GETTING STARTED

This guide is a "Hello World"-style tutorial which shows how to install, configure, and use Prometheus in a simple example setup. You will download and run Prometheus locally, configure it to scrape itself and an

- Downloading and running Prometheus
- Configuring Prometheus to monitor itself
- Starting Prometheus

example application, and then work with queries, rules, and graphs to make use of the collected time series data.

- Using the expression browser
- Using the graphing interface
- Starting up some sample targets
- Configuring Prometheus to monitor the sample targets
- Configure rules for aggregating scraped data into new time series

Downloading and running Prometheus

Download the latest release (/download) of Prometheus for your platform, then extract and run it:

```
tar xvfz prometheus-*.tar.gz
cd prometheus-*
```

Before starting Prometheus, let's configure it.

Configuring Prometheus to monitor itself

Prometheus collects metrics from monitored targets by scraping metrics HTTP endpoints on these targets. Since Prometheus also exposes data in the same manner about itself, it can also scrape and monitor its own health.

While a Prometheus server that collects only data about itself is not very useful in practice, it is a good starting example. Save the following basic Prometheus configuration as a file named `prometheus.yml`:

```
global:
  scrape_interval:     15s # By default, scrape targets every 15 seconds.

  # Attach these labels to any time series or alerts when communicating with
  # external systems (federation, remote storage, Alertmanager).
  external_labels:
    monitor: 'codelab-monitor'

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this conf
  - job_name: 'prometheus'

    # Override the global default and scrape targets from this job every 5 seconds.
    scrape_interval: 5s

    static_configs:
      - targets: ['localhost:9090']
```

For a complete specification of configuration options, see the configuration documentation (../configuration/configuration/).

Starting Prometheus

To start Prometheus with your newly created configuration file, change to the directory containing the Prometheus binary and run:

```
# Start Prometheus.
# By default, Prometheus stores its database in ./data (flag --storage.tsdb.path).
./prometheus --config.file=prometheus.yml
```

Prometheus should start up. You should also be able to browse to a status page about itself at localhost:9090 (http://localhost:9090). Give it a couple of seconds to collect data about itself from its own HTTP metrics endpoint.

You can also verify that Prometheus is serving metrics about itself by navigating to its metrics endpoint: localhost:9090/metrics (http://localhost:9090/metrics)

Using the expression browser

Let us try looking at some data that Prometheus has collected about itself. To use Prometheus's built-in expression browser, navigate to http://localhost:9090/graph (http://localhost:9090/graph) and choose the "Console" view within the "Graph" tab.

As you can gather from localhost:9090/metrics (http://localhost:9090/metrics), one metric that Prometheus exports about itself is called `prometheus_target_interval_length_seconds` (the actual amount of time between target scrapes). Go ahead and enter this into the expression console:

```
prometheus_target_interval_length_seconds
```

This should return a number of different time series (along with the latest value recorded for each), all with the metric name `prometheus_target_interval_length_seconds` , but with different labels. These labels designate different latency percentiles and target group intervals.

If we were only interested in the 99th percentile latencies, we could use this query to retrieve that information:

```
prometheus_target_interval_length_seconds{quantile="0.99"}
```

To count the number of returned time series, you could write:

```
count(prometheus_target_interval_length_seconds)
```

For more about the expression language, see the expression language documentation (../querying/basics/).

Using the graphing interface

To graph expressions, navigate to http://localhost:9090/graph (http://localhost:9090/graph) and use the "Graph" tab.

For example, enter the following expression to graph the per-second rate of chunks being created in the self-scraped Prometheus:

```
rate(prometheus_tsdb_head_chunks_created_total[1m])
```

Experiment with the graph range parameters and other settings.

Starting up some sample targets

Let us make this more interesting and start some example targets for Prometheus to scrape.

The Go client library includes an example which exports fictional RPC latencies for three services with different latency distributions.

Ensure you have the Go compiler installed (https://golang.org/doc/install) and have a working Go build environment (https://golang.org/doc/code.html) (with correct `GOPATH`) set up.

Download the Go client library for Prometheus and run three of these example processes:

```
# Fetch the client library code and compile example.
git clone https://github.com/prometheus/client_golang.git
cd client_golang/examples/random
go get -d
go build

# Start 3 example targets in separate terminals:
./random -listen-address=:8080
./random -listen-address=:8081
./random -listen-address=:8082
```

You should now have example targets listening on http://localhost:8080/metrics (http://localhost:8080/metrics), http://localhost:8081/metrics (http://localhost:8081/metrics), and http://localhost:8082/metrics (http://localhost:8082/metrics).

Configuring Prometheus to monitor the sample targets

Now we will configure Prometheus to scrape these new targets. Let's group all three endpoints into one job called `example-random`. However, imagine that the first two endpoints are production targets, while the third one represents a canary instance. To model this in Prometheus, we can add several groups of endpoints to a single job, adding extra labels to each group of targets. In this example, we will add the `group="production"` label to the first group of targets, while adding `group="canary"` to the second.

To achieve this, add the following job definition to the `scrape_configs` section in your `prometheus.yml` and restart your Prometheus instance:

```
scrape_configs:
  - job_name:       'example-random'

    # Override the global default and scrape targets from this job every 5 seconds.
    scrape_interval: 5s

    static_configs:
      - targets: ['localhost:8080', 'localhost:8081']
        labels:
          group: 'production'

      - targets: ['localhost:8082']
        labels:
          group: 'canary'
```

Go to the expression browser and verify that Prometheus now has information about time series that these example endpoints expose, such as the `rpc_durations_seconds` metric.

Configure rules for aggregating scraped data into new time series

Though not a problem in our example, queries that aggregate over thousands of time series can get slow when computed ad-hoc. To make this more efficient, Prometheus allows you to prerecord expressions into completely new persisted time series via configured recording rules. Let's say we are interested in recording the per-second rate of example RPCs (`rpc_durations_seconds_count`) averaged over all instances (but preserving the `job` and `service` dimensions) as measured over a window of 5 minutes. We could write this as:

```
avg(rate(rpc_durations_seconds_count[5m])) by (job, service)
```

Try graphing this expression.

To record the time series resulting from this expression into a new metric called `job_service:rpc_durations_seconds_count:avg_rate5m`, create a file with the following recording rule and save it as `prometheus.rules.yml`:

```
groups:
- name: example
  rules:
  - record: job_service:rpc_durations_seconds_count:avg_rate5m
    expr: avg(rate(rpc_durations_seconds_count[5m])) by (job, service)
```

To make Prometheus pick up this new rule, add a `rule_files` statement in your `prometheus.yml`. The config should now look like this:

```
global:
  scrape_interval:     15s # By default, scrape targets every 15 seconds.
  evaluation_interval: 15s # Evaluate rules every 15 seconds.

  # Attach these extra labels to all timeseries collected by this Prometheus instance.
  external_labels:
    monitor: 'codelab-monitor'

rule_files:
  - 'prometheus.rules.yml'

scrape_configs:
  - job_name: 'prometheus'

    # Override the global default and scrape targets from this job every 5 seconds.
    scrape_interval: 5s

    static_configs:
      - targets: ['localhost:9090']

  - job_name:       'example-random'


    # Override the global default and scrape targets from this job every 5 seconds.
    scrape_interval: 5s

    static_configs:
      - targets: ['localhost:8080', 'localhost:8081']
        labels:
          group: 'production'

      - targets: ['localhost:8082']
        labels:
          group: 'canary'
```

Restart Prometheus with the new configuration and verify that a new time series with the metric name `job_service:rpc_durations_seconds_count:avg_rate5m` is now available by querying it through the expression browser or graphing it.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.