

JAMES TURNBULL

THE
ART
OF
MONITORING



The Art of Monitoring

James Turnbull

September 16, 2016

Version: v1.0.3 (2c4a7d0)

Website: [The Art of Monitoring](#)



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 2016 - James Turnbull <james@lovedthanlost.net>

ISBN 9780988820241

A standard linear barcode representing the ISBN 9780988820241. The number 9 780988 820241 is printed below the barcode.

90000 >

A standard linear barcode representing the number 90000. The number 90000 is printed below the barcode.

Contents

	Page
Foreword	1
Who is this book for?	1
Credits and Acknowledgments	1
Technical Reviewers	2
Caitie McCaffrey	2
Paul Stack	2
Jamie Wilkinson	2
Editor	3
Author	3
Conventions in the book	3
Code and Examples	4
Colophon	4
Errata	4
Disclaimer	4
Copyright	5
Version	6
Chapter 1 Introduction	7
Welcome to the Art of Monitoring	8
What is monitoring?	9
The business as a customer	9
Information Technology as a customer	10
What does monitoring actually look like?	10

Manual, user-initiated, or no monitoring	11
Reactive	11
Proactive	12
Model distribution	13
Becoming Proactive	15
What's in the book?	18
Tool choices	19
Chapter 2 A Monitoring and Measurement Framework	21
Blackbox versus Whitebox, or Pull versus Push	23
Event, log, and metric-centered	26
More about metrics	26
So what's a metric?	27
Types of metrics	29
Metric summaries	31
Metric aggregation	32
Contextual and useful notifications	33
Visualization	36
So why this architecture? What's wrong with traditional monitoring?	37
Static configuration	38
Inflexible logic and thresholds	38
Object-centric	39
An interlude into pets and cattle	40
So what do we do differently?	41
Smarter threshold inputs	42
Collecting data for our monitoring framework	53
Overhead and the observer effect	54
Summary	54
Chapter 3 Managing events and metrics with Riemann	55
Introducing Riemann	56
Riemann architecture and implementation	57

Installing Riemann	58
Configuring Riemann	67
Learning some Clojure	67
Riemann's base configuration	68
Events, streams, and the index	73
Configuring events, streams, and the index	77
Sending our first event to Riemann	82
Creating our first Riemann monitoring check	85
An interlude into Riemann filtering	87
Connecting Riemann servers	90
Configuring the upstream Riemann servers	92
Configuring the downstream Riemann server	97
Enabling the send of our Riemann events downstream	97
Alerting on the upstream Riemann servers	100
Throttling Riemann events	109
Rolling up Riemann events	110
Alternatives to email notifications	111
Testing your Riemann configuration	111
Validating Riemann configuration	116
Performance, scaling, and making Riemann highly available	117
Alternatives to Riemann	120
Summary	121
Chapter 4 Introducing Graphite and Grafana	122
Introducing Graphite	123
Carbon	124
Whisper	124
Graphite Web, Graphite-API, and Grafana	125
Graphite architecture	126
Installing Graphite	126
Installing Graphite on Ubuntu	129

Installing Graphite on Red Hat	130
Installing Graphite-API	133
Installing Grafana	136
Installing Graphite and Grafana via configuration management	139
Configuring Graphite and Carbon	140
Configuring Carbon's metric retention	148
Estimating Graphite storage	153
Carbon and Graphite service management	155
Configuring Graphite-API	162
Service management for Graphite-API	165
Testing the Graphite-API	167
Configuring Grafana	168
Configuring Riemann for Graphite	174
A brief introduction to Grafana	183
Graphite and Carbon Redundancy	191
Time and time zones	196
Managing time manually	197
Managing Time via configuration management	202
Checking the time status	203
Alternatives to Graphite and Grafana	204
Commercial tools	204
Open-source tools	204
Whisper alternatives	205
InfluxDB	206
Cyanite	206
Summary	206
Chapter 5 Host monitoring	208
Introducing collectd	210
What host components should we monitor?	213
Installing collectd	214

Installing collectd on Ubuntu	215
Installing collectd on Red Hat	216
Installing collectd via configuration management	217
Configuring collectd	218
Loading and configuring collectd plugins for monitoring	224
Finishing up	247
Enabling and running collectd	247
The collectd events	248
Sending our collectd events to Graphite	252
Refactoring the collectd metric names	253
Summary	261
Chapter 6 Using collectd events in Riemann	262
Checking processes are running	263
Other actions and enhancements	270
Replicating some classic monitoring	271
Better monitoring through smarter data	274
Building a median-based check	274
Using percentiles for host-based checks	276
Creating check abstractions	279
Organizing our checks	286
Graphing collectd metrics with Grafana	287
Creating the Hosts dashboard	288
Creating our first host graph	289
Creating a memory graph	294
Single host graphs	296
Additional graphs	297
Network, device, and Microsoft Windows monitoring	299
Alternatives to collectd	299
Commercial tools	299
Open source	300

Summary	301
Chapter 7 Containers: another kind of host	302
Challenges with container monitoring	303
Monitoring Docker containers	307
Docker collectd plugin	310
Installing the Docker collectd plugin	311
Configuring the Docker collectd plugin	313
Processing Docker collectd statistics with Riemann	316
Adding metadata to our Docker events	324
Specifying different resolution for Docker metrics	334
Cleaning up old Graphite Docker metrics	337
Using Docker metrics for monitoring	338
Other container monitoring tools	340
Summary	341
Chapter 8 Logs and logging	342
Introducing Elasticsearch, Logstash, and Kibana	343
Logstash architecture	345
Installing Logstash	347
On Debian & Ubuntu	347
On Red Hat	347
Testing Java is installed	348
Installing the Logstash package	348
Testing Logstash is installed	351
Configuring Logstash	351
Installing Elasticsearch	355
On Debian and Ubuntu	356
On Red Hat	357
Installing Elasticsearch via configuration management	359
Testing Elasticsearch is installed	360
Determining Elasticsearch is running	360

Configuring our Elasticsearch cluster and nodes	362
Adding a cluster management plugin	365
Time and time zone	367
Integrating Logstash and Elasticsearch	368
What happens inside Logstash?	372
What happens inside Elasticsearch?	382
Installing Kibana	386
Configuring Kibana	387
Running Kibana	389
Using Kibana	391
Connecting our hosts to Logstash via Syslog	391
Configuring Logstash	391
A quick introduction to Syslog	394
Configuring Syslog	395
Logging from Docker	400
Configuring the Docker Daemon for logging	401
Sending data from Logstash to Riemann	410
Sending data from Riemann to Logstash	415
Scaling Elasticsearch and Logstash	419
Scaling Logstash	419
Scaling Elasticsearch	423
Monitoring our components	423
Monitoring RSyslog	423
Monitoring Logstash	424
Monitoring Elasticsearch	433
Alternatives to Logstash	438
Splunk	438
Heka	438
Graylog	438
mtail	438
Summary	439

Chapter 9 Building Monitored Applications	440
An application monitoring primer	442
Where should I instrument?	443
Instrument schemas	443
Time and the observer effect	444
Metrics	445
Application metrics	445
Business metrics	446
Monitoring patterns, or where to put your metrics	446
The utility pattern	447
The external pattern	449
Building metrics into a sample application	449
Logging	476
Adding our own structured log entries	477
Adding structured logging to our sample application	480
Working with your existing logs	489
Health checks, endpoints, and external monitoring	494
Checking an internal endpoint	497
Deployments	501
Adding deployment notifications to our sample application	502
Working with our deployment events	505
Tracing	511
Summary	512
Chapter 10 Notifications	513
Our current notifications	515
Updating expired event configuration	515
Upgrading our email notifications	516
Formatting the email subject	519
Formatting the email body	520
Adding graphs to notifications	528

