Create HPA with Custom Metrics pulled through Instrumented metric

In this lesson, we will confirm that HPA can also pull metrics using Instrumented Metrics.

WE'LL COVER THE FOLLOWING

- Pull metrics through Instrumented metrics
  - Combining both rules
  - `resources` sections
  - Two different custom metrics
  - Service-based metric used by updated `HPA`
  - Test autoscaling based on custom metrics

We confirmed that `Prometheus'` metrics, fetched by `Prometheus Adapter` and converted into Kubernetes' custom metrics, can be used in `HPA`. So far, we used metrics pulled by `Prometheus` through exporters (`nginx_ingress_controller_requests`). Given that the adapter fetches metrics from `Prometheus`, it shouldn't matter how they got there. Nevertheless, we'll confirm that instrumented metrics can be used as well. That will give us an opportunity to cement what we learned so far and, at the same time, maybe learn a few new tricks.

Pull metrics through Instrumented metrics

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

The **output** is yet another set of `Prometheus Adapter Chart` values.

```
image:
  tag: v0.5.0
```

```
metricsRelistInterval: 90s
prometheus:

  url: http://prometheus-server.metrics.svc
  port: 80
rules:
  default: false
  custom:
  - seriesQuery: 'http_server_resp_time_count{kubernetes_namespace!="",kub
ernetes_name!=""}'
    resources:
      overrides:
        kubernetes_namespace: {resource: "namespace"}
        kubernetes_name: {resource: "service"}
    name:
      matches: "^(.*)server_resp_time_count"
      as: "${1}req_per_second_per_replica"
    metricsQuery: 'sum(rate(<<.Series>>{<<.LabelMatchers>>}[5m])) by (<<.G
roupBy>>) / count(<<.Series>>{<<.LabelMatchers>>}) by (<<.GroupBy>>)'
  - 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>>)'
```

Combining both rules

This time, we're combining rules containing different metric series. The first rule is based on the `http_server_resp_time_count` instrumented metric that originates in `go-demo-5`. We used it in the Debugging Issues Discovered Through Metrics And Alerts chapter and there's nothing truly extraordinary in its definition. It follows the same logic as the rules we used before. The second rule is a copy of one of the rules we used before.

What is interesting about those rules is that there are two completely different queries producing different results. However, the name is the same (`http_req_per_second_per_replica`) in both cases.

"Wait a minute", you might say. The names are not the same. One is called `${1}req_per_second_per_replica` while the other is

`${1}req_per_second_per_replica` while the other is `http_req_per_second_per_replica`. While that is true, the final name, excluding resource type, is indeed the same. I wanted to show you that you can use regular expressions to form a name. In the first rule, the name consists of `matches` and `as` entries. The `(.*)` part of the `matches` entry becomes the first variable (there can be others) which is later on used as part of the `as` value (`${1}`). Since the metric is `http_server_resp_time_count`, it will extract `http_` from `^(.*)server_resp_time_count` which, in the next line, is used instead of `${1}`. The final result is `http_req_per_second_per_replica`, which is the same as the name of the second rule.

`resources` sections

Now that we established that both rules will provide custom metrics with the same name, we might think that will result in a conflict. How will `HPA` know which metric to use, if both are called the same? Will the adapter have to discard one and keep the other? The answer lies in the `resources` sections.

A true identifier of a metric is a combination of its name and the resource it ties with. The first rule produces two custom metrics, one for Services and the other for Namespaces. The second also generates custom metrics for Namespaces, but for Ingresses as well. How many metrics is that in total? I'll let you think about the answer before we check the result. To do that, we'll have to `upgrade` the Chart for the new values to take effect.

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

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

We upgraded the Chart with the new values and waited until the Deployment rolls out.

Now we can go back to our pending question "how many custom metrics we've got?" Let's see...

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

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

```
{
  ...
    {
      "name": "services/http_req_per_second_per_replica",
      ...
    },
    {
      "name": "namespaces/http_req_per_second_per_replica",
      ...
    },
    {
      "name": "ingresses.extensions/http_req_per_second_per_replica",
      ...
```

Now we have **three** custom metrics, not four. I already explained that the unique identifier is the name of the metric combined with the Kubernetes resource it's tied to. All the metrics are called `http_req_per_second_per_replica`. But, since both rules override two resources, and `namespace` is set in both, one had to be discarded. We don't know which one was removed and which stayed. Or maybe, they were merged. It does not matter since we shouldn't override the same resource with the metrics with the same name. There was no practical reason for me to include `namespace` in adapter's rule other than to show you that there can be multiple overrides and what happens when they are the same. Other than that silly reason, you can mentally ignore the `namespaces/http_req_per_second_per_replica` metric.

Two different custom metrics

We used two different `Prometheus` expressions to create two different custom metrics, with the same name but related to other resources. One (based on `nginx_ingress_controller_requests` expression) comes from Ingress resources, while the other (based on `http_server_resp_time_count`) comes from Services. Even though the latter originates in `go-demo-5` Pods, `Prometheus` discovered it through Services (as discussed in the previous chapter).

