Comparison with Docker Swarm

In this lesson, we will compare Kubernetes Pods, ReplicaSets and Services with Docker Swarm equivalents.

WE'LL COVER THE FOLLOWING

- Comparing Pods, ReplicaSets and Services
  - Looking into Kubernetes Definition
  - Looking into Docker Swarm Definition
  - Pods
  - Replica Sets
  - Services
  - Conclusion

Comparing Pods, ReplicaSets and Services

Starting from this chapter, we'll compare each Kubernetes feature with Docker Swarm equivalents. That way, Swarm users can have a smoother transition into Kubernetes or, depending on their goals, choose to stick with Swarm.

Please bear in mind that the comparisons will be made only for a specific set of features. You will not (yet) be able to conclude whether Kubernetes is better or worse than Docker Swarm. You'll need to grasp both products in their entirety to make an educated decision. The comparisons like those that follow are useful only as a base for more detailed examinations of the two products.

For now, we'll limit the comparison scope to Pods, ReplicaSets, and Services on the one hand, and Docker Service stacks, on the other.

Looking into Kubernetes Definition

Let's start with Kubernetes file go-demo-2.yml (the same one we used before).

The definition is as follows.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: go-demo-2-db
spec:
  selector:
    matchLabels:
      type: db
      service: go-demo-2
  template:
    metadata:
      labels:
        type: db
        service: go-demo-2
    spec:
      containers:
      - name: db
        image: mongo:3.3
        ports:
        - containerPort: 28017

---

apiVersion: v1
kind: Service
metadata:
  name: go-demo-2-db
spec:
  ports:
  - port: 27017
  selector:
```

```
    type: db
    service: go-demo-2

---

apiVersion: apps/v1beta2
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
    spec:
      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

---

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

Looking into Docker Swarm Definition

Now, let's take a look at the Docker stack defined in go-demo-2-swarm.yml.

The specification is as follows.

```
version: "3"
```

```
services:
  api:
    image: vfarcic/go-demo-2
    environment:
      - DB=db
    ports:
      - 8080
    deploy:
      replicas: 3
  db:
    image: mongo
```

Pods

Both definitions accomplish the same result. There is no important difference from the functional point of view, except in Pods. Docker does not have the option to create something similar. When Swarm services are created, they are spread across the cluster, and there is no easy way to specify that multiple containers should run on the same node. Whether multi-container Pods are useful or not is something we'll explore later. For now, we'll ignore that feature.

Replica Sets

If we execute something like `docker stack deploy -c svc/go-demo-2-swarm.yml go-demo-2`, the result would be equivalent to what we got when we run `kubectl create -f svc/go-demo-2.yml`. In both cases, we get three replicas of `vfarcic/go-demo-2`, and one replica of `mongo`. Respective schedulers are making sure that the desired state (almost) always matches the actual state. Networking communication through internal DNSes is also established with both solutions. Each node in a cluster would expose a randomly defined port that forwards requests to the `api`. All in all, there are no functional differences between the two solutions.

Services

When it comes to the way services are defined, there is indeed, a considerable difference. Docker's stack definition is much more compact and straight-forward. We defined, in twelve lines, what took around eighty lines in the Kubernetes format.

One might argue that Kubernetes YAML file could have been smaller. Maybe it could. Still, it'll be bigger and more complex no matter how much we simplify it. One might also say that Docker's stack is missing `readinessProbe` and

`livenessProbe`. Yes it is, and that is because we decided not to put it there,

because the `vfarcic/go-demo-2` image already has `HEALTHCHECK` definition that Docker uses for similar purposes. In most cases, Dockerfile is a better place to define health checks than a stack definition. That does not mean that it cannot be set, or overwritten, in a YAML file. It can, when needed. But, that is not the case in this example.

Conclusion

All in all, if we limit ourselves only to Kubernetes Pods, ReplicaSets, and Services, and their equivalents in Docker Swarm, the latter wins due to a much simpler and more straightforward way to define specs. From the functional perspective, both are very similar.

Should you conclude that Swarm is a better option than Kubernetes? Not at all. At least, not until we compare other features. Swarm won the battle, but the war has just begun. As we progress, you'll see that there's much more to Kubernetes. We only scratched the surface.