Why Services Are Not the Best Fit for External Access?

In this lesson, we will discover why services are not the best fit for enabling external access to the applications.

WE'LL COVER THE FOLLOWING

Only Services won't Suffice

We cannot explore solutions before we know what the problems are. Therefore, we'll re-create a few objects using the knowledge we already gained. That will let us see whether Kubernetes Services satisfy all the needs users of our applications might have. Or, to be more explicit, we'll explore which features we're missing when making our applications accessible to users.

We already discussed that it is a bad practice to publish fixed ports through Services. That method is likely to result in conflicts or, at the very least, create the additional burden of carefully keeping track of which port belongs to which Service. We already discarded that option before, and we won't change our minds now.

Since we've clarified that, let's go back and create the Deployments and the Services from the previous chapter.

```
kubectl create \
    -f ingress/go-demo-2-deploy.yml

kubectl get \
    -f ingress/go-demo-2-deploy.yml
```

```
-f ingress/go-demo-2-deploy.yml
```

The **output** of the `get` command is as follows.

```
NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/go-demo-2-db    0/1     1            0           2m15s

NAME                    TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)     AGE
service/go-demo-2-db    ClusterIP   10.111.211.179   <none>        27017/TCP   2m15s

NAME                             READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/go-demo-2-api    0/3     3            0           2m15s

NAME                     TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
service/go-demo-2-api    NodePort   10.103.180.226   <none>        8080:30753/TCP   2m15s
```

As you can see, these are the same Services and Deployments we previously created.

Before we move on, we should wait until all the Pods are up and running.

```
kubectl get pods
```

The **output** is as follows.

```
NAME                            READY STATUS  RESTARTS AGE
go-demo-2-api-68df567fb5-8qcmv 1/1   Running 0        3m
go-demo-2-api-68df567fb5-k55d4 1/1   Running 0        3m
go-demo-2-api-68df567fb5-ws9cj 1/1   Running 0        3m
go-demo-2-db-dd48b7dfc-hdxbz   1/1   Running 0        3m
```

If, in your case, some of the Pods are not yet running, please wait a few moments and re-execute the `kubectl get pods` command. We'll continue once they're ready.

Access Through Services

One obvious way to access the applications is through Services.

```
IP=$(minikube ip)

PORT=$(kubectl get svc go-demo-2-api \
    -o jsonpath="{.spec.ports[0].nodePort}")

curl -i "http://$IP:$PORT/demo/hello"
```

We retrieved the Minikube IP and the port of the `go-demo-2-api` Service. We used that information to send a request.

The **output** of the `curl` command is as follows.

```
HTTP/1.1 200 OK
Date: Sun, 24 Dec 2017 13:35:26 GMT
Content-Length: 14
Content-Type: text/plain; charset=utf-8

hello, world!
```

The application responded with the status code `200` thus confirming that the Service indeed forwards the requests.

While publishing a random, or even a hard-coded port of a single application might not be so bad, if we'd apply the same principle to more applications, the user experience would be horrible. To make the point a bit clearer, we'll deploy another application.

```
kubectl create \
    -f ingress/devops-toolkit-dep.yml \
    --record --save-config

kubectl get \
    -f ingress/devops-toolkit-dep.yml
```

This application follows similar logic to the first. From the latter command, we can see that it contains a Deployment and a Service. The details are of no importance since the YAML definition is very similar to those we used before. What matters is that now we have two applications running inside the cluster.

Understanding the Process

Let's check whether the new application is indeed reachable.

```
PORT=$(kubectl get svc devops-toolkit \
    -o jsonpath="{.spec.ports[0].nodePort}")

open "http://$IP:$PORT"
```

