

The background of the entire image is a dark, deep red. Overlaid on this is a large, stylized flame graphic. The flame is composed of numerous overlapping, teardrop-shaped layers in various shades of red, ranging from a very dark, almost blackish-red at the edges to a bright, vibrant red in the center. The overall effect is a sense of depth and movement, reminiscent of a fire. The text is positioned in the upper half of the image, with the author's name at the top and the title below it.

JAMES TURNBULL

**MONITORING
WITH
PROMETHEUS**

Monitoring With Prometheus

James Turnbull

March 25, 2019

Version: v1.0.3 (d4b6456)

Website: [Monitoring With Prometheus](#)



Some rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical or photocopying, recording, or otherwise, for commercial purposes without the prior permission of the publisher.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit [here](https://creativecommons.org/licenses/by-nc-nd/3.0/).

© Copyright 2018 - James Turnbull <james@lovedthanlost.net>



Contents

	Page
Chapter 1 Installation and Getting Started	1
Installing Prometheus	2
Installing Prometheus on Linux	3
Installing Prometheus on Microsoft Windows	4
Alternative Microsoft Windows installation	5
Alternative Mac OS X installation	6
Stacks	7
Installing via configuration management	7
Deploying via Kubernetes	8
Configuring Prometheus	8
Global	10
Alerting	11
Rule files	12
Scrape configuration	12
Starting the server	14
Running Prometheus via Docker	15
First metrics	16
Prometheus expression browser	17
Time series aggregation	20
Capacity planning	24
Memory	24
Disk	25

Summary	26
List of Figures	27
List of Listings	28
Index	29


Chapter 1

Installation and Getting Started

In the last chapter we got an overview of Prometheus. In this chapter, we'll take you through the process of installing Prometheus on a variety of platforms. This chapter doesn't provide instructions for the full list of supported platforms, but a representative sampling to get you started. We'll look at installing Prometheus on:

- Linux.
- Microsoft Windows.
- Mac OS X.

The lessons here for installing Prometheus can be extended to other supported platforms.

 **NOTE** We've written the examples in this book assuming Prometheus is running on a Linux distribution. The examples should also work for Mac OS X but might need tweaking for Microsoft Windows.

We'll also explore the basics of Prometheus configuration and scrape our first target: the Prometheus server itself. We'll then use the metrics scraped to walk through the basics of the inbuilt expression browser and see how to use the Prometheus query language, PromQL, to glean interesting information from our metrics. This will give you a base Prometheus server that we'll build on in subsequent chapters.

Installing Prometheus

Prometheus is shipped as a single binary file. The [Prometheus download page](#) contains tarballs containing the binaries for specific platforms. Currently Prometheus is supported on:

- Linux: 32-bit, 64-bit, and ARM.
- Max OS X: 32-bit and 64-bit.
- FreeBSD: 32-bit, 64-bit, and ARM.
- OpenBSD: 32-bit, 64-bit, and ARM.
- NetBSD: 32-bit, 64-bit, and ARM.
- Microsoft Windows: 32-bit and 64-bit.
- DragonFly: 64-bit.

Older versions of Prometheus are available from the GitHub [Releases page](#).



NOTE At the time of writing, Prometheus was at version 2.3.2.

To get started, we're going to show you how to manually install Prometheus in the next few sections. At the end of this section we'll also provide some links to configuration management modules for installing Prometheus. If you're deploying

Prometheus into production or at scale you should always choose configuration management as the installation approach.

Installing Prometheus on Linux

To install Prometheus on a 64-bit Linux host, we first download the binary file. We can use `wget` or `curl` to get the file from the download site.

Listing 1.1: Download the Prometheus tarball

```
$ cd /tmp
$ wget https://github.com/prometheus/prometheus/releases/
download/v2.3.2/prometheus-2.3.2.linux-amd64.tar.gz
```

Now let's unpack the `prometheus` binary from the tarball and move it somewhere useful. We'll also install `promtool`, which is a linter for Prometheus configuration.

Listing 1.2: Unpack the prometheus binary

```
$ tar -xzf prometheus-2.3.2.linux-amd64.tar.gz
$ sudo cp prometheus-2.3.2.linux-amd64/prometheus /usr/local/bin/

$ sudo cp prometheus-2.3.2.linux-amd64/promtool /usr/local/bin/
```

We can test if Prometheus is installed and in our path by checking its version using the `--version` flag.

