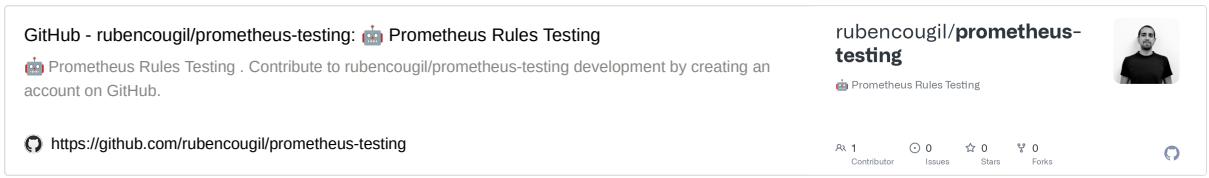
Talk



1. Introduction

In this session, we will discuss one of the unique features of Prometheus: the ability to programmatically test alerting rules.

When individuals are on-call (meaning they need to be available outside of regular working hours to respond to incidents, including nights, weekends, and holidays), they hope that they won't be rudely awakened at 4 am.

Unfortunately, sometimes an issue arises, disrupting the system's normal functioning. This can lead to frustrated customers and significant revenue loss during such downtime. In such cases, we want to be notified as soon as possible with relevant information about the problem and ideally receive tips for putting a solution in place (usually found in "Runbooks").

However, there are also instances of "false positives." The on-call person receives a notification, rushes to their laptop, investigates the issue, only to discover that there was no real problem. Nada. Sometimes, the alert itself is flawed. Since humans write alert rules, they are prone to human errors, and a mistake in the expression can lead to such situations. Unfortunately, in many monitoring tools, there is no way to identify these issues in advance. This is because these expressions require real data for evaluation, usually obtained from live environments.

Prometheus addresses this problem with a built-in mechanism. It provides a tool called `promtool` , which allows people to test alert rules before deployment. This ensures that what the developer has in mind when writing the rule aligns with its actual behavior. The `promtool` offers this testing capability among other features.

These tests follow a specific syntax but are similar in essence to unit tests in any programming language. They assert that the code performs as intended.

Moreover, you can integrate these tests into a continuous integration (CI) pipeline, so they run whenever someone attempts to modify or create a rule. If there is a change in the expression that affects how the rule is evaluated, the test will fail, providing prompt feedback to the developer regarding the impact of the change.

Implementing this feature will undoubtedly improve your sleep quality. Here, we often use the phrase "sleep like a baby" to signify sound sleep. However, in my opinion, it's one of the biggest lies in human history. Trust me, I recently became a father,

and "sleeping like a baby" actually means waking up every 45 minutes, crying loudly due to various unidentified and inexpressible reasons.

We will dive into these details shortly. However, before that, let's provide a brief explanation of how Prometheus works and why it is crucial to understand how alerting rules are evaluated.

2. Bit of context about Prometheus

As you may already know, Prometheus has skyrocketed in popularity as a monitoring platform. It joined the Cloud Native Computing Foundation (CNCF) back in 2016, proudly claiming the second spot after Kubernetes. The moment SoundCloud unleashed its code, the adoption was massive, with everyone embracing Prometheus with open arms. Its effectiveness had already been proven because Prometheus follows in the footsteps of Google's in-house platform, Borgmon. They share a multitude of similarities: the pull-based metric gathering mechanism, the nearly identical query language (DSL), and the dedicated Time Series Database (TSDB) for data storage.

Prometheus is now the de-facto standard for monitoring in Kubernetes environments. It has very powerful service discovery implementations that allow developers to easily configure it for scalable deployments. Also, it has lots of "exporters" for well-known technologies, providing the ability to export the metrics in the appropriate format.

3. How Alerting Rules work

An alerting rule is simply an expression written in PromQL language that undergoes frequent evaluation to determine if it meets specific criteria. When the rule fails to comply, it transitions from an "inactive" state to an "active" state and eventually becomes "pending" or "firing."

When an alerting rule is "firing," it sends a notification to an external component called Alert Manager, which is part of the Prometheus Stack. The Alert Manager usually knows what to do: send an email, a message to a Slack Channel, or even ping PagerDuty, which potentially could wake you up in the middle of the night.

