explore the container as long as you want, without having to perform a `kubectl exec` for every command you want to run.

**NOTE** The shell's binary executable must be available in the container image for this to work.

To use the shell properly, you need to pass the `-it` option to `kubectl exec`:

```
$ kubectl exec -it kubia-3inly bash
root@kubia-3inly:/#
```

You're now inside the container. You can use the `curl` command to access the kuberneta service in any of the following ways:

```
root@kubia-3inly:/# curl http://kubia.default.svc.cluster.local
You've hit kubia-5asi2

root@kubia-3inly:/# curl http://kubia.default
You've hit kubia-3inly
```

```
root@kubia-3inly:/# curl http://kubia
You've hit kubia-8awf3
```

You can hit your service by using the service's name as the hostname in the requested URL. You can omit the namespace and the `svc.cluster.local` suffix because of how the DNS resolver inside each pod's container is configured. Look at the `/etc/resolv.conf` file in the container and you'll understand:

```
root@kubia-3inly:/# cat /etc/resolv.conf
search default.svc.cluster.local svc.cluster.local cluster.local ...
```

### UNDERSTANDING WHY YOU CAN'T PING A SERVICE IP

One last thing before we move on. You know how to create services now, so you'll soon create your own. But what if, for whatever reason, you can't access your service?

You'll probably try to figure out what's wrong by entering an existing pod and trying to access the service like you did in the last example. Then, if you still can't access the service with a simple `curl` command, maybe you'll try to ping the service IP to see if it's up. Let's try that now:

```
root@kubia-3inly:/# ping kubia
PING kubia.default.svc.cluster.local (10.111.249.153): 56 data bytes
^C--- kubia.default.svc.cluster.local ping statistics ---
54 packets transmitted, 0 packets received, 100% packet loss
```

Hmm. `curl`-ing the service works, but pinging it doesn't. That's because the service's cluster IP is a virtual IP, and only has meaning when combined with the service port. We'll explain what that means and how services work in chapter 11. I wanted to mention that here because it's the first thing users do when they try to debug a broken service and it catches most of them off guard.

## 5.2 Connecting to services living outside the cluster

Up to now, we've talked about services backed by one or more pods running inside the cluster. But cases exist when you'd like to expose external services through the Kubernetes services feature. 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.

### 5.2.1 Introducing service endpoints

Before going into how to do this, let me first shed more light on services. 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, as shown in the following listing.

**Listing 5.7 Full details of a service displayed with `kubectl describe`**

```
$ kubectl describe svc kubia
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.
```

The service's pod selector is used to create the list of endpoints.

The list of pod IPs and ports that represent the endpoints of this service

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

Although the pod selector is defined in the service spec, it's not used directly when redirecting incoming connections. Instead, the selector is used to build a list of IPs and ports, which is then stored in the Endpoints resource. When a client connects to a service, the service proxy selects one of those IP and port pairs and redirects the incoming connection to the server listening at that location.

**5.2.2 Manually configuring service endpoints**

You may have probably realized this already, but 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). It's up to you to create the Endpoints resource to specify the list of endpoints for the service.

To create a service with manually managed endpoints, you need to create both a Service and an Endpoints resource.

**CREATING A SERVICE WITHOUT A SELECTOR**

You'll first create the YAML for the service itself, as shown in the following listing.

**Listing 5.8 A service without a pod selector: `external-service.yaml`**

```
apiVersion: v1
kind: Service
metadata:
  name: external-service
spec:
  ports:
  - port: 80
```

The name of the service must match the name of the Endpoints object (see next listing).

This service has no selector defined.

You're defining a service called `external-service` that will accept incoming connections on port 80. You didn't define a pod selector for the service.

#### CREATING AN ENDPOINTS RESOURCE FOR A SERVICE WITHOUT A SELECTOR

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, so it's up to you to create it. The following listing shows its YAML manifest.

**Listing 5.9** A manually created Endpoints resource: `external-service-endpoints.yaml`

```
apiVersion: v1
kind: Endpoints
metadata:
  name: external-service
subsets:
  - addresses:
    - ip: 11.11.11.11
    - ip: 22.22.22.22
    ports:
    - port: 80
```

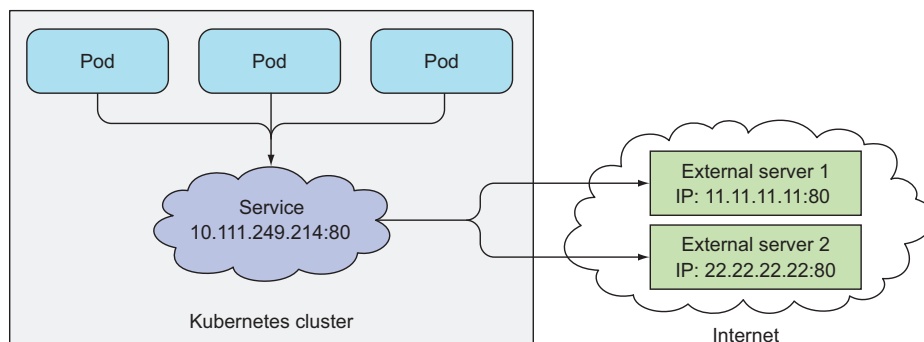
The name of the Endpoints object must match the name of the service (see previous listing).

The IPs of the endpoints that the service will forward connections to

The target port of the endpoints

The Endpoints object needs to have the same name as the service and contain the list of target IP addresses and ports for the service. After both the Service and the Endpoints resource are posted to the server, the service is ready to be used like any regular service with a pod selector. Containers created after the service is created will include the environment variables for the service, and all connections to its IP:port pair will be load balanced between the service's endpoints.

Figure 5.4 shows three pods connecting to the service with external endpoints.



**Figure 5.4** Pods consuming a service with two external endpoints.

If you later decide to migrate the external service to pods running inside Kubernetes, you can add a selector to the service, thereby making its Endpoints managed automatically. The same is also true in reverse—by removing the selector from a Service,

Kubernetes stops updating its Endpoints. This means a service IP address can remain constant while the actual implementation of the service is changed.

### 5.2.3 Creating an alias for an external service

Instead of exposing an external service by manually configuring the service's Endpoints, a simpler method allows you to refer to an external service by its fully qualified domain name (FQDN).

#### CREATING AN ExternalName SERVICE

To create a service that serves as an alias for an external service, you create a Service resource with the `type` field set to `ExternalName`. For example, let's imagine there's a public API available at [api.somecompany.com](https://api.somecompany.com). You can define a service that points to it as shown in the following listing.