Defining our data source	529
Defining our query parameters	531
Defining our graph panels and rows	535
Rendering the dashboard	538
Adding our dashboard to the Riemann notification	539
Some sample scripted dashboards	542
Other context	542
Adding Slack as a destination	543
Adding PagerDuty as a destination	549
Maintenance and downtime	555
Learning from your notifications	561
Other alerting tools	567
Summary	567
Chapter 11 Monitoring Tornado: a capstone	569
The Tornado application	571
Application architecture	572
Monitoring strategy	574
Tagging our Tornado events	575
Monitoring Tornado — Web tier	577
Monitoring HAProxy	577
Monitoring Nginx	596
Addressing the Web tier monitoring concerns	611
Setting up the Tornado checks in Riemann	614
The webtier function	620
Adding Tornado checks to Riemann	629
Summary	631
Chapter 12 Monitoring Tornado: Application Tier	632
Monitoring the Application tier JVM	633
Configuring collectd for JMX	634
Collecting our Application tier JVM logs	639

Monitoring the Tornado API application	647
Addressing the Tornado Application tier monitoring concerns	656
Summary	662
Chapter 13 Monitoring Tornado: Data tier	663
Monitoring the Data tier MySQL server	664
Using MySQL data for metrics	669
Query timing	675
Monitoring the Data tier's Redis server	681
Addressing the Tornado Data tier monitoring concerns	684
The Tornado dashboard	688
Expanding monitoring beyond Tornado	696
Summary	697
Appendix A An Introduction to Clojure and Functional Programming	699
A brief introduction to Clojure	701
Installing Leiningen	701
Clojure syntax and types	703
Clojure functions	704
Lists	708
Vectors	710
Sets	712
Maps	714
Strings	717
Creating our own functions	718
Creating variables	720
Creating named functions	721
Learning more Clojure	724
List of Figures	729
List of Listings	754

Index	755
--------------	------------

Foreword

Who is this book for?

This book is for engineers, developers, sysadmins, operations staff, and those with an interest in monitoring and DevOps. It provides a simple, hands-on introduction to the art of modern application and infrastructure monitoring.

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.

Credits and Acknowledgments

- Ruth Brown, who continues to be the most amazing person in my life.
- Kyle Kingsbury, for writing Riemann and being an excellent resource when I had dumb questions.
- Pierre-Yves Ritschard, for providing Riemann and Clojure help.
- Ben Lindsay, who provided feedback on the introductory Clojure material.
- Baron Schwartz, Dean Wilson, Brice Figureau and Marc Fournier, who provided valuable feedback on the book.
- Jeff Danzinger, for kindly letting me use his cartoon about averages.
- Simone Bottecchia, Katherine Daniels, Laurie Denness, Ryan Frantz, Kelvin Jasperson, Marc Fournier, Pierre-Yves Ritschard, Javier Uruen Val, Avleen

Vig, and John Vincent, for answering monitoring questions.

- The folks at PagerDuty, for giving me free access to their platform.
- Bimlendu Mishra, for his scripted Grafana dashboard example.
- Michael Jakl, for his RESTful Clojure example application.

Technical Reviewers

Caitie McCaffrey

Caitie McCaffrey is a Backend Brat and Distributed Systems Diva at Twitter, where she is the Tech Lead of the Observability Team. Prior to that she spent the majority of her career building large-scale services and systems that power the entertainment industry at 343 Industries, Microsoft Game Studios, and HBO. Caitie has a degree in Computer Science from Cornell University, and has worked on several video games, including Gears of War 2, Gears of War 3, Halo 4, and Halo 5. She maintains a blog at CaitieM.com and frequently discusses technology on Twitter @Caitie.

Paul Stack

Paul Stack is an infrastructure coder and is passionate about continuous integration, continuous delivery, and good operational procedures—and how they should be part of what developers and system administrators do on a day-to-day basis. He believes that reliably delivering software is as important as its development.

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 Bachelors 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. He lives with his family in Sydney, Australia.

Editor

Sid Orlando is a writer and editor (among other things), currently word-nerding out as Managing Editor at Kickstarter. Since starting work on more tech-focused projects, she may or may not be having recurring dreams about organizing her closet with dreamscape Docker containers.

Author

James is an author and open-source geek. His most recent books were [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 the [earlier book](#) about Puppet). He is the author of three other books, including [Pro Linux System Administration](#), [Pro Nagios 2.0](#), and [Hardening Linux](#).

For a real job, James is CTO at Kickstarter. He was formerly at Docker as VP of Services and Support, Venmo as VP of Engineering, and Puppet as VP of Technical Operations. 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.

Code and Examples

The code and example configurations contained in the book are available [on GitHub](#) at:

<https://github.com/jamtur01/aom-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 assume 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 monitoring program.

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 2016 - James Turnbull



Figure 2: ISBN

Version

This is version v1.0.3 (2c4a7d0) of The Art of Monitoring.

Chapter 1

Introduction

Let's begin with an origin story for a company called Example.com. Once upon a time(-series), Example.com had a sysadmin. She managed infrastructure that lived in data centers. Every time a new host was added to that environment she installed a monitoring agent and set up some monitoring checks. Every now and again one of those hosts would break and a check would trigger. A notification would be sent, and she would wake up and run `rm -fr /var/log/*.log` to fix it.

For many years this approach worked just fine. Of course, there was some drama. Occasionally something would go wrong for which there wasn't a check, or there just wasn't time to act on a notification, or some applications and services on top of the hosts weren't monitored. But largely, monitoring was fine.

Then the Information Technology (IT) industry started to change. Virtualization and Cloud computing were introduced, and the number of hosts that needed to be monitored increased by one or more orders of magnitude. Some of those hosts were run by people who weren't sysadmins, or the hosts were outsourced to third parties. Some of the hosts in her data center were moved into the Cloud, or even replaced with Software-as-a-Service applications.

Most importantly, IT became a core channel for businesses to communicate with and sell to their customers. Applications and services that had previously been

seen as just technology now became critical to customer satisfaction and providing high-quality customer service. IT was no longer a cost center, but something a company's revenue relied on.

As a result, aspects of monitoring began to break down. It became hard to keep track of hosts (there were a lot more of them), applications and infrastructure became more complex, and expectations around availability and quality became more aggressive.

It got harder and harder to check for all the possible things that could go wrong using the current system. Notifications piled up. More hosts and services meant more demand on monitoring systems—most of which were only able to scale by adding bigger, more powerful hosts, and could not be easily distributed. Under these loads, detecting and locating faults and outages grew ever slower and more challenging.

The organization began demanding more data to both demonstrate the quality of the service they were delivering to customers and to justify the increased spending on IT services. Many of these demands were made for data that existing monitoring simply wasn't measuring or couldn't generate. Her monitoring system became a tangled mess.

For many in the industry, this is the state of monitoring right now, and it's not a happy place. It doesn't have to be like this—you can build a better solution that addresses the change in how IT works and scales for the future.

Welcome to the Art of Monitoring

This is a hands-on guide to building a modern, scalable monitoring framework using up-to-date tools and techniques. We're going to build that framework from the ground up. We'll include best practices for both sysadmins and developers. We'll show developers how they can better enable monitoring and metrics, and we'll show sysadmins how to take advantage of metrics to do better fault detection and

gain insights into performance. We'll address the change in IT environments as a result of the dynamic infrastructure introduced by virtualization, containerization, and the Cloud. The goal of this book is to provide a monitoring framework that helps you and your customers better manage IT.

Before we launch into the guide, it's important to talk about what monitoring is, why it exists, and some of the challenges that exist in each monitoring domain.

We'll then talk about what's in the book, what you'll learn, and how you can change the way you perceive and implement monitoring.

What is monitoring?

