CHAPTER 4

Service Discovery

Service discovery is the ability to locate the address of one or more Pods from within other Pods as well as the external world—the Internet. Kubernetes provides a Service controller to satisfy service discovery and connectivity use cases such as Pod-to-Pod, LAN-to-Pod, and Internet-to-Pod.

Service discovery is a necessity because Pods are volatile; they may be created and destroyed many times over throughout their life cycle, acquiring different IP address each time. The self-healing and scaling nature of Kubernetes also means that we often want a virtual IP address—and a round-robin load balancing mechanism—as opposed to the discrete address of specific, pet-like Pods.

In this chapter, we will first explore the three aforementioned connectivity use cases (Pod-to-Pod, LAN-to-Pod, and Internet-to-Pod). Then, we will look at the peculiarities of publishing services across different spaces and the exposure of multiple ports. Finally, we will contemplate how the Service controller helps instrument graceful startup and shutdown, as well as zero-downtime deployments.

Connectivity Use Cases Overview

The Service controller performs a variety of functions, but its main purpose is keeping track of Pods addresses and ports and publishing this information to interested service consumers. The Service controller also provides a single point of entry in a cluster scenario—multiple Pod replicas.

To achieve its purpose, it uses other Kubernetes services such as `kube-dns` and `kube-proxy` which in turn leverage the underlying kernel and networking resources from the OS such as *iptables*.

The Service controller caters for a variety of use cases, but these are the most typical ones:

- **Pod-to-Pod:** This scenario involves a Pod connecting to other Pods within the same Kubernetes cluster. The *ClusterIP* service type is used for this purpose; it consists of a virtual IP address and DNS entry which is addressable by all Pods within the same cluster and that will add and remove replicas from the Cluster as they become ready and "unready," respectively.
- **LAN-to-Pod:** In this case, service consumers typically sit outside of the Kubernetes cluster—but are located within the same local area network (LAN). The *NodePort* service type is appropriate in typically used to satisfy this use case; the Service controller publishes a discrete port in every worker Node which is mapped to the exposed deployment.
- **Internet-to-Pod:** In most cases, we will want to expose at least one Deployment to the Internet. Kubernetes will interact with the Google Cloud Platform's load balancer so that an external—public IP—address is created and routed to the NodePorts. This is a *LoadBalancer* service type.

There is also the special case of a *headless service* used primarily in conjunction with StatefulSets to provide DNS-wise access to every individual Pod. This special case is covered separately in Chapter 9.

It is worth understanding that the Service controller provides a layer of indirection—be in the form of a DNS entry, an extra IP address, or port number—to Pods that have their own discrete IP address and that can be accessed directly. If what we need is simply to find out a Pods' IP addresses, we can use the `kubectl get pods -o wide -l <LABEL>` command where `<LABEL>` differentiates the Pods we are looking for from others. The `-l` flag is only required when a large number of Pods are running. Otherwise, we can probably tell the Pods apart by their naming convention.

In the next example, we first create three Nginx replicas and then find out their IP address:

```
$ kubectl run nginx --image=nginx --replicas=3
deployment.apps/nginx created
```

```
$ kubectl get pods -o wide -l run=nginx
NAME            READY STATUS    RESTARTS   IP
nginx-*-5s7fb   1/1   Running   0          10.0.1.13
nginx-*-fkjx4   1/1   Running   0          10.0.0.7
nginx-*-sg9bv   1/1   Running   0          10.0.1.14
```

If instead, we want the IP addresses by themselves—say, for piping them into some program—we can use a programmatic approach by specifying a JSON Path query:

```
$ kubectl get pods -o jsonpath \
   --template="{.items[*].status.podIP}" -l run=nginx
10.36.1.7 10.36.0.6 10.36.2.8
```

Pod-to-Pod Connectivity Use Case

Addressing a Pod from an external Pod involves creating a Service object which will observe the Pod(s) so that they are added or removed from a virtual IP address—called a *ClusterIP*—as they become ready and *unready*, respectively. The instrumentation of a container's "readiness" may be implemented via custom probes as explained in Chapter 2. The Service controller can target a variety of objects such as bare Pods and ReplicaSets, but we will concentrate on Deployments only.

