

Exploring Prometheus Adapter

In this lesson, we will introduce and explore the Prometheus Adapter and the benefit of using it.

WE'LL COVER THE FOLLOWING

- Prometheus Adapter
 - Prometheus data provided through the adapter
 - Data flow of Metrics Aggregator
 - Defining goals of `go-demo-5` application
 - Install the application
 - Calculate how many requests passed through Ingress
 - Average number of requests per replica
 - Retrieve `http_server_resp_time_count`
 - Setup complications

Given that we want to extend the metrics available through the Metrics API and that Kubernetes allows us to do so through its [Custom Metrics API](#), one option to accomplish our goals could be to create our own adapter. Depending on the application (DB) where we store metrics, that might be a good option. But, given that it is pointless to reinvent the wheel, our first step should be to search for a solution. If someone already created an adapter that suits our needs, it would make sense to adopt it instead of creating a new one by ourselves. Even if we do choose something that provides only part of the features we're looking for, it's easier to build on top of it (and contribute back to the project) than to start from scratch.

Prometheus Adapter

Given that our metrics are stored in `Prometheus`, we need a `metrics adapter` that will be capable of fetching data from it. Since `Prometheus` is very popular and adopted by the community, there is already a project waiting for us to

use. It's called **Kubernetes Custom Metrics Adapter for Prometheus**. It is an implementation of the Kubernetes Custom Metrics API that uses **Prometheus** as the data source.

Since we adopted **Helm** for all our installations, we'll use it to install the adapter.


```
helm install prometheus-adapter \
  stable/prometheus-adapter \
  --version 1.4.0 \
  --namespace metrics \
  --set image.tag=v0.5.0 \
  --set metricsRelistInterval=90s \
  --set prometheus.url=http://prometheus-server.metrics.svc \
  --set prometheus.port=80

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

We installed the **prometheus-adapter** Helm Chart from the **stable** repository. The resources were created in the **metrics** Namespace, and the **image.tag** is set to **v0.3.0**.

We changed **metricsRelistInterval** from the default value of **30s** to **90s**. That is the interval the adapter will use to fetch metrics from **Prometheus**. Since our **Prometheus** setup is fetching metrics from its targets every sixty seconds, we had to set the adapter's interval to a value higher than that. Otherwise, the adapter's frequency would be higher than pulling frequency of **Prometheus**, and we'd have iterations that would be without new data.

The last two arguments specified the URL and the port through which the adapter can access the **Prometheus API**. In our case, the URL is set to go through the **Prometheus**'s Service.

 Please visit [Prometheus Adapter Chart README](#) for more information about all the values you can set to customize the installation.

Finally, we waited until the `prometheus-adapter` rolled out.

Prometheus data provided through the adapter

If everything is working as expected, we should be able to query Kubernetes' Custom Metrics API and retrieve some of the `Prometheus` data provided through the adapter.

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

🔍 Given the promise that each chapter will feature a different Kubernetes' flavor and that AWS did not have its turn yet, all the outputs are taken from EKS. Depending on which platform you're using, your outputs might be slightly different.

The first entries of the **output** from querying Custom Metrics are as follows.

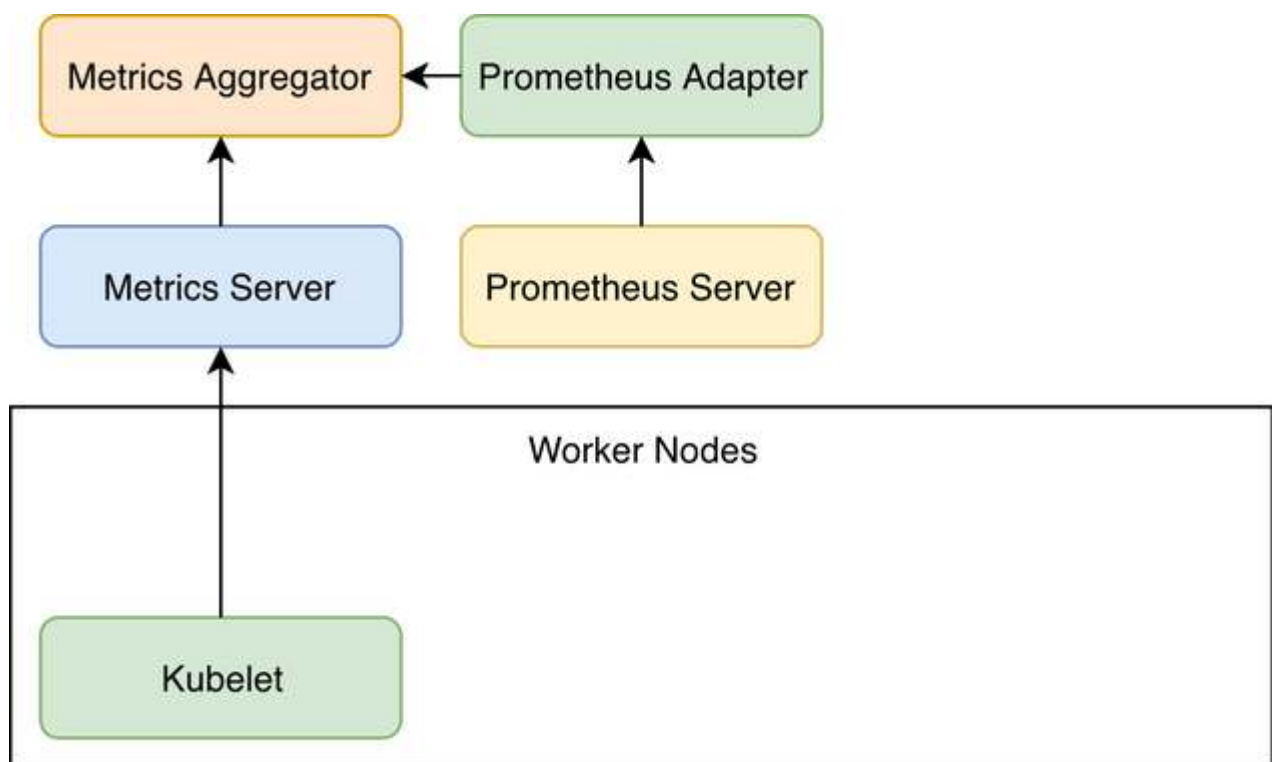
```
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "custom.metrics.k8s.io/v1beta1",
  "resources": [
    {
      "name": "namespaces/memory_max_usage_bytes",
      "singularName": "",
      "namespaced": false,
      "kind": "MetricValueList",
      "verbs": [
        "get"
 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

```
} ,  
...
```

