

Use Instrumentation to Provide More Detailed Metrics

In this lesson, we will see how we can use Instrumentation to provide more detailed metrics.

WE'LL COVER THE FOLLOWING



- Issue with current metrics
- Instrumented metrics in `go-demo-5`
 - `Prometheus` Histogram Vector
 - `recordMetrics` function
- Instrumented metrics of the same type
 - Structure and responsibilities of teams
- `Prometheus` discovery of Node Exporter and Kube State metrics
 - `kubernetes-service-endpoints`
- Write expressions using instrumented metrics
 - Average response time per request
- Requests from `readinessProbe` and the `livenessProbe`

Issue with current metrics

We shouldn't just say that the `go-demo-5` application is slow. That would not provide enough information for us to quickly inspect the code in search of the exact cause of slowness. We should be able to do better and deduce which part of the application is misbehaving. Can we pinpoint a specific path that produces slow responses? Are all methods equally slow, or is the issue limited only to one? Do we know which function produces slowness? There are many similar questions we should be able to answer in situations like that, but we can't with the current metrics. They are too generic, and they can usually only tell us that a specific Kubernetes resource is misbehaving. The metrics we're collecting are too broad to answer application-specific questions.

The metrics we explored so far are a combination of **exporters** and

instrumentations. **Exporters** are in charge of taking existing metrics and converting them into the `Prometheus`-friendly format. An example would be `Node Exporter` which is taking “standard” Linux metrics and converting them into `Prometheus`'s time-series format. Another example is `kube-state-metrics`, which listens to the Kube API server and generates metrics with the state of the resources.

🔍 **Instrumented** metrics are baked into applications. They are an integral part of the code of our apps, and they are usually exposed through the `/metrics` endpoint.

The easiest way to add metrics to your applications is through one of the `Prometheus client libraries`. At the time of this writing, `Go`, `Java` and `Scala`, `Python`, and `Ruby` libraries are officially provided. On top of those, the community supports `Bash`, `C++`, `Common Lisp`, `Elixir`, `Erlang`, `Haskell`, `Lua for Nginx`, `Lua for Tarantool`, `.NET / C#`, `Node.js`, `Perl`, `PHP`, and `Rust`. Even if you code in a different language, you can easily provide `Prometheus`-friendly metrics by outputting results in a text-based `exposition format`.

The overhead of collecting metrics should be negligible and, since `Prometheus` pulls them periodically, outputting them should have a tiny footprint as well. Even if you choose not to use `Prometheus`, or to switch to something else, the format is becoming the standard, and your next metrics collector tool is likely to expect the same data.

All in all, there is no excuse not to bake metrics into your applications, and, as you'll see soon, they provide invaluable information that we cannot obtain from outside.

Instrumented metrics in `go-demo-5`

Let's take a look at an example of the instrumented metrics in *go-demo-5*.


🔍 The application is written in `Go`. Don't worry if that's not your language of choice. We'll just take a quick look at a few examples as a way to understand the logic behind **instrumentation**, not the exact implementation.

The first interesting part is as follows.

```
...
var (
    histogram = prometheus.NewHistogramVec(prometheus.HistogramOpts{
        Subsystem: "http_server",
        Name:       "resp_time",
        Help:       "Request response time",
    }, []string{
        "service",
        "code",
        "method",
        "path",
    })
)
...
```

Prometheus Histogram Vector

We defined a variable that contains a **Prometheus Histogram Vector** with a few options. The **Subsystem** and the **Name** form the base metric **http_server_resp_time**. Since it is a histogram, the final metrics will be created by adding **_bucket**, **_sum**, and **_count** suffixes.

 Please consult [histogram](#) documentation for more info about that **Prometheus' metric type**.

The last part is a string array (**[]string**) that defines all the labels we'd like to add to the metric. In our case, those labels are **service**, **code**, **method**, and **path**. Labels can be anything we need, just as long as they provide sufficient information we might require when querying those metrics.

recordMetrics function

The next point of interest is the **recordMetrics** function.

```
...
func recordMetrics(start time.Time, req *http.Request, code int) {
    duration := time.Since(start)
    histogram.With(
        prometheus.Labels{
```

```

        "service": serviceName,
        "code":    fmt.Sprintf("%d", code),

        "method":  req.Method,
        "path":    req.URL.Path,
    },
).Observe(duration.Seconds())
}
...

