Creating Services through Declarative Syntax

In this lesson, we will learn to create Services through declarative syntax.

WE'LL COVER THE FOLLOWING

Looking into the Syntax

We can accomplish a similar result as the one using `kubectl expose` through the `svc/go-demo-2-svc.yml` specification.

```
cat svc/go-demo-2-svc.yml
```

The **output** is as follows.

```
apiVersion: v1
kind: Service
metadata:
  name: go-demo-2
spec:
  type: NodePort
  ports:
  - port: 28017
    nodePort: 30001
    protocol: TCP
  selector:
    type: backend
    service: go-demo-2
```

- **Line 1-4:** You should be familiar with the meaning of `apiVersion`, `kind`, and `metadata`, so we'll jump straight into the `spec` section.

- **Line 5:** Since we already explored some of the options through the `kubectl expose` command, the `spec` should be relatively easy to grasp.
- **Line 6:** The type of the Service is set to `NodePort` meaning that the ports will be available both within the cluster as well as from outside by sending requests to any of the nodes.
- **Line 7-10:** The `ports` section specifies that the requests should be forwarded to the Pods on port `28017`. The `nodePort` is new. Instead of letting the service expose a random port, we set it to the explicit value of `30001`. Even though, in most cases, that is not a good practice, I thought it might be a good idea to demonstrate that option as well. The protocol is set to `TCP`. The only other alternative would be to use `UDP`. We could have skipped the protocol altogether since `TCP` is the default value but, sometimes, it is a good idea to leave things as a reminder of an option.
- **Line 11-13:** The `selector` is used by the Service to know which Pods should receive requests. It works in the same way as ReplicaSet selectors. In this case, we defined that the service should forward requests to Pods with labels `type` set to `backend` and `service` set to `go-demo`. Those two labels are set in the Pods `spec` of the ReplicaSet.

Creating the Service

Now that there's no mystery in the definition, we can proceed and create the Service.

```
kubectl create -f svc/go-demo-2-svc.yml
kubectl get -f svc/go-demo-2-svc.yml
```

We created the Service and retrieved its information from the API server. The **output** of the latter command is as follows.

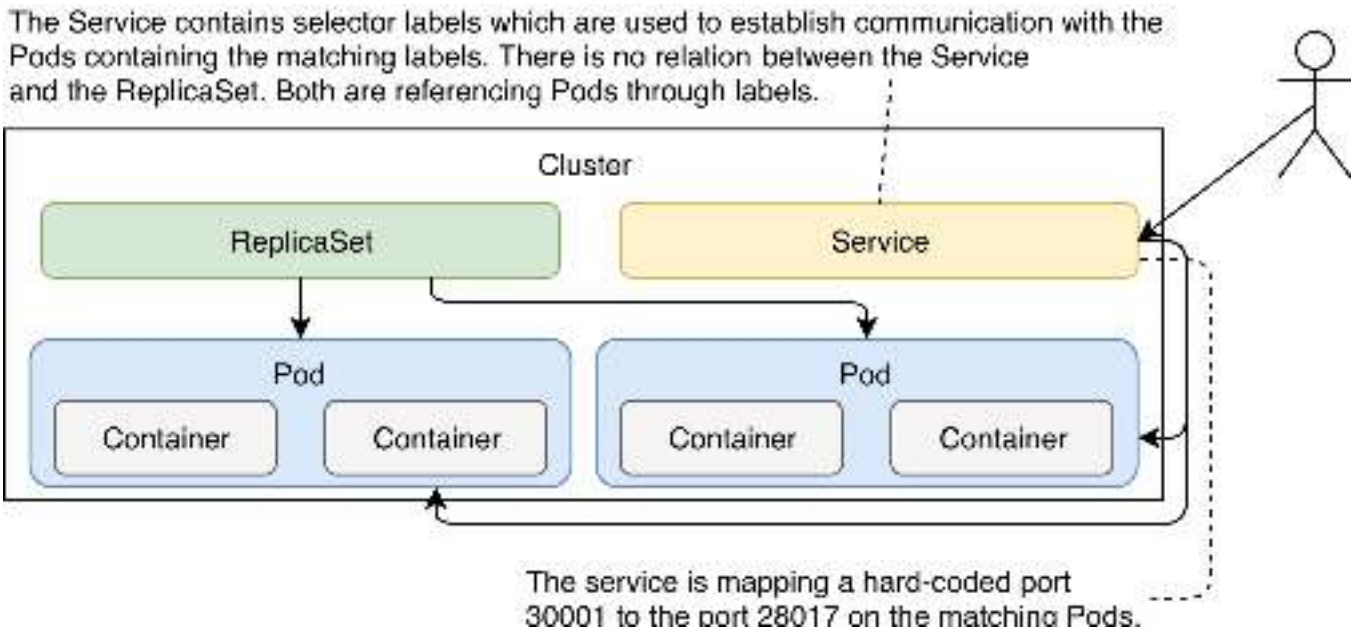
```
NAME      TYPE     CLUSTER-IP EXTERNAL-IP PORT(S)         AGE
go-demo-2 NodePort 10.0.0.129 <none>      28017:30001/TCP 10m
```

