Defining Resource Quotas for a Namespace

In this lesson, we will find out why Resource Quotas are used and how to define them.

WE'LL COVER THE FOLLOWING

⚠ A Problem Still Exists

Resource defaults and limitations are a good first step towards preventing malicious or accidental deployment of Pods that can potentially produce adverse effects on the cluster. Still, any user with the permissions to create Pods in a Namespace can overload the system. Even if `max` values are set to some reasonably small amount of memory and CPU, a user could deploy thousands, or even millions of Pods, and "eat" all the available cluster resources. Such an effect might not be even produced out of malice but accidentally.

A Pod might be attached to a system that scales it automatically without defining upper bounds and, before we know it, it might scale to too many replicas. There are also many other ways things might get out of control.

The Solution

What we need is to define Namespace boundaries through quotas.

With quotas, we can guarantee that each Namespace gets its fair share of resources. Unlike `LimitRange` rules that are applied to each container, `ResourceQuota` defines Namespace limits based on aggregate resource consumption.

We can use `ResourceQuota` objects to define the total amount of compute resources (memory and CPU) that can be spent in a Namespace. We can also use it to limit storage utilization or the number of objects of a certain type that can be created in a Namespace.

Devising a Plan

Let's take a look at the cluster resources we have in our Minikube cluster. It is small, and it's not even a real cluster. However, it's the only one we have (for now), so please use your imagination and pretend that it's "real".

Our cluster has 2 CPUs and 2 GB of memory. Now, let's say that this cluster serves only development and production purposes. We can use the `default` Namespace for production and create a `dev` Namespace for development. We can assume that the production should consume all the resources of the cluster minus those given to the `dev` Namespace which, on the other hand, should not exceed a specific limit.

The truth is that with 2 CPUs and 2 GB of memory, there isn't much we can give to developers. Still, we'll try to be generous. We'll give them 500 MB and 0.8 CPUs for requests. We'll allow occasional bursts in resource usage by defining limits of 1 CPU and 1 GB of memory. Furthermore, we might want to limit the number of Pods to ten. Finally, as a way to reduce risks, we will deny developers the right to expose node ports.

Isn't that a decent plan? I'll imagine that, at this moment, you are nodding as a sign of approval so we'll move on and create the quotas we discussed.

Defining the Quotas

Let's take a look at the `dev.yaml` definition.

```
cat res/dev.yml
```

The **output** is as follows.

```
apiVersion: v1
kind: Namespace
```

```
metadata:
  name: dev


---

apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev
  namespace: dev
spec:
  hard:
    requests.cpu: 0.8
    requests.memory: 500Mi
    limits.cpu: 1
    limits.memory: 1Gi
    pods: 10
    services.nodeports: "0"
```

Besides creating the `dev` Namespace, we're also creating a `ResourceQuota`. It specifies a set of `hard` limits. Remember, they are based on aggregated data, and not on per-container basis like LimitRanges.

We set requests quotas to `0.8` CPUs and `500Mi` of RAM. Similarly, limit quotas is set to `1` CPU and `1Gi` of memory. Finally, we specified that the `dev` Namespace can have only `10` Pods and that there can be no NodePorts. That's the plan we formulated and defined.

In the next lesson, we will get practical and violate some Quotas to analyze the effect.