**Listing 5.10** An `ExternalName`-type service: `external-service-externalname.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: external-service
spec:
  type: ExternalName
  externalName: someapi.somecompany.com
  ports:
    - port: 80
```



After the service is created, pods can connect to the external service through the `external-service.default.svc.cluster.local` domain name (or even `external-service`) instead of using the service's actual FQDN. This hides the actual service name and its location from pods consuming the service, allowing you to modify the service definition and point it to a different service any time later, by only changing the `externalName` attribute or by changing the type back to `ClusterIP` and creating an Endpoints object for the service—either manually or by specifying a label selector on the service and having it created automatically.

`ExternalName` services are implemented solely at the DNS level—a simple `CNAME` DNS record is created for the service. Therefore, clients connecting to the service will connect to the external service directly, bypassing the service proxy completely. For this reason, these types of services don't even get a cluster IP.

**NOTE** A `CNAME` record points to a fully qualified domain name instead of a numeric IP address.

## 5.3 Exposing services to external clients

Up to now, we've only talked about how services can be consumed by pods from inside the cluster. But you'll also want to expose certain services, such as frontend web servers, to the outside, so external clients can access them, as depicted in figure 5.5.

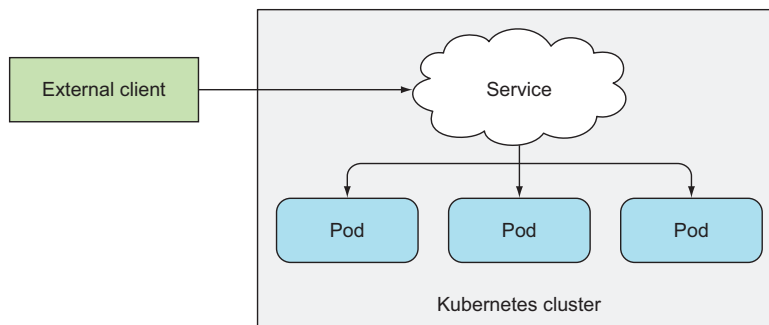


Figure 5.5 Exposing a service to external clients

You have a few ways to make a service accessible externally:

- *Setting the service type to NodePort*—For a NodePort service, 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.
- *Setting the service type to LoadBalancer, an extension of the 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.
- *Creating an Ingress resource, a 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. We’ll explain Ingress resources in section 5.4.

### 5.3.1 Using a NodePort service

The first method of exposing a set of pods to external clients is by creating a service and setting its type to NodePort. 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.

#### CREATING A NODEPORT SERVICE

You’ll now create a NodePort service to see how you can use it. The following listing shows the YAML for the service.



**Listing 5.11** A NodePort service definition: kubia-svc-nodeport.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: kubia-nodeport
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30123
  selector:
    app: kubia

```

Set the service type to NodePort.

This is the port of the service's internal cluster IP.

This is the target port of the backing pods.

The service will be accessible through port 30123 of each of your cluster nodes.

You set the type to NodePort and specify the node port this service should be bound to across all cluster nodes. Specifying the port isn't mandatory; Kubernetes will choose a random port if you omit it.

**NOTE** When you create the service in GKE, `kubectl` prints out a warning about having to configure firewall rules. We'll see how to do that soon.

#### EXAMINING YOUR NODEPORT SERVICE

Let's see the basic information of your service to learn more about it:

```

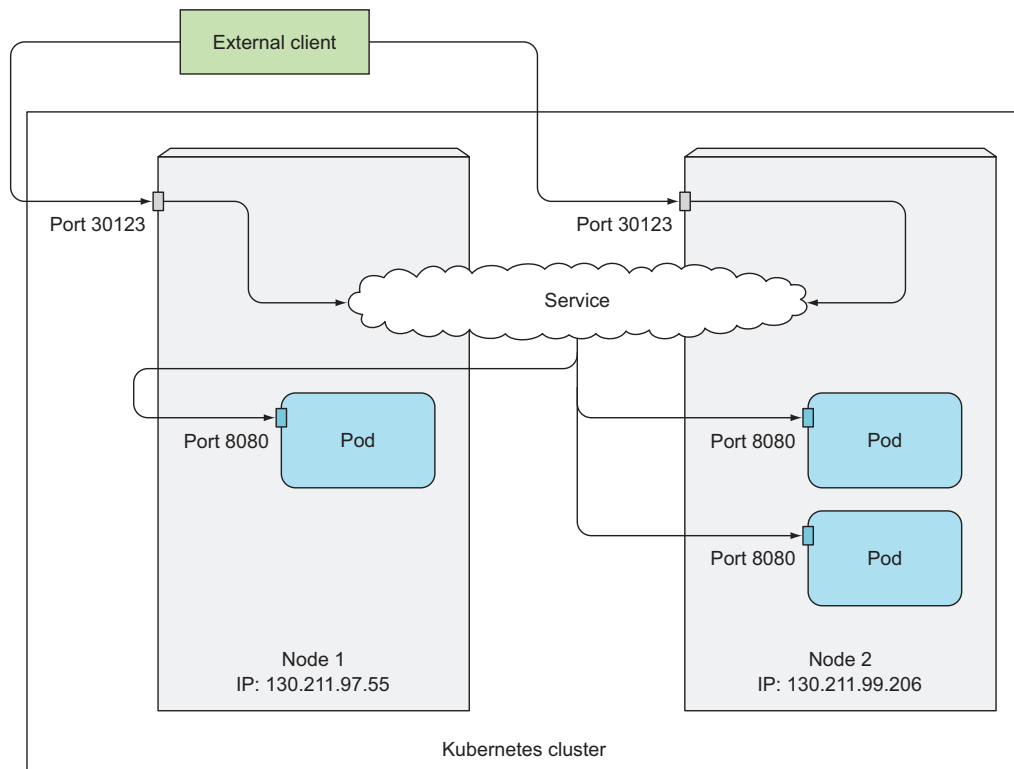
$ kubectl get svc kubia-nodeport
NAME                CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
kubia-nodeport      10.111.254.223  <nodes>          80:30123/TCP     2m

```

Look at the EXTERNAL-IP column. It 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:

- 10.111.254.223:80
- <1st node's IP>:30123
- <2nd node's IP>:30123, and so on.

Figure 5.6 shows your service exposed on port 30123 of both of your cluster nodes (this applies if you're running this on GKE; Minikube only has a single node, but the principle is the same). An incoming connection to one of those ports will be redirected to a randomly selected pod, which may or may not be the one running on the node the connection is being made to.



**Figure 5.6** An external client connecting to a NodePort service either through Node 1 or 2

A connection received on port 30123 of the first node might be forwarded either to the pod running on the first node or to one of the pods running on the second node.