The list of the custom metrics available through the adapter is big, and we might be compelled to think that it contains all those stored in `Prometheus`. We'll find out whether that's true later. For now, we'll focus on the metrics we might need with `HPA` tied to `go-demo-5` Deployment. After all, providing metrics for auto-scaling is an adapter's primary, if not the only function.

Data flow of Metrics Aggregator

From now on, *Metrics Aggregator* contains not only the data from the `Metrics Server` but also those from the `Prometheus Adapter` which, in turn, is fetching metrics from `Prometheus Server`. We are yet to confirm whether the data we're getting through the adapter is enough and whether `HPA` works with custom metrics.



Custom Metrics with Prometheus Adapter (arrows show the flow of data)

Q

The adapter can access the `Prometheus API` through the URL and the port.

Before we go deeper into the adapter, we'll define our goals for the `go-demo-5` application.

Defining goals of `go-demo-5` application

We should be able to scale the Pods not only based on memory and CPU usage but also by the number of requests entering through Ingress or observed through instrumented metrics. There can be many other criteria we could add but, as a learning experience, those should be enough. We already know how to configure `HPA` to scale based on CPU and memory, so our mission is to extend it with a request counter. That way, we'll be able to set the rules that will increase the number of replicas when the application is receiving too many requests, as well as to descale it if traffic decreases.

Install the application

Since we want to extend the `HPA` connected to `go-demo5`, our next step is to install the application.

```
GD5_ADDR=go-demo-5.$LB_IP.nip.io

kubectl create namespace go-demo-5

helm install go-demo-5 \
  https://github.com/vfarcic/go-demo-5/releases/download/0.0.1/go-demo-5-0.0.1.tgz \
  --namespace go-demo-5 \
  --set ingress.host=$GD5_ADDR

kubectl -n go-demo-5 \
  rollout status \
  deployment go-demo-5
```

We defined the address of the application, we installed the Chart, and we waited until the Deployment rolled out.

A note to EKS users

🔍 If you received `error: deployment "go-demo-5" exceeded its progress deadline` message, the cluster is likely auto-scaling to accommodate all the Pods and zones of the PersistentVolumes. That might take more time than the `progress deadline`. In that case, wait for a few moments and repeat the `rollout` command.

Next, we'll generate a bit of traffic by sending a hundred requests to the application through its Ingress resource.