```

I created that as a helper function that can be called from different locations in the code. It accepts `start` time, the `Request`, and then return `code` as arguments. The function itself calculates `duration` by subtracting the current `time` with the `start` time. That `duration` is used in the `Observe` function and provides the value of the metric. There are also labels that will help us fine-tune our expressions later on.

Finally, we'll take a look at one of the examples where the `recordMetrics` is invoked.

```

...
func HelloServer(w http.ResponseWriter, req *http.Request) {
    start := time.Now()
    defer func() { recordMetrics(start, req, http.StatusOK) }()
    ...
}
...

```

The `HelloServer` function is the one that returns the `hello, world!` response you already saw quite a few times. The details of that function are not important. In this context, the only part that matters is the line `defer func() { recordMetrics(start, req, http.StatusOK) }()`. In `Go`, `defer` allows us to execute something at the end of the function where it resides. In our case, that something is the invocation of the `recordMetrics` function that will record the duration of a request. In other words, before the execution leaves the `HelloServer` function, it'll record the duration by invoking the `recordMetrics` function.

I won't go further into the code that contains instrumentation since that would assume that you are interested in intricacies behind `Go` and I'm trying to keep this course language-agnostic. I'll let you consult the documentation and examples from your favorite language. Instead, we'll take a look at the

`go-demo-5` instrumented metrics in action.

```
kubectl -n metrics \  
  run -it test \  
  --image=appropriate/curl \  
  --restart=Never \  
  --rm \  
  -- go-demo-5.go-demo-5:8080/metrics
```

We created a Pod based on the `appropriate/curl` image, and we sent a request through the Service using the address `go-demo-5.go-demo-5:8080/metrics`. The first `go-demo-5` is the name of the Service, and the second is the Namespace where it resides. As a result, we got output with all the instrumented metrics available in that application. We won't go through all of them, but only those created by the `http_server_resp_time` histogram.

The relevant parts of the **output** are as follows.

```
...  
# HELP http_server_resp_time Request response time  
# TYPE http_server_resp_time histogram  
http_server_resp_time_bucket{code="200",method="GET",path="/demo/hello",service="go-demo",le="0.005"} 931  
http_server_resp_time_bucket{code="200",method="GET",path="/demo/hello",service="go-demo",le="0.01"} 931  
http_server_resp_time_bucket{code="200",method="GET",path="/demo/hello",service="go-demo",le="0.025"} 931  
http_server_resp_time_bucket{code="200",method="GET",path="/demo/hello",service="go-demo",le="0.05"} 931  
http_server_resp_time_bucket{code="200",method="GET",path="/demo/hello",service="go-demo",le="0.1"} 934  
http_server_resp_time_bucket{code="200",method="GET",path="/demo/hello",service="go-demo",le="0.25"} 935  
http_server_resp_time_bucket{code="200",method="GET",path="/demo/hello",service="go-demo",le="0.5"} 935  
http_server_resp_time_bucket{code="200",method="GET",path="/demo/hello",service="go-demo",le="1"} 936  
http_server_resp_time_bucket{code="200",method="GET",path="/demo/hello",service="go-demo",le="2.5"} 936  
http_server_resp_time_bucket{code="200",method="GET",path="/demo/hello",service="go-demo",le="5"} 937  
http_server_resp_time_bucket{code="200",method="GET",path="/demo/hello",service="go-demo",le="10"} 942  
http server resp time bucket{code="200",method="GET",path="/demo/hello",se
```

```
service="go-demo",le="+Inf"} 942
http_server_resp_time_sum{code="200",method="GET",path="/demo/hello",service="go-demo"} 38.87928942600006
http_server_resp_time_count{code="200",method="GET",path="/demo/hello",service="go-demo"} 942
...
```

We can see that the `Go library` we used in the application code created quite a few metrics from the `http_server_resp_time` histogram. We got one for each of the twelve buckets (`http_server_resp_time_bucket`), one with the sum of the durations (`http_server_resp_time_sum`), and one with the count (`http_server_resp_time_count`). We would have much more if we made requests that would have different labels. For now, those fourteen metrics are all coming from requests that responded with the HTTP code `200`, that used the `GET` method, that were sent to the `/demo/hello` path, and that is coming from the `go-demo` service (application). If we create requests with different methods (e.g., `POST`) or to different paths, the number of metrics would increase. Similarly, if we implement the same instrumented metric in other applications (but with different `service` labels), we'd have metrics with the same key (`http_server_resp_time`) that would provide insights into multiple apps. That raises the question of whether we should unify metric names across all the apps, or not.