All of this depends on custom labels, such as "severity," which are added to the existing set of labels for the rule. These labels help group related alerts together. Additionally, annotations come into play. Annotations are extra parameters added to the rule, like a "summary," "description", or perhaps a link to a helpful "Runbook." They are incredibly valuable when someone receives an alert notification and needs to swiftly understand what went wrong and how to fix it.

PromQL is an incredibly powerful DSL language, but it can be challenging to master. That's why thorough testing of rules is crucial. In complex queries, the rule's outcome may not align with your expectations. Therefore, by testing the rules, you can promptly identify and address such issues.

4. How to Unit Test a Rule

A useful command-line tool called `promtool` is included with a regular installation of Prometheus. It enables the execution of files containing unit tests for rules, along with other functionalities. The syntax for executing these tests is as follows:

`promtool test rules test.yml`

Similar to rules, the file containing the tests is specified in YAML format.

```
rule_files:
    - alerts.yml

evaluation_interval: 1m #1m is the default

tests:
  [ - <test_group> ]
```

`alerts.yml` is a file that contains an alerting rule. This is the content of this file:

```
groups:
  - name: example
    rules:
      - alert: InstancesDownV1
        expr: sum(up{job="app"}) == 0
        labels:
          severity: sev1
        annotations:
          summary: "All instances of App are down"
            description: "All instances of App are down"
```

The PromQL expression `sum(up{job="app"}) == 0` implies that the rule evaluation will yield a `true` value if the *sum* of the uptime of all scraped targets with the name "app" equals zero.

It is crucial to ensure that the expression accurately represents the intended meaning in plain words. While this example may seem relatively straightforward, it's important to consider more complex PromQL queries like the following:

```
sum by(namespace, application) (increase(http_server_requests_seconds_count{namespace="live", application="backend", status=~"500|503"
sum by(namespace, application) (increase(http_server_requests_seconds_count{namespace="live", application="backend"}[10m])) * 100 > 10
and ON() sum by(namespace, application) (increase(http_server_requests_seconds_count{namespace="live", application="backend"}[10m])) >
```

Understanding complex PromQL queries, especially those involving counters and functions like `increase()` or `rate()`, can indeed take more time and effort. These queries often require converting measurements into percentages on the fly, making the evaluation even more challenging.

The complexity of the query increases the chances of errors, making it crucial to thoroughly test it. However, manual execution in an environment with real data is the only way to validate its functionality. Since the data cannot be altered, it's impossible to achieve 100% certainty that the query will work correctly in all potential scenarios.

Returning to the previous simple expression, `sum(up{job="app"}) == 0`, let's examine the various components involved in a test before coding it:

```
interval: 1m

input_series:
  - series: 'up{job="app", instance="app-1:2223"}'
    # 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    values: "0x14"

  - series: 'up{job="app", instance="app-2:2223"}'
    # 1 1 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1
    values: "1x4 0x9 1x4"

alert_rule_test:

  - eval_time: 4m
    alertname: InstancesDownV1

  - eval_time: 5m
    alertname: InstancesDownV1
    exp_alerts:
      - exp_labels:
          severity: page
        exp_annotations:
          summary: "All instances of the App are down"
          description: "All instances of the App are down"

  - eval_time: 15m
    alertname: InstancesDownV1
```

- **Interval**: The interval represents the time window within which a metric reports a single value. In this example, the metric `up` will report either `0` or `1` every minute.
- **Input series**: The input series corresponds to the "Arrange" stage in the Unit Test framework. It involves specifying the values reported by one or more series belonging to the same metric (dimensions). In the provided example, two different series belong to the metric `up` with different instances:
  - `up{job="app", instance="app-1:2223"}`
  - `up{job="app", instance="app-2:2223"}`

  Prometheus will report values for each series independently.

  Some explanation about the notation here:

  - `a+bxc` becomes `a a+b a+(2 b) a+(3 b) … a+(c*b)`
  - `a-bxc` becomes `a a-b a-(2 b) a-(3 b) … a-(c*b)`
  - `_` represents a missing sample from scrape
  - `stale` represents a missing sample from scrape. A time series goes "stale" when it has no samples in the last 5 minutes.

