Comparison with Docker Swarm

In this lesson, we will compare Kubernetes Deployments with Docker Swarm Stacks.

WE'LL COVER THE FOLLOWING

Kubernetes Deployments Compared To Docker Swarm Stacks

If you have already used Docker Swarm, the logic behind Kubernetes Deployments should be familiar. Both serve the same purpose and can be used to deploy new applications or update those that are already running inside a cluster. In both cases, we can easily deploy new releases without any downtime (when application architecture permits that).

Comparing Definitions

Let's have a look at the way we define objects in both Kubernetes and Docker Swarm.

Kubernetes

An example Kubernetes Deployment and Service definition is as follows.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: go-demo-2-db
spec:
  selector:
    matchLabels:
      type: db
      service: go-demo-2
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        type: db
        service: go-demo-2
        vendor: MongoLabs
    spec:
      containers:
      - name: db
        image: mongo:3.3
        ports:
        - containerPort: 27017

---

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: Deployment
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:
      - 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
```

Docker Swarm

An equivalent Docker Swarm stack definition is as follows.

```
version: "3"
services:
  api:
    image: vfarcic/go-demo-2
    environment:
      - DB=db
    ports:
      - 8080
    deploy:
      replicas: 3
  db:
    image: mongo:3.3
```

Both definitions provide, more or less, the same functionality.

It is evident that a Kubernetes Deployment requires a much longer definition with more a complex syntax. It is worth noting that Swarm's equivalent to `readinessProbe` and `livenessProbe` is not present in the stack because it is defined as a `HEALTHCHECK` inside the Dockerfile. Still, even if we remove them, a Kubernetes Deployment remains longer and more complicated.

> When comparing only the differences in the ways to define objects, Docker Swarm is a clear **winner**.

Comparing Functionality

Let's see what we can conclude from the functional point of view.

Creating Objects

Creating the objects is reasonably straight-forward. Both `kubectl create` and `docker stack deploy` will deploy new releases without any downtime. New containers or, in case of Kubernetes, Pods will be created and, in parallel, the old ones will be terminated. So far, both solutions are, more or less, the same.

Deployment Status

One of the **main differences** is what happens in case of a failure. A Kubernetes Deployment will not perform any corrective action in case of a failure. It will stop the update, leaving a combination of new and old Pods running in parallel. Docker Swarm, on the other hand, can be configured to rollback automatically. That might seem like another win for Docker Swarm.

However, Kubernetes has something Swarm doesn't. We can use `kubectl rollout status` command to find out whether the update was a success or failure and, in case of the latter, we can `undo` the `rollout`. Even though we need a few commands to accomplish the same result, that might fare better when updates are automated. Knowing whether an update succeeded or failed allows us to not only execute a subsequent rollback action but also notify someone that there is a problem.

> Both approaches have their pros and cons. Docker Swarm's automated

> Both approaches have their pros and cons. Docker Swarm's automated rollback is better suited in some cases, and Kubernetes update status works better in others. The methods are different, and there is no clear winner, so we'll proclaim it a **tie**.

Rolling Back

Kubernetes Deployments can record history. We can use the `kubectl rollout history` command to inspect past rollout. When updates are working as expected, `history` is not very useful. But, when things go wrong, it might provide additional insight. That can be combined with the ability to rollback to a specific revision, not necessarily the previous one. However, most of the time, we rollback to the previous version. The ability to go back further in time is not very useful. Even when such a need arises, both products can do that.

The **difference** is that Kubernetes Deployments allow us to go to a specific revision (e.g., we're on the revision five, rollback to the revision two). With Docker Swarm, we'd have to issue a new update (e.g., update the image to the tag 2.0). Since containers are immutable, the result is the same, so the difference is only in the syntax behind a rollback.

> The ability to rollback to a specific version or a tag exists in both products. We can argue which syntax is more straightforward or more useful. The differences are minor and we'll proclaim that there is no winner for that functionality. It's another **tie**.

Updating Multiple Deployments

Since almost everything in Kubernetes is based on label selectors, it has a feature that Docker Swarm doesn't. We can update multiple Deployments at the same time. We can, for example, issue an update (`kubetl set image`) that uses filters to find all Mongo databases and upgrade them to a newer release. It is a feature that would require a few lines of bash scripting with Docker Swarm.

However, while the ability to update all Deployments that match specific labels might sound like a useful feature, it often isn't. More often than not,

such actions can produce undesirable effects. If, for example, we have five back-end applications that use Mongo database (one for each), we'd probably want to upgrade them in a more controlled fashion. Teams behind those services would probably want to test each of those upgrades and give their blessings. We probably wouldn't wait until all are finished, but upgrade a single database when the team in charge of it feels confident.

> There are the cases when updating multiple objects is useful so we must give this one to Kubernetes. It is a minor **win**, but it still counts.

Conclusion

Don't make a decision based on the differences between Kubernetes Deployments and Docker Swarm stacks.

Definition syntax is where Swarm has a clear win, while on the functional front Kubernetes has a tiny edge over Swarm.

In any case, Kubernetes has much more to offer, and any conclusion based on such a limited comparison scope is bound to be incomplete. We only scratched the surface. Stay tuned for more.

In the next chapter, we'll learn using Ingress to forward traffic.