Instrumented metrics of the same type

I prefer having instrumented metrics of the same type with the same name across all the applications. For example, all those that collect response times can be called `http_server_resp_time`. That simplifies querying data in `Prometheus`. Instead of learning about instrumented metrics from each individual application, learning those from one provides knowledge about all. On the other hand, I am in favor of giving each team full control over their applications. That includes the decisions which metrics to implement, and how to call them.

Structure and responsibilities of teams

All in all, it depends on the structure and responsibilities of the **teams**. If a **team** is entirely in charge of their applications and they debug problems specific to their apps, there is no inherent need for standardization of the

names of instrumented metrics. On the other hand, if monitoring is centralized and the other **teams** might expect help from experts in that area, creating naming conventions is a must. Otherwise, we could easily end up with thousands of metrics with different names and types, even though most of them are providing the same information.

For the rest of this chapter, I will assume that we did agree to have `http_server_resp_time` histogram in all applications, where that's applicable.

Prometheus discovery of Node Exporter and Kube State metrics

Now, let's see how we can tell `Prometheus` that it should pull the metrics from the `go-demo-5` application. It would be even better if we could tell `Prometheus` to pull data from all the apps that have instrumented metrics. Actually, now when I think about it, we have not yet discussed how `Prometheus` finds Node Exporter and Kube State Metrics in the previous chapter. So, let's go briefly through the discovery process.

A good starting point is the `Prometheus` target screen.

```
open "http://$PROM_ADDR/targets"
```

kubernetes-service-endpoints

The most interesting group of targets is `kubernetes-service-endpoints`. If we take a closer look at the labels, we'll see that each has `kubernetes_name` and that three of the targets have it set to `go-demo-5`. `Prometheus` somehow found that we have three replicas of the application and that metrics are available through the port `8080`. If we look further, we'll notice that `prometheus-node-exporter` is there as well, one for each node in the cluster. The same goes for `prometheus-kube-state-metrics`. There might be others in that group.



If monitoring is centralized and the other **teams** might expect help from experts in that area, creating naming conventions is not a must.

COMPLETED 0%

1 of 1

**kubernetes-service-endpoints (8/11 up)** [show less](#)

Endpoint	State	Labels	Last Scrape	Error
http://10.0.2.15:9100/metrics	UP	<code>app="prometheus"</code> <code>chart="prometheus-7.1.3"</code> <code>component="node-exporter"</code> <code>heritage="Tiller"</code> <code>instance="10.0.2.15:9100"</code> <code>kubernetes_name="prometheus-node-exporter"</code> <code>kubernetes_namespace="metrics"</code> <code>release="prometheus"</code>	26.001s ago	
http://172.17.0.12:8080/metrics	UP	<code>app="go-demo-5"</code> <code>chart="go-demo-5-0.0.1"</code> <code>heritage="Tiller"</code> <code>instance="172.17.0.12:8080"</code> <code>kubernetes_name="go-demo-5"</code> <code>kubernetes_namespace="go-demo-5"</code> <code>release="go-demo-5"</code>	21.523s ago	
http://172.17.0.13:8080/metrics	UP	<code>app="go-demo-5"</code> <code>chart="go-demo-5-0.0.1"</code> <code>heritage="Tiller"</code> <code>instance="172.17.0.13:8080"</code> <code>kubernetes_name="go-demo-5"</code> <code>kubernetes_namespace="go-demo-5"</code> <code>release="go-demo-5"</code>	49.894s ago	
http://172.17.0.14:8080/metrics	UP	<code>app="go-demo-5"</code> <code>chart="go-demo-5-0.0.1"</code> <code>heritage="Tiller"</code> <code>instance="172.17.0.14:8080"</code> <code>kubernetes_name="go-demo-5"</code> <code>kubernetes_namespace="go-demo-5"</code> <code>release="go-demo-5"</code>	29.902s ago	

kubernetes-service-endpoints Prometheus' targets

Prometheus discovered all the targets through **Kubernetes Services**. It extracted the port from each of the **Services**, and it assumed that data is available through the `/metrics` endpoint. As a result, every application we have in the cluster, that is accessible through a **Kubernetes Service**, was automatically added to the `kubernetes-service-endpoints` group of **Prometheus' targets**. There was no need for us to fiddle with the **Prometheus' configuration** to add `go-demo-5` to the mix. It was just discovered. Pretty neat, isn't it?