- **Alert rule test**: The alert rule test corresponds to the "Assert" stage (the preceding "Act" stage is implicit and executed automatically by `promtool`). Here, we define the expected behavior over time. In the example, after 5 minutes, the `exp_alerts` should contain no content, indicating that the alert is "inactive." However, after 15 minutes, the alert should be "firing," and we can assert the content of labels and annotations provided by the rule. It's important to note that labels and annotations may have dynamic values depending on the values in the PromQL query itself.

5. Tips and Tricks

After some time already working with Unit Tests in Prometheus Alerts, I've decided to gather some recommendations. There's not too much information about Prometheus Unit Testing out theres, that's why this section is so opinionated.

1. Samples can be empty (`_` or `stale`)

Sometimes in the tests we are tempted to set `0` when a metric is not being reported, and this is wrong. A metric that is not reported should be represented as `_` which is not the same as `0` (`0` is a valid value for a metric). Be mind with that and always take into account this kind of situation when coding a test.

In the example we shown before we relied in the ability of `app` to report `0` when it's down. But what it really happens is that if it is down it won't report anything, so the sample should goes `_`. instead. Let's rewrite our test:

```
interval: 1m

input_series:

  - series: 'up{job="app", instance="app-1:2223"}'
    # _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
    values: "_x14"

  - series: 'up{job="app", instance="app-2:2223"}'
    # 1 1 1 1 1 _ _ _ _ _ _ _ _ _ _ 1 1 1 1 1
    values: "1x4 _x9 1x4"
```

Now, the test should fail. Because *sum* of empty samples won't be never equals to `0` . We need to update our expression and add `... OR on() vector(0)` :

- `vector(0)` returns `0` when sample is empty or stale.
- `on()` Prometheus uses label matching in expressions. If your expression returns anything with labels, it won't match the time series generated by `vector(0)`. In order to make this possible, it's necessary to tell Prometheus explicitly to not trying to match any labels by adding `on()`.

Finally the expression is: `sum(up{job="app"} OR on() vector(0)) == 0` .

There's another thing to be aware of, and is the "staleness" handling of Prometheus. "staleness" refers to the state of a metric or a sample when it becomes outdated or no longer considered fresh or current. From the documentation:

> *If no sample is found (by default) 5 minutes before a sampling timestamp, no value is returned for that time series at this point in time. This effectively means that time series "disappear" from graphs at times where their latest collected sample is older than 5 minutes or after they are marked stale.*

Then, taking into account those 5 minutes, the tests should looks like this:

```
interval: 1m

input_series:

  - series: 'up{job="app", instance="app-1:2223"}'
    # _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
    values: "_x14"

  - series: 'up{job="app", instance="app-2:2223"}'
    # 1 1 1 1 1 _ _ _ _ _ _ _ _ _ _ 1 1 1 1 1
    values: "1x4 _x9 1x4"

alert_rule_test:

  - eval_time: 4m
    alertname: InstancesDownV2
```

```
      # This is 10 minutes and not 6 because of staleness handling.
      # See https://prometheus.io/docs/prometheus/latest/querying/basics/#staleness
      - eval_time: 10m
        alertname: InstancesDownV2
        exp_alerts:
          - exp_labels:
              severity: page
            exp_annotations:
              summary: "All instances of the App are down"
              description: "All instances of the App are down"

      - eval_time: 15m
        alertname: InstancesDownV2
```

There's another way of achievent this and is by using `absent()` function. It will return `1` value if the result of the scrapping is empty. So we could rephrase our expresion as `absent(up{job="app"})` , when the `up` query returns empty, `absent()` will return `1` and this will be evaluated as `true` , so the alert will fire, and the test will pass.

2. Usage of `for` in Rules

There is a commonly used parameter in Alerting rules called `for` in Prometheus. This parameter determines how long Prometheus should check if an alert remains active during each evaluation period before actually firing the alert. During this time, elements that are considered "active" but not yet "firing" are in the "pending" state.

