Examining QoS in Action

In this lesson, we will get practical and see QoS in action.

WE'LL COVER THE FOLLOWING

Checking QoS with Initial Definition

Let's take a look which QoS our `go-demo-2-db` Pod got assigned.

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

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

```
...
Containers:
  db:
    ...
    Limits:
      cpu:    100m
      memory: 100Mi
    Requests:
      cpu:    10m
      memory: 50Mi
...
QoS Class:       Burstable
...
```

The Pod was assigned Burstable QoS.

> Its limits are different from requests, so it did not qualify for Guaranteed QoS.

Since its resources are set, and it is not eligible for Guaranteed QoS, Kubernetes assigned it the second best QoS.

Checking QoS with Modified Definition

Now, let's take a look at a slightly modified definition.

```
cat res/go-demo-2-qos.yml
```

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: "50Mi"
            cpu: 0.1
          requests:
            memory: "50Mi"
            cpu: 0.1
...
apiVersion: apps/v1
kind: Deployment
metadata:
  name: go-demo-2-api
spec:
  ...
  template:
    ...
    spec:
      containers:
      - name: api
        image: vfarcic/go-demo-2
...
```

This time, we specified that both `cpu` and `memory` should have the same values for both the `requests` and the `limits` for the containers that will be created with the `go-demo-2-db` Deployment.

> As a result, `db` Deployment should be assigned Guaranteed QoS.

The containers of the `go-demo-2-api` Deployment are void of any `resources` definitions.

> `api` Deployment should be be assigned BestEffort QoS.

Applying the Definition

Let's confirm that both assumptions (not to say guesses) are indeed correct.

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

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

We applied the new definition and output the rollout status of the `go-demo-2-db` Deployment.

Verification of `db` Deployment

Now we can describe the Pod created thought the `go-demo-2-db` Deployment and check its QoS.

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

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

```
Containers:
  db:
    ...
    Limits:
```

```
      cpu:    100m
      memory: 50Mi
    Requests:
      cpu:    100m
      memory: 50Mi
...
QoS Class: Guaranteed
...
```

Memory and CPU limits and requests are the same and, as a result, the QoS is Guaranteed.

Verification of `api` Deployment

Let's check the QoS of the Pods created through the `go-demo-2-api` Deployment.

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

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

```
...
QoS Class:       BestEffort
...
QoS Class:       BestEffort
...
QoS Class:       BestEffort
...
```

The three Pods created through the `go-demo-2-api` Deployment are without any resources definitions and, therefore, their QoS is set to BestEffort.

Destroying the Objects

We won't be needing the objects we created so far so we'll remove them before moving onto the next subject.

```
kubectl delete \
    -f res/go-demo-2-qos.yml
```

In the next lesson, we will explore defining resource defaults and limitations within a Namespace.