Listing 1.3: Checking the Prometheus version on Linux

```
$ prometheus --version
prometheus, version 2.3.2 (branch: HEAD, revision: 3569
eef8b1bc062bb5df43181b938277818f365b)
  build user:      root@bd4857492255
  build date:      20171006-22:16:15
  go version:      go1.9.1
```

Now that we have Prometheus installed, you can skip down to looking at its configuration, or you can continue to see how we install it on other platforms.

Installing Prometheus on Microsoft Windows

To install Prometheus on Microsoft Windows we need to download the [prometheus.exe](#) executable and put it in a directory. Let's create a directory for the executable using Powershell.

Listing 1.4: Creating a directory on Windows

```
C:\> MKDIR prometheus
C:\> CD prometheus
```

Now download Prometheus from the GitHub site:

Listing 1.5: Prometheus Windows download

```
https://github.com/prometheus/prometheus/releases/download/v2.3.2/prometheus-2.3.2.windows-amd64.tar.gz
```

Unzip the executable using a tool like [7-Zip](#) and put the contents of the unzipped directory into the `C:\prometheus` directory.

Finally, add the `C:\prometheus` directory to the path. This will allow Windows to find the executable. To do this, run this command inside Powershell.

Listing 1.6: Setting the Windows path

```
$env:Path += ";C:\prometheus"
```

You should now be able to run the `prometheus.exe` executable.

Listing 1.7: Checking the Prometheus version on Windows

```
C:\> prometheus.exe --version
prometheus, version 2.3.2 (branch: HEAD, revision: 3569
eef8b1bc062bb5df43181b938277818f365b)
  build user:      root@bd4857492255
  build date:      20171006-22:16:15
  go version:      go1.9.1
```

You can use something like [nssm, the Non-Sucking Service Manager](#), if you want to run the Prometheus server as a service.

Alternative Microsoft Windows installation

You can also use a package manager to install Prometheus on Windows. The [Chocolatey](#) package manager has a Prometheus package available. You can use [these instructions to install Chocolatey](#) and then use the `choco` binary to install Prometheus.

Listing 1.8: Installing Prometheus via Chocolatey

```
C:\> choco install prometheus
```

Alternative Mac OS X installation

In addition to being [available as a binary for Mac OS X](#), Prometheus is also available from [Homebrew](#). If you use Homebrew to provision your Mac OS X hosts then you can install Prometheus via the `brew` command.

Listing 1.9: Installing Prometheus via Homebrew

```
$ brew install prometheus
```

Homebrew will install the `prometheus` binary into the `/usr/local/bin` directory. We can test that it is operating via the `prometheus --version` command.

Listing 1.10: Checking the Prometheus version on Mac OS X

```
$ prometheus --version
prometheus, version 2.3.2 (branch: HEAD, revision: 3569
eef8b1bc062bb5df43181b938277818f365b)
  build user:      root@bd4857492255
  build date:      20171006-22:16:15
  go version:      go1.9.1
```

Stacks


In addition to installing Prometheus standalone, there are several prebuilt stacks available. These combine Prometheus with other tools—the Grafana console, for instance.

- A [Prometheus, Node Exporter, and Grafana docker-compose stack](#).
- Another [Docker Compose single-node stack](#) with Prometheus, Alertmanager, Node Exporter, and Grafana.
- A [Docker Swarm stack for Prometheus](#).

Installing via configuration management

There are also configuration management resources available for installing Prometheus. Here are some examples for a variety of configuration management tools:

- A [Puppet module for Prometheus](#).
- A [Chef cookbook for Prometheus](#).
- An [Ansible role for Prometheus](#).
- A [SaltStack formula for Prometheus](#).


 **TIP** Remember that configuration management is the recommended approach for installing and managing Prometheus!

Deploying via Kubernetes

Last, there are many ways to deploy Prometheus on Kubernetes. The best way for you to deploy likely depends greatly on your environment. You can build your own deployments and expose Prometheus via a service, use one of a number of [bundled configurations](#), or you can use the [Prometheus Operator from CoreOS](#).

