Get Started with Auto-Scaling Pods

In this lesson, we will first deploy an app and see how to change the number of replicas based on memory, CPU, or other metrics through `HorizontalPodAutoScaler` resource.

WE'LL COVER THE FOLLOWING

- Deploy an application
  - Create resources
- Where to set replicas?
  - Scale or Descale
    - Current number of replicas below `minReplicas`
    - Confirm desired replicas are running

Our goal is to deploy an application that will be automatically scaled (or de-scaled) depending on its use of resources. We'll start by deploying an app first, and discuss how to accomplish auto-scaling later.

> I already warned you that I assume that you are familiar with Kubernetes and that in this course we'll explore a particular topic of monitoring, alerting, scaling, and a few other things. We will not discuss Pods, StatefulSets, Deployments, Services, Ingress, and other "basic" Kubernetes resources.

Deploy an application

Let's take a look at a definition of the application we'll use in our examples.

```
cat scaling/go-demo-5-no-sidecar-mem.yml
```

If you are familiar with Kubernetes, the YAML definition should be self-explanatory. We'll comment on only the parts that are relevant for auto-scaling.

The **output**, limited to the relevant parts, is as follows.

```
...
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: db
  namespace: go-demo-5
spec:
  ...
  template:
    ...
    spec:
      ...
      containers:
      - name: db
        ...
        resources:
          limits:
            memory: "150Mi"
            cpu: 0.2
          requests:
            memory: "100Mi"
            cpu: 0.1
        ...
      - name: db-sidecar
        ...

apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
  namespace: go-demo-5
spec:
  ...
  template:
    ...
    spec:
      containers:
      - name: api
        ...
        resources:
          limits:
            memory: 15Mi
            memory: 15Mi
            cpu: 0.1
          requests:
            memory: 10Mi
            cpu: 0.01
...
```

We have two Pods that form an application. The `api` Deployment is a backend API that uses `db` StatefulSet for its state.

The essential parts of the definition are `resources`. Both the `api` and the `db` have `requests` and `limits` defined for memory and CPU. The database uses a sidecar container that will join MongoDB replicas into a replica set. Please note that, unlike other containers, the sidecar does not have `resources`. The importance behind that will be revealed later. For now, just remember that two containers have the `requests` and the `limits` defined and that one doesn't.

Create resources

Now, let's create those resources.

```
kubectl apply \
    -f scaling/go-demo-5-no-sidecar-mem.yml \
    --record
```

The **output** should show that quite a few resources were created and our next action is to wait until the `api` Deployment is rolled out thus confirming that the application is up-and-running.

```
kubectl -n go-demo-5 \
    rollout status \
    deployment api
```

After a few moments, you should see the message stating that `deployment "api"` was `successfully rolled out`.

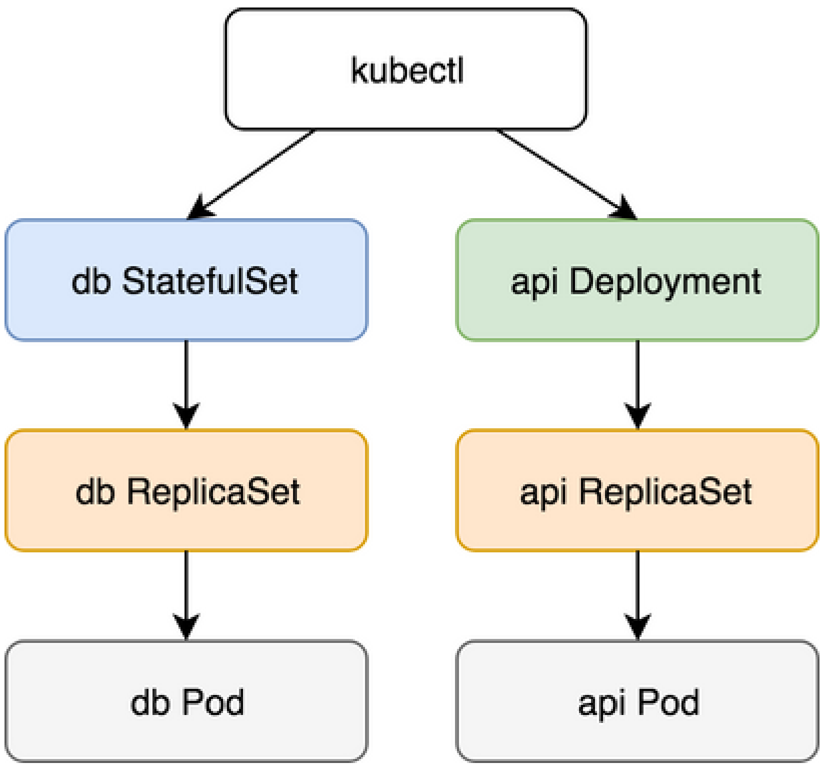
To be on the safe side, we'll list the Pods in the `go-demo-5` Namespace and confirm that one replica of each is running.

```
kubectl -n go-demo-5 get pods
```