From a technology perspective, **monitoring** is the tools and processes by which you measure and manage your IT systems. But monitoring is much more than that. Monitoring provides the translation between business value and the metrics generated by your systems and applications. Your monitoring system translates those metrics into a measurable user experience. That measurable user experience provides feedback to the business to help ensure it's delivering what customers want. The user experience also provides feedback to IT to indicate what isn't working and what's delivering insufficient quality of service.

Your monitoring system has two customers:

- The business
- Information Technology

The business as a customer

The first customer of your monitoring system is the business. Your monitoring exists to support the business—and to make sure it continues to do business. Monitoring provides the user experience data that allows the business to make good

product and technology investments. Monitoring also helps the business measure the value technology delivers.

Information Technology as a customer

IT is the second customer. That's you, your team, and the other folks who manage and maintain your technology environment. 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. Monitoring contributes much of the data that informs your critical product and technology decisions, and measures the success of those projects. It's a key part of your product management life cycle, your relationship with your internal customers, and it helps demonstrate that the business's money is being well spent. Without monitoring you are not doing your job.

What does monitoring actually look like?

So, does this vision of monitoring mesh with the real-world implementation of most monitoring systems? That depends. The evolution of monitoring in organizations varies dramatically, or as William Gibson put it:

The future is not evenly distributed.

To explore this we've created a three-level maturity model that reflects the various stages of monitoring evolution organizations tend to experience. The stages are:

- Manual, user-initiated, or no monitoring
- Reactive
- Proactive

We don't believe or claim this model is perfect. The stages identified are broad. Organizations may find they're at any number of points on the broad spectrums

inside those stages. Additionally, what makes measuring this maturity difficult is that not all organizations experience this evolution in linear or holistic ways. This can be the consequence of having employees with varying levels of skill and experience over different periods. It can be due to different segments, business units, or divisions of an organization having very different levels of maturity. Or it can be both.

Now on to the stages.

Manual, user-initiated, or no monitoring

Monitoring is largely manual, user initiated, or not done at all. If monitoring is performed, it's commonly managed via checklists, simple scripts, and other non-automated processes. Often monitoring becomes [cargo cult](#) behavior, with only the components that have broken in the past being monitored. Faults in these components are remediated by repeatedly following rote steps that have also "worked in the past."

The focus here is entirely on minimizing downtime and managing assets. Monitoring in this way provides little or no value in measuring quality or service, and provides little or no data that helps IT justify budgets, costs, or new projects.

This is typically found in small organizations with limited IT staffing, no dedicated IT staff, or where the IT function is run or managed by non-IT staff, such as a finance team.

Reactive

Reactive monitoring is mostly automatic with some remnants of manual or unmonitored components. Tooling of varying sophistication has been deployed to perform the monitoring. You will commonly see tools like Nagios with stock checks of basic concerns like disk, CPU, and memory. Some performance data may be collected. Most alerting will be based on simple thresholds, and sent via email

or messaging services. There may be one or more centralized consoles displaying monitoring status.

There is a broad focus on measuring availability and managing IT assets. There may be some movement towards using monitoring data to measure customer experience. Monitoring provides some data that measures quality or service and provides some data that helps IT justify budgets, costs, or new projects. Most of this data needs to be manipulated or transformed before it can be used. A small number of operationally focused dashboards exist.

This is typical in small to medium enterprises and common in divisional IT organizations inside larger enterprises. Typically reactive monitoring is built and deployed by an operations team. You'll often find large backlogs of notifications, and stale check configuration and architecture. Updates to monitoring systems tend to be reactive in response to incidents and outages. New monitoring checks are usually the last step in application or infrastructure deployments.

Proactive

Monitoring is considered core to managing infrastructure and the business. Monitoring is automatic and generated by configuration management. You'll see tools like Nagios, Sensu, and Graphite with widespread use of metrics. Checks will tend to be more application-centric, with many applications instrumented as part of development. Checks will also focus on measuring application performance and business outcomes rather than just stock concerns like disk and CPU. Performance data will be collected and used frequently for analysis and fault resolution. Alerting will be annotated with context and will likely include escalations and automatic responses.

There is a focus on measuring quality of service and customer experience. Monitoring provides data that measures quality or service and provides data that helps IT justify budgets, costs, or new projects. Much of this data is provided directly to business units, application teams, and other relevant parties via dashboards and

reports.

This is typical in web-centric organizations and many mature startups. This type of approach is also commonly espoused by organizations that have adopted a [DevOps](#) culture/methodology. Monitoring will still largely be managed by an operations team, but responsibility for ensuring new applications and services are monitored may be delegated to application developers. Products will not be considered feature complete or ready for deployment without monitoring.

Model distribution

[Broadly based on some of our monitoring research](#), we've created a distribution for our monitoring maturity model.

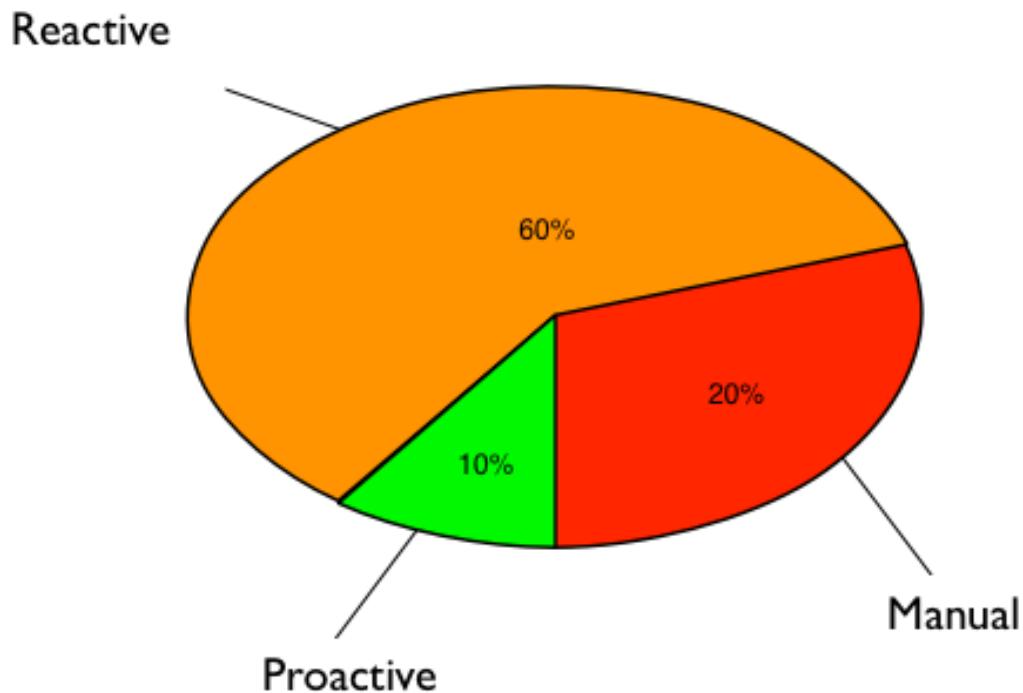


Figure 1.1: Monitoring Maturity Model Distribution.

As you can see, the vast majority of environments fall into the Reactive level, which may not come as a surprise to most engineers. The Reactive level of maturity is relatively simple to achieve and appears to satisfy most of the basic needs for monitoring.

As stated, neither the model nor the proposed distribution is perfect. But, even given the broadness of the potential distribution, we can make some architectural predictions about the implementation of monitoring inside a Reactive level organization.

