



This book is provided in digital form with the permission of the rightsholder as part of a Google project to make the world's books discoverable online.

The rightsholder has graciously given you the freedom to download all pages of this book. No additional commercial or other uses have been granted.

Please note that all copyrights remain reserved.

About Google Books

Google's mission is to organize the world's information and to make it universally accessible and useful. Google Books helps readers discover the world's books while helping authors and publishers reach new audiences. You can search through the full text of this book on the web at <http://books.google.com/>

JAMES TURNBULL

MONITORING
WITH
PROMETHEUS

Monitoring With Prometheus

James Turnbull

June 23, 2018

Version: v1.0.2 (c174afc)

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](#).

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

ISBN 978-0-9888202-8-9

A standard one-dimensional barcode representing the ISBN 9780988820289.

9 780988 820289

Contents

	Page
Foreword	1
Who is this book for?	1
Credits and Acknowledgments	1
Technical Reviewers	2
Jamie Wilkinson	2
Paul Gier	2
Editor	2
Author	3
Conventions in the book	3
Code and Examples	4
Colophon	4
Errata	4
Disclaimer	4
Copyright	5
Version	5
Chapter 1 Introduction	6
What is monitoring?	6
Technology as a customer	7
The business as a customer	8
Monitoring fundamentals	8
Monitoring as afterthought	9

Monitoring by rote	10
Not monitoring for correctness	12
Monitoring statically	12
Not monitoring frequently enough	13
No automation or self-service	14
Good monitoring summary	14
Monitoring mechanics	15
Probing and introspection	16
Pull versus push	17
Types of monitoring data	17
Metrics	18
So what's a metric?	19
Types of metrics	21
Metric summaries	23
Metric aggregation	24
Monitoring methodologies	36
The USE Method	36
The Google Four Golden Signals	38
Contextual, useful alerts and notifications	39
Visualization	41
But didn't you write that other book?	42
What's in the book?	43
Summary	44
Chapter 2 Introduction to Prometheus	46
The Prometheus backstory	47
Prometheus architecture	48
Metric collection	49
Service discovery	50
Aggregation and alerting	50
Querying data	51

Autonomy	52
Redundancy and high availability	53
Visualization	54
The Prometheus data model	54
Metric names	54
Labels	55
Samples	56
Notation	56
Metrics retention	57
Security model	57
Prometheus ecosystem	58
Useful Prometheus links	58
Summary	59
Chapter 3 Installation and Getting Started	60
Installing Prometheus	61
Installing Prometheus on Linux	62
Installing Prometheus on Microsoft Windows	63
Alternative Microsoft Windows installation	64
Alternative Mac OS X installation	65
Stacks	66
Installing via configuration management	66
Deploying via Kubernetes	67
Configuring Prometheus	67
Global	69
Alerting	70
Rule files	71
Scrape configuration	71
Starting the server	73
Running Prometheus via Docker	74
First metrics	75

Prometheus expression browser	76
Time series aggregation	79
Capacity planning	83
Memory	84
Disk	85
Summary	86
Chapter 4 Monitoring Nodes and Containers	87
Monitoring nodes	88
Installing the Node Exporter	89
Configuring the Node Exporter	90
Configuring the Textfile collector	91
Enabling the systemd collector	93
Running the Node Exporter	93
Scraping the Node Exporter	94
Filtering collectors on the server	95
Monitoring Docker	97
Running cAdvisor	97
Scraping cAdvisor	100
Scrape lifecycle	101
Labels	104
Label taxonomies	105
Relabelling	107
The Node Exporter and cAdvisor metrics	112
The trinity and the USE method	113
Service status	124
Availability and the up metric	126
The metadata metric	128
Query permanence	131
Recording rules	131
Configuring recording rules	132

Adding recording rules	133
Visualization	138
Installing Grafana	138
Starting and configuring Grafana	142
Configuring the Grafana web interface	144
First dashboard	149
Summary	150
Chapter 5 Service Discovery	151
Scrape lifecycle and static configuration redux	152
File-based discovery	154
Writing files for file discovery	157
Inbuilt service discovery plugins	159
Amazon EC2 service discovery plugin	159
DNS service discovery	165
Summary	168
Chapter 6 Alerting and Alertmanager	170
Alerting	171
How the Alertmanager works	173
Installing Alertmanager	175
Installing Alertmanager on Linux	175
Installing Alertmanager on Microsoft Windows	177
Stacks	178
Installing via configuration management	179
Configuring the Alertmanager	179
Running Alertmanager	183
Configuring Prometheus for Alertmanager	184
Alertmanager service discovery	185
Monitoring Alertmanager	187
Adding alerting rules	187

Adding our first alerting rule	188
What happens when an alert fires?	192
The alert at the Alertmanager	193
Adding new alerts and templates	195
Routing	201
Routes	203
Receivers and notification templates	206
Silences and maintenance	209
Controlling silences via the Alertmanager	210
Controlling silences via amtool	213
Summary	216
Chapter 7 Scaling and Reliability	217
Reliability and fault tolerance	218
Duplicate Prometheus servers	220
Setting up Alertmanager clustering	220
Configuring Prometheus for an Alertmanager cluster	224
Scaling	226
Functional scaling	227
Horizontal shards	229
Remote storage	236
Third-party tools	237
Summary	237
Chapter 8 Instrumenting Applications	238
An application monitoring primer	238
Where should I instrument?	240
Instrument taxonomies	240
Metrics	240
Application metrics	241
Business metrics	241

Where to put your metrics	242
The utility pattern	242
The external pattern	244
Building metrics into a sample application	244
Summary	255
Chapter 9 Logging as Instrumentation	256
Processing logs for metrics	257
Introducing mtail	257
Installing mtail	258
Using mtail	259
Running mtail	261
Processing web server access logs	264
Parsing Rails logs into a histogram	268
Deploying mtail	271
Scraping our mtail endpoint	271
Summary	272
Chapter 10 Probing	274
Probing architecture	274
The blackbox exporter	276
Installing the exporter	276
Installing the exporter on Linux	277
Installing the exporter on Microsoft Windows	278
Installing via configuration management	279
Configuring the exporter	280
HTTP check	281
ICMP check	282
DNS check	282
Starting the exporter	283
Creating the Prometheus job	285

Summary	288
Chapter 11 Pushing Metrics and the Pushgateway	290
The Pushgateway	291
When not to use the Pushgateway	292
Installing the Pushgateway	293
Installing the Pushgateway on Linux	294
Installing the Pushgateway on Microsoft Windows	295
Installing via configuration management	297
Configuring and running the Pushgateway	297
Sending metrics to the Pushgateway	299
Viewing metrics on the Pushgateway	302
Deleting metrics in the Pushgateway	304
Sending metrics from a client	305
Summary	310
Chapter 12 Monitoring a Stack - Kubernetes	311
Our Kubernetes cluster	311
Running Prometheus on Kubernetes	312
Monitoring Kubernetes	313
Monitoring our Kubernetes nodes	314
Node Exporter DaemonSet	314
Node Exporter service	318
Deploying the Node Exporter	320
The Node Exporter job	322
Node Explorer rules	326
Kubernetes	328
Kube-state-metrics	328
Kube API	332
CAdvisor and Nodes	336
Summary	338

Chapter 13 Monitoring a Stack - Tornado	339
Sidecar pattern	340
MySQL	342
MySQL Prometheus configuration	345
Redis	349
Redis Prometheus configuration	351
Tornado	353
Adding the Clojure wrapper	353
Adding a registry	355
Adding metrics	356
Exporting the metrics	357
Tornado Prometheus configuration	358
Summary	359
List of Figures	364
List of Listings	373
Index	374

Foreword

Who is this book for?

This book is a hands-on introduction to monitoring with Prometheus.

Most of the book's examples are Ubuntu Linux-based, and there is an expectation that the reader has basic Unix/Linux skills and is familiar with the command line, editing files, installing packages, managing services, and basic networking.

Finally, Prometheus is evolving quickly. That means "Here Be Dragons," and you should take care to confirm what versions you're using of the tools in this book.

The book is designed to be used with Prometheus version 2.3.1 and later. This Material will not work with earlier releases.

Credits and Acknowledgments

- Ruth Brown, who continues to humor these books and my constant tap-tap-tap of keys late into the night.
- Sid Orlando, who makes my words good.
- Bryan Brazil for his excellent Prometheus blog. He also runs training that you should check out.
- David Karlsen for his technical review work.

Technical Reviewers

Thanks to the folks who helped make this book more accurate and useful!

Jamie Wilkinson

Jamie is a Site Reliability Engineer in Google’s Storage Infrastructure team. He began in Linux systems administration in 1999, while earning a Bachelor’s in Computer Science, so knows just enough theory of computation to be dangerous in his field. He contributed a chapter on monitoring to the Google SRE Book. Jamie lives with his family in Sydney, Australia.

Paul Gier

As a curious kid growing up at a time when proprietary software was the rule, Paul was frustrated by a lack of money and licenses. Soon after learning about a new operating system called Linux, Paul was hooked on the ideas of free software—ideas that eventually led him to his current role as a Principal Software Engineer at Red Hat, where he has been happily developing free software for more than 10 years. Paul is excited about new container-based infrastructures and all the solutions and problems they bring. He lives in Austin, Texas, with his wife, three children, two dogs, and one mischievous cat.

Editor

Sid Orlando is an editor and writer, among some other things. She’s currently making Increment, Stripe’s software engineering/tech magazine, while drawing lots of friendly monsters and raising a giant army of plants in her NYC apartment.

Author

James is an author and engineer. His most recent books are *The Packer Book*; *The Terraform Book*; *The Art of Monitoring*; *The Docker Book*, about the open-source container virtualization technology; and *The Logstash Book*, about the popular open-source logging tool. James also authored two books about Puppet, *Pro Puppet* and *Pulling Strings with Puppet*. He is the author of three other books: *Pro Linux System Administration*, *Pro Nagios 2.0*, and *Hardening Linux*.

He is currently CTO at Empatico and was formerly CTO at Kickstarter, VP of Services and Support at Docker, VP of Engineering at Venmo, and VP of Technical Operations at Puppet. He likes food, wine, books, photography, and cats. He is not overly keen on long walks on the beach or holding hands.

Conventions in the book

This is an inline code statement.

This is a code block:

Listing 1: Sample code block

This is a code block

Long code strings are broken. If you see . . . in a code block it indicates that the output has been shortened for brevity's sake.

Code and Examples

The code and example configurations contained in the book are available on GitHub at:

<https://github.com/turnbullpress/prometheusbook-code>

Colophon

This book was written in Markdown with a large dollop of LaTeX. It was then converted to PDF and other formats using PanDoc (with some help from scripts written by the excellent folks who wrote Backbone.js on Rails).

Errata

Please email any errata you find to james+errata@lovedthanlost.net.

Disclaimer

This book is presented solely for educational purposes. The author is not offering it as legal, accounting, or other professional services advice. While best efforts have been used in preparing this book, the author makes no representations or warranties of any kind and assumes no liabilities of any kind with respect to the accuracy or completeness of the contents and specifically disclaims any implied warranties of merchantability or fitness of use for a particular purpose. The author shall not be held liable or responsible to any person or entity with respect to any loss or incidental or consequential damages caused, or alleged to have been caused, directly or indirectly, by the information or programs contained herein. Every company is different and the advice and strategies contained herein may

not be suitable for your situation. You should seek the services of a competent professional before beginning any infrastructure project.

Copyright

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.



Figure 1: License

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit [here](#).

© Copyright 2018 - James Turnbull & Turnbull Press



Figure 2: ISBN

Version

This is version v1.0.2 (c174afc) of Monitoring with Prometheus.

Chapter 1

Introduction

This book is an introduction to Prometheus, an open-source monitoring system. Prometheus provides real-time collection of time series data from your applications backed by a powerful rules engine to help identify the information you need to monitor your environment. In the next chapter we'll introduce you to Prometheus and its architecture and components. We'll use Prometheus in the book to take you through building a monitoring environment, with a focus on monitoring dynamic cloud, Kubernetes, and container environments. We'll also look at instrumenting applications and using that data for alerting and visualization.

This is also a book about monitoring in general—so, before we introduce you to Prometheus, we're going to take you through some monitoring basics. We'll go through what monitoring is, some approaches to monitoring, and we'll explain some terms and concepts that we'll rely on later in this book.

What is monitoring?

From a technology perspective, **monitoring** is the tools and processes by which you measure and manage your technology systems. But monitoring is much more

than that. Monitoring provides the translation to business value from metrics generated by your systems and applications. Your monitoring system translates those metrics into a measure of user experience. That measure provides feedback to the business to help ensure it's delivering what customers want. The measure also provides feedback to technology, as we'll define below, to indicate what isn't working and what's delivering an insufficient quality of service.

A monitoring system has two customers:

- Technology
- The business

Technology as a customer

The first customer of your monitoring system is Technology. That's you, your team, and the other folks who manage and maintain your technology environment (you might also be called Engineering or Operations or DevOps or Site Reliability Engineering). You rely on monitoring to let you know the state of your technology environment. You also use monitoring quite heavily to detect, diagnose, and help resolve faults and other issues in your technology environment, preferably before it impacts your users. Monitoring contributes much of the data that informs your critical product and technology decisions, and measures the success of those projects. It's a foundation of your product management life cycle and your relationship with your internal customers, and it helps demonstrate that the business's money is being well spent. Without monitoring you're winging it at best—and at worst being negligent.

 **NOTE** There's a great diagram from Google's SRE book that shows how monitoring is the foundation of the hierarchy of building and managing applications.

The business as a customer

The business is the second customer of your monitoring. Your monitoring exists to support the business—and to make sure it continues to do business. Monitoring provides the reporting that allows the business to make good product and technology investments. Monitoring also helps the business measure the value that technology delivers.

Monitoring fundamentals

Monitoring should be a core tool for managing infrastructure and your business. Monitoring should also be mandatory, built, and deployed with your applications. Without it you will not be able to understand the state of your world, readily diagnose problems, capacity plan, or provide information to your organization about performance, costs, or status.

An excellent exposition of this foundation is the Google service hierarchy chart we mentioned earlier.¹

¹Site Reliability Engineering, edited by Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy (O'Reilly). Copyright 2016 Google, Inc., 978-1-491-92912-4.

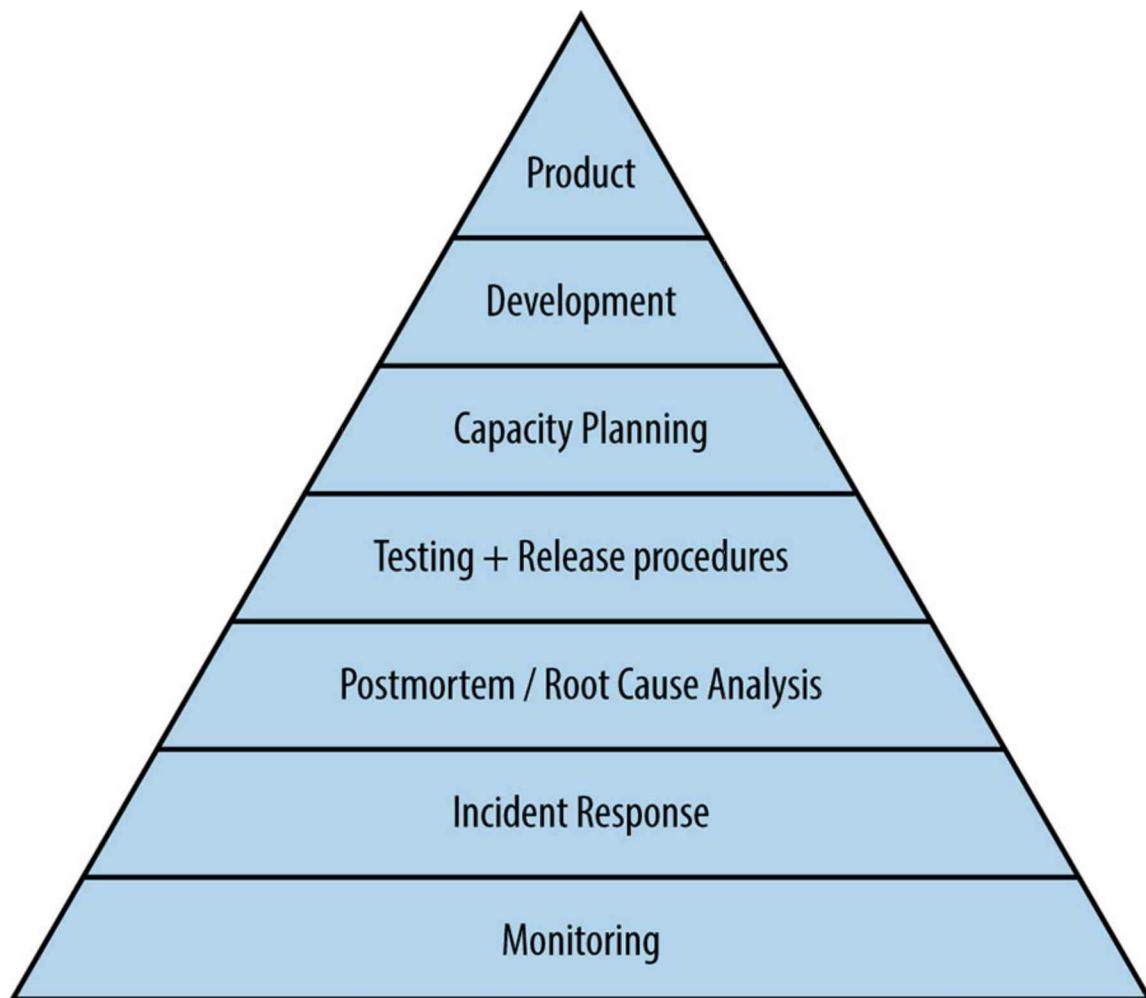


Figure 1.1: Service hierarchy

But monitoring can be hard to implement well, and it can very easily be bad if you're monitoring the wrong things or in the wrong way. There are some key monitoring anti-patterns and mitigation:

Monitoring as afterthought

In any good application development methodology, it's a good idea to identify what you want to build before you build it. Sadly, there's a common anti-pattern

of considering monitoring, and other operational functions like security, as value-add components of your application rather than core features. Monitoring, like security, is a core feature of your applications. If you’re building a specification or user stories for your application, include metrics and monitoring for each component of your application. Don’t wait until the end of a project or just before deployment. I guarantee you’ll miss something that needs to be monitored.

 **TIP** See the discussion about automation and self-service below for ideas on how to make this process easier.

Monitoring by rote

Many environments create cargo cult monitoring checks for all your applications. A team reuses the checks they have built in the past rather than evolving those checks for a new system or application. A common example is to monitor CPU, memory, and disk on every host, but not the key services that indicate the application that runs on the host is functional. If an application can go down without you noticing, even with monitoring in place, then you need to reconsider what you are monitoring.

A good approach to your monitoring is to design a top-down monitoring plan based on value. Identify the parts of the application that deliver value and monitor those first, working your way down the stack.

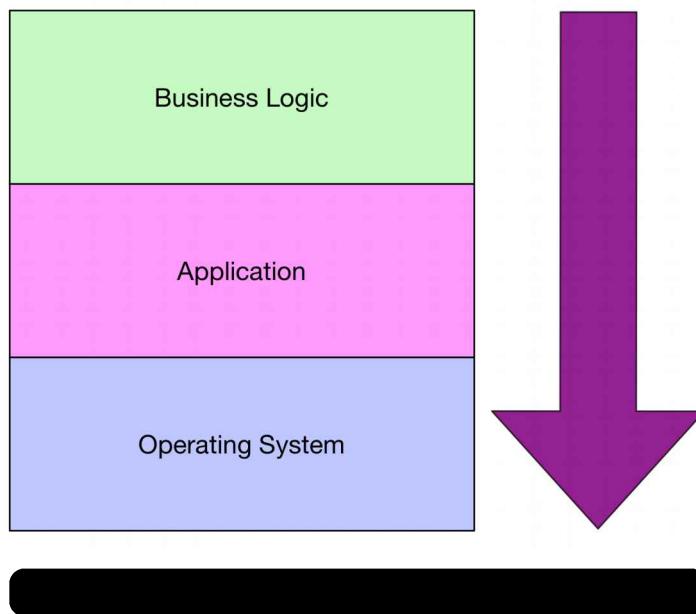


Figure 1.2: Monitoring design

Start with business logic and business outputs, move down to application logic, and finally into infrastructure. This doesn't mean you shouldn't collect infrastructure or operating system metrics—they provide value in diagnostics and capacity planning—but you're unlikely to need them to report the value of your applications.



NOTE If you can't start with business metrics, then start monitoring close

to the user. They are the ultimate customer and their experience is what drives your business. Understanding what their experience is and detecting when they have issues is valuable in its own right.

Not monitoring for correctness

Another common variant of this anti-pattern is to monitor the status of services on a host but not the correctness. For example, you may monitor if a web application is running by checking for an HTTP 200 response code. This tells you the application is responding to connections, but not if it's returning the correct data in response to those requests.

A better approach is monitoring for the correctness of a service first—for example, monitor the content or rates of a business transaction rather than the uptime of the web server it runs on. This allows you to get the value of both: if the content of a service isn't correct because it is misconfigured, buggy, or broken you'll see that. If the content isn't correct because underlying web service goes down, you'll also know that.

Monitoring statically

A further check anti-pattern is the use of static thresholds—for example, alerting if CPU usage on a host exceeds 80 percent. Checks are often inflexible Boolean logic or arbitrary static in time thresholds. They generally rely on a specific result or range being matched. The checks don't consider the dynamism of most complex systems. A match or a breach in a threshold may be important or could have been triggered by an exceptional event—or could even be a natural consequence of growth.

Arbitrary static thresholds are almost always wrong. Baron Schwartz, CEO of

database performance analysis vendor VividCortex, put it well:

They're worse than a broken clock, which is at least right twice a day. A threshold is wrong for any given system, because all systems are slightly different, and it's wrong for any given moment during the day, because systems experience constantly changing load and other circumstances.

To monitor well we need to look at windows of data, not static points in time, and we need to use smarter techniques to calculate values and thresholds.

Not monitoring frequently enough

In many monitoring tools, scaling is a challenge or the default check period is set to a high value—for example, only checking an application once every five to 15 minutes. This often results in missing critical events that occur between your checks. You should monitor your applications frequently enough to:

- Identify faults or anomalies.
- Meet human response time expectations—you want to find the fault before your users report the fault.
- Provide data at sufficient granularity for identifying performance issues and trends.

Always remember to store sufficient historical data to identify performance issues and trends. In many cases this might only need to be days or weeks of data—but it's impossible to identify a trend or reoccurring problem if you have thrown away the data that shows it.

No automation or self-service

A frequent reason monitoring is poor or not implemented correctly is that it can be hard to implement. If you make it hard for application developers to instrument their applications, collect the data, or visualize its results, they won't do it. If your monitoring infrastructure is manual or overly complex then fault and issues will result in monitoring gaps, failures, and the potential for you to spend more time fixing and maintaining your monitoring than actually monitoring.

Monitoring implementations and deployments should be automated wherever possible:

- Deployments should be managed by configuration management.
- Configuration of hosts and services should be via discovery or self-service submission, so new applications can be automatically monitored rather than needing someone to add them.
- Adding instrumentation should be simple and based on a pluggable utility pattern, and developers should be able to include a library or the like rather than having to configure it themselves.
- Data and visualization should be self-service. Everyone who needs to see the outputs of monitoring should be able to query and visualize those outputs. (This is not to say that you shouldn't build dashboards for people, but rather that if they want more they shouldn't have to ask you for it.)

Good monitoring summary

Good monitoring should provide:

- The state of the world, from the top (the business) down.
- Assistance in fault diagnostics.
- A source of information for infrastructure, application development, and business folks.

And it should be:

- Built into design and the life cycle of application development and deployment.
- Automated and provided as self-service, where possible.

 **NOTE** This definition of “good” monitoring heavily overlaps with an emerging term: observability. You can read more about this in Cindy Sridharan’s excellent blog post on the differences between the two.

Let’s now look at the actual mechanics of monitoring.

Monitoring mechanics

There are a variety of ways you can monitor. Indeed, you could argue that everything from unit testing to checklists are a form of monitoring.

 **NOTE** Lindsay Holmwood has a useful presentation on test-driven monitoring that talks about the connection between testing and monitoring. Additionally, Cindy Sridharan’s post on testing microservices draws some interesting parallels between testing and monitoring.

Traditionally, though, the definition of monitoring focuses on checking and measuring the state of an application.

Probing and introspection

There are two major approaches to monitoring applications: probing and introspection.² Probing monitoring probes the outside of an application. You query the external characteristics of an application: does it respond to a poll on an open port and return the correct data or response code? An example of probing monitoring is performing an ICMP check and confirming you have received a response. Nagios is an example of a monitoring system that is largely based around probe monitoring.

Introspection monitoring looks at what's inside the application. The application is instrumented and returns measurements of its state, the state of internal components, or the performance of transactions or events. This is data that shows exactly how your application is functioning, rather than just its availability or the behavior of its surface area. Introspection monitoring either emits events, logs, and metrics to a monitoring tool or exposes this information on a status or health endpoint of some kind, which can then be collected by a monitoring tool.

The introspection approach provides an idea of the actual running state of applications. It allows you to communicate a much richer, more contextual set of information about the state of your application than probing monitoring does. It also provides a better approach to exposing the information both you and the business require to monitor your application.

This is not to say that probing monitoring has no place. It is often useful to know the state of external aspects of an application, especially if the application is provided by a third party and if you don't have insight into its internal operations. It is often also useful to view your application from outside to understand certain types of networking, security, or availability issues. It's generally recommended to have probing for your safety net, a catchall that something is wrong, but to use introspection to drive reporting and diagnostics. We'll see some probe monitoring in Chapter 10.

²Some folks call probing and introspection, black-box and white-box monitoring respectively.

Pull versus push

There are two approaches to how monitoring checks are executed that are worth briefly discussing. These are the pull versus push approaches.

Pull-based systems scrape or check a remote application—for example, an endpoint containing metrics or, as from our probing example, a check using ICMP. In push-based systems, applications emit events that are received by the monitoring system.

Both approaches have pros and cons. There's considerable debate in monitoring circles about those pros and cons, but for the purposes of many users, the debate is largely moot. Prometheus is primarily a pull-based system, but it also supports receiving events pushed into a gateway. We'll show you how to use both approaches in this book.

Types of monitoring data

Monitoring tools can collect a variety of different types of data. That data primarily takes two forms:

- Metrics — Most modern monitoring tools rely most heavily on metrics to help us understand what's going on in our environments. Metrics are stored as time series data that record the state of measures of your applications. We'll see more about this shortly.
- Logs — Logs are (usually textual) events emitted from an application. While they're helpful for letting you know what's happening, they're often most useful for fault diagnosis and investigation. We won't look at logs much in this book, but there are plenty of tools available, like the ELK stack, for collecting and managing log events.

 **TIP** I've written a book about the ELK stack that might interest you.

As Prometheus is primarily focused on collecting time series data, let's take a deeper look at metrics.

Metrics

Metrics always appear to be the most straightforward part of any monitoring architecture. As a result, we sometimes don't invest quite enough time in understanding what we're collecting, why we're collecting it, and what we're doing with those metrics.

In a lot of monitoring frameworks, the focus is on fault detection: detecting if a specific system event or state has occurred (this is very much the Nagios style of operation—more on this below). When we receive a notification about a specific system event, usually we go look at whatever metrics we're collecting, if any, to find out what exactly has happened and why. In this world, metrics are seen as a by-product of, or a supplement to, our fault detection.

 **TIP** See the discussion later in this chapter about notification design for further reasons why this is a challenging problem.

Prometheus changes this idea of “metrics as supplement.” Metrics are the most important part of your monitoring workflow. Prometheus turns the fault-detection-centric model on its head. Metrics provide the state and availability of your environment and its performance.

 **NOTE** This book generally avoids duplicating Boolean status checks when a metric can provide information on both state and performance.

Harnessed correctly, metrics provide a dynamic, real-time picture of the state of your infrastructure that will help you manage and make good decisions about your environment. Additionally, through anomaly detection and pattern analysis, metrics have the potential to identify faults or issues before they occur or before the specific system event that indicates an outage is generated.

So what's a metric?

As metrics and measurement are so critical to our monitoring framework, we're going to help you understand what metrics are and how to work with them. This is intended to be a simplified background that will allow you to understand what different types of metrics, data, and visualizations will contribute to our monitoring framework.

Metrics are measures of properties of components of software or hardware. To make a metric useful we keep track of its state, generally recording data points over time. Those data points are called observations. An observation consists of the value, a timestamp, and sometimes a series of properties that describe the observation, such as a source or tags. A collection of observations is called a time series.

A classic example of time series data we might collect is website visits, or hits. We periodically collect observations about our website hits, recording the number of hits and the times of the observations. We might also collect properties such as the source of a hit, which server was hit, or a variety of other information.

We generally collect observations at a fixed-time interval—we call this the granularity or resolution. This could range from one second to five minutes to 60

minutes or more. Choosing the right granularity at which to record a metric is critical. Choose too coarse a granularity and you can easily miss the detail. For example, sampling CPU or memory usage at five-minute intervals is highly unlikely to identify anomalies in your data. Alternatively, choosing fine granularity can result in the need to store and interpret large amounts of data.

Time series data is a chronologically ordered list of these observations. Time series metrics are often visualized, sometimes with a mathematical function applied, as a two-dimensional plot with data values on the y-axis and time on the x-axis. Often you'll see multiple data values plotted on the y-axis—for example, the CPU usage values from multiple hosts or successful and unsuccessful transactions.

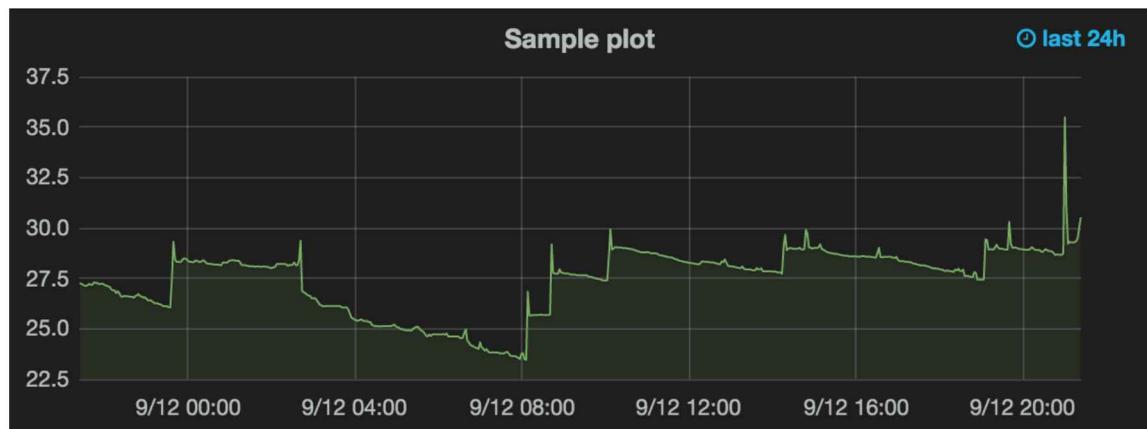


Figure 1.3: A sample plot

These plots can be incredibly useful. They provide us with a visual representation of critical data that is (relatively) easy to interpret, certainly with more facility than perusing the same data in the form of a list of values. They also present us with a historical view of whatever we're monitoring: they show us what has changed and when. We can use both of these capabilities to understand what's happening in our environment and when it happened.

Types of metrics

There are a variety of different types of metrics you'll see in the wild.

Gauges

The first type of metric we'll look at is a gauge. Gauges are numbers that are expected to go up or down. A gauge is essentially a snapshot of a specific measurement. The classic metrics of CPU, memory, and disk usage are usually articulated as gauges. For business metrics, a gauge might be the number of customers present on a site.

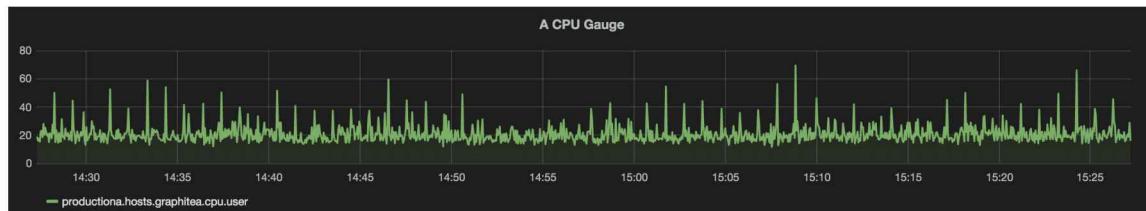


Figure 1.4: A sample gauge

Counters

The second type of metric we'll see frequently is a counter. Counters are numbers that increase over time and never decrease. Although they never decrease, counters can sometimes reset to zero and start incrementing again. Good examples of application and infrastructure counters are system uptime, the number of bytes sent and received by a device, or the number of logins. Examples of business counters might be the number of sales in a month or the number of orders received by an application.

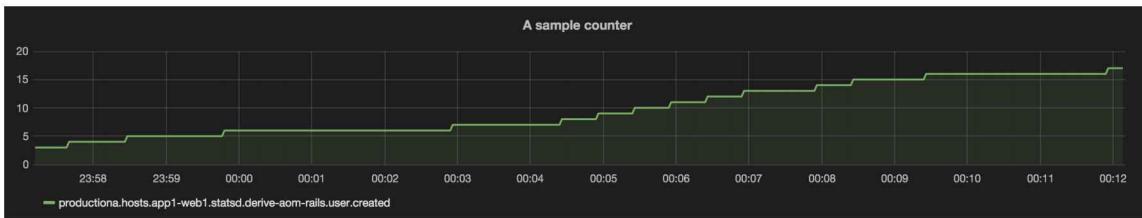


Figure 1.5: A sample counter

In this figure we have a counter incrementing over a period.

A useful thing about counters is that they let you calculate rates of change. Each observed value is a moment in time: t . You can subtract the value at t from the value at $t+1$ to get the rate of change between the two values. A lot of useful information can be understood by understanding the rate of change between two values. For example, the number of logins is marginally interesting, but create a rate from it and you can see the number of logins per second, which should help identify periods of site popularity.

Histograms

A histogram is a metric that samples observations. This is a frequency distribution of a dataset. You group data together—a process called “binning”—and present the groups in a such a way that their relative sizes are visualized. Each observation is counted and placed into buckets. This results in multiple metrics: one for each bucket, plus metrics for the sum and count of all values.

A common visualization of a frequency distribution histogram looks like a bar graph.

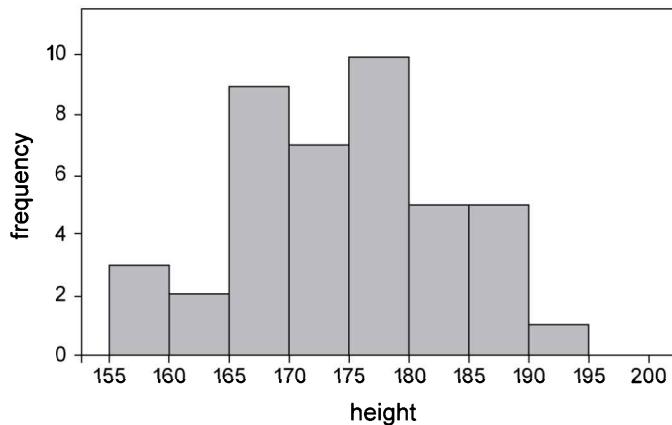


Figure 1.6: A histogram example

Here we see a sample histogram for the frequency distribution of heights. On the y-axis we have the frequency and on the x-axis we have the distribution of heights. We see that for the height 160–165 cm tall there is a distribution of two.

NOTE There's another metric type, called a summary, which is similar to a histogram, but it also calculates percentiles. You can read more about implementation details and some caveats of histogram and summaries specific to Prometheus.

Histograms can be powerful representations of your time series data and especially useful for visualizing data such as application latencies.

Metric summaries

Often the value of a single metric isn't useful to us. Instead, visualization of a metric requires applying mathematical transformations to it. For example, we might apply statistical functions to our metric or to groups of metrics. Some common functions we might apply include:

- Count or n — Counts the number of observations in a specific time interval.
- Sum — To *sum* is to add together values from all observations in a specific time interval.
- Average — Provides the *mean* of all values in a specific time interval.
- Median — The *median* is the dead center of our values: exactly 50 percent of values are below it, and 50 percent are above it.
- Percentiles — Measures the values below which a given percentage of observations in a group of observations fall.
- Standard deviation — Shows standard deviation from the mean in the distribution of our metrics. This measures the variation in a data set. A standard deviation of 0 means the distribution is equal to the mean of the data. Higher deviations mean the data is spread out over a range of values.
- Rates of change — Rates of change representations show the degree of change between data in a time series.



TIP This is a brief introduction to these summary methods. We'll use some of them in more detail later in the book.

Metric aggregation

In addition to summaries of specific metrics, you often want to show aggregated views of metrics from multiple sources, such as the disk space usage of all your application servers. The most typical example of this results in multiple metrics being displayed on a single plot. This is useful in identifying broad trends over

your environment. For example, an intermittent fault in a load balancer might result in web traffic dropping off for multiple servers. This is often easier to see in aggregate than by reviewing each individual metric.

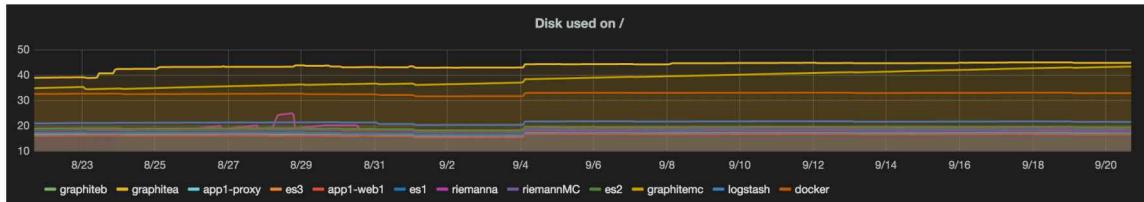


Figure 1.7: An aggregated collection of metrics

In this plot we see disk usage from numerous hosts over a 30-day period. It gives us a quick way to ascertain the current state (and rate of change) of a group of hosts.

Ultimately you'll find that a combination of single and aggregate metrics provide the most representative view of the health of your environment: the former to drill down into specific issues, and the latter to see the high-level state.

Let's take a deeper dive into types of metric summaries: the whys, why nots, and hows of using averages, the median, standard deviation, percentiles, and other statistical choices.



NOTE This is a high-level overview of some statistical techniques rather than a deep dive into the topic. Exploration of some topics may appear overly simplistic to folks with strong statistics or mathematics backgrounds.

Averages

Averages are the de facto metric analysis method. Indeed, pretty much everyone who has ever monitored or analyzed a website or application has used averages.

In the web operations world, for example, many companies live and die by the average response time of their site or API.

Averages are attractive because they are easy to calculate. Let's say we have a list of seven time series values: 12, 22, 15, 3, 7, 94, and 39. To calculate the average we sum the list of values and divide the total by the number of values in the list.

$$(12 + 22 + 15 + 3 + 7 + 94 + 39) / 7 = 27.428571428571$$

We first sum the seven values to get the total of 192. We then divide the sum by the number of values, here 7, to return the average: 27.428571428571. Seems pretty simple, huh? The devil, as they say, is in the details.

Averages assume there is a normal event or that your data is a normal (or Gaussian) distribution—for example, in our average response time, it's assumed that all events run at equal speed or that response time distribution is roughly bell curved. But this is rarely the case with applications. In fact, there's an old statistics joke about a statistician who jumps in a lake with an average depth of only 10 inches and nearly drowns...

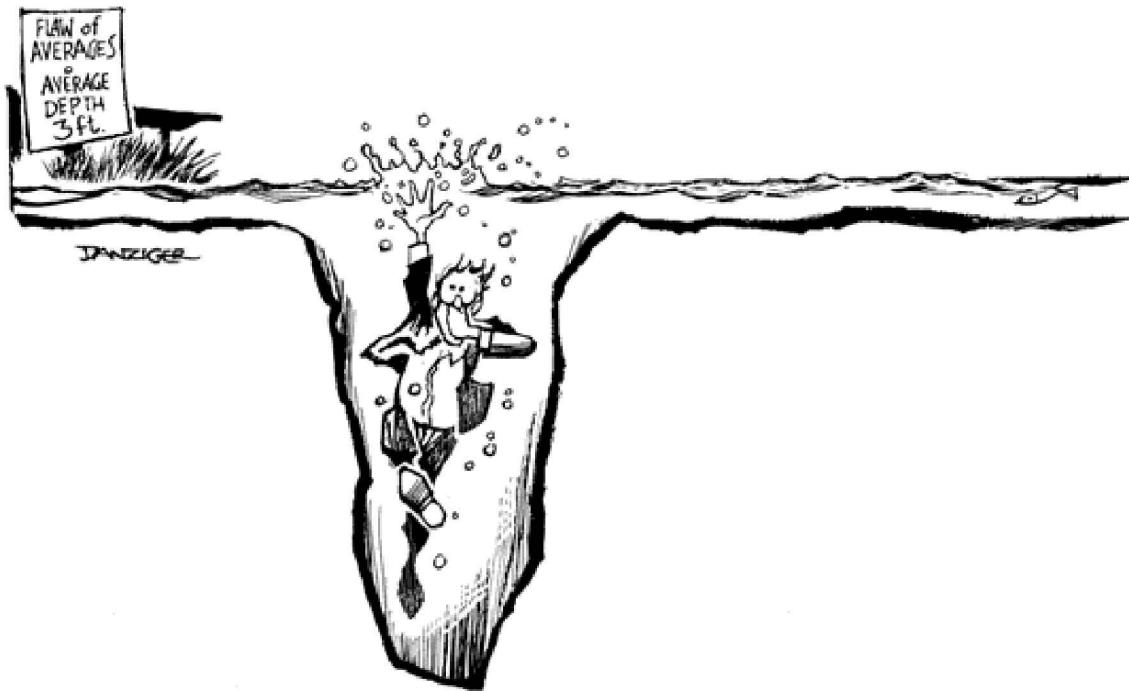


Figure 1.8: The flaw of averages - copyright Jeff Danzinger

Why did he nearly drown? The lake contained large areas of shallow water and some areas of deep water. Because there were larger areas of shallow water, the average depth was lower overall. In the monitoring world the same principal applies: lots of low values in our average distort or hide high values and vice versa. These hidden outliers can mean that while we think most of our users are experiencing a quality service, there may be a significant number that are not.

Let's look at an example using response times and requests for a website.

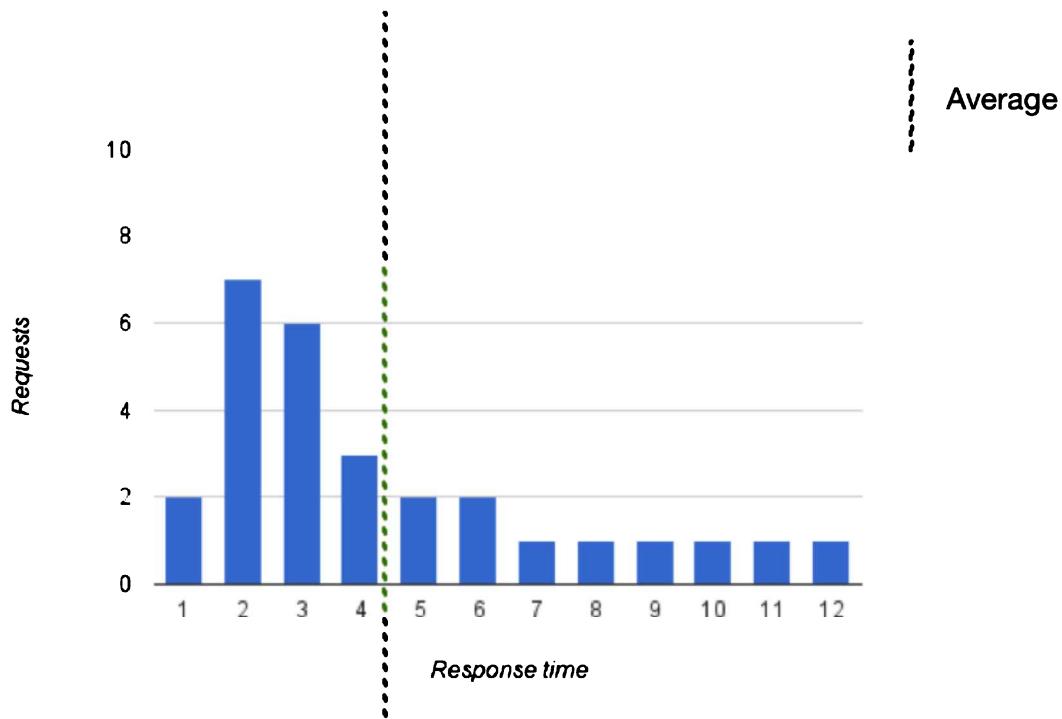


Figure 1.9: Response time average

Here we have a plot showing response time for a number of requests. Calculating the average response time would give us 4.1 seconds. The vast majority of our users would experience a (potentially) healthy 4.1 second response time. But many of our users are experiencing response times of up to 12 seconds, perhaps considerably less acceptable.

Let's look at another example with a wider distribution of values.

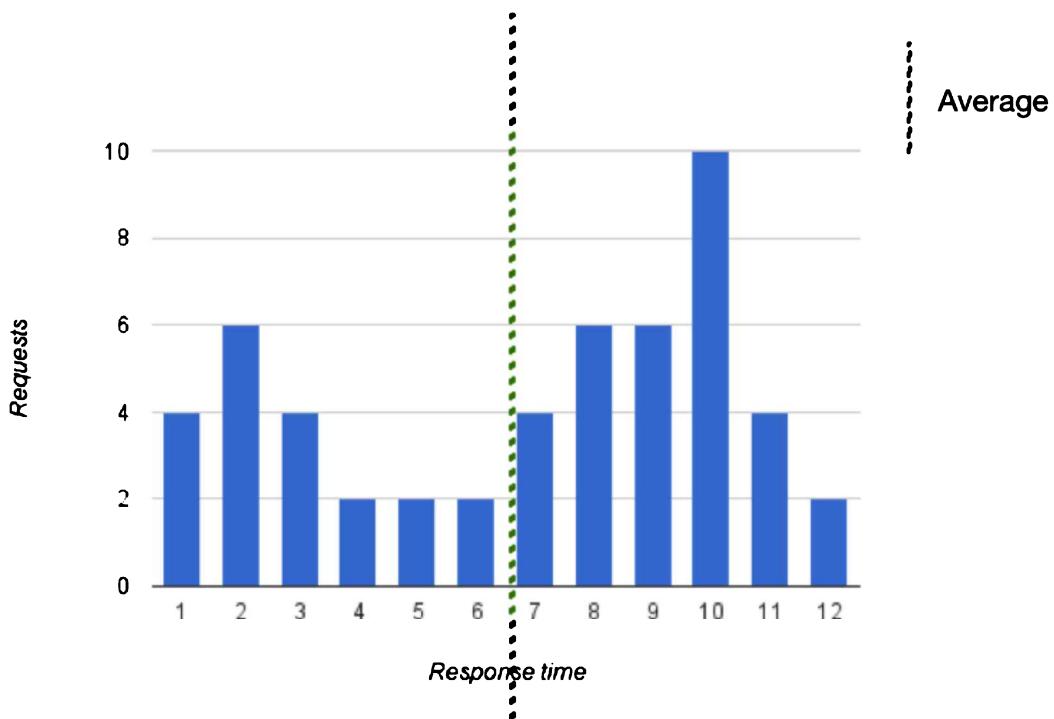


Figure 1.10: Response time average Mk II

Here our average would be a less stellar 6.8 seconds. But worse, this average is considerably better than the response time received by the majority of our users with a heavy distribution of request times around 9, 10, and 11 seconds long. If we were relying on the average alone, we'd probably think our application was performing a lot better than it is.

Median

At this point you might be wondering about using the median. The median is the dead center of our values: exactly 50 percent of values are below it, and 50 percent are above it. If there's an odd number of values, then the median will be the value in the middle. For the first data set we looked at—12, 22, 15, 3, 7, 94, and 39—the median is 15. If there were an even number of values, the median

would be the mean of the two values in the middle. So if we were to remove 39 from our data set to make it even, the median would become 13.5.

Let's apply this to our two plots.

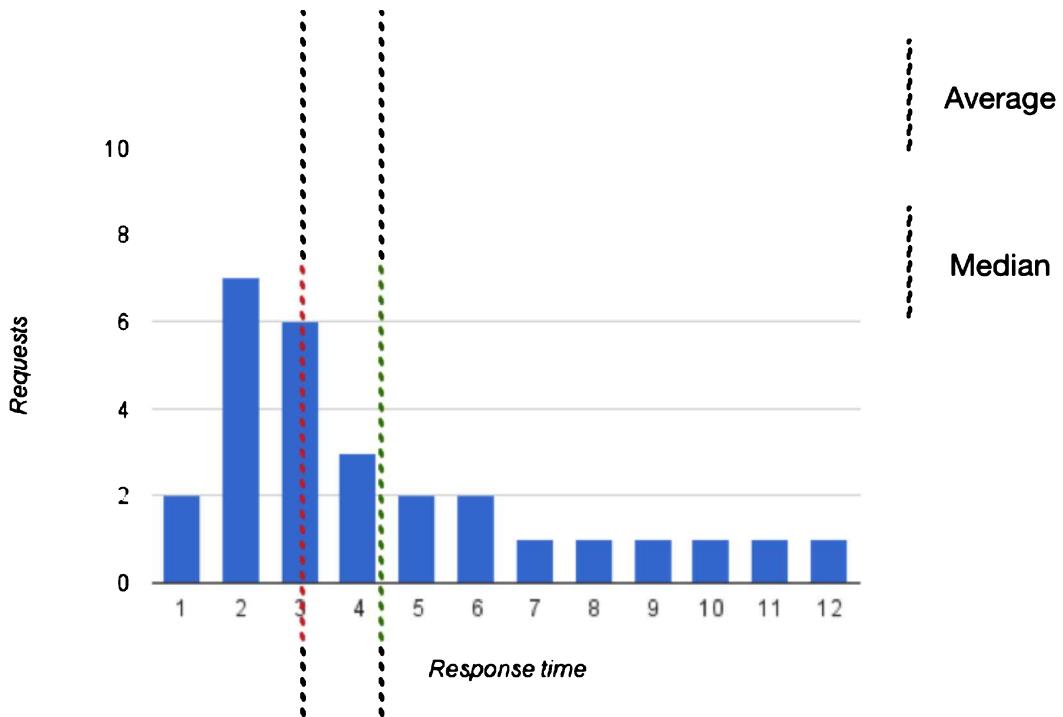


Figure 1.11: Response time average and median

We see in our first example figure that the median is 3, which provides an even rosier picture of our data.

In the second example the median is 8, a bit better but close enough to the average to render it ineffective.

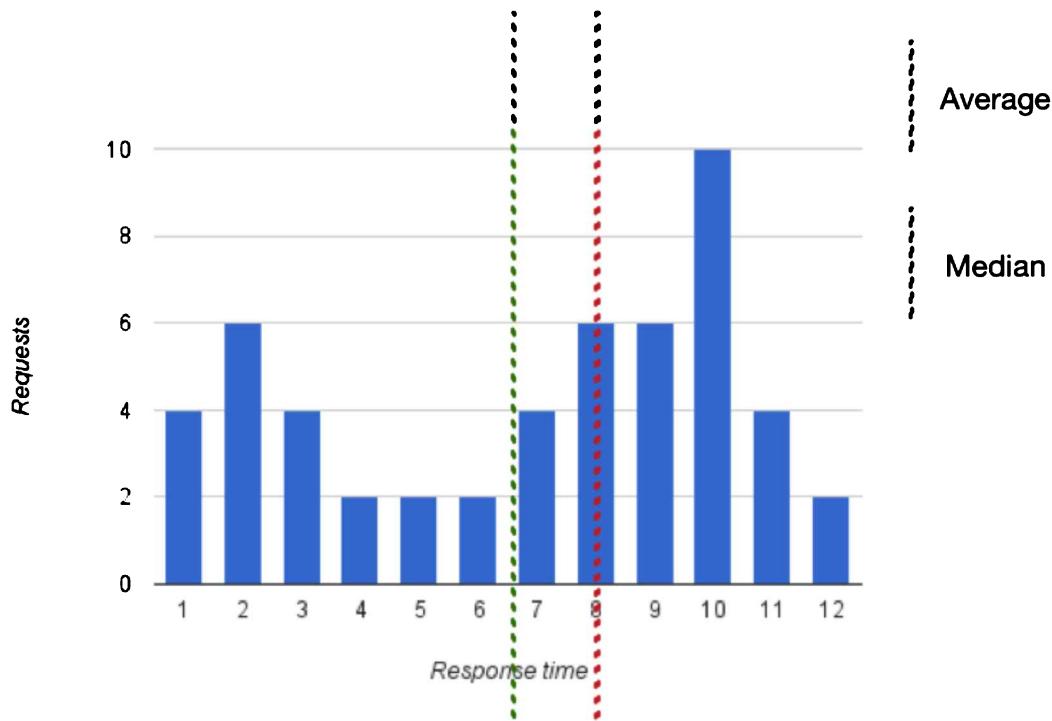


Figure 1.12: Response time average and median Mk II

You can probably already see that the problem again here is that, like the mean, the median works best when the data is on a bell curve... And in the real world that's not realistic.

