Creating the Split API Pods

In this lesson, we will create API Pods using ReplicaSet and establish communication by creating Service.

WE'LL COVER THE FOLLOWING

Looking into the Definition

Moving to the backend API...

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

The **output** is as follows.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: go-demo-2-api
spec:
  replicas: 3
  selector:
    matchLabels:
      type: api
      service: go-demo-2
  template:
    metadata:
      labels:
        type: api
        service: go-demo-2
        language: go
    spec:
      containers:
```

```
      containers:
      - name: api
        image: vfarcic/go-demo-2
        env:
        - name: DB
          value: go-demo-2-db
        readinessProbe:
          httpGet:
            path: /demo/hello
            port: 8080
          periodSeconds: 1
        livenessProbe:
          httpGet:
            path: /demo/hello
            port: 8080
```

Just as with the database, this ReplicaSet should be familiar since it's very similar to the one we used before. We'll comment only on the differences.

- **Line 6:** The number of `replicas` is set to `3`. That solves one of the main problems we had with the previous ReplicaSets that defined Pods with both containers. Now the number of replicas can differ, and we have one Pod for the database, and three for the backend API.

- **Line 14:** In the `labels` section, `type` label is set to `api` so that both the ReplicaSet and the (soon to come) Service can distinguish the Pods from those created for the database.

- **Line 22-23:** We have the environment variable `DB` set to `go-demo-2-db`. The code behind the `vfarcic/go-demo-2` image is written in a way that the connection to the database is established by reading that variable. In this case, we can say that it will try to connect to the database running on the DNS `go-demo-2-db`. If you go back to the database Service definition, you'll notice that its name is `go-demo-2-db` as well. If everything works correctly, we should expect that the DNS was created with the Service and that it'll forward requests to the database.

The readinessProbe

The `readinessProbe` should be used as an indication that the service is ready to serve requests. When combined with `Services` construct, only containers with the `readinessProbe` state set to `Success` will receive requests.

In earlier Kubernetes versions it used `userspace` proxy mode. Its advantage is that the proxy would retry failed requests to another Pod. With the shift to the `iptables` mode, that feature is lost. However, `iptables` are much faster and `iptables` mode, that feature is lost. However, `iptables` are much faster and more reliable, so the loss of the retry mechanism is well compensated. That

does not mean that the requests are sent to Pods “blindly”. The lack of the retry mechanism is mitigated with `readinessProbe`, which we added to the ReplicaSet.

The `readinessProbe` has the same fields as the `livenessProbe`. We used the same values for both, except for the `periodSeconds`, where instead of relying on the default value of `10`, we set it to `1`.

While `livenessProbe` is used to determine whether a Pod is alive or it should be replaced by a new one, the `readinessProbe` is used by the `iptables`. A Pod that does not pass the `readinessProbe` will be excluded and will not receive requests. In theory, requests might be still sent to a faulty Pod, between two iterations. Still, such requests will be small in number since the `iptables` will change as soon as the next probe responds with HTTP code less than `200`, or equal or greater than `400`.

Creating the ReplicaSet

Let’s create the ReplicaSet.

```
kubectl create \
    -f svc/go-demo-2-api-rs.yml
```

Creating the Service

Only one object is missing, that is Service.

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

The **output** is as follows.

```
apiVersion: v1
kind: Service
metadata:
  name: go-demo-2-api
spec:
  type: NodePort
  ports:
  - port: 8080
  selector:
    type: api
    type: api
    service: go-demo-2
```

There’s nothing truly new in this definition. The `type` is set to `NodePort` since the API should be accessible from outside the cluster. The `selector` label `type` is set to `api` so that it matches the labels defined for the Pods.

That is the last object we’ll create (in this section), so let’s move on and do it.

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

We’ll take a look at what we have in the cluster.

```
kubectl get all
```

The **output** is as follows.

```
NAME                          READY   STATUS    RESTARTS   AGE
pod/go-demo-2-api-r55fs       1/1     Running   0          3m32s
pod/go-demo-2-api-sng48       1/1     Running   0          3m32s
pod/go-demo-2-api-vvcbp       1/1     Running   0          3m32s
pod/go-demo-2-db-bwvkb        1/1     Running   0          4m20s

NAME                        TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
service/go-demo-2-api       NodePort    10.110.71.67    <none>        8080:31148/TCP   3m23s
service/go-demo-2-db        ClusterIP   10.104.40.176   <none>        27017/TCP        4m1s
service/kubernetes          ClusterIP   10.96.0.1       <none>        443/TCP          16m

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/go-demo-2-api       3         3         3       3m33s
replicaset.apps/go-demo-2-db        1         1         1       4m20s
```

Both ReplicaSets for db and api are there, followed by the three replicas of the `go-demo-2-api` Pods and one replica of the `go-demo-2-db` Pod. Finally, the two Services are running as well, together with the one created by Kubernetes itself.

Accessing the API

Before we proceed, it might be worth mentioning that the code behind the `vfarcic/go-demo-2` image is designed to fail if it cannot connect to the database. The fact that the three replicas of the `go-demo-2-api` Pod are running means that the communication is established. The only verification left is to check whether we can access the API from outside the cluster.

left is to check whether we can access the API from outside the cluster.

Let’s try that out.

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

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

We retrieved the port of the service (we still have the Minikube node `IP` from before) and used it to send a request.

The **output** of the last command is as follows.

```
HTTP/1.1 200 OK
Date: Tue, 12 Dec 2017 21:27:51 GMT
Content-Length: 14
Content-Type: text/plain; charset=utf-8

hello, world!
```

We got the response `200` and a friendly `hello, world!` message indicating that the API is indeed accessible from outside the cluster.

Destroying Services

Before we move further, we’ll delete the objects we created.

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

Everything we created is gone, and we can start over.

At this point, you might be wondering whether it is overkill to have four YAML files for a single application. Can’t we simplify the definitions? Not really. Can we define everything in a single file? Read the next lesson.