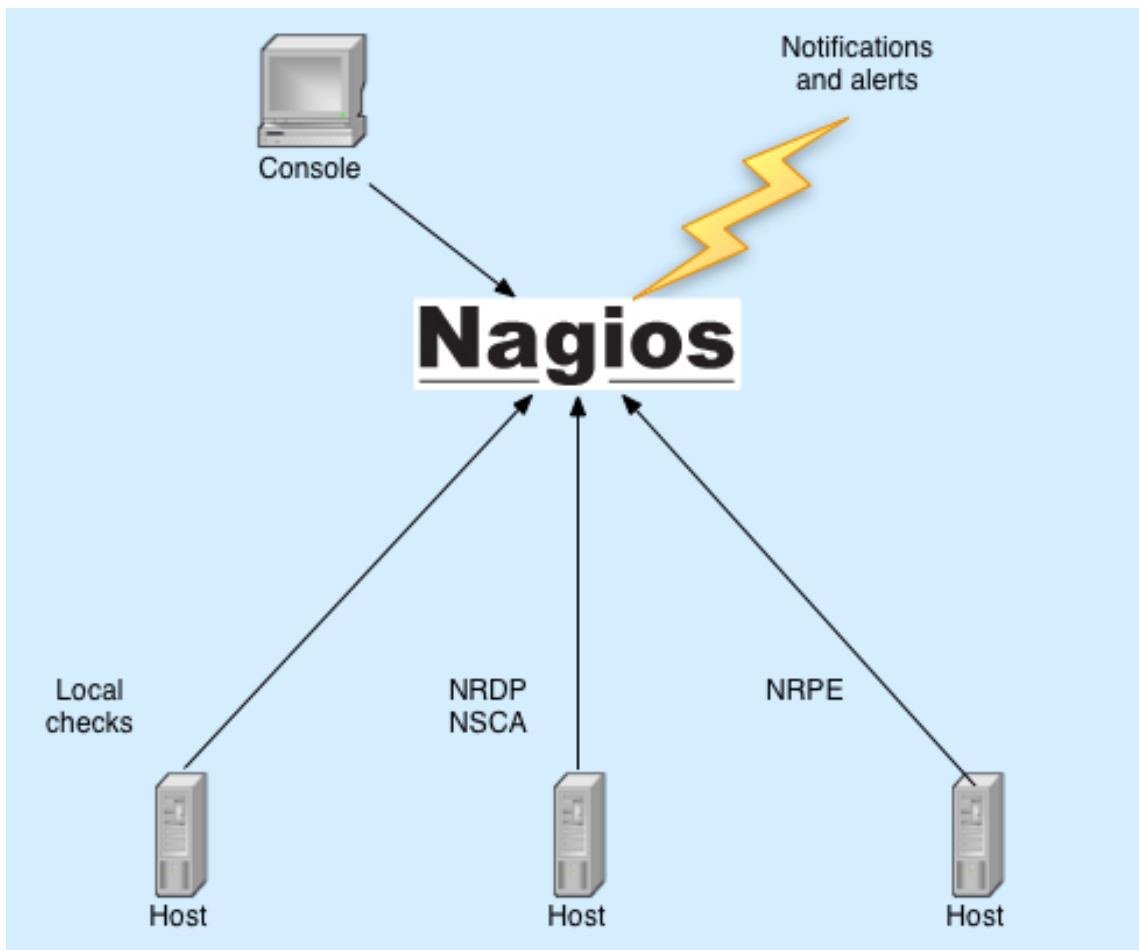


Figure 1.2: Traditional Monitoring

This image represents the classic monitoring configuration we've seen repeated in a wide cross-section of Reactive-level organizations.

We have a Nagios instance that runs host and service checks, sends SMS or email notifications when something is wrong, and serves as the primary dashboard for interacting with notifications. There are numerous variants of this base setup with both open source and commercial tools but this remains the basic configuration you're likely to see in Reactive-level organizations.

It's also a basic configuration that is fundamentally flawed. We talked earlier about the two customers of monitoring: the business and technology. Our Reactive environment doesn't serve the former at all and barely services the latter.

Becoming Proactive

A Reactive environment generates infrastructure-centric monitoring outputs: a host is down, a service is broken. There's no business or application-centric outputs. Without those outputs the business can't rely on monitoring to provide inputs to business decisions. You certainly can't use the data to justify budget for improving or updating the infrastructure. Or, often more importantly, for investing in your team.

As the Reactive environment is infrastructure-centric, it also only serves a segment of our technology customer—generally only operational teams—and doesn't provide useful, application-centric data to developers. As a result, non-operations staff are disconnected from the reality of the performance and availability of the infrastructure and applications being monitored. Developers usually receive outputs secondhand, discouraging accountability for issues and faults.

NOTE It's important to mention here that this critique of the Reactive model of monitoring does not (yet) touch on choices of tools and technology. This is not about picking on one tool or another or wars between toolchains. It's purely about the ability to deliver customers what they need, and to make it easier for you to do your job.

So how do we take our typical Reactive environment and turn it into a much more palatable Proactive environment? Measurement. We're going to update our Reactive environment to focus on events, metrics, and logs. We'll replace a lot of our existing monitoring infrastructure—for example, service and host-centric checks—and replace them with event and metric-driven checks.

In our monitoring framework, events, metrics, and logs are going to be at the core of our solution. The data points that make up our events, metrics, and logs will provide the source of truth for:

- The state of our environment.
- The performance of our environment.

So, rather than infrastructure-centric checks like pinging a host to return its availability or monitoring a process to confirm if a service is running, we're going to replace most of those fault detection checks with metrics.

If a metric is measuring then the service is available. If it stops measuring then it's likely the service is not available.

Visualization of those events, metrics, and logs will also allow for the ready expression and interpretation of complex ideas that would otherwise take thousands of words or hours of explanation.

In Chapter 2, we'll walk through our proposed framework in detail including how we've chosen to design it, and why we've chosen certain types of tools and techniques.

To help articulate this framework in the book we've used a make-believe company called Example.com so you can see what a real-world build might look like. Let's take a quick look at the world of Example.com. Example has three main sites:

- Production A
- Production B (DRP)
- Mission Control

Each site is geographically separated. We're going to focus on applications in our production site, Production A, but we're going to show you how you can build as resiliently as possible across multiple sites. Example also has a DRP site, Production B, and a Mission Control site that contains management infrastructure including consoles and dashboards. Where relevant, we'll demonstrate how to connect these sites into your monitoring framework as well.

Example also has test environments. In the real world we'd replicate much, if not all, of our new monitoring in these environments. This helps catch regressions and performance issues, and helps ensure monitoring is a first-class requirement when building applications and services.

Example is primarily a Linux environment, running recent versions of Red Hat Enterprise Linux and Ubuntu, and operates a number of customer-facing internal and external applications. Almost all of its applications are web based, with the stack including:

- Java and JVM-based applications
- Ruby on Rails
- LAMP-stack applications

Their database stack is a mix of MySQL/MariaDB, PostgreSQL, and Redis.

Much of the environment is managed with configuration management tools, and each environment has a Nagios server for monitoring.

Lastly, Example is beginning to explore the use of tools like Docker, and SaaS products like GitHub, PagerDuty, and others.

This environment provides a representative sample of technologies you're likely to manage yourself that can be adapted to a wide variety of other environments and stacks.

What's in the book?

In this book, you'll learn how to build a monitoring framework. We'll describe our proposed framework in Chapter 2 and build it, piece by piece, in subsequent chapters, then finally make use of the framework to monitor infrastructure, services, and applications.

It's really important to understand that this isn't a monitoring bible for every technology stack. We do use a lot of example applications covering a wide range of technologies to show you how to monitor different components. We don't, however, provide detailed lists of exactly what you should monitor for every technology stack. This is because every environment and application is developed, built, and coded differently. Every organization also has different architecture and monitoring objectives, thresholds, and concerns.

We'll explore much of what you might need to monitor, identify critical checks, and introduce a series of patterns you can adopt or adapt. You should be able to build the framework into a solution for your organization that meets your specific needs.

Let's look at what's in each chapter.

- Chapter 1: This introduction.
- Chapter 2: Our monitoring framework: monitoring, metrics, and measurement. This chapter provides background on the decisions and architecture of our monitoring framework.