#### CHANGING FIREWALL RULES TO LET EXTERNAL CLIENTS ACCESS OUR NODEPORT SERVICE

As I've mentioned previously, before you can access your service through the node port, you need to configure the Google Cloud Platform's firewalls to allow external connections to your nodes on that port. You'll do this now:

```
$ gcloud compute firewall-rules create kubia-svc-rule --allow=tcp:30123
Created [https://www.googleapis.com/compute/v1/projects/kubia-1295/global/firewalls/kubia-svc-rule].
NAME          NETWORK  SRC_RANGES  RULES      SRC_TAGS  TARGET_TAGS
kubia-svc-rule  default  0.0.0.0/0   tcp:30123
```

You can access your service through port 30123 of one of the node's IPs. But you need to figure out the IP of a node first. Refer to the sidebar on how to do that.

**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}'
130.211.97.55 130.211.99.206
```

You're telling `kubectl` to only output the information you want by specifying a JSONPath. You're probably familiar with XPath and how it's used with XML. JSONPath is basically XPath for JSON. The JSONPath in the previous example instructs `kubectl` to do the following:

- Go through all the elements in the `items` attribute.
- For each element, enter the `status` attribute.
- Filter elements of the `addresses` attribute, taking only those that have the `type` attribute set to `ExternalIP`.
- Finally, print the `address` attribute of the filtered elements.

To learn more about how to use JSONPath with `kubectl`, refer to the documentation at <http://kubernetes.io/docs/user-guide/jsonpath>.

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

```
$ curl http://130.211.97.55:30123
You've hit kubia-ym8or
$ curl http://130.211.99.206:30123
You've hit kubia-xueq1
```

**TIP** When using Minikube, you can easily access your NodePort services through your browser by running `minikube service <service-name> [-n <namespace>]`.

As you can see, your pods are now accessible to the whole internet through port 30123 on any of your nodes. It doesn't matter what node a client sends the request to. But if you only point your clients to the first node, when that node fails, your clients can't access the service anymore. That's why it makes sense to put a load balancer in front of the nodes to make sure you're spreading requests across all healthy nodes and never sending them to a node that's offline at that moment.

If your Kubernetes cluster supports it (which is mostly true when Kubernetes is deployed on cloud infrastructure), the load balancer can be provisioned automatically by creating a `LoadBalancer` instead of a `NodePort` service. We'll look at this next.

**5.3.2 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

To create a service with a load balancer in front, create the service from the following YAML manifest, as shown in the following listing.

**Listing 5.12** 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
```

← This type of service obtains a load balancer from the infrastructure hosting the Kubernetes cluster.

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

After you create the service, it takes time for the cloud infrastructure to create the load balancer and write its IP address into the Service object. Once it does that, the IP address will be listed as the external IP address of your service:

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

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

In this case, the load balancer is available at IP `130.211.53.173`, so you can now access the service at that IP address:

```
$ curl http://130.211.53.173
You've hit kubia-xueql
```

Success! As you may have noticed, this time you didn't need to mess with firewalls the way you had to before with the `NodePort` service.

### 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 describe`, 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).

See figure 5.7 to see how HTTP requests are delivered to the pod. External clients (`curl` in your case) connect to port 80 of the load balancer and get routed to the

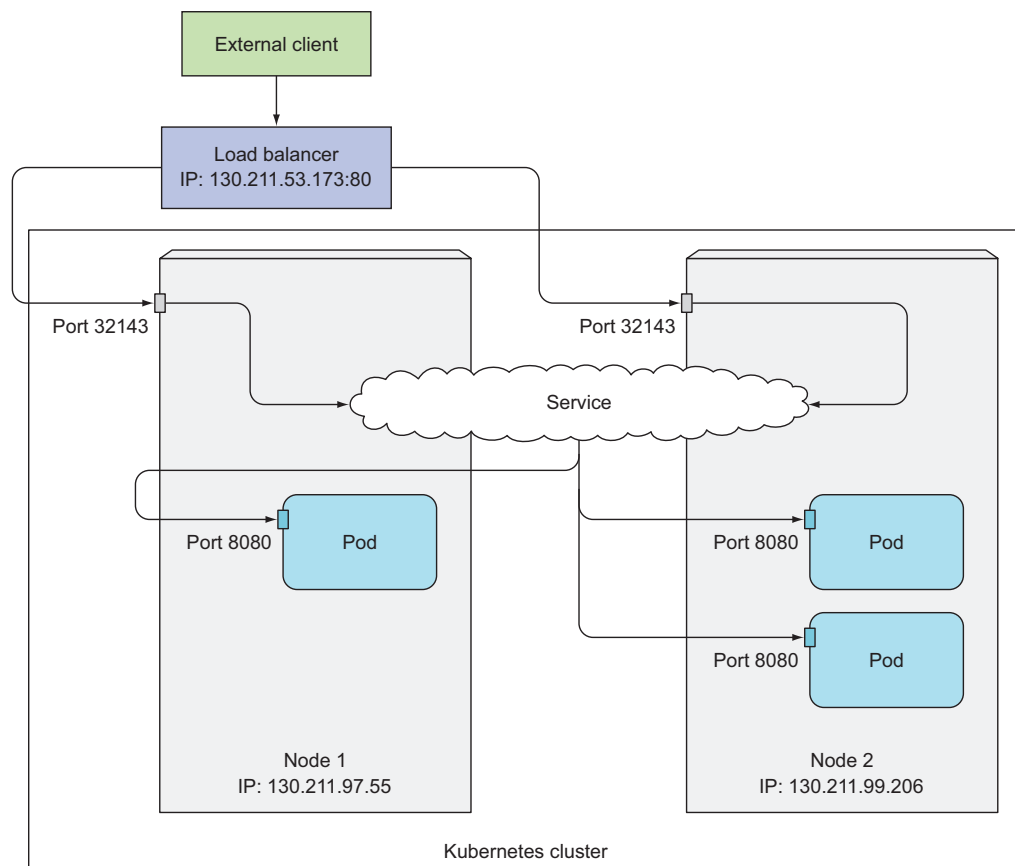


Figure 5.7 An external client connecting to a LoadBalancer service

implicitly assigned node port on one of the nodes. From there, the connection is forwarded to one of the pod instances.

As already mentioned, a LoadBalancer-type service is a NodePort service with an additional infrastructure-provided load balancer. If you use `kubectl describe` to display additional info about the service, you'll see that a node port has been selected for the service. If you were to open the firewall for this port, the way you did in the previous section about NodePort services, you could access the service through the node IPs as well.

**TIP** If you're using Minikube, even though the load balancer will never be provisioned, you can still access the service through the node port (at the Minikube VM's IP address).

