Measuring the Actual Memory and CPU Consumption

In this lesson, we will figure out how to measure the actual memory and CPU consumption.

WE'LL COVER THE FOLLOWING

Exploring the Options

How did we come up with the current memory and CPU values? Why did we set the memory of the MongoDB to `100Mi`? Why not `50Mi` or `1Gi`? It is embarrassing to admit that the values we have right now are random. We guessed that the containers based on the `vfarcic/go-demo-2` image require less resources than Mongo database, so their values are comparatively smaller. That was the only criteria we used to define the resources.

Before you frown upon the decision to put random values for resources, you should know that we do not have any metrics to back us up. Anybody's guess is as good as ours.

The only way to truly know how much memory and CPU an application uses is by retrieving metrics. We'll use Metrics Server for that purpose.

Metrics Server collects and interprets various signals like compute resource usage, lifecycle events, etc. In our case, we're interested only in CPU and memory consumption of the containers we're running in our cluster.

When we created the cluster, we enabled the `metrics-server` addon and Minikube deployed it as a system application.

You might be inclined to think that Metrics Server might be the solution for your monitoring needs. I advise against such a decision. We're using Metrics Server only because it's readily available as a Minikube addon.

The idea to develop a Metrics Server as a tool for monitoring needs is mostly abandoned. Its primary focus is to serve as an internal tool required for some of the Kubernetes features.

Instead, I'd suggest a combination of Prometheus combined with the Kubernetes API as the source of metrics and Alertmanager for your alerting needs. However, those tools are not in the scope of this chapter, so you might need to educate yourself from their documentation, or wait until the sequel to this book is published (the tentative name is *Advanced Kubernetes*).

> ℹ Use Metrics Server only as a quick-and-dirty way to retrieve metrics. Explore the combination of Prometheus and Alertmanager for your monitoring and alerting needs.

Now that we clarified what Metrics Server is good for, as well as what it isn't, we can proceed and confirm that it is indeed running inside our cluster.

```
kubectl --namespace kube-system \
    get pods
```

The **output** is as follows.

```
NAME                              READY STATUS  RESTARTS AGE
coredns-...                       1/1   Running 0        3m49s
coredns-...                       1/1   Running 0        3m49s
etcd-minikube                     1/1   Running 0        3m56s
kube-addon-manager-minikube       1/1   Running 0        3m56s
kube-apiserver-minikube           1/1   Running 0        3m56s
kube-controller-manager-minikube 1/1   Running 0        3m56s
kube-proxy-...                    1/1   Running 0        3m49s
kube-scheduler-minikube           1/1   Running 0        3m56s
metrics-server-...                1/1   Running 0        3m48s
nginx-ingress-controller-...      1/1   Running 0        3m47s
storage-provisioner               1/1   Running 0        3m47s
```

As you can see, `metrics-server` is running.

Let's try a very simple query of Metrics Server.

```
kubectl top pods
```

The **output** is as follows.

The **output** is as follows.

```
NAME              CPU(cores) MEMORY(bytes)
go-demo-2-api-... 0m         2Mi
go-demo-2-api-... 0m         2Mi
go-demo-2-api-... 0m         2Mi
go-demo-2-db-...  4m         35Mi
```

We retrieved all the Pods in the `default` Namespace. As you can see, most the available metrics are related to memory and CPU.

We can see that memory usage of the DB Pod is somewhere around 35 megabytes. That's quite a big difference from `100Mi` we set. Sure, this service is not under real production load but, since we're simulating a "real" cluster, we'll pretend that `35Mi` is indeed memory usage under "real" conditions. That means that we overestimated the requests by assigning a value almost three times larger than the actual usage.

How about the CPU? Did we make such a colossal mistake with it as well? As a reminder, we set the CPU request to `0.3` and the limit to `0.5`. However, based on the previous output, the CPU usage is around `5m` or `0.005` CPU. We, again, made a huge mistake with resource specification. Our value is around sixty times higher.

Such deviations between our expectations (resource requests and limits) and the actual usage can lead to very unbalanced scheduling with undesirable effects. We'll correct the resources soon. For now, we'll explore what happens if the amount of resources is below the actual usage.

In the next lesson, we will explore allocating insufficient resources than the actual usage of the containers.