- Chapter 3: Managing events and metrics with an event router called [Riemann](#).
- Chapter 4: Storing and visualizing metrics with [Graphite](#) and [Grafana](#).
- Chapter 5: Host monitoring with [collectd](#).
- Chapter 6: Using [collectd](#) events in Riemann and Graphite.
- Chapter 7: Monitoring containers. In this chapter we look at monitoring containers, primarily [Docker](#).
- Chapter 8: Collecting logs for diagnosis and status, covers the [Elasticsearch Logstash Kibana](#) or ELK stack.
- Chapter 9: Building monitored applications: How to add instrumentation, metrics, logging, and events to your applications.
- Chapter 10: Notifications: Building contextual and human-friendly notifications.
- Chapters 11 to 13: Monitoring a stack. We'll put all our components together to monitor an example host, service, and application stack. These chapters will present a full picture of how our framework will work.
- Appendix A: An introduction to Clojure, which Riemann uses as a configuration language. (We recommend you read this prior to Chapter 3.)

Finally, one topic we're not covering directly in the book is the monitoring of non-host devices: networking equipment, storage devices, data center equipment. However, many of the techniques we're exploring in the book can be replicated on these kinds of devices. Modern devices allow you to push metrics, provide metric and status endpoints, and generate appropriate events and logs.

Tool choices

In this book we look at mostly free and open source monitoring tools and solutions. There are a number of commercial tools and online services that provide monitoring services but we won't cover them in much detail.

We recognize that there are a lot of moving pieces here. You might look at the list of tools we're introducing and say "that's a lot of software I have to learn and manage." To help with this, the book is arranged so that you can potentially implement pieces of the framework rather than the whole. Most chapters have a stand-alone component that could use in addition to integration with the other components.

We've also chosen tools we think are best of breed in their domains. These choices are based on research, experience, and consultation with colleagues in the industry. Where possible, in each chapter we've listed alternative tools you could explore if you find the tools introduced don't suit you or don't meet your needs.

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.

Chapter 2

A Monitoring and Measurement Framework

In forthcoming chapters, we'll build our monitoring framework. But first, in these initial chapters, we're going to look at data collection, metrics, aggregation, and visualization. Then we'll expand the framework to collect application and business metrics, culminating in a capstone chapter where we'll put everything together. We'll build a framework that focuses on events and metrics and collects data in a scalable and robust way.

In our new monitoring paradigm, events and metrics are going to be at the core of our solution. This data will provide the source of truth for:

- The state of our environment
- The performance of our environment

Visualization of this data will also allow for the ready expression and interpretation of complex ideas that would otherwise take thousands of words or hours of explanation.

In this chapter we're going to step through our proposed monitoring framework. We'll introduce the basic concepts and lay the groundwork that will help you understand the choice of tools and techniques we've made later in the book.

To implement our monitoring framework we're proposing a new architecture.

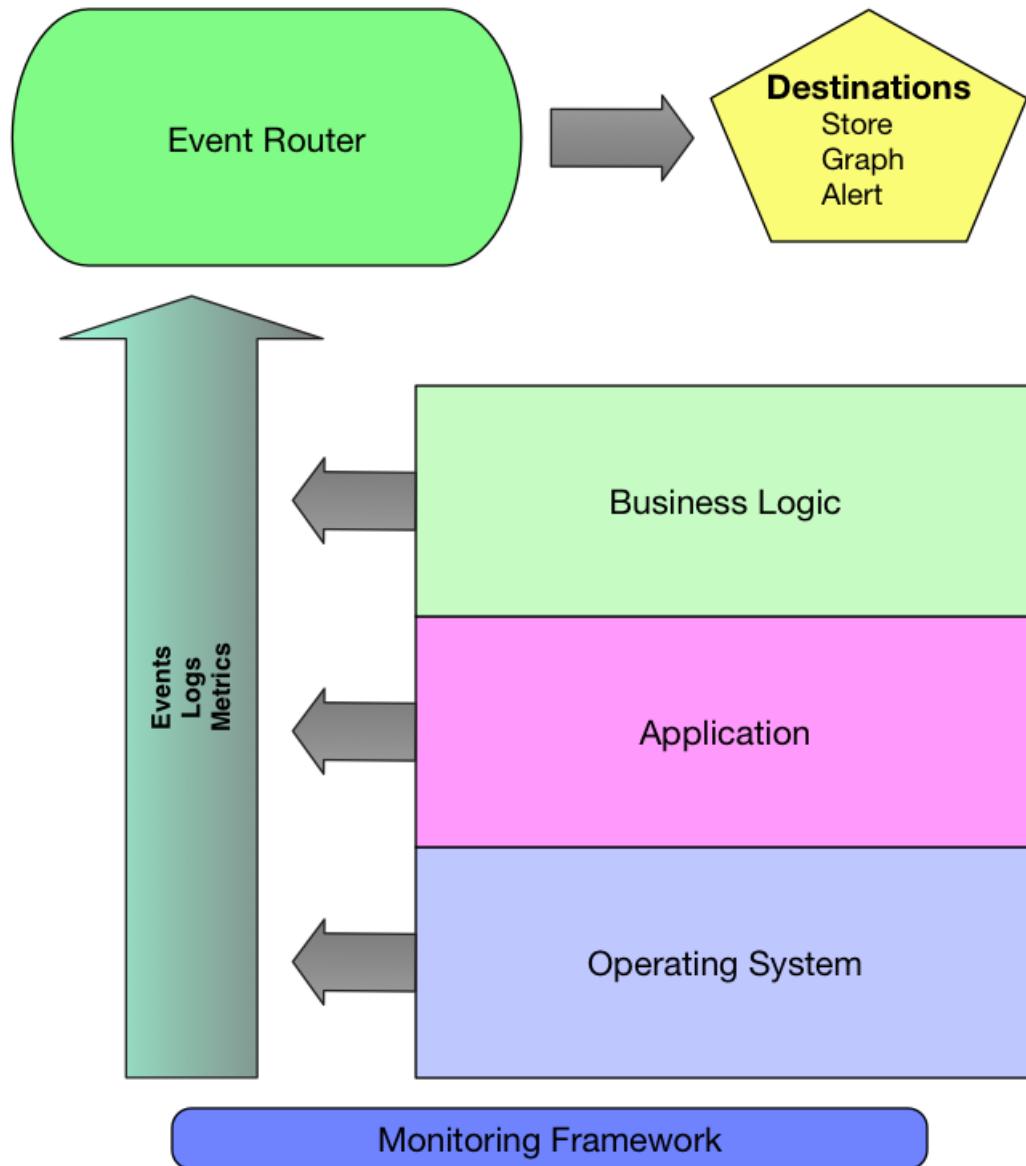


Figure 2.1: Monitoring framework

Our new architecture is going to:

- Allow us to easily visualize the state of our environment.
- Be event, log, and metrics-centric.
- Focus on “whitebox” or push-based monitoring instead of “blackbox” or pull-based monitoring.
- Provide contextual and useful notifications.

These objectives will allow us to take our Reactive Example.com environment closer to the Proactive model, and to ensure we monitor the right components in the right way.

Let’s examine our new architecture.

Blackbox versus Whitebox, or Pull versus Push

We’re going to fundamentally change the architecture of how we perform monitoring. Most monitoring systems are “blackbox” or pull/polling-based systems. An excellent example is Nagios. With Nagios, your monitoring system generally queries the components being monitored; a classic check might be an ICMP-based ping of a host. This means that the more hosts and services you manage in your environment, the more checks your Nagios host needs to execute and process. We then need to scale our monitoring vertically or via partition to address growth.