The first step is creating a Service controller. The imperative command for creating a service that observes an existing deployment is `kubectl expose deploy/<NAME>`. The optional flag `--name=<NAME>` will give the service a name different than its target deployment, and `--port=<NUMBER>` is only necessary if we had not specified a port on the target object. For example:

```
$ kubectl run nginx --image=nginx --replicas=3
deployment.apps/nginx created

$ kubectl expose deploy/nginx --port=80
service/nginx exposed
```

The declarative approach is similar, but it requires the use of a label selector (Chapter 2) to identify the target deployment:

```
# service.yaml
kind: Service
apiVersion: v1
metadata:
  name: nginx
spec:
  selector:
    run: nginx
  ports:
  - protocol: TCP
    port: 80
```

This manifest can be applied right after creating a deployment as follows:

```
$ kubectl run nginx --image=nginx --replicas=3
deployment "nginx" created

$ kubectl apply -f service.yaml
service "nginx" created
```

So far, we have solved the problem of "exposing" a deployment, but what is the outcome? What is it that we can do now that was not possible before? First, let us type `kubectl get services` to see the details about the object we have just created:

```
$ kubectl get services
NAME       TYPE      CLUSTER-IP    EXT-IP PORT(S)
kubernetes ClusterIP 10.39.240.1   <none> 443/TCP
nginx      ClusterIP 10.39.243.143 <none> 80/TCP
```

The IP address `10.39.243.143` is now ready to receive ingress traffic from any Pod on port 80. We can check this by running a dummy, disposable Pod that connects to this endpoint:

```
$ kubectl run test --rm -i --image=alpine \
    --restart=Never \
    -- wget -O - http://10.39.243.143 | grep title
<title>Welcome to nginx!</title>
```

We have a virtual IP to reach out to our Deployment's Pod replicas, but how does a Pod get to find out about the IP address in the first place? Well, the good news is that Kubernetes creates a DNS entry for each Service controller. In a simple, single cluster, single namespace scenario—like all the examples in this text—we can simply use the Service name itself. For example, assuming that we are inside another Pod (e.g., an Alpine instance), we can access the Nginx web server as follows:

```
$ kubectl run test --rm -i --image=alpine \
    --restart=Never \
    -- wget -O - http://nginx | grep title
<title>Welcome to nginx!</title>
```

Note that rather than using `http://10.39.243.143`, we now use `http://nginx` instead. Each HTTP request to Nginx will hit one of the three Pod replicas in a round-robin fashion. If we want to convince us that this is indeed the case, we can change the `index.html` content of each Nginx Pod so that it displays the Pod's name rather than the same default welcome page. Assuming our three-replica Nginx deployment is still running, we can apply the suggested change by first extracting the Deployment's Pod names, and then by overwriting the contents of `index.html` in each Pod with the value of `$HOSTNAME`:

```
# Extract Pod names
$ pods=$(kubectl get pods -l run=nginx -o jsonpath \
       --template="{.items[*].metadata.name})"

# Change the contents of index.html for every Pod
$ for pod in $pods; \
    do kubectl exec -ti $pod \
           -- bash -c "echo \$HOSTNAME > \
           /usr/share/nginx/html/index.html"; \
    done
```

Now we can launch an adhoc Pod again; this time, though, we will be running requests against `http://nginx` in a loop until we press Ctrl+C:

```
$ kubectl run test --rm -i --image=alpine \
    --restart=Never -- \
    sh -c "while true; do wget -q -O \
    - http://nginx ; sleep 1 ; done"
nginx-dbddb74b8-t728t
nginx-dbddb74b8-h87s4
nginx-dbddb74b8-mwcg4
nginx-dbddb74b8-h87s4
```

```
nginx-dbddb74b8-h87s4
nginx-dbddb74b8-t728t
nginx-dbddb74b8-h87s4
nginx-dbddb74b8-h87s4
...
```

As we can see in the resulting output, a round-robin mechanism is in action since each request lands on a random Pod.