Configuring Prometheus

Now that we have Prometheus installed let's look at [its configuration](#). Prometheus is configured via YAML configuration files. When we run the `prometheus` binary (or `prometheus.exe` executable on Windows), we specify a configuration file. Prometheus ships with a default configuration file: `prometheus.yml`. The file is in the directory we've just unpacked. Let's take a peek at it.

 **TIP** YAML configuration is fiddly and can be a real pain. You can validate YAML online at [YAML Lint](#) or from the command line with a tool like [this](#).


Listing 1.11: The default Prometheus configuration file

```
global:
  scrape_interval:     15s
  evaluation_interval: 15s

alerting:
  alertmanagers:
  - static_configs:
    - targets:
      # - alertmanager:9093

rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"

scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
```

 **NOTE** We've removed some comments from the file for brevity's sake. The default file changes from time to time, so yours might not look exactly like this one.

Our default configuration file has four YAML blocks defined: `global`, `alerting`, `rule_files`, and `scrape_configs`.

Let's look at each block.

Global

The first block, `global`, contains global settings for controlling the Prometheus server's behavior.

The first setting, the `scrape_interval` parameter, specifies the interval between scrapes of any application or service—in our case, 15 seconds. This value will be the resolution of your time series, the period in time that each data point in the series covers.

It is possible to override this global scrape interval when collecting metrics from specific places. *Do not do this.* Keep a single scrape interval globally across your server. This ensures that all your time series data has the same resolution and can be combined and calculated together. If you override the global scrape interval, you risk having incoherent results from trying to compare data collected at different intervals.

 **WARNING** Only configure scrape intervals globally and keep resolution consistent!

The `evaluation_interval` tells Prometheus how often to evaluate its rules. Rules come in two major flavors: recording rules and alerting rules:

- Recording rules - Allow you to precompute frequent and expensive expressions and to save their result as derived time series data.
- Alerting rules - Allow you to define alert conditions.

With this parameter, Prometheus will (re-)evaluate these rules every 15 seconds. We'll see more about rules in subsequent chapters.

 **NOTE** You can find the [full Prometheus configuration reference](#) in the documentation.


Alerting

The second block, `alerting`, configures Prometheus' alerting. As we mentioned in the last chapter, alerting is provided by a standalone tool called [Alertmanager](#). Alertmanager is an independent alert management tool that can be clustered.


Listing 1.12: Alertmanager configuration

```
alerting:
  alertmanagers:
    - static_configs:
      - targets:
        # - alertmanager:9093
```

In our default configuration, the `alerting` block contains the alerting configuration for our server. The `alertmanagers` block lists each Alertmanager used by this Prometheus server. The `static_configs` block indicates we're going to specify any Alertmanagers manually, which we have done in the `targets` array.

 **TIP** Prometheus also supports service discovery for Alertmanagers—for example, rather than specifying each Alertmanager individually, you could query an external source like a Consul server to return a list of available Alertmanagers. We'll see more about this in Chapters 5 and 6.

In our case we don't have an Alertmanager defined; instead we have a commented-out example at `alertmanager:9093`. We can leave this commented out because you don't specifically need an Alertmanager defined to run Prometheus. We'll add an Alertmanager and configure it in Chapter 6.

 **TIP** We'll see more about alerting in Chapter 6 and clustering alerting in Chapter 7.

Rule files

The third block, `rule_files`, specifies a list of files that can contain recording or alerting rules. We'll make some use of these in the next chapter.

Scrape configuration


The last block, `scrape_configs`, specifies all of the targets that Prometheus will scrape.

As we discovered in the last chapter, Prometheus calls the source of metrics it can scrape endpoints. To scrape an endpoint, Prometheus defines configuration called a target. This is the information required to perform the scrape—for example, what labels to apply, any authentication required to connect, or other information that defines how the scrape will occur. Groups of targets are called jobs. Inside jobs, each target has a label called `instance` that uniquely identifies it.


Listing 1.13: The default Prometheus scrape configuration

```
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
```

Our default configuration has one job defined called `prometheus`. Inside this job we have a `static_config` block, which lists the targets this job will scrape. The `static_config` block indicates that we're going to individually list the targets we want to scrape, rather than use any automated service discovery method. You can think about static configuration as manual or human service discovery.