We’re going to, wherever possible, avoid “blackbox” monitoring in favor of “whitebox” or push-based monitoring. With a “whitebox” (hereafter push-based) architecture, hosts, services, and applications are emitters, sending data to a central collector. The collection is fully distributed on the hosts, services, and applications that emit data, resulting in linear scalability. This means monitoring is no longer a monolithic central function, and we don’t need to vertically scale or partition that monolith as more checks are added.

Emitters report when they are available. Generally emitters are stateless, sending data as soon as it is generated. They can use transports and mechanisms local and appropriate to themselves, rather than being forced into a choice by your monitoring tools. This enables us to build modular, functionally separated, compartmentalized monitoring solutions with selected best-of-breed tools rather than monolithic silos.

“Blackbox” (hereafter pull-based) approaches also require your monitoring targets to be centrally configured on what and where to monitor. With push, your emitters, host, services, and applications send data when they start, and push metrics at destinations you’ve configured. This is especially important in dynamic environments, where a short-lived activity might not have sufficient time to be discovered or converged into configuration by a pull-based monitor. With a push-based architecture this isn’t an issue because the emitter controls when and where the data is sent.

We also get a broad security dividend from a push-based architecture: emitters are inherently more secure against remote attacks since they do not listen for network connections. This decreases the attack surface of our hosts, services, and applications. Additionally, this reduces the operational complexity of any security model, as networks and firewalls only need to be configured for unidirectional communication from emitters to collector.

Polling-based systems also generally emphasize monitoring availability—“Is it up?”—and the minimization of downtime.¹ Where we do use polling systems we’ll limit their focus to this sort of coarse-grained availability monitoring. Polling-based systems also provide a strong focus on small, atomic actions—for example, telling you that an Nginx daemon has stopped working. This can be hugely attractive, because fixing those atomic actions is often much easier and simpler than addressing more systemic issues, such as a 10% side-wide increase in HTTP 500 errors.

¹With some notable exceptions like Prometheus and Google’s Borgmon.

You may be thinking, “Hey, what’s wrong with that?” Well, there’s nothing fundamentally wrong with it except that it reinforces the view that IT is a cost center. Orienting your focus toward availability, rather than quality and service, treats IT assets as pure capital and operational expenditure. They aren’t assets that deliver value, they are just assets that need to be managed. Organizations that view IT as a cost center tend to be happy to limit or cut budgets, outsource services, and not invest in new programs because they only see cost and not value.

Thankfully, IT organizations have started to be viewed in a more positive light. Organizations have recognized that it’s not only impossible to do business without high-quality IT services but that they are actually market differentiators. If you do IT better than your competitors then this is a marketable asset. Adding to this is the popularity and flexibility of virtualization, elastic computing like Cloud, and the introduction of Software-as-a-Service (SaaS). Now the perception has started to move IT from a cost center to, if not an actual revenue center, then at least a lever for increasing revenue. This change has consequences though, with the most important being that we now need to measure the quality and performance of IT, not just its availability. This data is crucial to the business and technology both making good decisions.

Push-based models also tend to be more focused on measurement. You still get availability measurement, but as a side effect of measuring components and services. As collection is distributed and generally low overhead, you can also push a lot of data and store it at high precision. This increased precision of data can then be used to more quickly answer questions about quality of service, performance, and availability, and to power decisions around spending, headcount, and new programs. This changes the focus inside your IT organization towards measuring value, throughput, and performance—all levers that are about revenue rather than cost.

Event, log, and metric-centered

Our new push-centric architecture is going to be centered around collecting event and metric data. We'll use that data to monitor our environment and detect when things go wrong.

- Events — We'll generally use events to let us know about changes and occurrences in our environment.
- Logs — Logs are a subset of events. While they're helpful for letting us know what's happening, they're often most useful for fault diagnosis and investigation.
- Metrics — Of all these data sources, we'll rely most heavily on metrics to help us understand what's going on in our environment. Let's take a deeper look at metrics.

More about 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 our metrics.

Indeed, in a lot of monitoring frameworks, the focus is on fault detection: detecting if a specific system event or state has occurred (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.

We're going to change this idea of metrics-as-supplement. Metrics are going to be the most important part of your monitoring workflow. We're going to turn the fault-detection-centric model on its head. Metrics will provide the state and availability of your environment and its performance.

Our framework 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 in pieces of software or hardware. To make a metric useful we keep track of its state, generally recording data points or observations over time. An observation is a value, a timestamp, and sometimes a series of properties that describe the observation, such as a source or tags. The combination of these data point observations is called a time series.

A classic example of a metric we might collect as a time series 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. We'll talk more in Chapter 4 about this.

So, a time-series metric is generally 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.

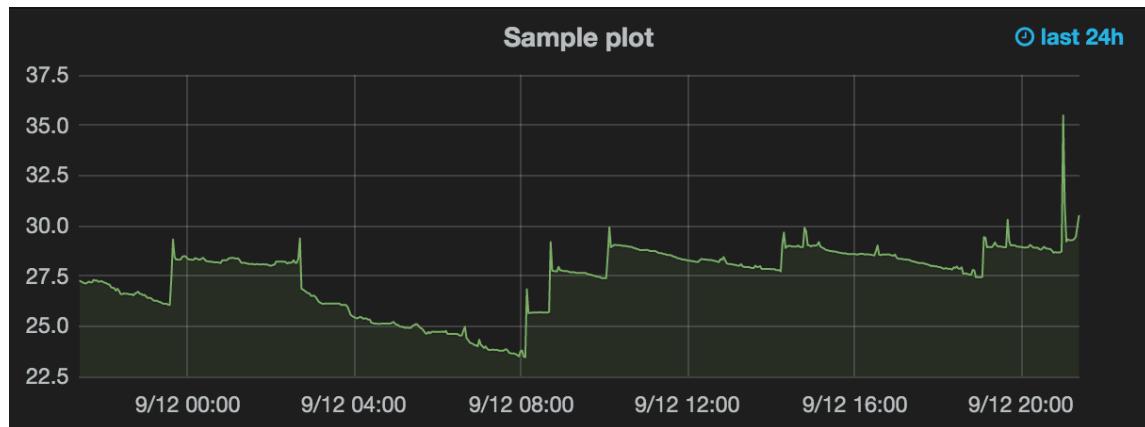


Figure 2.2: A sample plot

These plots can be incredibly useful to us. 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 we'll see in our environment.

Gauges

The first, and most common, type of metric we'll see is a gauge. Gauges are numbers that are expected to change over time. 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.

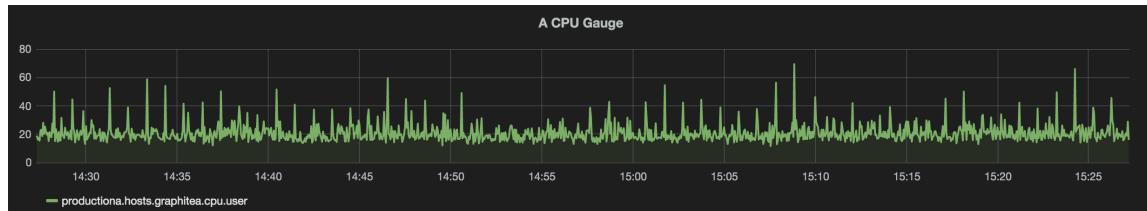


Figure 2.3: A sample gauge

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 cost of sales for a time period.

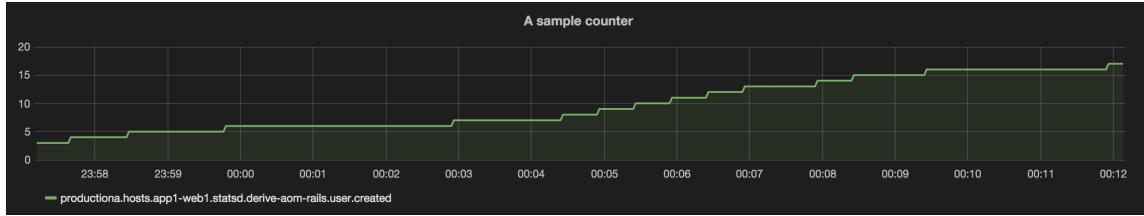


Figure 2.4: A sample counter

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

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 range 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.

