# Create HPA with Custom Metrics pulled through Exporters

In this lesson, we will create `HPA` with custom metrics which will be pulled by HPA through exporters.

WE'LL COVER THE FOLLOWING



As you already saw, `Prometheus Adapter` comes with a set of default rules that provide many metrics that we do not need, and not all those that we do. It's wasting CPU and memory by doing too much, but not enough. We'll explore how we can customize the adapter with our own rules. Our next goal is to make the adapter retrieve only the `nginx_ingress_controller_requests` metric since that's the only one we need. On top of that, it should provide that metric in two forms. First, it should retrieve the rate, grouped by the resource. The second form should be the same as the first but divided with the number of replicas of the Deployment that hosts the Pods where Ingress forwards the resources. That one should give us an average number of requests per replica and will be a good candidate for our first `HPA` definition based on custom metrics.

## Make adapter retrieve

`nginx_ingress_controller_requests`

## metric #

I already prepared a file with Chart values that might accomplish our current objectives, so let's take a look at it.

```
cat mon/prom-adapter-values-ing.yml
```

The **output** is as follows.

```
image:
  tag: v0.5.0
metricsRelistInterval: 90s
prometheus:
  url: http://prometheus-server.metrics.svc
  port: 80
rules:
  default: false
  custom:
  - seriesQuery: 'nginx_ingress_controller_requests'
    resources:
      overrides:
        namespace: {resource: "namespace"}
        ingress: {resource: "ingress"}
    name:
      as: "http_req_per_second"
    metricsQuery: 'sum(rate(<<.Series>>{<<.LabelMatchers>>}[5m])) by (<<.G
roupBy>>)'
  - seriesQuery: 'nginx_ingress_controller_requests'
    resources:
      overrides:
        namespace: {resource: "namespace"}
        ingress: {resource: "ingress"}
    name:
      as: "http_req_per_second_per_replica"
    metricsQuery: 'sum(rate(<<.Series>>{<<.LabelMatchers>>}[5m])) by (<<.G
roupBy>>) / sum(label_join(kube_deployment_status_replicas, "ingres
s", ",", "deployment")) by (<<.GroupBy>>)'
```

The first few entries in that definition are the same values as the ones we used previously through `--set` arguments. We'll skip those, and jump into the `rules` section.

Within the `rules` section, we're setting the `default` entry to `false`. That will get rid of the default rules we explored previously and allows us to start with a clean slate. Further on, there are two `custom` rules.

## First custom rule #

The first rule is based on the `seriesQuery` with `nginx_ingress_controller_requests` as the value. The `overrides` entry inside the `resources` section helps the adapter find out which Kubernetes resources are associated with the metric. We're setting the value of the `namespace` label to the `namespace` resource. There's a similar entry for `ingress`. In other words, we're associating `Prometheus` labels with Kubernetes resources `namespace` and `ingress`.

As you will see soon, the metric itself will be a part of a full query that will be treated as a single metric by `HPA`. Since we are creating something new, we need a name. So, we specified the `name` section with a single `as` entry set to `http_req_per_second`. That will be the reference in our `HPA` definitions.

You already know that `nginx_ingress_controller_requests` is not very useful by itself. When we used it in `Prometheus`, we had to put it inside a `rate` function, we had to `sum` everything, and we had to group the results by a resource. We're doing something similar through the `metricsQuery` entry. Think of it as an equivalent of expressions we're writing in `Prometheus`. The only difference is that we are using "special" syntax like `<<.Series>>`. That's the adapter's templating mechanism. Instead of hard-coding the name of the metric, the labels, and the group by statements, we have `<<.Series>>`, `<<.LabelMatchers>>`, and `<<.GroupBy>>` clauses that will be populated with the correct values depending on what we put in API calls.

## Second custom rule #

The second rule is almost the same as the first. The difference is in the name (now it's `http_req_per_second_per_replica`) and in the `metricsQuery`. The latter is now dividing the result with the number of replicas of the associated Deployment, just as we practiced in the Collecting And Querying Metrics And Sending Alerts chapter.

Next, we'll update the Chart with the new values.

```
helm upgrade prometheus-adapter \
    stable/prometheus-adapter \
    --version 1.4.0 \
    --namespace metrics \
    --values mon/prom-adapter-values-ing.yml

kubectl -n metrics \
    rollout status \
    deployment prometheus-adapter
```

Now that the Deployment has rolled out successfully, we can double-check that the configuration stored in the ConfigMap is indeed correct.

```
kubectl -n metrics \
    describe cm prometheus-adapter
```

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

```
...
Data
====
config.yaml:
----
rules:
- metricsQuery: sum(rate(<<.Series>>{<<.LabelMatchers>>}[5m])) by (<<.Grou
pBy>>)
  name:
    as: http_req_per_second
  resources:
    overrides:
      ingress:
        resource: ingress
      namespace:
        resource: namespace
  seriesQuery: nginx_ingress_controller_requests
- metricsQuery: sum(rate(<<.Series>>{<<.LabelMatchers>>}[5m])) by (<<.Grou
pBy>>) /
    sum(label_join(kube_deployment_status_replicas, "ingress", ",", "deplo
yment"))
    by (<<.GroupBy>>)
  name:
    as: http_req_per_second_per_replica
```

```
  resources:
    overrides:

      ingress:
        resource: ingress
      namespace:
        resource: namespace
  seriesQuery: nginx_ingress_controller_requests
...
```

We can see that the default `rules` we explored earlier are now replaced with the two rules we defined in the `rules.custom` section of our Chart values file.