Another commonly used technique to identify performance issues is to calculate the standard deviation of a metric from the mean.

Standard deviation

As we learned earlier in the chapter, standard deviation measures the variation or spread in a data set. A standard deviation of 0 means most of the data is close to the mean. Higher deviations mean the data is more distributed. Standard deviations are represented by positive or negative numbers suffixed with the *sigma* symbol—for example, 1 *sigma* is one standard deviation from the mean.

Like the mean and the median, however, standard deviation works best when the data is a normal distribution. In a normal distribution there's a simple way of articulating the distribution: the empirical rule, also known as the 68–95–99.7 rule or three-sigma rule. Within the rule, one standard deviation or 1 to -1 will represent 68.27 percent of all transactions on either side of the mean, two standard deviations or 2 to -2 would be 95.45 percent, and three standard deviations will represent 99.73 percent of all transactions.

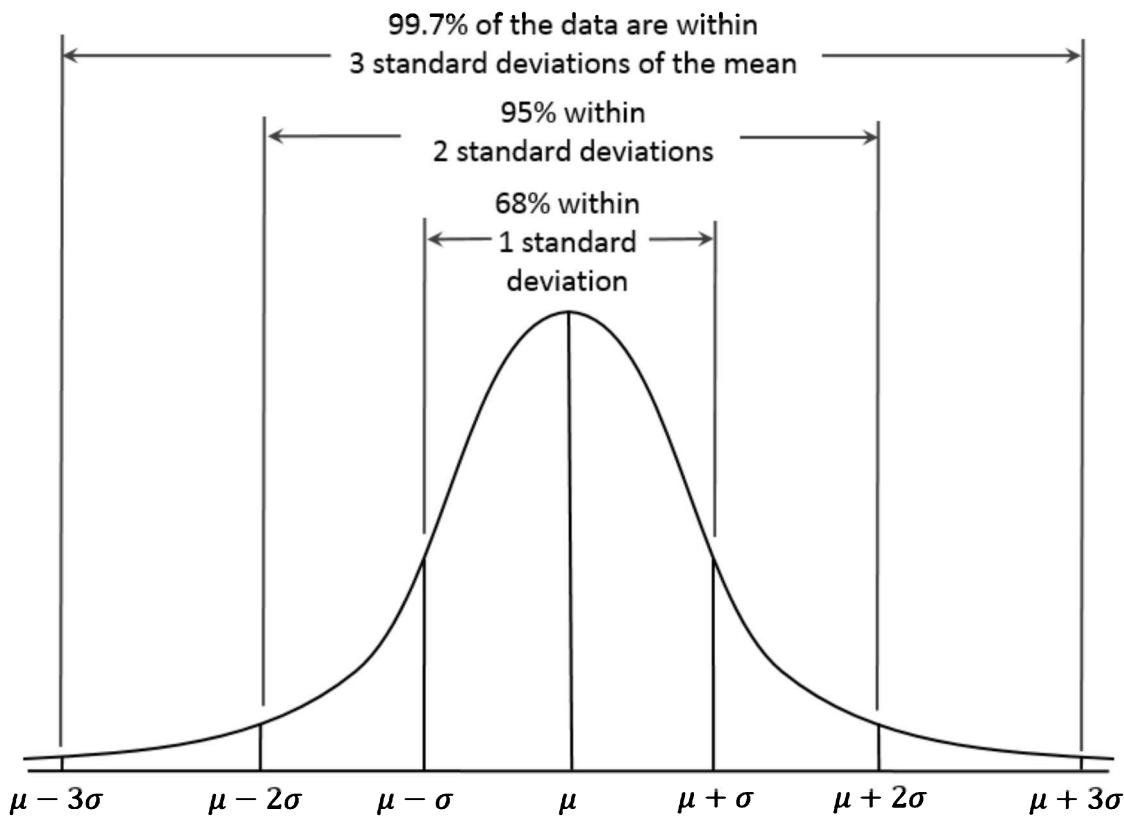


Figure 1.13: The empirical rule

Many monitoring approaches take advantage of the empirical rule and trigger on transactions or events that are more than two standard deviations from the mean, potentially catching performance outliers. In instances like our two previous examples, however, the standard deviation isn't overly helpful either. And without

a normal distribution of data, the resulting standard deviation can be highly misleading.

Thus far, our methods for identifying anomalous data in our metrics haven't been overly promising. But all is not lost! Our next method, percentiles, offer a little more hope.

Percentiles

Percentiles measure the values below which a given percentage of observations in a group of observations fall. Essentially they look at the distribution of values across your data set. For example, the median we looked at above is the 50th percentile (or p50). In the median, 50 percent of values fall below and 50 percent above. For metrics, percentiles make a lot of sense because they make the distribution of values easy to grasp. For example, the 99th-percentile value of 10 milliseconds for a transaction is easy to interpret: 99 percent of transactions were completed in 10 milliseconds or less, and 1 percent of transactions took more than 10 milliseconds.

 **TIP** Percentiles are a type of quantile.

Percentiles are ideal for identifying outliers. If a great experience on your site is a response time of less than 10 milliseconds then 99 percent of your users are having a great experience—but 1 percent of them are not. Once you're aware of this, you can focus on addressing the performance issue that's causing a problem for that 1 percent.

Let's apply this to our previous request and response time graphs and see what appears. We'll apply two percentiles, the 75th and 99th percentiles, to our first example data set.

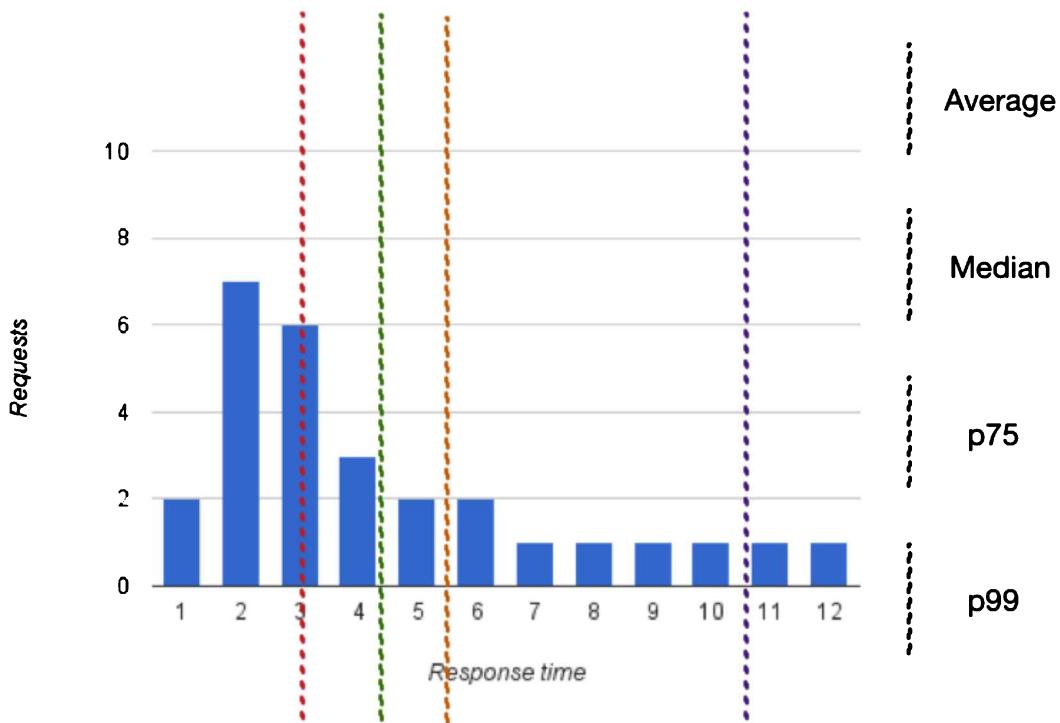


Figure 1.14: Response time average, median, and percentiles

We see that the 75th percentile is 5.5 seconds. That indicates that 75 percent completed in 5.5 seconds, and 25 percent were slower than that. Still pretty much in line with the earlier analysis we've examined for the data set. The 99th percentile, on the other hand, shows 10.74 seconds. This means 99 percent of users had request times of less than 10.74 seconds, and 1 percent had more than 10.74 seconds. This gives us a real picture of how our application is performing. We can also use the distribution between p75 and p99. If we're comfortable with 99 percent of users getting 10.74 second response times or better and 1 percent being slower than that, then we don't need to consider any further tuning. Alternatively, if we want a uniform response, or if we want to lower that 10.74 seconds across our distribution, we've now identified a pool of transactions we can trace, profile, and improve. As we adjust the performance, we'll also be able to see the p99 response time improve.

The second data set is even more clear.

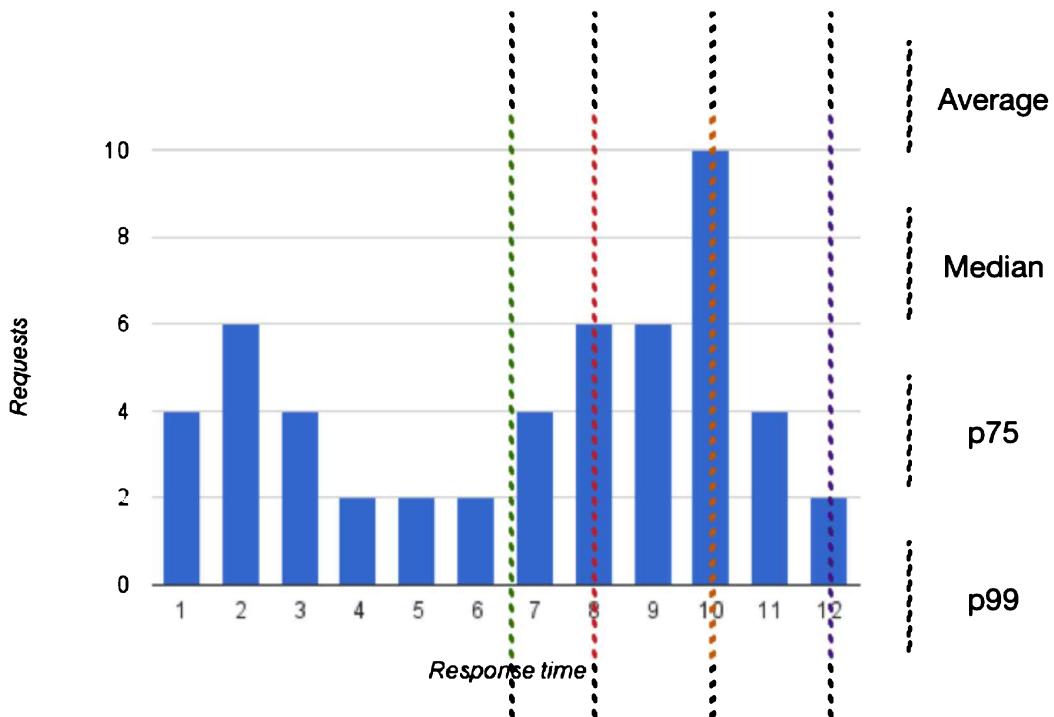


Figure 1.15: Response time average, median, and percentiles Mk II

The 75th percentile is 10 seconds and the 99th percentile is 12 seconds. Here the 99th percentile provides a clear picture of the broader distribution of our transactions. This is a far more accurate reflection of the outlying transactions from our site. We now know that—as opposed to what the mean response times would imply—not all users are enjoying an adequate experience. We can use this data to identify elements of our application we can potentially improve.

Percentiles, however, aren't perfect all the time. We recommend graphing several combinations of metrics to get a clear picture of the data. For example, when measuring latency it's often a good idea to display a graph that shows:

- The 50th percentile, or median.
- The 99th percentile.

- The max value.

The addition of the max value helps visualize the upward bounds of the metric you are measuring. It's again not perfect though—a high max value can dwarf other values in a graph.

We're going to apply percentiles and other calculations later in the book as we start to build checks and collect metrics.

Monitoring methodologies

We'll also make use of a combination of several monitoring methodologies on top of our metrics and metric aggregations to help focus our monitoring. We're going to combine elements of two monitoring methodologies:

- Brendan Gregg's USE or Utilization Saturation and Errors Method, which focuses on host-level monitoring.
- Google's Four Golden Signals, which focus on application-level monitoring.

Monitoring methodologies provide guidelines that allow you to narrow down and focus on specific metrics in the sea of time series you collect. When combined, these two frameworks—one focused on host-level performance, the other on application-level performance—represent a reasonably holistic view of your environment that should assist you in tackling any issues.

The USE Method

The USE, or Utilization Saturation and Errors, Method was developed by Brendan Gregg, a kernel and performance engineer at Netflix. The methodology proposes creating a checklist for server analysis that allows the fast identification of issues.

You work down the checklist to identify common performance issues, making use of data collected from your environment.

The USE Method can be summarized as: For every resource, check utilization, saturation, and errors. The method is most effective for the monitoring of resources that suffer performance issues under high utilization or saturation. Let's quickly define each term to help understand this.

- A resource - A component of a system. In Gregg's definition of the model it's traditionally a physical server component like CPUs, disks, etc., but many folks also include software resources in the definition.
- Utilization - The average time the resource is busy doing work. It's usually expressed as a percentage over time.
- Saturation - The measure of queued work for a resource, work it can't process yet. This is usually expressed as queue length.
- Errors - The scalar count of error events for a resource.

We combine these definitions to create a checklist of the resources and an approach to monitor each element of the methodology: utilization, saturation, or errors. How might this work? Well, let's say we have a serious performance issue, and we want to dive into some diagnosis. We refer to our checklist and check each element for each monitored component. In our example, we'll start with CPU:

- CPU utilization as a percentage over time.
- CPU saturation as the number of processes awaiting CPU time.
- Errors, generally less important for the CPU resource.

And then, perhaps, memory:

- Memory utilization as a percentage over time.
- Memory saturation measured via monitoring swapping.
- Errors, generally less important here but also can be captured.

And so on through other components on the system until we've identified the bottleneck or signal that points us to the issue.

We'll see more of this in Chapter 4 when we look at monitoring some system-level metrics.



TIP You can find an example checklist for a Linux system [here](#).

The Google Four Golden Signals

The Google Four Golden Signals come out of the Google SRE book. They take a similar approach to the USE Method, specifying a series of general metric types to monitor. Rather than being system-level-focused time series, the metric types in this methodology are more application or user-facing:

- **Latency** - The time taken to service a request, distinguishing between the latency of successful and failed requests. A failed request, for example, might return with very low latency skewing your results.
- **Traffic** - The demand on your system—for example, HTTP requests per second or transactions for a database system.
- **Errors** - The rate that requests fail, whether explicit failures like HTTP 500 errors, implicit failures like wrong or invalid content being returned, or policy-based failures—for instance if you've mandated that failures over 30ms should be considered errors.
- **Saturation** - The “fullness” of your application or the resources that are constraining it—for example, memory or IO. This also includes impending saturation, such as a rapidly filling disk.

Using the golden signals is easy. Select high-level metrics that match each signal and build alerts for them. If one of those signals becomes an issue then an alert will be generated and you can diagnose or resolve the issue.

We'll see golden signals again in Chapter 7 and 8 when we look at monitoring some applications.

 **TIP** There's a related framework called RED—or Rate, Errors, and Duration, developed by the team at Weaveworks, that might also interest you.

Contextual, useful alerts and notifications

Alerts and notifications are the primary output from monitoring tools. So what's the difference between an alert and a notification? An alert is raised when something happens—for example, when a threshold is reached. This, however, doesn't mean anyone's been told about the event. That's where notifications come in. A notification takes the alert and tells someone or something about it: an email is sent, an SMS is triggered, a ticket is opened, or the like. It may seem like this should be a really simple domain, but it contains a lot of complexity and is often poorly implemented and managed.

To build a good notification system you need to consider the basics of:

- What problems to notify on.
- Who to tell about a problem.
- How to tell them.
- How often to tell them.
- When to stop telling them, do something else, or escalate to someone else.

If you get it wrong and generate too many notifications then people will be unable to take action on them all and may even mute them. We all have war stories of mailbox folders full of thousands of notification emails from monitoring systems.³ Sometimes so many notifications are generated that you suffer from alert fatigue and ignore them (or worse, conduct notification management via Select All -> Delete). Consequently, you're likely to miss actual critical notifications when they are sent.

Most importantly, you need to work out *what* to tell whoever is receiving the notifications. Notifications are usually the sole signal that you receive to tell you that something is amiss or requires your attention. They need to be concise, articulate, accurate, digestible, and actionable. Designing your notifications to actually be useful is critical. Let's make a brief digression and see why this matters. We'll look at a typical Nagios notification for disk space.

Listing 1.1: Sample Nagios notification

```
PROBLEM Host: server.example.com
Service: Disk Space

State is now: WARNING for 0d 0h 2m 4s (was: WARNING) after 3/3
checks

Notification sent at: Thu Aug 7th 03:36:42 UTC 2015 (
notification number 1)

Additional info:
DISK WARNING - free space: /data 678912 MB (9% inode=99%)
```

Imagine you've just received this notification at 3:36 a.m. What does it tell you? That we have a host with a disk space warning. And that the /data volume is 91 percent full. At first glance this seems useful, but in reality it's not all that practical. First, is this a sudden increase, or has this grown gradually? And what's the rate

³Or cron.

of expansion? (Consider that 9 percent disk space free on a 1 GB partition is quite different from 9 percent disk space free on a 1 TB disk.) Can you ignore or mute this notification or do you need to act now? Without the additional context your ability to take action on the notification is limited, and you need to invest considerably more time to gather context.

In our framework we're going to focus on:

- Making notifications actionable, clear, and articulate. Just the use of notifications written by humans rather than by computers can make a significant difference in the clarity and utility of those notifications.
- Adding context to notifications. We're going to send notifications that contain additional information about the component we're notifying on.
- Only sending those notifications that make sense.

 **TIP** The simplest advice we can give here is to remember/ notifications are read by humans, not computers. Design them accordingly.

Visualization

Visualizing data is both an incredibly powerful analytic and interpretive technique and an amazing learning tool. Metrics and their visualizations are often tricky to interpret. Humans tend towards apophenia—the perception of meaningful patterns within random data—when viewing visualizations. This often leads to making sudden leaps from correlation to causation, and can be further exacerbated by the granularity and resolution of our available data, how we choose to represent it, and the scale on which we represent it.

Our ideal visualizations will clearly show the data, with an emphasis on highlighting substance over visuals. In this book we're not going to look at a lot of visualizations but where we have, we've tried to build visuals that subscribe to these broad rules:

- Clearly show the data.
- Induce the viewer to think about the substance, not the visuals.
- Avoid distorting the data.
- Make large data sets coherent.
- Allow changing perspectives of granularity without impacting comprehension.

We've drawn most of these ideas from Edward Tufte's *The Visual Display of Quantitative Information* and thoroughly recommend reading it to help you build good visualizations.

There's also a great post from the Datadog team on visualizing time series data that is worth reading.

But didn't you write that other book?

As many folks know, I am one of the maintainers of Riemann, an event stream processor focused on monitoring distributed systems. I wrote a book about monitoring in which I used Riemann as a centerpiece to explore new monitoring patterns and approaches. In the book, I described an architecture of introspection monitoring (with some selective probing monitoring).

In the book I also focused on push-based monitoring over pull-based monitoring. There are lots of reasons I favor the push model versus the pull model but, as we mentioned earlier, for many folks the distinction is arbitrary. Indeed, many of the concerns of either approach don't impact implementation due to issues like scale. Other concerns, like many arguments over implementation or tool choice,

don't change the potential success of the implementation. I'm a strong exponent of using tools that work for you, rather than unreviewed adoption of the latest trend or dogmatism.

It's this lack of distinction for folks, and a desire not to be dogmatic about my beliefs, that has encouraged me to write another book, this one about one of the leading pull-based monitoring tools: Prometheus. In the Art of Monitoring I wrote:

Perhaps a better way of looking at these tool choices is that they are merely ways to articulate the change in monitoring approach that is proposed in this book. They are the trees in the woods. If you find other tools that work better for you and achieve the same results then we'd love to hear from you. Write a blog post, give a talk, or share your configuration.

Hence, you'll see much of the methodology of The Art of Monitoring reflected in this book—indeed, much of this chapter is a distillation of some of the book's elements. We're taking the core motivation of that book—a better way to monitor applications—and applying it with an alternative tool, and a different architecture and approach.

What's in the book?

This book covers an introduction to a good approach to monitoring, and then uses Prometheus to instantiate that monitoring approach. By the end of the book you should have a readily extensible and scalable monitoring platform.

The book assumes you want to **build, rather than buy** a monitoring platform. There are a lot of off-the-shelf Software-as-a-Service (SaaS) and cloud-based monitoring solutions that might work for you. There's even some hosted Prometheus options. For many folks, this is a better solution when starting out with monitoring rather than investing in building their own. It's our view that ultimately

most folks, as their environment and requirements grow, will discover that these platforms don't quite suit their needs, and will build (some) monitoring in house. But whether that's the case for you is something you'll need to determine yourself.

In this book, we're going to introduce you to the Prometheus monitoring platform piece by piece, starting with monitoring node and container metrics, service discovery, alerting, and then instrumenting and monitoring applications. The book will try to cover a representative sample of technologies you're likely to manage yourself but that can be adapted to a wide variety of other environments and stacks.

The book's chapters are:

- Chapter 1: This introduction.
- Chapter 2: Introducing Prometheus.
- Chapter 3: Installing Prometheus.
- Chapter 4: Monitoring nodes and containers.
- Chapter 5: Service discovery.
- Chapter 6: Alerting and AlertManager.
- Chapter 7: Scaling.
- Chapter 8: Instrumenting an application.
- Chapter 9: Logging as instrumentation.
- Chapter 10: Probing.
- Chapter 11: Pushgateway.
- Chapter 12: Monitoring a stack - Kubernetes.
- Chapter 13: Monitoring a stack - Application.

Summary

In this chapter we introduced you to modern monitoring approaches. We laid out the details of several types of monitoring implementations. We discussed what makes good and bad monitoring, and how to avoid poor monitoring outcomes.

We also introduced the details of time series data and metrics to you. We broke down the types of data that can be delivered as metrics. And we demonstrated some common mathematical functions applied to metrics to manipulate and aggregate them.

In the next chapter, we're going to introduce you to Prometheus and give some insight into its architecture and components.

Chapter 2

Introduction to Prometheus

In this chapter we're going to introduce you to Prometheus, its origins, and give you an overview of:

- Where Prometheus came from and why.
- Prometheus architecture and design.
- The Prometheus data model.
- The Prometheus ecosystem.

This should give you an introduction and understanding of what Prometheus is and where it fits into the monitoring universe.



NOTE This book focuses on Prometheus version 2.0 and later. Much of the book's information will not work for earlier releases.

The Prometheus backstory

Once upon a time there was a company in Mountain View, California, called Google. They ran a swathe of products, most famously an advertising system, search engine platform. To run these diverse products they built a platform called Borg. The Borg system is “a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines.”¹ The open-source container manager Kubernetes owes much of its heritage to Borg. Shortly after Borg was deployed at Google, folks realized that this complexity required a similarly capable monitoring system. Google built that system and called it Borgmon. Borgmon is a real-time-focused time series monitoring system that uses that data to identify issues and alert on them.

 **NOTE** Neither Borg nor Borgmon have ever been open sourced. It’s only recent that one can learn about how they work. You can read a bit more about it in the Practical Alerting chapter of the SRE book.

Prometheus owes its inspiration to Google’s Borgmon. It was originally developed by Matt T. Proud, an ex-Google SRE, as a research project. After Proud joined SoundCloud, he teamed up with another engineer, Julius Volz, to develop Prometheus in earnest. Other developers joined the effort, and it continued development internally at SoundCloud, culminating in a public release in January 2015.

Like Borgmon, Prometheus was primarily designed to provide near real-time introspection monitoring of dynamic cloud- and container-based microservices, services, and applications. SoundCloud was an earlier adopter of these architec-

¹Abhishek Verma et al, Large-scale cluster management at Google with Borg, EuroSys, 2015.

tural patterns, and Prometheus was built to respond to those needs. These days, Prometheus is used by a wide range of companies, generally for similar monitoring needs, but also for monitoring of more traditional architectures.

Prometheus is focused on what's happening right now, rather than tracking data over weeks or months. This is based on the premise that the majority of monitoring queries and alerts are generated from recent, usually less than day-old, data. Facebook validated this in a paper on Gorilla, its internal time series database. Facebook discovered that 85 percent of queries were for data less than 26 hours old. Prometheus assumes that the problems you may be trying to fix are likely recent, hence the most useful data is the most recent data. This is reflected in the powerful query language available and the typically limited retention period for monitoring data.

Prometheus is written in Go, open source, and licensed under the Apache 2.0 license. It is incubated under the Cloud Native Computing Foundation.

Prometheus architecture

Prometheus works by scraping or pulling time series data exposed from applications. The time series data is exposed by the applications themselves often via client libraries or via proxies called exporters, as HTTP endpoints. Exporters and client libraries exist for many languages, frameworks, and open-source applications—for example, for web servers like Apache and databases like MySQL.

Prometheus also has a push gateway you can use to receive small volumes of data—for example, data from targets that can't be pulled, like transient jobs or targets behind firewalls.

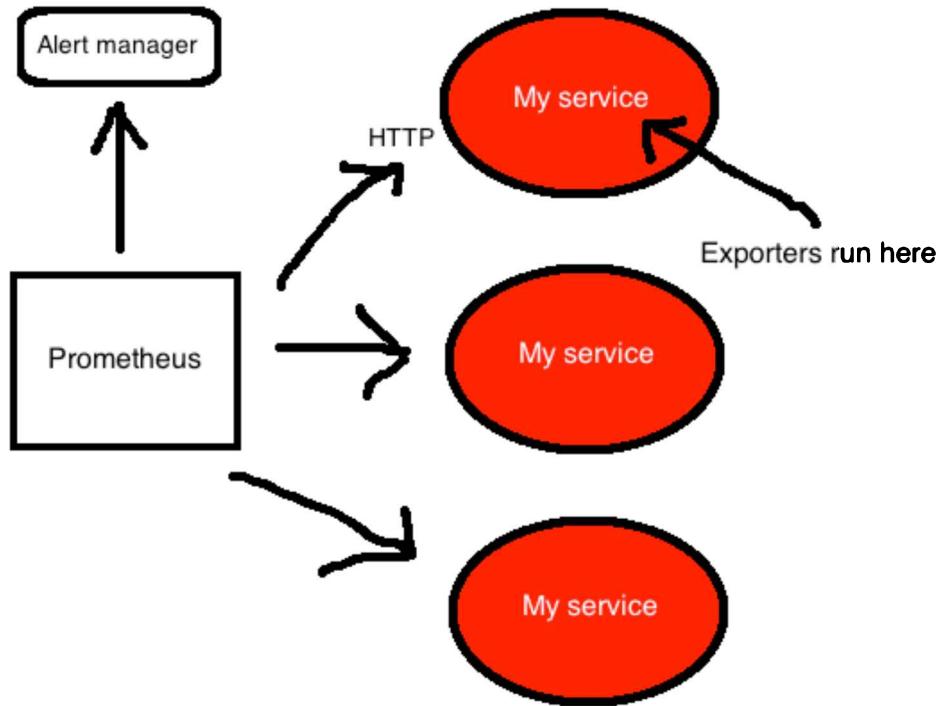


Figure 2.1: Prometheus architecture

Metric collection

Prometheus calls the source of metrics it can scrape endpoints. An endpoint usually corresponds to a single process, host, service, or application. To scrape an endpoint, Prometheus defines configuration called a target. This is the information required to perform the scrape—for example, how to connect to it, what metadata to apply, any authentication required to connect, or other information that defines how the scrape will occur. Groups of targets are called jobs. Jobs are usually groups of targets with the same role—for example, a cluster of Apache servers behind a load balancer. That is, they're effectively a group of like pro-

cesses.

The resulting time series data is collected and stored locally on the Prometheus server. It can also be sent from the server to external storage or to another time series database.

Service discovery

Discovery of resources to be monitored can be handled in a variety of ways including:

- A user-provided static list of resources.
- File-based discover—for example, using a configuration management tool to generate a list of resources that are automatically updated in Prometheus.
- Automated discovery—for example, querying a data store like Consul, running instances in Amazon or Google, or using DNS SRV records to generate a list of resources.

 **TIP** We'll see how to use a variety of service discovery approaches in Chapter 5.

Aggregation and alerting

The server can also query and aggregate the time series data, and can create rules to record commonly used queries and aggregations. This allows you to create new time series from existing time series—for example, calculating rates and ratios or producing aggregations like sums. This saves you having to recreate common aggregations, say ones you use for debugging, and the precomputation is potentially more performant than running the query each time it is required.

Prometheus can also define rules for alerting. These are criteria—for example, a resource time series starting to show escalated CPU usage—that can be configured to trigger an alert when the criteria are met. The Prometheus server doesn’t come with an inbuilt alerting tool. Instead, alerts are pushed from the Prometheus server to a separate server called Alertmanager. Alertmanager can manage, consolidate, and distribute alerts to a variety of destinations—for example, it can trigger an email when an alert is raised, but prevent duplicates.



TIP We’ll see a lot more about Alertmanager in Chapter 6.

Querying data

The Prometheus server also comes with an inbuilt querying language, PromQL; an expression browser; and a graphing interface you can use to explore the data on your server.

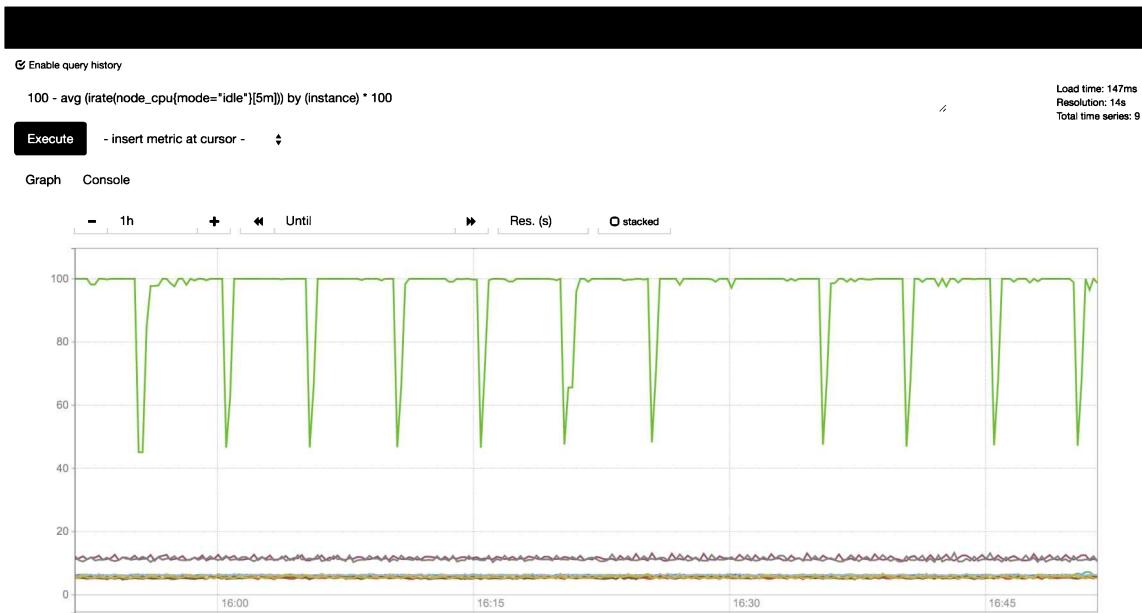


Figure 2.2: Prometheus expression browser

Autonomy

Each Prometheus server is designed to be as autonomous as possible. It is designed to scale to millions of time series from many thousands of hosts. Its data storage format is designed to keep disk use down and provide fast retrieval of time series during queries and aggregations.

TIP A good helping of memory (Prometheus does a lot in memory) and SSD disks are recommended for Prometheus servers, for speed and reliability. You are using SSDs, right?

Redundancy and high availability

Redundancy and high availability center on alerting resilience rather than data durability. The Prometheus team recommends deploying Prometheus servers to specific purposes and teams rather than to a single monolithic Prometheus server. If you do want to deploy in an HA configuration, two or more identically configured Prometheus servers collect the time series data, and any alerts generated are handled by a highly available Alertmanager configuration that deduplicates alerts.

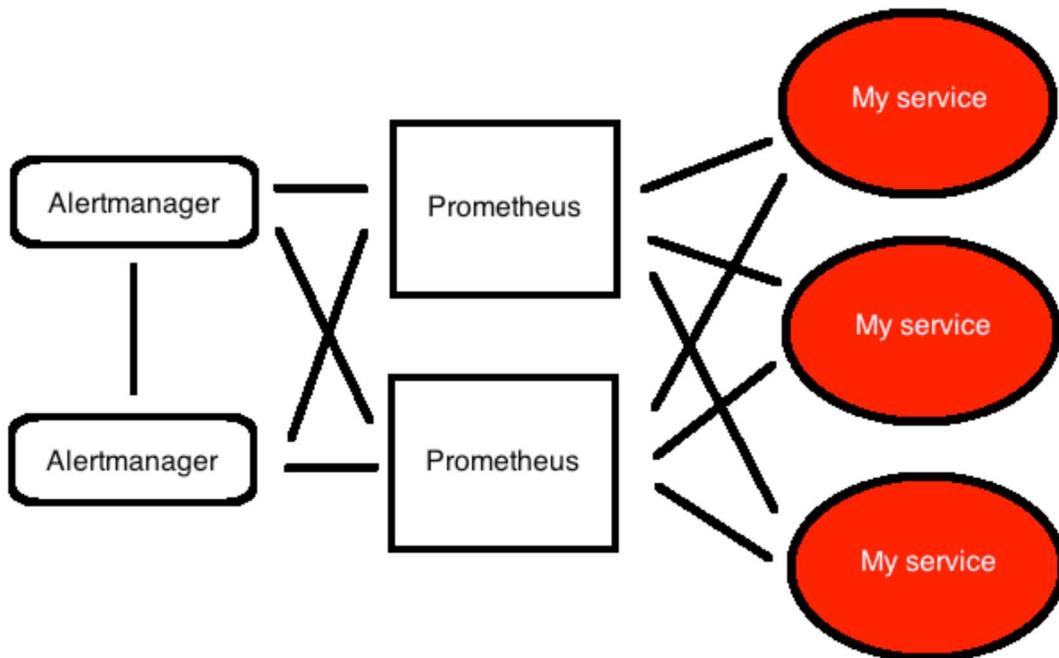


Figure 2.3: Redundant Prometheus architecture

 **TIP** We'll see how to implement this configuration in Chapter 7.

Visualization

Visualization is provided via an inbuilt expression browser and integration with the open-source dashboard Grafana. Other dashboards are also supported.

 **TIP** We'll get to know this integration in Chapter 4.

The Prometheus data model

As we've seen, Prometheus collects time series data. To handle this data it has a multi-dimensional time series data model. The time series data model combines time series names and key/value pairs called labels; these labels provide the dimensions. Each time series is uniquely identified by the combination of time series name and any assigned labels.

Metric names

The time series name usually describes the general nature of the time series data being collected—for example, `website_visits_total` as the total number of website visits.

The name can contain ASCII letters, digits, underscores, and colons.

Labels

Labels enable the Prometheus dimensional data model. They add context to a specific time series. For example, our `total_website_visits` time series could have labels that identify the name of the website, IP of the requester, or other dimensions that specifically identify that time series and connect it to its source. Prometheus can query on these dimensions to select one time series, groups of time series, or all relevant time series.

Labels come in two broad types: instrumentation labels and target labels. Instrumentation labels come from the resource being monitored—for example, for a HTTP-related time series, a label might show the specific HTTP verb used. These labels are added to the time series before they are scraped, such as by a client or exporter. Target labels relate more to your architecture—they might identify the data center where the time series originated. Target labels are added by Prometheus during and after the scrape.

A time series is identified by both its name and labels (although technically the name itself is also a label called `_name_`). If you add or change a label on a time series, Prometheus treats this as a new time series.

 **TIP** You can generally think of labels as tags, albeit in key/value form and where a new tag creates a new time series.

Label names can contain ASCII letters, digits, and underscores.

 **TIP** Label names prefixed with `__` are reserved for internal Prometheus use.

Samples

The actual value of the time series is called a sample. It consists of:

- A float64 value.
- A millisecond-precision timestamp.

Notation

Combining these elements we can see how Prometheus represents a time series as notation.

Listing 2.1: Time series notation

```
<time series name>{<label name>=<label value>, ...}
```

For example, our `total_website_visits` time series, with attached labels, might look like:

Listing 2.2: Example time series

```
total_website_visits{site="MegaApp", location="NJ", instance="webserver", job="web"}
```

The time series name is represented first, with a map of key/value pair labels attached. All time series generally have an `instance` label, which identifies the source host or application, and a `job` label, which contains the name of the job that scraped the specific time series.

 **NOTE** This is roughly the same notation that OpenTSDB uses, which in turn was influenced by Borgmon.

Metrics retention

Prometheus is designed for short-term monitoring and alerting needs. By default, it keeps 15 days of time series locally in its database. If you want to retain data for longer, the recommended approach is to send the required data to remote, third-party platforms. Prometheus has the ability to write to external data stores, which we'll see in Chapter 7.

Security model

Prometheus can be configured and deployed in a wide variety of ways. It makes two broad assumptions about trust:

- That untrusted users will be able to access the Prometheus server's HTTP API and hence all the data in the database.
- That only trusted users will have access to the command line, configuration files, rule files, and runtime configuration of Prometheus and its components.

 **TIP** Since Prometheus 2.0, some administrative elements of the HTTP API are disabled by default.

As such, Prometheus and its components do not provide any server-side authentication, authorization, or encryption. If you are working in a more secure environment you'll need to implement additional controls yourself—for instance by front-ending the Prometheus server with a reverse proxy or by proxying your exporters. Because of the huge potential variations in configuration, this book does not document how to do this.

Prometheus ecosystem

The Prometheus ecosystem has a mix of components provided by the Prometheus project itself and a rich collection of open-source integrations and tools. The heart of the ecosystem is the Prometheus server that we'll see in more detail in the next chapter. Also available is Alertmanager, which provides an alerting manager and engine for Prometheus.

The Prometheus project also includes a collection of exporters, used to instrument applications and services and to expose relevant metrics on an endpoint for scraping. Common tools—like web servers, databases, and the like—are supported by core exporters. Many other exporters are available open source from the Prometheus community.

Prometheus also published a collection of client libraries, used for instrumenting applications and services written in a number of languages. These include common choices like Python, Ruby, Go, and Java. Additional client libraries are also available from the open-source community.

Useful Prometheus links

- The Prometheus home page.
- The Prometheus documentation.
- Prometheus organization on GitHub.

- Prometheus source code GitHub.
- Prometheus and time series at scale presentation by Jamie Wilkinson.
- Grafana.

Summary

In this chapter we've been introduced to Prometheus. We also walked through the Prometheus architecture, data model, and other aspects of the ecosystem.

In the next chapter, we'll install Prometheus, configure it, and collect our first metrics.

Chapter 3

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 2.3.1.

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 3.1: Download the Prometheus tarball

```
$ cd /tmp  
$ wget https://github.com/prometheus/prometheus/releases/  
download/v2.3.1/prometheus-2.3.1.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 3.2: Unpack the prometheus binary

```
$ tar -xzf prometheus-2.3.1.linux-amd64.tar.gz  
$ sudo cp prometheus-2.3.1.linux-amd64/prometheus /usr/local/bin/  
$ sudo cp prometheus-2.3.1.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 3.3: Checking the Prometheus version on Linux

```
$ prometheus --version
prometheus, version 2.3.1 (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 3.4: Creating a directory on Windows

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

Now download Prometheus from the GitHub site:

Listing 3.5: Prometheus Windows download

```
https://github.com/prometheus/prometheus/releases/download/v2.3.1/prometheus-2.3.1.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 3.6: Setting the Windows path

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

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

Listing 3.7: Checking the Prometheus version on Windows

```
C:\> prometheus.exe --version
prometheus, version 2.3.1 (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 3.8: 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 3.9: 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 3.10: Checking the Prometheus version on Mac OS X

```
$ prometheus --version
prometheus, version 2.3.1 (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 3.11: 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 3.12: 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 3.13: 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 3.14: 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 3.15: 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 3.16: 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 3.17: 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 3.18: 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 3.19: 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 3.20: 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.

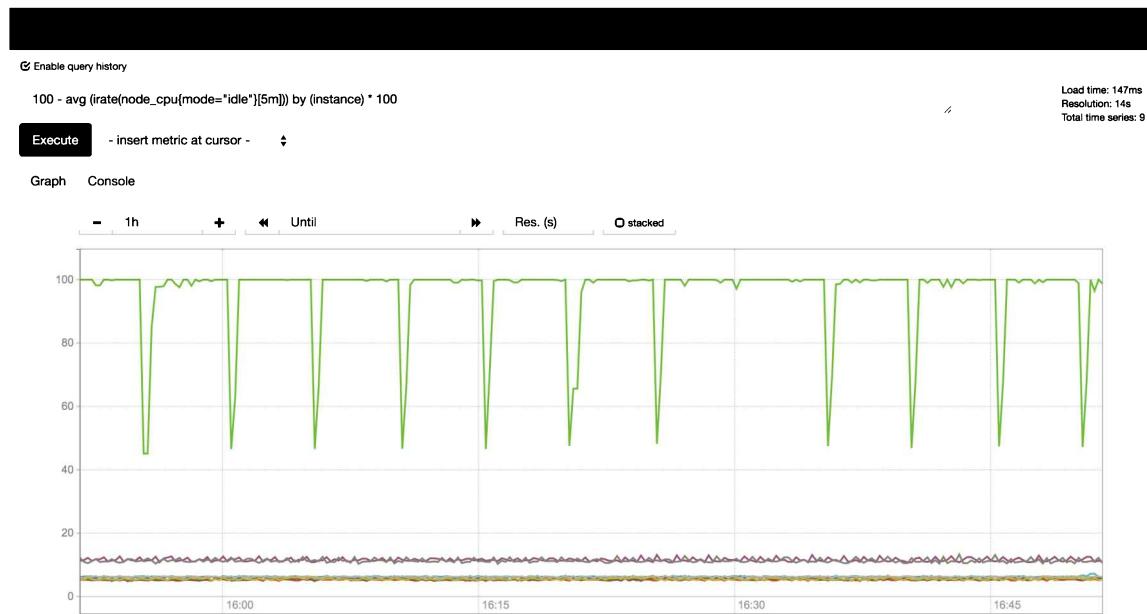


Figure 3.1: Prometheus expression browser

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.

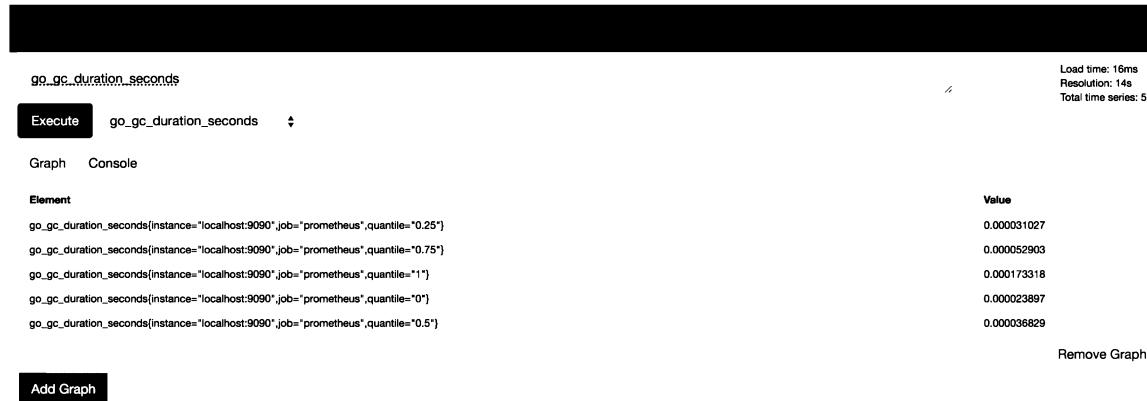


Figure 3.2: List of metrics

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 3.21: 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 3.3: 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 3.22: The `prometheus_build_info` metric

```
prometheus_build_info{branch="HEAD",goversion="go1.9.1",instance  
="localhost:9090",job="prometheus",revision="5  
ab8834befbd92241a88976c790ace7543edcd59",version="2.3.1"}
```

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

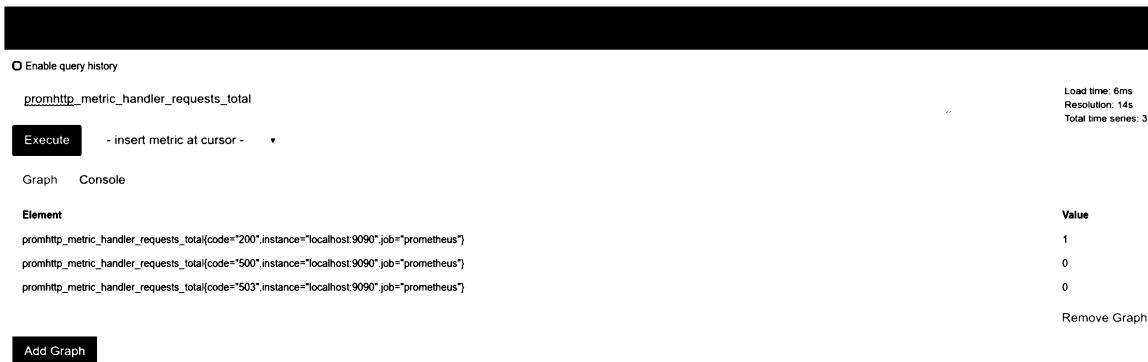


Figure 3.4: Querying total HTTP requests

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:

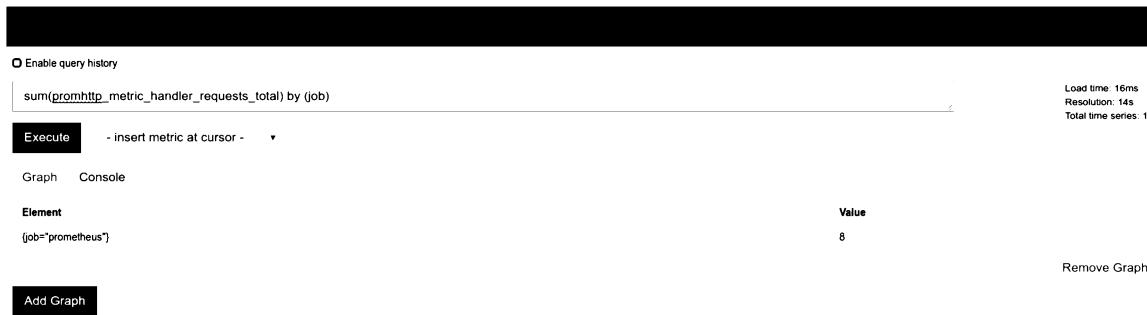


Figure 3.5: Calculating total HTTP requests by job

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 querying 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 scrap. 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.

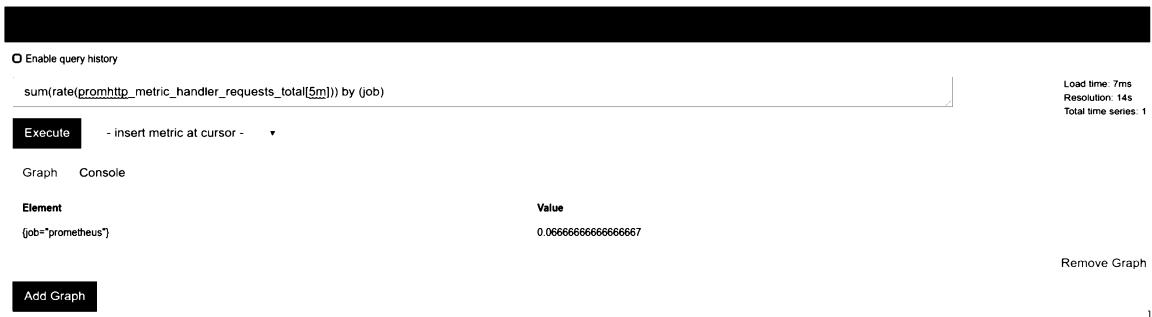


Figure 3.6: Our rate query

Cool! We've now got a new metric that is actually useful for tracking or graphing.

TIP If you want help constructing PromQL queries, there's a query editor called Promeditor available that you can run locally with Prometheus.

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.

Chapter 4

Monitoring Nodes and Containers

In the last chapter we installed Prometheus and did some basic configuration. We also scraped some time series data from the Prometheus server itself. In this chapter, we're going to look at using Prometheus to monitor the metrics of both hosts and containers. We're going to demonstrate this on a cluster of three Ubuntu hosts running the Docker daemon.

First, we'll install exporters on each host, configure exporting of node and Docker metrics, and configure Prometheus to scrape them.

Next, we'll look at monitoring some basic host resources, including:

1. CPU.
2. Memory.
3. Disk.
4. Availability.

To determine what to monitor, we'll revisit the USE Method monitoring methodology to help assist in identifying the right metrics. We'll also look at how we might use Prometheus to detect the state of services and the availability of hosts.

Then we'll make use of the collected metrics to build some aggregated metrics and save them as recording rules.

Last, we'll very briefly introduce Grafana to do basic visualizations of some of the data we're collecting.

These are probably the most standard tasks for which monitoring tools are deployed, and they provide a solid foundation for learning more about Prometheus. This base set of data will allow us to identify host performance issues or will provide sufficient supplemental data for the fault diagnosis of application issues.

Monitoring nodes

Prometheus uses tools called exporters to expose metrics on hosts and applications. There are a number of exporters available for a variety of purposes. Right now we're going to focus on one specific exporter: the Node Exporter. The Node Exporter is written in Go and has a library of collectors for various host metrics including CPU, memory, and disk. It also has a `textfile` collector that allows you to export static metrics, which is useful for sending information about the node, as we'll see shortly, or metrics exported from batch jobs.



NOTE We'll use the term "node" at times to refer to hosts.

Let's start by downloading and installing the Node Exporter on a Linux host. We're going to choose one of our Docker daemon hosts.



TIP If you don't want to use one of the Prometheus exporters there are a swath of host-monitoring clients that support Prometheus. For example, `collectd` can also write Prometheus metrics.

Installing the Node Exporter

The Node Exporter is available as a tarball and for a limited number of platforms via packages. The tarball of the Node Exporter is available, with a number of other exporters, from the Prometheus website.

Let's download and extract the Node Exporter for Linux and move the binary into our path.

Listing 4.1: Downloading the Node Exporter

```
wget https://github.com/prometheus/node_exporter/releases/download/v0.16.0/node_exporter-0.16.0.linux-amd64.tar.gz
$ tar -xzf node_exporter-*
$ sudo cp node_exporter-* /node_exporter /usr/local/bin/
```

 **NOTE** At the time of writing the Node Exporter was at version 0.16.0. You should download the latest version.

The Node Exporter is also available as a CentOS and Fedora package via a COPR build.

 **NOTE** Using configuration management is the best way to run and install any Prometheus exporters. This is an easy way to control configuration, and to provide automation and service management.

Let's test that the `node_exporter` binary is working.

Listing 4.2: Testing the Node Exporter binary

```
$ node_exporter --version
node_exporter, version 0.16.0 (branch: HEAD, revision: 6
e2053c557f96efb63aef3691f15335a70baaffd)
.
```

Configuring the Node Exporter

The `node_exporter` binary is configured via flags. You can see a full list of flags by running the binary with the `--help` flag.

Listing 4.3: Running the help for Node Exporter

```
$ node_exporter --help
```

You'll see a list of available flags. The `node_exporter` runs, by default, on port 9100 and exposes metrics on the `/metrics` path. You can control the interface and port via the `--web.listen-address` and `--web.telemetry-path` flags like so:

Listing 4.4: Controlling the port and path

```
$ node_exporter --web.listen-address=:9600 --web.telemetry-
path="/node_metrics"
```

This will bind the `node_exporter` to port 9600 and return metrics on the `/node-metrics` path.

These flags also control which collectors are enabled. By default, many of the collectors are enabled. Collectors either have a disabled or enabled status, and

the status can be flipped by specifying the relevant flag with a `no-` prefix. For example, the `arp` collector, which exposes statistics from `/proc/net/arp`, is enabled by default. It is controlled by the `--collector.arp` flag. To disable this collector we'd run:

Listing 4.5: Disabling the arp collector

```
$ node_exporter --no-collector.arp
```

Configuring the Textfile collector

We also want to configure one specific collector, the `textfile` collector, that we're going to use later in this chapter. The `textfile` collector is very useful because it allows us to expose custom metrics. These custom metrics might be the result of tasks like batch or cron jobs, which can't be scraped; they might come from sources that don't have an exporter; or they might even be static metrics which provide context for the host.

The collector works by scanning files in a specified directory, extracting any strings that are formatted as Prometheus metrics, and exposing them to be scraped.

Let's set the collector up now, starting with creating a directory to hold our the metric definition files.

Listing 4.6: Creating a textfile directory

```
$ mkdir -p /var/lib/node_exporter/textfile_collector
```

Now let's create a new metric in this directory. Metrics are defined in files ending in `.prom` inside the directory we've just created. Metrics are defined using the Prometheus text exposition format.

 **NOTE** The text exposition format allows us to specify all the metric types that Prometheus supports: counters, gauges, timers, etc.