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

Calculate how many requests passed through Ingress

Now that we generated some traffic, we can try to find a metric to help us calculate how many requests passed through Ingress. Since we already know (from the previous chapters) that `nginx_ingress_controller_requests` provides the number of requests that enter through Ingress, we should check whether it is now available as a custom metric.

```
kubectl get --raw \
  "/apis/custom.metrics.k8s.io/v1beta1" \
  | jq '.resources[]
  | select(.name
  | contains("nginx_ingress_controller_requests"))'
```

We sent a request to `/apis/custom.metrics.k8s.io/v1beta1`. But, as you already saw, that alone would return all the metrics, and we are interested in only one. That's why we piped the output to `jq` and used its filter to retrieve only the entries that contain `nginx_ingress_controller_requests` as the `name`.

If you received an empty **output**, please wait for a few moments until the adapter pulls the metrics from `Prometheus` (it does that every ninety seconds), and re-execute the command.

The **output** is as follows.

```

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

```

We got three results. The name of each consists of the resource type and the name of the metric. We'll discard those related to `jobs.batch` and `namespaces`, and concentrate on the metrics related to `ingresses.extensions` since it provides the information we need. We can see that it is `namespaced`, meaning that the metrics are, among other things, separated by namespaces of their origin. The `kind` and the `verbs` are (almost) always the same, and there's not much value going through them.

Average number of requests per replica

The major problem with

`ingresses.extensions/nginx_ingress_controller_requests` is that it provides the number of requests for an Ingress resource. We couldn't use it in its current form as an `HPA` criterion. Instead, we should divide the number of requests with the number of replicas. That would give us the average number

of requests per replica which should be a better `HPA` threshold. We'll explore how to use expressions instead of simple metrics later.

Retrieve `http_server_resp_time_count`

Knowing the number of requests entering through Ingress is useful, but it might not be enough. Since `go-demo-5` already provides instrumented metrics, it would be helpful to see whether we can retrieve

`http_server_resp_time_count`. As a reminder, that's the same metric we used in the [Debugging Issues Discovered Through Metrics And Alerts](#) chapter.

```
kubectl get --raw \
  "/apis/custom.metrics.k8s.io/v1beta1" \
  | jq '.resources[]
  | select(.name
  | contains("http_server_resp_time_count"))'
```

We used `jq` to filter the result so that only `http_server_resp_time_count` is retrieved. Don't be surprised by empty output. That's normal since `Prometheus Adapter` is not configured to process all the metrics from `Prometheus`, but only those that match its own internal rules. So, this might be a good moment to take a look at the `prometheus-adapter` ConfigMap that contains its configuration.

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

The **output** is too big to be presented in a lesson, so we'll go only through the first rule. It is as follows.

```
...
rules:
- seriesQuery: '{__name__=~"^container_.*",container_name!="POD",namespac
e!="",pod_name!=""}'
  seriesFilters: []
  resources:
    overrides:
      namespace:
        resource: namespace
      pod_name:
        resource: pod
  name:
    matches: ^container_(.*)_seconds_total$
```



```

as: ""
metricsQuery: sum(rate(<<.Series>>{<<.LabelMatchers>>,container_name!="P
OD"}[5m]))
  by (<<.GroupBy>>)
...

```

The first rule retrieves only the metrics with the name that starts with `container` (`__name__=~"^container_.*"`), with the label `container_name` being anything but `POD`, and with `namespace` and `pod_name` not empty. Each rule has to specify a few resource overrides. In this case, the `namespace` label contains the `namespace` resource. Similarly, the `pod` resource is retrieved from the label `pod_name`. Further on, we can see the `name` section uses regular expressions to name the new metric. Finally, `metricsQuery` tells the adapter which `Prometheus` query it should execute when retrieving data.

Setup complications

If that setup looks confusing, you should know that you're not the only one bewildered at first sight. `Prometheus Adapter`, just as `Prometheus Server` configs, is complicated to grasp at first. Nevertheless, they are very powerful allowing us to define rules for service discovery, instead of specifying individual metrics (in the case of the adapter) or targets (in the case of `Prometheus Server`). Soon we'll go into more details with the setup of the adapter rules. For now, the important note is that the default configuration tells the adapter to fetch all the metrics that match a few rules.

So far, we saw that `nginx_ingress_controller_requests` metric is available through the adapter, but that it is of no use since we need to divide the number of requests with the number of replicas. We also saw that the `http_server_resp_time_count` metric originating in `go-demo-5` Pods is not available. All in all, we do not have all the metrics we need, while most of the metrics currently fetched by the adapter are of no use. It is wasting time and resources with pointless queries.

Our next mission is to reconfigure the adapter so that only the metrics we need are retrieved from `Prometheus`. We'll try to write our own expressions that will fetch only the data we need. If we manage to do that, we should be able to create `HPA` as well. Let's do all this in the next lesson.

