## Combine Metric Server Data with Custom Metrics

In this lesson, we will discuss how to combine Metric Server data with Custom Metrics, such that HPA scales up the Deployment.

## WE'LL COVER THE FOLLOWING

- ^
- Combining Metrics Server data with Custom Metrics
  - HPA scaled up the Deployment

So far, the few HPA examples used a single custom metric to decide whether to scale the Deployment. You already know from the Autoscaling Deployments and StatefulSets Based On Resource Usage chapter that we can combine multiple metrics in an HPA. However, all the examples in that chapter used data from the Metrics Server. We learned that in many cases memory and CPU metrics from the Metrics Server are not enough, so we introduced the Prometheus Adapter that feeds custom metrics to the Metrics Aggregator. We successfully configured an HPA to use those custom metrics. Still, more often than not, we'll need a combination of both types of metrics in our HPA definitions. While memory and CPU metrics are not enough by themselves, they are still essential. Can we combine both?

## Combining Metrics Server data with Custom Metrics #

Let's take a look at yet another HPA definition.

```
cat mon/go-demo-5-hpa.yml
```

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

```
...
metrics:
   - type: Resource
```

```
resource:
    name: cpu

    targetAverageUtilization: 80
- type: Resource
    resource:
    name: memory
    targetAverageUtilization: 80
- type: Object
    object:
     metricName: http_req_per_second_per_replica
     target:
        kind: Service
        name: go-demo-5
        targetValue: 1500m
```

This time, HPA has three entries in the metrics section. The first two are the "standard" cpu and memory entries based on the Resource type. The last entry is one of the Object types we used earlier. With those combined, we're telling HPA to scale up if any of the three criteria are met. Similarly, it will scale down as well but for that to happen all three criteria need to be below the targets.

Let's apply the definition.

```
kubectl -n go-demo-5 \
apply -f mon/go-demo-5-hpa.yml
```

Next, we'll describe the HPA. But, before we do that, we'll have to wait for a bit until the updated HPA goes through its next iteration.

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

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

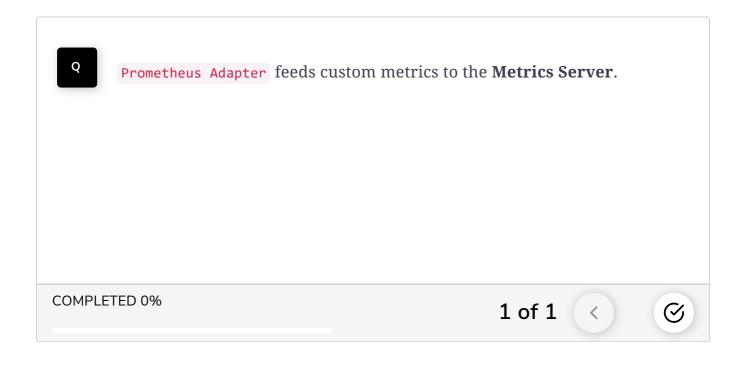
```
sired
...

Events:
... Message
... -----
... New size: 6; reason: Ingress metric http_req_per_second_per_replica ab ove target
... New size: 9; reason: Ingress metric http_req_per_second_per_replica ab ove target
... New size: 4; reason: Service metric http_req_per_second_per_replica ab ove target
... New size: 3; reason: All metrics below target
... New size: 3; reason: memory resource utilization (percentage of reques t) above target
```

## HPA scaled up the Deployment #

We can see that the memory-based metric is above the threshold from the start. In my case, it is 110%, while the target is 80%. As a result, HPA scaled up the Deployment. In my case, it set the new size to 5 replicas.

There's no need to confirm that the new Pods are running. By now, we should trust HPA to do the right thing.



In the next lesson, we will see the complete HPA flow of events.