


Sequential Breakdown of the Process

In this lesson, we will go through the sequential processes kicked off by a Service creation.

WE'LL COVER THE FOLLOWING



- The Sequence
-  A note to the Windows users

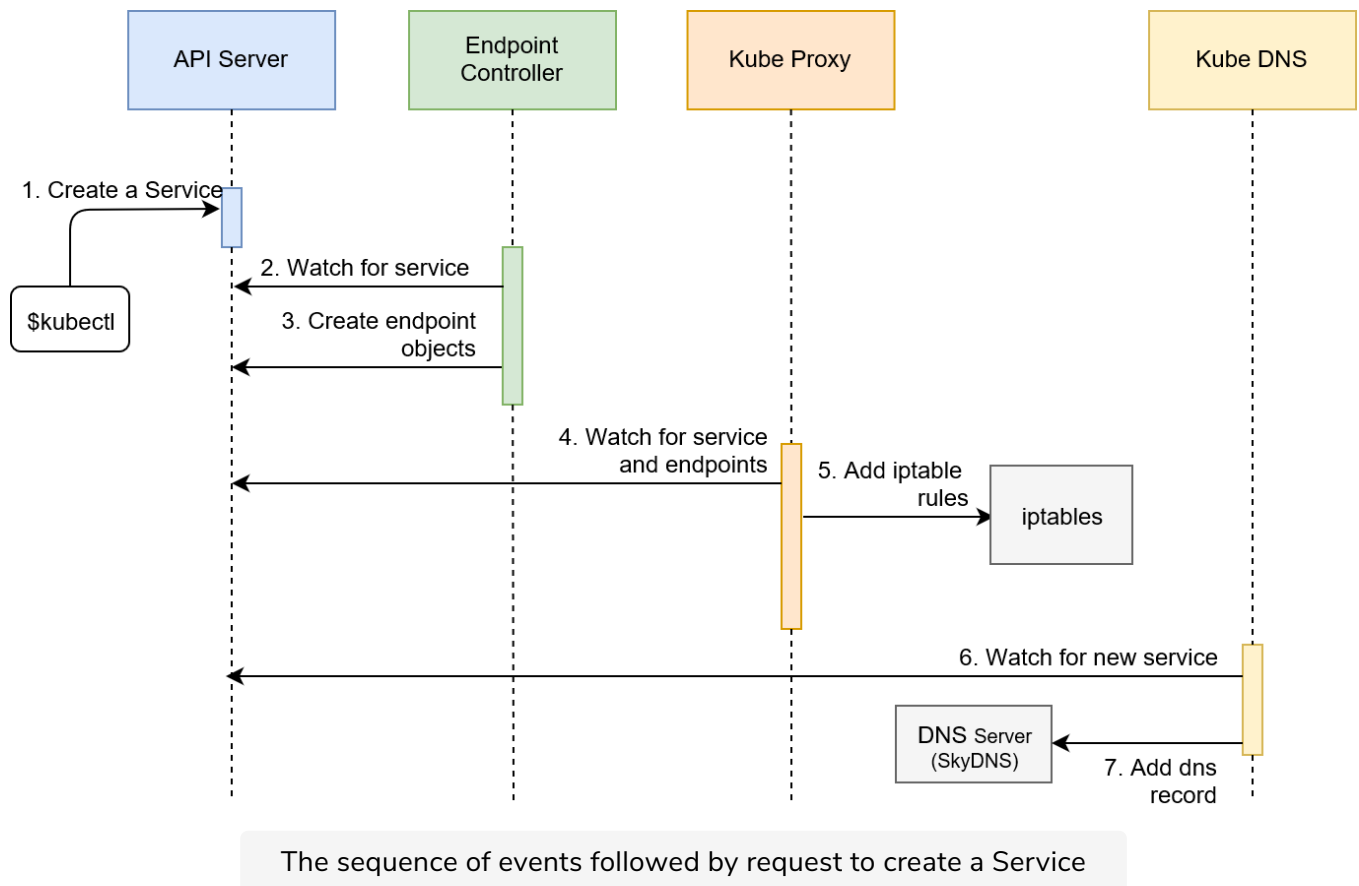
The Sequence

The processes that were initiated with the creation of the Service are as follows:

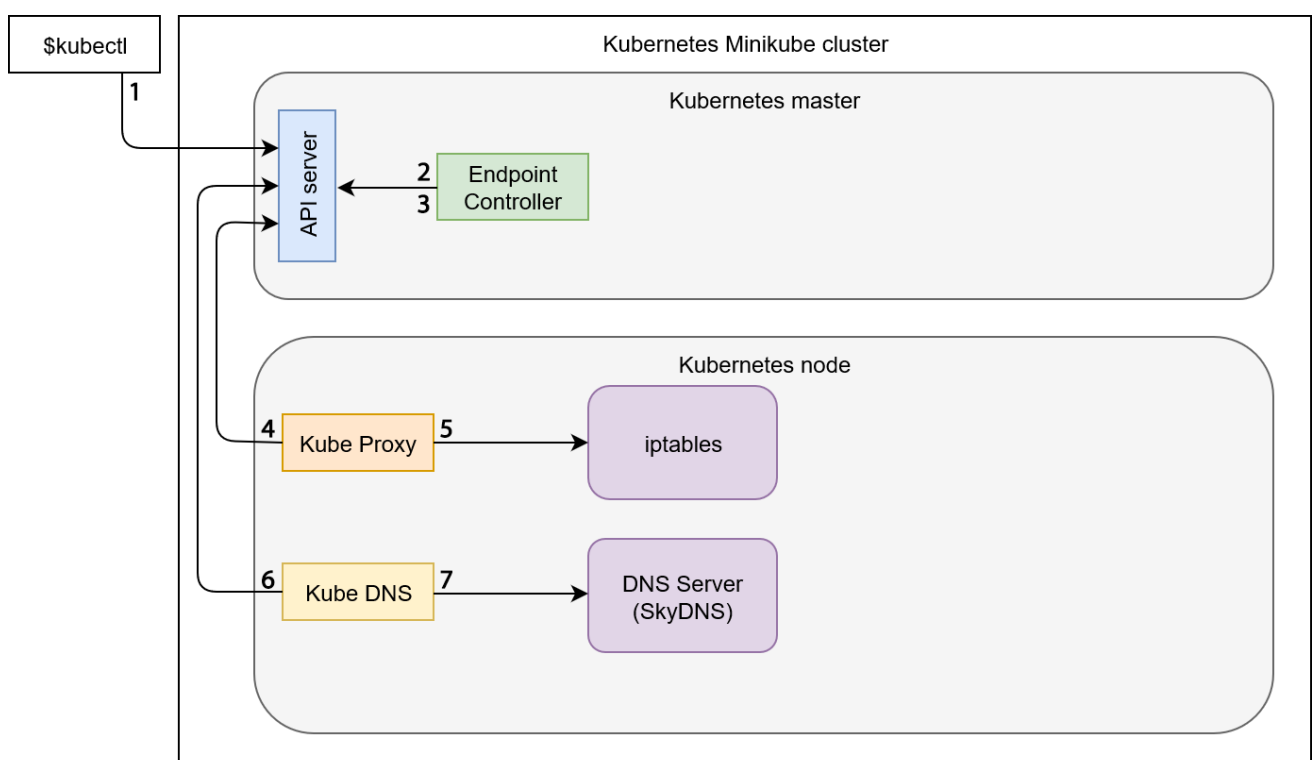
1. Kubernetes client (`kubectl`) sent a request to the API server requesting the creation of the Service based on Pods created through the go-demo-2 ReplicaSet.
2. Endpoint controller is watching the API server for new service events. It detected that there is a new Service object.
3. Endpoint controller created endpoint objects with the same name as the Service, and it used Service selector to identify endpoints (in this case the IP and the port of go-demo-2 Pods).
4. kube-proxy is watching for service and endpoint objects. It detected that there is a new Service and a new endpoint object.
5. kube-proxy added iptables rules which capture traffic to the Service port and redirect it to endpoints. For each endpoint object, it adds iptables rule which selects a Pod.
6. The kube-dns add-on is watching for Service. It detected that there is a new service.

7. The kube-dns added `db`'s record to the dns server (skydns)

7. The Kube-dns added a record to the dns server (skydns).



The sequence we described is useful when we want to understand everything that happened in the cluster from the moment we requested the creation of a new Service. However, it might be too confusing so we'll try to explain the same process through a diagram that more closely represents the cluster.



Let's take a look at our new Service.

```
kubectl describe svc go-demo-2-svc
```



The **output** is as follows.

```
Name:                go-demo-2-svc
Namespace:           default
Labels:              db=mongo
                    language=go
                    service=go-demo-2
                    type=backend
Annotations:         <none>
Selector:            service=go-demo-2,type=backend
Type:               NodePort
IP:                 10.0.0.194
Port:               <unset> 28017/TCP
TargetPort:         28017/TCP
NodePort:           <unset> 31879/TCP
Endpoints:          172.17.0.4:28017,172.17.0.5:28017
Session Affinity:    None
External Traffic Policy: Cluster
Events:             <none>
```



- **Line 1-2:** We can see the name and the namespace. We did not yet explore namespaces (coming up later) and, since we didn't specify any, it is set to `default`.
- **Line 3-6:** Since the Service is associated with the Pods created through the ReplicaSet, it inherited all their labels. The selector matches the one from the ReplicaSet. The Service is not directly associated with the ReplicaSet (or any other controller) but with Pods through matching labels.
- **Line 9-13:** Next is the `NodePort` type which exposes ports to all the nodes. Since `NodePort` automatically created `ClusterIP` type as well, all the Pods in the cluster can access the `TargetPort`. The `Port` is set to `28017`. That is the port that the Pods can use to access the Service. Since we did not specify it explicitly when we executed the command, its value is the same as the value of the `TargetPort`, which is the port of the associated Pod that will receive all the requests. `NodePort` was generated automatically since we did not set it explicitly. It is the port which we can use to access

the Service and, therefore, the Pods from outside the cluster. In most cases, it should be randomly generated, that way we avoid any clashes.