🔍 In some cases, some of the metrics will not be accessible, and that target will be marked as red. As an example, *kube-dns* in minikube is not reachable from *Prometheus*. That's common, and it's not a reason to be alarmed, just as long as that's not one of the metric sources we do need.

Write expressions using instrumented metrics

Next, we'll take a quick look at a few expressions we can write using the instrumented metrics coming from *go-demo-5*.

```
open "http://$PROM_ADDR/graph"
```

Please type the expression that follows, press the *Execute* button, and switch to the *Graph* tab.

```
http_server_resp_time_count
```

We can see three lines that correspond to three replicas of *go-demo-5*. That should not come as a surprise since each of those is pulled from instrumented metrics coming from each of the replicas of the application. Since those metrics are counters that can only increase, the lines of the graph are continuously going up.

☐ Enable query history

http_server_resp_time_count

Load time: 32ms
Resolution: 14s
Total time series: 3

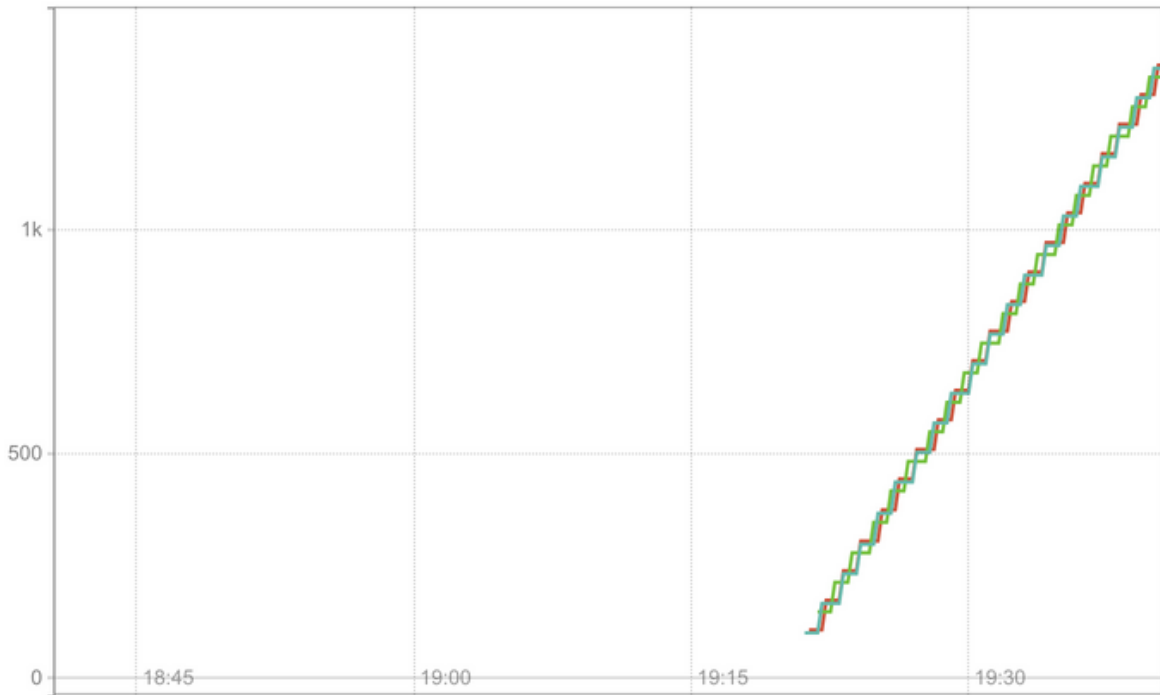
Execute

- insert metric at cursor -

Graph

Console

1h Until Res. (s) stacked



The graph with the http_server_resp_time_count counter



kube-dns in minikube is reachable from Prometheus .

COMPLETED 0%

1 of 1



That wasn't very useful. If we were interested in the rate of the count of

requests, we'd envelop the previous expression inside the `rate()` function.

We'll do that later. For now, we'll write the simplest expression that will give us the average response time per request.

Average response time per request

Please type the expression that follows, and press the *Execute* button.

```
http_server_resp_time_sum{
  kubernetes_name="go-demo-5"
} /
http_server_resp_time_count{
  kubernetes_name="go-demo-5"
}
```

The expression itself should be easy to understand. We are taking the sum of the response time over all the requests and dividing it with the count. Since we already discovered that the problem is somewhere inside the `go-demo-5` app, we used the `kubernetes_name` label to limit the results. Even though it is the only application with that metric currently running in our cluster, it is a good idea to get used to the fact that there might be others at some later date when we extend instrumentation to other applications.