### 5.3.3 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.

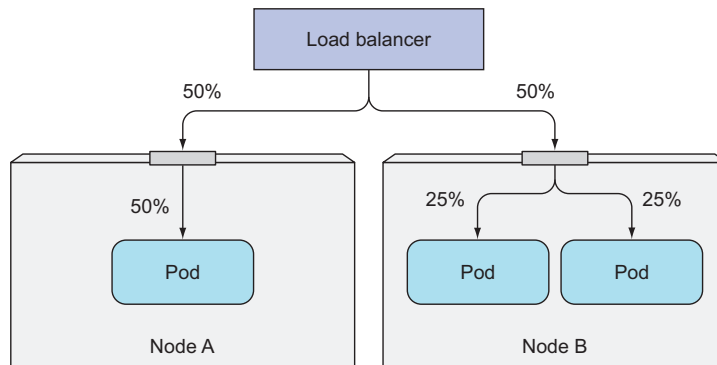
You can prevent this additional hop by configuring the service to redirect external traffic only to pods running on the node that received the connection. This is done by setting the `externalTrafficPolicy` field in the service's spec section:

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

If a service definition includes this setting and an external connection is opened through the service's node port, the service proxy will choose a locally running pod. If no local pods exist, the connection will hang (it won't be forwarded to a random global pod, the way connections are when not using the annotation). You therefore need to ensure the load balancer forwards connections only to nodes that have at least one such pod.

Using this annotation also has other drawbacks. Normally, connections are spread evenly across all the pods, but when using this annotation, that's no longer the case.

Imagine having two nodes and three pods. Let's say node A runs one pod and node B runs the other two. If the load balancer spreads connections evenly across the two nodes, the pod on node A will receive 50% of all connections, but the two pods on node B will only receive 25% each, as shown in figure 5.8.



**Figure 5.8** A Service using the `Local` external traffic policy may lead to uneven load distribution across pods.

### 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).

## 5.4 Exposing services externally through an Ingress resource

You've now seen two ways of exposing a service to clients outside the cluster, but another method exists—creating an Ingress resource.

**DEFINITION** *Ingress* (noun)—The act of going in or entering; the right to enter; a means or place of entering; entryway.

Let me first explain why you need another way to access Kubernetes services from the outside.

### UNDERSTANDING WHY INGRESSES ARE NEEDED

One important reason is that 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, as shown in figure 5.9.

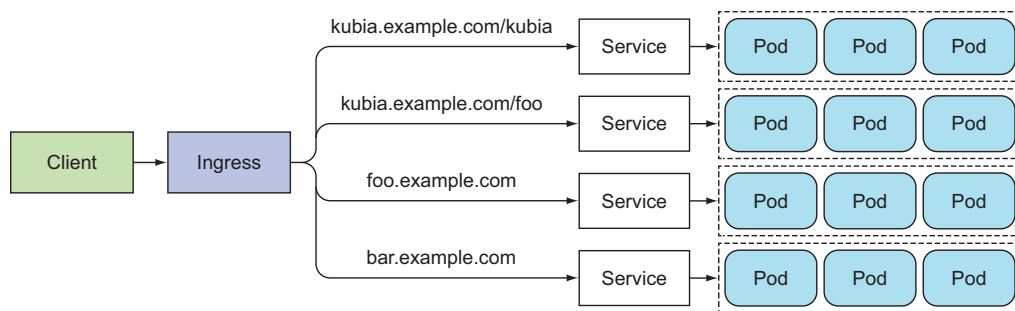


Figure 5.9 Multiple services can be exposed through a single Ingress.

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

Before we go into the features an Ingress object provides, let me emphasize that to make Ingress resources work, an Ingress controller needs to be running in the cluster. Different Kubernetes environments use different implementations of the controller, but several don't provide a default controller at all.

For example, Google Kubernetes Engine uses Google Cloud Platform's own HTTP load-balancing features to provide the Ingress functionality. Initially, Minikube didn't provide a controller out of the box, but it now includes an add-on that can be enabled to let you try out the Ingress functionality. Follow the instructions in the following sidebar to ensure it's enabled.

#### Enabling the Ingress add-on in Minikube

If you're using Minikube to run the examples in this book, you'll need to ensure the Ingress add-on is enabled. You can check whether it is by listing all the add-ons:

```

$ minikube addons list
- default-storageclass: enabled
- kube-dns: enabled
- heapster: disabled
- ingress: disabled
- registry-creds: disabled
- addon-manager: enabled
- dashboard: enabled
  
```

← The Ingress add-on isn't enabled.

You'll learn about what these add-ons are throughout the book, but it should be pretty clear what the dashboard and the kube-dns add-ons do. Enable the Ingress add-on so you can see Ingresses in action:

```

$ minikube addons enable ingress
ingress was successfully enabled
  
```



**(continued)**

This should have spun up an Ingress controller as another pod. Most likely, the controller pod will be in the `kube-system` namespace, but not necessarily, so list all the running pods across all namespaces by using the `--all-namespaces` option:

```
$ kubectl get po --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
default	kubia-rsv5m	1/1	Running	0	13h
default	kubia-fe4ad	1/1	Running	0	13h
default	kubia-ke823	1/1	Running	0	13h
kube-system	default-http-backend-5wb0h	1/1	Running	0	18m
kube-system	kube-addon-manager-minikube	1/1	Running	3	6d
kube-system	kube-dns-v20-101vq	3/3	Running	9	6d
kube-system	kubernetes-dashboard-jxd9l	1/1	Running	3	6d
kube-system	nginx-ingress-controller-gdts0	1/1	Running	0	18m

At the bottom of the output, you see the Ingress controller pod. The name suggests that Nginx (an open-source HTTP server and reverse proxy) is used to provide the Ingress functionality.

**TIP** The `--all-namespaces` option mentioned in the sidebar is handy when you don't know what namespace your pod (or other type of resource) is in, or if you want to list resources across all namespaces.

### 5.4.1 Creating an Ingress resource

You've confirmed there's an Ingress controller running in your cluster, so you can now create an Ingress resource. The following listing shows what the YAML manifest for the Ingress looks like.

**Listing 5.13** An Ingress resource definition: `kubia-ingress.yaml`

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kubia
spec:
  rules:
  - host: kubia.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: kubia-nodeport
          servicePort: 80
```

← This Ingress maps the **kubia.example.com** domain name to your service.

┌ All requests will be sent to port 80 of the kubia-nodeport service.

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.

