Using Internal Metrics to Debug Potential Issues

In this lesson, we will use internal metrics to debug the potential issues we face.

WE'LL COVER THE FOLLOWING

- Issue with `nginx_ingress_controller_request_duration_seconds`
- Switch to `http_server_resp_time` metric
  - Include `labels` into expressions
  - Adding a threshold
- Benefit of Instrumentation
- Combine generic metrics with detailed metrics

We'll resend requests with slow responses again so that we get to the same point where we started this chapter.

```
for i in {1..20}; do
    DELAY=$[ $RANDOM % 10000 ]
    curl "http://$GD5_ADDR/demo/hello?delay=$DELAY"
done

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

We sent twenty requests that will result in responses with random duration (up to ten seconds). Later on, we opened the `Prometheus' alerts screen`.

A while later, the *AppTooSlow* alert should fire (remember to refresh your screen), and we have a (simulated) problem that needs to be solved. Before we start panicking and do something hasty, we'll try to find the cause of the issue.

Please click the expression of the *AppTooSlow* alert.

Issue with `nginx_ingress_controller_request_duration_sec`

onds

We are redirected to the graph screen with the pre-populated expression from the alert. Feel free to click the *Expression* button, even though it will not provide any additional info, apart from the fact that the application was fast, and then it slowed down for some inexplicable reason. You will not be able to gather more details from that expression. You will not know whether it's slow on all methods, whether only a specific path responds slow, nor much of any other application-specific details. Simply put, the `nginx_ingress_controller_request_duration_seconds` metric is too generic. It served us well as a way to notify us that the application's response time increased, but it does not provide enough information about the cause of the issue. For that, we'll switch to the `http_server_resp_time` metric `Prometheus` is retrieving directly from `go-demo-5` replicas.

Switch to `http_server_resp_time` metric

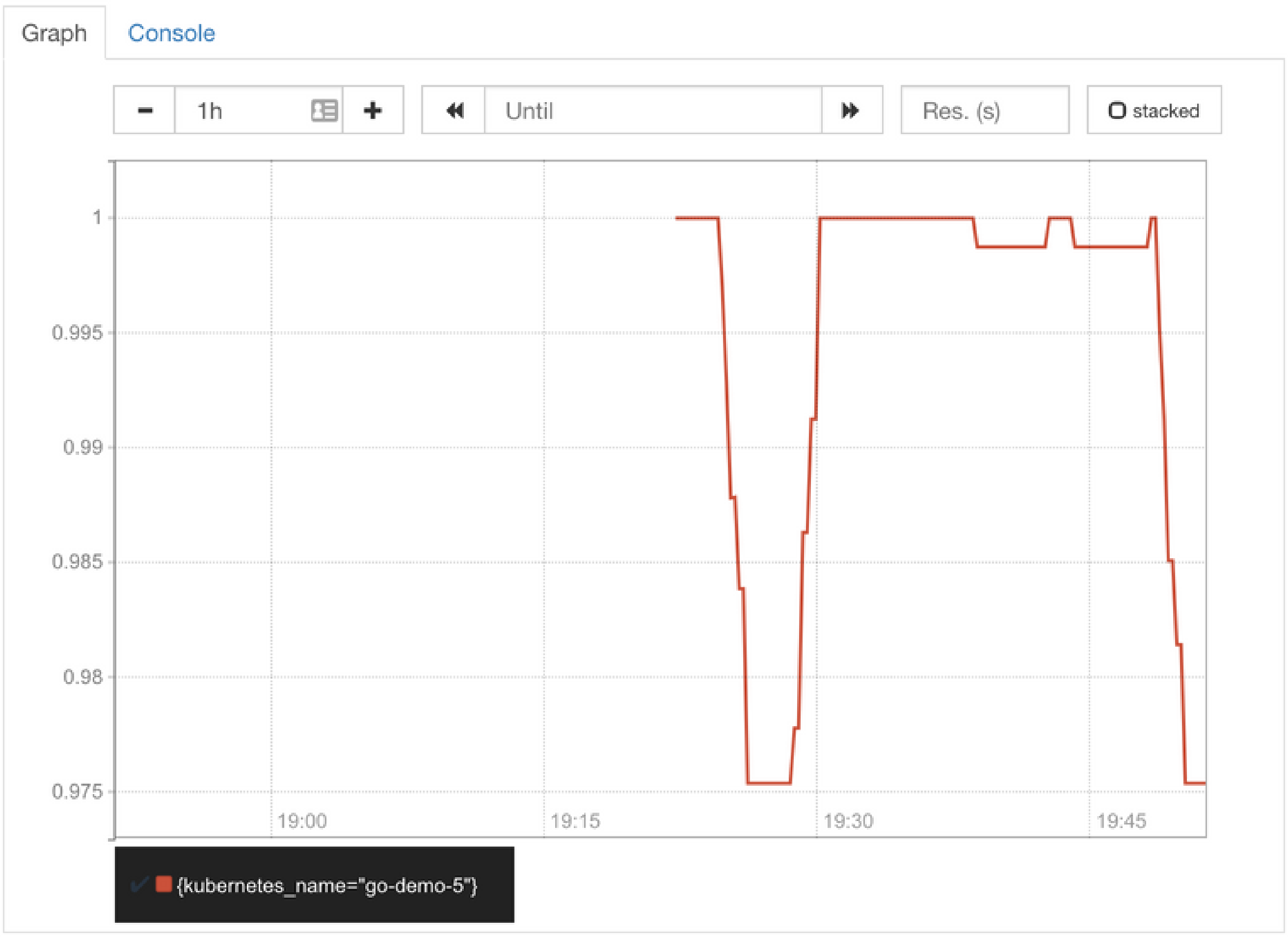
Please type the expression that follows, and press the *Execute* button.

