Getting Practical with Container Memory and CPU Resources

In this lesson, we will explore the usage of resources with the help of some practical examples.

WE'LL COVER THE FOLLOWING

In the previous lesson, we discussed the theory of `resources`. Let's get practical by looking into some practical examples related to `resources`.

Creating the Resources

We'll move on and create the resources defined in the `go-demo-2-random.yml` file.

```
kubectl create \
    -f res/go-demo-2-random.yml \
    --record --save-config

kubectl rollout status \
    deployment go-demo-2-api
```

We created the resources and waited until the `go-demo-2-api` Deployment was rolled out. The **output** of the later command should be as follows.

```
deployment "go-demo-2-api" successfully rolled out
```

Let's describe the `go-demo-2-api` Deployment and see its `limits` and `requests`.

```
kubectl describe deploy go-demo-2-api
```

The **output**, limited to the limits and the requests, is as follows.

```
...
Pod Template:
  ...
  Containers:
    ...
   Limits:
      cpu:    200m
      memory: 100Mi
    Requests:
      cpu:    100m
      memory: 50Mi
...
```

We can see that the limits and the requests correspond to those we defined in the `go-demo-2-random.yml` file. That should come as no surprise.

Looking into the Nodes' Description

Let's describe the nodes that form the cluster (even though there's only one).

```
kubectl describe nodes
```

The **output**, limited to the resource-related entries, is as follows.

```
...
Capacity:
 cpu:     2
 memory:  2048052Ki
 pods:    110
...
Non-terminated Pods: (12 in total)
  Namespace          Name                          CPU Requests CPU Limits Memory Requests Mem
  ---------          ----                          ------------ ---------- --------------- ---
  default            go-demo-2-api-...             100m (5%)    200m (10%) 50Mi (2%)       100
  default            go-demo-2-api-...             100m (5%)    200m (10%) 50Mi (2%)       100
  default            go-demo-2-api-...             100m (5%)    200m (10%) 50Mi (2%)       100
  default            go-demo-2-db-...              300m (15%)   500m (25%) 100Mi (5%)      200
  kube-system        default-http-...              10m (0%)     10m (0%)   20Mi (1%)       20M
  kube-system        metrics-server-...            0 (0%)       0 (0%)     0 (0%)          0 (
  kube-system        kube-addon-manager-minikube   5m (0%)      0 (0%)     50Mi (2%)       0 (
  kube-system        kube-dns-54cccfbdf8-...       260m (13%)   0 (0%)     110Mi (5%)      170
  kube-system        kubernetes-dashboard-...      0 (0%)       0 (0%)     0 (0%)          0 (
  kube-system        nginx-ingress-controller-... 0 (0%)       0 (0%)     0 (0%)          0 (
  kube-system        storage-provisioner           0 (0%)       0 (0%)     0 (0%)          0 (
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests CPU Limits  Memory Requests Memory Limits
  ------------ ----------  --------------- -------------
  875m (43%)   1110m (55%) 430Mi (22%)     690Mi (36%)
...
```

Line 2-7: The `Capacity` represents the overall capacity of a node. In our case, the `minikube` node has 2 CPUs, 2GB of RAM, and can run up to one hundred and ten Pods. Those are the upper limits imposed by the hardware or, in our case, the size of the VM created by Minikube.

Line 10-29: Further down is the `Non-terminated Pods` section. It lists all the Pods with the CPU and memory limits and requests. We can, for example, see that the `go-demo-2-db` Pod has the memory limit set to `100Mi`, which is `5%` of the capacity. Similarly, we can see that not all Pods have specified resources. For example, the `metrics-server-snq2f` Pod has all the values set to `0`. Kubernetes will not be able to handle those Pods appropriately. However, since this is a demo cluster, we'll give the Minikube authors a pass and ignore the lack of resource specification.

Line 31-36: The `Allocated resources` section provides summed values from all the Pods. We can, for example, see that the CPU limits are `56%`. Limits can be even higher than `100%`, and that would not necessarily be a thing to worry about. Not all the containers will have memory and CPU bursts over the requested values. Even if that happens, Kubernetes will know what to do.

What truly matters is that the total amount of requested memory and CPU is within the limits of the capacity. That, however, leads us to an interesting question. What is the basis for the resources we defined so far?

In the next lesson, we will measure actual memory and CPU consumption.