Let's use this format to create a metric that will contain some metadata about this host.

```
metadata{role="docker_server",datacenter="NJ"} 1
```

We can see we have a metric name, `metadata`, and two labels. One label is called `role` to define a role for this node. In this case this label has a value of `docker_server`. We also have a label called `datacenter` to define the geographical location of the host. Finally, the metric has a static value of 1 because it's providing context rather than recording a counter, gauge, or timer.

Let's add this metric to a file called `metadata.prom` in our `textfile_collector` directory.

Listing 4.7: A metadata metric

```
$ echo 'metadata{role="docker_server",datacenter="NJ"} 1' | sudo tee /var/lib/node_exporter/textfile_collector/metadata.prom
```

Here we've piped our metric into a file called `metadata.prom`.

 **TIP** In the real world, you'd populate this file using your configuration management tool. For example, when a new host is provisioned, a `metadata` metric could be created from a template. This could allow you to automatically classify your hosts and services.

To enable the `textfile` collector we don't need to set a flag—it's loaded by default—but we do need to specify our `textfile_exporter` directory so the Node Exporter knows where to find our custom metrics. To do this, we specify the `--collector.textfile.directory` flag.

Enabling the `systemd` collector

Let's also turn on an extra collector, `systemd`, which records services and system status from `systemd`. This collector gathers a lot of metrics, but we don't want to collect the status of everything `systemd` is managing, just some key services. To keep things clean, we can whitelist specific services. We're only going to collect metrics for:

- `docker.service`
- `ssh.service`
- `rsyslog.service`

Which are the Docker daemon, the SSH daemon, and the RSyslog daemon. We do this using the `--collector.systemd.unit-whitelist` flag, which takes a regular expression matching `systemd` units.

Running the Node Exporter

Finally, we can launch `node_exporter` on one of our Docker nodes like so:

Listing 4.8: Starting Node Exporter with the `textfile` collector and `systemd`

```
$ node_exporter --collector.textfile.directory /var/lib/
node_exporter/textfile_collector --collector.systemd --collector.
systemd.unit-whitelist="(docker|ssh|rsyslog).service"
```

We've specified the directory for the `textfile` collector to find our metrics, enabled the `systemd` collector, and used a regular expression whitelist to match the three services for which we want to collect metrics.

Now that the Node Exporter is running on one of our Docker daemon nodes, let's add it to the others. We have three nodes, and we've identically configured two of them. The name and IP address of each node is:

- Docker1 - 138.197.26.39
- Docker2 - 138.197.30.147
- Docker3 - 138.197.30.163

Now let's see how to scrape the time series data that we've just exported.

Scraping the Node Exporter

Back on our Prometheus server, let's configure a new job to scrape the data exported by the Node Exporter. Let's examine the `scrape_configs` block from our current `prometheus.yml` file and our existing `scrape` configuration.

Listing 4.9: The current Prometheus scrape configuration

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

To get this new data, we need to add another job to this configuration. We're going to call our new job `node`. We're also going to continue to add individual targets using `static_configs`, rather than by using any kind of service discovery. (We'll see more about service discovery in the next chapter.) Let's add that new job now.

Listing 4.10: Adding the node job

```
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
  - job_name: 'node'
    static_configs:
      - targets: ['138.197.26.39:9100', '138.197.30.147:9100', '138.197.30.163:9100']
```

You can see that we've added the new job called `node`. It contains a `static_configs` block with a list of our three Docker hosts listed via their IP addresses and the relevant port, 9100. Prometheus assumes the Node Exporter has the default path, `/metrics`, and scrapes a target of:

`138.197.26.39:9100/metrics`

If we now SIGHUP or restart the Prometheus server, our configuration will be reloaded and the server will start scraping. We'll see the time series data start flowing into the Prometheus server shortly.

Filtering collectors on the server

The Node Exporter can return a lot of metrics though, and perhaps you don't want to collect them all. In addition to controlling which collectors the Node Exporter runs locally via local configuration, Prometheus also has a way we can limit the collectors actually scraped from the server side. This is especially useful when you don't control the configuration of the host you're scraping.

Prometheus achieves this by adding a list of the specific collectors to scrape to our job configuration.

Listing 4.11: Filtering collectors

```
scrape_configs:  
  . . .  
  - job_name: 'node'  
    static_configs:  
      - targets: ['138.197.26.39:9100', '138.197.30.147:9100', '  
138.197.30.163:9100']  
      params:  
        collect[]:  
          - cpu  
          - meminfo  
          - diskstats  
          - netdev  
          - netstat  
          - filefd  
          - filesystem  
          - xfs  
          - systemd
```

Here we've limited the metrics being scraped to this list of collectors, specified using the `collect[]` list inside the `params` block. These are then passed to the `scrape` request as URL parameters. You can test this using the `curl` command on a Node Exporter instance.

Listing 4.12: Testing collect params

```
$ curl -g -X GET http://138.197.26.39:9100/metrics?collect[]=cpu
```

This would return the base Node Exporter metrics, like the Go metrics we saw for the Prometheus server, and the metrics generated by the CPU collector. All other metrics will be disregarded.

For now though, on our Prometheus server, we're going to collect everything.

Now that we have our node metrics, let's instrument our Docker daemons too.

Monitoring Docker

There are several ways to monitor Docker with Prometheus, including several custom exporters. However these exporters have generally been deprecated in favor of the recommended approach: Google's cAdvisor tool. cAdvisor runs as a Docker container on your Docker daemon. A single cAdvisor container returns metrics for your Docker daemon and all running containers. It has native Prometheus support to export metrics, as well as support for a variety of other storage destinations like InfluxDB, Elasticsearch, and Kafka.

 **NOTE** We're going to assume that you have installed and are running Docker daemons, and that you understand the basics of how Docker works. If you're new to Docker, I have a book on it that might interest you.

Running cAdvisor

As cAdvisor is just another container on our Docker host, we can launch it with the `docker run` command. Let's run a cAdvisor container on our Docker1 host.

Listing 4.13: Running the caAdvisor container

```
$ docker run \
--volume=/:/rootfs:ro \
--volume=/var/run:/var/run:rw \
--volume=/sys:/sys:ro \
--volume=/var/lib/docker:/var/lib/docker:ro \
--volume=/dev/disk:/dev/disk:ro \
--publish=8080:8080 \
--detach=true \
--name=cadvisor \
google/cadvisor:latest
```

Let's break this `docker run` command down a little. First, we mount a few directories inside the container. The directories are broken into two types. The first are read-only mounts from which cAdvisor will gather data—for example, mounting the `/sys` directory like so:

```
--volume=/sys:/sys:ro
```

 **TIP** The `ro` indicates read-only.

The second type, which contains one mount, is a read-write mount of the Docker socket, usually located in the `/var/run` directory. We also publish port 8080 from inside the container to 8080 on the host. You could override this with any port that suited you. We run the container with the `--detach` flag to daemonize it and name it `cadvisor`. Last, we use the `google/cadvisor` image with the `latest` tag.

If we now run `docker ps`, we can see our running cAdvisor container.

Listing 4.14: The cAdvisor container

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS		NAMES		
6fca3002e351	google/cadvisor	<code>"/usr/bin/..."</code>	1 hours ago	Up 1 hours
0.0.0.0:8080->8080/tcp		cadvisor		

cAdvisor should start monitoring immediately. We can browse to port 8080 on the host to see cAdvisor's web interface and confirm that it is operational.

← ⌂ ⓘ 138.197.30.163:8080/containers/



/

root

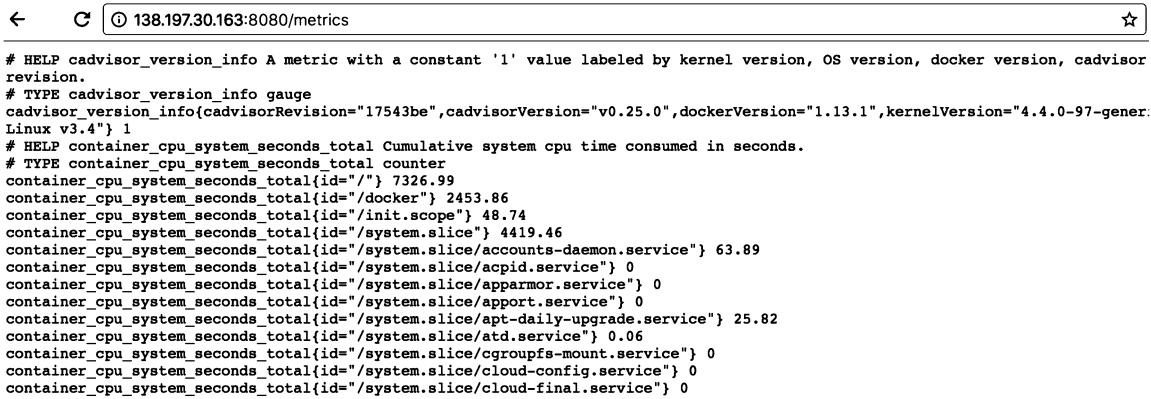
Docker Containers

Subcontainers

/docker

Figure 4.1: cAdvisor web interface

If we browse to the path `/metrics` on port 8080, we'll see the built-in Prometheus metrics being exposed.



```
# HELP cadvisor_version_info A metric with a constant '1' value labeled by kernel version, OS version, docker version, cadvisor revision.
# TYPE cadvisor_version_info gauge
cadvisor_version_info{cadvisorRevision="17543be",cadvisorVersion="v0.25.0",dockerVersion="1.13.1",kernelVersion="4.4.0-97-gener-Linux v3.4"} 1
# HELP container_cpu_system_seconds_total Cumulative system cpu time consumed in seconds.
# TYPE container_cpu_system_seconds_total counter
container_cpu_system_seconds_total{id="/" } 7326.99
container_cpu_system_seconds_total{id="/docker" } 2453.86
container_cpu_system_seconds_total{id="/init.scope" } 48.74
container_cpu_system_seconds_total{id="/system.slice" } 4419.46
container_cpu_system_seconds_total{id="/system.slice/accounts-daemon.service" } 63.89
container_cpu_system_seconds_total{id="/system.slice/acpid.service" } 0
container_cpu_system_seconds_total{id="/system.slice/apparmor.service" } 0
container_cpu_system_seconds_total{id="/system.slice/apport.service" } 0
container_cpu_system_seconds_total{id="/system.slice/apt-daily-upgrade.service" } 25.82
container_cpu_system_seconds_total{id="/system.slice/atd.service" } 0.06
container_cpu_system_seconds_total{id="/system.slice/cgrafs-mount.service" } 0
container_cpu_system_seconds_total{id="/system.slice/cloud-config.service" } 0
container_cpu_system_seconds_total{id="/system.slice/cloud-final.service" } 0
```

Figure 4.2: cAdvisor Prometheus metrics

We'll also install cAdvisor on our other two Docker daemons as well.

Scraping cAdvisor

With cAdvisor running on our Docker daemons, we need to tell Prometheus about it. To do this, we're going to add a third job to our configuration. Let's edit `prometheus.yml` on our Prometheus server.

We're again going to add individual targets using `static_configs`, rather than by using any kind of service discovery.

Listing 4.15: Adding the Docker job

```
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
  - job_name: 'node'
    static_configs:
      - targets: ['138.197.26.39:9100', '138.197.30.147:9100', '138.197.30.163:9100']
  - job_name: 'docker'
    static_configs:
      - targets: ['138.197.26.39:8080', '138.197.30.147:8080', '138.197.30.163:8080']
```

You can see we've added the new job called `docker`. It contains a `static_configs` block with a list of our three Docker daemon servers with their IP addresses and the relevant port, 8080. Again we assume the default `/metrics` path. If we again SIGHUP or restart the Prometheus server, then our configuration will be reloaded, it will start scraping, and the new time series will appear.

I think it's important, though, before we continue, that we understand how a scrape works and a bit about the lifecycle of labels. We'll use our cAdvisor metrics to explore this lifecycle.

Scrape lifecycle

Let's look at the lifecycle of a scrape itself, and into the lifecycle of labels. Every `scrape_interval` period, in our case 15 seconds, Prometheus will check for jobs to be executed. Inside those jobs it'll generate a list of targets: the service discovery process. In the cases we've seen so far we've got manually specified, statically configured hosts. There are other service discovery mechanisms, like loading targets from a file or querying an API.

 **TIP** We'll learn more about service discovery in Chapter 5.

Service discovery returns a list of targets with a set of labels attached called metadata. These labels are prefixed with `_meta_`. Each service discovery mechanism has different metadata—for example, the AWS EC2 discovery mechanism returns the availability zone of instances in a label called `_meta_ec2_availability_zone`.

Service discovery also sets additional labels based on the configuration of the target. These configuration labels are prefixed and suffixed with `_`. They include the `_scheme_`, `_address_`, and `_metrics_path_` labels. These contain the scheme, `http` or `https`, of the target; the address of the target; and the specific path to the metrics respectively.

Each label usually has a default—for example, `_metrics_path_` would default to `/metrics`, and `_scheme_` to `http`. Additionally, if any URL parameters are present in the path then they're set into labels prefixed with `_param_*`.

The configuration labels are also reused during the lifecycle of the scrape to populate other labels. For example, the default contents of the `instance` label on our metrics is the contents of the `_address_` label.

 **NOTE** So, wait—why haven't we seen any of those `_` prefixed and suffixed labels? That's because some are removed later in the lifecycle, and all of them are specifically excluded from display on the Web UI.

This list of targets and labels are then returned to Prometheus. Some of those labels can be overridden in configuration—for example, the metrics path via the `metrics_path` parameter, and the scheme to be used via the `scheme` parameter.

Listing 4.16: Overriding the discovered labels

```
scrape_configs:
  - job_name: 'node'
    scheme: https
    metrics_path: /moremetrics
    static_configs:
      - targets: ['138.197.26.39:9100', '138.197.30.147:9100', '138.197.30.163:9100']
```

Here we're overriding the scheme to `https` and the metric's path to `/moremetrics`.

Prometheus then offers an opportunity to relabel your targets and to potentially make use of some metadata your service discovery has added. You can also filter targets to drop or keep specific items.

After this, the actual scrape takes place, and the metrics are returned. When the metrics are being scraped you are offered a final opportunity to relabel and filter them before they are saved to the server.

Phew. That's complicated. Let's see a simplified image of that lifecycle:

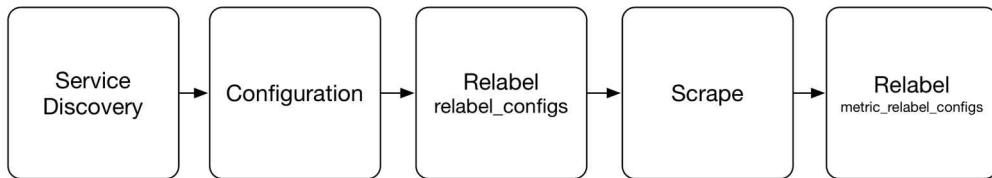


Figure 4.3: Scrape lifecycle

You can see we've introduced a bunch of concepts, including two blocks where Prometheus relabels metrics. This is a good time to talk a bit more about labels, relabelling, and taxonomies. Let's take a little interlude.

Labels

We learned in Chapter 2 that labels provide the dimensions of our time series. They can define what the target is and add context to the time series. But most importantly, combined with the metric name, they make up the identity of your time series. They represent the identity of your time series—if they change, so does the identity of the time series.

Changing a label or adding a new label creates a new time series.

This means that labels should be used judiciously and remain as constant as possible. Failure to adhere to this can spawn new time series, creating a dynamic data environment that makes your monitoring data sources harder to track. Imagine you have a time series that you're using to track the state of a service. You have an alert configured for that time series that relies on the labels of the metric to determine the right criteria. By changing or adding a label, that alert definition is rendered invalid. The same applies to historical time series data: By changing or adding a label we lose track of the previous time series, breaking graphs and expressions, and causing general mayhem.

 **TIP** What happens to the old time series if it's not being written anymore? If a scrape no longer returns data for a time series that was previously present, that series will be marked as stale. The same applies for any targets that are removed: All of their time series will be marked as stale. Stale data is not returned in graphs.

Label taxonomies

So when should we add labels and what labels should we add? Well, like all good monitoring architectures, it's worth building a taxonomy. Labels, like most monitoring taxonomies, are probably best when broadly hierarchical. A good way of thinking about a taxonomy is in terms of topological and schematic labels.

The topological labels slice the service components by their physical or logical makeup, e.g., the datacenter label we saw above. We already get two topological labels for free with every metric: `job` and `instance`. The `job` label is set from the job name in the scrape configuration. We tend to use `job` to describe the type of thing we're monitoring. In the case of our Node Exporter job we called it `node`

. This will label all the Node Exporter metrics with a job label of node. The instance label identifies the target. It's usually the IP address and port of the target, and it's usually sourced from the `_address` label.

Schematic labels are things like `url`, `error_code`, or `user` which let you match time series at the same level in the topology together—for example, to create ratios of one against the other.

If you need to add additional labels consider a hierarchy something like this:

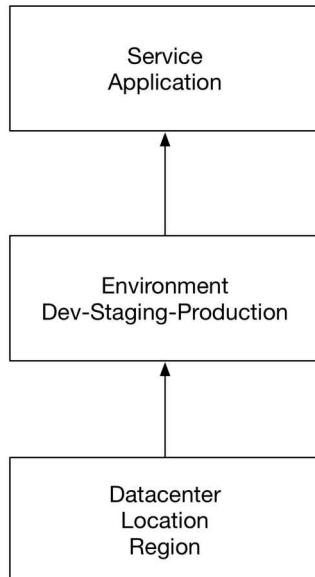


Figure 4.4: Sample label taxonomy

A little later in this chapter, we'll look at metrics like the `metadata` metric we created earlier with the Textfile collector that can be decorated with contextual information.

We can also create and manipulate existing labels to help us better manage our time series data.

Relabelling

Given the desire to judiciously use labels, why would we relabel things? In a word: control. In a centralized, complex monitoring environment you sometimes don't control all the resources you are monitoring and the monitoring data they expose. Relabelling allows you to control, manage, and potentially standardize metrics in your environment. Some of the most common use cases are:

- Dropping unnecessary metrics.
- Dropping sensitive or unwanted labels from the metrics.
- Adding, editing, or amending the label value or label format of the metrics.

Remember there are two phases at which we can relabel. The first phase is relabelling targets that have come from service discovery. This is most useful for applying information from metadata labels from service discovery into labels on your metrics. This is done in a `relabel_configs` block inside a job. We'll see more of that in the next chapter.

The second phase is after the scrape but before the metric is saved in the storage system. This allows us to determine what metrics we save, what we drop, and what those metrics will look like. This is done in the `metric_relabel_configs` block in our job.

 **TIP** The easiest way to remember the two phases are: `relabel_configs` happens **before** the scrape and `metric_relabel_configs` happens **after** the scrape.

Let's take a look at some relabelling of our cAdvisor metrics. cAdvisor collects a lot of data. Not all of it is always useful. So let's see how we might drop some of these metrics before they hit our storage and take up unnecessary space.

Listing 4.17: Dropping metrics with relabelling

```
- job_name: 'docker'
  static_configs:
    - targets: ['138.197.26.39:8080', '138.197.30.147:8080', '138.197.30.163:8080']
      metric_relabel_configs:
        - source_labels: [__name__]
          regex: '(container_tasks_state|container_memory_failures_total)'
          action: drop
```

Here we have our docker job. After our static_configs block we've added a new block: metric_relabel_configs. Inside the block we specify a series of relabelling actions.

Dropping metrics

Let's look at our first action. We select the metrics we want to take action on using the source_labels parameter. This takes an array of label names. In our case we're using the __name__ label. The __name__ label is a reserved label used for the name of a metric. So our source label for our docker job in this case would be the concatenated names of all the metrics scraped from cAdvisor.

Multiple labels are concatenated together using a separator, by default ;. The separator can be overridden using the separator parameter.

Listing 4.18: Specifying a new separator

```
...
metric_relabel_configs:
  - source_labels: [__name__]
    separator: ','
    regex: '(container_tasks_state|container_memory_failures_total)'
    action: drop
```

Here our `__name__` label values would be separated with a `,`.

Next, we specify a regular expression to search our concatenated metric names and match specific names. The regular expression uses the RE2 expression syntax, which is what the Go regular expression's library RegExp uses.

 **TIP** Suck at regular expressions? You're not alone. There are some good expression testers available online.

Our regular expression, contained in the `regex` parameter, is:

```
(container_tasks_state|container_memory_failures_total)
```

Which will match and capture two metrics:

- `container_tasks_state`
- `container_memory_failures_total`

If we had specified multiple source labels we would specify each regular expression using the separator, for example:

```
regex1;regex2;regex3
```

We then perform an action, specified in the `action` parameter. In this case, both of these metrics contain a significant number of time series—of potentially limited usefulness—so we’re taking the `drop` action. This will drop the metrics before storage. Other actions include `keep`, which keeps the metrics that match the regular expression and drops all others.

Replacing label values

We can also replace a label’s value with a new value. Let’s take an example. Many cAdvisor metrics have an `id` label that contains the name of the running process. If that process is a container we’ll see something like:

```
id="/docker/6fca3002e3513d23ed7e435ca064f557ed1d4226ef788e771b8f933a49d55804  
"
```

This is a bit unwieldy. So we’d like to take the container ID:

```
6fca3002e3513d23ed7e435ca064f557ed1d4226ef788e771b8f933a49d55804
```

And put it into a new label: `container_id`. Using relabelling we can do this like so:

Listing 4.19: Replacing a label

```
metric_relabel_configs:  
- source_labels: [id]  
  regex: '/docker/([a-z0-9]+);'  
  replacement: '$1'  
  target_label: container_id
```

 **TIP** Relabelling is applied sequentially, using top-down ordering in the configuration file.

Our source label is `id`. We then specify a regex to match and capture the container ID. The `replacement` field holds the new value, in this case our capture group `$1`. We then specify the destination for the captured information, here `container_id`, in the `target_label` parameter.

You'll notice we didn't specify an action for this relabel. This is because the default action is `replace`. If you don't specify an action, Prometheus will assume you want to perform a replacement.

Prometheus also has a parameter, `honor_labels`, that controls the conflict behavior if you try to overwrite and attach a label that already exists. Let's say your scraped data already has a label called `job`. Using the default behavior, in which `honor_labels` is set to `false`, Prometheus will rename the existing label by prefixing it with `exported_`. So our `job` label would become `exported_job`. If `honor_labels` is set to `true` then Prometheus will keep the label on the scraped data and ignore any relabelling on the server.

Dropping labels

In our last example, we're going to drop a label. This is often useful for hiding sensitive information or simplifying a time series. In this (somewhat contrived) example we're going to remove the `kernelVersion` label, hiding the kernel version of our Docker hosts.

Listing 4.20: Dropping a label

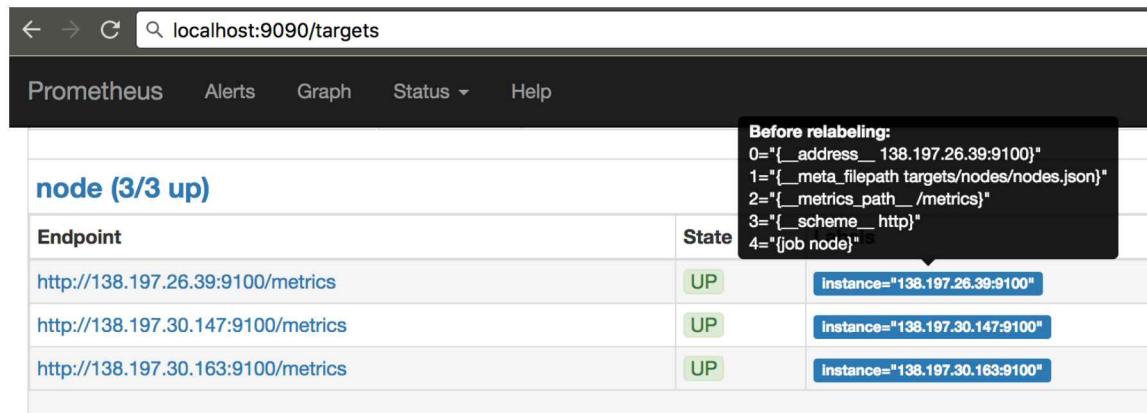
```
metric_relabel_configs:  
  - regex: 'kernelVersion'  
    action: labeldrop
```

For dropping a label, we specify a regex watching our label and then the `labeldrop` action. This will remove all labels that match the regular expression.

This action also has an inverse action, `labelkeep`, which will keep all labels that match the regular expression and drop all others.

⚠️ WARNING Remember that labels are uniqueness constraints for time series. If you drop a label and that results in duplicate time series, you will have issues!

Usefully, you can see the state of labels prior to relabelling in the Prometheus web interface. We can see this in the list of targets at `http://localhost:9090/targets`. Hover your mouse over the `instance` label in the `Labels` box to see a list of the labels as they were before relabelling.



The screenshot shows the Prometheus web interface at `localhost:9090/targets`. The page title is "node (3/3 up)". Below the title, there is a table with three columns: "Endpoint", "State", and "Labels". The "Labels" column contains a tooltip for the first row, which lists the original labels before relabelling:

Endpoint	State	Labels
<code>http://138.197.26.39:9100/metrics</code>	UP	Before relabeling: 0=" <code>__address__</code> 138.197.26.39:9100" 1=" <code>__meta_filepath</code> targets/nodes/nodes.json" 2=" <code>__metrics_path__</code> /metrics" 3=" <code>__scheme__</code> http" 4=" <code>[job node]</code> "
<code>http://138.197.30.147:9100/metrics</code>	UP	<code>instance="138.197.30.147:9100"</code>
<code>http://138.197.30.163:9100/metrics</code>	UP	<code>instance="138.197.30.163:9100"</code>

Figure 4.5: Labels prior to relabelling

Now let's take a closer look at our new metrics.

The Node Exporter and cAdvisor metrics

We're now collecting seven individual sets of metrics from four unique hosts:

- The Prometheus server's own metrics.
- Node Exporter metrics from three hosts.
- cAdvisor metrics from three hosts.

 **TIP** You can see the status of each target being scraped by looking at the Prometheus web interface. Browse to `http://localhost:9090/targets` to see a list of what Prometheus is scraping and the status of each.

Let's skip over the Prometheus server's own metrics and focus on the Node Exporter and cAdvisor metrics. Let's use some of these metrics to explore the capabilities of Prometheus and ensure our hosts are properly monitored.

The trinity and the USE method

We're going to make use of one of the monitoring frameworks we introduced at the start of the book: the USE Method. You'll remember this method suggests collecting and focusing on utilization, saturation, and error metrics to assist with performance diagnostics. We're going to apply this method, broadly, to one of the common monitoring patterns—CPU, memory, and disk—to see how we can make use of our Node Exporter metrics and how PromQL can be used.

 **TIP** Remember, these host metrics are useful mostly as broad signals of performance trouble on your hosts. We're using them to learn more about working with metrics. Most of the time, though, we're going to focus on application metrics, which are better indicators of poor user experience.

Let's start by looking at CPU metrics.

CPU Utilization

To get the U-for-utilization in USE, we're going to use a metric the Node Exporter collects named `node_cpu_seconds_total`. This is the utilization of the CPUs on our host, broken down by mode and presented in seconds used. Let's query for that metric now from the Prometheus web interface. Navigate to `http://localhost:9090/graph`, select `node_cpu_seconds_total` from the metric dropdown and click Execute.

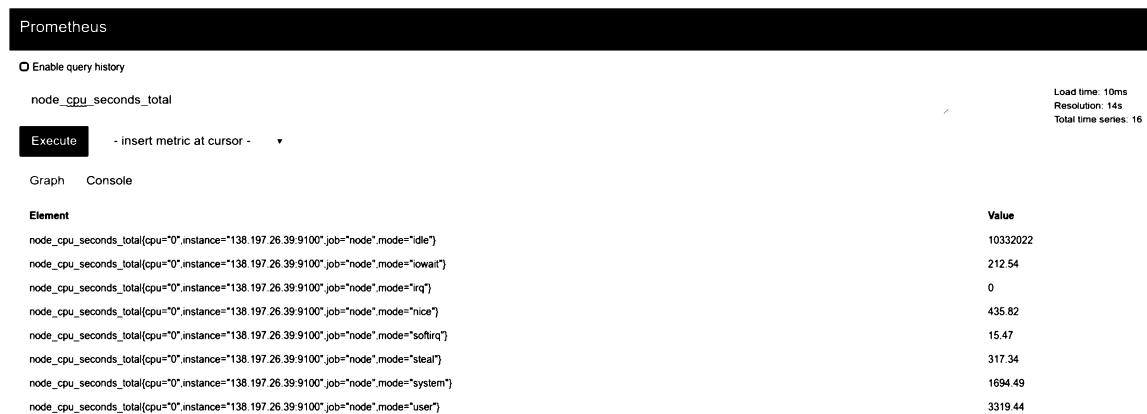


Figure 4.6: `node_cpu_seconds_total` metrics

You should see a list of metrics, much like so:

```
node_cpu_seconds_total{cpu="cpu0",instance="138.197.26.39:9100",job="node",mode="user"}
```

The `node_cpu_seconds_total` metric has a number of labels including the instance and job labels, which identify what host it came from and what job scraped the metric, respectively.

NOTE The `instance` label is generally made up of the address of the host and the port that was scraped.

We also have two labels specific to CPUs: the `cpu` the metric was collected from—for example, `cpu0`—and `mode` for the CPU mode being measured—for example, `user`, `system`, `idle`, etc. Drawn from `/proc/stat`, the data are counters that tell us how many seconds each CPU spent in each mode.

This list of metrics isn't overly useful as is. For any performance analysis, we'll need to make use of PromQL to turn these into useful metrics. What we'd *really* like here is to get the percentage CPU used on each instance—but to get there we'll need to work with our metrics a little. Let's step towards this outcome by looking at a sequence of PromQL calculations.

We start with calculating the per-second rate for each CPU mode. PromQL has a function called `irate` that calculates the per-second instant rate of increase of a time series in a range vector. Let's use the `irate` function over our `node_cpu_seconds_total` metric. Enter this into the query box:

```
irate(node_cpu_seconds_total{job="node"}[5m])
```

And click Execute. This wraps the `node_cpu_seconds_total` metric in the `irate` function and queries a five-minute range. It'll return the list of per-cpu, per-mode metrics from the `node` job, now represented as per-second rates in a five-minute range. But this still isn't overly helpful—we need to aggregate our metrics across CPUs and modes too.

To do this, we can use the `avg` or `average` operator and the `by` clause we saw in Chapter 3.

```
avg(irate(node_cpu_seconds_total{job="node"}[5m])) by (instance)
```

Now we've wrapped our `irate` function inside an `avg` aggregation and added a `by` clause that aggregates by the `instance` label. This will produce three new metrics that average CPU usage by host using the values from all CPUs and all modes.

But this metric is still not quite right. It still includes idle usage, and it isn't represented in a useful form like a percentage. Let's constrain our calculation by querying only the per-instance idle usage and, as it's already a ratio, multiplying it by 100 to convert it into a percentage.

```
avg (irate(node_cpu_seconds_total{job="node",mode="idle"}[5m])) by (instance) * 100
```

Here we've added the mode label with a value of idle to our `irate` query. This only queries the idle data. We've averaged the result by instance and multiplied it by 100. Now we have the average percentage of idle usage in a five-minute range on each host. We can turn this into the percentage used by subtracting this value from 100, like so:

```
100 - avg (irate(node_cpu_seconds_total{job="node",mode="idle"}[5m]))  
by (instance) * 100
```

And now we have three metrics, one for each host, showing the average percentage CPU used in a five-minute window.



Figure 4.7: Per-host average percentage CPU usage metrics

Now, if we click the Graph tab, we can also see these represented as a plot.

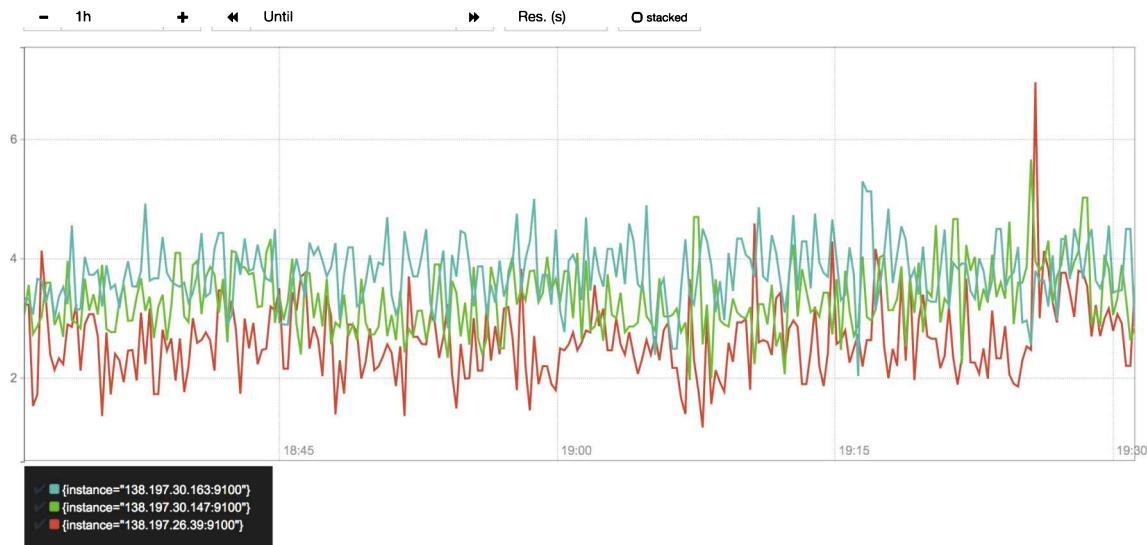


Figure 4.8: Per-host percentage CPU plot

CPU Saturation

One of the ways to get the saturation of CPU on a host is to track the load average, essentially the average run queue length over a time period, taking into consideration the number of CPUs on the host. An average less than the number of CPUs is generally normal; averages over that number for prolonged periods indicate the CPU is saturated.

To see the host's load average, we can use the `node_load*` metrics for these. They show load average over one minute, five minutes, and 15 minutes. We're going to use the one-minute load average: `node_load1`.

Let's take a quick look at this metric. Select `node_load1` from the metric dropdown and click Execute. A list of the nodes being monitored with the current one-minute load average will be listed.

We also need to calculate the number of CPUs on our hosts. We can do this using the `count` aggregation like so:

```
count by (instance)(node_cpu_seconds_total{mode="idle"})
```

Here we're counting the number of occurrences of the `node_cpu_seconds_total` time series with a mode of `idle`. We're then using the `by` clause to remove all labels except `instance` from the result vector, giving us a list of hosts with the number of CPUs in each.

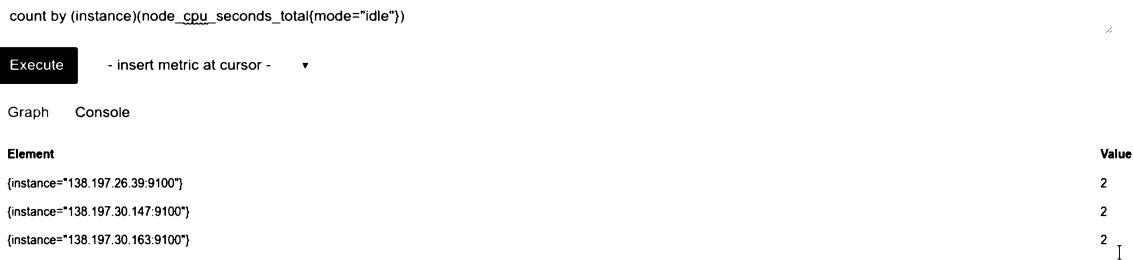


Figure 4.9: Number of CPUs in each host

We can see our three nodes have two CPUs apiece.

TIP Since we're also collecting Docker metrics we could use one of cAdvisor's metrics here too, `machine_cpu_cores`, as a shortcut.

We can then combine this count with the `node_load1` metric like so:

```
node_load1 > on (instance) 2 * count by (instance)(node_cpu_seconds_total  
{mode="idle"})
```

Here we're showing if the one-minute load average is two times more than the CPU count on the host. This is not necessarily an issue, but we'll see in Chapter 6 how to turn it into an alert that should tell you when there is an issue.

Now let's see if we can't do something similar with our memory metrics.

TIP We're going to skip the E-for-error in USE for CPU errors because it's

unlikely there will be anything useful in any data we could collect.

Memory utilization

Let's look at the utilization of memory on a host. The Node Exporter's memory metrics are broken down by type and usage of memory. You'll find them in the list of metrics prefixed with `node_memory`.



Figure 4.10: The `node_memory_MemTotal_bytes`

We're going to focus on a subset of the `node_memory` metrics to provide our utilization metric:

- `node_memory_MemTotal_bytes` - The total memory on the host.
- `node_memory_MemFree_bytes` - The free memory on the host.
- `node_memory_Buffers_bytes` - The memory in buffer cache.
- `node_memory_Cached_bytes` - The memory in the page cache.

All of these metrics are represented in bytes.

We're going to use this combination of metrics to calculate the percentage of memory used on each host. To do this, we're going to add the values of the `node_memory_MemFree_bytes`, `node_memory_Cached_bytes`, and `node_memory_Buffers_bytes` metrics. This represents the free memory on our

host. We're then going to calculate the percentage of free memory using this value and the `node_memory_MemTotal_bytes` metric. We're going to use this query to do that:

```
(node_memory_MemTotal_bytes - (node_memory_MemFree_bytes + node_memory_Cached_bytes
+ node_memory_Buffers_bytes)) / node_memory_MemTotal_bytes * 100
```

Here we've added together our three memory metrics, subtracted them from the total, divided by the total, and then multiplied by 100 to convert it into a percentage. This will produce three metrics showing percentage memory used per host.

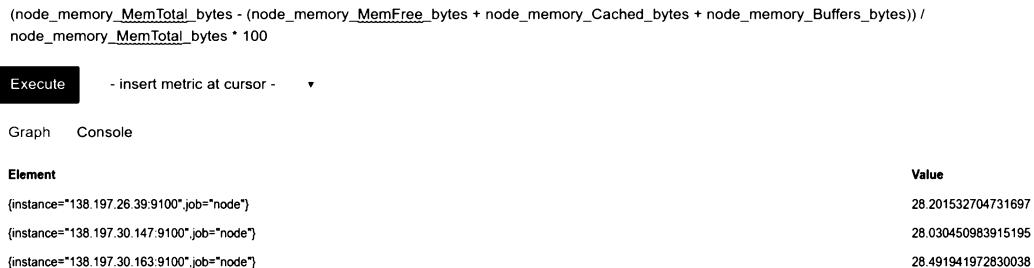


Figure 4.11: Per-host percentage memory usage

Here we don't need to use the `by` clause to preserve distinct dimensions because the metrics have the same dimensional labels; each metric will have the query applied to it in turn.

Memory saturation

We can also monitor our memory saturation by checking on the rate of paging in and out of memory. We can use data gathered from `/proc/vmstat` on paging exposed in two Node Exporter metrics:

- `node_vmstat_pswpin` - Number of kilobytes the system has paged in from disk per second.

- node_vmstat_pswpout - Number of kilobytes the system has paged out to disk per second.

Both are totals in kilobytes since last boot.

To get our saturation metric, we generate a one-minute rate for each metric, add the two rates, and then multiply them by 1024 to get bytes. Let's create a query to do this now.

Listing 4.21: Memory saturation query

```
1024 * sum by (instance) (
  rate(node_vmstat_pgpgin[1m])
  + rate(node_vmstat_pgpgout[1m]))
)
```

We can then graph or alert on this to identify hosts with misbehaving applications.

Disk usage

For disks we're only going to measure disk usage rather than utilization, saturation, or errors. This is because it's the most useful data in most cases for visualization and alerting. The Node Exporter's disk usage metrics are in the list of metrics prefixed with `node_filesystem`.

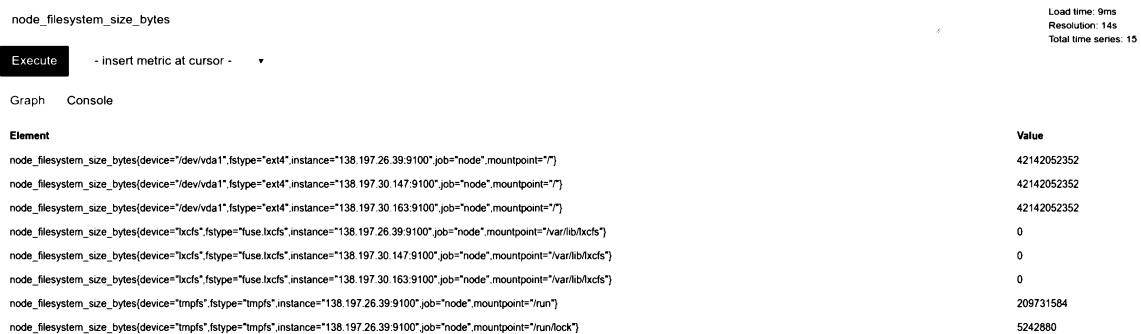


Figure 4.12: Disk metrics

Here, for example, the `node_filesystem_size_bytes` metric shows the size of each file system mount being monitored. We can use a similar query to our memory metrics to produce a percentage figure of disk space used on our hosts.

```
(node_filesystem_size_bytes{mountpoint="/" } - node_filesystem_free_bytes{mountpoint="/" }) / node_filesystem_size_bytes{mountpoint="/" } * 100
```

Unlike the memory metrics, though, we have filesystem metrics per mount point on each host. So we've added the `mountpoint` label, specifically the `/` filesystem mount. This will return a disk usage metric for that filesystem on each host being monitored.

<pre>(node_filesystem_size_bytes{mountpoint="/" } - node_filesystem_free_bytes{mountpoint="/" }) / node_filesystem_size_bytes{mountpoint="/" } * 100</pre>	
Execute	- insert metric at cursor - ▾
Graph	Console
Element	Value
{device="/dev/vda1",fstype="ext4",instance="138.197.26.39:9100",job="node",mountpoint="/"}	36.875714809040346
{device="/dev/vda1",fstype="ext4",instance="138.197.30.147:9100",job="node",mountpoint="/"}	33.88453633137378
{device="/dev/vda1",fstype="ext4",instance="138.197.30.163:9100",job="node",mountpoint="/"}	35.00321278325197

Figure 4.13: Per-host disk space metrics

If we wanted or needed to, we could add additional queries for specific mount points to the configuration now. To monitor a mount point called `/data` we would use:

```
(node_filesystem_size_bytes{mountpoint="/data"} - node_filesystem_free_bytes{mountpoint="/data"}) / node_filesystem_size_bytes{mountpoint="/data"} * 100
```

Or we could use a regular expression to match more than one mountpoint.

```
(node_filesystem_size_bytes{mountpoint=~"/|/run"} - node_filesystem_free_bytes{mountpoint=~"/|/run"}) / node_filesystem_size_bytes{mountpoint=~"/|/run"} * 100
```

 **TIP** You cannot use a regular expression that matches an empty string.

You can see that we've updated our `mountpoint` label to change the operator from `=` to `==` which tells Prometheus that the right-hand-side value will be a regular expression. We've then matched both the `/run` and `/root` filesystems.

 **TIP** There's also a `~` operator for regular expressions that do not match.

This is still a fairly old-school measure of disk usage. It tells us a current percentage usage of the filesystem. In many cases, this data is useless. An 80 percent full 1 GB filesystem might not be a concern at all if it's growing at 1 percent a year. A 10 percent full 1 TB filesystem might be at serious risk of filling up if it's growing at 10 percent every 10 minutes. With disk space, we really need to understand the trend and direction of a metric. The question we usually want answered is: "Given the usage of the disk now, combined with its growth, in what time frame will we run out of disk space?"

Prometheus actually has a mechanism, a function called `predict_linear`, by which we can construct a query to answer this exact question. Let's look at an example:

```
predict_linear(node_filesystem_free_bytes{mountpoint="/"}[1h], 4*3600)
< 0
```

Here we're grabbing the root filesystem, `node_filesystem_free_bytes{mountpoint="/"}`. We could select all the filesystems by specifying the job name or selectively using a regular expression, as we did earlier in this section.

```
predict_linear(node_filesystem_free_bytes{job="node"}[1h], 4*3600) < 0
```

We have selected a one-hour time window, [1h]. We've also placed this time series snapshot inside the `predict_linear` function. The function uses simple linear regression to determine when a filesystem will run out of space based on previous growth. The function takes a range vector, our one-hour window, and the point in the future for which to predict the value, measured in seconds. Hence, four times 3600 (the number of seconds in an hour), or four hours. The `< 0` filters for values less than 0, i.e., the filesystem running out of space.

So, if, based on the last hour's worth of growth history, the filesystem is going to run out of space in the next four hours, the query will return a negative number, which we can then use to trigger an alert. We'll see how this alert would work in Chapter 6.

Service status

Now let's look at the data from the `systemd` collector. Remember this shows us the state of services and various other `systemd` configuration on our hosts. The state of the services is exposed in the `node_systemd_unit_state` metric. There's a metric for each service and service state you're collecting. In our case we're only gathering metrics for the Docker, SSH, and RSyslog daemons.

Listing 4.22: The `node_systemd_unit_state` metrics

```
node_systemd_unit_state{name="docker.service",state="activating"
} 0
node_systemd_unit_state{name="docker.service",state="active"} 1
node_systemd_unit_state{name="docker.service",state=
"deactivating"} 0
node_systemd_unit_state{name="docker.service",state="failed"} 0
node_systemd_unit_state{name="docker.service",state="inactive"} 0
...
.
```

We can query a segment of this data via the Expression Browser and look specifically for this docker service. To do this, we query using the name label.

```
node_systemd_unit_state{name="docker.service"}
```

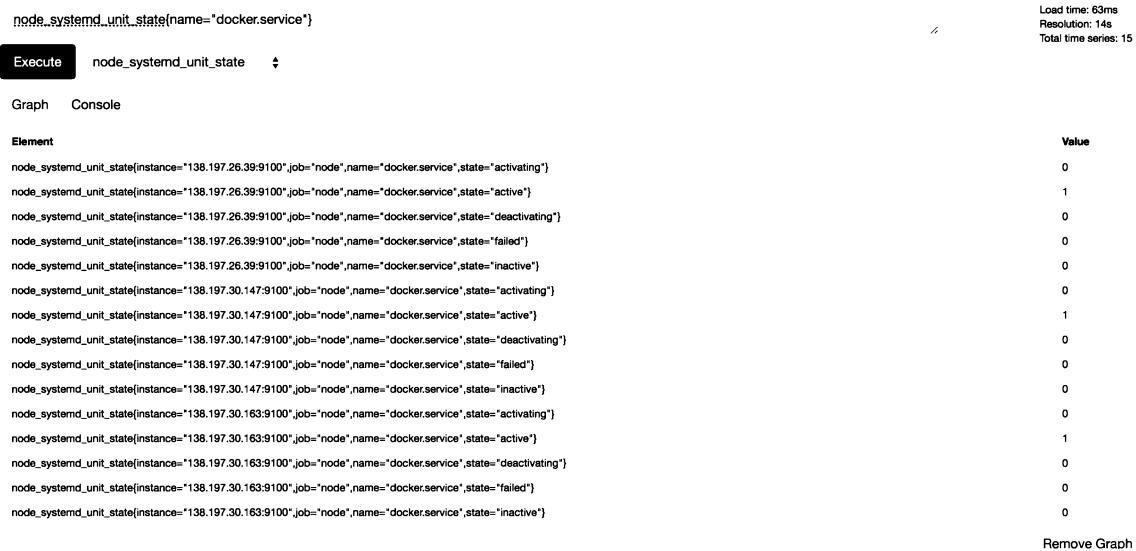


Figure 4.14: The systemd time series data

This query produces a metric for each combination of potential service and state: failed, inactive, active, etc. The metric that represents the current state of each service is set to 1. We could narrow this down further by adding the state label to our search and only returning the active state.

```
node_systemd_unit_state{name="docker.service",state="active"}
```

Alternatively, we could search for all of the metrics with the value 1, which would return the state of the current service.

```
node_systemd_unit_state{name="docker.service"} == 1
```

Here we've seen a new query, one that uses a comparison binary operator: ==. This will retrieve all metrics with a value equal to 1 with a name label of docker.service.



Figure 4.15: The active services

We’re going to make use of the systemd metrics to monitor the availability of services on our host—for example, our Docker daemon—and alert on this in Chapter 6.

Availability and the up metric

Worth mentioning is another useful metric for monitoring the state of specific nodes: the `up` metric. For each instance scrape, Prometheus stores a sample in the following time series:

Listing 4.23: The up metric

```
up{job="<job-name>", instance="<instance-id>"}
```

The metric is set to 1 if the instance is healthy—i.e., the scrape successfully returned—or to 0 if the scrape failed. The metric is labelled with the job name and the instance of the time series.

TIP Prometheus also populates some other instrumentation metrics, including `scrape_duration_seconds`, the duration of the scrape, and `scrape_samples_scraped`, the number of samples that the target exposed.

We can query all the `up` metrics for our hosts.



Figure 4.16: The `up` metrics

In addition, many exporters have specific metrics designed to identify the last successful scrape of a service. The cAdvisor metrics include `container_last_seen`, for example, which provides a list of containers and the last time they were active. The MySQL Exporter returns a metric, `mysql_up`, that is set to 1 if a successful SELECT query works on a database server.

We'll see in Chapter 6 how we can use the `up` metrics to help us do availability monitoring.

NOTE You cannot relabel autopopulated metrics like `up` because they are generated after the relabelling phase.

The metadata metric

Last, let's look at the metric we created, `metadata`, using the Node Exporter's Textfile collector.

```
metadata{role="docker_server",datacenter="NJ"} 1
```

This metric provides context for the resource: its role, `docker_server`, and the location of the host, `datacenter`. This data is useful in its own right, but why create a separate metric rather than just add these as labels to all of our metrics from this host? Well, we already know that labels provide the dimensions of our time series and, combined with the metric name, they make up the identity of our time series. We've also already been warned that:

Changing a label or adding a new label creates a new time series.

This means that labels should be used judiciously and should remain as constant as possible. So, instead of decorating every time series with the set of complete labels, we instead create a time series that we can use to query specific types or classes of resources.

Let's see how we could make use of the labels on this metric. Suppose we want to select metrics only from a specific data center or set of data centers. We can quickly find all hosts in, say, a non-New Jersey (NJ) data center by querying like so:

```
metadata{datacenter != "NJ"}
```

You can see that we've queried the `metadata` metric and specified the `datacenter` with an operator, `!=` or not equal to, to return any `metadata` metric from a non-New Jersey data center.

 **TIP** Prometheus has a full set of arithmetic and comparison binary operators you can use.

Vector matches

We can also use our `metadata` metric to make vector matches. Vector matches can use any of the PromQL binary operators. Vector matches attempt to find a matching element in the right-hand-side vector for each entry on the left-hand side.

There are two kinds of vector matches: One-to-one and many-to-one (or one-to-many).

One-to-one matches

One-to-one matches find a unique pair of entries from each side of the operation. Two entries match if they have the exact same set of labels and values. You can modify the set of labels considered by using the `ignoring` modifier, which ignores specific labels, or by using the `on` modifier, which reduces the set of considered labels to a list. Let's see an example.

Listing 4.24: A one-to-one vector match

```
node_systemd_unit_state{name="docker.service"} == 1  
and on (instance, job)  
    metadata{datacenter="SF"}
```

This queries any `node_systemd_unit_state` metrics with the `name` label of `docker.service` and a value of 1. We then use the `on` modifier to reduce the considered label set to the `instance` and `job` labels of the `metadata` metric, where the `datacenter` label has a value of SF.

In our case, this will return a single metric:

```
node_systemd_unit_state{instance="138.197.30.147:9100", job="node", name="docker.service", state="active"}
```

If we were to change the datacenter label in our query to NJ, we'd return two metrics: one for each of the two Docker servers in the NJ data center.

Many-to-one and one-to-many matches

Many-to-one and one-to-many matches are where each vector element on the “one” side can match with multiple elements on the “many” side. These matches are explicitly specified using the `group_left` or `group_right` modifiers, where left or right determines which vector has the higher cardinality. The Prometheus documentation contains some examples of this kind of match, but they are generally not required. In most cases one-to-one matches will suffice.

Metadata-style metrics

Many existing exporters use this “metadata” pattern to provide information about extra state—for example, cAdvisor has the `cadvisor_version` metric that provides information about the local Docker daemon and related configuration.



Figure 4.17: The `cadvisor_version` metric

This type of metric allows you to use vector matches to list all metrics that match some contextual criteria: a location, a version, etc.

So now that we've seen how to use some of the metrics, how do we persist the queries we've seen?

Query permanence

Until now, we've just run queries in the Expression Browser. Whilst viewing the output of that query is interesting, the result is stuck on the Prometheus server and is transitory. There are three ways we can make our queries more permanent:

- Recording rules - Create new metrics from queries.
- Alerting rules - Generate alerts from queries.
- Visualization - Visualize queries using a dashboard like Grafana.

The queries we've looked at can be used interchangeably in all three of these mechanisms because all of these mechanisms can understand and execute PromQL queries.

In this chapter we're going to make use of some recording rules to create new metrics from our queries and configure Grafana as a dashboard to visualize metrics. In Chapter 6 we'll make use of alerting rules to generate alerts.

Recording rules

We talked about recording rules and their close cousin, alerting rules in Chapter 2.

Recording rules are a way to compute new time series, particularly aggregated time series, from incoming time series. We might do this to:

- Produce aggregates across multiple time series.
- Precompute expensive queries.
- Produce a time series that we could use to generate an alert.

