Defining Resource Defaults and Limitations within a Namespace

In this lesson, we will find out why resource defaults and limitations are needed and how to define these.

WE'LL COVER THE FOLLOWING

- •
- Why we Need Resource Defaults and Limitations?
- Defining Default Requests and Limits
 - Creating a Namespace
 - Looking into the Definition
 - default and defaultRequest
 - min and max
 - Creating the Resource
 - Looking into the Description
 - Looking into Updated Definition
 - Creating Resources
 - Looking into the Description

Why we Need Resource Defaults and Limitations?

We already learned how to leverage Kubernetes Namespaces to create clusters within a cluster. When combined with RBAC, we can create Namespaces and give users permissions to use them without exposing the whole cluster. Still, one thing is missing.

We can, let's say, create a test Namespace and allow users to create objects without permitting them to access other Namespaces. Even though that is better than allowing everyone full access to the cluster, such a strategy would not prevent people from bringing the whole cluster down or affecting the

performance of applications running in other Namespaces. The piece of the puzzle we're missing is resource control on the Namespace level.

We already discussed that every container should have resource limits and requests defined. That information helps Kubernetes schedule Pods more efficiently. It also provides it with the information it can use to decide whether a Pod should be evicted or restarted.

Still, the fact that we can specify resources does not mean that we are forced to define them. We should have the ability to set default resources that will be applied when we forget to specify them explicitly.

Even if we define default resources, we also need a way to set limits.

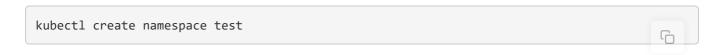
Otherwise, everyone with permissions to deploy a Pod can potentially run an application that requests more resources than we're willing to give.

Defining Default Requests and Limits

Our next task is to define default requests and limits as well as to specify minimum and maximum values someone can define for a Pod.

Creating a Namespace

We'll start by creating a test Namespace.



Looking into the Definition

With a playground Namespace created, we can take a look at a new definition.

```
cat res/limit-range.yml
```

The **output** is as follows.

```
apiVersion: v1
kind: LimitRange
metadata:
   name: limit-range
spec:
   limits:
   - default:
       memory: 50Mi
       cpu: 0.2
```

```
defaultRequest:
   memory: 30Mi
   cpu: 0.05

max:
   memory: 80Mi
   cpu: 0.5

min:
   memory: 10Mi
   cpu: 0.01
   type: Container
```

We specified that the resource should be of LimitRange kind. It's spec has four limits.

```
default and defaultRequest #
```

The default limit and defaultRequest entries will be applied to the containers that do not specify resources. If a container does not have memory or CPU limits, it'll be assigned the values set in the LimitRange. The default entries are used as limits, and the defaultRequest entries are used as requests.

```
min and max #
```

When a container does have the resources defined, they will be evaluated against LimitRange thresholds specified as max and min. If a container does not meet the criteria, the Pod that hosts the containers will not be created.

Creating the Resource

We'll see a practical implementation of the four limits soon. For now, the
next step is to create the limit-range resource.

```
kubectl --namespace test create \
   -f res/limit-range.yml \
   --save-config --record
```

We created the LimitRange resource.

Looking into the Description

Let's describe the test Namespace where the resource was created.

```
kubectl describe namespace test
```

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

```
Resource Limits
Type Resource Min Max Default Request Default Limit Max Limit/Request Ratio
---- Container cpu 10m 500m 50m 200m -
Container memory 10Mi 80Mi 30Mi 50Mi -
```

We can see that the **test** Namespace has the resource limits we specified. We set four out of five possible values.

The maxLimitRequestRatio is missing and we'll describe it only briefly. When MaxLimitRequestRatio is set, container request and limit resources must both be non-zero, and the limit divided by the request must be less than or equal to the enumerated value.

Looking into Updated Definition

Let's take a look at yet another variation of the go-demo definition.

```
cat res/go-demo-2-no-res.yml
```

The only thing to note is that none of the containers have any resources defined.

Creating Resources

Next, we'll create the objects defined in the go-demo-2-no-res.yml file.

```
kubectl --namespace test create \
    -f res/go-demo-2-no-res.yml \
    --save-config --record

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

We created the objects inside the test Namespace and waited until the deployment "go-demo-2-api" was successfully rolled out.

Looking into the Description

Let's describe one of the Pods we created.

```
kubectl --namespace test describe \
pod go-demo-2-db
```

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

```
Containers:

db:

...

Limits:

cpu: 200m

memory: 50Mi

Requests:

cpu: 50m

memory: 30Mi

...
```

Even though we did not specify the resources of the db container inside the go-demo-2-db Pod, the resources are set. The db container was assigned the default limits of the test Namespace as the container limit. Similarly, the defaultRequest limits were used as container requests.

As we can see, any attempt to create Pods hosting containers without resources will result in the Namespace limits applied.

We should still define container resources instead of relying on Namespace default limits. They are, after all, only a fallback in case someone forgot to define resources.

In the next lesson, we will explore what happens when the resources are defined but they exceed the Namespace's set limits.