LAN-to-Pod Connectivity Use Case

Accessing Pods from an external host to the Kubernetes cluster involves exposing the services using the `NodePort` service type (the default service type is `ClusterIP`). This is just a matter of adding the `--type=NodePort` flag to the `kubectl expose` command. For example:

```
$ kubectl run nginx --image=nginx --replicas=3
deployment.apps/nginx created
```

```
$ kubectl expose deploy/nginx --type="NodePort" \
    --port=80
service/nginx exposed
```

The result is that Nginx HTTP server can be accessed now through a discrete port on any of the Node's IP addresses. First, let us find what the assigned port is:

```
$ kubectl describe service/nginx | grep NodePort
NodePort:                 <unset>  30091/TCP
```

We can see that the automatically assigned port is 30091. We can now make requests to the Nginx's web server through any of the Kubernetes' worker Nodes on this port from a machine *outside the Kubernetes cluster* in the same local area using the external IP address:

```
$ kubectl get nodes -o wide
NAME                   STATUS  AGE EXTERNAL-IP
gke-*-9777d23b-9103   Ready   7h  35.189.64.73
gke-*-9777d23b-m6hk   Ready   7h  35.197.208.108
gke-*-9777d23b-r4s9   Ready   7h  35.197.192.9
```

```
$ curl -s http://35.189.64.73:30091 | grep title
<title>Welcome to nginx!</title>
$ curl -s http://35.197.208.108:30091 | grep title
<title>Welcome to nginx!</title>
$ curl -s http://35.197.192.9:30091 | grep title
<title>Welcome to nginx!</title>
```

Note The Lan-to-Pod examples may not work directly in the Google Cloud Shell unless further security/network settings are applied. Such settings are outside the scope of this book.

To conclude this section, the declarative version of the `kubectl expose deploy/nginx --type="NodePort" --port=80` command is provided here:

```
# serviceNodePort.yaml
kind: Service
apiVersion: v1
metadata:
  name: nginx
```

```
spec:
  selector:
    run: nginx
  ports:
  - protocol: TCP
    port: 80
  type: NodePort
```

The only difference between the ClusterIP manifest used for Pod-to-Pod access is that the attribute `type` is added and set to `NodePort`. This attribute's value is `ClusterIP` by default if left undeclared.

Internet-to-Pod Connectivity Use Case

Accessing Pods from the Internet involves creating `LoadBalancer` service type. The `LoadBalancer` service type is similar to the `NodePort` service type in that it will publish the exposed object on a discrete port in every Node of the Kubernetes cluster. The difference is that, in addition to this, it will interact with the Google Cloud Platform's load balancer and allocate a public IP address that can direct traffic to these ports.

The following example creates a cluster of three Nginx replicas and exposes the Deployment to the Internet. Please note that the last command uses the `-w` flag in order to wait until the external—public—IP address is allocated:

```
$ kubectl run nginx --image=nginx --replicas=3
deployment.apps/nginx created

$ kubectl expose deploy/nginx --type=LoadBalancer \
    --port=80
service/nginx exposed
```

```
$ kubectl get services -w
NAME  TYPE         CLUSTER-IP    EXT-IP
nginx LoadBalancer 10.39.249.178 <pending>
nginx LoadBalancer 10.39.249.178 35.189.65.215
```

Whenever the load balancer is assigned a public IP address, the `service.status.loadBalancer.ingress.ip` property—or `.hostname` on other cloud vendors such as AWS—will be populated. If we want to capture the public IP address programmatically, all we must do is wait until this attribute is set. We can instrument this solution through a *while loop* in Bash, for example:

```
while [ -z $PUBLIC_IP ]; \
 do PUBLIC_IP=$(kubectl get service/nginx \
 -o jsonpath \
 --template="{.status.loadBalancer.ingress[*].ip}");\
 sleep 1; \
 done; \
 echo $PUBLIC_IP
35.189.65.215
```

The declarative version of `kubectl expose deploy/nginx --type=LoadBalancer --port=80` is presented in the next code listing. The only difference between the manifest used for the LAN-to-Pod use case is that the `type` attribute is set to `LoadBalancer`. The manifest is applied using the `kubectl apply -f serviceLoadBalancer.yaml` command. We might want to dispose of any running clashing Service before applying this command by issuing the `kubectl delete service/nginx` command first.

```
# serviceLoadBalancer.yaml
kind: Service
apiVersion: v1
metadata:
  name: nginx
```

```
spec:
  selector:
    run: nginx
  ports:
  - protocol: TCP
    port: 80
  type: LoadBalancer
```

Accessing Services in Different Namespaces

All the examples we have seen so far live in the *default* namespace. This is as though every `kubectl` command had the `-n default` flag added by default. As such, we never had to be concerned with full DNS names. Whenever we expose the Nginx deployment by typing `kubectl expose deploy/nginx`, we can access the resulting service from Pods without any additional domain components, for example, by typing `wget http://nginx`.