Let's write some rules.

Configuring recording rules

Recording rules are stored on the Prometheus server, in files that are loaded by the Prometheus server. Rules are calculated automatically, with a frequency controlled by the `evaluation_interval` parameter in the `global` block of the `prometheus.yml` configuration file we saw in Chapter 3.

Listing 4.25: The `evaluation_interval` parameter

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

Rule files are specified in our Prometheus configuration inside the `rules_files` block.

Let's create a sub-directory called `rules` in the same directory as our `prometheus.yml` file, to hold our recording rules. We'll also create a file called `node_rules.yml` for our node metrics. Prometheus rules are written, like Prometheus configuration, in YAML.

⚠️ WARNING YAML rules were updated in Prometheus 2.0. Earlier releases used a different structure. Your older rule files will not work in Prometheus 2.0 or later. You can use the `promtool` to upgrade older rules files. There's a good blog post on the upgrading process [here](#).

Listing 4.26: Creating a recorded rules file

```
$ mkdir -p rules
$ cd rules
$ touch node_rules.yml
```

Let's add that file to our Prometheus configuration in the `rule_files` block in the `prometheus.yml` file.

Listing 4.27: Adding the rules file

```
rule_files:
  - "rules/node_rules.yml"
```

Now let's populate this file with some rules.

Adding recording rules

Let's convert our CPU, memory, and disk calculations into recording rules. We have a lot of hosts to be monitored, so we're going to precompute all the trinity queries. That way we'll also have the calculations as metrics that we can alert on or visualize via a dashboard like Grafana.

Let's start with our CPU calculation.

Listing 4.28: A recording rule

```
groups:  
  - name: node_rules  
    rules:  
      - record: instance:node_cpu:avg_rate5m  
        expr: 100 - avg (irate(node_cpu_seconds_total{job="node",  
mode="idle"})[5m]) by (instance) * 100
```

Recording rules are defined in rule groups; here ours is named `node_rules`. Rule group names must be unique in a server. Rules within a group are run sequentially at a regular interval. By default, this is the `global evaluation_interval`, but it can be overridden in the rule group using the `interval` clause.

The sequential nature of rule execution in groups means that you can use rules you create in subsequent rules. This allows you to create a metric from a rule and then reuse that metric in a later rule. This is only true *within* rule groups though—rule groups are run concurrently, so it's not safe to use rules *across* groups.

 **TIP** This also means you can use recording rules as parameters, for example you might want to create a rule with a threshold in it. You can then set the threshold once in the rule and re-use it multiple times. If you need to change the threshold you just need to change it that one place.

Listing 4.29: A recording group interval

```
groups:  
- name: node_rules  
  interval: 10s  
  rules:
```

This would update the rule group to be run every 10 seconds rather than the globally defined 15 seconds.

Next, we have a YAML block called `rules`, which contains this group's recording rules. Each rule contains a `record`, which tells Prometheus what to name the new time series. You should name rules so you can identify quickly what they represent. The general recommended format is:

```
level:metric:operations
```

Where `level` represents the aggregation level and labels of the rule output. `Metric` is the metric name and should be unchanged other than stripping `_total` off counters when using the `rate()` or `irate()` functions. This makes it easier to find the new metric. Finally, `operations` is a list of operations that were applied to the metric, the newest operation first.

So our CPU query would be named:

```
instance:node_cpu:avg_rate5m
```

 **TIP** There are some useful best practices on naming in the Prometheus documentation.

We then specify an `expr` field to hold the query that should generate the new time series.

We could also add a `labels` block to add new labels to the new time series. Time series created from rules inherit the relevant labels of the time series used to create them, but you can also add or overwrite labels. For example:

Listing 4.30: A recording rule

```
groups:  
- name: node_rules  
  rules:  
    - record: instance:node_cpu:avg_rate5m  
      expr: 100 - avg (irate(node_cpu_seconds_total{job="node",  
mode="idle"}[5m])) by (instance) * 100  
      labels:  
        metric_type: aggregation
```

Let's create rules for some of our other trinity queries, and add them, too.

Listing 4.31: A recording rule

```
groups:  
- name: node_rules  
  rules:  
    - record: instance:node_cpu:avg_rate5m  
      expr: 100 - avg (irate(node_cpu_seconds_total{job="node",  
mode="idle"}[5m])) by (instance) * 100  
    - record: instance:node_memory_usage:percentage  
      expr: (node_memory_MemTotal_bytes - (  
node_memory_MemFree_bytes + node_memory_Cached_bytes +  
node_memory_Buffers_bytes)) / node_memory_MemTotal_bytes * 100  
    - record: instance:root:node_filesystem_usage:percentage  
      expr: (node_filesystem_size_bytes{mountpoint="/" } -  
node_filesystem_free_bytes{mountpoint="/" }) /  
node_filesystem_size_bytes{mountpoint="/" } * 100
```

TIP The configuration files and code for this book are located on GitHub.

We now need to restart or SIGHUP the Prometheus server to activate the new rules. This will create a new time series for each rule. You should be able to find the new time series on the server in a few moments.

TIP The rule files can be reloaded at runtime by sending SIGHUP to the Prometheus process (or by restarting on Microsoft Windows). The reload will only work if the rules file is well formatted. The Prometheus server ships with a utility called `promtool` that can lint rule files.

If we now search for one of the new time series, `instance:node_cpu:avg_rate5m` for example, we should see:



Figure 4.18: The `instance:node_cpu:avg_rate5m` recorded rule

Last, let's quickly look at how we might visualize the metrics we've just created.

TIP You can see the current rules defined on your server in the `/rules` path

of the Web UI. This includes useful information, like the execution time of each rule, that can help you debug expensive rules that might need optimization.

Visualization

As we've seen, Prometheus has an inbuilt dashboard and graphing interface. It's fairly simple and generally best for reviewing metrics and presenting solitary graphs. To add a more fully featured visualization interface to Prometheus, the platform integrates with the open-source dashboard Grafana. Grafana is a dashboard fed via data sources. It supports a variety of formats including Graphite, Elasticsearch, and Prometheus.

It's important to note that Prometheus isn't generally used for long-term data retention—the default is 15 days worth of time series. This means that Prometheus is focused on more immediate monitoring concerns than, perhaps, other systems where visualization and dashboards are more important. The judicious use of the Expression Browser, graphing inside the Prometheus UI, and building appropriate alerts are often more practical uses of Prometheus' time series data than building extensive dashboards.

With that said, in this last section we're going to quickly install Grafana and connect Prometheus to it.

Installing Grafana

Installing Grafana depends on the platform you're installing on. Grafana supports running on Linux, Microsoft Windows, and Mac OS X. Let's look at installation on each platform.

Installing Grafana on Ubuntu

For Ubuntu and Debian systems, we can add the Grafana package repository. We first need to add the PackageCloud public key, like so:

Listing 4.32: Getting the PackageCloud public key on Ubuntu

```
$ curl https://packagecloud.io/gpg.key | sudo apt-key add -
```

We add the following Apt configuration so we can find the Grafana repository:

Listing 4.33: Adding the Grafana packages

```
$ echo "deb https://packagecloud.io/grafana/stable/debian/ stretch main" | sudo tee -a /etc/apt/sources.list.d/grafana.list
```

Then we update Apt and install the grafana package with the apt-get command.

Listing 4.34: Updating Apt and installing the Grafana package

```
$ sudo apt-get update  
$ sudo apt-get install grafana
```

On Red Hat

To install Grafana on Red Hat systems, we first need to add the Elastic.co public key, like so:

Listing 4.35: Getting the Grafana public key on Red Hat

```
$ sudo rpm --import https://packagecloud.io/gpg.key
```

Then we add the following to our `/etc/yum.repos.d/` directory in a file called `grafana.repo`:

Listing 4.36: The Grafana Yum configuration

```
[grafana]
name=grafana
baseurl=https://packagecloud.io/grafana/stable/el/7/$basearch
repo_gpgcheck=1
enabled=1
gpgcheck=1
gpgkey=https://packagecloud.io/gpg.key https://grafanarel.s3.amazonaws.com/RPM-GPG-KEY-grafana
sslverify=1
sslcacert=/etc/pki/tls/certs/ca-bundle.crt
```

Now we install Grafana using the `yum` or `dnf` commands.

Listing 4.37: Installing Grafana on Red Hat

```
$ sudo yum install grafana
```

Installing Grafana on Microsoft Windows

To install Grafana on Microsoft Windows, we need to download and put it in a directory. Let's create a directory for the executable using Powershell.

Listing 4.38: Creating a Grafana directory on Windows

```
C:\> MKDIR grafana  
C:\> CD grafana
```

Now download Grafana from the Grafana site.

Listing 4.39: Grafana Windows download

```
https://s3-us-west-2.amazonaws.com/grafana-releases/release/  
grafana-5.1.3.windows-x64.zip
```

The zip file contains a folder with the current Grafana version. Unzip the file using a tool like 7-Zip, and put the contents of the unzipped directory into the C:\grafana directory. Finally, add the C:\grafana directory to the path. This will allow Windows to find the executable. To do this, run this command inside Powershell.

Listing 4.40: Setting the Windows path for Grafana

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

We then need to make some quick configuration changes to adjust the default port. The default Grafana port is 3000, a port which requires extra permissions on Microsoft Windows. We want to change it to port 8080 to make Grafana easier to use.

Go into the c:\grafana\conf\ directory and copy the sample.ini file to custom.ini. Edit the custom.ini file and uncomment the http_port configuration option. It'll be prefixed with the ; character, which is the comment character in ini files.

Change the port number to 8080. That port will not require extra Microsoft Windows privileges.

Installing Grafana on Mac OS X

Grafana is also available from Homebrew. If you use Homebrew to provision your Mac OS X hosts then you can install Grafana via the `brew` command.

Listing 4.41: Installing Grafana via Homebrew

```
$ brew install grafana
```

Installing Grafana via configuration management or a stack

There are a variety of options for installing Grafana via configuration management. Many of the stacks and configuration management modules we saw in Chapter 3 also support Grafana installations.

- Chef cookbooks for Grafana at <https://supermarket.chef.io/cookbooks/grafana>.
- Puppet modules for Grafana at <https://forge.puppetlabs.com/modules?utf-8=%E2%9C%93&sort=rank&q=grafana>.
- Ansible roles for Grafana at <https://galaxy.ansible.com/list#/roles/3563>.
- Docker images Grafana at <https://hub.docker.com/search/?q=grafana>.

Starting and configuring Grafana

There are two places where we can configure Grafana: a local configuration file and the Grafana web interface. The local configuration file, which is primarily for

configuring server-level settings like authentication and networking, is available at either:

- `/etc/grafana/grafana.ini` on Linux.
- `/usr/local/etc/grafana/grafana.ini` on OS X.
- `c:\grafana\conf\custom.ini` on Microsoft Windows.

The Grafana web interface is used to configure the source of our data and our graphs, views, and dashboards. Our configuration in this chapter is going to be via the web interface.

To access the web interface we need to start the Grafana web service, so let's do that first. On Linux we'd use the service.

Listing 4.42: Starting the Grafana Server on Linux

```
$ sudo service grafana-server start
```

This will work on both Ubuntu and Red Hat.

On OS X, if we want to start Grafana at boot, we need to run:

Listing 4.43: Starting Grafana at boot on OSX

```
$ brew services start grafana
```

Or to start it ad hoc on OS X, run:

Listing 4.44: Starting Grafana server on OS X

```
$ grafana-server --config=/usr/local/etc/grafana/grafana.ini --  
homepath /usr/local/share/grafana cfg:default.paths.logs=/usr/  
local/var/log/grafana cfg:default.paths.data=/usr/local/var/lib/  
grafana cfg:default.paths.plugins=/usr/local/var/lib/grafana/  
plugins
```

On Microsoft Windows we would run the `grafana-server.exe` executable and use something like NSSM if we want to run it as a service.

Configuring the Grafana web interface

Grafana is a Go-based web service that runs on port 3000 (or 8080 on Microsoft Windows, as we configured it) by default. Once it's running you can browse to it using your web browser—for example, if it's running on the local host: `http://localhost:3000`.

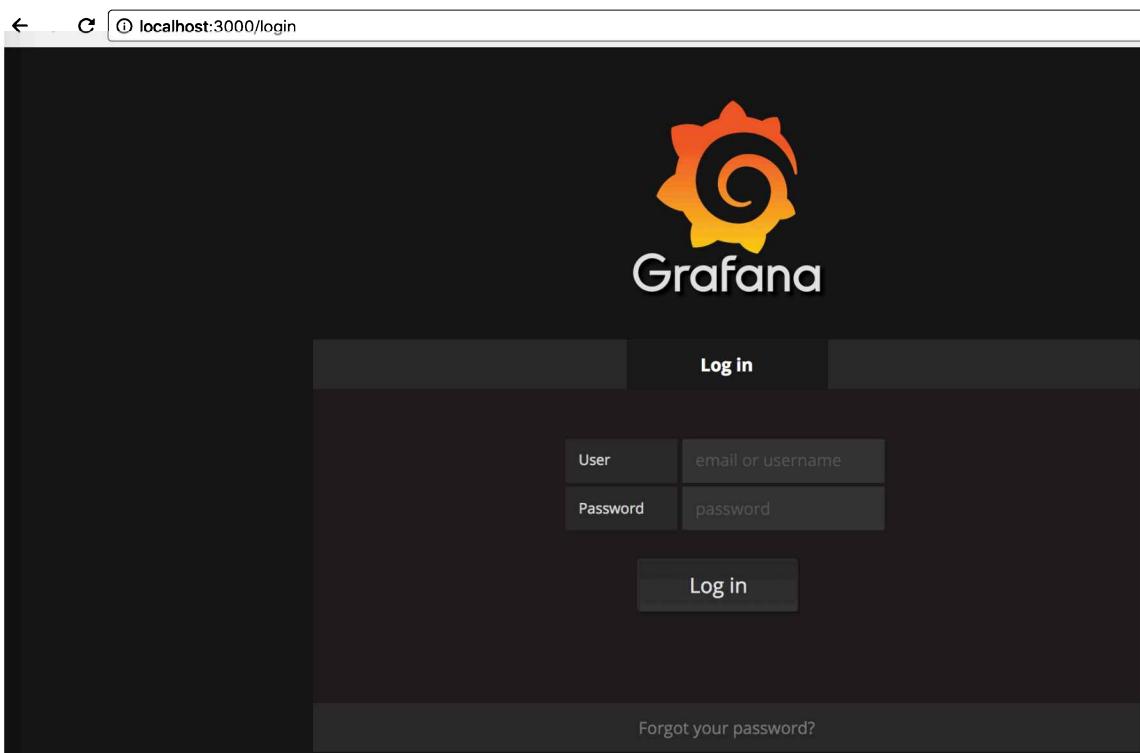


Figure 4.19: The Grafana console login

You'll see a login screen initially. The default username and password are `admin` and `admin`. You can control this by updating the `[security]` section of the Grafana configuration file.

You can configure user authentication, including integration with Google authentication, GitHub authentication, or local user authentication. The Grafana configuration documentation includes sections on user management and authentication. For our purposes, we're going to assume the console is inside our environment and stick with local authentication.

Log in to the console by using the `admin / admin` username and password pair and clicking the `Log in` button. You should see the Grafana default console view.

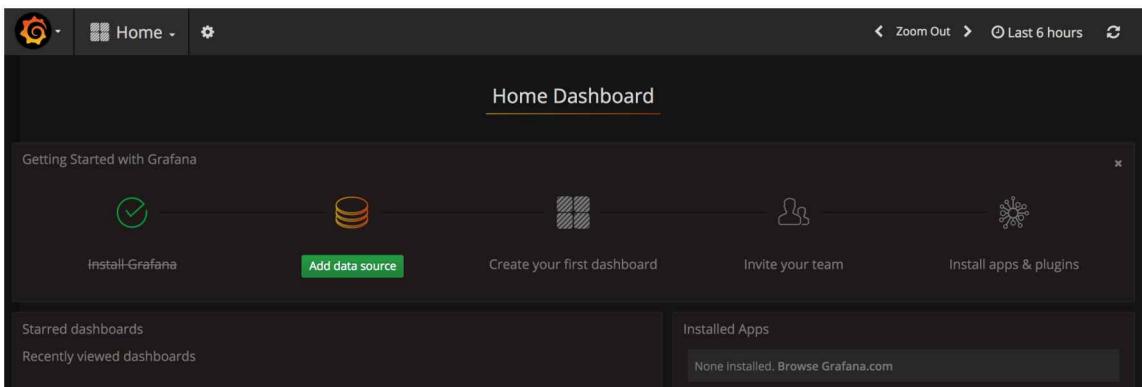


Figure 4.20: The Grafana console

It contains a *Getting Started* workflow. We first want to connect Grafana to our Prometheus data. Click on Add data source in the Getting Started workflow. You'll see a new definition for a data source.

To add a new data source we need to specify a few details. First, we need to name our data source. We're going to call ours Prometheus. Next, we need to check the Default checkbox to tell Grafana to search for data in this source by default. We also need to ensure the data source Type is set to Prometheus.

We also need to specify the HTTP settings for our data source. This is the URL of the Prometheus server we wish to query. Here, let's assume we're running Grafana on the same host as Prometheus—for our local server, it's `http://localhost:9090`. If you're running Prometheus elsewhere you'll need to specify the URL to Prometheus and to ensure connectivity is available between the Grafana host and the Prometheus server.

NOTE The Prometheus server needs to be running for Grafana to retrieve data.

We also need to set the Access option to proxy. Surprisingly this doesn't configure

an HTTP proxy for our connection, but it tells Grafana to use its own web service to proxy connections to Prometheus. The other option, direct, makes direct connections from the web browser. The proxy setting is much more practical, as the Grafana service takes care of connectivity.

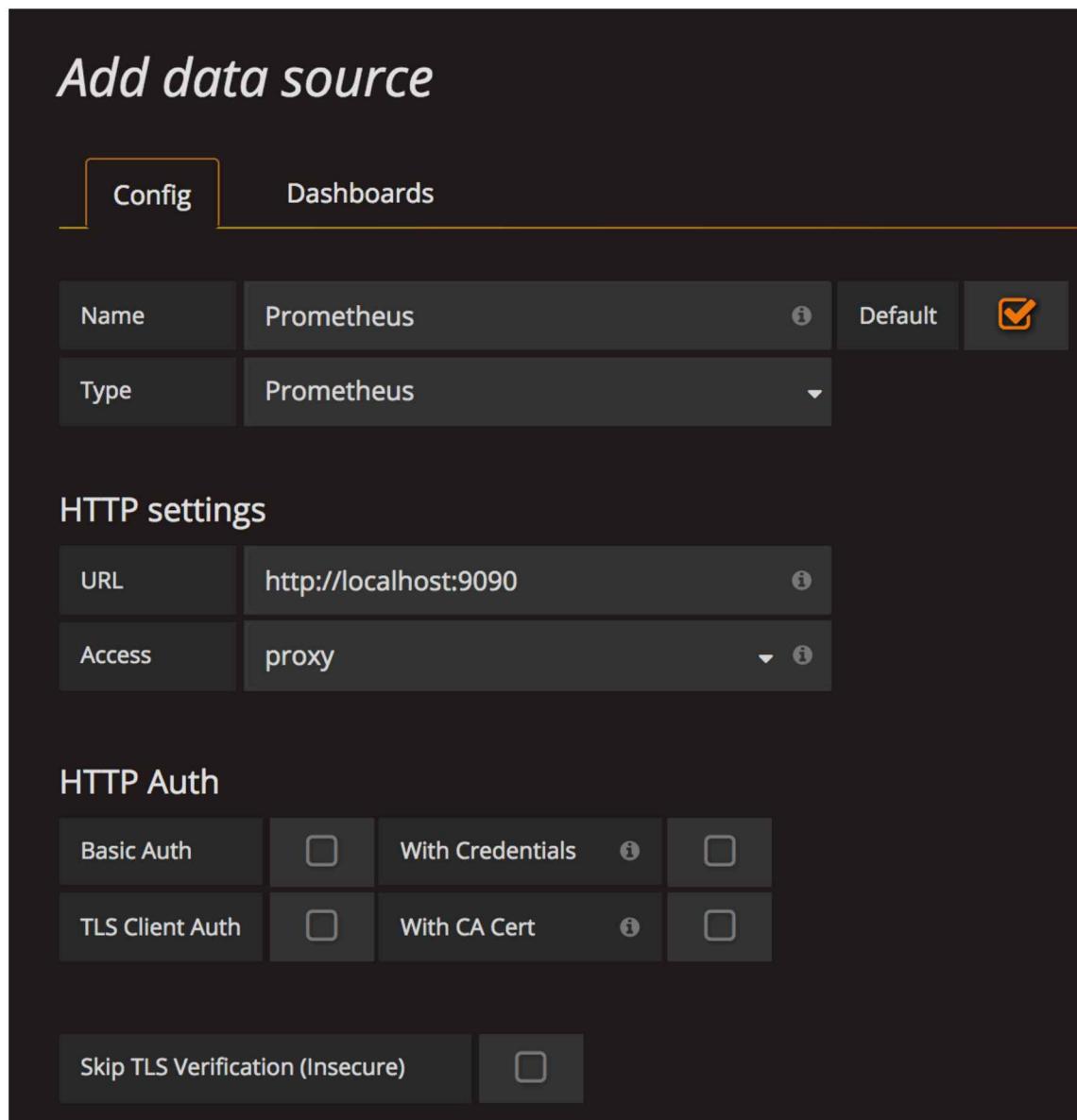


Figure 4.21: Adding a Grafana data source for Prometheus

To add our new data source, click the Add button. This will save it. On the screen we can now see our data source displayed. If it saved with a banner saying Data source is working then it is working!

Click the Grafana logo and then click on Dashboards -> Home to return to the main console view.

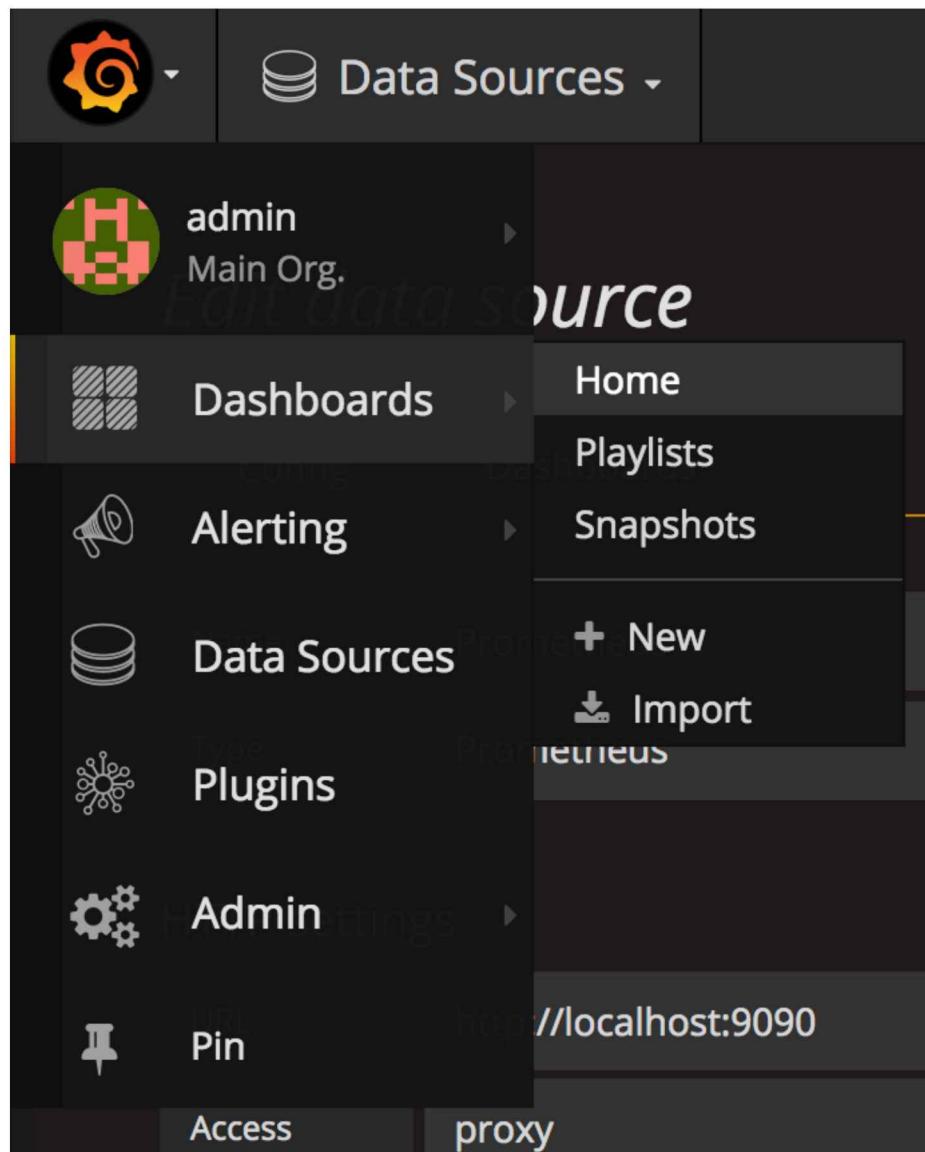


Figure 4.22: Adding a Grafana data source for Prometheus

First dashboard

Now that you're back on the Getting Started workflow, click on the New dashboard button to create a new dashboard.

You can then see the first dashboard here:

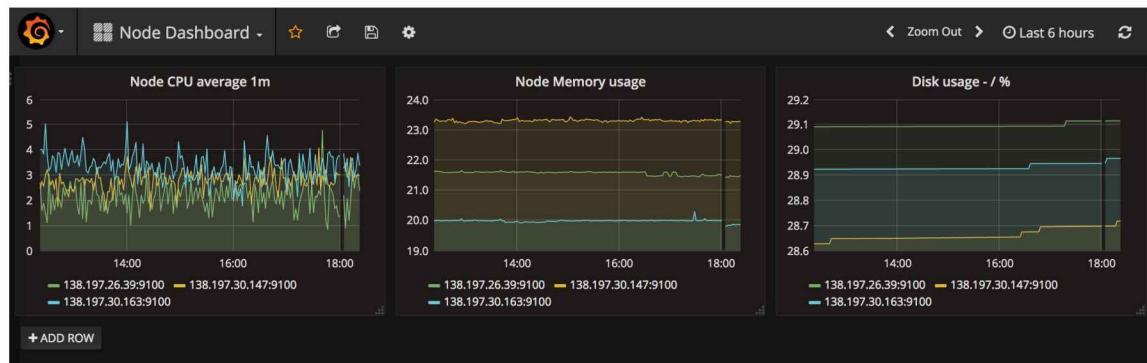


Figure 4.23: The Node dashboard

The process of creating graphs and dashboards is reasonably complex and beyond the scope of this book. But there are a large number of resources and examples that can help:

- [Grafana Getting Started](#)
- [Grafana Tutorials and screencasts](#)
- [Grafana Prometheus documentation](#)
- [Grafana Prebuilt Dashboard collection](#)

Many projects also include prebuilt Grafana dashboards for their specific needs—for example, monitoring MySQL or Redis.

You can then add graphs for some of the other metrics we've explored in this chapter. We've included the JSON for our complete dashboard in the code for the book that you can import and play with.

Summary

In this chapter we used our first exporters and scraped node and container metrics. We've started to explore the PromQL query language, how to make use of it to aggregate those metrics and report on the state of some of our node resources, and we've delved into the USE Method to find some key metrics to monitor. We also learned how to save those queries as recording rules.

From here we can extend the use of those exporters to our whole fleet of hosts. This presents a challenge, though: how does Prometheus know about new hosts? Do we continue to manually add IP addresses to our scrape configuration? We can quickly see that this will not scale. Thankfully, Prometheus has a solution: service discovery. In the next chapter we'll explore how Prometheus can discover your hosts and services.

Chapter 5

Service Discovery

In the last chapter we installed exporters and scraped node and container metrics. For each target we specified, we manually listed their IP address and port in the scrape configuration. This approach is fine for a few hosts but not for a larger fleet, especially not a dynamic fleet using containers and cloud-based instances, where the instances can change, appear, and disappear.

Prometheus solves this issue by using service discovery: automated mechanisms to detect, classify, and identify new and changed targets. Service discovery can work via a variety of mechanisms:

- Receiving lists of targets from files populated via configuration management tools.
- Querying an API, such as the Amazon AWS API, for lists of targets.
- Using DNS records to return lists of targets.

In this chapter, we're going to use service discovery to learn how to discover our hosts and services and expose them to Prometheus. We'll see a variety of discovery mechanisms including file-based, API-driven, and DNS-powered.

Scrape lifecycle and static configuration redux

To understand how service discovery works we need to harken back to our scrape lifecycle. When Prometheus runs a job, the very first step initiated is service discovery. This populates the list of targets and metadata labels that the job will scrape.

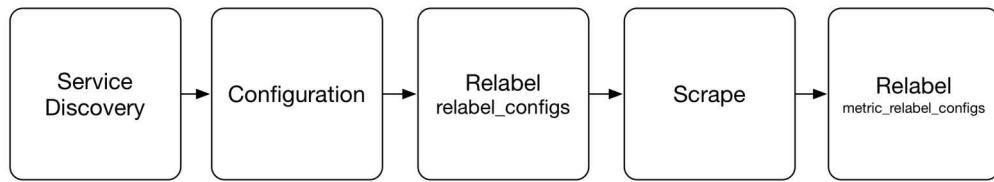


Figure 5.1: Scrape lifecycle

In our existing configuration, our service discovery mechanism is the `static_configs` block:

Listing 5.1: Our static service discovery

```
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
  - job_name: 'node'
    static_configs:
      - targets: ['138.197.26.39:9100', '138.197.30.147:9100', '138.197.30.163:9100']
```

The list of targets and any associated labels are manual service discovery. It's pretty easy to see that maintaining a long list of hosts in a variety of jobs isn't going to be a human-scalable task (nor is HUP'ing the Prometheus server for each change overly elegant). This is especially true with the dynamic nature of most environments and the scale of hosts, applications, and services that you're likely to want to monitor.

This is where more sophisticated service discovery comes into its own. So what alternatives do we have? We're going to explore several service discovery methods:

- File-based.
- Cloud-based.
- DNS-based.

We'll start with file-based discovery.

 **NOTE** Jobs can use one or more than type of service discovery. We can source targets from multiple service discovery techniques by specifying them in a job.

File-based discovery

File-based discovery is only a small step more advanced than static configurations, but it's great for provisioning by configuration management tools. With file-based discovery Prometheus consumes targets specified in files. The files are usually generated by another system—such as a configuration management system like Puppet, Ansible, or Chef—or queried from another source, like a CMDB. Periodically a script or query runs or is triggered to (re)populate these files. Prometheus then reloads targets from these files on a specified schedule.

The files can be in YAML or JSON format and contain lists of targets defined much like we'd define them in a static configuration. Let's start by moving our existing jobs to file-based discovery.

Listing 5.2: File-based discovery

```
- job_name: node
  file_sd_configs:
    - files:
        - targets/nodes/*.json
        refresh_interval: 5m

- job_name: docker
  file_sd_configs:
    - files:
        - targets/docker/*.json
        refresh_interval: 5m
  . . .
```

We've replaced the `static_configs` blocks in our `prometheus.yml` file with `file_sd_configs` blocks. Inside these blocks we've specified a list of files, contained in the `files` array. We've specified our files for each job under a parent directory, `targets`, and created a sub-directory for each job. You can create whatever structure works for you.

We've then specified the files using a glob: `*.json`. This will load targets from all files ending in `.json` in this directory, whenever those files change. I've chosen JSON for our files because it's a popular format that's easy to write using a variety of languages and integrations.

Every time the job runs or these files change, Prometheus will reload the files' contents. As a safeguard, we've also specified the `refresh_interval` option. This option will load the targets in the listed files at the end of each interval—for us this is five minutes.

 **TIP** There's also a metric called `prometheus_sd_file_mtime_seconds` that will tell you when your file discovery files were last updated. You could monitor this metric to identify any staleness issues.

Let's quickly create this directory structure.

Listing 5.3: Creating the target directory structure

```
$ cd /etc/prometheus  
$ mkdir -p targets/{nodes,docker}
```

Let's move our nodes and Docker daemons to new JSON files. We'll create two files to hold the targets.

Listing 5.4: Creating JSON files to hold our targets

```
$ touch targets/nodes/nodes.json  
$ touch targets/docker/daemons.json
```

And now populate them with our existing targets.

Listing 5.5: The nodes.json file

```
[{  
  "targets": [  
    "138.197.26.39:9100",  
    "138.197.30.147:9100",  
    "138.197.30.163:9100"  
  ]  
}]
```

And the daemons.json file.

Listing 5.6: The daemons.json file

```
[{  
  "targets": [  
    "138.197.26.39:8080",  
    "138.197.30.147:8080",  
    "138.197.30.163:8080"  
  ]  
}]
```

We can also articulate the same list of targets we've created in JSON in YAML.

Listing 5.7: The daemons file in YAML

```
- targets:  
  - "138.197.26.39:8080"  
  - "138.197.30.147:8080"  
  - "138.197.30.163:8080"
```

This moves our existing static configuration into our files. We could add labels to

these targets, too.

Listing 5.8: Adding labels

```
[{  
    "targets": [  
        "138.197.26.39:8080",  
        "138.197.30.147:8080",  
        "138.197.30.163:8080"  
    ],  
    "labels": {  
        "datacenter": "nj"  
    }  
}]
```

Here we've added the label `datacenter` with a value of `nj` to the Docker daemon targets. File-based discovery automatically adds one metadata label during the relabelling phase to every target: `__meta_filepath`. This contains the path and filename of the file containing the target.

 **NOTE** You can see a full list of the service discovery targets and their meta labels on the Web UI at <https://localhost:9090/service-discovery>.

Writing files for file discovery

Since writing files out to JSON is fairly specific to the source of the targets, we're not going to cover any specifics, but we'll provide a high-level overview of some approaches.

First, if your configuration management tool can emit a list of the nodes it is managing or configuring, that's an ideal starting point. Several of the configuration

management modules we introduced in Chapter 3 have such templates.

If those tools include a centralized configuration store or configuration management database (CMDB) of some kind, this can be a potential source for the target data. For example, if you are using PuppetDB, there's a file-based discovery script you can use to extract your nodes from the database.

Alternatively, if you're going to write your own, there's a few simple rules to follow:

- Make your file discovery configurable—don't hardcode options. Preferably, ensure that your file discovery will also work automatically with its default configuration. For instance, ensure the default configuration options assume the default installation state of the source.
- Don't expose secrets like API keys or passwords in configuration. Instead, rely on secret stores or the environment.
- Operations on the files to which you output your targets should be atomic.

Here are some file discovery scripts and tools that might provide examples you can crib from:

- Amazon ECS.
- An API-driven file discovery script that Wikimedia uses with its CMDB.
- Docker Swarm.

 **TIP** There's also a list of file-based discovery integrations in the Prometheus documentation.

Inbuilt service discovery plugins

Some tools and platforms are supported by native service discovery integrations. These ship with Prometheus. These service discovery plugins use those tools and platform's existing data stores or APIs to return lists of targets.

The currently available native service discovery plugins include platforms like:

- Amazon EC2
- Azure
- Consul
- Google Compute Cloud
- Kubernetes

 **TIP** We'll see the Kubernetes service discovery in Chapter 7 when we instrument an application running on Kubernetes.

Let's take a look at the Amazon EC2 service discovery plugin.

Amazon EC2 service discovery plugin

The Amazon EC2 service discovery plugin uses the Amazon Web Services EC2 API to retrieve a list of EC2 instances to use as Prometheus targets. In order to use the discovery plugin you'll need to have an Amazon account and credentials. We're going to assume you already have an Amazon account, but if you haven't already got an AWS account, you can create one at the AWS Console.

Then follow the Getting Started process.

As part of the *Getting Started* process you'll receive an access key ID and a secret access key. If you have an Amazon Web Services (AWS) account you should already have a pair of these.

Let's add a new job to our Prometheus configuration to retrieve our EC2 instances.

Listing 5.9: An EC2 discovery job

```
- job_name: amazon_instances
  ec2_sd_configs:
    - region: us-east-1
      access_key: AKIAIOSFODNN7EXAMPLE
      secret_key: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

We've specified a new `amazon_instances` job. Our service discovery is provided via the `ec2_sd_configs` block. Inside this block we've specified three parameters: `region` for the AWS region, and `access_key` and `secret_key` for our Amazon credentials.

If you don't want to specify your keys in the file (and, remember, you shouldn't hardcode your secrets in configuration), Prometheus supports Amazon's local CLI configuration approaches. If you don't specify keys, Prometheus will look for the appropriate environment variables, or for AWS credentials in the user running Prometheus' home directory.

Alternatively, you can specify a role ARN to use IAM roles.

Prometheus also supports profiles if you have multiple AWS accounts specified on the host.

Listing 5.10: An EC2 discovery job with a profile

```
- job_name: amazon_instances
  ec2_sd_configs:
    - region: us-east-1
      profile: work
```

Here Prometheus will use the `work` profile when discovering instances.

The discovery

The EC2 discovery plugin will return all running instances in that region. By default, it'll return targets with the private IP address of the instance, with a default port of 80, and with a metrics path of `/metrics`. So, if you have an EC2 instance with the private IP address of `10.2.1.1`, it will return a scrape target address of `http://10.2.1.1:80/metrics`. We can override the default port with the `port` parameter.

Listing 5.11: An EC2 discovery job with a port

```
- job_name: amazon_instances
  ec2_sd_configs:
    - region: us-east-1
      port: 9100
```

This will override the default port of 80 with a port of 9100.

Often, though, we want to override more than just the port. If this isn't where you have metrics exposed, we can adjust this prior to the scrape by relabelling. This relabelling takes place in the first relabel window, prior to the scrape, and uses the `relabel_configs` block.

Let's assume each of our EC2 instances has the Node Exporter configured, and we want to scrape the public IP address—not the private IP address—and relabel the targets accordingly.

Listing 5.12: Relabelling an EC2 discovery job

```
- job_name: amazon_instances
  ec2_sd_configs:
    - region: us-east-1
      access_key: AKIAIOSFODNN7EXAMPLE
      secret_key: wJalrXUtnFEMI/K7MDENG/bPxRfCYEXAMPLEKEY
  relabel_configs:
    - source_labels: [__meta_ec2_public_ip]
      regex: '(.*)'
      target_label: __address__
      replacement: '$1:9100'
```

TIP Remember, a job can have more than one type of service discovery present. For example, we could discover some targets via file-based service discovery and others from Amazon, we could statically specify some targets, and so on. Despite this, any relabelling will be applied to all targets. If you're using service discovery metadata labels they won't be available for all targets.

The configuration syntax and structure for relabelling is identical between the `relabel_configs` and `metric_relabel_configs` blocks. The only difference is when they take place: `relabel_configs` is after service discovery and before the scrape, and `metrics_relabel_configs` is after the scrape.

Here we've specified one of the metadata labels collected by the EC2 service discovery plugin, `__meta_ec2_public_ip`, as the value for our `source_labels`. We specified `(.*)` as our regular expression. We technically don't need to specify this

as this is the default value of the `regex` parameter. But we've included it to make it clear what's happening. This expression captures the entire contents of the source label.

We then specify the target for our replacement in the `target_label` parameter. As we need to update the IP address we're writing into the `__address__` label. Finally, the `replacement` parameter contains the regular expression capture from the `regex` and suffixes it with the Node Exporter default port: `9100`. If the public IP address of our instance was `34.201.102.225`, then the instance would be relabelled as a target to `34.201.102.225:9100` and the default scheme, `http`, and the metrics path, `/metrics`, would be added. The final target would be scraped at `http://34.201.102.225:9100/metrics`.

 **TIP** The `_meta` labels are dropped after the first relabelling phase, `relabel_configs`.

Other metadata collected by the EC2 discovery plugin includes:

- `__meta_ec2_availability_zone` - The availability zone of the instance.
- `__meta_ec2_instance_id` - The EC2 instance ID.
- `__meta_ec2_instance_state` - The state of the EC2 instance.
- `__meta_ec2_instance_type` - The type of the EC2 instance.
- `__meta_ec2_private_ip` - The private IP address of the EC2 instance, if available.
- `__meta_ec2_public_dns_name` - The public DNS name of the instance, if available.
- `__meta_ec2_subnet_id` - A comma-separated list of subnet IDs in which the instance is running, if available.
- `__meta_ec2_tag_<tagkey>` - Each tag value of the instance. (One label per tag.)

- `__meta_ec2_vpc_id` - The ID of the VPC in which the instance is running, if available.



TIP The full list of metadata is available in the Prometheus configuration documentation.

The `__meta_ec2_tag_<tagkey>` metadata label for EC2 tags also allows us to use relabelling to better name our targets. Rather than using the IP address for the instance label, effectively the public name of the target, we can make use of the tag values. Let's say we had an EC2 tag called `Name` that contained the hostname (or a friendly name of some sort) of the instance. We could use relabelling to make use of that tag value.

Listing 5.13: Relabelling the instance name in a EC2 discovery job

```
- job_name: amazon_instances
  ec2_sd_configs:
    - region: us-east-1
  relabel_configs:
    - source_labels: [__meta_ec2_public_ip]
      regex: '(.*)'
      target_label: __address__
      replacement: '$1:9100'
    - source_labels: [__meta_ec2_tag_Name]
      target_label: instance
```

You can see that we've added a second relabel that uses the `__meta_ec2_tag_Name` label, which contains the value of the `Name` tag as the source label, and writes it into the `instance` label. Assuming the `Name` tag, for instance `10.2.1.1`, contained `bastion`, then our `instance` label would be relabelled from:

```
node_cpu_seconds_total{cpu="cpu0",instance="10.2.1.1",job="nodes",mode="system"}
```

to:

```
node_cpu_seconds_total{cpu="cpu0",instance="bastion",job="nodes",mode="system"}
```

Making it easier to parse metrics from that target.

DNS service discovery

If file discovery doesn't work for you, or your source or service doesn't support any of the existing service discovery tools, then DNS discovery may be an option. DNS discovery allows you to specify a list of DNS entries and then query records in those entries to discover a list of targets. It relies on querying A, AAAA, or SRV DNS records.

 **TIP** The DNS records will be resolved by the DNS servers that are defined locally on the Prometheus server—for example, `/etc/resolv.conf` on Linux.

Let's look at a new job that uses DNS service discovery.

Listing 5.14: DNS service discovery job

```
- job_name: webapp
  dns_sd_configs:
    - names: [ '_prometheus._tcp.example.com' ]
```

We've defined a new job called `webapp` and specified a `dns_sd_configs` block.

Inside that block we've specified the `names` parameter which contains an array of the DNS entries we're going to query.

By default, Prometheus's DNS service discovery assumes you're querying SRV or Service records. Service records are a way to define services in your DNS configuration. A service generally consists of one or more target host and port combinations upon which your service runs. The format of a DNS SRV entry looks like:

Listing 5.15: A SRV record

```
_service._proto.name. TTL IN SRV priority weight port target.
```

Where `_service` is the name of the service being queried, `_proto` is the protocol of the service, usually TCP or UDP. We specify the name of the entry, ending in a dot. We then have the TTL, or time to live, of the record. `IN` is the standard DNS class (it's always `IN`). And we specify a priority of the target host: lower values are higher priority. The weight controls preferences for targets with the same priority; higher values are preferred. Last, we specify the port the service runs on and the host name of the host providing the service, ending in a dot.

So, for Prometheus, we might define records like:

Listing 5.16: Example SRV records

```
_prometheus._tcp.example.com. 300 IN SRV 10 1 9100 webappl.  
example.com.  
_prometheus._tcp.example.com. 300 IN SRV 10 1 9100 webapp2.  
example.com.  
_prometheus._tcp.example.com. 300 IN SRV 10 1 9100 webapp3.  
example.com.
```

 **NOTE** There is a whole RFC for DNS service discovery: RFC6763. Prometheus's DNS discovery does not support it.

When Prometheus queries for targets it will look up the DNS server for the example.com domain. It will then search for a SRV record called _prometheus._tcp.example.com in that domain and return the service records in that entry. We only have the three records in that entry, so we'd see three targets returned.

Listing 5.17: The DNS targets from the SRV

```
webapp1.example.com:9100  
webapp2.example.com:9100  
webapp3.example.com:9100
```

We can also query individual A or AAAA records using DNS service discovery. To do so we need to explicitly specify the query type and a port for our scrape. We need to specify the port because the A and AAAA records only return the host, not the host and port combination of the SRV record.

Listing 5.18: DNS A record service discovery job

```
- job_name: webapp  
  dns_sd_configs:  
    - names: [ 'example.com' ]  
      type: A  
      port: 9100
```

This will only return any A records at the root of the example.com domain. If we wanted to return records from a specific DNS entry, we'd use this:

Listing 5.19: DNS subdomain A record service discovery job

```
- job_name: webapp
  dns_sd_configs:
    - names: [ 'web.example.com' ]
      type: A
      port: 9100
```

Here we're pulling A records that resolve for `web.example.com` and suffixing them with the 9100 port.

 **TIP** There's only one metadata label available from DNS service discovery: `_meta_dns_name`. This is set to the specific record that generated the target.

Summary

In this chapter we learned about service discovery. We've seen several mechanisms for discovering targets for Prometheus to scrape, including:

- File-based discovery, populated by external data sources.
- Platform-based service discovery, using the APIs and data of platforms like AWS, Kubernetes, or Google Cloud.
- DNS-based using SRV, A, or AAAA records.

Between these three discovery approaches, you should have sufficient means to identify the resources you wish to monitor.

We also learned a bit more about relabelling, looking at the pre-scrape relabelling phase and seeing how to use metadata to add more context to our metrics.

Now that we've got metrics coming into Prometheus, let's tell folks about them. In the next chapter, we're going to look at alerting.

Chapter 6

Alerting and Alertmanager

I think we ought to take the men out of the loop.

— *War Games*, 1983

In the last few chapters we've installed, configured, and done some basic monitoring with Prometheus. Now we need to understand how to generate useful alerts from our monitoring data. Prometheus is a compartmentalized platform, and the collection and storage of metrics is separate from alerting. Alerting is provided by a tool called Alertmanager, a standalone piece of your monitoring environment. Alerting rules are defined on your Prometheus server. These rules can trigger events that are then propagated to Alertmanager. Alertmanager then decides what to do with the respective alerts, handling issues like duplication, and determines what mechanism to use when sending the alert on: realtime messages, email, or via tools like PagerDuty and VictorOps.

In this chapter, we're going to discuss what makes good alerting, install and configure Alertmanager and look at how to use it to route notifications and manage maintenance. We'll then define our alerting rules on our Prometheus server, using metrics we've collected thus far in the book, and then trigger some alerts.

First let's talk a bit about good alerting.

Alerting

Alerting provides us with indication that some state in our environment has changed, usually for the worse. The key to a good alert is to send it for the right reason, at the right time, with the right tempo, and to put useful information in it.

The most common anti-pattern seen in alerting approaches is sending too many alerts. Too many alerts is the monitoring equivalent of “the boy who cried wolf”. Recipients will become numb to alerts and tune them out. Crucial alerts are often buried in floods of unimportant updates.

The reasons you’re usually sending too many alerts can include:

- An alert is not actionable, it’s informational. You should turn all of these alerts off or turn them into counters that count the rate rather than alert on the symptom.
- A failed host or service upstream triggers alerts for everything downstream of it. You should ensure your alerting system identifies and suppresses these duplicate, adjacent alerts.
- You’re alerting for causes and not symptoms. Symptoms are signs your application has stopped working, they are the manifestation of issues that may have many causes. High latency of an API or website is a symptom. That symptom could be caused by any number of issues: high database usage, memory issues, disk performance, etc. Alerting on symptoms identifies real problems. Alerting on causes alone, for example high database usage, could identify an issue but most likely will not. High database usage might be perfectly normal for this application and may result in no performance issues for an end user or the application. Alerting on it is meaningless as its an internal state. This alert is likely to result in engineers missing more critical issues because they have become numb to the volume of non-actionable,

cause-based alerts. You should focus on symptom-based alerts and rely on your metrics or other diagnostic data to identify causes.

The second most common anti-pattern is misclassification of alerts. Sometimes this also means a crucial alert is buried in other alerts. But other times the alert is sent to the wrong place or with the incorrect urgency.

The third most common anti-pattern is sending alerts that are not useful, especially when the recipient is often a tired, freshly woken engineer on her third or fourth on-call notification for the night. Here's an example of a stock Nagios alert:

Listing 6.1: Stock Nagios alert

```
PROBLEM Host: server.example.com
Service: Disk Space
```

```
State is now: WARNING for 0d 0h 2m 4s (was: WARNING) after 3/3
checks
```

```
Notification sent at: Thu Aug 7th 03:36:42 UTC 2015 (
notification number 1)
```

```
Additional info:
DISK WARNING - free space: /data 678912 MB (9% inode=99%)
```

This notification appears informative but it isn't really. Is this a sudden increase? Or has this grown gradually? What's the rate of expansion? For example, as we noted in the introduction, 9 percent disk space free on a 1 GB partition is different from 9 percent disk free on a 1 TB disk. Can we ignore or mute this notification or do we need to act now?

Good alerting has some key characteristics:

1. An appropriate volume of alerts that focus on symptoms not causes - Noisy alerting results in alert fatigue and, ultimately, alerts being ignored. It's

- easier to fix under-alerting than over-alerting.
2. The right alert priority should be set. If the alert is urgent then it should be routed quickly and simply to the party responsible for responding. If the alert isn't urgent, we should send it with an appropriate tempo, to be responded to when required.
 3. Alerts should include appropriate context to make them immediately useful.



TIP There's a great chapter on alerting in the SRE book.

Now, let's look a little more closely at the Alertmanager.

How the Alertmanager works

The Alertmanager handles alerts sent from a client, generally a Prometheus server. (It can also receive alerts from other tools, but this is beyond the scope of this book.) Alertmanager handles deduplicating, grouping, and routing alerts to receivers like email, SMS, or SaaS services like PagerDuty. You can also manage maintenance using Alertmanager.

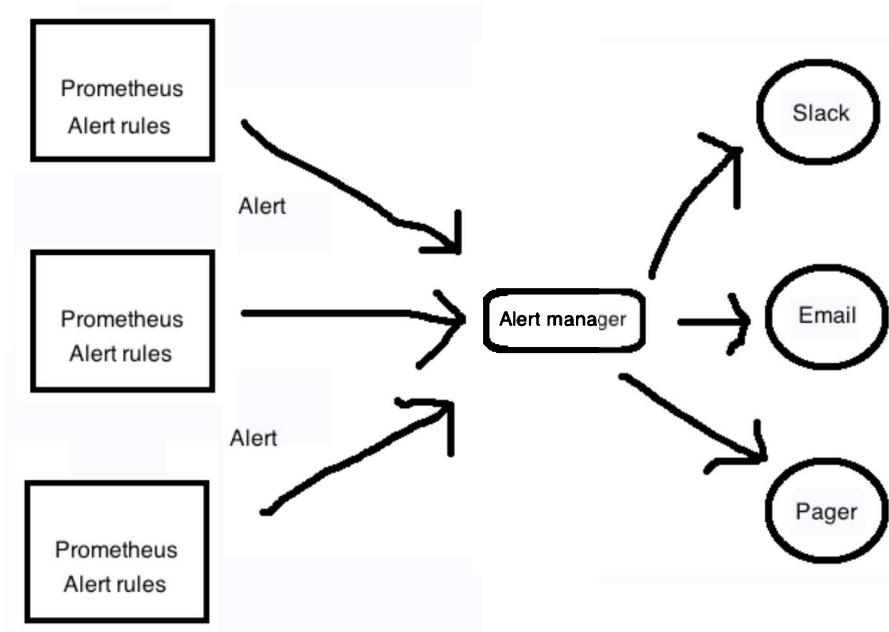


Figure 6.1: Alertmanager architecture

On our Prometheus server we'll be writing alerting rules. These rules will use the metrics we're collecting and trigger on thresholds or criteria we've specified. We'll also see how we might add some context to the alerts. When the threshold or criteria is met, an alert will be generated and pushed to Alertmanager. The alerts are received on an HTTP endpoint on the Alertmanager. One or many Prometheus servers can direct alerts to a single Alertmanager, or you can create a highly available cluster of Alertmanagers, as we'll see later in Chapter 7.

After being received, alerts are processed by the Alertmanager and routed according to their labels. If their path determines it, they are sent by the Alertmanager to external destinations like email, SMS, or chat.

Let's continue with installing Alertmanager.

Installing Alertmanager

Alertmanager is a standalone Go binary. The Prometheus.io download page contains files with the binaries for specific platforms. Currently Alertmanager 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 Alertmanager are available from the GitHub Releases page.



NOTE At the time of writing, Alertmanager was at version 0.15.0.

Installing Alertmanager on Linux

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

Listing 6.2: Download the Alertmanager tarball

```
$ cd /tmp  
$ wget  
https://github.com/prometheus/alertmanager/releases/download/v  
0.15.0/alertmanager-0.15.0.linux-amd64.tar.gz
```

Now let's unpack the alertmanager binary from the tarball, copy it somewhere useful, and change its ownership to the root user.

Listing 6.3: Unpack the alertmanager binary

```
$ tar -xzf alertmanager-0.15.0.linux-amd64.tar.gz  
$ sudo cp alertmanager-0.15.0.linux-amd64/alertmanager /usr/  
local/bin/
```

Let's also copy the amtool binary into our path. The amtool binary is used to help manage the Alertmanager and schedule maintenance windows from the command line.

Listing 6.4: Moving the amtool binary