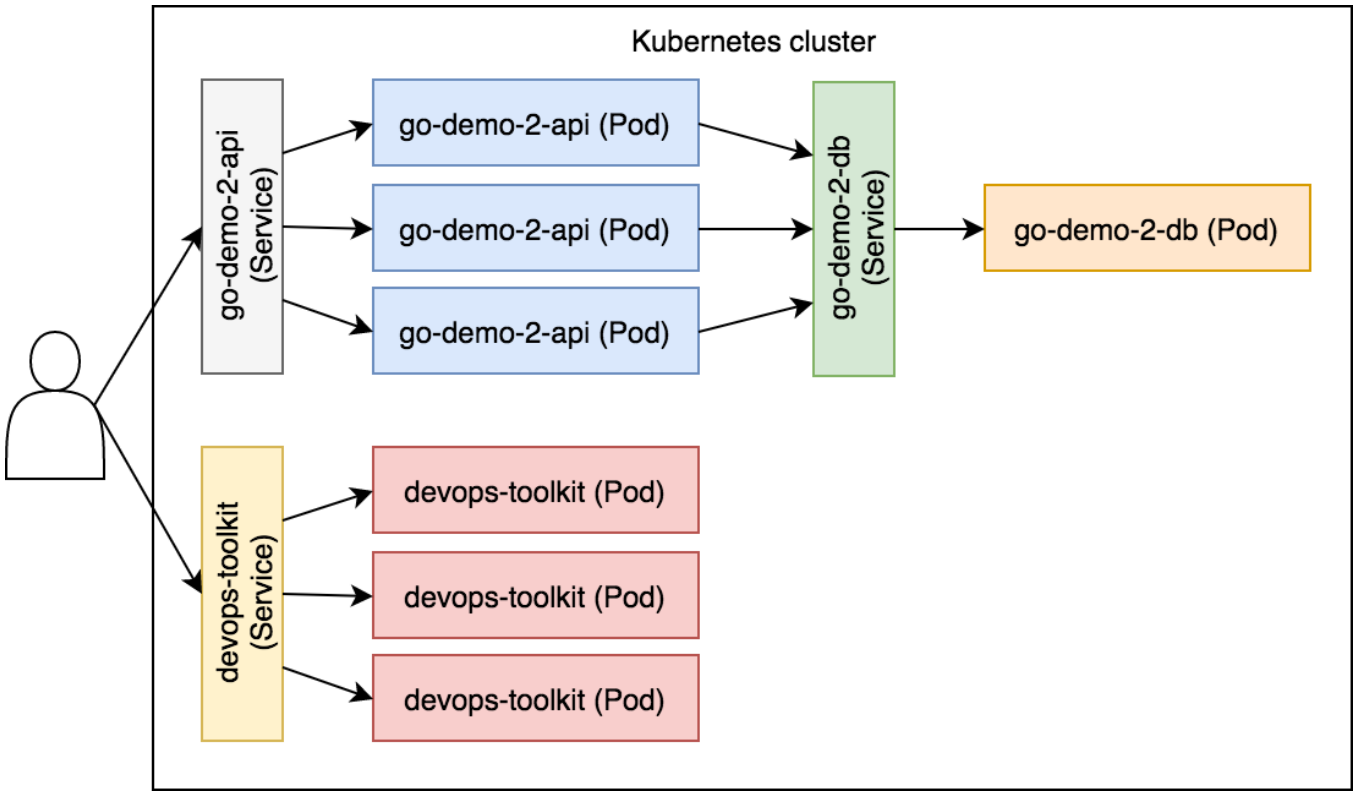
We retrieved the port of the new Service and opened the application in a

We retrieved the port of the new Service and opened the application in a browser. If you get a *page not found* error, you might want to wait a bit longer

until the containers are pulled, and try again

A simplified flow of requests is depicted in the below-given illustration.

A user sends a request to one of the nodes of the cluster. That request is received by a Service and load balanced to one of the associated Pods. It's a bit more complicated than that, with iptables, kube DNS, kube proxy, and a few other things involved in the process. We explored them in more detail in the *Using Services To Enable Communication Between Pods* chapter, and there's probably no need to go through them all again. For the sake of brevity, the simplified diagram should do.



Applications access through Services

We cannot expect our users to know specific ports behind each of those applications. Even with only two, that would not be very user-friendly. If that number would rise to tens or even hundreds of applications, our business would be very short-lived.

What we need is a way to make all services accessible through standard HTTP (`80`) or HTTPS (`443`) ports. Kubernetes Services alone cannot get us there. We need more.

The Solution

What we need is to grant access to our services on predefined paths and domains. Our `go-demo-2` service could be distinguished from others through the base path `/demo`. Similarly, the books application could be reachable through the `devopstoolkitseries.com` domain. If we could accomplish that, we could access them with the commands as follows.

```
curl "http://$IP/demo/hello"
```

The request received the `default backend - 404` response. There is no process listening on port `80`, so this outcome is not a surprise. We could have changed one of the Services to publish the fixed port `80` instead assigning a random one. Still, that would provide access only to one of the two applications.

We often want to associate each application with a different domain or sub-domain. Outside the examples we're running, the books application is accessible through the devopstoolkitseries.com domain. Since access to the domain is not feasible, we'll simulate it by adding the domain to the `Host` header.

The command that should verify whether the application running inside our cluster is accessible through the `devopstoolkitseries.com` domain is as follows.

```
curl -i \
    -H "Host: devopstoolkitseries.com" \
    "http://$IP"
```

As expected, the request is still refused.

SSL Certificates

Last, but not the least, we should be able to make some, if not all, applications (partly) secure by enabling HTTPS access. That means that we should have a place to store our SSL certificates. We could put them inside our applications, but that would only increase the operational complexity. Instead, we should aim towards SSL offloading somewhere between clients and the applications, and it should come as no surprise that Kubernetes has a solution for all these.

In the next lesson, we will try solving the problems discussed in this lesson by enabling Ingress controllers.