If more namespaces are in use, though, things may get tricky, and it might be useful to understand the shape of the full DNS record associated with each service. Let us suppose that there is a `nginx` Service in the `default` namespace—in which case there is no need to indicate a specific namespace since this is the default—and another equally named service in the `production` namespace as follows:

```
# nginx in the default namespace

$ kubectl run nginx --image=nginx --port=80
deployment.apps/nginx created

$ kubectl expose deploy/nginx
service/nginx exposed
```

```
# nginx in the production namespace

$ kubectl create namespace production
namespace/production created

$ kubectl run nginx --image=nginx --port=80 \
    -n production
deployment.apps/nginx created

$ kubectl expose deploy/nginx -n production
service/nginx exposed
```

The result is two Services named `nginx` but that live in different namespaces:

```
$ kubectl get services --all-namespaces | grep nginx
NAMESPACE  NAME  TYPE      CLUSTER-IP      PORT(S)
default    nginx ClusterIP 10.39.243.143   80/TCP
production nginx ClusterIP 10.39.244.112   80/TCP
```

Whether we get the Nginx service published on `10.39.243.143` or `10.39.244.112` will depend on the namespace the requesting Pod is running on:

```
$ kubectl run test --rm -ti --image=alpine \
    --restart=Never \
    -- getent hosts nginx | awk '{ print $1 }'
10.39.243.143

$ kubectl run test --rm -ti --image=alpine \
    --restart=Never \
    -n production \
    -- getent hosts nginx | awk '{ print $1 }'
10.39.244.112
```

The Pods within the `default` space will connect to `10.39.243.143` when using `nginx` as a host, whereas those in the `production` namespace will connect to `10.39.244.112`. The way to reach a `default` ClusterIP from `production` and vice versa is to use the full domain name.

The default configuration uses the `service-name.namespace.svc.cluster.local` convention where `service-name` is `nginx` and `namespace` is either `default` or `production` in our example:

```
$ kubectl run test --rm -ti --image=alpine \
    --restart=Never \
    -- sh -c \
    "getent hosts nginx.default.svc.cluster.local; \
    getent hosts nginx.production.svc.cluster.local"
10.39.243.143     nginx.default.svc.cluster.local
10.39.244.112     nginx.production.svc.cluster.local
```

Exposing Services on a Different Port

The `kubectl expose` command and its equivalent declarative form will introspect the target object and expose it on its declared port. If no port information is available, then we specify the port using the `--port` flag or the `service.spec.ports.port` attribute. In our previous examples, the exposed port has always coincided with the actual Pod's port; whenever the exposed port differs from published one, it must be specified using either the `--target-port` flag or the `service.spec.ports.targetPort` attribute in the Service manifest.

In the next example, we create a Nginx deployment as usual—on port 80—but expose it on port 8000 on the public load balancer. Please note that given that the exposed port and published port are different, we must specify the exposed port using the `--target-port` flag:

```
$ kubectl run nginx --image=nginx
deployment.apps/nginx created
```

```
$ kubectl expose deploy/nginx --port=8000 \
    --target-port=80 \
    --type=LoadBalancer
service/nginx exposed
```

```
$ kubectl get services -w
NAME  TYPE         EXTERNAL-IP   PORT(S)
nginx LoadBalancer <pending>     8000:31937/TCP
nginx LoadBalancer 35.189.65.99  8000:31937/TCP
```

The result is that Nginx is now accessible on the public Internet on port 8000 even though it is exposed on port 80 at the Pod level:

```
$ curl -s -i http://35.189.65.99:8000 | grep title
<title>Welcome to nginx!</title>
```

For completeness, here we have the declarative equivalent of the presented `kubectl expose` command; it is applied using the `kubectl apply -f serviceLoadBalancerMapped.yaml` command. We might need to delete the service created using the imperative approach first, by running `kubectl delete service/nginx`:

```
# serviceLoadBalancerMapped.yaml
kind: Service
apiVersion: v1
metadata:
  name: nginx
spec:
  selector:
    run: nginx
  ports:
  - protocol: TCP
    port: 8000
    targetPort: 80
  type: LoadBalancer
```