```
$ sudo cp alertmanager-0.15.0.linux-amd64/amtool /usr/local/bin
```

We can now test if Alertmanager is installed and in our path by checking its version.

Listing 6.5: Checking the Alertmanager version on Linux

```
$ alertmanager --version
alertmanager, version 0.15.0 (branch: HEAD, revision: 30
dd0426c08b6479d9a26259ea5efd63bc1ee273)
  build user:      root@3e103e3fc918
  build date:     20171116-17:45:26
  go version:    go1.9.2
```

 **TIP** This same approach will work on Mac OS X with the Darwin version of the Alertmanager binary.

Installing Alertmanager on Microsoft Windows

To install Alertmanager on Microsoft Windows, we need to download the alertmanager.exe executable and put it in a directory. Let's create a directory for the executable using Powershell.

Listing 6.6: Creating a directory on Windows

```
C:\> MKDIR alertmanager
C:\> CD alertmanager
```

Now download the alertmanager.exe executable from GitHub into the C:\alertmanager directory:

Listing 6.7: Alertmanager Windows download

```
https://github.com/prometheus/alertmanager/releases/download/v0.15.0/alertmanager-0.15.0.windows-amd64.tar.gz
```

Unzip the executable using a tool like 7-Zip into the C:\alertmanager directory. Finally, add the C:\alertmanager directory to the path. This will allow Windows to find the executable. To do this, run this command inside Powershell.

Listing 6.8: Setting the Windows path

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

You should now be able to run the alertmanager.exe executable.

Listing 6.9: Checking the Alertmanager version on Windows

```
C:\> alertmanager.exe --version
alertmanager, version 0.15.0 (branch: HEAD, revision: 30
dd0426c08b6479d9a26259ea5efd63bc1ee273)
  build user:      root@3e103e3fc918
  build date:     20171116-17:45:26
  go version:    go1.9.2
```

Stacks

The stacks we saw in Chapter 3 also include Alertmanager installations.

- 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

Some of the configuration management modules we saw in Chapter 3 can also install Alertmanager: You could review their capabilities to identify which install and configure Alertmanager.

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

Configuring the Alertmanager

Like Prometheus, Alertmanager is configured with a YAML-based configuration file. Let's create a new file and populate it.

Listing 6.10: Creating the alertmanager.yml file

```
$ sudo mkdir -p /etc/alertmanager/  
$ sudo touch /etc/alertmanager/alertmanager.yml
```

Now let's add some configuration to the file. Our basic configuration will send any alerts received out via email. We'll build on this configuration as the chapter unfolds.

Listing 6.11: A simple alertmanager.yml configuration file

```
global:  
  smtp_smarthost: 'localhost:25'  
  smtp_from: 'alertmanager@example.com'  
  smtp_require_tls: false  
  
templates:  
  - '/etc/alertmanager/template/*.tmpl'  
  
route:  
  receiver: email  
  
receivers:  
  - name: 'email'  
    email_configs:  
      - to: 'alerts@example.com'
```

This configuration file contains a basic setup that processes alerts and sends them via email to one address. Let's look at each block in turn.

The first block, `global`, contains global configuration for the Alertmanager. These options set defaults for all the other blocks, and are valid in those blocks as overrides. In our case, we're just configuring some email/SMTP settings: the email server to use for sending emails, the source/from address of those emails, and we're disabling the requirement for automatically using TLS.

 **TIP** This assumes you have a SMTP server running on the localhost on port 25.

The `templates` block contains a list of directories that hold alert templates. As

Alertmanager can send to a variety of destinations, you often need to be able to customize what an alert looks like and the data it contains. Let's just create this directory for the moment.

Listing 6.12: Creating the templates directory

```
$ sudo mkdir -p /etc/alertmanager/template
```

We'll see more about templates later.

Next, we have the `route` block. Routes tell Alertmanager what to do with specific incoming alerts. Alerts are matched against rules and actions taken. You can think about routing like a tree with branches. Every alert enters at the root of the tree—the base route or node. Each route, except the base node, has matching criteria which should match all alerts. You can then define child routes or nodes—the branches of the tree that take specific action or interest in specific alerts. For example, all the alerts from a specific cluster might be processed by a specific child route.

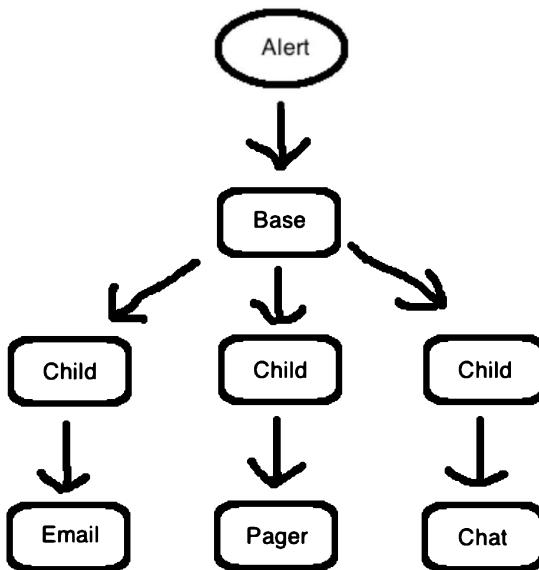


Figure 6.2: Alertmanager routing

In our current configuration, we have only defined the base route, the node at the root of the tree. Later in this chapter, we'll take advantage of routes to ensure our alerts have the right volume, frequency, and destinations.

We've also only defined one parameter: `receiver`. This is the default destination for our alerts, in our case `email`. We'll define that receiver next.

The last block, `receivers`, specifies alert destinations. You can send alerts via email, to services like PagerDuty and VictorOps, and to chat tools like Slack and HipChat. We only have one destination defined: an email address.

Each receiver has a name and associated configuration. Here we've named our receiver `email`. We then provide configuration for the specific types of receivers.

For our email alerts, we use the `email_configs` block to specify email options, like the `to` address to receive alerts. We could also specify SMTP settings, which would override the settings in `global`, and additional items to be added, like mail headers.

 **TIP** One of the built-in receivers is called the webhook receiver. You can use this receiver to send alerts to other destinations that do not have specific receivers in Alertmanager.

Now that we have configured Alertmanager, let's launch it.

Running Alertmanager

Alertmanager runs as a web service, by default on port 9093. It is started by running the `alertmanager` binary on Linux and OS X, or the `alertmanager.exe` executable on Windows, specifying the configuration file we've just created. Let's start Alertmanager now.

Listing 6.13: Starting Alertmanager

```
$ alertmanager --config.file alertmanager.yml
```

We've specified our `alertmanager.yml` configuration file with the `--config.file` flag. Alertmanager has a web interface at:

`http://localhost:9093/`

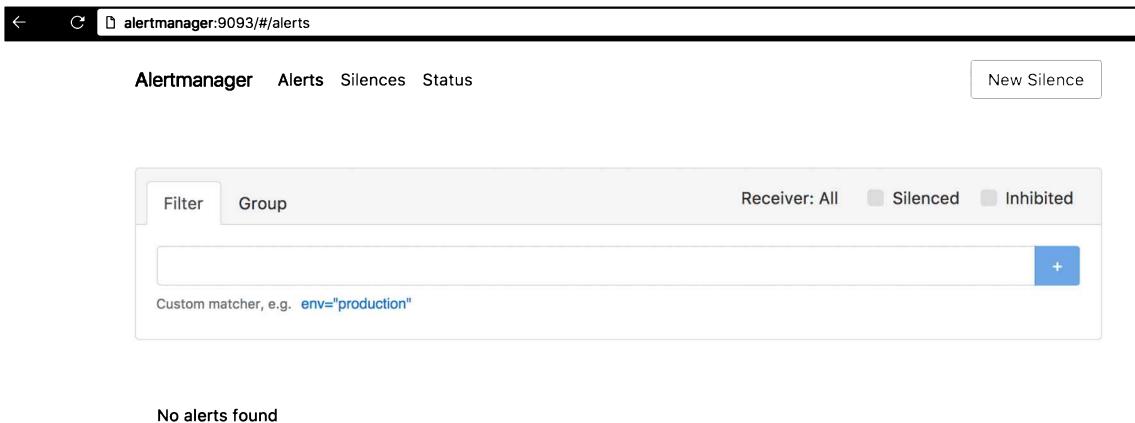


Figure 6.3: Alertmanager web interface

You can use this interface to view current alerts and manage maintenance window alert suppression, named “silences” in Prometheus terminology.

TIP There's also a command line tool `amtool`, that ships with Alertmanager that allows you to query alerts, manage silences and work with an Alertmanager server.

Now's lets configure Prometheus to find our Alertmanager.

Configuring Prometheus for Alertmanager

Let's quickly detour back to our Prometheus configuration to tell it about our new Alertmanager. In Chapter 3 we saw the default Alertmanager configuration in the `prometheus.yml` configuration file. The Alertmanager configuration is contained in the `alerting` block. Let's have a look at the default block.

Listing 6.14: The alerting block

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

The alerting block contains configuration that allows Prometheus to identify one or more Alertmanagers. To do this, Prometheus reuses the same discovery mechanisms it uses to find targets to scrape. In the default configuration this is `static_configs`. Like a monitoring job this specifies a list of targets, here in the form of a host name, `alertmanager`, and a port, 9093—the Alertmanager default port. This listing assumes your Prometheus server can resolve the `alertmanager` hostname to an IP address and that the Alertmanager is running on port 9093 on that host.

 **TIP** You'll also be able to see any configured Alertmanagers in the Prometheus web interface on the status page: <http://localhost:9090/status>.

Alertmanager service discovery

As we have access to service discovery mechanisms, we could also use one of those to identify one or more Alertmanagers. Let's add a DNS SRV record that allows Prometheus to discover our Alertmanagers.

Let's create that record now.

Listing 6.15: The Alertmanager SRV record

```
_alertmanager._tcp.example.com. 300 IN SRV 10 1 9093  
alertmanager1.example.com.
```

Here we've specified a TCP service called `_alertmanager` in the form of a SRV record. Our record returns the hostname `alertmanager1.example.com` and port number 9093 where Prometheus will expect to find an Alertmanager running. Let's configure the Prometheus server to search there.

Listing 6.16: Discovering the Alertmanager

```
alerting:  
  alertmanagers:  
    - dns_sd_configs:  
      - names: [ '_alertmanager._tcp.example.com' ]
```

Here Prometheus will query the `_alertmanager._tcp.example.com` SRV record to return our Alertmanager's hostname. We can do the same with other service discovery mechanisms to identify Alertmanagers to Prometheus.

 **TIP** You'll need to reload or restart Prometheus to enable the Alertmanager configuration.

Monitoring Alertmanager

Like Prometheus, Alertmanager exposes metrics about itself. Let's create a Prometheus job for monitoring our Alertmanager.

Listing 6.17: The Alertmanager Prometheus job

```
- job_name: 'alertmanager'  
  static_configs:  
    - targets: ['localhost:9093']
```

This will collect metrics from `http://localhost:9093/metrics` and scrape a series of time series prefixed with `alertmanager_`. These include counts of alerts by state, and counts of successful and failed notifications by receiver—for example, all failed notifications to the `email` receiver. It also contains cluster status metrics that we can make use of when we look at clustering Alertmanagers in Chapter 7.

Adding alerting rules

Now that we've got Alertmanager set up, let's add our first alerting rules. We're going to create alerts from the node queries we developed in Chapter 4 as well as some basic availability alerting using the `up` metric.

Like recording rules, alerting rules are defined as YAML statements in rules files loaded in the Prometheus server configuration. Let's create a new file, `node_alerts.yml`, in our `rules` directory to hold our node alerting rules.

 **TIP** You can comingle recording rules and alerting rules in the same file, but I like to keep them in separate files for clarity.

Listing 6.18: Creating an alerting rules file

```
$ cd rules  
$ touch node_alerts.yml
```

Rather than add this file to the `rule_files` block in our `prometheus.yml` configuration file, let's use globbing to load all files that end in either `_rules.yml` or `_alerts.yml` in that directory.

Listing 6.19: Adding globbing rule_files block

```
rule_files:  
  - "rules/*_rules.yml"  
  - "rules/*_alerts.yml"
```

You can see that we've added configuration that will load all files with the right naming convention. We'd need to restart the Prometheus server to load this new alerting rules file.

Adding our first alerting rule

Let's add our first rule: a CPU alerting rule. We're going to create an alert that will trigger if the CPU query we created, the average node CPU five-minute rate, is over 80 percent for at least 60 minutes.

Let's see that rule now.

Listing 6.20: Our first alerting rule

```
groups:
- name: node_alerts
  rules:
    - alert: HighNodeCPU
      expr: instance:node_cpu:avg_rate5m > 80
      for: 60m
      labels:
        severity: warning
      annotations:
        summary: High Node CPU for 1 hour
        console: You might want to check the Node Dashboard at
          http://grafana.example.com/dashboard/db/node-dashboard
```

Like our recording rules, alerting rules are grouped together. We've specified a group name: `node_alerts`. The rules in this group are contained in the `rules` block. Each has a name, specified in the `alert` clause. Ours is called `HighNodeCPU`. In each alert group, the alert name needs to be unique.

We also have the test or expression that will trigger the alert. This is specified in the `expr` clause. Our test expression uses the `instance:node_cpu:avg_rate5m` metric we created in Chapter 4 using a recording rule.

`instance:node_cpu:avg_rate5m > 80`

We append a simple check—is the metric greater than 80, or 80 percent?

The next clause, `for`, controls the length of time the test expression must be true for before the alert is fired. In our case, the `instance:node_cpu:avg_rate5m` needs to be greater than 80 percent for 60 minutes before the alert is fired. This limits the potential of the alert being a false positive or a transitory state.

Last, we can decorate our alert with labels and annotations. All the current labels on time series in the alert rule are carried over to the alert. The `labels` clause allows us to specify additional labels to be attached to the alert; here we've added

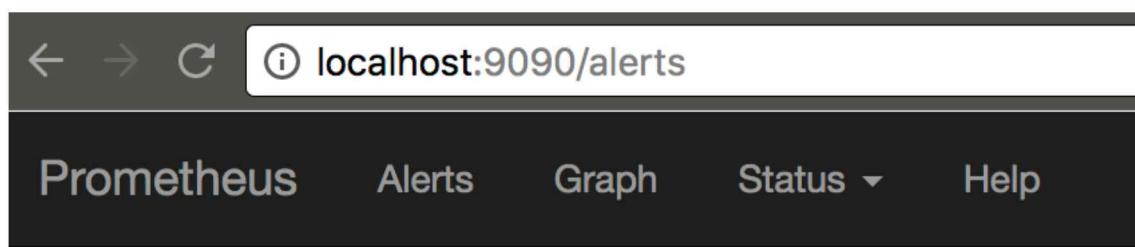
a severity label with a value of warning. We'll see how we can use this label shortly.

The labels on the alert, combined with the name of the alert, represent the identity of the alert. This is the same premise as time series, where the metric name and labels represent the identity of a time series.

The annotations clause allows us to specify informational labels like a description, a link to a run book, or instructions on how to handle this alert. We've added a label called summary that describes the alert. We've also added an annotation called console that points the recipient to a Grafana dashboard for node-based metrics. This is an excellent example of providing context with an annotation.

Now we need to reload Prometheus to enable our new alerting rule.

Once Prometheus is restarted, you'll be able to see your new alert in the Prometheus web interface at <http://localhost:9090/alerts>.



Alerts

HighNodeCPU (0 active)

```
alert: HighNodeCPU
expr: instance:node_cpu:avg_rate5m
  > 80
for: 1h
labels:
  severity: medium
annotations:
  summary: High CPU for 1 hour
```

Figure 6.4: List of Prometheus alerts

This is both a summary of the alerting rule and, as we'll see shortly, a way to see the status of each alert.

What happens when an alert fires?

So how does an alert fire? Prometheus evaluates all rules at a regular interval, defined by the `evaluation_interval`, which we've set to 15 seconds. At each evaluation cycle, Prometheus runs the expression defined in each alerting rule and updates the alert state.

An alert can have three potential states:

- Inactive - The alert is not active.
- Pending - The alert has met the test expression but is still waiting for the duration specified in the `for` clause to fire.
- Firing - The alert has met the test expression and has been Pending for longer than the duration of the `for` clause.

The Pending to Firing transition ensures an alert is more likely to be valid and not flapping. Alerts without a `for` clause automatically transition from Inactive to Firing and only take one evaluation cycle to trigger. Alerts with a `for` clause will transition first to Pending and then to Firing, thus taking at least two evaluation cycles to trigger.

So far, the lifecycle of our alert is:

1. The CPU of a node constantly changes, and it gets scraped by Prometheus every `scrape_interval`. For us this is every 15 seconds.
2. Alerting rules are then evaluated against the metrics every `evaluation_interval`. For us this is 15 seconds again.
3. When the alerting expression is true—for us, CPU is over 80 percent—an alert is created and transitions to the Pending state, honoring the `for` clause.
4. Over the next evaluation cycles, if the alert test expression continues to be true, then the duration of the `for` is checked. If that duration is then complete, the alert transitions to Firing and a notification is generated and pushed to the Alertmanager.

5. If the alert test expression is no longer true then Prometheus changes the alerting rule's state from Pending to Inactive.

The alert at the Alertmanager

Our alert is now in the Firing state, and a notification has been pushed to the Alertmanager. We can see this alert and its status in the Prometheus web interface at <http://localhost:9090/alerts>.



NOTE The Alertmanager API receives alerts on the URI /api/v1/alerts.

Prometheus will also create a metric for each alert in the Pending and Firing states. The metric will be called ALERT and will be constructed like this example for our HighNodeCPU alert.

Listing 6.21: The ALERT time series

```
ALERTS{alertname="HighNodeCPU",alertstate="firing",severity=warning,instance="138.197.26.39:9100"}
```

Each alert metric has a fixed value of 1 and exists for the period the alert is in the Pending or Firing states. After that it receives no updates and is eventually expired.

The notification is sent to the Alertmanager(s) defined in the Prometheus configuration—in our case at the alertmanager host on port 9093. The notification is pushed to an HTTP endpoint:

<http://alertmanager:9093/api/v1/alerts>

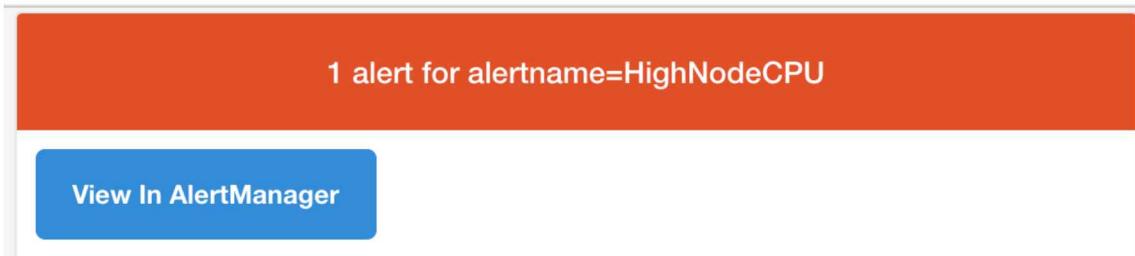
Let's assume one of our HighNodeCPU alerts has fired. We'll be able to see that alert in the Alertmanager web console at <http://alertmanager:9093/#/alerts>.

The screenshot shows the Alertmanager web interface. At the top, there are tabs for 'Alertmanager', 'Alerts', 'Silences', and 'Status'. A 'New Silence' button is located in the top right corner. Below the tabs, there are two buttons: 'Filter' and 'Group'. To the right of these buttons are three status indicators: 'Receiver: All' (selected), 'Silenced' (disabled), and 'Inhibited' (disabled). A search bar with placeholder text 'Custom matcher, e.g. env="production"' is followed by a blue '+' button. In the main area, a single alert is listed with the label 'alertname="HighNodeCPU"'. Below the alert, its timestamp is '00:21:25, 2017-12-18', severity is 'medium', and instance is '138.197.30.163:9100'. There are also 'Info', 'Source', and 'Silence' buttons.

Figure 6.5: Fired alert in Alertmanager

You can use this interface to search for, query, and group current alerts, according to their labels.

In our current Alertmanager configuration, our alert will immediately be routed to our email receiver, and an email like this one below will be generated:



[1] Firing

Labels

alertname = HighNodeCPU
instance = 138.197.30.147:9100
severity = medium

Annotations

summary = High CPU for 1 hour
[Source](#)

Sent by AlertManager

Figure 6.6: HighNodeCPU alert email



TIP We'll see how to update this template later in the chapter.

This doesn't seem very practical if we have many teams, or alerts of different severities. This is where Alertmanager routing is useful.

Adding new alerts and templates

So that we have more alerts to route, let's quickly add some other alert rules to the `node_alerts.yml` alerting rule file.

Listing 6.22: Adding more alerting rules

```
groups:
- name: node_alerts
  rules:
    .
    .
    - alert: DiskWillFillIn4Hours
      expr: predict_linear(node_filesystem_free_bytes{mountpoint="/"}[1h], 4*3600) < 0
      for: 5m
      labels:
        severity: critical
      annotations:
        summary: Disk on {{ $labels.instance }} will fill in
approximately 4 hours.
    - alert: InstanceDown
      expr: up{job="node"} == 0
      for: 10m
      labels:
        severity: critical
      annotations:
        summary: Host {{ $labels.instance }} of {{ $labels.job }} is down!
```

The first alert replicates the `predict_linear` disk prediction we saw in Chapter 4. Here, if the linear regression predicts the disk space of the / root filesystem will be exhausted within four hours, the alert will fire. You'll also notice that we've added some template values to the `summary` annotation.

Templates

Templates are a way of making use of the labels and value of your time series data in your alerts. Templates can be used in annotations and labels. The templates use the standard Go template syntax and expose some variables that contain the labels and value of a time series. The labels are made available in a convenience

variable, `$labels`, and the value of the metric in the variable `$value`.

 **TIP** The `$labels` and `$value` variables are more convenient names for the underlying Go variables: `.Labels` and `.Value`, respectively.

To refer to the instance label in our summary annotation we use `{{ $labels.instance }}`. If we wanted to refer to the value of the time series, we'd use `{{ $value }}`. Prometheus also provides some functions, which you can see in the template reference. An example of this is the `humanize` function, which turns a number into a more human-readable form using metric prefixes. For example:

Listing 6.23: Humanizing a value

```
...
annotations:
    summary: High Node CPU of {{ humanize $value }}% for 1
hour
```

This would display the value of the metric as a two-decimal-place percentage, e.g., 88.23%.

Prometheus alerts

We shouldn't forget that things can go wrong with our Prometheus server, too. Let's add a couple of rules to identify issues there and alert on them. We'll create a new file, `prometheus_alerts.yml`, in the `rules` directory to hold these. As this matches our `rules glob`, it'll also be loaded by Prometheus.

Listing 6.24: Creating the prometheus_alerts.yml file

```
$ touch rules/prometheus_alerts.yml
```

And let's populate this file.

Listing 6.25: The prometheus_alerts.yml file

```
groups:
- name: prometheus_alerts
  rules:
    - alert: PrometheusConfigReloadFailed
      expr: prometheus_config_last_reload_successful == 0
      for: 10m
      labels:
        severity: warning
      annotations:
        description: Reloading Prometheus configuration has failed
        on {{ $labels.instance }}.
    - alert: PrometheusNotConnectedToAlertmanagers
      expr: prometheus_notifications_alertmanagers_discovered < 1
      for: 10m
      labels:
        severity: warning
      annotations:
        description: Prometheus {{ $labels.instance }} is not
        connected to any Alertmanagers
```

Here we've added two new rules. The first, PrometheusConfigReloadFailed, lets us know if our Prometheus configuration has failed a reload. This lets us know, using the metric `prometheus_config_last_reload_successful`, if the last reload failed. If the reload did fail, the metric will have a value of 0.

The second rule makes sure our Prometheus server can discover Alertmanagers. This uses the `prometheus_notifications_alertmanagers_discovered` metric,

which is a count of Alertmanagers this server has found. If it is less than 1, Prometheus hasn't discovered any Alertmanagers and this alert will fire. As there aren't any Alertmanagers, it will only show up on the Prometheus console on the /alerts page.

Availability alerts

Our last alerts help us determine the ability of hosts and services. The first of these alerts takes advantage of the systemd metrics we are collecting using the Node Exporter. We're going to generate an alert if any of the services we're monitoring on our nodes is no longer active.

Listing 6.26: Node service alert

```
- alert: NodeServiceDown
  expr: node_systemd_unit_state{state="active"} != 1
  for: 60s
  labels:
    severity: critical
  annotations:
    summary: Service {{ $labels.name }} on {{ $labels.instance }} is no longer active!
    description: Werner Heisenberg says - "OMG Where's my service?"
```

This alert will trigger if the `node_systemd_unit_state` metric with the `active` label is 0, indicating that a service has failed for at least 60 seconds.

The next alert uses the `up` metric we also saw in Chapter 4. This metric is useful for monitoring the availability of a host. It's not perfect because what we're really monitoring is the success or failure of a job's scrape of that target. But it's useful to know if the instance has stopped responding to scrapes, which potentially indicates a larger problem. To do this, the alert detects if the `up` metric has a value of 0, indicating a failed scrape.

```
up{job="node"} == 0
```

We've added a new value to our severity label of `critical` and added a templated annotation to help indicate which instance and job have failed.

In many cases, knowing a single instance is down isn't actually very important. Instead we could also test for a number of failed instances—for example, a percentage of our instances:

```
avg(up) by (job) <= 0.50
```

This test expression works out the average of the `up` metric, aggregates it by job, and fires if that value is below 50 percent. If 50 percent of the instances in a job fail their scrapes, the alert will fire.

Another approach might be:

```
sum by job (up) / count(up) <= 0.8
```

Here we're summing the `up` metric by job, dividing it by the count, and firing if the result is greater than or equal to 0.8 or indicating that 20 percent of instances in a specific job are not up.

We can make our `up` alert slightly more robust by identifying when targets disappear. If, for example, our target is removed from service discovery, then its metrics will no longer be updated. If all targets disappear from service discovery, no metrics will be recorded—hence our `up` alert won't be fired. Prometheus has a function, `absent`, that detects the presence of missing metrics.

Listing 6.27: The up metric missing alert

```
- alert: InstancesGone
  expr: absent(up{job="node"})
  for: 10s
  labels:
    severity: critical
  annotations:
    summary: Host {{ $labels.instance }} is no longer
    reporting!
    description: 'Werner Heisenberg says, OMG Where are my
    instances?'
```

Here our expression uses the `absent` function to detect if any of the `up` metrics from the `node` job disappear, and it fires an alert if they do.

 **TIP** Another approach for availability monitoring is the probing of endpoints over HTTP, HTTPS, DNS, TCP, and ICMP. We'll see more of it in Chapter 10.

Finally, we'll need to restart the Prometheus server to load these new alerts.

Routing

Now that we have a selection of alerts with some varying attributes, we need to route them to various folks. We discovered earlier that routing is a tree. The top, default route is always configured and matches anything that isn't matched by a child route.

Going back to our Alertmanager configuration, let's add some routing configuration to our `alertmanager.yml` file.

Listing 6.28: Adding routing configuration

```
route:
  group_by: ['instance']
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 3h
  receiver: email
  routes:
    - match:
        severity: critical
        receiver: pager
    - match_re:
        severity: ^warning|critical)$
        receiver: support_team
receivers:
  - name: 'email'
    email_configs:
      - to: 'alerts@example.com'
  - name: 'support_team'
    email_configs:
      - to: 'support@example.com'
  - name: 'pager'
    email_configs:
      - to: 'alert-pager@example.com'
```

You can see that we've added some new options to our default route. The first option, `group_by`, controls how the Alertmanager groups alerts. By default, all alerts are grouped together, but if we specify `group_by` and any labels, then Alertmanager will group alerts by those labels. For example, we've specified the `instance` label, which means that all alerts from a specific instance will be grouped together. If you list more than one label, alerts are grouped if every specified label value matches, for example:

Listing 6.29: Grouping

```
route:  
  group_by: ['service', 'cluster']
```

Here the values of the `service` and `cluster` labels need to match for an alert to be grouped.

**NOTE** This only works for labels, not annotations.

Grouping also changes the behavior of Alertmanager. If a new alert is raised, Alertmanager will wait for the period specified in our next option, `group_wait`, to see if other alerts from that group are received, before firing the alert(s). You can think about this like a group alert buffer. In our case, this wait is 30 seconds.

After the alert(s) are fired, if new alerts from the next evaluation are received for that grouping, Alertmanager will wait for the period specified in the `group_interval` option, five minutes for us, before sending the new alerts. This prevents alert floods for groupings of alerts.

We've also specified the `repeat_interval`. This is a pause that applies not to our groups of alerts, but rather to each single alert, and is the period to wait to resend the same alert. We've specified three hours.

Routes

We've then listed our branched routes. Our first route uses a new receiver we've defined, `pager`. This sends the alerts on this route to a new email address. It finds the specific alerts to be sent using the `match` option. There are two kind

of matching: label matching and regular expression matching. The `match` option does simple label matching.

Listing 6.30: Label matching

```
match:  
  severity: critical
```

Here we're matching all severity labels with a value of `critical` and sending them to the `pager` receiver.

As routes are branches, we can also branch the route again if we need. For example:

Listing 6.31: Routing branching

```
routes:  
- match:  
  severity: critical  
  receiver: pager  
  routes:  
    - match:  
      service: application1  
      receiver: support_team
```

You can see our new `routes` block nested inside our existing route. To trigger this route our alert would first need a severity label of `critical` and then a service label of `application1`. If both these criteria matched, then our alert would be routed to the receiver `support_team`.

We can nest our routes as far down as we need. By default, any alert that matches a route is handled by that route. We can, however, override that behavior using the `continue` option. The `continue` option controls whether an alert will traverse the route and then return to traverse the route tree.

 **NOTE** Alertmanager routes are post-order traversed.

Listing 6.32: Routing branching

```
routes:  
- match:  
  severity: critical  
  receiver: pager  
  continue: true
```

The `continue` option defaults to `false`, but if set to `true` the alert will trigger in this route if matched, and continue to the next sibling route. This is sometimes useful for sending alerts to two places, but a better approach is to specify multiple endpoints in your receiver. For example:

Listing 6.33: Multiple endpoints in a receiver

```
receivers:  
- name: 'email'  
  email_configs:  
    - to: 'alerts@example.com'  
  pagerduty_configs:  
    - service_key: TEAMKEYHERE
```

This adds a second `pagerduty_configs` block that sends to PagerDuty as well as via email. We could specify any of the available receiver destinations—for example, we could send email and a message to a chat service like Slack.

 **TIP** Used to seeing resolution alerts? These are alerts generated when the

alert condition has been resolved. They can be sent with Alertmanager by setting the `send_resolved` option to `true` in your receiver configuration. Sending these resolution alerts is often not recommended as it can lead to a cycle of alerting “false alarms” that result in alert fatigue. Think carefully before enabling them.

Our second route uses the `match_re` option to match a regular expression against a label. The regular expression also uses the `severity` label.

Listing 6.34: A regular expression match

```
- match_re:  
  severity: ^informational|warning)$  
  receiver: support_team
```

 **NOTE** Prometheus and Alertmanager regular expressions are fully anchored.

It matches either `informational` or `warning` values in the `severity` label.

Once you’ve reloaded or restarted Alertmanager to load the new routes, you can try to trigger alerts and see the routing in action.

Receivers and notification templates

Now that we’ve got some basic rules in place, let’s add a non-email receiver. We’re going to add the Slack receiver, which sends messages to Slack instances. Let’s see our new receiver configuration in the `alertmanager.yml` configuration file.

First, we'll add a Slack configuration to our pager receiver.

Listing 6.35: Adding a Slack receiver

```
receivers:  
- name: 'pager'  
  email_configs:  
    - to: 'alert-pager@example.com'  
  slack_configs:  
    - api_url: https://hooks.slack.com/services/ABC123/ABC123/  
EXAMPLE  
  channel: '#monitoring'
```

Now, any route that sends alerts to the pager receiver will be sent both to Slack in the `#monitoring` channel and via email to the `alert-pager@example.com` email address.

The generic alert message that Alertmanager sends to Slack is pretty simple. You can see the default template that Alertmanager uses in its source code. This template contains the defaults for email and other receivers, but we can override these values for many of the receivers. For example, we can add a text line to our Slack alerts.

Listing 6.36: Adding a Slack receiver

```
slack_configs:  
- api_url: https://hooks.slack.com/services/ABC123/ABC123/  
EXAMPLE  
  channel: '#monitoring'  
  text: '{{ .CommonAnnotations.summary }}'
```

Alertmanager notification customization uses Go templating syntax. The data contained in the alerts is also exposed via variables. We're using the `CommonAnnotations` variable, which contains the set of annotations common to

a group of alerts. We're using the `summary` annotation as the text of the Slack notification.



TIP You can find a full reference to notification template variables in the Alertmanager documentation.

We can also use the Go template function to reference external templates, to save on having long, complex strings embedded in our configuration file. We referenced the template directory earlier in this chapter—ours is at `/etc/alertmanager/templates/`. Let's create a template in this directory.

Listing 6.37: Creating a template file

```
$ touch /etc/alertmanager/templates/slack.tmpl
```

And let's populate it.

Listing 6.38: The slack.tmpl file

```
{{ define "slack.example.text" }}{{ .CommonAnnotations.summary }}{{ end}}
```

Here we've defined a new template using the `define` function and ending with `end`. We've called it `slack.example.text` and moved the content from `text` inside the template. We can now reference that template inside our Alertmanager configuration.

Listing 6.39: Adding a Slack receiver

```
slack_configs:  
  - api_url: https://hooks.slack.com/services/ABC123/ABC123/  
EXAMPLE  
  channel: #monitoring  
  text: '{{ template "slack.example.text" . }}'
```

We've used the `template` option to specify the name of our template. The `text` field will now be populated with our template notification. This is useful for decorating notifications with context.

 **TIP** There are some other examples of notification templates in the Alertmanager documentation.

Silences and maintenance

Often we need to let our alerting system know that we've taken something out of service for maintenance and that we don't want alerts triggered. Or we need to mute downstream services and applications when something upstream is broken. Prometheus calls this muting of alerts a “silence.” Silences can be set for specific periods—for example, an hour—or over a set window—for example, until midnight today. This is the silence's expiry time or expiration date. If required, we can also manually expire a silence early, if, say, our maintenance is complete earlier than planned.

You can schedule silences using two methods.

- Via the Alertmanager web console.
- Via the amtool command line tool.

Controlling silences via the Alertmanager

The first method is to use the web interface and click the New Silence button.

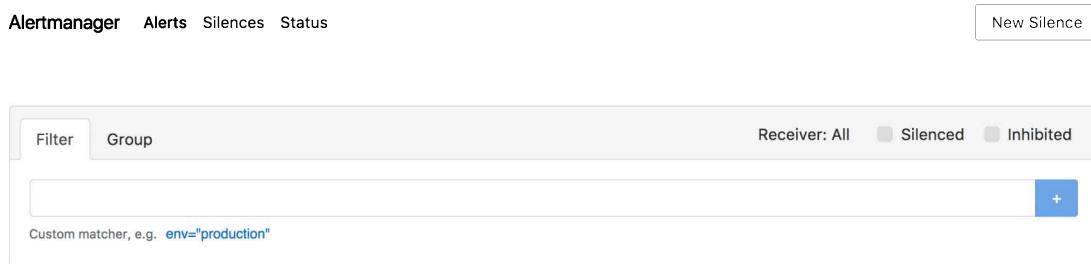


Figure 6.7: Scheduling silences

Silences specify a start time, end time, or a duration. The alerts to be silenced are identified by matching alerts using labels, much like alert routing. You can use straight matches—for example, matching every alert that has a label with a specific value—or you can use a regular expression match. You also need to specify an author for the silence and a comment explaining why alerts are being silenced.

New Silence

Start Duration End

2018-01-02T17:29:53.922Z	✓	4h	✓	2018-01-02T21:29:53.922Z	✓
--------------------------	---	----	---	--------------------------	---

Matchers Alerts affected by this silence.

Name	Value	Regex
severity	critical	<input checked="" type="checkbox"/>

Creator

Werner Heisenberg	✓
-------------------	---

Comment

Silence is a virtue

Buttons: Preview Alerts, Create, Reset

Figure 6.8: A new silence

We click Create to create the new silence (and we can use Preview Alerts to identify if any current alerts will be silenced). Once created we can edit a silence or expire it to remove the silence.

Silence	
	Edit Expire
ID	0430a422-3463-42cd-b48b-d6ca2dea6cfb
Starts at	2018-01-02 18:36:22
Ends at	2018-01-02 21:29:53
Updated at	2018-01-02 18:36:22
Created by	Werner Heisenberg
Comment	Silence is a virtue
State	active
Matchers	severity="critical"
Affected alerts	No silenced alerts

Figure 6.9: Editing or expiring silences

You can see a list of the currently defined silences in the web interface by clicking on the `Silences` menu item in the Alertmanager top menu.

 **NOTE** There's an alternative Alertmanager console called Unsee you might like to check out.

Controlling silences via amtool

The second method is using the `amtool` command line. The `amtool` binary ships with the Alertmanager installation tarball, and we installed it when we installed Alertmanager earlier in the chapter.

Listing 6.40: Using amtool to schedule a silence

```
$ amtool --alertmanager.url=http://localhost:9093 silence add  
alertname=InstancesGone service=application1  
784ac68d-33ce-4e9b-8b95-431a1e0fc268
```

This will add a new silence on the Alertmanager at `http://localhost:9093`. The silence will match alerts with two labels: `alertname`, an automatically populated label containing the alert's name, and `service`, a label we've set.

 **TIP** Silences created with `amtool` are set to automatically expire after one hour. You can specify longer times or a set window with the `--expires` and `--expire-on` flags.

A silence ID will also be returned that you can use to later work with the silence. Here ours is:

784ac68d-33ce-4e9b-8b95-431a1e0fc268

We can query the list of current silences using the `query` sub-command.

Listing 6.41: Querying the silences

```
$ amtool --alertmanager.url=http://localhost:9093 silence query
```

This will return a list of silences and their configurations. You can expire a specific silence via its ID.

Listing 6.42: Expiring the silence

```
$ amtool --alertmanager.url=http://localhost:9093 silence expire  
784ac68d-33ce-4e9b-8b95-431a1e0fc268
```

This will expire the silence on the Alertmanager.

Rather than having to specify the `--alertmanager.url` flag every time, you can create a YAML configuration file for some options. The default configuration file paths that `amtool` will look for are `$HOME/.config/amtool/config.yml` or `/etc/amtool/config.yml`. Let's see a sample file.

Listing 6.43: Sample amtool configuration file

```
alertmanager.url: "http://localhost:9093"  
author: sre@example.com  
comment_required: true
```

You can see that we've added an Alertmanager to work with. We've also specified an author. This is the setting for the creator of a silence; it defaults to your local username, unless overridden like this, or on the command line with the `-a` or `--author` flag. The `comment_required` flag controls whether a silence requires a comment explaining what it does.

 **NOTE** You can specify all `amtool` flags in the configuration file, but some don't make a lot of sense.

Back to creating silences. You can also use a regular expression as the label value when creating a silence.

Listing 6.44: Using amtool to schedule a silence

```
$ amtool silence add --comment "App1 maintenance" alertname=~'Instance.*' service=application1
```

Here we've used `=~` to indicate the label match is a regular expression and matched on all alerts with an `alertname` that starts with `Instance`. We've also used the `--comment` flag to add information about our alert.

We can also control further details of the silence, like so:

Listing 6.45: Omitting alertname

```
$ amtool silence add --author "James" --duration "2h" alertname=InstancesGone service=application1
```

Here we've overridden the silence's creator with the `--author` flag and specified the duration of the silence as two hours, instead of the default one hour.

 **TIP** The `amtool` also allows us to work with Alertmanager and validate its configuration files, among other useful tasks. You can see the full list of command line flags by running `amtool` with the `--help` flag. You can also get help for specific

sub-commands, `amtool silence --help`. You can generate Bash completions and a man page for `amtool` using instructions from here.

Summary

In this chapter, we had a crash course on alerting with Prometheus and Alertmanager.

We touched upon what good alerts look like. We installed Alertmanager on a variety of platforms and configured it.

We saw how to use our time series as a source for alerts, and how to generate those alerts using alerting rules. We saw how to use time series directly or how to build further expressions that analyze time series data to identify alert conditions. We also saw how to add new labels and decorate alerts with additional information and context.

We also saw how to control alerting using silences to mute alerts during maintenance windows or outages.

In the next chapter we'll see how to make Prometheus and the Alertmanager more resilient and scalable. We'll also see how to extend the retention life of your metrics by sending them to remote destinations.

Chapter 7

Scaling and Reliability

Up until now we've seen Prometheus operate as a single server with a single Alertmanager. This fits many monitoring scenarios, especially at the team level when a team is monitoring their own resources, but it often doesn't scale to multiple teams. It's also not very resilient or robust. In these situations, if our Prometheus server or Alertmanager becomes overloaded or fails, our monitoring or alerting will fail, too.

We're going to separate these into two concerns:

- Reliability and fault tolerance.
- Scaling.

Prometheus addresses each concern differently, but we'll see how some architectural choices address both. In this chapter we'll discuss the philosophy and methodology by which Prometheus approaches each concern and understand how to build more scalable and robust Prometheus implementations.

Reliability and fault tolerance

Prometheus's approach to addressing the issue of fault tolerance is tempered by concern about the operational and technical complexities of achieving a high tolerance. In many cases, fault tolerance for monitoring services is addressed by making the monitoring service highly available, usually by clustering the implementation. Clustering solutions, however, require relatively complex networking and management of state between nodes in the cluster.

It's also important to note, as we mentioned in Chapter 2, that Prometheus is focused on real time monitoring, typically with limited data retention, and configuration is assumed to be managed by a configuration management tool. An individual Prometheus server is generally considered disposable from an availability perspective. Prometheus architecture argues that the investment required to achieve that cluster, and consensus of data between nodes of that cluster, is higher than the value of the data itself.

Prometheus doesn't ignore the need to address fault tolerance though. Indeed, the recommended fault-tolerant solution for Prometheus is to run two identically configured Prometheus servers in parallel, both active at the same time. Duplicate alerts generated by this configuration are handled upstream in Alertmanager using its grouping (and its inhibits capability). Instead of focusing on the fault tolerance of the Prometheus server, the recommended approach is to make the upstream Alertmanagers fault tolerant.

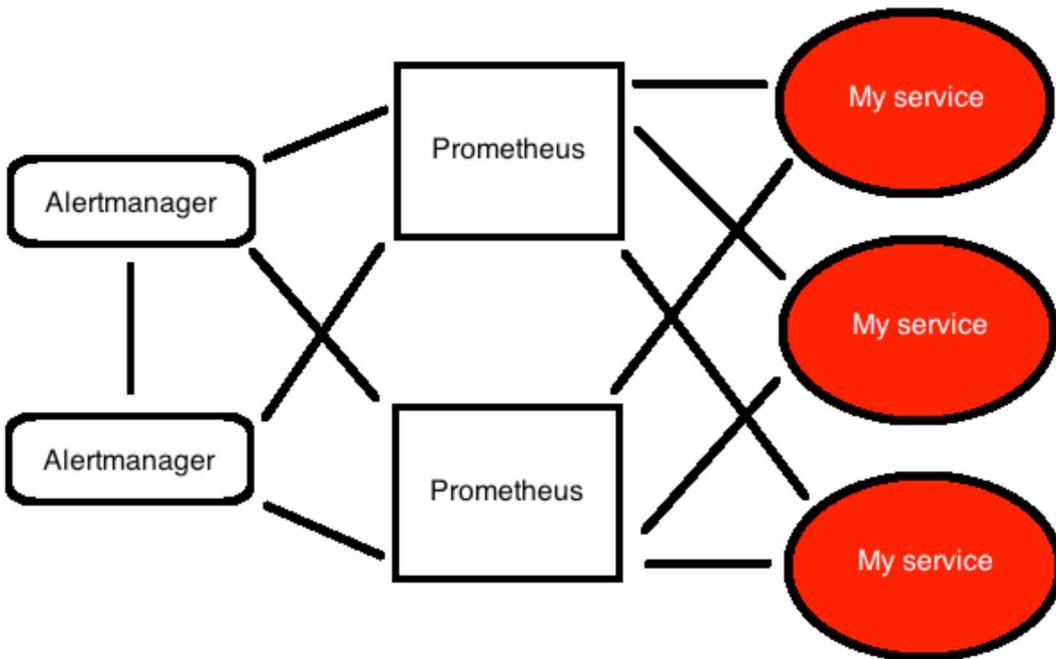


Figure 7.1: Fault-tolerant architecture

This is achieved by creating a cluster of Alertmanagers. All Prometheus servers send alerts to all Alertmanagers. As mentioned, the Alertmanagers take care of deduplication and share alert state through the cluster.

There are obviously downsides to this approach. First, both Prometheus servers will be collecting metrics, doubling any potential load generated by that collection. However, one could argue that the load generated by a scrape is likely low enough that this isn't an issue. Second, if an individual Prometheus server fails or suffers an outage, you'll have a gap in data on one server. This means being aware of that gap when querying data on that server. This is also a relatively minor concern given there's another server to query and that the general focus is on immediate data, not on using Prometheus data for long-term trending analysis.

 **TIP** There are ways to compensate for this in PromQL. For example, when asking for a single metric value from two sources, you could use the `max by` of both metrics. Or, when alerting from a single worker shard with possible gaps, you might increase the `for` clause to ensure you have more than one measure.

Duplicate Prometheus servers

We're not going to document the details of building two duplicate Prometheus servers; this should be relatively easy to achieve using your configuration management tool. We recommend replicating the installation steps from Chapter 3 using one of the configuration management solutions documented there.

Instead we're going to focus on the more complex operation of clustering Alertmanagers.

Setting up Alertmanager clustering

Alertmanager contains a cluster capability provided by Hashicorp's memberlist library. Memberlist is a Go library that manages cluster membership and member-failure detection using a gossip-based protocol, in this case an extension of the SWIM protocol.

To configure clustering we need to install Alertmanager on more than one host. In our case we're going to run it on three hosts: `am1`, `am2`, and `am3`. We first install Alertmanager on each host as we did in Chapter 6. We'll then use the `am1` host to initiate the cluster.

Listing 7.1: Starting Alertmanager cluster

```
am1$ alertmanager --config.file alertmanager.yml --cluster.  
listen-address 172.19.0.10:8001
```

We've run the `alertmanager` binary specifying a configuration file, we can just use the file we created in Chapter 6, and a cluster listen address and port. You should use identical configuration on every node in the cluster. This ensures that alert handling is identical and that your cluster will behave consistently.



WARNING All Alertmanagers should use identical configuration! If it's not identical, it's not actually highly available.

We've specified the IP address of the `am1` host, `172.19.0.10`, and a port of `8001`. Other nodes in the Alertmanager cluster will use this address to connect to the cluster, so that port will need to be open on the network between your Alertmanager cluster nodes.



TIP If you don't specify the cluster listen address, it'll default to `0.0.0.0` on port `9094`.

We can then run the Alertmanager on the remaining two hosts, listening on their local IP addresses, and referencing the IP address and port of the cluster node we've just created.

Listing 7.2: Starting Alertmanager cluster remaining nodes

```
am2$ alertmanager --config.file alertmanager.yml --cluster.  
listen-address 172.19.0.20:8001 --cluster.peer 172.19.0.10:8001  
am3$ alertmanager --config.file alertmanager.yml --cluster.  
listen-address 172.19.0.30:8001 --cluster.peer 172.19.0.10:8001
```

You can see that we've run the `alertmanager` binary on the other two Alertmanager hosts: `am2` and `am3`. We've specified a cluster listen address for each using their own IP addresses and the 8001 port. We've also specified, using the `cluster.peer` flag, the IP address and port of the `am1` node as a peer so they can join the cluster.

You won't see any specific messages indicating the cluster has started (although if you pass the `--debug` flag you'll get more informative output) but you can confirm it on one of the Alertmanager's console status page at `/status`. Let's look at `am1` at <https://172.19.0.10:9093/status>.

Alertmanager Alerts Silences Status

Status

Uptime: 2018-03-17T03:39:54.954534041Z

Cluster Status

Name: 01C8S0NPP8RW9Y2FS45R6R08CR

Status: ready

- Peers:**
- **Name:** 01C8S0NPP8RW9Y2FS45R6R08CR
Address: 172.19.0.10:8001
 - **Name:** 01C8S0NR4QYCF6D6W2FA7ESYZR
Address: 172.19.0.20:8001
 - **Name:** 01C8S0NS7J7E0ZAXPY4T4XT5FK
Address: 172.19.0.30:8001

Figure 7.2: Alertmanager cluster status

We can see our `am1` Alertmanager can see three nodes in the cluster: itself plus `am2` and `am3`.

You can test that the cluster is working by scheduling a silence on one Alertmanager and seeing if it is replicated to the other Alertmanagers. To do this, click the New Silence button on `am1` and schedule a silence. Then check the `/silences` path on `am2` and `am3`. You should see the same silence replicated on all hosts.

Now that our cluster is running, we need to tell Prometheus about all the Alert-

managers.

Configuring Prometheus for an Alertmanager cluster

For resilience purposes, we have to specifically identify all Alertmanagers to the Prometheus server. This way, if an Alertmanager goes down, Prometheus can find an alternative to send an alert to. The Alertmanager cluster itself takes care of sharing any received alert with the other active members of the cluster and potentially handles any deduplication. Thus you should not load balance your Alertmanagers—Prometheus handles that for you.

We could define all of the Alertmanagers to Prometheus using static configuration like so:

Listing 7.3: Defining alertmanagers statically

```
alerting:  
  alertmanagers:  
    - static_configs:  
      - targets:  
        - am1:9093  
        - am2:9093  
        - am3:9093
```

With this configuration the Prometheus server will connect to all three of our Alertmanagers. This assumes that our Prometheus server can resolve DNS entries for each of the Alertmanagers. A smarter approach is to use service discovery to find all of the Alertmanagers. For example, to use DNS-based discovery as we saw in Chapter 6, we can add DNS SRV records for each Alertmanager.

Listing 7.4: The Alertmanager SRV record

```
_alertmanager._tcp.example.com. 300 IN SRV 10 1 9093 am1.example.com.  
_alertmanager._tcp.example.com. 300 IN SRV 10 1 9093 am2.example.com.  
_alertmanager._tcp.example.com. 300 IN SRV 10 1 9093 am3.example.com.
```

Here we've specified a TCP service called `_alertmanager` in the form of a SRV record. Our record returns three host names—`am1`, `am2`, and `am3`—and port number `9093` where Prometheus can expect to find an Alertmanager running. Let's configure the Prometheus server to discover them.

Listing 7.5: Discovering the Alertmanager