**NOTE** 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.

### 5.4.2 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

To look up the IP, you need to list Ingresses:

```
$ kubectl get ingresses
NAME          HOSTS              ADDRESS          PORTS    AGE
kubia         kubia.example.com  192.168.99.100  80       29m
```

**NOTE** 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.

#### ENSURING THE HOST CONFIGURED IN THE INGRESS POINTS TO THE INGRESS' IP ADDRESS

Once you know the IP, you can then either configure your DNS servers to resolve [kubia.example.com](http://kubia.example.com) to that IP or you can add the following line to `/etc/hosts` (or `C:\windows\system32\drivers\etc\hosts` on Windows):

```
192.168.99.100    kubia.example.com
```

#### ACCESSING PODS THROUGH THE INGRESS

Everything is now set up, so you can access the service at <http://kubia.example.com> (using a browser or curl):

```
$ curl http://kubia.example.com
You've hit kubia-ke823
```

You've successfully accessed the service through an Ingress. Let's take a better look at how that unfolded.

#### UNDERSTANDING HOW INGRESSES WORK

Figure 5.10 shows how the client connected to one of the pods through the Ingress controller. The client first performed a DNS lookup of [kubia.example.com](http://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](http://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.

As you can see, the Ingress controller didn't forward the request to the service. It only used it to select a pod. Most, if not all, controllers work like this.

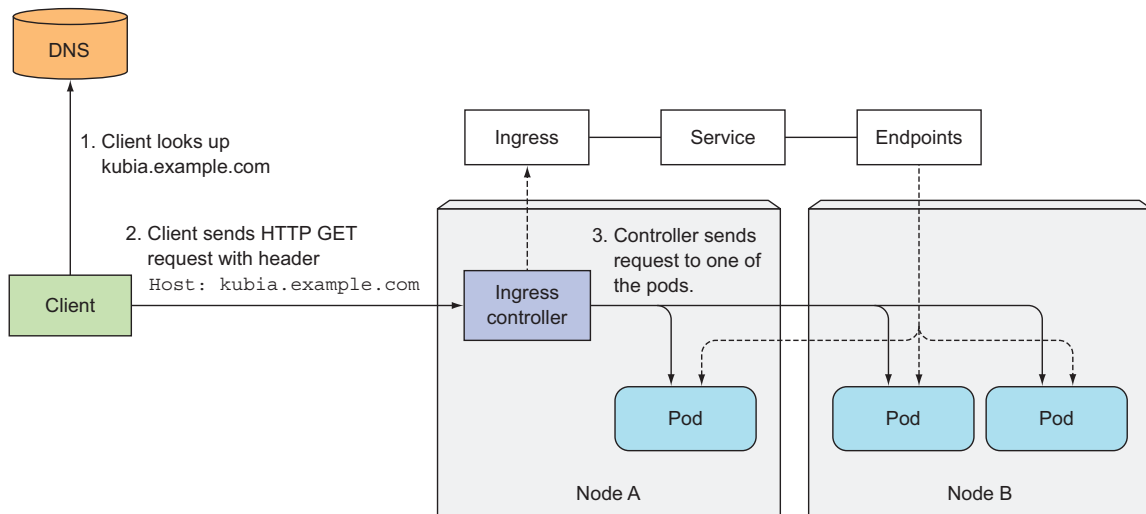


Figure 5.10 Accessing pods through an Ingress

### 5.4.3 Exposing multiple services through the same Ingress

If you look at the Ingress spec closely, you'll see that both rules and paths are arrays, so they can contain multiple items. An Ingress can map multiple hosts and paths to multiple services, as you'll see next. Let's focus on paths first.

#### MAPPING DIFFERENT SERVICES TO DIFFERENT PATHS OF THE SAME HOST

You can map multiple paths on the same host to different services, as shown in the following listing.

#### Listing 5.14 Ingress exposing multiple services on same host, but different paths

```
...
- host: kuba.example.com
  http:
    paths:
      - path: /kuba
        backend:
          serviceName: kuba
          servicePort: 80
      - path: /bar
        backend:
          serviceName: bar
          servicePort: 80
```

Requests to kuba.example.com/kuba  
will be routed to the kuba service.

Requests to kuba.example.com/bar  
will be routed to the bar service.


In this case, requests will be sent to two different services, depending on the path in the requested URL. Clients can therefore reach two different services through a single IP address (that of the Ingress controller).

**MAPPING DIFFERENT SERVICES TO DIFFERENT HOSTS**

Similarly, you can use an Ingress to map to different services based on the host in the HTTP request instead of (only) the path, as shown in the next listing.

**Listing 5.15** Ingress exposing multiple services on different hosts

```
spec:
  rules:
    - host: foo.example.com
      http:
        paths:
          - path: /
            backend:
              serviceName: foo
              servicePort: 80
    - host: bar.example.com
      http:
        paths:
          - path: /
            backend:
              serviceName: bar
              servicePort: 80
```



Requests for **foo.example.com** will be routed to service foo.

Requests for **bar.example.com** will be routed to service bar.

Requests received by the controller will be forwarded to either service foo or bar, depending on the Host header in the request (the way virtual hosts are handled in web servers). DNS needs to point both the [foo.example.com](http://foo.example.com) and the [bar.example.com](http://bar.example.com) domain names to the Ingress controller's IP address.

**5.4.4 Configuring Ingress to handle TLS traffic**

You've seen how an Ingress forwards HTTP traffic. But what about HTTPS? Let's take a quick look at how to configure Ingress to support TLS.

**CREATING A TLS CERTIFICATE FOR THE INGRESS**

When a client opens a TLS connection to an Ingress controller, the controller terminates the TLS connection. The communication between the client and the controller is encrypted, whereas the communication between the controller and the backend pod isn't. The application running in the pod doesn't need to support TLS. For example, if the pod runs a web server, it can accept only HTTP traffic and let the Ingress controller take care of everything related to TLS. To enable the controller to do that, you need to attach a certificate and a private key to the Ingress. The two need to be stored in a Kubernetes resource called a Secret, which is then referenced in the Ingress manifest. We'll explain Secrets in detail in chapter 7. For now, you'll create the Secret without paying too much attention to it.

First, you need to create the private key and certificate:

```
$ openssl genrsa -out tls.key 2048
$ openssl req -new -x509 -key tls.key -out tls.cert -days 360 -subj
➡ /CN=kubia.example.com
```

Then you create the Secret from the two files like this:

```
$ kubectl create secret tls tls-secret --cert=tls.cert --key=tls.key
secret "tls-secret" created
```

### Signing certificates through the CertificateSigningRequest resource

Instead of signing the certificate ourselves, you can get the certificate signed by creating a CertificateSigningRequest (CSR) resource. Users or their applications can create a regular certificate request, put it into a CSR, and then either a human operator or an automated process can approve the request like this:

```
$ kubectl certificate approve <name of the CSR>
```

The signed certificate can then be retrieved from the CSR's `status.certificate` field.

Note that a certificate signer component must be running in the cluster; otherwise creating `CertificateSigningRequest` and approving or denying them won't have any effect.