 **TIP** We're going to look at methods to automatically discover targets to be scraped in Chapter 5.

The default `prometheus` job has one target: the Prometheus server itself. It scrapes `localhost` on port `9090`, which returns the server's own health metrics. Prometheus assumes that metrics will be returned on the path `/metrics`, so it appends this to the target and scrapes the address `http://localhost:9090/metrics`.

 **TIP** You can override the default metrics path.

Starting the server

Let's start the server and see what happens. First, though, let's move our configuration file somewhere more suitable.

Listing 1.14: Moving the configuration file

```
$ sudo mkdir -p /etc/prometheus
$ sudo cp prometheus.yml /etc/prometheus/
```

Here we've created a directory, `/etc/prometheus`, to hold our configuration file, and we've moved our new file into this directory.

Listing 1.15: Starting the Prometheus server

```
$ prometheus --config.file "/etc/prometheus/prometheus.yml"
level=info ts=2017-10-23T14:03:02.274562Z caller=main.go:216 msg
="Starting prometheus"...
```

We run the binary and specify our configuration file in the `--config.file` command line flag. Our Prometheus server is now running and scraping the instances of the `prometheus` job and returning the results.

If something doesn't work, you can validate your configuration with `promtool`, a linter that ships with Prometheus.

Listing 1.16: Validating your configuration with promtool

```
$ promtool check config prometheus.yml
Checking prometheus.yml
SUCCESS: 0 rule files found
```

Running Prometheus via Docker

It's also easy to run Prometheus in Docker. There's a [Docker image provided by the Prometheus team](#) available on the Docker Hub. You can execute it with the `docker` command.

Listing 1.17: Running Prometheus with Docker


```
$ docker run -p 9090:9090 prom/prometheus
```

This will run a Prometheus server locally, with port `9090` bound to port `9090` inside the Docker container. You can then browse to that port on your local host to see your Prometheus server. The server is launched with a default configuration, and you will need to provide custom configuration and data storage. You can take a number of approaches here—for example, you could mount a configuration file into the container.

Listing 1.18: Mounting a configuration file into the Docker container


```
$ docker run -p 9090:9090 -v /tmp/prometheus.yml:/etc/prometheus/prometheus.yml prom/prometheus
```

This would bind mount the file `/tmp/prometheus.yml` into the container as the Prometheus server's configuration file.

 **TIP** You can find more information on running [Prometheus with Docker](#) in the documentation.

First metrics

Now that the server is running, let's take a look at the endpoint we are scraping and see some raw Prometheus metrics. To do this, let's browse to the URL <http://localhost:9090/metrics> and see what gets returned.

 **NOTE** In all our examples we assume you're browsing on the server running Prometheus, hence `localhost`.

Listing 1.19: Some sample raw metrics

```
# HELP go_gc_duration_seconds A summary of the GC invocation
durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 1.6166e-05
go_gc_duration_seconds{quantile="0.25"} 3.8655e-05
go_gc_duration_seconds{quantile="0.5"} 5.3416e-05
. . .
```

Here we can see our first Prometheus metrics. These look much like the data model we saw in the last chapter.

Listing 1.20: A raw metric

```
go_gc_duration_seconds{quantile="0.5"} 1.6166e-05
```

The name of our metric is `go_gc_duration_seconds`. We can see one label on the metric, `quantile="0.5"`, indicating this is measuring the 50th percentile, and the

value of the metric.

Prometheus expression browser

It is not user friendly to view our metrics this way, though, so let's make use of Prometheus' inbuilt expression browser. It's available on the Prometheus server by browsing to <http://localhost:9090/graph>.

TIP The Prometheus Expression browser and web interface have other useful information, like the status of targets and the rules and configuration of the Prometheus server. Make sure you check out all the interface menu items.



Figure 1.1: Prometheus expression browser

Let's find the `go_gc_duration_seconds` metric using the expression browser. To

do this, we can either open the dropdown list of available metrics or we can type the metric name into the query box. We then click the **Execute** button to display all the metrics with this name.