The fact that the configuration looks correct does not necessarily mean that the adapter now provides data as Kubernetes custom metrics. Let's check that as well.

```
kubectl get --raw \
    "/apis/custom.metrics.k8s.io/v1beta1" \
    | jq "."
```

The **output** is as follows.

```
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "custom.metrics.k8s.io/v1beta1",
  "resources": [
    {
      "name": "namespaces/http_req_per_second_per_replica",
      "singularName": "",
      "namespaced": false,
      "kind": "MetricValueList",
      "verbs": [
        "get"
      ]
    },
    {
      "name": "ingresses.extensions/http_req_per_second_per_replica",
      "singularName": "",
      "namespaced": true,
      "kind": "MetricValueList",
      "verbs": [
        "get"
      ]
```

```
    },
    {

      "name": "ingresses.extensions/http_req_per_second",
      "singularName": "",
      "namespaced": true,
      "kind": "MetricValueList",
      "verbs": [
        "get"
      ]
    },
    {
      "name": "namespaces/http_req_per_second",
      "singularName": "",
      "namespaced": false,
      "kind": "MetricValueList",
      "verbs": [
        "get"
      ]
    }
  ]
}
```

We can see that there are four metrics available, two of `http_req_per_second` and two of `http_req_per_second_per_replica`. Each of the two metrics we defined are available as both `namespaces` and `ingresses`. Right now, we do not care about `namespaces`, and we'll concentrate on `ingresses`.

## Create `HPA` based on custom metrics #

I'll assume that at least five minutes (or more) passed since we sent a hundred requests. If it didn't, you are a speedy reader, and you'll have to wait for a while before we send another hundred requests. We are about to create our first `HorizontalPodAutoscaler (HPA)` based on custom metrics, and I want to make sure that you see how it behaves both before and after it is activated.

Now, let's take a look at an `HPA` definition.

```
cat mon/go-demo-5-hpa-ing.yml
```

The **output** is as follows.

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
```

```
metadata:
  name: go-demo-5

spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: go-demo-5
  minReplicas: 3
  maxReplicas: 10
  metrics:
  - type: Object
    object:
      metricName: http_req_per_second_per_replica
      target:
        kind: Namespace
        name: go-demo-5
      targetValue: 50m
```

The first half of the definition should be familiar since it does not differ from what we used before. It will maintain between `3` and `10` replicas of the `go-demo-5` Deployment. The new stuff is in the `metrics` section.

In the past, we used `spec.metrics.type` set to `Resource`. Through that type, we defined CPU and memory targets. This time, however, our type is `Object`. It refers to a metric describing a single Kubernetes object which, in our case, happens to be a custom metric coming from `Prometheus Adapter`.

If we go through the ObjectMetricSource v2beta1 autoscaling documentation, we can see that the fields of the `Object` type are different than those we used before when our type was `Resources`. We set the `metricName` to the metric we defined in `Prometheus Adapter` (`http_req_per_second_per_replica`). Remember that it is not a metric, but that we defined an expression that the adapter uses to fetch data from `Prometheus` and convert it into a custom metric. In this case, we are getting the number of requests entering an Ingress resource divided by the number of replicas of a Deployment.

Finally, the `targetValue` is set to `50m` or 0.05 requests per second. I intentionally set it to a very low value so that we can easily reach the target and observe what happens.

Let's `apply` the definition.

```
kubectl -n go-demo-5 \
    apply -f mon/go-demo-5-hpa-ing.yml
```

Next, we'll describe the newly created `HPA`, and see whether we can observe anything interesting.

```
kubectl -n go-demo-5 \
    describe hpa go-demo-5
```

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

```
...
Metrics:          ( current / target )
  "http_req_per_second_per_replica" on Namespace/go-demo-5: 0 / 50m
Min replicas:    3
Max replicas:    10
Deployment pods: 3 current / 3 desired
...
```

## Maintains minimum number of replicas #

We can see that there is only one entry in the `Metrics` section. The `HPA` is using the custom metric `http_req_per_second_per_replica` based on `Namespace/go-demo-5`. At the moment, the current value is `0`, and the `target` is set to `50m` (0.05 requests per second). If, in your case, the `current` value is `unknown`, please wait for a few moments, and re-run the command.

Further down, we can see that both the `current` and the `desired` number of Deployment Pods is set to `3`.

All in all, the target is not reached (there are `0` requests) so there's no need for the `HPA to do anything`. It maintains the minimum number of replicas.

## Generate some traffic #

Let's spice it up a bit by generating some traffic.