The private key and the certificate are now stored in the Secret called `tls-secret`. Now, you can update your Ingress object so it will also accept HTTPS requests for [kubia.example.com](http://kubia.example.com). The Ingress manifest should now look like the following listing.

#### Listing 5.16 Ingress handling TLS traffic: `kubia-ingress-tls.yaml`

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kubia
spec:
  tls:
    - hosts:
      - kubia.example.com
      secretName: tls-secret
  rules:
    - host: kubia.example.com
      http:
        paths:
          - path: /
            backend:
              serviceName: kubia-nodeport
              servicePort: 80
```

The whole TLS configuration is under this attribute.

TLS connections will be accepted for the `kubia.example.com` hostname.

The private key and the certificate should be obtained from the `tls-secret` you created previously.

**TIP** Instead of deleting the Ingress and re-creating it from the new file, you can invoke `kubectl apply -f kubia-ingress-tls.yaml`, which updates the Ingress resource with what's specified in the file.

You can now use HTTPS to access your service through the Ingress:

```
$ curl -k -v https://kubia.example.com/kubia
* About to connect() to kubia.example.com port 443 (#0)
...
* Server certificate:
*   subject: CN=kubia.example.com
...
> GET /kubia HTTP/1.1
> ...
You've hit kubia-xueq1
```

The command's output shows the response from the app, as well as the server certificate you configured the Ingress with.

**NOTE** Support for Ingress features varies between the different Ingress controller implementations, so check the implementation-specific documentation to see what's supported.

Ingresses are a relatively new Kubernetes feature, so you can expect to see many improvements and new features in the future. Although they currently support only L7 (HTTP/HTTPS) load balancing, support for L4 load balancing is also planned.

## 5.5 Signaling when a pod is ready to accept connections

There's one more thing we need to cover regarding both Services and Ingresses. You've already learned that pods are included as endpoints of a service if their labels match the service's pod selector. As soon as a new pod with proper labels is created, it becomes part of the service and requests start to be redirected to the pod. But what if the pod isn't ready to start serving requests immediately?

The pod may need time to load either configuration or data, or it may need to perform a warm-up procedure to prevent the first user request from taking too long and affecting the user experience. In such cases you don't want the pod to start receiving requests immediately, especially when the already-running instances can process requests properly and quickly. It makes sense to not forward requests to a pod that's in the process of starting up until it's fully ready.

### 5.5.1 Introducing readiness probes

In the previous chapter you learned about liveness probes and how they help keep your apps healthy by ensuring unhealthy containers are restarted automatically. Similar to liveness probes, Kubernetes allows you to also define a readiness probe for your pod.

The readiness probe is invoked periodically and determines whether the specific pod should receive client requests or not. When a container's readiness probe returns success, it's signaling that the container is ready to accept requests.

This notion of being ready is obviously something that's specific to each container. Kubernetes can merely check if the app running in the container responds to a simple

GET / request or it can hit a specific URL path, which causes the app to perform a whole list of checks to determine if it's ready. Such a detailed readiness probe, which takes the app's specifics into account, is the app developer's responsibility.

#### TYPES OF READINESS PROBES

Like liveness probes, three types of readiness probes exist:

- An *Exec* probe, where a process is executed. The container's status is determined by the process' exit status code.
- An *HTTP GET* probe, which sends an HTTP GET request to the container and the HTTP status code of the response determines whether the container is ready or not.
- A *TCP Socket* probe, which opens a TCP connection to a specified port of the container. If the connection is established, the container is considered ready.

#### UNDERSTANDING THE OPERATION OF READINESS PROBES

When a container is started, Kubernetes can be configured to wait for a configurable amount of time to pass before performing the first readiness check. After that, it invokes the probe periodically and acts based on the result of the readiness probe. If a pod reports that it's not ready, it's removed from the service. If the pod then becomes ready again, it's re-added.

Unlike liveness probes, if a container fails the readiness check, it won't be killed or restarted. This is an important distinction between liveness and readiness probes. Liveness probes keep pods healthy by killing off unhealthy containers and replacing them with new, healthy ones, whereas readiness probes make sure that only pods that are ready to serve requests receive them. This is mostly necessary during container start up, but it's also useful after the container has been running for a while.

As you can see in figure 5.11, if a pod's readiness probe fails, the pod is removed from the Endpoints object. Clients connecting to the service will not be redirected to the pod. The effect is the same as when the pod doesn't match the service's label selector at all.

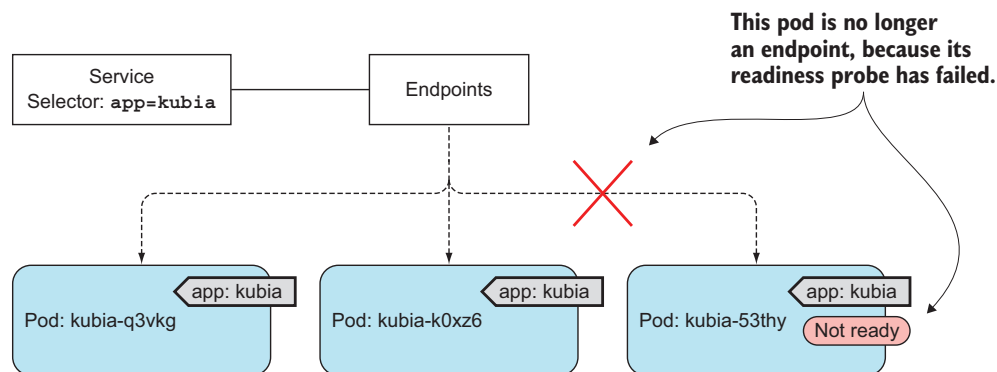


Figure 5.11 A pod whose readiness probe fails is removed as an endpoint of a service.

**UNDERSTANDING WHY READINESS PROBES ARE IMPORTANT**

Imagine that a group of pods (for example, pods running application servers) depends on a service provided by another pod (a backend database, for example). If at any point one of the frontend pods experiences connectivity problems and can't reach the database anymore, it may be wise for its readiness probe to signal to Kubernetes that the pod isn't ready to serve any requests at that time. If other pod instances aren't experiencing the same type of connectivity issues, they can serve requests normally. A readiness probe makes sure clients only talk to those healthy pods and never notice there's anything wrong with the system.

**5.5.2 Adding a readiness probe to a pod**

Next you'll add a readiness probe to your existing pods by modifying the ReplicationController's pod template.

**ADDING A READINESS PROBE TO THE POD TEMPLATE**

You'll use the `kubectl edit` command to add the probe to the pod template in your existing ReplicationController:

```
$ kubectl edit rc kubia
```

When the ReplicationController's YAML opens in the text editor, find the container specification in the pod template and add the following readiness probe definition to the first container under `spec.template.spec.containers`. The YAML should look like the following listing.

**Listing 5.17 RC creating a pod with a readiness probe: kubia-rc-readinessprobe.yaml**