Let's see whether the Service indeed works.

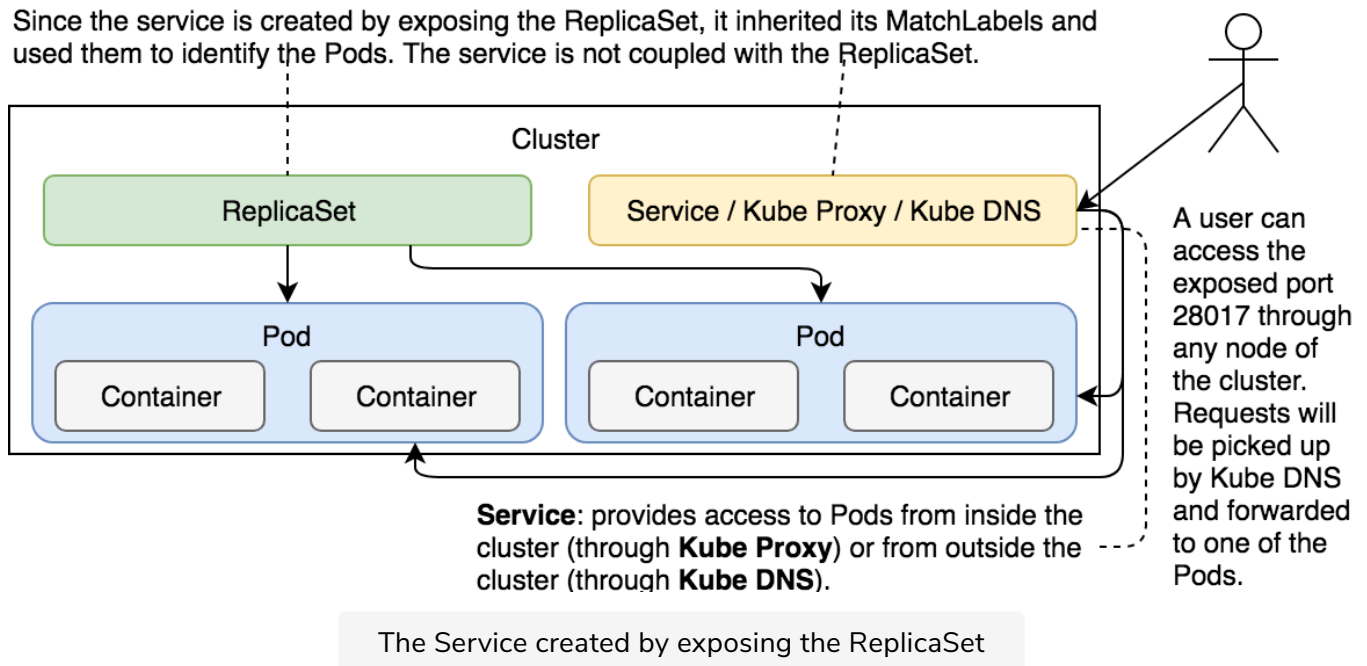
```
PORT=$(kubectl get svc go-demo-2-svc \
  -o jsonpath="{.spec.ports[0].nodePort}")
IP=$(minikube ip)
open "http://$IP:$PORT"
```

A note to the Windows users

Git Bash might not be able to use the `open` command. If that's the case, replace the `open` command with `echo`. As a result, you'll get the full address that should be opened directly in your browser of choice.

- **Line 1-2:** We used the filtered output of the `kubectl get` command to retrieve the `nodePort` and store it as the environment variable `PORT`.
- **Line 3:** We retrieved the IP of the minikube VM.
- **Line 4:** Finally, We opened MongoDB UI in a browser through the service port.

Since the service is created by exposing the ReplicaSet, it inherited its MatchLabels and used them to identify the Pods. The service is not coupled with the ReplicaSet.



As it has been already mentioned in the previous chapters, creating Kubernetes objects using imperative commands is **not a good idea** unless we're trying some quick hack.

The same applies to Services. Even though `kubectl expose` did the work, we

should try to use a documented approach through YAML files. In that spirit, we'll destroy the service we created and start over.

```
kubect1 delete svc go-demo-2-svc
```



Now that we have destroyed the Service, we will explore creating Services through declarative syntax in the next lesson.