```
alerting:  
  alertmanagers:  
    - dns_sd_configs:  
      - names: [ '_alertmanager._tcp.example.com' ]
```

Here Prometheus will query the `alertmanager.example.com` SRV record to return our list of Alertmanagers. We could do the same with other service discovery mechanisms to identify all the Alertmanagers in our cluster to Prometheus.

If we now restart Prometheus we can see all of our connected Alertmanagers in the Prometheus server's status page.

Alertmanagers

Endpoint

`http://am1:9093/api/v1/alerts`

`http://am2:9093/api/v1/alerts`

`http://am3:9093/api/v1/alerts`

Figure 7.3: Prometheus clustered Alertmanagers

Now when an alert is raised it is sent to all the discovered Alertmanagers. The Alertmanagers receive the alert, handle deduplication, and share state across the cluster.

Together this provides the upstream fault tolerance that ensures your alerts are delivered.

Scaling

In addition to fault tolerance, we also have options for scaling Prometheus. Most of the options are essentially manual and involve selecting specific workloads to run on specific Prometheus servers.

Scaling your Prometheus environment usually takes two forms: functional scaling or horizontal scaling.

Functional scaling

Functional scaling uses shards to split monitoring concerns onto separate Prometheus servers. For example, this could be splitting servers via geography or logical domains.

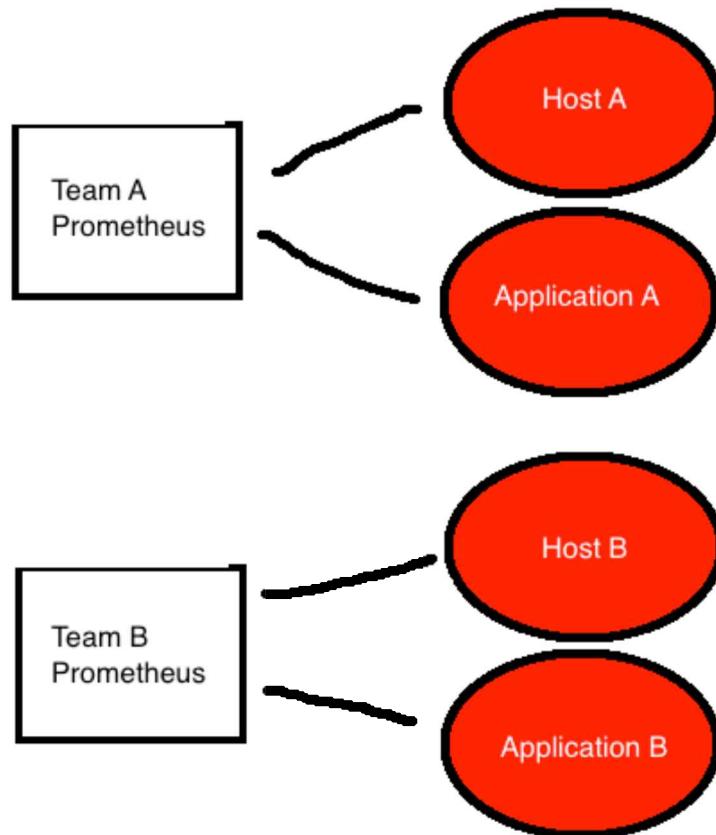


Figure 7.4: Organizational sharding

Or it could be via specific functions, sending all infrastructure monitoring to one server and all application monitoring to another server.

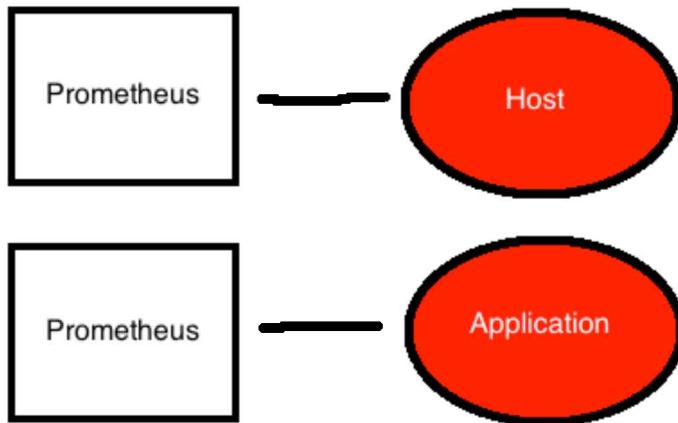


Figure 7.5: Functional sharding

It is a relatively simple process to create otherwise identical Prometheus servers with specific jobs running on each server. It is best done using configuration management tools to ensure creating servers, and the specific jobs that run on them, is an automated process.

From here, if you need a holistic view of certain areas or functions, you can potentially use federation (more on this shortly) to extract time series to centralized Prometheus servers. Usefully, Grafana supports pulling data from more than one Prometheus server to construct a graph. This allows you to federate data from multiple servers at the visualization level, assuming some consistency in the time series being collected.

Horizontal shards

At some point, usually in huge installations, the capacity and complexity of vertical sharding will become problematic. This is especially true when individual jobs contain thousands of instances. In that case, you can consider an alternative: horizontal sharding. Horizontal sharding uses a series of worker servers, each of which scrapes a subset of targets. We then aggregate specific time series we're interested in on the worker servers. For example, if we're monitoring host metrics, we might aggregate a subset of those metrics. A primary server then scrapes each of the worker's aggregated metrics using Prometheus's federation API.

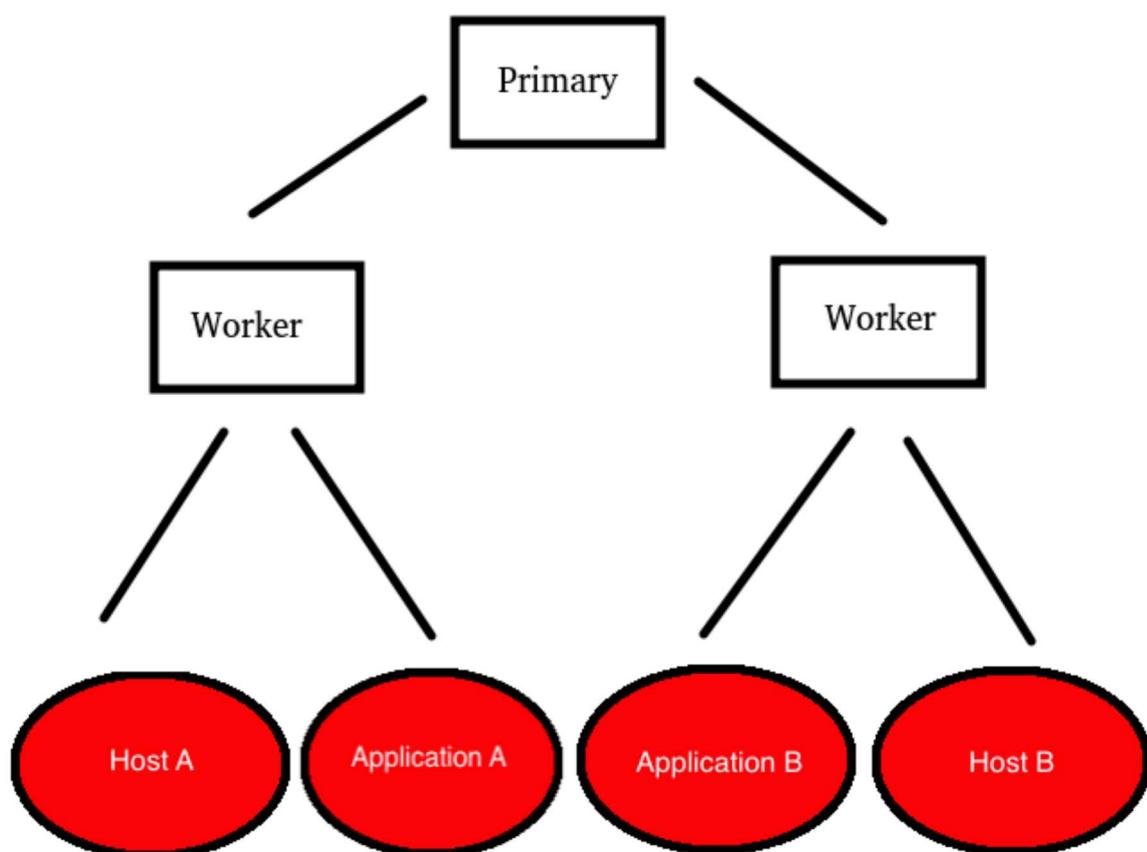


Figure 7.6: Horizontal sharding

Our primary server not only pulls in the aggregated metrics but now also acts as the default source for graphing or exposing metrics to tools like Grafana. You can add tiered layers of workers and primaries if you need to go deeper or scale further. A good example of this is a zone-based primary and workers, perhaps for a failure domain or a logical zone like an Amazon Availability Zone, reporting up to a global primary that treats the zone-based primaries as workers.

 **TIP** If you need to query metrics that are not being aggregated, you will need to refer to the specific worker server that is collecting for the specific target or targets you are interested in. You can use the `worker` label to help you identify the right worker.

It's important to note that this sort of scaling does have risks and limitations, perhaps the most obvious being that you need to scrape a subset of metrics from the worker servers rather than a large volume or all of the metrics the workers are collecting. This is a pyramid-like hierarchy rather than a distributed hierarchy. The scraping requests of the primary onto the workers is also load that you will need to consider.

Next, you're creating a more complex hierarchy of Prometheus servers in your environment. Rather than just the connection between the workers and the targets, you also need to worry about the connection between the primary and the workers. This could reduce the reliability of your solution.

Last, the potential consistency and correctness of your data could be reduced. Your workers are scraping targets according to their intervals, and your primary server is in turn scraping the workers. This introduces a delay in the results reaching the primary server and could potentially skew data or result in an alert being delayed.

A consequence of the latter two issues is that it's probably not a good idea to centralize alerting on the primary server. Instead, push alerting down onto the

worker servers where they are more likely to identify issues, like a missing target, or reduce the lag between identifying the alert condition and the alert firing.



NOTE Horizontal sharding is generally a last resort. We'd expect you to have tens of thousands of targets or large volumes of time series being scraped per target before you'd need to scale out in this manner.

With these caveats in mind, let's look at how we might use this configuration.

Creating shard workers

Let's create some workers and see how they can scrape the target subsets. We're going to create workers and number them 0 through 2. We're going to assume that the primary job our workers will execute is scraping the node_exporter. Every worker needs to be uniquely identifiable. We're going to use external labels to do this. External labels are added to every time series or alert that leaves a Prometheus server. External labels are provided via the `external_labels` configuration block in our `prometheus.yml`.

Let's create the base configuration for our first worker, `worker0`, now.



TIP As always, use configuration management to do this.

Listing 7.6: The worker0 configuration

```
global:
  external_labels:
    worker: 0

rule_files:
  - "rules/node_rules.yml"

scrape_configs:
  - job_name: 'node'
    file_sd_configs:
      - files:
          - targets/nodes/*.json
        refresh_interval: 5m
    relabel_configs:
      - source_labels: [__address__]
        modulus: 3
        target_label: __tmp_hash
        action: hashmod
      - source_labels: [__tmp_hash]
        regex: ^0$
        action: keep
```

We can see our `external_labels` block contains a label, `worker`, with a value of 0. We'll use `worker: 1`, `worker: 2`, and so on for our remaining workers. We've defined a single job, which uses file-based service discovery, to load a list of targets from any file ending in `*.json` in the `targets/nodes` directory. We would use a service discovery tool or a configuration management tool to populate all of our nodes into the JSON file or files.

We then use relabelling to create a modulus of the `source_labels` hash. In our case, we're just creating a modulus of the hash of the concatenated address label. We use a modulus of 3, the number of workers scraping metrics. You'll need to update this value if you add workers (another good reason to use a configuration management tool that can automatically increment the modulus). The hash is

created using the `hashmod` action. The result is then stored in a target label called `__tmp_hash`.

We then use the `keep` action to match any time series from any targets that match the modulus. So `worker0` would retrieve time series from targets with a modulo of 0, `worker1` those targets with a modulo of 1, etc. This evenly distributes targets between the workers. If you need to scale to more targets you can add workers and update the modulus used on the hash.

We can then aggregate the worker time series we want to federate using rules. Let's say we'd like to gather the memory, CPU, and disk metrics from the Node Exporter for federation. To aggregate the time series we want we're going to use the rules we created in Chapter 4—for example, the CPU rule:

Listing 7.7: The instance CPU rule

```
groups:
- name: node_rules
  rules:
    - record: instance:node_cpu:avg_rate5m
      expr: 100 - avg (irate(node_cpu_seconds_total{job="node",
mode="idle"})[5m])) by (instance) * 100
```

This will create a series of new time series that we'll then scrape upstream using a primary Prometheus server.

Primary shard server

Let's now configure a primary Prometheus server to scrape the workers for the time series. The primary Prometheus server has a job or jobs to scrape workers; each worker is a target in a job. Let's look at the `prometheus.yml` configuration for our primary server.

Listing 7.8: The primary configuration

```

    ...
scrape_configs:
- job_name: 'node_workers'
  file_sd_configs:
    - files:
      - 'targets/workers/*.json'
      refresh_interval: 5m
  honor_labels: true
  metrics_path: /federate
  params:
    'match[]':
      - '{__name__=~"^instance:.+"}'

```

On our primary server, we've got a job called `node_workers`. This job discovers the list of workers using file-based service discovery. Our workers are in `workers/targets/workers.json`.

Listing 7.9: Worker file discovery

```

[{
  "targets": [
    "worker0:9090",
    "worker1:9090",
    "worker2:9090"
  ]
}]

```

You'll note we've enabled the `honor_labels` flag. This flag controls how Prometheus handles conflicts between labels. By setting it to `true` we ensure that an upstream primary server doesn't overwrite labels from downstream workers.

We've overridden the standard metrics path to use the `/federate` API. The

federate API endpoint allows us to query a remote Prometheus server for specific time series, specified by a matching parameter.

Listing 7.10: Matching parameter

```
metrics_path: /federate
params:
  'match[]':
    - '{__name__=~"^instance:.*"}'
```

We use the `params` option to specify the `match[]` parameter. The `match[]` parameter takes an instant vector selector that has to match the specific time series we want to return. In our case we're matching against the name of the time series.

Listing 7.11: condition] The match [] condition

```
'{__name__=~"^instance:.*"}'
```

 **TIP** You can specify multiple `match[]` parameters, and Prometheus will return the union of all of the conditions.

This match is a regular expression match that returns all of the time series that start with `instance:`. All of the rules we used to aggregate our Node Exporter metrics are prefixed with `instance:`, so the time series for CPU, memory, and disk will be selected and scraped by the primary server.

We can see what is going to be selected by the query parameter by using `curl` or browsing to the `/federate` path, with an appropriate `match[]` parameter, on one of the worker servers.

```

← ⌂ worker0 9090/federate?match[]&__name__=~"instance"
# TYPE instance:node_cpu:avg_rate5m untyped
instance:node_cpu:avg_rate5m{instance="ne12:9100",worker="0"} 31.833333333334707 1522087637378
instance:node_cpu:avg_rate5m{instance="ne16:9100",worker="0"} 30.89999999999918 1522087637378
instance:node_cpu:avg_rate5m{instance="ne17:9100",worker="0"} 28.350000000001884 1522087637378
instance:node_cpu:avg_rate5m{instance="ne20:9100",worker="0"} 27.108333333333135 1522087637378
instance:node_cpu:avg_rate5m{instance="ne25:9100",worker="0"} 28.799999999999585 1522087637378
instance:node_cpu:avg_rate5m{instance="ne27:9100",worker="0"} 27.69166666666706 1522087637378
instance:node_cpu:avg_rate5m{instance="ne28:9100",worker="0"} 28.774999999999793 1522087637378
instance:node_cpu:avg_rate5m{instance="ne2:9100",worker="0"} 27.18333333334758 1522087637378
instance:node_cpu:avg_rate5m{instance="ne5:9100",worker="0"} 31.816666666667658 1522087637378
instance:node_cpu:avg_rate5m{instance="ne8:9100",worker="0"} 32.633333333335486 1522087637378
instance:node_cpu:avg_rate5m{instance="ne9:9100",worker="0"} 28.94999999999905 1522087637378
# TYPE instance:node_cpus:count untyped
instance:node_cpus:count{instance="ne12:9100",worker="0"} 8 1522087637378
instance:node_cpus:count{instance="ne16:9100",worker="0"} 8 1522087637378
instance:node_cpus:count{instance="ne17:9100",worker="0"} 8 1522087637378
instance:node_cpus:count{instance="ne20:9100",worker="0"} 8 1522087637378
instance:node_cpus:count{instance="ne25:9100",worker="0"} 8 1522087637378
instance:node_cpus:count{instance="ne27:9100",worker="0"} 8 1522087637378
instance:node_cpus:count{instance="ne28:9100",worker="0"} 8 1522087637378
instance:node_cpus:count{instance="ne2:9100",worker="0"} 8 1522087637378
instance:node_cpus:count{instance="ne5:9100",worker="0"} 8 1522087637378
instance:node_cpus:count{instance="ne8:9100",worker="0"} 8 1522087637378
instance:node_cpus:count{instance="ne9:9100",worker="0"} 8 1522087637378

```

Figure 7.7: The Federate API

Our query has returned all of the time series starting with `instance:`.

The primary server's `node_workers` job will scrape these metrics each time it's run. You can then use the primary server to query and graph the aggregated metrics from all of the targets scraped by the worker servers.

Remote storage

There's one last aspect of scaling we should mention: remote storage. Prometheus has the capability to write to (and in some cases read from) remote stores of metrics. The ability to write to remote storage allows you to send metrics from Prometheus, working around its constraints in scalability, to a remote system.

Prometheus has two types of remote storage integration:

- It can write metric samples to a remote destination.
- It can read metric samples to a remote destination.

The remote storage protocol uses a Snappy-compressed protocol buffer encoding over HTTP. It's configured in Prometheus via the `remote_write` and `remote_read` blocks.

Currently, Prometheus supports a variety of endpoints for writing and reading. You can find a full list in the Prometheus documentation but highlights include Chronix, CrateDB, Graphite, InfluxDB, OpenTSDB, and PostgreSQL.

We're not going to cover any of these in any detail, but you should be able to follow the documentation and examples to get started.

Third-party tools

There's a small selection of third-party tools that aim to make Prometheus scaling easier. These include:

- Cortex - A scalable Prometheus-as-a-Service tool.
- Thanos - A highly available Prometheus setup with long-term storage capabilities.
- Vulcan - A now discontinued attempt to build a more scalable Prometheus.
Mostly useful as a reference.

Summary

In this chapter we learned how Prometheus handles fault tolerance for monitoring. We saw how to create a cluster of Alertmanagers to ensure that your alerts are sent.

We also saw how to scale Prometheus monitoring using additional servers or via sharding with federation.

In the next chapter we'll look at instrumenting applications for monitoring.

Chapter 8

Instrumenting Applications

In the last few chapters we've seen the mechanics of Prometheus. We've collected metrics to process and visualize. We've gathered host and container metrics using Prometheus and exporters.

In this chapter we're going to extend our monitoring and collection to applications. We're going to focus on monitoring applications and how to emit metrics by instrumenting code. We're going to see how to add a Prometheus client to an application, add metrics to the application, and then use a Prometheus job to scrape those metrics.

First, though, we're going to go through some high-level design patterns and principles you should consider when thinking about application monitoring.

An application monitoring primer

Let's look at some basic tenets for application monitoring. First, in any good application development methodology, it's a great idea to identify what you want to build before you build it. Monitoring is no different. Sadly there's a common anti-pattern in application development of considering monitoring and other oper-

ational functions like security as value-add components of your application rather than core features. Monitoring (and security!) are core functional features of your applications. If you’re building a specification or user stories for your application, include monitoring for each component of your application. Not building metrics or monitoring is a serious business and operational risk resulting in:

- An inability to identify or diagnose faults.
- An inability to measure the operational performance of your application.
- An inability to measure the business performance and success of an application or a component, such as tracking sales figures or the value of transactions.

Another common anti-pattern is not instrumenting enough. We’ll always recommended that you over-instrument your applications. One will often complain about having too little data, but rarely will one worry about having too much.



NOTE Within constraints of storage capacity, your monitoring stopping working because you exceeded that capacity is obviously undesirable. It’s often useful to look at retention time as a primary way to reduce storage without losing useful information.

Third, if you use multiple environments—for example development, testing, staging, and production—ensure that your monitoring configuration provides labels so you know that the data is from a specific environment. This way you can partition your monitoring and metrics. We’ll talk more about this later in the chapter.

Where should I instrument?

Good places to start adding instrumentation for your applications are at points of ingress and egress. For example:

- Measure counts and timings of requests and responses, such as to specific web pages or API endpoints. If you’re instrumenting an existing application, make a priority-driven list of specific pages or endpoints, and instrument them in order of importance.
- Measure counts and timings of calls to external services and APIs, such as if your application uses a database, cache, or search service, or if it uses third-party services like a payments gateway.
- Measure counts and timings of job scheduling, execution, and other periodic events like cron jobs.
- Measure counts and timings of significant business and functional events, such as users being created, or transactions like payments and sales.

Instrument taxonomies

You should ensure that metrics are categorized and clearly identified by the application, method, function, or similar marker so that you can ensure you know what and where a metric is generated. We talked about label taxonomies in Chapter 4.

Metrics

Like much of the rest of our monitoring, metrics are going to be key to our application monitoring. So what should we monitor in our applications? We want to look at two broad types of metrics—albeit types with considerable overlap:

- Application metrics, which generally measure the state and performance of your application code.

- Business metrics, which generally measure the value of your application. For example, on an e-commerce site, it might be how many sales you made.

We're going to look at examples of both types of metrics in this chapter, with the caveat that Prometheus tends to focus on more immediate metrics. For longer-term business metrics, you may, in many cases, use event-based systems.

Application metrics

Application metrics measure the performance and state of your applications. They include characteristics of the end user experience of the application, like latency and response times. Behind this we measure the throughput of the application: requests, request volumes, transactions, and transaction timings.

 **TIP** Good examples of how to measure application performance are the USE and RED Methods and Google Golden Signals that we mentioned earlier.

We also look at the functionality and state of the application. A good example of this might be successful and failed logins or errors, crashes, and failures. We could also measure the volume and performance of activities like jobs, emails, or other asynchronous activities.

Business metrics

Business metrics are the next layer up from our applications metrics. They are usually synonymous with application metrics. If you think about measuring the number of requests made to a specific service as being an application metric, then the business metric usually does something with the content of the request. An

example of the application metric might be measuring the latency of a payment transaction; the corresponding business metric might be the value of each payment transaction. Business metrics might include the number of new users/customers, number of sales, sales by value or location, or anything else that helps measure the state of a business.

Where to put your metrics

Once we know what we want to monitor and measure, we need to work out where to put our metrics. In almost all cases the best place to put these metrics is inside our code, as close as possible to the action we're trying to monitor or measure.

We don't, however, want to put our metrics configuration inline everywhere that we want to record a metric. Instead we want to create a utility library: a function that allows us to create a variety of metrics from a centralized setup. This is sometimes called the utility pattern: a metrics-utility class that does not require instantiation and only has static methods.

The utility pattern

A common pattern is to create a utility library or module using one of the available clients. The utility library would expose an API that allows us to create and increment metrics. We can then use this API throughout our code base to instrument the areas of the application we're interested in.

Let's take a look at an example of this. We've created some Ruby-esque code to demonstrate, and we've assumed that we have already created a utility library called Metric.

 **NOTE** We'll see a functioning example of this pattern later in this chapter.

Listing 8.1: A sample payments method

```
include Metric

def pay_user(user, amount)
  pay(user.account, amount)
  Metric.increment 'payment'
  Metric.increment "payment-amount, #{amount.to_i}"
  send_payment_notification(user.email)
end

def send_payment_notification(email)
  send_email(payment, email)
  Metric.increment 'email-payment'
end
```

Here we've first included our `Metric` utility library. We can see that we've specified both application and business metrics. We've first defined a method called `pay_user` that takes `user` and `amount` values as parameters. We've then made a payment using our data and incremented two metrics in our first method:

- A payment metric — Here we increment the metric each time we make a payment.
- A payment-amount metric — This metric records each payment by amount.

Finally, we've sent an email using a second method, `send_payment_notification`, where we've incremented a third metric: `email-payment`. The `email-payment` metric counts the number of payment emails sent.

The external pattern

What if you don't control the code base, can't insert monitors or measures inside your code, or perhaps have a legacy application that can't be changed or updated? Then you need to find the next closest place to your application. The most obvious places are the outputs and external subsystems around your application—for example, a database or cache.

If your application emits logs, then identify what material the logs contain and see if you can use their contents to measure the behavior of the application. Often you can track the frequency of events by simply recording the counts of specific log entries. If your application records or triggers events in other systems—things like database transactions, job scheduling, emails sent, calls to authentication or authorization systems, caches, or data stores—then you can use the data contained in these events or the counts of specific events to record the performance of your application.

We'll talk more about this in Chapter 9.

Building metrics into a sample application

Now that we have some background on monitoring applications, let's look at an example of how we might implement this in the real world. We're going to build an application that takes advantage of a utility library to send events from the application. We've created a sample Rails application using Rails Composer. We're going to call it `mwp-rails`, or Monitoring with Prometheus Rails application. The `mwp-rails` application allows us to create and delete users and sign in to the application.



NOTE You can find the `mwp-rails` application on GitHub.

To instrument our application we first need to add support for Prometheus using a Ruby-based client. The `prometheus-client` gem allows us to create a Prometheus client inside our application.

There are similar clients for a number of platforms including:

- Go
- Java/JVM
- Python

There is also a large collection of third-party clients for a variety of frameworks and languages.

Adding the client

Let's add the `prometheus-client` gem to our Rails application's `Gemfile`.

Listing 8.2: The mwp-rails Gemfile

```
source 'https://rubygems.org'  
ruby '2.4.2'  
gem 'rails', '5.1.5'  
.  
.  
.  
gem 'prometheus-client'  
..
```

We then install the new gem using the `bundle` command.

Listing 8.3: Install prometheus-client with the bundle command

```
$ sudo bundle install
Fetching gem metadata from https://rubygems.org/...
Fetching version metadata from https://rubygems.org/...
Fetching dependency metadata from https://rubygems.org/...
...
Installing prometheus-client 0.7.1
...
...
```

We can then test the client using a Rails console. Let's launch one now using the `rails c` command.

Listing 8.4: Testing the Prometheus client with the Rails console

```
$ rails c
Loading development environment (Rails 4.2.4)
[1] pry(main)> prometheus = Prometheus::Client.registry
[2] pry(main)> test_counter = prometheus.counter(:test_counter,
'A test counter')
=> #<Prometheus::Client::Counter:0x00007f9aea051dd8
@base_labels={},
@docstring="A test counter",
@mutex=#<Thread::Mutex:0x00007f9aea051d88>,
@name=:test_counter,
@validator=#<Prometheus::Client::LabelSetValidator:0
x00007f9aea051d10 @validated={}>,
@values={}
[3] pry(main)> test_counter.increment
=> 1.0
```

We've launched a Rails console and created a Prometheus registry using the code:

Listing 8.5: Creating a Prometheus registry

```
prometheus = Prometheus::Client.registry
```

The registry is the core of the Prometheus application instrumentation. Every metric you create needs to be registered first. We've created a registry called `prometheus`. We can now create metrics in this registry.

Listing 8.6: Registering a Prometheus metric

```
test_counter = prometheus.counter(:test_counter, 'A test counter')
```

We have a new metric called `test_counter`. It's created with the `counter` method on the registry. The metric name needs to be a symbol, `:test_counter`, and needs a description. Our is: A test counter.

We can increment our new metric with the `increment` method.

Listing 8.7: Incrementing a metric

```
test_counter.increment
```

Now the value of the `test_counter` metric will be `1.0`, which we can see by querying the value of the metric using the `get` method.

Listing 8.8: Incrementing a metric

```
test_counter.get  
1.0
```

We can register a number of types of metrics, including summaries and histograms.

Listing 8.9: The basic Prometheus client_ruby metrics

```
test_counter = prometheus.counter(:test_counter, 'A test counter')  
test_gauge = prometheus.gauge(:test_gauge, 'A test gauge')  
test_histogram = prometheus.histogram(:test_histogram, 'A test histogram')  
test_summary = prometheus.summary(:test_summary, 'A test summary')
```

We can now add the instrumentation to our Rails application.

Adding it to Rails

We're not going to manually create a registry and metrics every time we want to log any metrics, so let's set up some utility code to do this for us. We're going to create a `Metrics` module in our `lib` directory that we'll use in our Rails application. Let's do that now.

Listing 8.10: Creating a Metrics module

```
$ touch lib/metrics.rb
```

And let's populate the file with a module.

Listing 8.11: The Metrics module

```
module Metrics
  def self.counter(name, docstring, base_labels = {})
    provide_metric(name) || registry.counter(name, docstring,
base_labels)
  end

  def self.summary(name, docstring, base_labels = {})
    provide_metric(name) || registry.summary(name, docstring,
base_labels)
  end

  def self.gauge(name, docstring, base_labels = {})
    provide_metric(name) || registry.gauge(name, docstring,
base_labels)
  end

  def self.histogram(name, docstring, base_labels = {}, buckets
= ::Prometheus::Client::Histogram::DEFAULT_BUCKETS)
    provide_metric(name) || registry.histogram(name, docstring,
base_labels, buckets)
  end

  private

  def self.provide_metric(name)
    registry.get(name)
  end

  def self.registry
    @registry || ::Prometheus::Client.registry
  end
end
```

Our `Metrics` module has methods for each metric type. The metric methods check for the presence of an existing metric in the registry (using the `get` method in `provide_metric` which retrieves metric names) or creates a new metric.

We then need to extend Rails to load our Metrics library. There's a few ways to do this, but adding an initializer is my favorite.

Listing 8.12: Creating an initializer for the metrics library

```
$ touch config/initializers/lib.rb
```

And then requiring our library.

Listing 8.13: The config/initializers/lib.rb file

```
require 'metrics'
```

 **TIP** We could also extend the `autoload_paths` configuration option to load everything in `lib`, but I feel like this gives us less control over what loads.

We can then add metrics to some of our methods. Let's start with incrementing a counter when users are deleted.

Listing 8.14: Counter for user deletions

```
def destroy
  user = User.find(params[:id])
  user.destroy
  Metrics.counter(:users_deleted_counter, "Deleted users counter")
    .increment
  redirect_to users_path, :notice => "User deleted."
end
```

We can see the line:

```
Metrics.counter(:users_deleted_counter, "Deleted users counter").  
increment
```

We call the counter metric in the Metrics module and pass in a metric name in the form of a symbol, :users_deleted_counter, then a description of the metric, Deleted users counter. We've ended the line with the increment method that increments the counter once. This will create a metric called users_deleted_counter.

We could also add a label or increment by a specific value by using the increment method like so:

```
.increment({ service: 'foo' }, 2)
```

This would increment a counter with a value of 2 and add the label service: foo to the metric. You can specify multiple labels by separating each with commas: { service: 'foo', app: 'bar' }.

We could also create another counter for created users by adding it to the User model.

Listing 8.15: Counter for user creation

```
class User < ActiveRecord::Base  
  enum role: [:user, :vip, :admin]  
  after_initialize :set_default_role, :if => :new_record?  
  after_create do  
    Metrics.counter(:user_created_counter, "Users created  
counter").increment  
  end  
end
```

Here we've used an Active Record callback, after_create, to increment a counter, users_created_counter, when a new user is created.

 **NOTE** You may have a wide variety of applications you want to instrument. We're creating an example application to show you how we might apply some of these principles. While you might not be able to reuse the code, the high-level principles apply to almost every framework and language you're likely to have running.

We then need to expose our metrics to be scraped. We're also going to enable some Rack middleware to auto-create some useful metrics on HTTP requests. In our case we enable the metrics endpoint by adding an exporter (and the middleware collector) to our `config.ru` file.

Listing 8.16: Adding Prometheus to the config.ru file

```
require 'prometheus/middleware/collector'
require 'prometheus/middleware/exporter'

use Prometheus::Middleware::Collector
use Prometheus::Middleware::Exporter
```

Here we've required and used two components of the Prometheus client: the middleware exporter and collector. The exporter creates a route, `/metrics`, containing any metrics specified in Prometheus registries defined by the app. The collector adds some HTTP server metrics to the endpoint that are collected via Rack middleware.

If we browse to this endpoint, `/metrics`, we'll see some of those metrics.

Listing 8.17: The Rails /metrics endpoint

```
# HELP http_server_requests_total The total number of HTTP
requests handled by the Rack application.
http_server_requests_total{code="200",method="get",path="/" } 2.0
# HELP http_server_request_duration_seconds The HTTP response
duration of the Rack application.
http_server_request_duration_seconds_bucket{method="get",path="/",
",le="0.005"} 0.0
http_server_request_duration_seconds_bucket{method="get",path="/",
",le="0.01"} 0.0
...
# HELP users_updated_counter Users updated counter
users_updated_counter 1.0
```

We can see a selection of the metrics available. Perhaps most interesting is a series of histogram buckets showing the HTTP server request duration, with dimensions for method and path. This histogram is an easy way to measure request latency for specific paths and to identify any badly performing requests.

Using our metrics

Now our application has metrics being generated and we can make use of them in Prometheus. Let's create a job to scrape our /metrics endpoint. We're going to add our Rails servers to our file-based service discovery. We're going to add three Rails servers by their hostnames.

Listing 8.18: Our Rails servers service discovery

```
[{
    "targets": ["mwp-rails1.example.com", "mwp-rails2.example.com", "mwp-rails3.example.com"]
}]
```

We're then going to create our new job in our `prometheus.yml` configuration file.

Listing 8.19: The rails job

```
- job_name: rails
  file_sd_configs:
    - files:
        - targets/rails/*.json
  refresh_interval: 5m
```

If we reload Prometheus we'll be able to see our Rails servers as new targets.

rails (3/3 up) show less				
Endpoint	State	Labels	Last Scrape	Error
http://mwp-rails1.example.com:80/metrics	UP	instance="mwp-rails1.example.com:80"	203ms ago	
http://mwp-rails2.example.com:80/metrics	UP	instance="mwp-rails2.example.com:80"	3.378s ago	
http://mwp-rails3.example.com:80/metrics	UP	instance="mwp-rails3.example.com:80"	12.658s ago	

Figure 8.1: Rails server targets

And see our new metrics in the dashboard.



Figure 8.2: Rails metrics

Now we can make use of these metrics to monitor our Rails servers.

Summary

In this chapter we explored ways to monitor and instrument our applications and their workflows, including understanding where to place our application monitoring.

We learned about building our own metrics into our applications and services, and we built a sample Rails application to show how to expose and scrape metrics with Prometheus.

In the next chapter we will see how to turn external data, specifically log entries, into metric data you can consume with Prometheus.

Chapter 9

Logging as Instrumentation

In previous chapters we looked at application, host, and container-based monitoring. In this chapter we’re going to look at how we can use our logging data as the source of time series data that can be scraped by Prometheus. While our hosts, services, and applications can generate crucial metrics and events, they also often generate logs that can tell us useful things about their state and status.

This is especially true if you’re monitoring a legacy application that is not instrumented or that it’s not feasible to instrument. In this case, sometimes the cost of rewriting, patching, or refactoring that application to expose internal state is not a good engineering investment, or there are technological constraints to instrumentation. You still need to understand what’s happening inside the application, though—and one of the easiest ways is to adapt log output.

 **TIP** Another potential approach is to look at the contents of the `/proc` subsystem using the Process exporter.

Log output often contains useful status, timing, and measurement information.

For example, using the access log output from a web or application server is a useful way of tracking transaction timings or error volumes. Tools can parse these log entries, create metrics from matched output, and make them available to be scraped by a Prometheus job.

In this chapter we're going to look at using log entries to create metrics, and then scrape them with Prometheus.

Processing logs for metrics

In order to extract data from our log entries we're going to make use of a log processing tool. There are several we could use, including the Grok Exporter and a utility from Google called mtail. We've chosen to look at mtail because it's a little more lightweight and somewhat more popular.

 **TIP** Got a Logstash/ELK installation? You can't currently directly output to Prometheus but you can use Logstash's metric filter to create metrics and output them to Alertmanager directly.

Introducing mtail

The mtail log processor is written by SRE folks at Google. It's a Go application licensed with the Apache 2.0 license. The mtail log processor is specifically designed for extracting metrics from application logs to be exported into a time series database. It aims to fill the niche we described above: parsing log data from applications that cannot export their own internal state.

The mtail log processor works by running “programs” that define log matching

patterns, and specify the metrics to create from the matches and any actions to take. It works very well with Prometheus and exposes any created metrics for scraping, but can also be configured to send the metrics to tools like collectd, StatsD, or Graphite.

Installing mtail

The mtail log processor is shipped as a single binary: `mtail`. It's packaged for a variety of operating systems including Linux, OS X, and Microsoft Windows.

Let's download the binary now.

Listing 9.1: Download and install the mtail binary

```
$ wget https://github.com/google/mtail/releases/download/v3.0.0-rc14/mtail_v3.0.0-rc14_linux_amd64 -O mtail  
$ chmod 0755 mtail  
$ sudo cp mtail /usr/local/bin
```

We can confirm the `mtail` binary is working by running it with the `--version` flag.

Listing 9.2: Running the mtail binary

```
$ mtail --version  
mtail version v3.0.0-rc14-119-g01c76cd git revision 01  
c76cdelee5399be4d6c62536d338ba3077e0e7 go version go1.8.3
```

Using mtail

The `mtail` binary is configured via the command line. You specify a list of log files to parse, and a directory of programs to run over those files.

 **TIP** You can see a full list of `mtail`'s command line flags using the `--help` flag.

Let's start by creating a directory to hold our `mtail` programs. As always, this is usually better done by creating a configuration management module (or a Docker container) rather than being done manually, but we'll show you the details so you can understand what's happening.

Listing 9.3: Creating an mtail program directory

```
$ sudo mkdir /etc/mtail
```

Now let's create our first `mtail` program in a file in our new directory. Every `mtail` program needs to end with the suffix `.mtail`. Let's create a new program called `line_count.mtail`. This is the simplest `mtail` program: it increments a counter every time it parses a new line.

Listing 9.4: Creating the line_count.mtail program

```
$ sudo touch /etc/mtail/line_count.mtail
```

And let's populate that file.

Listing 9.5: The line_count.mtail program

```
counter line_count

/$/ {
    line_count++
}
```

We've started our program by defining a counter called `line_count`. Counter names are prefixed with `counter` (and, naturally, gauges are prefixed with `gauge`). These counters and gauges are exported by mtail to whatever destination you define; in our case, it'll be an endpoint that can be scraped by Prometheus. You must define any counters or gauges before you can work with them.

Next, we define the guts of our mtail program: the condition we want to match and the action we want to take, with the condition specified first and the action following, wrapped in `{ }`.

You can specify multiple sets of conditions and actions in a program. You can extend them with conditional logic in the form of an `else` clause too.



NOTE mtail programs look a lot like awk programs.

The condition can be a regular expression, matching some specific log entry. In our case, we've specified `/$/`, which matches the end of the line. The mtail processor uses RE2 regular expressions. You can see the full syntax on the RE2 wiki.

We could also specify a relational expression, much like those in a C `if` clause, for example:

Listing 9.6: A relational clause

```
line_count < 20 {  
    . . .  
}
```

Here the program would only take the action if the value of the `line_count` counter was greater than 20.

In the case of our initial program our action is:

```
line_count++
```

Which uses an operator, `++`, to increment the `line_counter` counter. It's about the simplest action we can take.

Let's count some log entries now.

 **TIP** You can find more documentation on the `mtail` syntax on GitHub and on the guide to programming in it.

Running mtail

To run `mtail` we need to specify some programs to run and some log files to parse. Let's do that now.

Listing 9.7: Running mtail

```
$ sudo mtail --progs /etc/mtail --logs '/var/log/*.log'
```

This will run `mtail` with two flags. The first flag, `--progs`, tells `mtail` where to find our programs. The second flag, `--logs`, tells `mtail` where to find log files to parse. We're using a glob pattern to match all log files in the `/var/log` directory. You can specify a comma-separated list of files or specify the `--logs` flag multiple times. `mtail` is also conscious of log file truncation so it can handle stopping, restarting, and actions like log rotation.



NOTE The user you're running `mtail` as will need permissions to the log files you're parsing, otherwise `mtail` will not be able to read the files. You will get a read error in the `mtail` log output, obtained using the `--logtostderr` flag, when it can't read a file.

When we run `mtail`, it'll launch a web server on port 3903 (you can control the IP address and port using the `--address` and `--port` flags). Let's browse to that web server now.

The home path shows some diagnostic information, like so:

mtail on :3903

Build: `mtail` version v3.0.0-rc5-119-g01c76cd git revision 01c76cde1ee5399be4d6c62536d338ba3077e0e7 go version go1.8.3

Metrics: `json`, `prometheus`, `varz`

Debug: `debug/pprof`, `debug/vars`

Program Loader

`line_count.mtail`

No compile errors

Total load errors ; successes: 1

Log Tailer

`/var/log`

Figure 9.1: mtail diagnostics

You can see the build of mtail as well as links to the default metric output formats, diagnostic information, and a list of programs loaded, any errors, and log files being tracked. You can see mtail outputs metrics in JSON, varz (an internal Google format for metrics collection), and the format we want: Prometheus. Let's click on the Prometheus link, which will take us to the /metrics path.



TIP You can also send metrics to tools like StatsD and Graphite.

Listing 9.8: The mtail /metrics path

```
# TYPE line_count counter
# line_count defined at line_count.mtail:1:9-18
line_count{prog="line_count.mtail"} 1561
```

We can see some familiar output: help text and a Prometheus metric with a single label, `prog`. This is added automatically to each metric and populated with the name of the program that generated the metric. You can omit this label by setting the `--emit_prog_label` flag to false.

In our case, our `line_count` metric has counted 1561 lines worth of log entries. We could then add a job to scrape this endpoint for our `line_counter` metric.

This isn't an overly useful example though. Let's look at some more complex programs.

Processing web server access logs

Let's use mtail to extract some metrics from an Apache access log, specifically one using the combined log format. To save some time, we can use an example program provided with mtail. We create the program in the /etc/mtail directory and name it apache_combined.mtail. The contents are:

Listing 9.9: The apache_combined program

```
# Parser for the common apache "NCSA extended/combined" log
format
# LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-
agent}i\""
counter apache_http_requests_total by request_method,
http_version, request_status
counter apache_http_bytes_total by request_method, http_version,
request_status

/^/ +
/(?P<hostname>[0-9A-Za-z\.-]+) / + # %h
/(?P<remote_logname>[0-9A-Za-z-]+) / + # %l
/(?P<remote_username>[0-9A-Za-z-]+) / + # %u
/(?P<timestamp>\[\d{2}\]\w{3}\d{4}:\d{2}:\d{2} (\+|-)\d
{4}\]) / + # %u
/"(?P<request_method>[A-Z]+) (?P<URI>\S+) (?P<http_version>HTTP
\|[0-9\.\.]+)" / + # \"%r\"
/(?P<request_status>\d{3}) / + # %>s
/(?P<response_size>\d+) / + # %b
/"(?P<referer>\S+)" / + # \"%{Referer}i\"
/"(?P<user_agent>[:print:]+)" / + # \"%{User-agent}i\"
/$/ {
    apache_http_requests_total[$request_method][$http_version][
$request_status]++
    apache_http_bytes_total[$request_method][$http_version][
$request_status] += $response_size
}
```

 **TIP** You can find the code for this program on GitHub. There's also a large collection of example programs to help you get started on GitHub.

We can see our program is well commented; comments are specified using the `#` symbol. It includes an example of the log format at the top of the program and then defines two counters:

- `apache_http_requests_total` by `request_method`, `http_version`, `request_status`
- `apache_http_bytes_total` by `request_method`, `http_version`, `request_status`

The `by` operator specifies additional dimensions to add to the metric. In the first counter, `apache_http_requests_total`, we've added the additional dimensions of `request_method`, `http_version`, and `request_status`, which will be added as labels on the resulting counter.

We then see a series of regular expressions that match each element of the access log line and are chained together using `+` operators.

 **TIP** These regular expressions can get quite complex when parsing convoluted log lines, so mtail also allows you to reuse regular expressions by defining them as constants.

Inside these regular expressions you can see a series of captures like so:

`(?P<request_status>\d{3})`

These are named capture groups. In this example, we're capturing a named value

of `request_status`. We can then use these captures in our actions.

Listing 9.10: The combined access log actions

```
{  
    apache_http_requests_total[$request_method][$http_version][  
    $request_status]++  
    apache_http_bytes_total[$request_method][$http_version][  
    $request_status] += $response_size  
}
```

The action increments the first counter, `apache_http_requests_total`, adding some of the captures, prefixed with `$`, to the counter as dimensions. Each dimension is wrapped in `[]` square brackets.

The second counter has an additive operation, using the `+=` operator to add each new response size in bytes to the counter.

 **TIP** mtail can record either integer or floating point values for metrics. By default, all metrics are integers, unless the compiler can infer a floating point. Inference is achieved by analyzing expressions, for example identifying that a regular expression is capturing a floating point value.

If we were to run mtail again, this time loading some Apache (or other web server that used the combined log format), we'd see these new metrics populated.

Listing 9.11: Running mtail

```
$ sudo mtail --progs /etc/mtail --logs '/var/log/apache/*.access'
```

And then browse to the `/metrics` path:

Listing 9.12: Apache combined metrics

```
# TYPE apache_http_requests_total counter
# apache_http_requests_total defined at apache_combined.mtail
:6:9-34
apache_http_requests_total{http_version="HTTP/1.1",
request_method="GET",request_status="200",prog="apache_combined.
mtail"} 73
# apache_http_requests_total defined at apache_combined.mtail
:6:9-34
apache_http_requests_total{http_version="HTTP/1.1",
request_method="GET",request_status="304",prog="apache_combined.
mtail"} 3
# TYPE apache_http_bytes_total counter
# apache_http_bytes_total defined at apache_combined.mtail:7:9-
31
apache_http_bytes_total{http_version="HTTP/1.1",request_method=
"GET",request_status="200",prog="apache_combined.mtail"} 2814654
# apache_http_bytes_total defined at apache_combined.mtail:7:9-
31
apache_http_bytes_total{http_version="HTTP/1.1",request_method=
"GET",request_status="304",prog="apache_combined.mtail"} 0
```

We can see a new set of counters, with one counter for each method and HTTP response code dimension.

We can also do more complex operations, like building histograms.

Parsing Rails logs into a histogram

To see a histogram being created, let's look at some lines from the example Rails mtail program. Rails request logging is useful for measuring performance, but somewhat unfriendly to parse. Let's see how mtail does it.

Listing 9.13: The mtail rails program

```

counter rails_requests_started_total
counter rails_requests_started by verb
counter rails_requests_completed_total
counter rails_requests_completed by status
counter rails_requests_completed_milliseconds_sum by status
counter rails_requests_completed_milliseconds_count by status
counter rails_requests_completed_milliseconds_bucket by le,
status

/^Started (?P<verb>[A-Z]+) .*/ {
    rails_requests_started_total++
    rails_requests_started[$verb]++
}

/^Completed (?P<status>\d{3}) .+ in (?P<request_milliseconds>\d+)
ms .*/ {
    rails_requests_completed_total++
    rails_requests_completed[$status]++

    rails_requests_completed_milliseconds_sum[$status] +=
$request_milliseconds
    rails_requests_completed_milliseconds_count[$status]++

    # 10ms bucket
    $request_milliseconds <= 10 {
        rails_requests_completed_milliseconds_bucket["10"][$status]++
    }
    # 50ms bucket
    $request_milliseconds <= 50 {
        rails_requests_completed_milliseconds_bucket["50"][$status]++


    . . .
}

```

Our program opens with defining counters for started and completed requests. We then see a condition and action that increments the request started counters, a

total and a set of counters with dimensions created from the status of the request.

Next, our program calculates completed requests. Here we're capturing the status code and the request time in milliseconds. We use these to create a sum of request time and a count of requests, both by status.

We then nest in another set of conditions and actions, this time to create our histogram. We have a series of conditions testing the length in milliseconds of the request:

```
$request_milliseconds <= 10
```

If our request time is less than or equal to 10, then a histogram bucket counter, with the length test attached as a dimension and also the status, is incremented. We can create counters for each bucket we want.

Let's run our new program over some Rails logs and see what our resulting metrics look like.

Listing 9.14: Rails mtail metric output

```
rails_requests_started_total{prog="rails.mtail"} 44
rails_requests_started{verb="POST",prog="rails.mtail"} 19
rails_requests_started{verb="PUT",prog="rails.mtail"} 18
rails_requests_started{verb="GET",prog="rails.mtail"} 7
rails_requests_completed_total{prog="rails.mtail"} 217
rails_requests_completed{status="200",prog="rails.mtail"} 217
rails_requests_completed_milliseconds_sum{status="200",prog="rails.mtail"} 3555
rails_requests_completed_milliseconds_count{status="200",prog="rails.mtail"} 217
rails_requests_completed_milliseconds_bucket{le="10",status="200",prog="rails.mtail"} 93
rails_requests_completed_milliseconds_bucket{le="50",status="200",prog="rails.mtail"} 217
.
.
```

 **TIP** The `1e` is a common abbreviation for “less than or equal to,” indicating the content of the specific bucket.

We can see that we have counters for each request started by total and verb. We can also see our completed total and a total by status code. And we can see our buckets—in our case just the 10 ms and 50 ms buckets.

Deploying mtail

We've now seen two mtail programs. We deploy them in a number of ways. We recommend running an mtail instance per application, adjacent to the application and deployed via configuration management as a dependency. This pattern is often called a sidecar and lends itself well to containerized applications. We'll see it in Chapter 13, when we look at monitoring applications running on Kubernetes.

We can also run multiple programs in a single mtail instance, but this has the caveat that mtail will run every program over every log file passed to it, which could have a performance impact on your host.

Scraping our mtail endpoint

Now that we've got some metrics being exposed, let's create a Prometheus job to scrape them.

Listing 9.15: The mtail job

```
    . . .

scrape_configs:
- job_name: 'mtail'
  file_sd_configs:
    - files:
      - 'targets/mtail/*.json'
  refresh_interval: 5m
```

Our job uses file-based service discovery to define a couple of targets, a web server and our rails server. Both targets are scraped on port 3903.

Listing 9.16: Worker file discovery

```
[{
  "targets": [
    "web:3903",
    "rails:3903"
  ]
}]
```

If we restart Prometheus we're now collecting the time series generated from our mtail programs on our Prometheus server, and can make use of these metrics.

Summary

In this chapter we saw how to use log entries to provide metrics for applications we can't, or can't afford to, instrument. We did this using the mtail log processor.

Note that we only scratched the surface of the capabilities of mtail's language for log parsing and processing. You should read the wiki on GitHub and review the

example programs to learn more about the language and how to write your own mtail programs.

In the next chapter we'll learn how to do probe monitoring using Prometheus.

Chapter 10

Probing

In Chapter 1 we discussed that there are two major approaches to monitoring applications: probing and introspection. In this chapter we're going to explore probe monitoring. Probe monitoring probes the outside of an application. You query the external characteristics of an application: does it respond to a poll on an open port and return the correct data or response code? An example of probe monitoring is performing an ICMP ping or echo check and confirming you have received a response. This type of probing is also called blackbox monitoring because we're treating the application inside as a black box.

We'll use probe monitoring to see the state of external aspects of our applications, which is especially useful from outside of our network. We're going to use an exporter called the blackbox exporter to conduct this monitoring.

Probing architecture

Probing with Prometheus works by running an exporter, the blackbox exporter, that probes remote targets and exposes any time series collected on a local endpoint. A Prometheus job then scrapes any metrics from the endpoint.

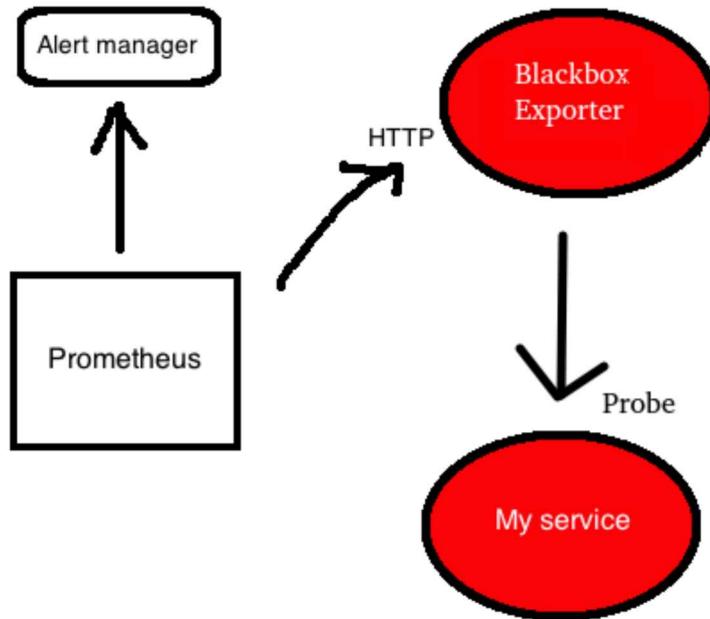


Figure 10.1: Probing architecture

Monitoring probes have three constraints:

- They need to be in a position to see the resources being probed.
- The position of the probe needs to test the right path to the resources. For example, if you're testing external access to an application, running the probe behind your firewall won't validate this access.
- The position of the probing exporter needs to be able to be scraped by your Prometheus server.

It's common to position probes in geographically distributed locations outside of an organization's network to ensure maximum coverage for the detection of faults and the collection of data about user experience with the application.

Because of the complexity of deploying probes externally, and if a wide distribution of probes is needed, it's common to outsource these probes to a third-party service. There are all sorts of commercial vendors that provide this service, some of which expose metrics on their platforms and others that allow metrics to be exported for use.

For our purposes, however, we're going to deploy the blackbox exporter to an external host and use it to monitor the outside of our applications.

The blackbox exporter

The blackbox exporter is a single binary Go application licensed under the Apache 2.0 license. The exporter allows probing of endpoints over HTTP, HTTPS, DNS, TCP, and ICMP. Its architecture is a little different from other exporters. Inside the exporter we define a series of modules that perform specific checks—for example, checking a web server is running, or that a DNS record resolves. When the exporter runs, it exposes these modules and an API on a URL. Prometheus passes targets and specific modules to run on those targets as parameters to that URL. The exporter executes the check and returns the resulting metrics to Prometheus.

Let's see about installing it.

Installing the exporter

The Prometheus.io download page contains zip files with the binaries for specific platforms. Currently, the exporter 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 the exporter are available from the GitHub Releases page.



NOTE At the time of writing, blackbox exporter was at version 0.12.0.

Installing the exporter on Linux

To install blackbox exporter on a 64-bit Linux host, we can download the zipped tarball. We can use `wget` or `curl` to get the file from the download site.

Listing 10.1: Download the blackbox exporter zip file

```
$ cd /tmp  
$ wget  
https://github.com/prometheus/blackbox_exporter/releases/  
download/v0.12.0/blackbox_exporter-0.12.0.linux-amd64.tar.gz
```

Now let's unpack the `blackbox_exporter` binary from the tarball and move it somewhere useful.

Listing 10.2: Unpack the blackbox_exporter binary

```
$ tar -xzf blackbox_exporter-0.12.0.linux-amd64.tar.gz  
$ sudo cp blackbox_exporter-0.12.0.linux-amd64/blackbox_exporter  
/usr/local/bin/
```

We can now test if the exporter is installed and in our path by checking its version.

Listing 10.3: Checking the blackbox exporter version on Linux

```
$ blackbox_exporter --version
blackbox_exporter, version 0.12.0 (branch: HEAD, revision: 30
dd0426c08b6479d9a26259ea5efd63bc1ee273)
  build user:      root@3e103e3fc918
  build date:     20171116-17:45:26
  go version:    go1.9.2
```

 **TIP** This same approach will work on Mac OS X with the Darwin version of the blackbox exporter binary.

Installing the exporter on Microsoft Windows

To install blackbox exporter on Microsoft Windows, we need to download the `blackbox_exporter.exe` executable and put it in a directory. Let's create a directory for the executable using Powershell.

Listing 10.4: Creating a directory on Windows

```
C:\> MKDIR blackbox_exporter
C:\> CD blackbox_exporter
```

Now download the `blackbox_exporter.exe` executable from GitHub into the `C:\blackbox_exporter` directory:

Listing 10.5: Blackbox exporter Windows download

```
https://github.com/prometheus/blackbox_exporter/releases/
download/v0.12.0/blackbox_exporter-0.12.0.windows-amd64.tar.gz
```

Unzip the executable, using a tool like 7-Zip, into the C:\blackbox_exporter directory. Finally, add the C:\blackbox_exporter directory to the path. This will allow Windows to find the executable. To do this, run this command inside Powershell.

Listing 10.6: Setting the Windows path

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

You should now be able to run the blackbox_exporter.exe executable.

Listing 10.7: Checking the blackbox exporter version on Windows

```
C:\> blackbox_exporter.exe --version
blackbox_exporter, version 0.12.0 (branch: HEAD, revision: 30
dd0426c08b6479d9a26259ea5efd63bc1ee273)
  build user:      root@3e103e3fc918
  build date:     20171116-17:45:26
  go version:    go1.9.2
```

Installing via configuration management

Some of the configuration management modules we saw in Chapter 3 can also install the blackbox exporter:

- A Puppet module for Prometheus.
- A Chef cookbook for Prometheus.
- A blackbox exporter Docker image.
- A SaltStack formula.

 **TIP** Remember, configuration management is the recommended approach for installing and managing Prometheus and its components!

Configuring the exporter

The exporter is configured via a YAML-based configuration file and driven with command line flags. The command line flags specify the location of the configuration file, the port to bind to, and logging. We can see the available command line flags by running the `blackbox_exporter` binary with the `-h` flag.

Let's create a configuration file to run the exporter now.

Listing 10.8: The prober.yml file

```
$ sudo mkdir -p /etc/prober
$ sudo touch /etc/prober/prober.yml
```

And let's populate it with some basic configuration. The exporter uses modules to define various checks. Each module has a name, and inside each is a specific prober—for example, an `http` prober to check HTTP services and web pages, and an `icmp` prober to check for ICMP connectivity. Prometheus jobs supply targets to each check inside the module, and the exporter returns metrics that are then scraped.

Listing 10.9: The /etc/prober/prober.yml file

```
modules:
  http_2xx_check:
    prober: http
    timeout: 5s
    http:
      valid_status_codes: []
      method: GET
  icmp_check:
    prober: icmp
    timeout: 5s
    icmp:
      preferred_ip_protocol: "ip4"
  dns_examplecom_check:
    prober: dns
    dns:
      preferred_ip_protocol: "ip4"
      query_name: "www.example.com"
```

We've defined three checks: an HTTP check that ensures that a web server returns a 2XX status code when queried, an ICMP check that pings the target, and a DNS check that makes a DNS query. Let's look at each in turn.

 **TIP** The exporter example configuration is also useful to help explain how the exporter works.

HTTP check

Our HTTP status check uses the `http` prober. This prober makes HTTP requests using a variety of methods like `GET` or `POST`. We specify a timeout of `5s`, or five

seconds, for any requests. We then configure the prober to make a GET request. We leave the `valid_status_codes` blank; it defaults to any 2XX status code. If we wanted to validate that a different status code was returned, we'd specify the code or codes in an array in this field.

Listing 10.10: Valid status codes

```
valid_status_codes: [ '200', '304' ]
```

Here our check will be for the status codes 200 and 304. We could also check valid HTTP versions, if an HTTP connection is SSL, or if the content matches or does not match a regular expression.

ICMP check

Our second check pings a target using ICMP. We set the prober to `icmp` and specify a timeout of five seconds. We configure the `icmp` prober with the protocol to use, `ip4`.

 **TIP** The ICMP requires some additional permissions. On Windows it needs Administrator privileges, on Linux the `root` or `CAP_NET_RAW` capability, and on BSD or OS X the `root` user.

DNS check

Our last check uses the `dns` prober to check if a DNS entry resolves. In this case our target will be the DNS server we want to make the resolution. We specify our

preferred protocol, again ip4, and we specify a query.