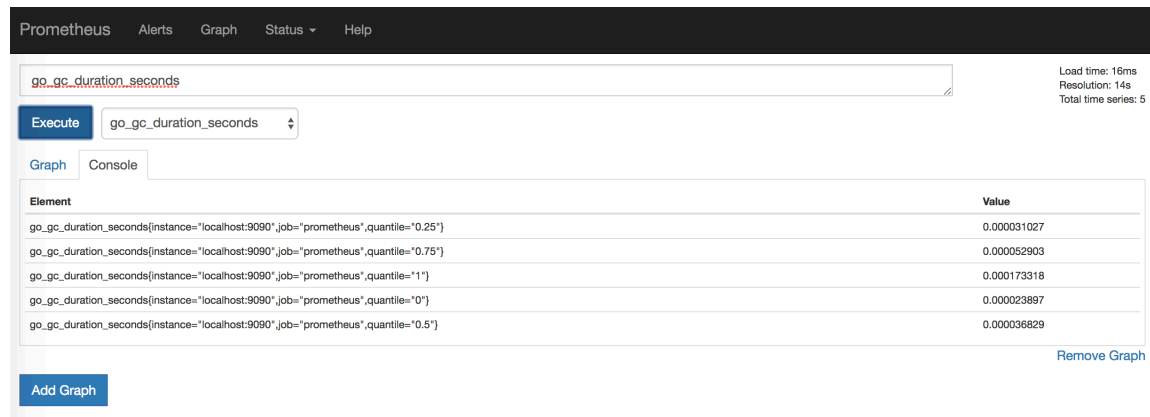


Figure 1.2: List of metrics

We can see a list of metrics here, each decorated with one or more labels. Let's find the 50th percentile in the list.

Listing 1.21: Go garbage collection 50th percentile

```
go_gc_duration_seconds{instance="localhost:9090",job="prometheus",quantile="0.5"}
```

We can see that two new labels have been added to our metrics. This has been done automatically by Prometheus during the scrape process. The first new label, **instance**, is the target from which we scraped the metrics. The second label, **job**, is the name of the job that scraped the metrics. Labels provide dimensions to our metrics. They allow us to query or work with multiple or specific metrics—for example, Go garbage collection metrics for multiple targets.

TIP We'll see a lot more about labels in the next chapter and later in the book.

Prometheus has a [highly flexible expression language](#) called PromQL built into the server, allowing you to query and aggregate metrics. We can use this query language in the query input box at the top of the interface.

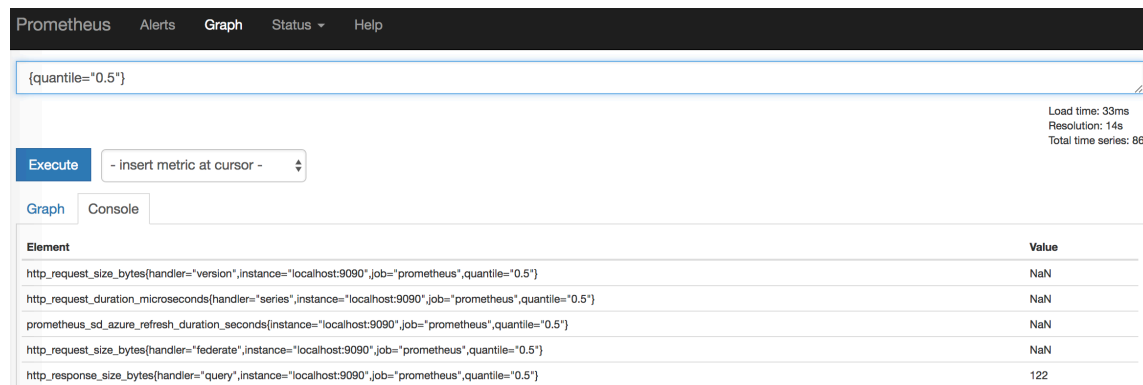


Figure 1.3: Querying quantiles


Here we've queried all metrics with a label of `quantile="0.5"` and it has returned a possible 86 metrics. This set is one of the [four data types](#) that expressions in the PromQL querying language can return. This type is called an instant vector: a set of time series containing a single sample for each time series, all sharing the same timestamp. We can also return instant vectors for metrics by querying a name and a label. Let's go back to our `go_gc_duration_seconds` but this time the 75th percentile. Specify:

```
go_gc_duration_seconds{quantile="0.75"}
```

In the input box and click `Execute` to search. It should return an instant vector that matches the query. We can also negate or match a label using a regular expression.

```
go_gc_duration_seconds{quantile!="0.75"}
```


This will return an instant vector of all the metrics with a `quantile` label not equal to `0.75`.