Exposing Multiple Ports

A Pod may expose multiple ports either because it contains multiple containers or because a single container listens on multiple ports. For example, a web server typically listens both on port 80 for regular, unencrypted traffic, and on port 443 for TLS traffic. The `spec.ports` attribute in the service manifest expects an array of port declarations, so all we have to do is append more elements to this array, keeping in mind that whenever two or more ports are defined, each must be given a unique name so that they can be disambiguated:

```
# serviceMultiplePorts.yaml
kind: Service
apiVersion: v1
metadata:
  name: nginx
spec:
  selector:
    run: nginx
  ports:
  - name: http  # user-defined name
    protocol: TCP
    port: 80
    targetPort: 80
  - name: https # user-defined name
    protocol: TCP
    port: 443
    targetPort: 443
  type: LoadBalancer
```

Canary Releases

The idea behind a canary release is that we expose a new version of a service only to a subset of users—before rolling it out to the entire userbase—so that we can observe the new service's behavior for a while until we are convinced that it does not present runtime defects.

An easy way to implement this strategy is by creating a Service object that—during the canary release—includes a new Pod, "the canary" in its load balanced cluster. For example, say that the production cluster includes three replicas of Pod version 1.0 in its current version, we can include an instance of the Pod v2.0 so that 1/4 of the traffic (in average) reaches the new Pod.

The key ingredients of this strategy are labels and selectors, which we have covered in Chapter 2. All we must do is add a label to the Pods that are meant to be in production and a matching selector in the service object. In this way, we can create the Service object in advance and let the Pods declare a label that will make them be selected by the Service object automatically. This is easier to see in action than to digest in words; let us follow this process step by step.

We first create a Service manifest whose selector will be looking for Pods whose label `prod` is equals to `true`:

```
# myservice.yaml
kind: Service
apiVersion: v1
metadata:
  name: myservice
spec:
  selector:
    prod: "true"
  ports:
  - protocol: TCP
```

```
    port: 80
    targetPort: 80
  type: LoadBalancer
```

After applying the manifest, we can keep a second window open in which we will see, interactively, which Endpoints join and leave the cluster:

```
$ kubectl apply -f myservice.yaml
service/myservice created
```

```
$ kubectl get endpoints/myservice -w
NAME        ENDPOINTS   AGE
myservice   <none>      29m
```

Given that we haven't created any Pods yet, there are no Endpoints in the cluster as evidenced by the value `<none>` under the `ENDPOINTS` column. Let us create the "incumbent" v1 production Deployment consisting of three replicas:

```
$ kubectl run v1 --image=nginx --port=80 \
    --replicas=3 --labels="prod=true"
deployment.apps/v1 created
```

If we check the terminal window in which we had left `kubectl get endpoints/myservice -w` running, we will notice that three new endpoints will be added. For example:

```
$ kubectl get endpoints/myservice -w
NAME      ENDPOINTS
myservice 10.36.2.10:80
myservice <none>
myservice 10.36.2.11:80
myservice 10.36.0.6:80,10.36.2.11:80
myservice 10.36.0.6:80,10.36.1.8:80,10.36.2.11:80
```

Since we have requested an external IP, we can check out that our v1 service is operational using `curl`:

```
$ kubectl get service/myservice
NAME      TYPE         EXTERNAL-IP   PORT(S)
myservice LoadBalancer 35.197.192.45 80:30385/TCP
```

```
$ curl -I -s http://35.197.192.45 | grep Server
Server: nginx/1.13.8
```

Now it is time to introduce a canary Pod. Let us create a v2 deployment. The differences are that the label `prod` is set to `false` and that we will be using the Apache server rather than Nginx as the container image for the new version:

```
$ kubectl run v2 --image=httpd --port=80 \
    --replicas=3 --labels="prod=false"
deployment.apps/v2 created
```

As of now, we can see that there are six Pod replicas in total. The `-L <LABEL>` displays the label's value:

```
$ kubectl get pods -L prod
NAME                  READY  STATUS   RESTARTS  PROD
v1-3781799777-219m3   1/1    Running  0         true
v1-3781799777-qc29z   1/1    Running  0         true
v1-3781799777-tbj4f   1/1    Running  0         true
v2-3597628489-2kl05   1/1    Running  0         false
v2-3597628489-p8jcv   1/1    Running  0         false
v2-3597628489-zc95w   1/1    Running  0         false
```