```
query_name: "www.example.com"
```

This will check that DNS for the `www.example.com` site will resolve. A query type of ANY is made of the target, and a successful DNS probe relies on the status code returned for the query. The default indicator for success is if the NOERROR response code is received. We can configure for other query types using the `query_type` option and other response codes using the `valid_rcodes` option.

Starting the exporter

Now that we have our three checks defined, let's start the exporter. We're going to run our exporter on an Ubuntu host called `prober.example.com` running on an AWS EC2 instance. We run the `blackbox_exporter` binary and pass it the configuration file we've just created.

Listing 10.11: Starting the exporter

```
$ sudo blackbox_exporter --config.file="/etc/prober/prober.yml"
```

The exporter runs on port 9115, and you can browse to its console page at `http://localhost:9115`.

 | ⓘ localhost:9115

Blackbox Exporter

[Probe prometheus.io for http_2xx](#)

[Debug probe prometheus.io for http_2xx](#)

[Metrics](#)

[Configuration](#)

Recent Probes

Module	Target	Result	Debug

Figure 10.2: The blackbox exporter console

The console includes the exporter's own metrics, available on the `http://localhost:9115/metrics` path, to allow us to monitor it too. It also contains a list of recent checks executed, their status, and debug logs showing what happened. These are useful to debug checks if they have failed.

Creating the Prometheus job

Now we can create some Prometheus jobs to scrape the exporter. As we've discussed, the blackbox exporter is a little different in how it operates: the exporter scrapes the targets passed to it using the checks defined on it. We'll use a separate job for each check.

Let's create a job called `http_probe` that will query our `http_2xx_check` module.

Listing 10.12: The `http_probes` job

```
- job_name: 'http_probe'
  metrics_path: /probe
  params:
    module: [http_2xx_check]
  file_sd_configs:
    - files:
        - 'targets/probes/http_probes.json'
        refresh_interval: 5m
  relabel_configs:
    - source_labels: [__address__]
      target_label: __param_target
    - source_labels: [__param_target]
      target_label: instance
    - target_label: __address__
      replacement: prober.example.com:9115
```

We specify the metrics path, `/probe`, on the exporter. We pass in the module name as a parameter to the scrape. We're using file-based discovery to list targets for this job.

Listing 10.13: The http_probe targets

```
[{  
  "targets": [  
    "http://www.example.com",  
    "https://www.example.com",  
    ""  
  ]  
}]
```

We're going to probe one website, `www.example.com`, using both HTTP and HTTPS.

So how does Prometheus know how to find the exporter? We use `relabel_configs` to overwrite the `__address__` label of the target to specify the exporter's hostname. We do three relabels:

1. Our first relabel creates a parameter by writing the `__address__` label, the current target's address, into the `__param_target` label.
2. Our second relabel writes that `__param_target` label as the `instance` label.
3. Last, we relabel the `__address__` label using the host name (and port) of our exporter, in our case `prober.example.com`.

The relabeling results in a URL being constructed for the scrape:

```
http://prober.example.com:9115/probe?target=www.example.com?module=  
http_2xx_check
```

We can browse to this URL to see the metrics that will be returned. Here are the metrics, minus the comments.

Listing 10.14: The http_2xx_check metrics

```
probe_dns_lookup_time_seconds 0.404881857
probe_duration_seconds 0.626351441
probe_failed_due_to_regex 0
probe_http_content_length -1
probe_http_duration_seconds{phase="connect"} 0.01319281699999999
probe_http_duration_seconds{phase="processing"} 0.01394864700000002
probe_http_duration_seconds{phase="resolve"} 0.531245733
probe_http_duration_seconds{phase="tls"} 0.073685882
probe_http_duration_seconds{phase="transfer"} 0.000128069
probe_http_redirects 1
probe_http_ssl 1
probe_http_status_code 200
probe_http_version 1.1
probe_ip_protocol 4
probe_ssl_earliest_cert_expiry 1.527696449e+09
probe_success 1
```

The key metric here is `probe_http_status_code` which shows the status code returned by the HTTP request. If this is a 2xx status code, then the probe is considered successful and the `probe_success` metric will be set to 1. The metrics here also supply useful information like the time of the probe and the HTTP version.

The other jobs operate in a similar manner to our HTTP check. They use the same relabelling rules to find the right target and the exporter's address.



NOTE You'll find the source code for this chapter, including the Prometheus jobs, on GitHub.

The ICMP job takes host names or IP addresses, performs an ICMP ping, and re-

turns the results. The targets for the DNS check are the DNS servers whose resolutions you wish to test.

If we now reload or restart Prometheus, we'll see the metrics from these jobs in the console.

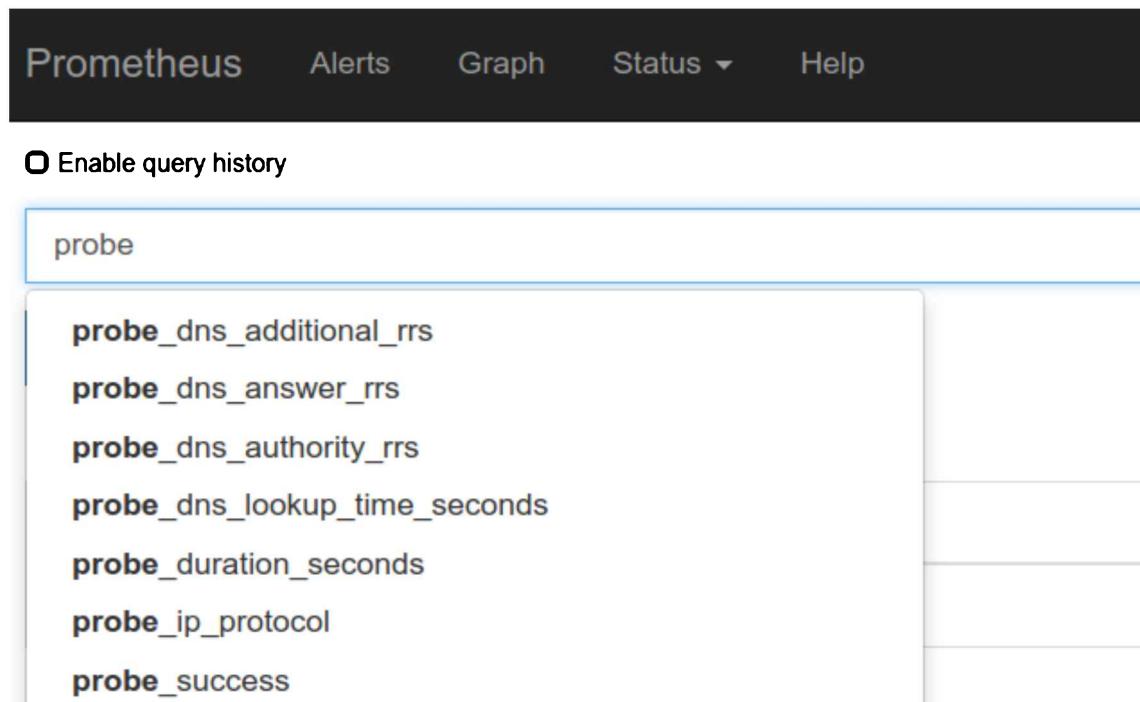


Figure 10.3: The probe metrics in Prometheus

Summary

In this chapter we used the blackbox exporter to probe some resources. We were introduced to the basics of probing architecture, and how to install, configure some basic probe checks with, and run the exporter. We also saw how to create Prometheus jobs to initiate probes and how to use relabelling rules to scrape the targets via the exporter.

In the next chapter we'll learn how to push metrics to Prometheus, especially for short-running processes like jobs and deployments, using the Pushgateway.

Chapter 11

Pushing Metrics and the Pushgateway

Up until now, we've seen the Prometheus server running jobs to scrape metrics from targets: a pull-based architecture. In some cases, though, there isn't a target from which to scrape metrics. There are a number of reasons why this might be so:

- You can't reach the target resources because of security or connectivity. This is quite a common scenario when a service or application only allows ingress to specific ports or paths.
- The target resource has too short a lifespan—for example, a container starting, executing, and stopping. In this case, a Prometheus job will run and discover the target has completed execution and is no longer available to be scraped.
- The target resource doesn't have an endpoint, such as a batch job, that can be scraped. It's unlikely that a batch job will have a running HTTP service that can be scraped, even assuming the job runs long enough to be available to be scraped.

In these cases we need some way to deliver or push our time series to the Prometheus server. In this chapter we're going to learn how to handle these scenarios using the Pushgateway.

The Pushgateway

The Pushgateway is a standalone service that receives Prometheus metrics on an HTTP REST API. The Pushgateway sits between an application sending metrics and the Prometheus server. The Pushgateway receives metrics and is then scraped as a target to deliver the metrics to the Prometheus server. You can think about it like a proxy service, or the opposite of the blackbox exporter's behavior: it's receiving metrics rather than probing for them.

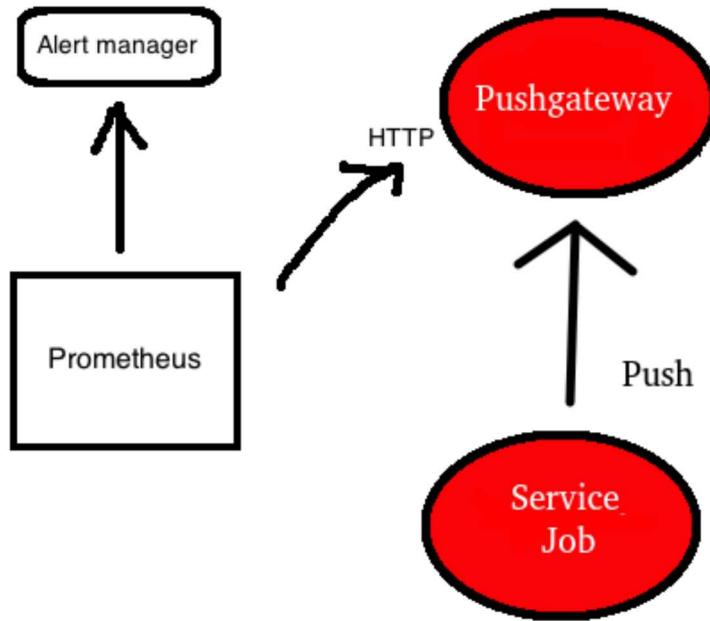


Figure 11.1: The Pushgateway

Like most of the Prometheus ecosystem, the Pushgateway is written in Go, open source, and licensed under Apache 2.0.

When not to use the Pushgateway

The Pushgateway is essentially a workaround for monitoring resources that can't be scraped by a Prometheus server for the reasons we discussed above. The gateway isn't a perfect solution and should only be used as a limited workaround, especially for monitoring otherwise inaccessible resources.

You also want to avoid making the gateway a single point of failure or a perfor-

mance bottleneck. The Pushgateway definitely does not scale in the same way a Prometheus server will scale.

 **TIP** Although you could use a variant of the worker scaling pattern we introduced in Chapter 7 or push metrics to multiple Pushgateways.

The gateway is also closer to a proxy than a fully featured push monitoring tool, hence, in using it, you lose a number of useful features that the Prometheus server provides. This includes instance state monitoring via up metrics and the expiration of metrics. It is also a static proxy by default, and remembers every metric sent to it, continuing to expose them as long as it is running (and the metrics aren't persisted) or until they are deleted. This means that metrics for instances that no longer exist may be persisted in the gateway.

You should focus the gateway on monitoring short-lifespan resources, like jobs, or the short-term monitoring of inaccessible resources. You should install Prometheus servers to monitor inaccessible resources in the longer term.

 **TIP** A useful tool to monitor some of these inaccessible resources is the PushProx proxy which is designed to allow scrapes through NAT'ed connections.

Installing the Pushgateway

The Prometheus.io download page contains zip files with the binaries for specific platforms. Currently Pushgateway 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 Pushgateway are available from the GitHub Releases page.



NOTE At the time of writing Pushgateway was at version 0.5.2.

Installing the Pushgateway on Linux

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

Listing 11.1: Download the Pushgateway zip file

```
$ cd /tmp  
$ wget  
https://github.com/prometheus/pushgateway/releases/download/v  
0.5.2/pushgateway-0.5.2.linux-amd64.tar.gz
```

Now let's unpack the `pushgateway` binary from the tarball and move it somewhere useful.

Listing 11.2: Unpack the pushgateway binary

```
$ tar -xzf pushgateway-0.5.2.linux-amd64.tar.gz  
$ sudo cp pushgateway-0.5.2.linux-amd64/pushgateway /usr/local/  
bin/
```

We can now test if the Pushgateway is installed and in our path by checking its version.

Listing 11.3: Checking the Pushgateway version on Linux

```
$ pushgateway --version  
pushgateway, version 0.5.2 (branch: HEAD, revision: 30  
dd0426c08b6479d9a26259ea5efd63bc1ee273)  
  build user:      root@3e103e3fc918  
  build date:     20171116-17:45:26  
  go version:    go1.9.2
```

 **TIP** This same approach will work on Mac OS X with the Darwin version of the Pushgateway binary.

Installing the Pushgateway on Microsoft Windows

To install Pushgateway on Microsoft Windows, we need to download the `pushgateway.exe` executable and put it in a directory. Let's create a directory for the executable using Powershell.

Listing 11.4: Creating a directory on Windows

```
C:\> MKDIR pushgateway  
C:\> CD pushgateway
```

Now download the `pushgateway.exe` executable from GitHub into the `C:\pushgateway` directory:

Listing 11.5: Pushgateway Windows download

```
https://github.com/prometheus/pushgateway/releases/download/v0.5.2/pushgateway-0.5.2.windows-amd64.tar.gz
```

Unzip the executable, using a tool like 7-Zip, into the `C:\pushgateway` directory. Finally, add the `C:\pushgateway` directory to the path. This will allow Windows to find the executable. To do this, run this command inside Powershell.

Listing 11.6: Setting the Windows path

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

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

Listing 11.7: Checking the Pushgateway version on Windows

```
C:\> pushgateway.exe --version
pushgateway, version 0.5.2 (branch: HEAD, revision: 30
dd0426c08b6479d9a26259ea5efd63bc1ee273)
  build user:      root@3e103e3fc918
  build date:     20171116-17:45:26
  go version:    go1.9.2
```

Installing via configuration management

Some of the configuration management modules we saw in Chapter 3 can also install the Pushgateway:

- A Puppet module for Prometheus.
- A Chef cookbook for Prometheus.

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

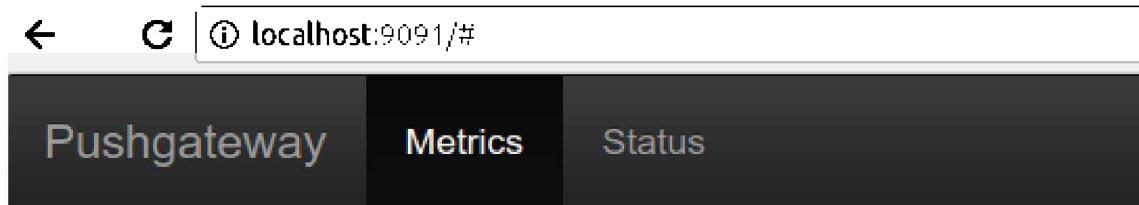
Configuring and running the Pushgateway

The Pushgateway doesn't need any configuration out of the box, but it can be configured by setting flags on the command line when you run the `pushgateway` binary. The gateway runs on port 9091, but you can override this port and any interface using the `--web.listen-address` flag.

Listing 11.8: Running the Pushgateway on an interface

```
$ pushgateway --web.listen-address="0.0.0.0:9091"
```

This will bind the Pushgateway on all interfaces. When the gateway is running, you can browse to its dashboard on that address and port.



Code Community © Prometheus Authors 2014

Figure 11.2: The Pushgateway dashboard

By default, the gateway stores all of its metrics in memory. This means if the gateway stops or is restarted you'll lose any metrics in memory. You can persist the metrics to disk by specifying the `--persistence.file` flag with a path a file.

Listing 11.9: Persisting the metrics

```
$ pushgateway --persistence.file="/tmp/pushgateway_persist"
```

The persistence file is written to every five minutes by default, but you can override this with the `--persistence.interval` flag.

You can see other available flags by running the binary with the `--help` flag.

Sending metrics to the Pushgateway

Once the Pushgateway is running you can start to send it metrics. Most Prometheus client libraries support sending metrics to the Pushgateway, in addition to exposing them for scraping. The easiest way to see how the gateway works is to use a command line tool like `curl` to post metrics. Let's push a single metric to our running gateway.

Listing 11.10: Posting a metric to the gateway

```
$ echo 'batchjob1_user_counter 2' | curl --data-binary @- http://localhost:9091/metrics/job/batchjob1
```

We push metrics to the path `/metrics`. The URL is constructed using labels, here `/metrics/job/<jobname>` where `batchjob1` is our job label. A full metrics path with labels looks like:

Listing 11.11: The Pushgateway metrics path

```
/metrics/job/<jobname>{/<label>/<label>}
```

The <jobname> will be used as the value of the job label, followed by any other specified labels. Labels specified in the path will override any labels specified in the metric itself.

Let's add an instance label to our metric using the URL path.

Listing 11.12: Posting a metric to the gateway

```
$ echo 'batchjob1_user_counter 2' | curl --data-binary @- http://localhost:9091/metrics/job/batchjob1/instance/sidekiq_server
```

⚠️ WARNING You cannot use / as part of a label value or job name, even if it is escaped. This is because the decoding sequences makes it impossible to determine what was escaped, see the Go URL documentation.

In the above example, we've echoed the metric batchjob1_user_counter 2 from the job batchjob1 to our gateway. This will create a new metric grouping for the job batchjob1 with an instance label of sidekiq_server. Metric groupings are collections of metrics. You can add and delete metrics within the grouping, or even delete the whole group. Because the gateway is a cache and not an aggregator, metric groupings will live on until the gateway is stopped or they are deleted.

This counter is the most simple metric we can send. We name the counter, batchjob1_user_counter, and give it a value: 2.

We can add labels to pushed metrics by enclosing them in {}.

Listing 11.13: Adding labels to pushed metrics

```
echo 'batchjob1_user_counter{job_id="123ABC"} 2' | curl --data-binary @- http://localhost:9091/metrics/job/batchjob1/instance/sidekiq_server
```

Currently, the metric will be uploaded untyped; the gateway won't know whether this is a counter, gauge, or any other metric type. You can add a type (and a description) to a metric by passing `TYPE` and `HELP` statements in the push.

Listing 11.14: Passing types and descriptions

```
$ cat <<EOF | curl --data-binary @- http://localhost:9091/metrics/job/batchjob1/instance/sidekiq_server
# TYPE batchjob1_user_counter counter
# HELP batchjob1_user_counter A metric from BatchJob1.
batchjob1_user_counter{job_id="123ABC"} 2
EOF
```

We can also add further metrics to our metric group.

Listing 11.15: Passing types and descriptions

```
$ cat <<EOF | curl --data-binary @- http://localhost:9091/metrics/job/batchjob1/instance/sidekiq_server
# TYPE batchjob1_avg_latency gauge
# HELP batchjob1_avg_latency Another metric from BatchJob1.
batchjob1_avg_latency{job_id="123ABC"} 74.5
# TYPE batchjob1_sales_counter counter
# HELP batchjob1_sales_counter A third metric from BatchJob1.
batchjob1_sales_counter{job_id="123ABC"} 1
EOF
```

This would add two metrics, `batchjob1_avg_latency` and `batchjob1_sales_counter`, to our `batchjob1` metric group.

Viewing metrics on the Pushgateway

We can then see the metrics we've pushed to the gateway by using `curl` on the `/metrics` path (or by browsing to the Pushgateway dashboard at `http://localhost:9091`).

Listing 11.16: Curling the gateway metrics

```
$ curl http://localhost:9091/metrics
# HELP batchjob1_user_counter A metric from BatchJob1.
# TYPE batchjob1_user_counter counter
batchjob1_{instance="sidekiq_server",job="batchjob1",job_id="123ABC"} 2
# HELP batchjob1_avg_latency Another metric from BatchJob1.
# TYPE batchjob1_avg_latency gauge
batchjob1_avg_latency{instance="sidekiq_server",job="batchjob1",job_id="123ABC"} 74.5
# HELP batchjob1_sales_counter A third metric from BatchJob1.
# TYPE batchjob1_sales_counter counter
batchjob1_sales_counter{instance="sidekiq_server",job="batchjob1",job_id="123ABC"} 1

. . .

# HELP push_time_seconds Last Unix time when this group was
changed in the Pushgateway.
# TYPE push_time_seconds gauge
push_time_seconds{instance="sidekiq_server",job="batchjob1"} 1.523303909484092e+09
# HELP pushgateway_build_info A metric with a constant '1' value
labeled by version, revision, branch, and goversion from which
pushgateway was built.
# TYPE pushgateway_build_info gauge
pushgateway_build_info{branch="master",goversion="g01.10.1",
revision="d07ed465fcfcf2be4a2d80026d057fb5944c9283",version="0.4.0"} 1
```

 **NOTE** We've skipped a number of health and performance metrics from the gateway in our output.

We can see our batchjob1 metrics. We can see the job label has been set to

batchjob1, and we have an instance label of sidekiq_server.

For batchjob1_user_counter, we can see that the value of the metric is 2 even though we sent three pushes to the gateway. This is because the gateway is NOT an aggregator, like StatsD or similar tools. Instead, the last push of the metric is exported until it is updated or deleted.

You'll also see another metric here: push_time_seconds. This is a per-job metric that indicates the last time a push occurred. You can use this metric to determine when the last push occurred, and to potentially identify missing pushes. It's only useful if you expect a push to occur within a specific time frame.

Deleting metrics in the Pushgateway

Metrics exist in the gateway until it is restarted (assuming no persistence is set), or until they are deleted. We can delete metrics using the Pushgateway API. Let's do this now, again using curl, as an example.

Listing 11.17: Deleting Pushgateway metrics

```
$ curl -X DELETE localhost:9091/metrics/job/batchjob1
```

This will delete all metrics for the job batchjob1. You can further limit the selection by making the path more granular—for example, by deleting only those metrics from a specific instance.

Listing 11.18: Deleting a selection of Pushgateway metrics

```
$ curl -X DELETE localhost:9091/metrics/job/batchjob1/instance/  
sidekiq_server
```

 **TIP** You can also delete an entire metrics grouping from the Pushgateway by using the Delete Group button on the dashboard.

Sending metrics from a client

Obviously curl’ing metrics to the gateway isn’t practical. Instead we’re going to use a Prometheus client to push metrics to the gateway. All of the official Prometheus clients, and many of the unofficial ones, support the push gateway as a target for metrics.

To demonstrate how to do this, we’ll use the Rails application we demonstrated in Chapter 8. We’re going to create a `MetricsPush` class in our `lib` directory to use in our Rails application. Let’s do that now.

Listing 11.19: Creating MetricsPush class

```
$ touch lib/metricspush.rb
```

And let’s populate the file with a class.

Listing 11.20: The MetricsPush module

```
require 'prometheus/client'
require 'prometheus/client/push'

class MetricsPush
  attr_reader :job, :registry, :pushgateway_url

  def initialize
    @job = 'mwp-rails'
    @pushgateway_url = 'http://localhost:9091'
  end

  def registry
    @registry ||= Prometheus::Client.registry
  end

  def counter(name, desc, labels = {})
    registry.get(name) || registry.counter(name, desc)
  end

  def gauge(name, desc, labels = {})
    registry.get(name) || registry.counter(name, desc)
  end

  def summary(name, desc, labels = {})
    registry.get(name) || registry.counter(name, desc)
  end

  def histogram(name, desc, labels = {}, buckets = Prometheus::
Client::Histogram::DEFAULT_BUCKETS)
    registry.get(name) || registry.counter(name, desc)
  end

  def push
    Prometheus::Client::Push.new(job, nil, pushgateway_url).add(
      registry)
  end
end
```

Our class is very similar to the code we used in Chapter 8. We can create a wide variety of metrics. We've also added a method called `push` that sends the metrics in the registry to a Pushgateway. In our case, we've assumed the gateway is running locally on the host.

\DN{In addition to the `add` method on `Prometheus::Client::Push`, we also have `replace` and `delete` methods we could use to replace or delete a metric on the gateway.}

We can then use our class when we run a job or some other transitory task.

Listing 11.21: Pushing a metric

```
mp = MetricsPush.new
mp.counter(:test_counter, "A test counter for a job").increment({
  service: 'mwp-rails-job' })
mp.push
```

We create an instance of the `MetricsPush` class and increment a counter called `test_counter`. We've added a label—one called `service` with a value of `mwp-rails-job`. We then use the `push` method to push the metric. If we were to run this snippet of code we could then check for the metric in the gateway, on the `http://localhost:9091/metrics` path or in the Pushgateway console.



Figure 11.3: The `test_counter` in the Pushgateway dashboard

Here we can see our pushed metric. Its instance label has been automatically populated with the IP address of our Rails server. We can override this during the push if required (especially as a lot of short-lived jobs are more likely associated with a service than a specific host). We see a job label and the service label we added.

Now we're ready to have the Prometheus server scrape the gateway to acquire our metrics.

Scraping the Pushgateway

The Pushgateway is only the interim stop for our metrics. We now need to get them into the Prometheus server. For that we're going to need a job. Let's create one now.

Listing 11.22: The pushgateway job

```
- job_name: pushgateway
  honor_labels: true
  file_sd_configs:
    - files:
      - targets/pushgateway/*.json
  refresh_interval: 5m
```

We can see our job is pretty typical and follows the pattern we've seen throughout the book, using file-based discovery. The job will load all targets specified in JSON files in the targets/pushgateway directory. We've specified a Pushgateway named pg1.example.com in our file-based service discovery configuration.

Listing 11.23: Our Pushgateway

```
[{
  "targets": ["pg1.example.com"]
}]
```

We've then specified the `honor_labels` option and set it to `true`. As we've learned, when Prometheus scrapes a target, it will attach the name of the job that did the scraping, here `pushgateway`, and an `instance` label populated with the host or IP address of the target. With the Pushgateway, our metrics already have `job` and `instance` labels that indicate where our metrics were pushed from. We want to perpetuate this information in Prometheus rather than have it rewritten by the server when it scrapes the gateway.

If `honor_labels` is set to `true`, Prometheus will use the `job` and `instance` labels on the Pushgateway. Set to `false`, it'll rename those values, prefixing them with `\texttt{\{exported_}}` and attaching new values for those labels on the server.

If we restart Prometheus it'll start scraping the gateway. Let's now look for our `test_counter` metric.



Figure 11.4: The `test_counter` metric

We can see it's been scraped, and the Prometheus server has honored the local labels.

Summary

In this chapter we saw how to use “push” mechanics with Prometheus via the Pushgateway. We articulated the limited circumstances in which it’s an appropriate use case. In those circumstances, we showed you how to install and configure the gateway and instrument your applications and jobs to push metrics to the gateway. And finally, we saw how to use a Prometheus job to scrape the gateway and acquire your pushed metrics.

In the next two chapters we’ll look at monitoring a whole application stack running on top of Kubernetes, first looking at Kubernetes, and then a multi-service application.

Chapter 12

Monitoring a Stack - Kubernetes

Now that we've got a handle on the building blocks of Prometheus, let's put the pieces together to monitor a modern application stack in the real world. To do this, we're going to monitor an API service application called Tornado. Tornado is written in Clojure, and runs on the JVM; it has a Redis data store and a MySQL database. We're going to deploy Tornado into a Kubernetes cluster we've built, so we'll also look at monitoring Kubernetes with Prometheus.

In this chapter, we're going to examine the Kubernetes portion of our stack and how to monitor it.

To make our monitoring simpler we've deployed Prometheus onto Kubernetes, too.

Our Kubernetes cluster

Our Kubernetes cluster is named `tornado.quicknuke.com`. The cluster is running Kubernetes 1.8.7 and is running in AWS. We built the cluster with kops, and you can find the cluster configuration [here](#). It has three masters and six worker nodes. All nodes are divided between three Availability Zones.

The cluster was created with the following kops command.

Listing 12.1: The cluster kops command

```
$ kops create cluster \
  --node-count 6 \
  --zones us-east-2a,us-east-2b,us-east-2c \
  --master-zones us-east-2a,us-east-2b,us-east-2c \
  --node-size t2.micro \
  --master-size t2.micro \
  --topology private \
  --networking kopeio-vxlan \
  --api-loadbalancer-type=public \
  --bastion \
  tornado.quicknuke.com
```

 **NOTE** This chapter assumes you've already installed Kubernetes and have some understanding of how it works. If you need more information on Kubernetes, the book *Kubernetes: Up and Running* is recommended reading.

Running Prometheus on Kubernetes

There are a variety of ways to deploy Prometheus on Kubernetes. The best way for you likely depends greatly on your environment. As possibilities, you can build your own deployments and expose Prometheus via a service, use one of a number of bundled configurations, or use the Prometheus Operator from CoreOS.

We've chosen to manually create a deployment and a service. We configure the Prometheus server and manage rules using ConfigMaps and mount these as volumes in our deployment. We also expose the Prometheus WebUI via an AWS Load

Balancer service.

We've also installed a cluster of three Alertmanagers running on the cluster. We've enclosed all of this in a namespace called `monitoring`.

You can find the configuration for all of this with the book's code on GitHub. However, we're not going to go into huge detail about how we deployed Prometheus onto Kubernetes; instead we'll focus on monitoring Kubernetes and applications running on Kubernetes with Prometheus.

 **NOTE** This decision also reflects the speed at which Kubernetes is evolving: any deployment documented here is likely to be dated very quickly. This configuration provided here is not guaranteed to work for later Kubernetes releases.

Monitoring Kubernetes

Let's start by talking about monitoring Kubernetes itself. While it's likely to change, it's more manageable as a topic. Kubernetes is a container orchestrator and scheduler with a lot of moving pieces. We're going to show you how to monitor aspects of Kubernetes with Prometheus jobs, and we'll match each of these jobs with some recording and alert rules.

This chapter will be broken into sections dealing with each piece, how the time series are collected, and any rules and alerts we're going to generate from those time series. We're not going to provide the definitive monitoring approach, but instead touch on some key highlights, especially where they expand on a Prometheus concept worth exploring.

To identify what we need to monitor we'll also make use of Prometheus's built-in service discovery mechanism for Kubernetes.

Let's start with monitoring the nodes upon which Kubernetes is running.

Monitoring our Kubernetes nodes

Our Kubernetes cluster is made up of nine AWS EC2 instances. To monitor them we're going to use the Node Exporter. There are several ways we can deploy the Node Exporter onto those instances. We can install the Node Exporter onto the base instances when they are provisioned, much as we did in Chapter 4. Or we can install the Node Exporter into a Kubernetes pod on each node. We can take advantage of Kubernetes DaemonSet controller that automatically deploys a pod on every node in the cluster. This approach is useful when you don't control the base instances—for example, if you're using a hosted Kubernetes solution.



WARNING There is a major caveat with this approach. The Node Exporter accesses a lot of root-level resources, and running it in a Docker container requires mounting those resources into the container and, for the systemd collector, running the container as root. This poses a potential security risk. If that risk isn't acceptable to you then you should install Node Exporter directly onto the instances.

Node Exporter DaemonSet

A DaemonSet ensures that a pod runs on all nodes, potentially including the masters, using a toleration. It's ideal for items like monitoring or logging agents. Let's look at some elements of our DaemonSet.

 **NOTE** You can find the full configuration for the Node Exporter on GitHub.

Listing 12.2: The Node Exporter DaemonSet tolerations

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: node-exporter
  namespace: monitoring

  ...
  spec:
    tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
    hostNetwork: true
    hostPID: true
    hostIPC: true
    securityContext:
      runAsUser: 0
```

First, you can see we've specified a DaemonSet with a name, `node-exporter`, and that we're using a toleration to ensure this pod is also scheduled on our Kubernetes masters, not just our normal nodes.

Now here's the caveat with this approach. We're running the pod as user 0 or root (this allows access to `systemd`). We've also enabled `hostNetwork`, `hostPID`, and `hostIPC` to specify that the network, process, and IPC namespace of the instance will be available in the container. This is a potential security exposure, and you must definitely consider if you want to take this risk. If this risk isn't acceptable, baking the Node Exporter into the image of your instances is potentially a better approach.

Let's look at the containers in the pod.

Listing 12.3: The Node Exporter DaemonSet containers

```
containers:
- image: prom/node-exporter:latest
  name: node-exporter
  volumeMounts:
    - mountPath: /run/systemd/private
      name: systemd-socket
      readOnly: true
  args:
    - "--collector.systemd"
    - "--collector.systemd.unit-whitelist=(docker|ssh|
      rsyslog|kubelet).service"
  ports:
    - containerPort: 9100
      hostPort: 9100
      name: scrape
```

Here we're using the DockerHub image for Node Exporter, `prom/node_exporter`, and grabbing the latest release. We're also mounting in a volume for the `/run/systemd/private` directory on the instances themselves. This allows the Node Exporter to access the `systemd` state and gather the service state of `systemd`-managed services on the instance.

We've also specified some arguments for the `node_exporter` binary. We saw both in Chapter 4: enabling the `systemd` collector, and specifying a regular expression of the specific services to monitor, rather than all the services on the host.

We've also specified the port we want our metrics exposed on, 9100: the default port.

To help keep the Node Exporter pods healthy and to enhance their uptime, we've also added liveness and readiness probes to our Node Exporter container. Liveness probes detect the status of applications inside containers.

Listing 12.4: Node Exporter liveness and readiness probes

```
livenessProbe:  
  httpGet:  
    path: /metrics  
    port: 9100  
  initialDelaySeconds: 30  
  timeoutSeconds: 10  
  periodSeconds: 1  
readinessProbe:  
  failureThreshold: 5  
  httpGet:  
    path: /metrics  
    port: 9100  
  initialDelaySeconds: 10  
  timeoutSeconds: 10  
  periodSeconds: 2
```

In our case we use an HTTP GET probe to the `/metrics` path on port 9100 to confirm the Node Exporter is still working. The probe runs every `periodSeconds`, one second in our case. If the liveness check fails, Kubernetes will restart the container.

 **NOTE** We'll see these probes in applications we monitor too. They can assist in managing the health of your applications by reducing possible false positives—such as a service triggering an alert by not being ready while it is starting—while monitoring. These checks can also restart containers that are faulty, potentially fixing issues before they trigger alerts.

Readiness probes confirm the application is functional. Here, that means an HTTP GET can connect to the `/metrics` path on port 9100 before marking the container

as available and delivering traffic to it. The remaining settings control the probe's behavior: it'll wait 10 seconds, the `initialDelaySeconds` setting, before checking the readiness; thereafter it will check every two seconds, the `periodSeconds` value, for readiness. If the probe times out after 10 seconds, the `timeoutSeconds`, more than five times, garnered from the `failureThreshold` setting, then the container will be marked as Unready.



NOTE You can find the full configuration for the Node Exporter on GitHub.

Node Exporter service

We also need a service to expose the Node Exporter so it can be scraped.

Listing 12.5: The Node Exporter service

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    prometheus.io/scrape: 'true'
  labels:
    app: node-exporter
    name: node-exporter
  name: node-exporter
  namespace: monitoring
spec:
  clusterIP: None
  ports:
    - name: scrape
      port: 9100
      protocol: TCP
  selector:
    app: node-exporter
  type: ClusterIP
```

Our service is relatively straightforward. We add an annotation, `prometheus.io /scrape: 'true'`, as metadata on the services. This will tell Prometheus that it should scrape this service. We'll see how it's used in the Prometheus job we'll create to scrape our Node Exporters.

We also expose port 9100 as a ClusterIP. This means it is only available to the internal cluster network. As Prometheus is on the local Kubernetes cluster it'll be able to internally scrape the Node Exporter, and there's no need to expose it externally.



NOTE You can find the complete Node Exporter service on GitHub.

Deploying the Node Exporter

Let's create our DaemonSet and service on our Kubernetes cluster using the `kubectl` command. We'll create both inside the `monitoring` namespace.

Listing 12.6: Deploying the Node Exporter daemonset and service

```
$ kubectl create -f ./node-exporter.yml -n monitoring
daemonset "node-exporter" created
service "node-exporter" created
```

If you don't want to keep specifying the `-n monitoring` namespace you can specify a default using.

Listing 12.7: The default namespace

```
$ kubectl config set-context $(kubectl config current-context) --
namespace=monitoring
```

We can now check our pods are running.

Listing 12.8: Checking the Node Exporter pods

```
$ kubectl get pods -n monitoring
NAME                               READY   STATUS    RESTARTS   AGE
alertmanager-6854b5d59b-jvjcw     1/1    Running   0          7d
node-exporter-4fx57                1/1    Running   0          5s
node-exporter-4nzfk                1/1    Running   0          5s
node-exporter-5n7kl                1/1    Running   0          5s
node-exporter-f2mvb                1/1    Running   0          5s
node-exporter-km7sc                1/1    Running   0          5s
node-exporter-lvrsq                1/1    Running   0          5s
node-exporter-mvstg                1/1    Running   0          5s
node-exporter-tj4cs                1/1    Running   0          5s
node-exporter-wh56c                1/1    Running   0          5s
prometheus-core-785bc8584b-7vfr4  1/1    Running   0          8d
```

We can see nine pods, one for each instance in the cluster: three masters and six nodes. We can also see our Prometheus server pod, `prometheus-core`, and our Alertmanager, `alertmanager`.

We can check the Node Exporter pods are running correctly by grabbing their logs.

Listing 12.9: A Node Exporter pod's logs

```
$ kubectl logs node-exporter-4fx57 -n monitoring
time="2018-01-18T22:46:05Z" level=info msg="Starting
node_exporter (version=0.15.2, branch=HEAD, revision=98
bc64930d34878b84a0f87dfe6e1a6da61e532d)" source="node_exporter.
go:43"
time="2018-01-18T22:46:05Z" level=info msg="Build context (go=
go1.9.2, user=root@d5c4792c921f, date=20171205-14:50:53)" source
="node_exporter.go:44"

.
```

We can see our Node Exporter daemon is running. We can also confirm our service

is in place.

Listing 12.10: Checking the Node Exporter service

```
$ kubectl get services -n monitoring
NAME           TYPE      CLUSTER-IP EXTERNAL-IP PORT(S) AGE
node-exporter   ClusterIP None        <none>     9100/TCP 8s
```

Here we can see our node-exporter service with a ClusterIP type and with the 9100 port exposed to the internal Kubernetes cluster, ready to be scraped. We're not scraping it yet, however, because we haven't added a Prometheus job.

The Node Exporter job

In our Prometheus configuration we now want to add a job to scrape our Node Exporter endpoints. We're going to kill many birds with one stone by defining a job that scrapes all the service endpoints that Kubernetes exposes. We're going to control which endpoints Prometheus actually scrapes by only scraping those with a specific annotation, `prometheus.io/scrape`, set to '`true`'. We'll also use the built-in Kubernetes service discovery to find our endpoints and return them as potential targets to Prometheus.

 **NOTE** All of these jobs are derived or based on the amazing example Kubernetes jobs shipped with Prometheus. Thanks to the contributors to that project for developing them.

Let's look at that job now.

Listing 12.11: The Kubernetes service endpoints job

```
- job_name: 'kubernetes-service-endpoints'
  kubernetes_sd_configs:
    - role: endpoints
      relabel_configs:
        - source_labels: [
            __meta_kubernetes_service_annotation_prometheus_io_scrape]
          action: keep
          regex: true
        - source_labels: [
            __meta_kubernetes_service_annotation_prometheus_io_scheme]
          action: replace
          target_label: __scheme__
          regex: (https?)
        - source_labels: [
            __meta_kubernetes_service_annotation_prometheus_io_path]
          action: replace
          target_label: __metrics_path__
          regex: (.)
        - source_labels: [__address__,
            __meta_kubernetes_service_annotation_prometheus_io_port]
          action: replace
          target_label: __address__
          regex: ([^:])(?::\d+)?;(\d+)
          replacement: $1:$2
        - action: labelmap
          regex: __meta_kubernetes_service_label_(.+)
        - source_labels: [__meta_kubernetes_namespace]
          action: replace
          target_label: kubernetes_namespace
        - source_labels: [__meta_kubernetes_service_name]
          action: replace
          target_label: kubernetes_name
```

We've called the job `kubernetes-service-endpoints`. We've specified service discovery using the `kubernetes_sd_discovery` mechanism. This is an inbuilt service discovery mechanism, specifically for Kubernetes. It queries the Kubernetes API

for targets that match specific search criteria.

As our Prometheus server is running inside Kubernetes we're able to automatically, with minimal configuration, fetch Kubernetes targets that match specific roles. There are roles for nodes, pods, services, and ingresses. Here, specified by the `role` parameter, we're asking our service discovery to return all the Kubernetes endpoints. The `endpoints` role returns targets for all listed endpoints of a service, with one target per port for each endpoint address. If the endpoint is backed by a pod, as our Node Exporter service is, then any additional container ports are also discovered as targets. In our case, we've only exposed port 9100.

Service discovery also populates a variety of metadata. We use this metadata to relabel and identify each endpoint. Let's see what our relabelling rules do and explore that metadata.

Our first rule checks the `prometheus.io/scrape`: 'true' annotation that we set in our Node Exporter service. During the service discovery process the `prometheus.io/scrape` annotation will be translated to `prometheus_io_scrape` to create a valid label name. This is because the dot and slash are not legal characters in a Prometheus metric label. Since this is an annotation on a Kubernetes service, the Prometheus service process also adds the prefix `_meta_kubernetes_service_annotation_` to the label.

Our job only keeps any targets that have the metadata label: `_meta_kubernetes_service_annotation_` set to true. All other targets are dropped. This lets you only scrape those endpoints that you want.

The next three rules check for the presence of more annotations: `prometheus.io/scheme`, `prometheus.io/path`, and `prometheus.io/port`. If these labels are present it'll use the contents of these annotations as the scheme, path, and port to be scraped. This lets us control, from the service endpoint, what precisely to scrape, allowing our job to be flexible.

Our next rule maps any labels on the service into Prometheus labels of the same name by using the `labelmap` action. In our case, this consumes the

`__meta_kubernetes_service_label_app` metadata label, which will become a label simply called `app`. Our next rule copies the `__meta_kubernetes_namespace` label as `kubernetes_namespace` and the `__meta_kubernetes_service_name` metadata label to `kubernetes_name`.

We now add our job to the ConfigMap we're using for our Prometheus server configuration. We then replace our existing configuration.

Listing 12.12: Replacing the ConfigMap

```
$ kubectl replace -f ./prom-config-map-v1.yml -n monitoring
```

We generally have to delete our Prometheus pod and allow it to be recreated in order to load our new configuration. Shortly, we should see some new targets on the Prometheus expression browser.

kubernetes-service-endpoints (13/13 up) show less

Endpoint	State	Labels	Last Scrape
http://100.96.3.2:9090/metrics	UP	<code>app="prometheus"</code> <code>component="core"</code> <code>instance="100.96.3.2:9090"</code> <code>kubernetes_name="prometheus"</code> <code>kubernetes_namespace="monitoring"</code>	4.693s ago
http://100.96.4.2:9093/metrics	UP	<code>app="alertmanager"</code> <code>component="core"</code> <code>Instance="100.96.4.2:9093"</code> <code>kubernetes_name="alertmanager-webui"</code> <code>kubernetes_namespace="monitoring"</code>	11.389s ago
http://100.96.5.3:9093/metrics	UP	<code>app="alertmanager"</code> <code>component="core"</code> <code>Instance="100.96.5.3:9093"</code> <code>kubernetes_name="alertmanager-webui"</code> <code>kubernetes_namespace="monitoring"</code>	6.569s ago
http://100.96.6.2:9093/metrics	UP	<code>app="alertmanager"</code> <code>component="core"</code> <code>Instance="100.96.6.2:9093"</code> <code>kubernetes_name="alertmanager-webui"</code> <code>kubernetes_namespace="monitoring"</code>	11.367s ago
http://172.20.112.25:9100/metrics	UP	<code>app="node-exporter"</code> <code>Instance="172.20.112.25:9100"</code> <code>kubernetes_name="node-exporter"</code> <code>kubernetes_namespace="monitoring"</code>	10.516s ago

Figure 12.1: The Kubernetes endpoint targets

You can see that we've got thirteen targets listed. Nine of them are the Node Exporter endpoints on our instances. The tenth and eleventh are Prometheus and Alertmanager. The Prometheus and Alertmanager targets have been discovered automatically because their interfaces are exposed as a service too.

Listing 12.13: The monitoring services

```
$ kubectl get services -n monitoring
NAME           TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)
alertmanager   LoadBalancer 100.68.82.44  a6f953a641191...
9093:32288/TCP 27m
node-exporter  ClusterIP    None          <none>         9100/TCP
15h
prometheus    LoadBalancer 100.68.154.121 a953a66970c13...
9090:30604/TCP 4d
```

This job is really useful because we only need to define it once and all future Kubernetes service endpoints will be automatically discovered and monitored. We'll see this in action in this and the next chapter.

We will also see `node_time` series start to appear in the expression browser soon after the job is loaded.

Node Explorer rules

We're not going to add any new recording or alert rules for our Kubernetes nodes. Rather we've added the rules we created in Chapter 4 to the ConfigMap we're using to populate Prometheus's rule files. So we're adding all the CPU, memory, and disk rules we created, and we're also adding some availability alert rules for our Kubernetes services. Let's look at those now.

Listing 12.14: Kubernetes availability alerting rules

```
- alert: KubernetesServiceDown
  expr: up{job="kubernetes-service-endpoints"} == 0
  for: 10m
  labels:
    severity: critical
  annotations:
    summary: Pod {{ $labels.instance }} is down!
- alert: KubernetesServicesGone
  expr: absent(up{job="kubernetes-service-endpoints"})
  for: 10m
  labels:
    severity: critical
  annotations:
    summary: No Kubernetes services are reporting!
    description: Werner Heisenberg says - OMG Where are my
servicez?
```

The first alert triggers when the value of the `up` metric for the `kubernetes-service-endpoints` job is 0. This indicates that Prometheus has failed to scrape a service. The second alert caters for a service disappearing and uses the `absent` function to check for the presence of the `up` metric.

We've also added alert rules for the services we're monitoring on individual nodes using the `node_systemd_unit_state` metric, which tracks the status of `systemd` services.

Listing 12.15: Kubernetes availability alerting rules

```
- alert: CriticalServiceDown
  expr: node_systemd_unit_state{state="active"} != 1
  for: 2m
  labels:
    severity: critical
  annotations:
    summary = {{ $labels.instance }}: Service {{ $labels.name }} failed to start.
    description = {{ $labels.instance }} failed to (re)start service {{ $labels.name }}.
```

This will alert when it detects that any of the services our Node Exporter is monitoring—Docker, Kubelet, RSyslog, and SSH—are in a failed state.

There are other rules and alerts in the configuration that you can explore and adapt for node monitoring.

Now let's look at monitoring some Kubernetes components.

Kubernetes

There are a number of ways to monitor Kubernetes itself. These include tools in the open-source Kubernetes ecosystem like Heapster and Kube-state-metrics as well as commercial and SaaS-based tools. In this chapter, we're going to focus on Kube-state-metrics to do our monitoring.

Kube-state-metrics

We'll install Kube-state-metrics on our Kubernetes cluster using a deployment and service. The deployment uses the Kube-state-metrics Docker image and runs it on

one of our nodes. The service exposes the metrics on port 8080. As it's a service, it allows us to take advantage of our existing Prometheus service job we created in the last section. When we run it, Prometheus will automatically discover the new service endpoint and start scraping the Kube-state-metrics.

Once we've added the service we'll see a new target in the kubernetes-service-endpoints job in the <http://prometheus.quicknuke.com:9090/targets> listing.

http://100.96.8.2:8080/metrics	UP	app="kube-state-metrics" Instance="100.96.8.2:8080" kubernetes_name="kube-state-metrics" kubernetes_namespace="monitoring"
--------------------------------	----	--

Figure 12.2: The Kube-state-metrics endpoint target

With Kube-state-metrics we're going to focus on the success and failure of the workloads we're deploying to Kubernetes and the state of our nodes. Let's look at some alerts for which we can use our Kube-state-metrics time series.

 **TIP** You can see a full list of the metrics that Kube-state-metrics produces in its documentation.

Listing 12.16: Kube-state-metrics deployment generation alert

```
- alert: DeploymentGenerationMismatch
  expr: kube_deployment_status_observed_generation != kube_deployment_metadata_generation
  for: 5m
  labels:
    severity: warning
  annotations:
    description: Observed deployment generation does not match expected one for
      deployment {{$labels.namespace}}/{{$labels.deployment}}
    summary: Deployment is outdated
```

Our first rule detects if a deployment has succeeded. It compares the running generation of a deployment with the generation in the metadata. If the two are not equal for five minutes then an alert is raised indicating that a deployment has failed.

Our second rule does similar but for deployment replicas.

Listing 12.17: Kube-state-metrics Deployment replicas not updated alert

```

- alert: DeploymentReplicasNotUpdated
  expr: ((kube_deployment_status_replicas_updated != kube_deployment_spec_replicas)
    or (kube_deployment_status_replicas_available != kube_deployment_spec_replicas))
    unless (kube_deployment_spec_paused == 1)
  for: 5m
  labels:
    severity: warning
  annotations:
    description: Replicas are not updated and available for deployment {{$labels.namespace}}/{{$labels.deployment}}
    summary: Deployment replicas are outdated
  
```

Here we perform a more complex expression that confirms that either the updated or available replicas should match the number of replicas in the deployment specification, assuming the deployment isn't paused.

Our next rule checks for pod restarts.

Listing 12.18: Kube-state-metrics pod restarting alert