```
for i in {1..100}; do
    curl "http://$GD5_ADDR/demo/hello"
done
```

We sent a hundred requests to the `go-demo-5` Ingress.

Let's `describe` the `HPA` again, and see whether there are some changes.

Let's `describe` the `HPA` again, and see whether there are some changes.

```
kubectl -n go-demo-5 \
    describe hpa go-demo-5
```

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

```
...
Metrics:                                                  ( current / tar
get )
  "http_req_per_second_per_replica" on Ingress/go-demo-5:  138m / 50m
Min replicas:                                             3
Max replicas:                                             10
Deployment pods:                                          3 current / 6 d
esired
...
Events:
  ... Message
  ... -------
  ... New size: 6; reason: Ingress metric http_req_per_second_per_replic
a above target
```

We can see that the `current` value of the metric increased. In my case, it is `138m` (0.138 requests per second). If your output still shows `0`, you'll have to wait until the metrics are pulled by `Prometheus` until the adapter fetches them, and until the `HPA` refreshes its status. In other words, wait for a few moments, and re-run the previous command.

Given that the `current` value is higher than the `target`, in my case, the `HPA` changed the `desired` number of `Deployment pods` to `6` (your number might differ depending on the value of the metric). As a result, `HPA` modified the Deployment by changing its number of replicas, and we should see additional Pods running. That becomes more evident through the `Events` section. There should be a new message stating `New size: 6; reason: Ingress metric http_req_per_second_per_replica above target`.

To be on the safe side, we'll list the Pods in the `go-demo-5` Namespace and confirm that the new ones are indeed running.

```
kubectl -n go-demo-5 get pods
```

The **output** is as follows.

```
NAME           READY STATUS  RESTARTS AGE
go-demo-5-db-0 2/2   Running 0        19m
go-demo-5-db-1 2/2   Running 0        19m
go-demo-5-db-2 2/2   Running 0        10m
go-demo-5-...  1/1   Running 2        19m
go-demo-5-...  1/1   Running 0        16s
go-demo-5-...  1/1   Running 2        19m
go-demo-5-...  1/1   Running 0        16s
go-demo-5-...  1/1   Running 2        19m
go-demo-5-...  1/1   Running 0        16s
```

We can see that there are now six `go-demo-5-*` Pods, with three of them much younger than the rest.

## Drop the traffic #

We'll explore what happens when traffic drops below the `HPAs` `target`. We'll accomplish that by not doing anything for a while. Since we are the only ones sending requests to the application, all we have to do is to stand still for five minutes or, even better, use this time to fetch coffee.

The reason we need to wait for at least five minutes lies in the frequencies `HPA` uses to scale up and down. By default, an HPA will scale up every three minutes, as long as the `current` value is above the `target`. Scaling down requires five minutes. An `HPA` will descale only if the `current` value is below the target for at least three minutes since the last time it scaled up.

All in all, we need to wait for five minutes, or more, before we see the scaling effect in the opposite direction.

```
kubectl -n go-demo-5 \
    describe hpa go-demo-5
```

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

```
...
Metrics:                ( current / target )
  "http_req_per_second_per_replica" on Ingress/go-demo-5:  0 / 50m
Min replicas:           3
Max replicas:           10
```

```
Deployment pods: 3 current / 3 desired
...

Events:
... Age   ... Message
... ----  ... -------
... 10m   ... New size: 6; reason: Ingress metric http_req_per_second_per_
replica above target
... 7m10s ... New size: 9; reason: Ingress metric http_req_per_second_per_
replica above target
... 2m9s  ... New size: 3; reason: All metrics below target
```

The most interesting part of the output is the events section. We'll focus on the `Age` and `Message` fields. Remember, scale-up events are executed every three minutes if the current value is above the target, while scale-down iterations are every five minutes.

## `HPA` scaled the Deployment up #

In my case, the `HPA` scaled the Deployment again, three minutes later. The number of replicas jumped from six to nine. Since the expression used by the adapter uses a five minutes rate, some of the requests entered the second `HPA` iteration. Scaling up even after we stopped sending requests, might not seem like a good idea (it isn't), but in the "real world" scenarios, that shouldn't occur, since there's much more traffic than what we generated. We wouldn't put `50m` (0.2 requests per second) as a target.

## `HPA` scaled the Deployment down #

Five minutes after the last scale-up event, the `current` value was at `0`, and the `HPA` scaled the Deployment down to the minimum number of replicas (`3`). There's no traffic anymore, and we're back to where we started.

Q

`Object` refers to a metric describing a single Kubernetes object which, in our case, happens to be a custom metric coming from `Prometheus Adapter`.

In the next lesson, we will see that Instrumented Metrics can be used to create `HPA` with custom metrics.