In order to get one of the v2 Pods into the `myservice` cluster, all we have to do is set the label accordingly. Here we pick the Pod named `v2-3597628489-2kl05` and set its `prod` label to `true`:

```
$ kubectl label pod/v2-3597628489-2kl05 \
    prod=true --overwrite
pod "v2-3597628489-2kl05" labeled
```

Right after the label operation, if we check the window in which the command `kubectl get endpoints/myservice -w` is running, we will see that an extra endpoint will have been added. At this moment, if we hit the public IP address repeatedly, we will notice that some of the requests land on the Apache web server:

```
$ while true ; do curl -I -s http://35.197.192.45 \
    | grep Server ; done
Server: nginx/1.13.8
Server: nginx/1.13.8
Server: nginx/1.13.8
Server: nginx/1.13.8
Server: Apache/2.4.29 (Unix)
Server: nginx/1.13.8
Server: nginx/1.13.8
...
```

Once we are happy with the behavior of v2, we can promote the rest of the v2 fleet to production. This can be accomplished on a step-by-step basis by applying labels as shown before; however, at this point, it is best to create a formal deployment manifest so that, once applied, Kubernetes takes care of introducing the v2 Pods and retiring the v1 ones in an seamless fashion—please refer to Chapter 3 for further details on rolling and blue/green Deployments.

To conclude this section, the combination of labels and selectors provide us with flexibility in terms of what Pods are exposed to service consumers. One practical application is the instrumentation of canary releases.

Canary Releases and Inconsistent Versions

A canary release may consist of an internal code enhancement—or bug fix—but it may also introduce new features to users. Such new features may relate both to the visual user interface itself (e.g., HTML and CSS) as well as to the data APIs that power such interfaces. Whenever the latter is the case, the fact that each request may land on any random Pod may be problematic. For example, the first request may retrieve a new v2 AngularJS code that relies on a v2 REST API, but when the second request hits the load balancer, the selected Pod may be v1 and serve an incorrection version of such said API. In essence, when a canary release introduces external changes—both UI and data wise—we usually want users to stay on the same version, either the current release or the canary one.

The technical term for users that land on the same instance of a service is *sticky sessions* or *session affinity*—the latter is the one used by Kubernetes. There are myriad of approaches to implementing session affinity depending on how much data is available to identify a single user. For example, cookies or session identifiers appended to the URL may be used in the scenario of web applications, but what if the interface is, say, Protocol Buffers or Thrift rather than HTTP? The only detail that can *somewhat* identify a given user from another is their client IP address, and this is exactly what the Service object can use to implement this behavior.

By default, session affinity is disabled. Implementing session affinity in an imperative context is simply a matter of adding the `--session-affinity=ClientIP` flag to the `kubectl expose` command. For example:

```
# Assume there is a Nginx Deployment running
$ kubectl expose deploy/nginx --type=LoadBalancer \
    --session-affinity=ClientIP
service "nginx" exposed
```

The declarative version involves setting the `service.spec.sessionAffinity` property and applying the manifest running the `kubectl apply -f serviceSessionAffinity.yaml` command:

```
# serviceSessionAffinity.yaml
kind: Service
apiVersion: v1
metadata:
  name: nginx
spec:
  sessionAffinity: ClientIP
  selector:
    run: nginx
  ports:
  - protocol: TCP
    port: 80
  type: LoadBalancer
```

The limitation of IP-based session affinity is that multiple users may share the same IP address as it is typically the case in companies and schools. Likewise, the same logical user may appear as originating from multiple IP addresses as in the case of a user who watches Netflix using their Wi-Fi broadband router at home and through their smartphone using LTE or similar. For such scenarios, the service's capabilities are insufficient; therefore, it is best to use a service that has layer 7 introspection capabilities such as the *Ingress* controller. For more information, refer to `https://kubernetes.io/docs/concepts/services-networking/ingress/`.

Graceful Startup and Shutdown

An application that benefits from the graceful startup property is only ready to accept user requests once its bootstrapping process is completed, whereas graceful shutdown means that an application should not stop abruptly—breaking its users or the integrity of its dependencies as a result. In other words, applications should be able to tell “I had my coffee and took my shower, I am ready to work” as well as “I call it a day. I am going to brush my teeth and go to bed now.”

The Service controller uses two main mechanisms to decide whether a given Pod should form part of its cluster: the Pod’s label and the Pod’s readiness status which is implemented using a Readiness probe (see Chapter 2). In a Spring-based Java application, for example, there are a few seconds of delay between the time that application is launched and the time the Spring Boot framework has fully initialized and is ready to accept http requests.