```

- alert: PodFrequentlyRestarting
  expr: increase(kube_pod_container_status_restarts_total[1h]) > 5
  for: 10m
  labels:
    severity: warning
  annotations:
    description: Pod {{ $labels.namespace }}/{{ $labels.pod }}
    was restarted {{ $value }}
      times within the last hour
    summary: Pod is restarting frequently
  
```

Here we measure the number of pod restarts using the `increase` function. The `increase` function measures the rate of increase in a time series in range vector, here one hour. If the rate is over five for 10 minutes then the alert is raised.

There are a number of other time series we can use to monitor Kubernetes. For example, we could use the `kube_node_status_condition` to determine the availability of the Kubernetes' nodes. You'll find some additional alerts in the alert rules we're creating for this chapter.

Kube API

We also want to create a job to monitor our Kubernetes API itself. The metrics associated with the API will form the central core of our Kubernetes monitoring, allowing us to monitor latency, error rate, and availability for our cluster. We're going to monitor the Kubernetes API specifically looking at latency, errors, and availability. Let's create a job to monitor the API now.

Listing 12.19: API server job

```
- job_name: 'kubernetes-apiservers'
  scheme: https
  tls_config:
    ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
    insecure_skip_verify: true
    bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  kubernetes_sd_configs:
    - role: endpoints
      relabel_configs:
        - source_labels: [__meta_kubernetes_namespace,
                        __meta_kubernetes_service_name,
                        __meta_kubernetes_endpoint_port_name]
          action: keep
          regex: default;kubernetes;https
```

We've called our job `kubernetes-apiservers`. We use `https` to scrape the metrics, and to specify the certification authority and a local token file to authenticate to Kubernetes. We again use Kubernetes discovery, this time to return a list of the Kubernetes endpoints. We won't use all of the endpoints, and our relabelling configuration uses the `keep` action to only retain services named `kubernetes` in the `default` namespace—which will only be our master Kubernetes nodes running the API.

Now that we're collecting API server metrics, let's create some recording rules to calculate the latency of the API servers.

Listing 12.20: The API server recording rules

```
- record: apiserver_latency_seconds:quantile
  expr: histogram_quantile(0.99, rate(
    apiserver_request_latencies_bucket[5m])) / 1e+06
  labels:
    quantile: "0.99"
- record: apiserver_latency_seconds:quantile
  expr: histogram_quantile(0.9, rate(
    apiserver_request_latencies_bucket[5m])) / 1e+06
  labels:
    quantile: "0.9"
- record: apiserver_latency_seconds:quantile
  expr: histogram_quantile(0.5, rate(
    apiserver_request_latencies_bucket[5m])) / 1e+06
  labels:
    quantile: "0.5"
```

We make use of the `apiserver_request_latencies_bucket` metric to calculate our latency. This bucket metric, with dimensions for the specific API resource, sub-resource, and verb, measures request latency. We've created three rules for the 50th, 90th, and 99th percentiles, setting the quantile to the specific percentile. We've used the `histogram_quantile` function to create the percentiles from the metric buckets. We've specified the percentile we're seeking, 0.99 for example, and then calculated a `rate` over a five minute vector and divided the result by `1e+06` or 1,000,000 to get microsecond latency.

We can then make use of the latency histograms our recording rules have created to create alerts. Let's start with an alert to detect high latency from the API.

Listing 12.21: API high latency alert

```

- alert: APIHighLatency
  expr: apiserver_latency_seconds:quantile{quantile="0.99",
subresource!="log",verb!~"^(?:WATCH|WATCHLIST|PROXY|CONNECT)$"} >
  4
  for: 10m
  labels:
    severity: critical
  annotations:
    description: the API server has a 99th percentile latency of
{{ $value }} seconds for {{ $labels.verb }} {{ $labels.resource
}}
  }
```

Our alert uses the `apiserver_latency_seconds:quantile` metric we just created. We use labels to select the 99th percentile, any sub-resource that isn't `log`, and any verb that isn't `WATCH`, `WATCHLIST`, `PROXY`, or `CONNECT`. If the latency of any of the remaining metrics exceeds 4 for 10 minutes then the alert will be raised.

Our next alert detects high levels of error rates in the API server.

Listing 12.22: API high error rate alert

```

- alert: APIServerErrorsHigh
  expr: rate(apiserver_request_count{code=~"^(?:5..)$"}[5m]) /
rate(apiserver_request_count[5m]) * 100 > 5
  for: 10m
  labels:
    severity: critical
  annotations:
    description: API server returns errors for {{ $value }}% of
requests
  
```

This alert calculates the error rate on API requests, using a regular expression to match any errors beginning with `5xx`. If the percentage rate over a five minute

vector exceeds 5 percent then the alert will be raised.

Our last two alerts monitor the availability of the API server, monitoring the `up` metrics and the presence or absence of the `up` metric.

Listing 12.23: API servers down or absent

```
- alert: KubernetesAPIServerDown
  expr: up{job="kubernetes-apiservers"} == 0
  for: 10m
  labels:
    severity: critical
  annotations:
    summary: Apiserver {{ $labels.instance }} is down!
- alert: KubernetesAPIServersGone
  expr: absent(up{job="kubernetes-apiservers"})
  for: 10m
  labels:
    severity: critical
  annotations:
    summary: No Kubernetes apiservers are reporting!
    description: Werner Heisenberg says - OMG Where are my
    apiserverz?
```

Last, we can monitor the Kubernetes nodes and the Docker daemons and containers running on them.

CAdvisor and Nodes

Kubernetes also has CAdvisor and node-specific time series available by default. We can create a job to scrape these time series from the Kubernetes API for each node. We can use these time series, much as we did in Chapter 4, to monitor the nodes, and the Docker daemons and container-level on each node.

Let's add a job for CAdvisor.

Listing 12.24: The CAdvisor job

```

- job_name: 'kubernetes-cadvisor'
  scheme: https
  tls_config:
    insecure_skip_verify: true
    ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
  bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  kubernetes_sd_configs:
    - role: node
      relabel_configs:
        - action: labelmap
          regex: __meta_kubernetes_node_label_(.+)
        - target_label: __address__
          replacement: kubernetes.default.svc:443
        - source_labels: [__meta_kubernetes_node_name]
          regex: (.+)
        target_label: __metrics_path__
        replacement: /api/v1/nodes/${1}/proxy/metrics/cadvisor

```

We've called our job `kubernetes-cadvisor` and used service discovery to return a list of the Kubernetes nodes using the `node` role. We use `https` to scrape the metrics, and to specify the certification authority and a local token file to authenticate to Kubernetes.

We're then relabelling our time series to create labels from the metadata labels we've discovered using `labelmap`. We replace the `__address__` label with the default DNS name of the Kubernetes API server. We then use one of the metadata labels, a label with the name of the node, to create a new `__metrics_path__` on the API that passes in the node name to the path.

`/api/v1/nodes/${1}/proxy/metrics/cadvisor`

This will scrape the required time series for each node discovered by the job. We also have a job for the nodes themselves in our configuration that exposes some

Kubernetes node-level metrics.

We can use these metrics to monitor the performance of the underlying containers, Docker daemons and the Kubernetes-level performance of the nodes.

Summary

In this chapter we started looking at monitoring a stack, starting with our compute platform: Kubernetes. We installed Prometheus onto Kubernetes to make our monitoring easier. We looked at monitoring Kubernetes nodes and the nodes upon which they are deployed using the Node Exporter.

We created several Prometheus jobs, including several that use Kubernetes service discovery to automatically discover the nodes, API servers, and services that make up our environment. The service discovery also allows us to configure jobs that automatically begin scraping specific Kubernetes or application services as they appear, using annotations to select the right addresses, ports, and paths.

In the next chapter we're going to monitor a multi-service application running on top of our Kubernetes cluster. We'll look at monitoring some specific services, like MySQL and Redis, as well as our application.

Chapter 13

Monitoring a Stack - Tornado

In the last chapter we saw the basics of monitoring Kubernetes, using Prometheus. In this chapter we're going to deploy an application, called Tornado, onto our Kubernetes cluster and monitor it. Tornado is a simple REST-ful HTTP API written in Clojure which runs on the JVM, has a Redis data store, and a MySQL database.

We've deployed each component of the application onto our Kubernetes cluster and will look at how we can monitor each component, collecting information on the component and identifying some key alerts. We'll monitor:

- MySQL,
- Redis, and
- the Tornado API application.

We're going to start with monitoring our two data stores. We're going to use a pattern called sidecar, which we referenced in Chapter 9. Let's take a quick look at that pattern.

Sidecar pattern

To perform much of our monitoring, we'll rely heavily on an architecture pattern called sidecar. The pattern is named sidecar because it resembles a sidecar attached to a motorcycle: the motorcycle is our application, and the sidecar is attached to this parent application. The sidecar provides supporting features for the application—for example, an infrastructure sidecar might collect logs or conduct monitoring. The sidecar also shares the same life cycle as the parent application, being created and deleted alongside the parent.



TIP Sidecars are sometimes called sidekicks.

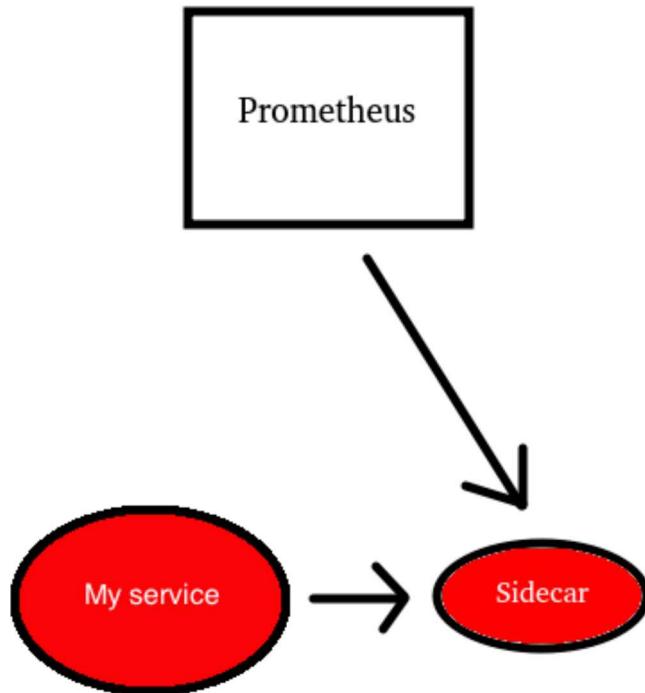


Figure 13.1: The sidecar

In our case, the sidebars run Prometheus exporters. The exporters query our applications and in turn are queried by Prometheus. This sidecar model works on more than just Kubernetes, too; anywhere you're deploying containers or services in clusters lends itself to this pattern.

We'll run sidecar-monitoring exporters next to our Redis and MySQL installations, starting with our MySQL database.

MySQL

Monitoring MySQL with Prometheus is done using an exporter: the MySQLd Exporter. The exporter works by connecting to a MySQL server using provided credentials and querying the state of the server. The queried data is then exposed and can be scraped by the Prometheus server. This means that the exporter needs to have both network access to the MySQL server as well as credentials for authentication. In our case, we're going to run the exporter inside a Docker container deployed to Kubernetes in our sidecar pattern.

Here's the segment of our MySQL Kubernetes deployment which runs the exporter in our sidecar container.

Listing 13.1: The exporter container

```
- image: prom/mysqld-exporter:latest
  name: tornado-db-exp
  args:
    - --collect.info_schema.innodb_metrics
    - --collect.info_schema.userstats
    - --collect.perf_schema.eventsstatements
    - --collect.perf_schema.indexiowaits
    - --collect.perf_schema.tableiowait
  env:
    - name: DATA_SOURCE_NAME
      value: "tornado-db-exp:anotherstrongpassword@(tornado-db
:3306)/"
  ports:
    - containerPort: 9104
      name: tornado-db-exp
```

You can see we're using the `prom/mysqld-exporter` image with the `latest` tag. We've called the container `tornado-db-exp`. We've specified our connection details using the `DATA_SOURCE_NAME` environment variable. This connection uses the DSN format to configure connection and credential details for our MySQL server.

The exporter running inside the container will automatically pick up the connection details from the environmental variable.

You should create a separate user with a limited set of permissions. To query the required data from the MySQL server, you'll need to grant your user the PROCESS, REPLICATION CLIENT, and SELECT permissions.

You can connect to the MySQL container using kubectl's exec command, like so:

Listing 13.2: Connecting to the MySQL container

```
$ kubectl exec -ti <pod> -- /usr/bin/mysql -p
```

We can then run CREATE USER and GRANT statements to assign the required permissions.

Listing 13.3: Creating a MySQL user

```
CREATE USER 'tornado-db-exp'@'localhost' IDENTIFIED BY '  
anotherstrongpassword';  
GRANT PROCESS, REPLICATION CLIENT, SELECT ON *.* TO 'tornado-db-  
exp';  
GRANT SELECT ON performance_schema.* TO 'tornado-db-exp';
```

Here we've created a user called `tornado-db-exp` with the required permissions including a SELECT grant to the `performance_schema` table containing query performance data.

 **TIP** If you have a `my.cnf` file then the exporter can also use credentials hardcoded in there.

We could also configure the exporter using a variety of flags to control its behavior. We've enabled some additional collectors:

Listing 13.4: Additional MySQL exporter collector

```
--collect.info_schema.innodb_metrics  
--collect.info_schema.userstats  
--collect.perf_schema.eventsstatements  
--collect.perf_schema.indexiowaits  
--collect.perf_schema.tableiowaits
```

These all collect data from MySQL's performance schema database, allowing us to track the performance of specific queries and operations.

In our container deployment, we've also exposed port 9104, the default port of the MySQL Exporter, which in turn we've exposed in a service.

Listing 13.5: The tornado-db service

```
apiVersion: v1  
kind: Service  
metadata:  
  name: tornado-db  
  annotations:  
    prometheus.io/scrape: 'true'  
    prometheus.io/port: '9104'  
spec:  
  selector:  
    app: tornado-db  
  type: ClusterIP  
  ports:  
    - port: 3306  
      name: tornado-db  
    - port: 9104  
      name: tornado-db-exp
```

We've used two annotations: `prometheus.io/scrape`, which tells Prometheus to scrape this service, and `prometheus.io/port`, which tells Prometheus which port to scrape. We specify this because we want Prometheus to hit the MySQL Exporter port at 9104 rather than the MySQL server directly. These annotations are automatically picked up by the `kubernetes-service-endpoints` job we created in Chapter 12, and parsed by the relabelling configuration in that job, which we can see below:

Listing 13.6: The Kubernetes endpoint job relabelling

```
relabel_configs:
  - source_labels: [
      __meta_kubernetes_service_annotation_prometheus_io_scrape]
    action: keep
    regex: true
  .
  .
  - source_labels: [__address__,
      __meta_kubernetes_service_annotation_prometheus_io_port]
    action: replace
    target_label: __address__
    regex: ([^:]+)(?:\d+)?;(\d+)
    replacement: $1:$2
  .
  .
```

The `prometheus.io/scrape` annotation ensures Prometheus will only keep metrics from service endpoints with the annotation set to `true`. The `prometheus.io/port` annotation will be placed into the `__address__` label to be scraped by the job. The next service discovery will start collection of the MySQL metrics.

MySQL Prometheus configuration

As our exporter is being exposed as a service endpoint, we don't need to configure a specific job to scrape it. We will, however, create some rules for our MySQL time series and add them to our rules ConfigMap. We're just going to create a

sampling of possible rules, loosely aligned with Google's Four Golden Signals, to give you an idea of how you might use your MySQL metrics. We'll focus on:

- Latency
- Traffic
- Errors
- Saturation

⚠️ WARNING Measuring MySQL performance is hard, especially when tracking signals like latency, and circumstances vary greatly depending on your application and server configuration. These rules give you starting point, not a definitive answer. There are a number of guides online that might prove useful.

First, let's look at some rules, starting with tracking the growth rate of slow queries using the `mysql_global_status_slow_queries` metric. This counter is incremented when a query exceeds the `long_query_time` variable, which defaults to 10 seconds.

Listing 13.7: MySQL slow query alert

```
- alert: MySQLHighSlowQueryRate
  expr: rate(mysql_global_status_slow_queries[2m]) > 5
  labels:
    severity: warning
  annotations:
    summary: MySQL Slow query rate is exceeded on {{ $labels.
  instance }} for {{ $labels.kubernetes_name }}
```

This will generate an alert if the rate over two minutes exceeds five. We can also create recording rules to track the request rates on our server.

Listing 13.8: MySQL request rate records

```

- record: mysql:write_requests:rate2m
  expr: sum(rate(mysql_global_status_commands_total{command=~"insert|update|delete"}[2m])) without (command)
- record: mysql:select_requests:rate2m
  expr: sum(rate(mysql_global_status_commands_total{command=~"select"}[2m]))
- record: mysql:total_requests:rate2m
  expr: rate(mysql_global_status_commands_total[2m])
- record: mysql:top5_statements:rate5m
  expr: topk(5, sum by (schema,digest_text) (rate(mysql_perf_schema_events_statements_total[5m])))

```

We use the `mysql_global_status_commands_total` metric and grab all the write requests for specific commands: insert, update, and delete. We then calculate a rate over two minutes for these requests. We do the same for read requests using the select command, and for total requests. We're also using the topk aggregation operator to get the most frequently used statements by schema and rate over the last five minutes, which helps us understand what the server is doing.



Figure 13.2: The topk operator over MySQL statements

We could graph or alert on these as needed. We can also track connection requests and errors.

Listing 13.9: Connections and aborted connections

```
- alert: MySQLAbortedConnectionsHigh
  expr: rate(mysql_global_status_aborted_connects[2m]) > 5
  labels:
    severity: warning
  annotations:
    summary: MySQL Aborted connection rate is exceeded on {{ $labels.instance }} for {{ $labels.kubernetes_name }}
- record: mysql:connection:rate2m
  expr: rate(mysql_global_status_connections[2m])
```

Here we're alerting if the rate of aborted connections exceeds a threshold, and creating a recording rule to track the rate of connections overall.

Last, we want to know when our MySQL service is unavailable. These alerts use the combination of the state and presence of the exporter-specific `up` metric: `mysql_up`. The `mysql_up` metric does a `SELECT 1` on the MySQL server and is set to 1 if the query succeeds. The first alert checks if the value of the `mysql_up` metric is 0, indicating the query has failed. The second alert checks for the presence of this metric in the event the service disappears and the metric is expired.

Listing 13.10: MySQL alerts

```
- alert: TornadoDBServerDown
  expr: mysql_up{kubernetes_name="tornado-db"} == 0
  for: 10m
  labels:
    severity: critical
  annotations:
    summary: MySQL Server {{ $labels.instance }} is down!
- alert: TornadoDBServerGone
  expr: absent(mysql_up{kubernetes_name="tornado-db"})
  for: 10m
  labels:
    severity: critical
  annotations:
    summary: No Tornado DB servers are reporting!
    description: Werner Heisenberg says - there is no
      uncertainty about the Tornado MySQL server being gone.
```

These are some useful starter rules. You can find these rules in the code for this chapter.

Redis

Like MySQL, Prometheus has an exporter for Redis. The Redis Exporter will export most of the items from the INFO command with details of server, client, memory, and CPU usage. There are also metrics for total keys, expiring keys, and the average TTL for keys in each database. You can also export values of keys.

And, again like MySQL, we can run the exporter as a sidecar of the Redis container. Here's a snippet of our Redis Kubernetes deployment.

Listing 13.11: Redis service and sidecar

```
apiVersion: apps/v1beta2
kind: Deployment
  ...
  - name: redis-exporter
    image: oliver006/redis_exporter:latest
    env:
      - name: REDIS_ADDR
        value: redis://tornado-redis:6379
      - name: REDIS_PASSWORD
        value: tornadoapi
    ports:
      - containerPort: 9121
```

We're running a container called `redis-exporter` from a Docker image, `oliver006/redis_exporter`. We've specified two environments variables: `REDIS_ADDR`, which specifies the address of the Redis server we want to scrape, and `REDIS_PASSWORD`, which specifies a password to connect to the server with. We also specify port 9121 to export our metrics on.

 **TIP** There are other environment variables and command line flags you can set, which you can read about in the documentation.

We then expose this port via a Kubernetes service.

Listing 13.12: The Redis Kubernetes service

```
apiVersion: v1
kind: Service
metadata:
  name: tornado-redis
  annotations:
    prometheus.io/scrape: 'true'
    prometheus.io/port: '9121'
spec:
  selector:
    app: tornado-redis
  ports:
    - port: 6379
      name: redis
    - port: 9121
      name: redis-exporter
  clusterIP: None
```

You can see that we've exposed port 9121, and specified two annotations—one to tell our Prometheus service endpoint job to scrape this service, and one which port to scrape. The next time Prometheus does service discovery it will detect the updated service and start collecting our Redis metrics.

Redis Prometheus configuration

As our exporter is being exposed as a service endpoint, we don't need to configure a specific job to scrape it. We will, however, create some rules for our Redis time series and add them. We're again going to show you a sampling of rules, for example:

Listing 13.13: Redis alerts

```
- alert: TornadoRedisCacheMissesHigh
  expr: redis_keyspace_hits_total / (redis_keyspace_hits_total +
    redis_keyspace_misses_total) > 0.8
  for: 10m
  labels:
    severity: warning
  annotations:
    summary: Redis Server {{ $labels.instance }} Cache Misses
    are high.
- alert: RedisRejectedConnectionsHigh
  expr: avg(redis_rejected_connections_total) by (addr) < 10
  for: 10m
  labels:
    severity: warning
  annotations:
    summary: "Redis instance {{ $labels.addr }} may be hitting
    maxclient limit."
    description: "The Redis instance at {{ $labels.addr }} had {{
      $value }} rejected connections during the last 10m and may be
      hitting the maxclient limit."
```

Here we're measuring if cache misses exceed 0.8 and if the rejected connections average is high.

Last, like our MySQL service, we want to know when our Redis service is unavailable. These alerts use the combination of the state and presence of the exporter-specific `up` metric, `redis_up`. The `redis_up` metric is set to 1 if the scrape of the Redis server succeeds. The first alert checks if the value of the `redis_up` metric is 0, indicating the scrape has failed. The second alert checks for the presence of this metric in the event the service disappears and the metric is expired.

Listing 13.14: Redis availability alerts

```
- alert: TornadoRedisServerDown
  expr: redis_up{kubernetes_name="tornado-redis"} == 0
  for: 10m
  labels:
    severity: critical
  annotations:
    summary: Redis Server {{ $labels.instance }} is down!
- alert: TornadoRedisServerGone
  expr: absent(redis_up{kubernetes_name="tornado-redis"})
  for: 10m
  labels:
    severity: critical
  annotations:
    summary: No Tornado Redis servers are reporting!
    description: Werner Heisenberg says - there is no
      uncertainty about the Tornado Redis server being gone.
```

Now that we've added some monitoring to our MySQL and Redis services, we want to monitor our API service.

Tornado

The Tornado API is a Clojure application that uses Ring and runs on the JVM. It has a single API endpoint that buys and sells items. We're going to instrument the application in much the same way we saw in Chapter 8 to create metrics that monitor each API action.

Adding the Clojure wrapper

To instrument our application we're using the iapetos Clojure wrapper. There are several Clojure wrappers and clients for Prometheus; we chose iapetos because

its up to date and easy to use. To enable the iapetos wrapper we need to add it to the project's dependencies in the `project.clj` file.

Listing 13.15: Adding the client to the `project.clj`

```
(defproject tornado-api-prometheus "0.1.0-SNAPSHOT"
  :description "Example Clojure REST service for "
  :url "http://artofmonitoring.com"
  :dependencies [[org.clojure/clojure "1.8.0"]
    .
    .
    [iapetos "0.1.8"]
    [io.prometheus/simpleclient_hotspot "0.4.0"]
  ]]
  :plugins [[lein-ring "0.7.3"]]
  .
  .
```

Here we've added the `iapetos` and the `Prometheus simpleclient_hotspot` (which we need for exporting some JVM metrics) dependencies.

We can then require the relevant components of the wrapper in our application's source code.

Listing 13.16: Requiring the wrapper components

```
(:require [compojure.handler :as handler]
  .
  .
  [iapetos.core :as prometheus]
  [iapetos.collector.ring :as ring]
  [iapetos.collector.jvm :as jvm]
  .
  .
```

We've included the base `iapetos` wrapper as `prometheus` and two context-specific components for exporting Ring and JVM metrics respectively.

Adding a registry

Like our Ruby application in Chapter 8, we need to define a registry to hold all of our metrics. Our application is pretty simple so we're just adding one registry, but you can add more than one or a registry per subsystem, if, for instance, you want the same counter in different subsystems.

Listing 13.17: Defining the registry

```
(defonce registry
  (-> (prometheus/collector-registry)
       (jvm/initialize)
       (ring/initialize)
       (prometheus/register
         (prometheus/counter :tornado/item-get)
         (prometheus/counter :tornado/item-bought)
         (prometheus/counter :tornado/item-sold)
         (prometheus/counter :tornado/update-item)
         (prometheus/gauge   :tornado/up))))
```

We've created a registry called `registry` and we've initialized the Ring and JVM metrics, which will be automatically collected and exported. We've then defined five specific metrics, four of them counters and one gauge, all prefixed with `tornado`. We have one counter for each of the API's actions and a gauge to act as an `up` metric for the application. We can also add labels to the metrics we've defined.

Listing 13.18: Adding labels

```
(prometheus/counter :tornado/item-bought
  {:description "Total items bought"})
```

Here we've added a `description` label to the `item-bought` counter.

Adding metrics

We can now add function calls to each API method on our application to increment our counters. For example, here's the function that increments the metric for buying an item:

Listing 13.19: Adding metric calls

```
(defn buy-item [item]
  (let [id (uuid)]
    (sql/db-do-commands db-config
      (let [item (assoc item "id" id)]
        (sql/insert! db-config :items item)
        (prometheus/inc (registry :tornado/item-bought)))
      (wcar* (car/ping)
        (car/set id (item "title")))
      (get-item id))))
```

We're calling the `inc` function to increment our `item-bought` counter when an item is bought. We could also set gauges or other time series including histograms.

We've also added a gauge called `tornado_up` that will act as the `up` metric for our application. When the application starts it will automatically set the value of the gauge to 1.

Listing 13.20: The `tornado_up` gauge

```
(prometheus/set (registry :tornado/up) 1)
```

Exporting the metrics

Last, we want to enable the `/metrics` page itself, in our case by using the built-in Ring support.

Listing 13.21: Starting the export

```
(def app
  (-> (handler/api app-routes)
        (middleware/wrap-json-body)
        (middleware/wrap-json-response)
        (ring/wrap-metrics registry {:path "/metrics"})))
```

This will make the metrics we've defined, some JVM-centric metrics and some HTTP-specific metrics emitted from Ring, available on the application at the `/metrics` path.

If we now browse to this path we'll see our metrics emitted. Here's a quick sample.

Listing 13.22: Tornado metrics

```
# HELP http_request_latency_seconds the response latency for
HTTP requests.
# TYPE http_request_latency_seconds histogram
http_request_latency_seconds_bucket{method="GET", status="404",
statusClass="4XX", path="index.php", le="0.001",} 2.0
.
.
# HELP tornado_item_sold a counter metric.
# TYPE tornado_item_sold counter
tornado_item_sold 0.0
.
.
# HELP jvm_threads_peak Peak thread count of a JVM
# TYPE jvm_threads_peak gauge
jvm_threads_peak 14.0
```

Tornado Prometheus configuration

Like our other services, our Clojure exporter is being exposed as an endpoint, and we don't need to configure a specific job to scrape it. We get a wide variety of metrics—metrics from the JVM, HTTP metrics from Ring, and metrics from the application itself. We can now create some alerts and rules to monitor our API.

Here's a latency recording rule we created using one of the Ring HTTP metrics.

Listing 13.23: Ring latency rule

```
- record: tornado:request_latency_seconds:avg
  expr: http_request_latency_seconds_sum{status="200"} /
    http_request_latency_seconds_count{status="200"}
```

We've created a new metric, `tornado:request_latency_seconds:avg`, The average request latency in seconds for requests which result in a 200 HTTP code.

We can also take advantage of one of the Ring-related histograms to alert on high latency.

Listing 13.24: Ring high latency alert

```
- alert: TornadoRequestLatencyHigh
  expr: histogram_quantile(0.9, rate(
    http_request_latency_seconds_bucket{ kubernetes_name="tornado-
    api" [5m]})) > 0.05
  for: 10m
  labels:
    severity: warning
  annotations:
    summary: API Server {{ $labels.instance }} latency is over
    0.05.
```

Here we've used the `histogram_quantile` function to generate the 90th percentile

of our request latency over 5 minutes. Our alert will be triggered if that exceeds 0.05 for 10 minutes.

We can also take advantage of the up-style metric we created, `tornado_up`, to watch for the availability of our API service.

Listing 13.25: Monitoring the Tornado API availability

```
- alert: TornadoAPIServerDown
  expr: tornado_up{kubernetes_name="tornado-api"} != 1
  for: 10m
  labels:
    severity: critical
  annotations:
    summary: API Server {{ $labels.instance }} is down!
- alert: TornadoAPIServerGone
  expr: absent(tornado_up{kubernetes_name="tornado-api"})
  for: 10m
  labels:
    severity: critical
  annotations:
    summary: No Tornado API servers are reporting!
    description: Werner Heisenberg says - there is no
      uncertainty about the Tornado API server being gone.
```

Here we'll detect if the `tornado_up` metric has a value other than 0 or if it disappeared from our metrics.

This gives you a simple overview of how you might apply what you've learned in the book to monitoring an application stack.

Summary

In this chapter we've seen how we'd monitor services and applications running on top of Kubernetes. We used the sidecar pattern to do this, parallel monitoring

running next to our services and application, inside the same deployment.

We also saw another application instrumented, this time a Clojure-based application using the iapetos wrapper.

You can easily build upon this basis to monitor more complex applications and services using this simple building-block pattern.

List of Figures

1 License	5
2 ISBN	5
1.1 Service hierarchy	9
1.2 Monitoring design	11
1.3 A sample plot	20
1.4 A sample gauge	21
1.5 A sample counter	22
1.6 A histogram example	23
1.7 An aggregated collection of metrics	25
1.8 The flaw of averages - copyright Jeff Danzinger	27
1.9 Response time average	28
1.10 Response time average Mk II	29
1.11 Response time average and median	30
1.12 Response time average and median Mk II	31
1.13 The empirical rule	32
1.14 Response time average, median, and percentiles	34
1.15 Response time average, median, and percentiles Mk II	35
2.1 Prometheus architecture	49
2.2 Prometheus expression browser	52
2.3 Redundant Prometheus architecture	53
3.1 Prometheus expression browser	76
3.2 List of metrics	77

3.3 Querying quantiles	78
3.4 Querying total HTTP requests	80
3.5 Calculating total HTTP requests by job	81
3.6 Our rate query	83
4.1 cAdvisor web interface	99
4.2 cAdvisor Prometheus metrics	100
4.3 Scrape lifecycle	104
4.4 Sample label taxonomy	106
4.5 Labels prior to relabelling	112
4.6 node_cpu_seconds_total metrics	114
4.7 Per-host average percentage CPU usage metrics	116
4.8 Per-host percentage CPU plot	117
4.9 Number of CPUs in each host	118
4.10 The node_memory_MemTotal_bytes	119
4.11 Per-host percentage memory usage	120
4.12 Disk metrics	121
4.13 Per-host disk space metrics	122
4.14 The systemd time series data	125
4.15 The active services	126
4.16 The up metrics	127
4.17 The cadvisor_version metric	130
4.18 The instance:node_cpu:avg_rate5m recorded rule	137
4.19 The Grafana console login	145
4.20 The Grafana console	146
4.21 Adding a Grafana data source for Prometheus	147
4.22 Adding a Grafana data source for Prometheus	148
4.23 The Node dashboard	149
5.1 Scrape lifecycle	152
6.1 Alertmanager architecture	174

6.2 Alertmanager routing	182
6.3 Alertmanager web interface	184
6.4 List of Prometheus alerts	191
6.5 Fired alert in Alertmanager	194
6.6 HighNodeCPU alert email	195
6.7 Scheduling silences	210
6.8 A new silence	211
6.9 Editing or expiring silences	212
7.1 Fault-tolerant architecture	219
7.2 Alertmanager cluster status	223
7.3 Prometheus clustered Alertmanagers	226
7.4 Organizational sharding	227
7.5 Functional sharding	228
7.6 Horizontal sharding	229
7.7 The Federate API	236
8.1 Rails server targets	254
8.2 Rails metrics	254
9.1 mtail diagnostics	262
10.1 Probing architecture	275
10.2 The blackbox exporter console	284
10.3 The probe metrics in Prometheus	288
11.1 The Pushgateway	292
11.2 The Pushgateway dashboard	298
11.3 The test_counter in the Pushgateway dashboard	307
11.4 The test_counter metric	309
12.1 The Kubernetes endpoint targets	325
12.2 The Kube-state-metrics endpoint target	329

List of Figures

13.1 The sidecar	341
13.2 The topk operator over MySQL statements	347

Listings

1 Sample code block	3
1.1 Sample Nagios notification	40
2.1 Time series notation	56
2.2 Example time series	56
3.1 Download the Prometheus tarball	62
3.2 Unpack the prometheus binary	62
3.3 Checking the Prometheus version on Linux	63
3.4 Creating a directory on Windows	63
3.5 Prometheus Windows download	63
3.6 Setting the Windows path	64
3.7 Checking the Prometheus version on Windows	64
3.8 Installing Prometheus via Chocolatey	65
3.9 Installing Prometheus via Homebrew	65
3.10 Checking the Prometheus version on Mac OS X	65
3.11 The default Prometheus configuration file	68
3.12 Alertmanager configuration	70
3.13 The default Prometheus scrape configuration	72
3.14 Moving the configuration file	73
3.15 Starting the Prometheus server	73
3.16 Validating your configuration with promtool	73
3.17 Running Prometheus with Docker	74
3.18 Mounting a configuration file into the Docker container	74

3.19 Some sample raw metrics	75
3.20 A raw metric	75
3.21 Go garbage collection 50th percentile	77
3.22 The prometheus_build_info metric	79
4.1 Downloading the Node Exporter	89
4.2 Testing the Node Exporter binary	90
4.3 Running the help for Node Exporter	90
4.4 Controlling the port and path	90
4.5 Disabling the arp collector	91
4.6 Creating a textfile directory	91
4.7 A metadata metric	92
4.8 Starting Node Exporter with the textfile collector and systemd	93
4.9 The current Prometheus scrape configuration	94
4.10 Adding the node job	95
4.11 Filtering collectors	96
4.12 Testing collect params	96
4.13 Running the caAdvisor container	98
4.14 The cAdvisor container	99
4.15 Adding the Docker job	101
4.16 Overriding the discovered labels	103
4.17 Dropping metrics with relabelling	108
4.18 Specifying a new separator	109
4.19 Replacing a label	110
4.20 Dropping a label	111
4.21 Memory saturation query	121
4.22 The node_systemd_unit_state metrics	124
4.23 The up metric	126
4.24 A one-to-one vector match	129
4.25 The evaluation_interval parameter	132
4.26 Creating a recorded rules file	133
4.27 Adding the rules file	133

4.28 A recording rule	134
4.29 A recording group interval	135
4.30 A recording rule	136
4.31 A recording rule	136
4.32 Getting the PackageCloud public key on Ubuntu	139
4.33 Adding the Grafana packages	139
4.34 Updating Apt and installing the Grafana package	139
4.35 Getting the Grafana public key on Red Hat	140
4.36 The Grafana Yum configuration	140
4.37 Installing Grafana on Red Hat	140
4.38 Creating a Grafana directory on Windows	141
4.39 Grafana Windows download	141
4.40 Setting the Windows path for Grafana	141
4.41 Installing Grafana via Homebrew	142
4.42 Starting the Grafana Server on Linux	143
4.43 Starting Grafana at boot on OSX	143
4.44 Starting Grafana server on OS X	144
5.1 Our static service discovery	153
5.2 File-based discovery	154
5.3 Creating the target directory structure	155
5.4 Creating JSON files to hold our targets	155
5.5 The nodes.json file	156
5.6 The daemons.json file	156
5.7 The daemons file in YAML	156
5.8 Adding labels	157
5.9 An EC2 discovery job	160
5.10 An EC2 discovery job with a profile	161
5.11 An EC2 discovery job with a port	161
5.12 Relabelling an EC2 discovery job	162
5.13 Relabelling the instance name in a EC2 discovery job	164
5.14 DNS service discovery job	165

5.15 A SRV record	166
5.16 Example SRV records	166
5.17 The DNS targets from the SRV	167
5.18 DNS A record service discovery job	167
5.19 DNS subdomain A record service discovery job	168
6.1 Stock Nagios alert	172
6.2 Download the Alertmanager tarball	176
6.3 Unpack the alertmanager binary	176
6.4 Moving the amtool binary	176
6.5 Checking the Alertmanager version on Linux	177
6.6 Creating a directory on Windows	177
6.7 Alertmanager Windows download	178
6.8 Setting the Windows path	178
6.9 Checking the Alertmanager version on Windows	178
6.10 Creating the alertmanager.yml file	179
6.11 A simple alertmanager.yml configuration file	180
6.12 Creating the templates directory	181
6.13 Starting Alertmanager	183
6.14 The alerting block	185
6.15 The Alertmanager SRV record	186
6.16 Discovering the Alertmanager	186
6.17 The Alertmanager Prometheus job	187
6.18 Creating an alerting rules file	188
6.19 Adding globbing rule_files block	188
6.20 Our first alerting rule	189
6.21 The ALERT time series	193
6.22 Adding more alerting rules	196
6.23 Humanizing a value	197
6.24 Creating the prometheus_alerts.yml file	198
6.25 The prometheus_alerts.yml file	198
6.26 Node service alert	199

6.27 The up metric missing alert	201
6.28 Adding routing configuration	202
6.29 Grouping	203
6.30 Label matching	204
6.31 Routing branching	204
6.32 Routing branching	205
6.33 Multiple endpoints in a receiver	205
6.34 A regular expression match	206
6.35 Adding a Slack receiver	207
6.36 Adding a Slack receiver	207
6.37 Creating a template file	208
6.38 The slack tmpl file	208
6.39 Adding a Slack receiver	209
6.40 Using amtool to schedule a silence	213
6.41 Querying the silences	214
6.42 Expiring the silence	214
6.43 Sample amtool configuration file	214
6.44 Using amtool to schedule a silence	215
6.45 Omitting alertname	215
7.1 Starting Alertmanager cluster	221
7.2 Starting Alertmanager cluster remaining nodes	222
7.3 Defining alertmanagers statically	224
7.4 The Alertmanager SRV record	225
7.5 Discovering the Alertmanager	225
7.6 The worker0 configuration	232
7.7 The instance CPU rule	233
7.8 The primary configuration	234
7.9 Worker file discovery	234
7.10 Matching parameter	235
7.11 The match[.	235
8.1 A sample payments method	243

8.2 The mwp-rails Gemfile	245
8.3 Install prometheus-client with the bundle command	246
8.4 Testing the Prometheus client with the Rails console	246
8.5 Creating a Prometheus registry	247
8.6 Registering a Prometheus metric	247
8.7 Incrementing a metric	247
8.8 Incrementing a metric	248
8.9 The basic Prometheus client_ruby metrics	248
8.10 Creating a Metrics module	248
8.11 The Metrics module	249
8.12 Creating an initializer for the metrics library	250
8.13 The config/initializers/lib.rb file	250
8.14 Counter for user deletions	250
8.15 Counter for user creation	251
8.16 Adding Prometheus to the config.ru file	252
8.17 The Rails /metrics endpoint	253
8.18 Our Rails servers service discovery	254
8.19 The rails job	254
9.1 Download and install the mtail binary	258
9.2 Running the mtail binary	258
9.3 Creating an mtail program directory	259
9.4 Creating the line_count.mtail program	259
9.5 The line_count.mtail program	260
9.6 A relational clause	261
9.7 Running mtail	261
9.8 The mtail /metrics path	263
9.9 The apache_combined program	264
9.10 The combined access log actions	266
9.11 Running mtail	267
9.12 Apache combined metrics	267
9.13 The mtail rails program	269

9.14 Rails mtail metric output	270
9.15 The mtail job	272
9.16 Worker file discovery	272
10.1 Download the blackbox exporter zip file	277
10.2 Unpack the blackbox_exporter binary	277
10.3 Checking the blackbox exporter version on Linux	278
10.4 Creating a directory on Windows	278
10.5 Blackbox exporter Windows download	279
10.6 Setting the Windows path	279
10.7 Checking the blackbox exporter version on Windows	279
10.8 The prober.yml file	280
10.9 The /etc/prober/prober.yml file	281
10.10 Valid status codes	282
10.11 Starting the exporter	283
10.12 The http_probes job	285
10.13 The http_probe targets	286
10.14 The http_2xx_check metrics	287
11.1 Download the Pushgateway zip file	294
11.2 Unpack the pushgateway binary	295
11.3 Checking the Pushgateway version on Linux	295
11.4 Creating a directory on Windows	296
11.5 Pushgateway Windows download	296
11.6 Setting the Windows path	296
11.7 Checking the Pushgateway version on Windows	297
11.8 Running the Pushgateway on an interface	298
11.9 Persisting the metrics	299
11.10 Posting a metric to the gateway	299
11.11 The Pushgateway metrics path	299
11.12 Posting a metric to the gateway	300
11.13 Adding labels to pushed metrics	301
11.14 Passing types and descriptions	301

11.15 Passing types and descriptions	301
11.16 Curling the gateway metrics	303
11.17 Deleting Pushgateway metrics	304
11.18 Deleting a selection of Pushgateway metrics	304
11.19 Creating MetricsPush class	305
11.20 The MetricsPush module	306
11.21 Pushing a metric	307
11.22 The pushgateway job	308
11.23 Our Pushgateway	309
12.1 The cluster kops command	312
12.2 The Node Exporter DaemonSet tolerations	315
12.3 The Node Exporter DaemonSet containers	316
12.4 Node Exporter liveness and readiness probes	317
12.5 The Node Exporter service	319
12.6 Deploying the Node Exporter daemonset and service	320
12.7 The default namespace	320
12.8 Checking the Node Exporter pods	321
12.9 A Node Exporter pod's logs	321
12.10 Checking the Node Exporter service	322
12.11 The Kubernetes service endpoints job	323
12.12 Replacing the ConfigMap	325
12.13 The monitoring services	326
12.14 Kubernetes availability alerting rules	327
12.15 Kubernetes availability alerting rules	328
12.16 Kube-state-metrics deployment generation alert	330
12.17 Kube-state-metrics Deployment replicas not updated alert	331
12.18 Kube-state-metrics pod restarting alert	331
12.19 API server job	333
12.20 The API server recording rules	334
12.21 API high latency alert	335
12.22 API high error rate alert	335

12.23 API servers down or absent	336
12.24 The CAdvisor job	337
13.1 The exporter container	342
13.2 Connecting to the MySQL container	343
13.3 Creating a MySQL user	343
13.4 Additional MySQL exporter collector	344
13.5 The tornado-db service	344
13.6 The Kubernetes endpoint job relabelling	345
13.7 MySQL slow query alert	346
13.8 MySQL request rate records	347
13.9 Connections and aborted connections	348
13.10 MySQL alerts	349
13.11 Redis service and sidecar	350
13.12 The Redis Kubernetes service	351
13.13 Redis alerts	352
13.14 Redis availability alerts	353
13.15 Adding the client to the project.clj	354
13.16 Requiring the wrapper components	354
13.17 Defining the registry	355
13.18 Adding labels	355
13.19 Adding metric calls	356
13.20 The tornado_up gauge	356
13.21 Starting the export	357
13.22 Tornado metrics	357
13.23 Ring latency rule	358
13.24 Ring high latency alert	358
13.25 Monitoring the Tornado API availability	359

Index

- \$labels, 197
- \$value, 197
 - _ time series names, 55
 - _address_, 102, 163, 337
 - _meta_ec2_public_ip, 162
 - _metrics_path_, 102, 337
 - _name_, 108
 - _scheme_, 102
- Absent, 200
- action
 - hashmod, 233
 - keep, 233
- Aggregation, 80
- Alert
 - Annotations, 197
 - templates, 197
- Alerting, 70, 170
 - Symptoms versus causes, 172
- Alerting rules, 69, 131, 187
- Alertmanager, 51, 58, 70, 170
 - amtool, 184
 - API, 193
 - Cluster, 218
- configuration, 179
- continue, 204
- default route, 201
- Email, 180
- email
 - emails_configs, 183
- global, 180
- group_by, 202
- group_interval, 202
- group_wait, 202
- grouping, 202
- High availability, 218
- Installation, 175
 - Linux, 175
 - Mac OS X, 177
 - Windows, 177
- Installing via configuration management, 179
- match, 204
- match_re, 206
- Mesh, 218
- Notification template variable reference, 208

- receivers, 182
- Resolved alerts, 206
- routing, 181, 204
- send_resolved, 206
- Silences, 210
- Supported platforms, 175
- templates, 181, 207
- version, 176
- web hooks, 183
- web interface, 184
- Alerts, 39
- amtool, 184, 213, 216
- Annotations, 197
- Ansible, 66, 142
- Apophenia, 41
- Application architecture, 9, 238
- Application metrics, 238
- Application monitoring, 9, 238
- Architecture, 48, 51
- Availability monitoring, 126
- Average, 24
- Averages, 25, 29
- AWS, 159, 161, 164
 - Access Key ID, 160
 - Profile, 161
 - Secret Access Key, 160
- Batch jobs, 291
- Bell Curve, 26
- Binary operators, 128
- Black Exporter
 - Configuration, 280
 - Blackbox Exporter, 274, 276
 - Installation, 276
 - Linux, 277
 - Mac OS X, 278
 - Windows, 278
 - Installing via configuration management, 279
 - Scraping the exporter, 285
 - Supported platforms, 276
 - version, 278
 - Blackbox exporter, 201
 - DNS prober, 283
 - HTTP prober, 282
 - ICMP prober, 282
 - Blackbox monitoring, 15, 274
 - Borg, 47
 - Borgmon, 47
 - Business metrics, 238
 - Buy-v-build, 43
 - cAdvisor, 97
 - Capacity planning, 83
 - Chef, 66, 142, 280, 297
 - Chocolatey, 64
 - Client libraries, 58, 245
 - client_ruby, 245
 - Clustering, 218
 - CNCF, 48
 - collectd, 88, 258
 - Comparison binary operator, 125

Configuration, 67, 73
Configuration Management, 62, 66, 179, 279, 297
Configuration management, 142
container_last_seen, 127
Count, 24
Counters, 21
CPU, 114, 117

Data model, 54
Disabling collectors, 91
Disk, 121
DNS service discovery, 165
dns_sd_configs, 165
Docker, 97, 142, 280

EC2 Service Discovery
 metadata, 161, 164
 Profile, 161
 Role ARN, 160
EC2 Service discovery, 159
ec2_sd_config
 access_key, 160
 port, 161
 profile, 161
 region, 160
 secret_key, 160
ELK, 17, 257
Endpoints, 50, 71
Exporters, 48, 58
 Grok, 257
 Node, 88

Expression browser, 51, 76
Fault tolerance, 217
Federation, 228, 229
File-based service discovery, 154
file_sd_config, 154
 files, 154
 refresh_interval, 155
Frequency distribution, 22

Gauges, 21
global
 evaluation_interval, 69
 scrape_interval, 69
Go, 48
 client, 245
Google's Golden Signals, 36
Grafana, 54
 installation
 OS X, 142
 Windows, 140
Granularity, 13, 20, 69
Graph, 20
Graphite, 258, 263
Grok Exporter, 257

hashmod, 233
High Availability, 217
High availability, 51, 53
Histogram, 22
histogram_quantile, 334
Homebrew, 65, 142

honor_labels, 309
Host monitoring, 88
ICMP, 288
increase, 332
Installation, 61
 Linux, 62, 175, 277, 294
 Mac OS X, 65, 142
 Microsoft Windows, 63, 64, 140, 177, 278
 Windows, 63, 140
Installing mtail, 258
Installing onto Kubernetes, 67, 312
Installing via configuration management, 66
Instance label, 102, 115, 164
Instances, 50, 71
Instrumentation, 58, 238
Instrumentation labels, 55
Instrumenting applications, 305
Introspection monitoring, 15
irate, 115

Java
 client, 245
Job definition, 72
job_name, 72
Jobs, 50, 71, 94, 291
 Service discovery, 151

keep, 233
kops, 312

Kubernetes, 67, 159, 312
 Node Exporter, 314
kubernetes_sd_config, 313

Label
 Instance, 115
 instance, 102, 164
Labels, 54, 55, 105, 128
 address, 102, 163
 _meta_ec2_public_ip, 162
 _meta_filepath, 157
 _metrics_path_, 102
 name, 108
 scheme, 102
 Metadata, 102
Latency, 36
Logging, 17, 256
Logs, 17
Logstash, 257

Maintenance, 209
Mean, 25
Median, 24, 30, 35
Memory, 119
Metric names, 54, 55, 135
metric_relabel_configs, 107
 action, 110
 regex, 109, 111
 replacement, 111
 separator, 108
 source_labels, 108
 target_label, 111

Metrics, 18, 242
 latency, 36
metrics_relabel_configs, 162
modulus, 232
Monitoring, 6
Monitoring anti-patterns, 9
Monitoring CPU, 114, 117
Monitoring disk, 121
Monitoring jobs, 291
Monitoring Kubernetes, 313
Monitoring memory, 119
Monitoring methodologies, 36
 Google's Golden Signals, 36
 USE Method, 36
mtail, 257
 configuration, 259
 constants, 265
 histogram, 268
 installation, 258
 programs, 259
 running, 262
 types, 266
MySQL, 342
NAT, 293
Node Exporter, 88, 314
 disabling collectors, 91
 filtering collectors, 95
 Textfile collector, 92
Node monitoring, 88
Notification templates, 207
Notifications, 39
Observability, 15
Observations, 19
Percentiles, 24, 33, 35
Plot, 20
predict_linear, 123
Probe
 DNS, 288
 ICMP, 288
Probing, 274
 Architecture, 275
Probing monitoring, 15
Promeditor, 83
Prometheus, 6
 configuration, 67
 disk usage, 83
 duplicate servers, 220
 fault-tolerance, 220
 installation
 Linux, 62
 OS X, 65
 Windows, 63
 memory usage, 83
 Web interface, 76
prometheus
 --config.file, 73
 --version, 62
Prometheus server, 58
prometheus.yml, 67
PromQL, 51, 78, 80

Binary operators, 128
by, 80
count, 117
irate, 82, 115
predict_linear, 123
Range vectors, 82
rate, 81
regular expressions, 122
Scalar, 82
String, 82
Vector matches, 129
without, 80
promtool, 62, 67, 73, 132, 137
Pull-based monitoring, 17
Puppet, 66, 142, 280, 297
Push Gateway, 48
Push-based monitoring, 17
Pushgateway, 291
 Aggregation, 304
 clients, 305
 Configuration, 299
 Delete metrics, 304
 Installation, 293
 Linux, 294
 Mac OS X, 295
 Windows, 295
Installing via configuration management, 297
push_time_seconds, 304
Scaling, 293
Scraping the gateway, 308
 Sending metrics, 299
 Supported platforms, 293
 version, 295
 Viewing metrics, 302
PushProx, 293
Python
 client, 245
Quantile, 33
Querying labels, 78
Rails, 244
 metrics, 244
Prometheus, 244
Range vectors, 82
Rates of change, 24
RE2, 109
Receivers, 207
Recording rules, 69, 131, 132
 sequencing, 134
Redis, 349
Regex, 109
regex, 163
RegExp, 109
relabel_configs, 107, 161, 162
Relabelling, 107, 128, 162
 action
 drop, 110
 keep, 324
 labeldrop, 112
 labelmap, 325, 337
 replace, 324

honor_labels, 111
ordering, 110
Remote storage, 236
remote_read, 237
remote_write, 237
Resolution, 13, 20, 69
Ruby
 client, 245
Rule files, 71
rule_files, 71, 133
Rules, 69
 co-mingle, 187
SaltStack, 66, 280
Samples, 19
Scaling, 217
Scrape configuration, 50, 71
Scrape interval, 69
Scrape lifecycle, 101, 152
scrape_configs, 50, 71
Server, 58
Service discovery, 94, 151–153
 DNS, 165
 EC2, 159
 File-based, 154
 multiple configurations, 153, 162
Service records, 166
Sharding, 228
Sidecar, 271
Sidecar pattern, 340
Silences, 209, 210
 expiration, 213
SoundCloud, 48
source_labels, 162
SRV records, 166, 167
SSD, 52, 85
Standard Deviation, 24, 33
static_configs, 94
StatsD, 258, 263
Sum, 24
Summary, 23
Supported platforms, 61
Tags, 55
Target labels, 55
target_label, 163
Targets, 50, 71
Templates, 197
Text exposition format, 91
Textfile collector, 92
Thanos, 237
Thresholds, 12
Time series, 19
topk, 347
Unsee, 212
Up metric, 126
USE Method, 36, 113
Utility model, 242
Vector matches, 129
Visualization, 41, 131
Whitebox monitoring, 15

YAML, 67

YAML validation, 67

Thanks! I hope you enjoyed the book.

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