```
sum(rate(
    http_server_resp_time_bucket{
        le="0.1",
        kubernetes_name="go-demo-5"
    }[5m]
)) /
sum(rate(
    http_server_resp_time_count{
        kubernetes_name="go-demo-5"
    }[5m]
))
```

Switch to the Graph tab, if you're not there already.

That expression is very similar to the queries we wrote before when we were using the `nginx_ingress_controller_request_duration_seconds_sum` metric. We are dividing the rate of requests in the 0.1 seconds bucket with the rate of all the requests.

In my case (screenshot below), we can see that the percentage of fast responses dropped twice. That coincides with the simulated slow requests we sent earlier.



The graph with the percentage of fast requests measured with instrumented metrics

Q

The `nginx_ingress_controller_request_duration_seconds` did not provide enough information about the cause of the issue.

COMPLETED 0%

1 of 1

So far, using the instrumented metric `http_server_resp_time_count` did not provide any tangible benefit when compared with `nginx_ingress_controller_request_duration_seconds_sum`. If that would be all, we could conclude that the effort to add instrumentation was a waste.

However, we did not yet include `labels` into our expressions.

Include `labels` into expressions

Let’s say that we’d like to group requests by the `method` and the `path`. That could give us a better idea of whether slowness is global, or limited only to a specific type of request. If it’s latter, we’ll know where the problem is and, hopefully, will be able to find the culprit quickly.

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