Zero-Downtime Deployments

Zero-downtime deployments are achieved by the Kubernetes’ Deployment controller by registering new Pods with the Service controller and removing old ones, in a coordinated fashion, so that the end user is always being served by a minimum number of Pod replicas. As explained in Chapter 3, zero-downtime deployments may be implemented either as rolling deployments or as blue/green ones.

Seeing a zero-downtime deployment in action takes only a few steps. First, we create a regular Nginx Deployment object and expose it through the public load balancer. We set `--session-affinity=ClientIP` so that the consuming client experiences a seamless transition to the new upgraded Pod(s) once ready:

```
$ kubectl run site --image=nginx --replicas=3
deployment.apps/site created

$ kubectl expose deploy/site \
    --port=80 --session-affinity=ClientIP \
    --type=LoadBalancer
service/site exposed

# Confirm public IP address
$ kubectl get services -w
NAME    TYPE          EXTERNAL-IP      PORT(S)
site    LoadBalancer  35.197.210.194   80:30534/TCP
```

We then open a separate terminal window that we will use to leave a simple http client running in an infinite loop:

```
$ while true ; do curl -s -I http://35.197.210.194/ \
    | grep Server; sleep 1 ; done
Server: nginx/1.17.1
Server: nginx/1.17.1
Server: nginx/1.17.1
...
```

Now all we have to do is transition `deploy/site` to a new deployment which can be achieved simply by changing its underlying container image. Let us use the Apache HTTPD image from Docker Hub:

```
$ kubectl set image deploy/site site=httpd
deployment.extensions/site image updated
```

If we go back to the window where the sample client is running, we will see that soon we will be greeted by the Apache HTTPD server rather than Nginx:

```
Server: nginx/1.17.1
Server: nginx/1.17.1
Server: nginx/1.17.1
Server: Apache/2.4.39 (Unix)
Server: Apache/2.4.39 (Unix)
Server: Apache/2.4.39 (Unix)
...
```

It also interesting to leave another, parallel, window open to monitor Pod activity using the `kubectl get pod -w` command so that we can observe how new Pods are being spun up and old ones are being terminated.

Pods' Endpoints

In most cases, the role of the Service controller is that of providing a single endpoint for two or more Pods. However, in certain circumstances, we may want to identify which are the specific endpoints of those Pods that the Service controller has selected.

The `kubectl get endpoints/<SERVICE-NAME>` command displays up to three endpoints directly on the screen, and it is used for immediate debugging purposes. For example:

```
$ kubectl get endpoints/nginx -o wide
NAME      ENDPOINTS
nginx     10.4.0.6:80,10.4.1.6:80,10.4.2.6:80
```

The same information can be retrieved using the `kubectl describe service/<SERVICE-NAME>` command as we have seen before. If we want a more programmatic approach that allows us to react to changes in the number and value of the endpoints, we can use a JSONPath query instead. For example:

```
$ kubectl get endpoints/nginx -o jsonpath \
    --template= "{.subsets[*].addresses[*].ip}"
10.4.0.6 10.4.1.6 10.4.2.6
```

Management Recap

As we have seen, services may be created imperatively using the `kubectl expose` command or declaratively using a manifest file. Services are listed using the `kubectl get services` command and deleted using the `kubectl delete service/<SERVICE-NAME>` command. For example:

```
$ kubectl get services
NAME        TYPE         CLUSTER-IP  EXTERNAL-IP
kubernetes  ClusterIP    10.7.240.1  <none>
nginx       LoadBalancer 10.7.241.102 35.186.156.253
```

```
$ kubectl delete services/nginx
service "nginx" deleted
```

Please note that when a service is deleted, the underlying Deployment will still keep running since they both have separate life cycles.

Summary

In this chapter, we learned that the Service controller helps create a layer of indirection between service consumers and Pods in order to facilitate instrumenting properties such as self-healing, canary releases, load balancing, graceful startup and shutdown, and zero-downtime deployments.

We covered specific connectivity use cases such as Pod-to-Pod, LAN-to-Pod, and Internet-to-Pod and saw that the latter is of special usefulness because it allows to reach our applications using a public IP address.

We also explained how to use full DNS records to disambiguate clashing services across different namespaces and the need to name ports when more than one is declared.