The **output** is as follows.

```
NAME     READY STATUS  RESTARTS AGE
api-... 1/1   Running 0        1m
db-0     2/2   Running 0        1m
```

So far, we have not yet done anything beyond the ordinary creation of the StatefulSet and the Deployment. They, in turn, created ReplicaSets, which resulted in the creation of the Pods.



Creation of the StatefulSet and the Deployment

As you hopefully know, we should aim at having at least two replicas of each Pod, as long as they are scalable. Still, neither of the two had `replicas` defined. That is intentional. The fact that we can specify the number of replicas of a Deployment or a StatefulSet does not mean that we should. At least, not always.

Where to set replicas?

If the number of replicas is static and you have no intention to scale (or de-scale) your application over time, set `replicas` as part of your Deployment or StatefulSet definition. If, on the other hand, you plan to change the number of replicas based on memory, CPU, or other metrics, use `HorizontalPodAutoscaler` resource instead.

Let's take a look at a simple example of a `HorizontalPodAutoscaler`.

```
cat scaling/go-demo-5-api-hpa.yml
```

The **output** is as follows.

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: api
  namespace: go-demo-5
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: api
  minReplicas: 2
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 80
  - type: Resource
    resource:
      name: memory
      targetAverageUtilization: 80
```

The definition uses `HorizontalPodAutoscaler` targeting the `api` Deployment. Its boundaries are a minimum of two and a maximum of five replicas. Those limits are fundamental. Without them, we'd run a risk of scaling up into infinity or scaling down to zero replicas. The `minReplicas` and `maxReplicas` fields are a safety net.

Scale or Descale

The key section of the definition is `metrics`. It provides formulas Kubernetes should use to decide whether it should scale (or de-scale) a resource. In our case, we're using the `Resource` type entries. They are targeting the average utilization of eighty percent for memory and CPU. If the actual usage of either of the two deviates, Kubernetes will scale (or de-scale) the resource.

> Please note that we used the `v2beta1` version of the API and you might be wondering why we chose that one instead of the stable and production-ready v1. After all, `beta 1` releases are still far from being polished enough for general usage. The reason is simple. HorizontalPodAutoscaler v1 is too basic. It only allows scaling based on the CPU. Even our simple example goes beyond that by adding memory to the mix. Later on, we'll extend it even more. So, while v1 is considered stable, it does not provide much value, and we can either wait until v2 is released or start experimenting with v2beta releases right away. We are opting for the latter option. By the time you read this, more stable releases are likely to exist and to be supported in your Kubernetes cluster. If that's the case, feel free to change `apiVersion` before applying the definition.

Let's apply the definition that creates the `HorizontalPodAutoscaler (HPA)`.

```
kubectl apply \
    -f scaling/go-demo-5-api-hpa.yml \
    --record
```

Next, we'll take a look at the information we'll get by retrieving the `HPA` resources.

```
kubectl -n go-demo-5 get hpa
```

If you were quick, the **output** should be similar to the one that follows.

```
NAME REFERENCE      TARGETS                      MINPODS MAXPODS REPLICAS
AGE
api  Deployment/api <unknown>/80%, <unknown>/80% 2       5       0
20s
```

We can see that Kubernetes does not yet have the actual CPU and memory utilization and that its **output** is `<unknown>` instead. We need to give it a bit more time until the next iteration of data gathering from the `Metrics Server`. Get yourself some coffee before we repeat the same query.

Now let's repeat the query.

```
kubectl -n go-demo-5 get hpa
```

This time, the **output** is without unknowns.

```
NAME REFERENCE      TARGETS          MINPODS MAXPODS REPLICAS AGE
api  Deployment/api 38%/80%, 10%/80% 2       5       2        1m
```

Current number of replicas below `minReplicas`

We can see that both CPU and memory utilization are way below the expected utilization of `80%`. Still, Kubernetes increased the number of replicas from one to two because that's the minimum we defined. We made the contract stating that the `api` Deployment should never have less than two replicas, and Kubernetes complied with that by scaling up even if the resource utilization is way below the expected average utilization. We can confirm that behavior through the events of the `HorizontalPodAutoscaler`.