```
sum(rate(
    http_server_resp_time_bucket{
        le="0.1",
        kubernetes_name="go-demo-5"
    }[5m]
))
by (method, path) /
sum(rate(
    http_server_resp_time_count{
        kubernetes_name="go-demo-5"
    }[5m]
))
by (method, path)
```

That expression is almost the same as the one before. The only difference is the addition of the `by (method, path)` statements. As a result, we are getting a percentage of fast responses, grouped by the `method` and the `path`.

The **output** does not represent a “real” world use case. Usually, we’d see many different lines, one for each method and path that was requested. But, since we only made requests to */demo/hello* using HTTP GET, our graph is a bit boring. You’ll have to imagine that there are many other lines.

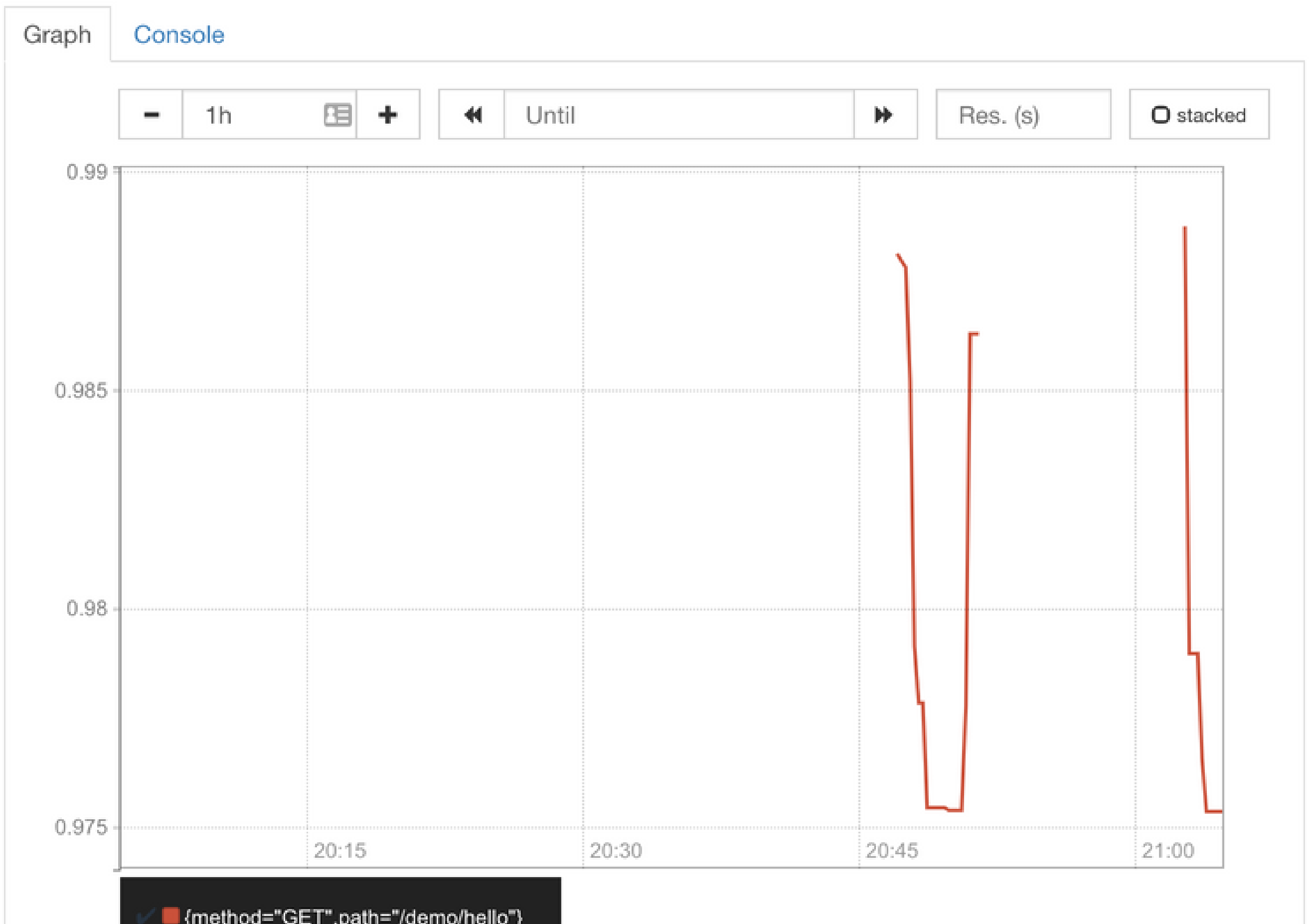
By studying the graph, we discover that all but one line (we’re still imagining many) are close to a hundred percent of fast responses. The one with the sharp drop would be the one with the */demo/hello* `path` and the GET `method`. However, if that would indeed be a real-world scenario, we would probably have too many lines in the graph, and we might not be able to distinguish them easily. Our expression could benefit from the addition of a threshold.

Adding a threshold

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

```
sum(rate(
    http_server_resp_time_bucket{
        le="0.1",
        kubernetes_name="go-demo-5"
    }[5m]
))
by (method, path) /
sum(rate(
    http_server_resp_time_count{
        kubernetes_name="go-demo-5"
    }[5m]
))
by (method, path) < 0.99
```

The only addition is the `< 0.99` threshold. As a result, our graph excludes all the results (all paths and methods) but those that are below 99 percent (0.99). We removed all the noise and focused only on the cases when more than one percent of all requests are slow (or less than 99 percent are fast). The result is now clear. The problem is in the function that handles *GET* requests on the path */demo/hello*. We know that through the `labels` available below the graph.



The graph with the percentage of fast requests measured with instrumented metrics limited to results below ninety-nine percent

Now that we know (almost) the exact location of the problem, all that's left is to fix the issue, push the change to our Git repository, and wait until our continuous deployment process upgrades the software with the new release.

Benefit of Instrumentation

In a relatively short period, we managed to find (debug) the issue or, to be more precise, to narrow it to a specific part of the code. Or, maybe we discovered that the problem is not in the code, but that our application needs to scale up. In either case, without instrumented metrics, we would only know that the application is slow and that could mean that any part of the app is misbehaving. Instrumentation gave us more detailed metrics that we used to be more precise and reduce the time we'd typically require to find the issue and act accordingly.

Combine generic metrics with detailed metrics

Usually, we'd have many other instrumented metrics, and our "debugging" process would be more complicated. We'd execute other expressions and dig through different metrics. Nevertheless, the point is that we should combine generic metrics with more detailed ones coming directly from our applications. Those in the former group are often used to detect that there is an issue, while the latter type is useful when looking for the cause of the problem. Both types of metrics have their place in monitoring, alerting, and debugging our clusters and applications. With instrumented metrics, we have more application-specific details. That allows us to narrow down the locations and causes of a problem. The more confident we are about the exact cause of an issue, the better we are equipped to react.

In the next lesson, we will revise and test the concepts of this chapter, through a short quiz.