Allocating Excessive Resource than the Actual Usage

In this lesson, we will explore what happens when we allocate excessive resource than the actual usage of an application.

WE'LL COVER THE FOLLOWING

Allocating Excessive Memory

Let's explore another possible situation through yet another updated definition.

```
cat res/go-demo-2-insuf-node.yml
```

Just as before, the change is only in the `resources` of the `go-demo-2-db` Deployment. The **output**, limited to the relevant parts, is as follows.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: go-demo-2-db
spec:
  ...
  template:
    ...
    spec:
      containers:
      - name: db
        image: mongo:3.3
        resources:
          limits:
            memory: 8Gi
            cpu: 0.5
          requests:
            memory: 4Gi
            cpu: 0.3
```

This time, we specified that the requested memory is twice as much as the total memory of the node (2GB). The memory limit is even higher.

Applying the Definition

Let’s apply the change and observe what happens.

```
kubectl apply \
    -f res/go-demo-2-insuf-node.yml \
    --record

kubectl get pods
```

The **output** of the latter command is as follows.

```
NAME              READY STATUS  RESTARTS AGE
go-demo-2-api-... 1/1   Running 0        8m
go-demo-2-api-... 1/1   Running 0        8m
go-demo-2-api-... 1/1   Running 0        9m
go-demo-2-db-...  0/1   Pending 0        13s
```

This time, the status of the Pod is `Pending`. Kubernetes could not place it anywhere in the cluster and is waiting until the situation changes.

Even though memory requests are associated with containers, it often makes sense to translate them into Pod requirements. We can say that the requested memory of a Pod is the sum of the requests of all the containers that form it. In our case, the Pod has only one container, so the requested memory for the Pod and the container are equal. The same can be said for limits.

During the scheduling process, Kubernetes sums the requests of a Pod and looks for a node that has enough available memory and CPU. If Pod’s request cannot be satisfied, it is placed in the pending state in the hope that resources will be freed on one of the nodes, or that a new server will be added to the cluster. Since such a thing will not happen in our case, the Pod created through the `go-demo-2-db` Deployment will be pending forever, unless we change the memory request again.

ℹ When Kubernetes cannot find enough free resources to satisfy the resource requests of all the containers that form a Pod, it changes its

state to `Pending`. Such Pods will remain in this state until requested resources become available.

Looking into the Deployment's Description

Let's describe the `go-demo-2-db` Deployment and see whether there is some additional useful information in it.

```
kubectl describe pod go-demo-2-db
```

The **output**, limited to the events section, is as follows.

```
...
Events:
  Type    Reason           Age    From              Message
  ----    ------           ----   ----              -------
  Warning FailedScheduling 11s    default-scheduler 0/1 nodes are available: 1 Insufficient
```

We can see that it has already `FailedScheduling` seven times and that the message clearly indicates that there is `Insufficient memory`.

Re-Applying the Initial Definition

We'll revert to the initial definition. Even though we know that its resources are incorrect, we know that it satisfies all the requirements and that all the Pods will be scheduled successfully.

```
kubectl apply \
    -f res/go-demo-2-random.yml \
    --record

kubectl rollout status \
    deployment go-demo-2-db

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

Now that all the Pods are running, we should try to write a better definition. For that, we need to observe memory and CPU usage and use that information to decide the requests and the limits.

In the next lesson, we will explore how to adjust resources based on their actual usage.