Timers

We'll also see a small selection of timers. Timers track how long something took. They are commonly used for application monitoring—for example, you might embed a timer at the start of a specific method and stop it at the end of the method. Each invocation of the method would result in the measurement of the time the method took to execute.

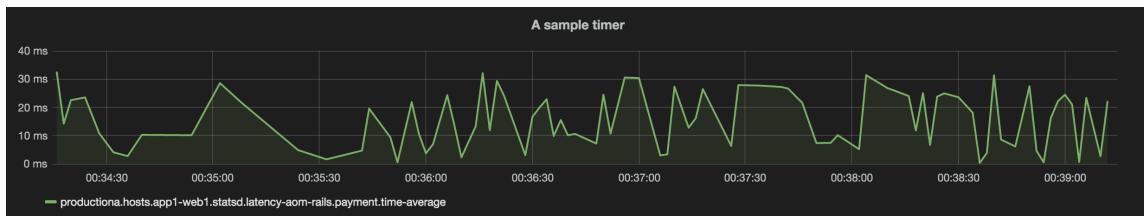


Figure 2.5: A sample timer

Here we have a timer measuring the average time in milliseconds of the execution of a payment method.

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 — We count the number of observations in a specific time interval.
- Sum — We *sum* (add together) values from all observations in a specific time interval.
- Average — The *mean* of all values in a specific time interval.
- Median — The *median* is the dead center of our values: exactly 50% of values are below it, and 50% are above it.
- Percentiles — Measure the values below which a given percentage of observations in a group of observations fall.
- Standard deviation — 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.

- Frequency distribution and histograms - This is a frequency distribution of a data set. You group data together—a process which is called “binning”—and present the groups in a such a way that their relative sizes are visualized. The most common visualization of a frequency distribution is a histogram.

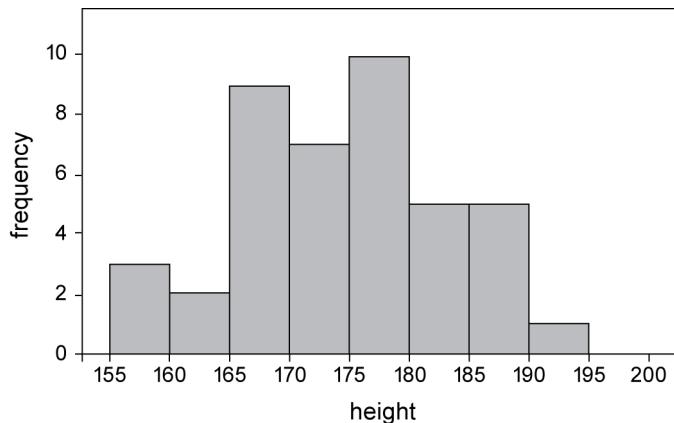


Figure 2.6: A histogram example

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.

TIP This is a brief introduction to these summary methods. We'll talk about 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 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.

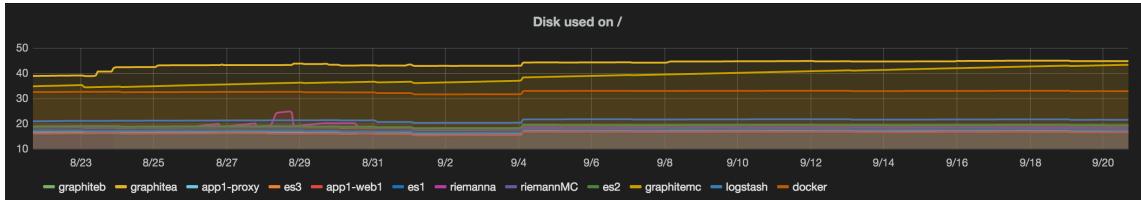


Figure 2.7: An aggregated collection of metrics

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 a combination of single and aggregate metrics—the former to drill down into specific issues, the latter to see the high-level state—provide the most representative view of the health of your environment.

Contextual and useful notifications

Notifications are the primary output from our monitoring architecture. They can consist of emails, instant messages, SMS messages, pop-ups, or anything else used to let you know about things in your environment that you need to be aware of. This seems like it should be a really simple domain but it contains a lot of complexity and is frequently poorly implemented and managed.