```
apiVersion: v1
kind: ReplicationController
...
spec:
  ...
  template:
    ...
    spec:
      containers:
      - name: kubia
        image: luksa/kubia
        readinessProbe:
          exec:
            command:
            - ls
            - /var/ready
        ...
```

A readinessProbe may be defined for each container in the pod.

The readiness probe will periodically perform the command `ls /var/ready` inside the container. The `ls` command returns exit code zero if the file exists, or a non-zero exit code otherwise. If the file exists, the readiness probe will succeed; otherwise, it will fail.



The reason you’re defining such a strange readiness probe is so you can toggle its result by creating or removing the file in question. The file doesn’t exist yet, so all the pods should now report not being ready, right? Well, not exactly. As you may remember from the previous chapter, changing a ReplicationController’s pod template has no effect on existing pods.

In other words, all your existing pods still have no readiness probe defined. You can see this by listing the pods with `kubectl get pods` and looking at the `READY` column. You need to delete the pods and have them re-created by the ReplicationController. The new pods will fail the readiness check and won’t be included as endpoints of the service until you create the `/var/ready` file in each of them.

#### OBSERVING AND MODIFYING THE PODS’ READINESS STATUS

List the pods again and inspect whether they’re ready or not:

```
$ kubectl get po
NAME          READY    STATUS    RESTARTS   AGE
kubia-2r1qb   0/1      Running   0           1m
kubia-3rax1   0/1      Running   0           1m
kubia-3yw4s   0/1      Running   0           1m
```

The `READY` column shows that none of the containers are ready. Now make the readiness probe of one of them start returning success by creating the `/var/ready` file, whose existence makes your mock readiness probe succeed:

```
$ kubectl exec kubia-2r1qb -- touch /var/ready
```

You’ve used the `kubectl exec` command to execute the `touch` command inside the container of the `kubia-2r1qb` pod. The `touch` command creates the file if it doesn’t yet exist. The pod’s readiness probe command should now exit with status code 0, which means the probe is successful, and the pod should now be shown as ready. Let’s see if it is:

```
$ kubectl get po kubia-2r1qb
NAME          READY    STATUS    RESTARTS   AGE
kubia-2r1qb   0/1      Running   0           2m
```

The pod still isn’t ready. Is there something wrong or is this the expected result? Take a more detailed look at the pod with `kubectl describe`. The output should contain the following line:

```
Readiness: exec [ls /var/ready] delay=0s timeout=1s period=10s #success=1
➡ #failure=3
```

The readiness probe is checked periodically—every 10 seconds by default. The pod isn’t ready because the readiness probe hasn’t been invoked yet. But in 10 seconds at the latest, the pod should become ready and its IP should be listed as the only endpoint of the service (run `kubectl get endpoints kubia-loadbalancer` to confirm).

**HITTING THE SERVICE WITH THE SINGLE READY POD**

You can now hit the service URL a few times to see that each and every request is redirected to this one pod:

```
$ curl http://130.211.53.173
You've hit kubia-2r1qb
$ curl http://130.211.53.173
You've hit kubia-2r1qb
...
$ curl http://130.211.53.173
You've hit kubia-2r1qb
```

Even though there are three pods running, only a single pod is reporting as being ready and is therefore the only pod receiving requests. If you now delete the file, the pod will be removed from the service again.

**5.5.3 Understanding what real-world readiness probes should do**

This mock readiness probe is useful only for demonstrating what readiness probes do. In the real world, the readiness probe should return success or failure depending on whether the app can (and wants to) receive client requests or not.

Manually removing pods from services should be performed by either deleting the pod or changing the pod's labels instead of manually flipping a switch in the probe.

**TIP** If you want to add or remove a pod from a service manually, add `enabled=true` as a label to your pod and to the label selector of your service. Remove the label when you want to remove the pod from the service.

**ALWAYS DEFINE A READINESS PROBE**

Before we conclude this section, there are two final notes about readiness probes that I need to emphasize. First, if you don't add a readiness probe to your pods, they'll become service endpoints almost immediately. If your application takes too long to start listening for incoming connections, client requests hitting the service will be forwarded to the pod while it's still starting up and not ready to accept incoming connections. Clients will therefore see "Connection refused" types of errors.

**TIP** You should always define a readiness probe, even if it's as simple as sending an HTTP request to the base URL.

**DON'T INCLUDE POD SHUTDOWN LOGIC INTO YOUR READINESS PROBES**

The other thing I need to mention applies to the other end of the pod's life (pod shutdown) and is also related to clients experiencing connection errors.

When a pod is being shut down, the app running in it usually stops accepting connections as soon as it receives the termination signal. Because of this, you might think you need to make your readiness probe start failing as soon as the shutdown procedure is initiated, ensuring the pod is removed from all services it's part of. But that's not necessary, because Kubernetes removes the pod from all services as soon as you delete the pod.

## 5.6 Using a headless service for discovering individual pods

You’ve seen how services can be used to provide a stable IP address allowing clients to connect to pods (or other endpoints) backing each service. Each connection to the service is forwarded to one randomly selected backing pod. But what if the client needs to connect to all of those pods? What if the backing pods themselves need to each connect to all the other backing pods? Connecting through the service clearly isn’t the way to do this. What is?

For a client to connect to all pods, it needs to figure out the IP of each individual pod. One option is to have the client call the Kubernetes API server and get the list of pods and their IP addresses through an API call, but because you should always strive to keep your apps Kubernetes-agnostic, using the API server isn’t ideal.

Luckily, Kubernetes allows clients to discover pod IPs through DNS lookups. Usually, when you perform a DNS lookup for a service, the DNS server returns a single IP—the service’s cluster IP. But if you tell Kubernetes you don’t need a cluster IP for your service (you do this by setting the `clusterIP` field to `None` in the service specification), the DNS server will return the pod IPs instead of the single service IP.

Instead of returning a single DNS A record, the DNS server will return multiple A records for the service, each pointing to the IP of an individual pod backing the service at that moment. Clients can therefore do a simple DNS A record lookup and get the IPs of all the pods that are part of the service. The client can then use that information to connect to one, many, or all of them.

### 5.6.1 Creating a headless service

Setting the `clusterIP` field in a service spec to `None` makes the service *headless*, as Kubernetes won’t assign it a cluster IP through which clients could connect to the pods backing it.

You’ll create a headless service called `kubia-headless` now. The following listing shows its definition.

**Listing 5.18** A headless service: `kubia-svc-headless.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: kubia-headless
spec:
  clusterIP: None
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: kubia
```

← This makes the service headless.