 **TIP** If we're used to tools like Graphite, querying labels is like parsing dotted-string named metrics. There's a [blog post that provides a side-by-side comparison of how Graphite, InfluxDB, and Prometheus handle a variety of queries](#).

Let's look at another metric, this one called `prometheus_build_info`, that contains information about the Prometheus server's build. Put `prometheus_build_info` into the expression browser's query box and click `Execute` to return the metric. You'll see an entry like so:

Listing 1.22: The `prometheus_build_info` metric

```
prometheus_build_info{branch="HEAD",goversion="go1.9.1",instance="localhost:9090",job="prometheus",revision="5ab8834befbd92241a88976c790ace7543edcd59",version="2.3.2"}
```

You can see the metric is heavily decorated with labels and has a value of `1`. This is a common pattern for passing information to the Prometheus server using a metric. It uses a metric with a perpetual value of `1`, and with the relevant information you might want attached via labels. We'll see more of these types of informational metrics later in the book.

Time series aggregation

The interface can also do complex aggregation of metrics. Let's choose another metric, `promhttp_metric_handler_requests_total`, which is the total HTTP re-

quests made by scrapes in the Prometheus server. Query for that now by specifying its name and clicking **Execute**.



Figure 1.4: Querying total HTTP requests

We have a list of HTTP request metrics. But what we really want is the total HTTP requests per job. To do this, we need to create a new metric via a query. Prometheus' querying language, PromQL, has a large collection of **expressions and functions** that can help us do this.

Let's start by summing the HTTP requests by job. Add the following to the query box and click **Execute**.

```
sum(promhttp_metric_handler_requests_total)
```

This new query uses the **sum()** operator on the `promhttp_metric_handler_requests_total` metric. It adds up all of the requests but doesn't break it down by job. To do that we need to aggregate over a specific label dimension. PromQL has a clause called **by** that will allow us to aggregate by a specific dimension. Add the following to the query box and then click **Execute**.

```
sum(promhttp_metric_handler_requests_total) by (job)
```

TIP PromQL also has a clause called **without** that aggregates without a specific dimension.

You should see something like the following output:

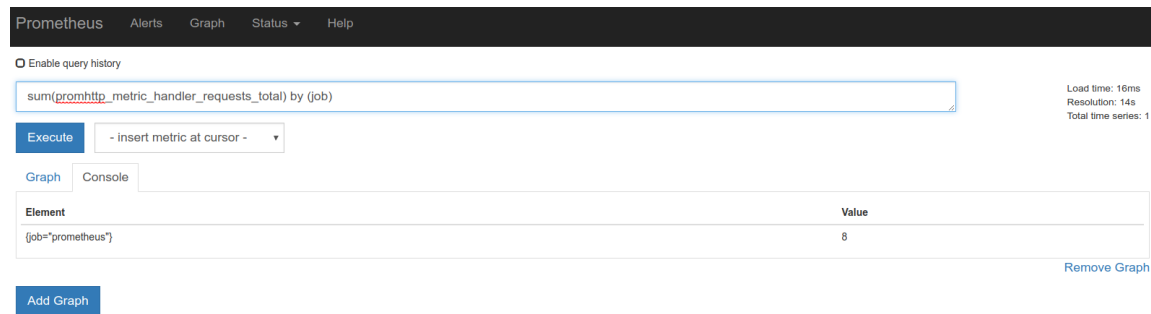



Figure 1.5: Calculating total HTTP requests by job

Now click the **Graph** tab to see this metric represented as a plot.