We can see that the average request duration increased for a while, only to drop close to the initial values a while later. That spike coincides with the twenty slow requests we sent a while ago. In my case (screenshot below), the peak is close to the average response time of 0.1 seconds, only to drop to around 0.02 seconds a while later.

☐ Enable query history

```
http_server_resp_time_sum{
  kubernetes_name="go-demo-5"
} /
http_server_resp_time_count{
  kubernetes_name="go-demo-5"
}
```

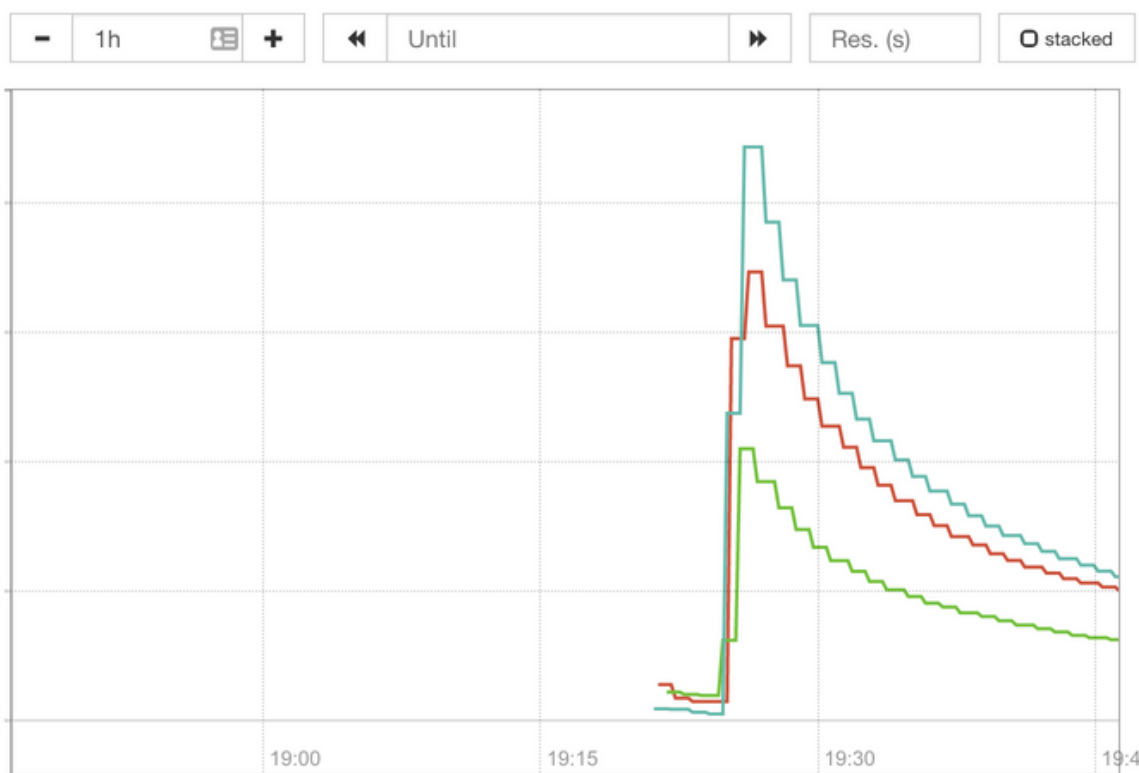
Load time: 50ms
Resolution: 14s
Total time series: 3

Execute

- insert metric at cursor -

Graph

Console



The graph with the cumulative average response time

Please note that the expression we just executed is deeply flawed. It shows the cumulative average response time, instead of displaying the **rate**. But, you already knew that. That was only a taste of the instrumented metric, not its “real” usage (that comes soon).

Requests from **readinessProbe** and the **livenessProbe**

You might notice that even the spike is very low. It is certainly lower than

your might notice that even the spike is very low. It is certainly lower than what we'd expect from sending only twenty slow requests through `curl`. The

reason for that lies in the fact that we were not the only ones making those requests. The `readinessProbe` and the `livenessProbe` are sending requests as well, and they are very fast. Unlike in the previous chapter when we were measuring only the requests coming through Ingress, this time we're capturing all the requests that enter the application, and that includes health-checks.

Now that we have seen a few examples of the `http_server_resp_time` metric that is generated inside our `go-demo-5` application, in the next lesson, we will use that knowledge to try to debug the simulated issue that led us towards instrumentation.