```
kubectl -n go-demo-5 describe hpa api
```

The **output**, limited to the event messages, is as follows.

```
...
Events:
... Message
... -------
... New size: 2; reason: Current number of replicas below Spec.MinReplicas
```

The message of the event should be self-explanatory. The `HorizontalPodAutoscaler` changed the number of replicas to `2` because the current number (1) was below the `MinReplicas` value.

Confirm desired replicas are running

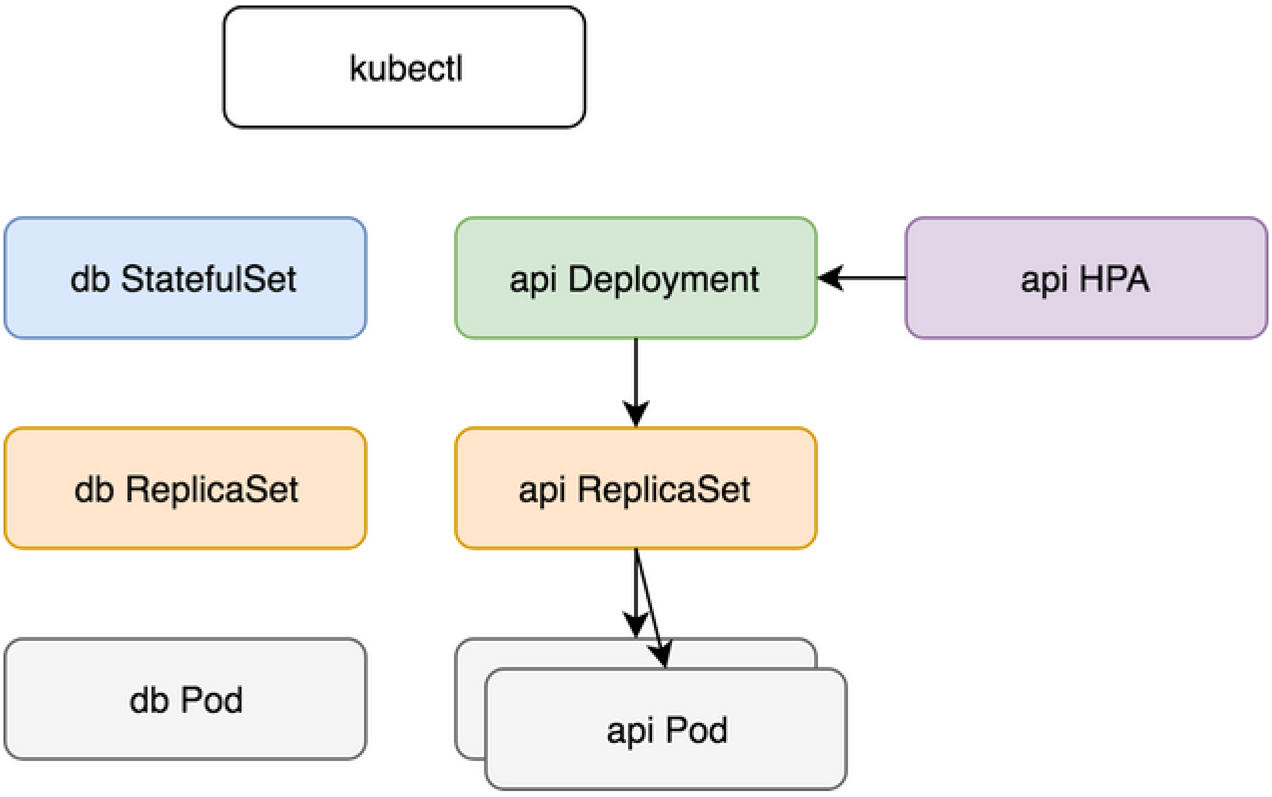
Finally, we'll list the Pods to confirm that the desired number of replicas are indeed running.

```
kubectl -n go-demo-5 get pods
```

The **output** is as follows.

```
NAME     READY STATUS   RESTARTS AGE
api-... 1/1    Running 0         2m
api-... 1/1    Running 0         6m
db-0     2/2    Running 0         6m
```

So far `HPA` only increased the number of **api** Pod to meet the specified minimum. It did so by manipulating the Deployment.



Scaling of the Deployment based on minimum number of replicas specified in the HPA.

Q

When both CPU and memory utilization were way below the expected utilization of `80%`, Kubernetes increased the number of replicas from one to two.

COMPLETED 0%

1 of 1

In the next lesson, we will see how the `HPA` performs auto-scaling based on resource usage.