 **TIP** The folks at Robust Perception have a [great blog post on common query-ing patterns](#).

The new output is still not quite useful—let’s convert it into a rate. Update our query to:


```
sum(rate(promhttp_metric_handler_requests_total[5m])) by (job)
```

Here we’ve added a new function: `rate()`. We’ve inserted it **inside our sum function**.

```
rate(promhttp_metric_handler_requests_total[5m])
```

The `rate()` function calculates the per-second average rate of increase of the time series in a range. The `rate` function should only be used with counters. It is quite clever and automatically adjusts for breaks, like a counter being reset when the resource is restarted, and extrapolates to take care of gaps in the time series, such


as a missed scrape. The `rate()` function is best used for slower-moving counters or for alerting purposes.

 **TIP** There's also an `irate()` function to calculate the instant rate of increase for faster-moving timers.

Here we're calculating the rate over a five-minute range vector. [Range vectors](#) are a second PromQL data type containing a set of time series with a range of data points over time for each time series. Range vectors allow us to display the time series for that period. The duration of the range is enclosed in `[]` and has an integer value followed by a unit abbreviation:

- `s` for seconds.
- `m` for minutes.
- `h` for hours.
- `d` for days.
- `w` for weeks.
- `y` for years.

So here `[5m]` is a five-minute range.

 **TIP** The other two PromQL data types are [Scalars](#), numeric floating-point values, and [Strings](#), which is a string value and is currently unused.

Let's [Execute](#) that query and see the resulting range vector of time series.

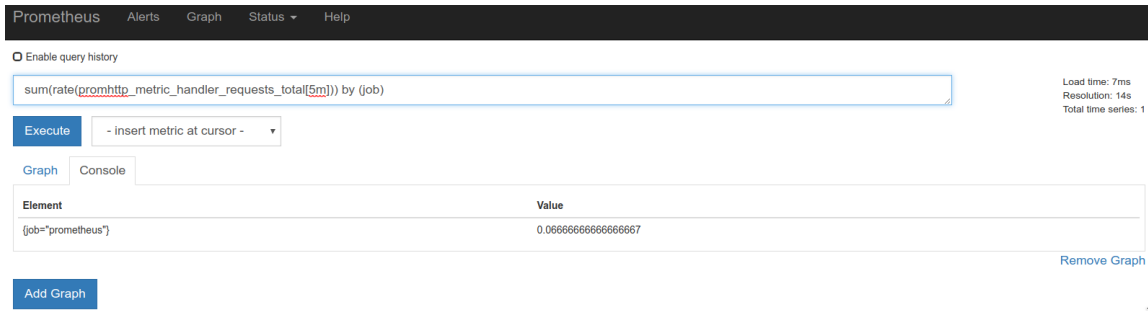



Figure 1.6: Our rate query

Cool! We’ve now got a new metric that is actually useful for tracking or graphing. Now that we’ve walked through the basics of Prometheus operation, let’s look at some of the requirements for running a Prometheus server.

Capacity planning

Prometheus performance is hard to estimate because it depends greatly on your configuration, the volume of time series you collect, and the complexity of any rules on the server. There are two capacity concerns: memory and disk.

 **TIP** We’ll look at Prometheus scaling concepts in Chapter 7.

Memory

Prometheus does a lot in memory. It consumes process memory for each time series collected and for querying, recording rules, and the like. There’s not a lot of data on capacity planning for Prometheus, especially since 2.0 was released, but a good, rough, rule of thumb is to multiply the number of samples being collected

per second by the size of the samples. We can see the rate of sample collection using this query.

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

This will show you the per-second rate of samples being added to the database over the last minute.

If you want to know the number of metrics you're collecting you can use:

```
sum(count by (__name__)( {__name__=~"\.\\.+"} ))
```

This uses the `sum` aggregation to add up a count of all metrics that match, using the `==` operator, the regular expression of `.+`, or all metrics.

Each sample is generally one to two bytes in size. Let's err on the side of caution and use two bytes. Assuming we're collecting 100,000 samples per second for 12 hours, we can work out memory usage like so:

```
100,000 * 2 bytes * 43200 seconds
```

Or roughly 8.64 GB of RAM.


You'll also need to factor in memory use for querying and recording rules. This is very rough and dependent on a lot of other variables. I recommend playing things by ear with regard to memory usage. You can see the memory usage of the Prometheus process by checking the `process_resident_memory_bytes` metric.