Now that the Service is running (again), we can double-check that it is working as expected by trying to access MongoDB UI.

working as expected by trying to access MongoDB UI.

```
open "http://$IP:30001"
```

Since we fixed the `nodePort` to `30001`, we did not have to retrieve the Port from the API server. Instead, we used the IP of the Minikube node and the hard-coded port `30001` to open the UI.



Let's take a look at the endpoint. It holds the list of Pods that should receive requests.

```
kubectl get ep go-demo-2 -o yaml
```

The **output** is as follows.

```
apiVersion: v1
kind: Endpoints
metadata:
  creationTimestamp: 2017-12-12T16:00:51Z
  name: go-demo-2
  namespace: default
  resourceVersion: "5196"
  selfLink: /api/v1/namespaces/default/endpoints/go-demo-2
  uid: a028b9a7-df55-11e7-a8ef-080027d94e34
subsets:
- addresses:
  - ip: 172.17.0.4
    nodeName: minikube
    targetRef:
      kind: Pod
      name: go-demo-2-j8kdw
      namespace: default
```

```
    resourceVersion: "5194"
    uid: ac70f868-df4d-11e7-a8ef-080027d94e34
- ip: 172.17.0.5

  nodeName: minikube
  targetRef:
    kind: Pod
    name: go-demo-2-5vlcc
    namespace: default
    resourceVersion: "5184"
    uid: ac7214d9-df4d-11e7-a8ef-080027d94e34
ports:
- port: 28017
  protocol: TCP
```

We can see that there are two subsets, corresponding to the two Pods that contain the same labels as the Service `selector`.

Request Forwarding

Each Pod has a unique IP that is included in the algorithm used when forwarding requests. Actually, it's not much of an algorithm. Requests will be sent to those Pods randomly. That randomness results in something similar to round-robin load balancing. If the number of Pods does not change, each will receive an approximately equal number of requests.

Random requests forwarding should be enough for most use cases. If it's not, we'd need to resort to a third-party solution. However soon, when the newer Kubernetes versions get released, we'll have an alternative to the *iptables* solution. We'll be able to apply different types of load balancing algorithms like last connection, destination hashing, newer queue, and so on. Still, the current solution is based on *iptables*, and we'll stick with it, for now.

Now We Can Split

So far, we have repeated a few times that our current Pod design is flawed. We have two containers (an API and a database) packaged together. This prevents us from scaling one without the other. Now that we learned how to use Services, we can redesign our Pod solution.

Destroying Everything

Before we move on, we'll delete the Service and the ReplicaSet we created.

```
kubectl delete -f svc/go-demo-2-svc.yml
kubectl delete -f svc/go-demo-2-rs.yml
```

```
kubectl delete -f svc/go-demo-2-svc.yml
```

Both the ReplicaSet and the Service are gone, and we can start anew.

In the next lesson, we will split the Pods and establish communication between them through Services.