After you create the service with `kubectl create`, you can inspect it with `kubectl get` and `kubectl describe`. You’ll see it has no cluster IP and its endpoints include (part of)

the pods matching its pod selector. I say “part of” because your pods contain a readiness probe, so only pods that are ready will be listed as endpoints of the service. Before continuing, please make sure at least two pods report being ready, by creating the `/var/ready` file, as in the previous example:

```
$ kubectl exec <pod name> -- touch /var/ready
```

### 5.6.2 Discovering pods through DNS

With your pods ready, you can now try performing a DNS lookup to see if you get the actual pod IPs or not. You’ll need to perform the lookup from inside one of the pods. Unfortunately, your `kubia` container image doesn’t include the `nslookup` (or the `dig`) binary, so you can’t use it to perform the DNS lookup.

All you’re trying to do is perform a DNS lookup from inside a pod running in the cluster. Why not run a new pod based on an image that contains the binaries you need? To perform DNS-related actions, you can use the `tutum/dnsutils` container image, which is available on Docker Hub and contains both the `nslookup` and the `dig` binaries. To run the pod, you can go through the whole process of creating a YAML manifest for it and passing it to `kubectl create`, but that’s too much work, right? Luckily, there’s a faster way.

#### RUNNING A POD WITHOUT WRITING A YAML MANIFEST

In chapter 1, you already created pods without writing a YAML manifest by using the `kubectl run` command. But this time you want to create only a pod—you don’t need to create a `ReplicationController` to manage the pod. You can do that like this:

```
$ kubectl run dnsutils --image=tutum/dnsutils --generator=run-pod/v1
➡ --command -- sleep infinity
pod "dnsutils" created
```

The trick is in the `--generator=run-pod/v1` option, which tells `kubectl` to create the pod directly, without any kind of `ReplicationController` or similar behind it.

#### UNDERSTANDING DNS A RECORDS RETURNED FOR A HEADLESS SERVICE

Let’s use the newly created pod to perform a DNS lookup:

```
$ kubectl exec dnsutils nslookup kubia-headless
...
Name:      kubia-headless.default.svc.cluster.local
Address: 10.108.1.4
Name:      kubia-headless.default.svc.cluster.local
Address: 10.108.2.5
```

The DNS server returns two different IPs for the `kubia-headless.default.svc.cluster.local` FQDN. Those are the IPs of the two pods that are reporting being ready. You can confirm this by listing pods with `kubectl get pods -o wide`, which shows the pods’ IPs.

This is different from what DNS returns for regular (non-headless) services, such as for your `kubia` service, where the returned IP is the service's cluster IP:

```
$ kubectl exec dnsutils nslookup kubia
...
Name:      kubia.default.svc.cluster.local
Address: 10.111.249.153
```

Although headless services may seem different from regular services, they aren't that different from the clients' perspective. Even with a headless service, clients can connect to its pods by connecting to the service's DNS name, as they can with regular services. But with headless services, because DNS returns the pods' IPs, clients connect directly to the pods, instead of through the service proxy.

**NOTE** A headless services still provides load balancing across pods, but through the DNS round-robin mechanism instead of through the service proxy.

### 5.6.3 *Discovering all pods—even those that aren't ready*

You've seen that only pods that are ready become endpoints of services. But sometimes you want to use the service discovery mechanism to find all pods matching the service's label selector, even those that aren't ready.

Luckily, you don't have to resort to querying the Kubernetes API server. You can use the DNS lookup mechanism to find even those unready pods. To tell Kubernetes you want all pods added to a service, regardless of the pod's readiness status, you must add the following annotation to the service:

```
kind: Service
metadata:
  annotations:
    service.alpha.kubernetes.io/tolerate-unready-endpoints: "true"
```

**WARNING** As the annotation name suggests, as I'm writing this, this is an alpha feature. The Kubernetes Service API already supports a new service spec field called `publishNotReadyAddresses`, which will replace the `tolerate-unready-endpoints` annotation. In Kubernetes version 1.9.0, the field is not honored yet (the annotation is what determines whether unready endpoints are included in the DNS or not). Check the documentation to see whether that's changed.

## 5.7 *Troubleshooting services*

Services are a crucial Kubernetes concept and the source of frustration for many developers. I've seen many developers lose heaps of time figuring out why they can't connect to their pods through the service IP or FQDN. For this reason, a short look at how to troubleshoot services is in order.

When you're unable to access your pods through the service, you should start by going through the following list:

- First, make sure you're connecting to the service's cluster IP from within the cluster, not from the outside.
- Don't bother pinging the service IP to figure out if the service is accessible (remember, the service's cluster IP is a virtual IP and pinging it will never work).
- If you've defined a readiness probe, make sure it's succeeding; otherwise the pod won't be part of the service.
- To confirm that a pod is part of the service, examine the corresponding Endpoints object with `kubectl get endpoints`.
- If you're trying to access the service through its FQDN or a part of it (for example, `myservice.mynamespace.svc.cluster.local` or `myservice.mynamespace`) and it doesn't work, see if you can access it using its cluster IP instead of the FQDN.
- Check whether you're connecting to the port exposed by the service and not the target port.
- Try connecting to the pod IP directly to confirm your pod is accepting connections on the correct port.
- If you can't even access your app through the pod's IP, make sure your app isn't only binding to localhost.

This should help you resolve most of your service-related problems. You'll learn much more about how services work in chapter 11. By understanding exactly how they're implemented, it should be much easier for you to troubleshoot them.

## 5.8 Summary

In this chapter, you've learned how to create Kubernetes Service resources to expose the services available in your application, regardless of how many pod instances are providing each service. You've learned how Kubernetes

- Exposes multiple pods that match a certain label selector under a single, stable IP address and port
- Makes services accessible from inside the cluster by default, but allows you to make the service accessible from outside the cluster by setting its type to either `NodePort` or `LoadBalancer`
- Enables pods to discover services together with their IP addresses and ports by looking up environment variables
- Allows discovery of and communication with services residing outside the cluster by creating a Service resource without specifying a selector, by creating an associated Endpoints resource instead
- Provides a DNS CNAME alias for external services with the `ExternalName` service type
- Exposes multiple HTTP services through a single Ingress (consuming a single IP)

- Uses a pod container's readiness probe to determine whether a pod should or shouldn't be included as a service endpoint
- Enables discovery of pod IPs through DNS when you create a headless service

Along with getting a better understanding of services, you've also learned how to

- Troubleshoot them
- Modify firewall rules in Google Kubernetes/Compute Engine
- Execute commands in pod containers through `kubectl exec`
- Run a bash shell in an existing pod's container
- Modify Kubernetes resources through the `kubectl apply` command
- Run an unmanaged ad hoc pod with `kubectl run --generator=run-pod/v1`