Disk

Disk usage is bound by the volume of time series stored and the retention of those time series. By default, metrics are stored for 15 days in the local time series database. The location of the database and the retention period are controlled by command line options.

- The `--storage.tsdb.path` option, which has a default directory of `data` located in the directory from which you are running Prometheus, controls your time series database location.

- The `--storage.tsdb.retention` controls retention of time series. The default is `15d` representing 15 days.

 **TIP** The best disk for time series databases is SSD. You should use [SSDs](#).

For our 100,000 samples per second example, we know each sample collected in a time series occupies about one to two bytes on disk. Assuming two bytes per sample, then a time series retained for 15 days would mean needing about 259 GB of disk.

 **TIP** There's more information on Prometheus disk usage in [the Storage documentation](#).

Summary

In this chapter we installed Prometheus and configured its basic operation. We also scraped our first target, the Prometheus server itself. We made use of the metrics collected by the scrape to see how the inbuilt expression browser works, including graphing our metrics and deriving new metrics using Prometheus's query language, PromQL.

In the next chapter we'll use Prometheus to collect some host metrics, including collecting from Docker containers. We'll also see a lot more about scraping, jobs, and labels, and we'll have our first introduction to recording rules.

List of Figures

1.1 Prometheus expression browser	17
1.2 List of metrics	18
1.3 Querying quantiles	19
1.4 Querying total HTTP requests	21
1.5 Calculating total HTTP requests by job	22
1.6 Our rate query	24

Listings

1.1 Download the Prometheus tarball	3
1.2 Unpack the prometheus binary	3
1.3 Checking the Prometheus version on Linux	4
1.4 Creating a directory on Windows	4
1.5 Prometheus Windows download	4
1.6 Setting the Windows path	5
1.7 Checking the Prometheus version on Windows	5
1.8 Installing Prometheus via Chocolatey	6
1.9 Installing Prometheus via Homebrew	6
1.10 Checking the Prometheus version on Mac OS X	6
1.11 The default Prometheus configuration file	9
1.12 Alertmanager configuration	11
1.13 The default Prometheus scrape configuration	13
1.14 Moving the configuration file	14
1.15 Starting the Prometheus server	14
1.16 Validating your configuration with promtool	14
1.17 Running Prometheus with Docker	15
1.18 Mounting a configuration file into the Docker container	15
1.19 Some sample raw metrics	16
1.20 A raw metric	16
1.21 Go garbage collection 50th percentile	18
1.22 The prometheus_build_info metric	20

Index

- Aggregation, 21
- Alerting, 11
- Alerting rules, 10
- Alertmanager, 11
- Ansible, 7
- Capacity planning, 24
- Chef, 7
- Chocolatey, 5
- Configuration, 8, 14
- Configuration Management, 3, 7
- Endpoints, 12
- Expression browser, 17
- global
 - evaluation_interval, 10
 - scrape_interval, 10
- Granularity, 10
- Homebrew, 6
- Installation, 2
 - Linux, 3
 - Mac OS X, 6
 - Microsoft Windows, 4, 5
 - Windows, 4
- Installing onto Kubernetes, 8
- Installing via configuration management, 7
- Instances, 12
- Job definition, 13
- job_name, 13
- Jobs, 12
- Kubernetes, 8
- Prometheus
 - configuration, 8
 - disk usage, 24
 - installation
 - Linux, 3
 - OS X, 6
 - Windows, 4
 - memory usage, 24
 - Web interface, 17
- prometheus
 - config.file, 14
 - version, 3
- prometheus.yml, 8

- PromQL, 19, 21
 - by, 21
 - irate, 23
 - Range vectors, 23
 - rate, 22
 - Scalar, 23
 - String, 23
 - without, 21
- promtool, 3, 8, 14
- Puppet, 7
- Querying labels, 19
- Range vectors, 23
- Recording rules, 10
- Resolution, 10
- Rule files, 12
- rule_files, 12
- Rules, 10
- SaltStack, 7
- Scrape configuration, 12
- Scrape interval, 10
- scrape_configs, 12
- SSD, 26
- Supported platforms, 2
- Targets, 12
- YAML, 8
- YAML validation, 8

Thanks! I hope you enjoyed the book.

© Copyright 2018 - James Turnbull <james@lovedthanlost.net>