We can use the `/apis/custom.metrics.k8s.io` endpoint not only to discover which custom metrics we have but also to inspect details, including values. For example, we can retrieve `services/http_req_per_second_per_replica` metric through the command that follows.

```
kubectl get --raw \
    "/apis/custom.metrics.k8s.io/v1beta1/namespaces/go-demo-5/services/*/h
ttp_req_per_second_per_replica" \
    | jq .
```

The **output** is as follows.

```
{
  "kind": "MetricValueList",
  "apiVersion": "custom.metrics.k8s.io/v1beta1",
  "metadata": {
    "selfLink": "/apis/custom.metrics.k8s.io/v1beta1/namespaces/go-demo-5/
services/%2A/http_req_per_second_per_replica"
  },
  "items": [
    {
      "describedObject": {
        "kind": "Service",
        "namespace": "go-demo-5",
        "name": "go-demo-5",
        "apiVersion": "/v1"
      },
      "metricName": "http_req_per_second_per_replica",
      "timestamp": "2018-10-27T23:49:58Z",
      "value": "1130m"
    }
  ]
}
```

The `describedObject` section shows us the details of the items. Right now, we have only one Service with that metric. We can see that the Service resides in the `go-demo-5` Namespace, that its name is `go-demo-5`, and that it is using `v1` API version.

Further down, we can see the current value of the metric. In my case, it is `1130m`, or slightly above one request per second. Since nobody is sending requests to the `go-demo-5` Service, that value is as expected, considering that a health check is executed once a second.

Service-based metric used by updated `HPA`

Next, we'll explore the updated `HPA` definition that will use the Service-based metric.

```
cat mon/go-demo-5-hpa-svc.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: Service
        name: go-demo-5
      targetValue: 1500m
```

When compared with the previous definition, the only change is in the `target` and `targetValue` fields. Remember, the full identifier is a combination of the `metricName` and `target`. Therefore, this time we changed the `kind` to `Service`. We also had to change the `targetValue` since our application is receiving not only external requests through Ingress but also internal ones. They could be originating in other applications that might communicate with `go-demo-5` or, as in our case, in Kubernetes' health checks. Since their frequency is one second, we set the `targetValue` to `1500m`, or 1.5 requests per second. That way, scaling will not be triggered if we do not send any requests to the application. Normally, you'd set a much bigger value. But, for now, we're only trying to observe how it behaves before and after scaling.

Next, we'll apply the changes to the `HPA` and describe it.

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

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

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

```
...
Metrics:                                                  ( current / targ
et )
  "http_req_per_second_per_replica" on Service/go-demo-5: 1100m / 1500m
...
Deployment pods:                                           3 current / 3 d
esired
...
Events:
  Type    Reason             Age    From                       Message
  ----    ------             ----   ----                       -------
  Normal  SuccessfulRescale  12m    horizontal-pod-autoscaler  New size: 6
; reason: Ingress metric http_req_per_second_per_replica above target
  Normal  SuccessfulRescale  9m20s  horizontal-pod-autoscaler  New size: 9
; reason: Ingress metric http_req_per_second_per_replica above target
  Normal  SuccessfulRescale  4m20s  horizontal-pod-autoscaler  New size: 3
; reason: All metrics below target
```

For now, there's no reason for the `HPA` to scale up the Deployment. The current value is below the threshold. In my case, it's `1100m`.

Test autoscaling based on custom metrics

Now we can test whether autoscaling based on custom metrics originating from instrumentation works as expected. Sending requests through Ingress might be slow, especially if our cluster runs in Cloud. The round-trip from our laptop all the way to the service might be too slow. So, we'll send requests from inside the cluster, by spinning up a Pod and executing a request loop from inside it.

```
kubectl -n go-demo-5 \
    run -it test \
    --image=debian \
```

```
    --image=debian \
    --restart=Never \

    --rm \
    -- bash
```

Normally, I prefer `alpine` images since they are much small and efficient. However, `for` loops do not work from `alpine`, so we switched to `debian` instead. It doesn't have `curl` though, so we'll have to install it.

```
apt update

apt install -y curl
```

Now we can send requests that will generate enough traffic for `HPA` to trigger the scale-up process.

```
for i in {1..500}; do
    curl "http://go-demo-5:8080/demo/hello"
done

exit
```

We sent five-hundred requests to `/demo/hello` endpoint, and we exited the container. Since we used the `--rm` argument when we created the Pod, it will be removed automatically from the system, so we do not need to execute any cleanup operation.

Let's `describe` the `HPA` and see what happened.

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

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

```
...
Reference:                                                 Deployment/go-de
mo-5
Metrics:                                                   ( current / targ
et )
  "http_req_per_second_per_replica" on Service/go-demo-5: 1794m / 1500m
Min replicas:                                              3
Max replicas:                                              10
```

```
Deployment pods:                                         3 current / 4 de
sired

...
Events:
... Message
... -------
... New size: 6; reason: Ingress metric http_req_per_second_per_replica ab
ove target
... New size: 9; reason: Ingress metric http_req_per_second_per_replica ab
ove target
... New size: 3; reason: All metrics below target
... New size: 4; reason: Service metric http_req_per_second_per_replica ab
ove target
```

`HPA` detected that the `current` value is above the target (in my case it's `1794m`) and changed the desired number of replicas from `3` to `4`. We can observe that from the last event as well. If, in your case, the `desired` number of replicas is still `3`, please wait for a few moments for the next iteration of `HPA` evaluations and repeat the `describe` command.

If we need an additional confirmation that scaling indeed worked as expected, we can retrieve the Pods in the `go-demo-5` Namespace.

```
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        33m
go-demo-5-db-1 2/2   Running 0        32m
go-demo-5-db-2 2/2   Running 0        32m
go-demo-5-...  1/1   Running 2        33m
go-demo-5-...  1/1   Running 0        53s
go-demo-5-...  1/1   Running 2        33m
go-demo-5-...  1/1   Running 2        33m
```

There's probably no need to confirm that the `HPA` will soon scale down the `go-demo-5` Deployment after we stopped sending requests.

Q

A true identifier of a metric is a combination of its name and the resource it ties with.

resources it ties with.

COMPLETED 0%

1 of 1

In the next lesson, we will combine Metrics Server data with Custom Metrics.