For instance, we can set the value of `for` to 1 minute:

```
groups:
  - name: InstancesDownV3
    rules:
      - alert: InstancesDownV3
        expr: sum(up{job="app"} OR on() vector(0)) == 0
        for: 1m
        labels:
          severity: page
        annotations:
          summary: "All instances of the App are down"
          description: "All instances of the App have been down for more than 1 minute(s)."
```

This single change can cause our unit test to fail, which is a common occurrence. The reason behind this failure is that we need to consider the delay time specified by the `for` parameter before the alert actually starts firing. Unfortunately, there is no direct way to test whether the alert is in the "pending" state.

To address this issue, we need to modify our test by waiting for the alert to fire after an additional minute. The updated test code would look like this:

```
      - eval_time: 4m
        alertname: InstancesDownV3

      - eval_time: 11m # <-- Changed because of the "for" clause
        alertname: InstancesDownV3
        exp_alerts:
          - exp_labels:
              severity: page
            exp_annotations:
              summary: "All instances of the App are down"
              description: "All instances of the App have been down for more than 1 minute(s)."

      - eval_time: 15m
        alertname: InstancesDownV3
```

3. Unit Test File next to the Rule File

We have developed an approach inspired by Golang to organize our unit tests by placing the test file next to the corresponding rule file. The naming convention we adopted is to use the same base name for both files, but append `*_test.yaml` at the end of the test file's name. This helps us easily locate and modify tests.

However, we encountered a limitation with the `promtool` command used to execute the tests. Surprisingly, it doesn't support passing a folder or a regular expression to include multiple test files. Instead, it requires a list of individual file names as arguments. This complicates our approach of having one test per rule, as we would need to update the command each time we add a new test, especially when running the entire test suite in a CI pipeline.

To overcome this limitation, we implemented a solution in our Makefile by parsing the folder and files, and manually creating a string that contains all the test file names. There are various ways to achieve this, but I will show you how I did it using the Makefile:

```
#!/bin/sh

TESTS := $(shell find ./etc/prometheus/rules -type f -regex '^.*_test\.yml$$' | sed 's/^.//' | xargs )

test:
  @docker compose exec -w '/etc/prometheus/rules' prometheus promtool test rules $(TESTS)
```

4. Prometheus scalability and Alerting

Prometheus is meant to be easy to install and easy to use. Unlike other software platforms that make scalability a convoluted mess of clustering and sharding mechanisms, the Prometheus creators deliberately opted for a different path—it doesn't scale horizontally.

That means that you can have multiple Prometheus instances but they do not talk to each other. They can scrape the same targets and store the metrics in their internal TSDB. If you have a Grafana instance to visualize the metrics, you'd have to configure multiple Data Sources that point to each different Prometheus instance (because, hey, there's no such thing as a "*connect to the cluster*" like you might find in RDS or Elasticsearch).

The way of solving this problem is just to forward the metrics to an external storage platform that will allow horizontal scaling (like Cortex or Thanos). Therefore, all the Prometheus instances will send metrics to a centralized place that can be attached as a unique Data Source for Grafana.

One important thing to take into account is that even if you enable such a mechanism to forward metrics to an external storage, Prometheus will still store the metrics in its internal TSDB (with a retention policy of 15 days or by surpassing a certain size on disk). And this is important because the alerting rules configured in Prometheus rely on data from its internal TSDB. That's precisely why alerting rules are crafted with short time windows like 5 minutes, 1 hour, or maybe 24 hours, but rarely beyond that.

6. Conclusions

Unit testing Prometheus rules offers several benefits:

1. **Ensuring rule correctness**: By writing tests, you can validate that your rules behave as expected and produce the desired results.
2. **Early detection of issues**: Unit tests help catch issues and bugs in your rules early in the development process. By identifying problems during testing, you can address them before deploying the rules to production, reducing the risk of false alerts or missed notifications.
3. **Facilitating code refactoring**: Unit tests provide a safety net when refactoring or making changes to your rule configurations.
4. **Supporting continuous integration and deployment**: Incorporating unit tests into your CI/CD pipeline ensures that your rules are thoroughly tested before deployment.
5. **Promoting collaboration and documentation**: Unit tests act as documentation for your rules, providing clear examples of their behavior and intended functionality.

Overall, unit testing for Prometheus rules helps increase the reliability, stability, and maintainability of your monitoring system by catching issues early, preventing regressions, and supporting ongoing development and deployment processes.