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

- 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 will generally mute them. We all have war stories of mailbox folders full of thousands of notification emails from monitoring systems. 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 2.1: 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%)
```

Now 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. That the `/data` volume is 91% full. This seems useful at first glance but in reality it's really not that practical. Firstly, is this a sudden increase? Or has this grown gradually? What's the rate of expansion? For example, 9% disk space free on a 1GB partition is different from 9% disk free on a 1TB disk. Can I ignore or mute this notification or do I need to act now? Without this additional context my ability to take action on this notification is limited and I 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 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 con-

tain additional information about the component we're notifying on.

- Aligning our notifications with the business needs of the service being monitored so we only notify on what's useful to the business.

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

In Chapter 10 we'll build notifications with greater context and add a notification system to the monitoring framework we're building.

Visualization

Visualizing data is both an incredibly powerful analytic and interpretive technique and an amazing learning tool. Throughout the book we'll look at ways to visualize the data and metrics we've collected. However 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 sudden leaps from correlation to causation. This 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'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 our 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.

So why this architecture? What's wrong with traditional monitoring?

In Chapter 1 we talked broadly about the problem space, how IT has changed, and why traditional monitoring fails to address that change. Let's look more deeply into what's broken and why this new architecture addresses those gaps.

When we describe "traditional monitoring," especially in Reactive environments, what we're usually talking about is fault detection. It is best articulated as watching an object so we know it's working. Traditional monitoring is heavily focused on this active polling of objects to return their state—for example, an ICMP ping-based host availability check.

Historically fault detection checks have relied on Boolean decisions that indicate whether something responds or if a value falls within a range. Check selection and implementation is also simplistic and may be:

- Experience or learning-based. You implement the same checks you've used in the past, or acquired through [cargo cult](#)'ed monitoring checks from sources like documentation, example configurations, or blog posts.

- Reactive. You implement a check or checks in response to an incident or outage that has occurred in the past.

Boolean check design and experience-based and Reactive checks have some major design flaws. Let's examine why they are issues.

Static configuration

The checks generally have static configuration. Your check probably needs to be updated every time your system grows, evolves, or changes. In virtual and cloud environments, a host or service being monitored may be highly ephemeral: appearing, disappearing, or migrating locations or hosts multiple times during its lifespan. Statically defined checks just don't handle this changing landscape, resulting in checks (and faults) on resources that do not exist or that have changed.

Further, many monitoring systems require you to duplicate configuration on both a server and the object being monitored. This lack of a single source of truth leads to increased risk of inconsistency and difficulty in managing checks. It also generally means that the monitoring server needs to know about resources being monitored before they can be monitored. This is clearly problematic in dynamic or changing landscapes.

Additionally, updates to monitoring are often considered secondary to scaling or evolving the systems themselves. Many faults are thus the result of incorrect configuration or orphaned checks. These false positives take time and effort to diagnose and resolve. They clutter your monitoring environment, hiding actual issues and concerns. Many teams do not realize they can change or delete the existing checks to remove these false positives—they take the monitoring as gospel.

Inflexible logic and thresholds

The 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 again

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 it could even be a natural consequence of growth.

It's our view that arbitrary static thresholds are always wrong. Database performance analysis vendor [VividCortex](#)'s CEO [Baron Schwartz](#) 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.

Arbitrary static thresholds set up a point-in-time boundary. During that period everything beneath that boundary is judged normal and everything above is abnormal. That boundary is not only inflexible but it's entirely artificial. One system's abnormality may be another's normal operation. That means notifications wired to arbitrary thresholds will frequently fire off false positives, unrelated to any actual problems.

Boolean checks suffer similar issues to arbitrary thresholds. They are usually singletons, and often can't take advantage of trends or prior event history. Is this really a failure? Is it a critical failure? Is it merely flapping? Could one or more failures of this check (or even across a series of checks) actually be survivable, especially in the context of a resilient and well-architected application?

Object-centric

The checks are object-centric, usually centric to single hosts or services. They require you to define a check on an object or objects. This breaks down quickly because each of those objects is usually part of a much larger, often much more complex system. These single object checks frequently lack any context and limit

your ability to understand what the check's output means to the broader system. As a consequence it is often hard to determine the criticality of the object's failure. Of course, some monitoring systems do attempt to provide contextual layers above object checks, usually via grouping, but rarely manage to model beyond basic constructs. They also lack the ability to process the dynamism in most modern environments.

An interlude into pets and cattle

By the end of this book a lot of folks are probably going to be surprised by how few fault detection checks we actually build. Traditional monitoring environments are often marked by thousands of checks. So why aren't we going to replicate those environments? Well, as we've discovered prior, those sorts of environments aren't easy to manage, scale, or massively duplicate, and often aren't actually helpful in doing fault diagnosis. There's another factor, though, related to how the process of fault resolution is changing.

Bill Baker, a former Distinguished Engineer at Microsoft, once quipped that hosts are either pets or cattle. Pets have sweet names like Fido and Boots. They are lovingly raised and looked after. If something goes wrong with them you take them to the vet and nurse them back to health. Cattle have numbers. They are raised in herds and are basically identical. If something goes wrong with one of them, you put it down and replace it with another.

In the past hosts were pets. If they broke you fixed them, often nursing a host—named for a Simpson's character—back to life multiple times, tweaking configuration, fiddling with settings, and generally investing time in resolving the issue.

In modern environments, hosts are cattle. They should be configured automatically and rebuilt automatically. If a server fails then you kill it and restart another, automatically building it back to a functioning state. Or if you need more capacity you can add additional hosts. In these environments you don't need hundreds or

thousands of checks on individual components because the default fix for significant numbers of those components is to rebuild the host or scale the service.

So what do we do differently?

We've identified issues with traditional fault detection checks, and we're advocating replacing these traditional status checks with events and metrics, but what does that mean? Rather than infrastructure-centric checks like pinging a host to return its availability or monitoring a process to confirm if a service is running, we configure our hosts, services, and applications to emit events and metrics. We get two benefits from events and metrics, firstly:

If a metric is measuring, an event is reporting, or a log is spooling, then the service is available. If it stops measuring or reporting then it's likely the service is not available.

NOTE What do we mean by available? The definition, for the purposes of this book, is that a host, service, or application is operable and functioning in line with expectations.

How will this work? The event router in our monitoring framework is responsible for tracking our events and metrics. It can potentially do a lot of useful things with those events and metrics including storing them, sending them to visualization tools, or using them and their values to notify us of performance issues. But most importantly it knows about the existence of those events and metrics. Let's look at an example. We configure a web server to emit metrics showing the current workload. We then configure our event router to detect:

- If the metric stops being reported.
- If the value of a metric matches some criteria we've developed.

In the former case, if the metric disappears from our event router, we can be fairly certain that something has gone wrong. Either the web server has stopped working or something has happened between us and the server to prevent data from reaching our event router. In either case we've identified a fault that we may wish to investigate.

In the latter case, we get useful data from the payload of the event or metric. Not only is this data useful for long-term analysis of trends, performance, and capacity but it presents an opportunity to build a new paradigm for checking state. In our traditional monitoring model we rely on arbitrary thresholds to determine if we have an issue—for example, polling our CPU usage and reporting a warning if it is above a certain percentage. Now, instead of checking those arbitrary thresholds, we use a smarter approach. We can't totally eliminate the need to set thresholds, but we can make our analysis a lot smarter by making the inputs to our thresholds more intelligent.

Smarter threshold inputs

In our new model we still use thresholds but the data we feed into those thresholds is considerably more sophisticated. We will generate better data and analysis and get a better understanding of the experience of our users from our collected metrics. All of this leads to the identification of valid issues and problems. In our new monitoring framework we will:

- Collect frequent and high-resolution data.
- Look at windows of data not static points in time.
- Calculate smarter input data.

Using this methodology we're more likely to identify if a state is an actual issue instead of an anomalous spike or transitory state.

We'll look at collection of high-frequency data and techniques for viewing windows of data in the forthcoming chapters. But calculating smarter input data for

our thresholds and checks requires some explanation of some of the possible techniques we could choose and some we shouldn't use. Let's take a look at why, why not, and how we might use 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. As a result, exploration of some topics may appear overly simplistic to folks with strong statistical or mathematical backgrounds.

Average

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 distribution](#)—for example, in our average response time it's assumed all events run at equal speed or that response time distribution is roughly [bell curved](#). But this is rarely the case with applications. 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.

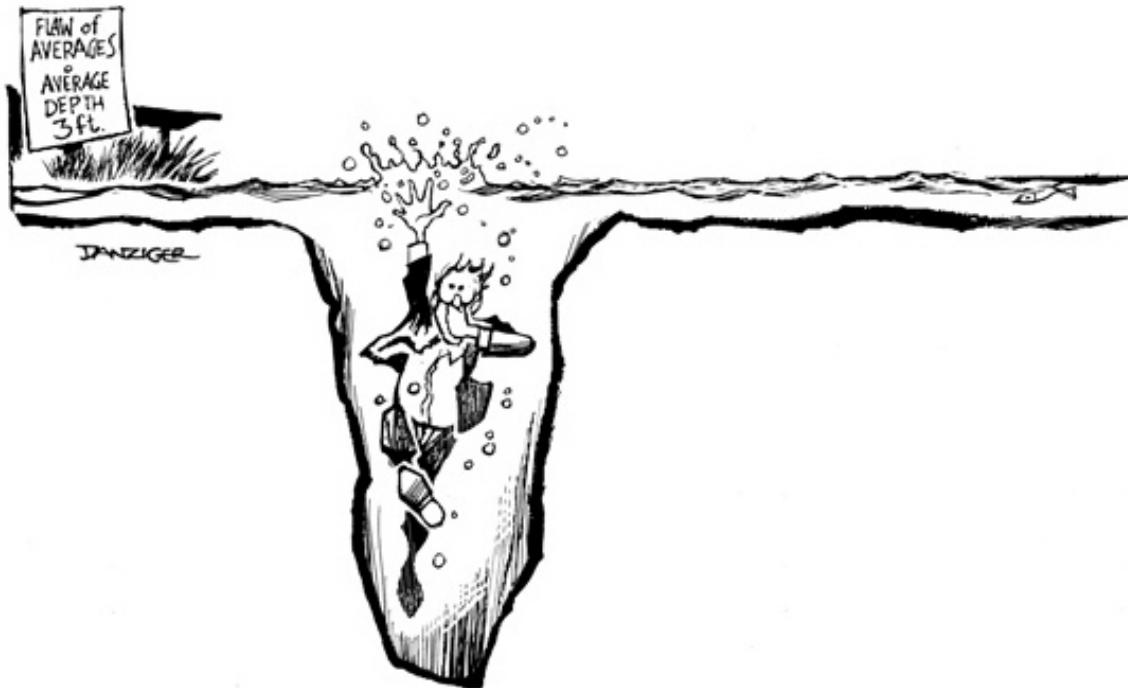


Figure 2.8: The flaw of averages - Copyright Jeff Danzinger

So 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 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 are potentially a significant number who are not.

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

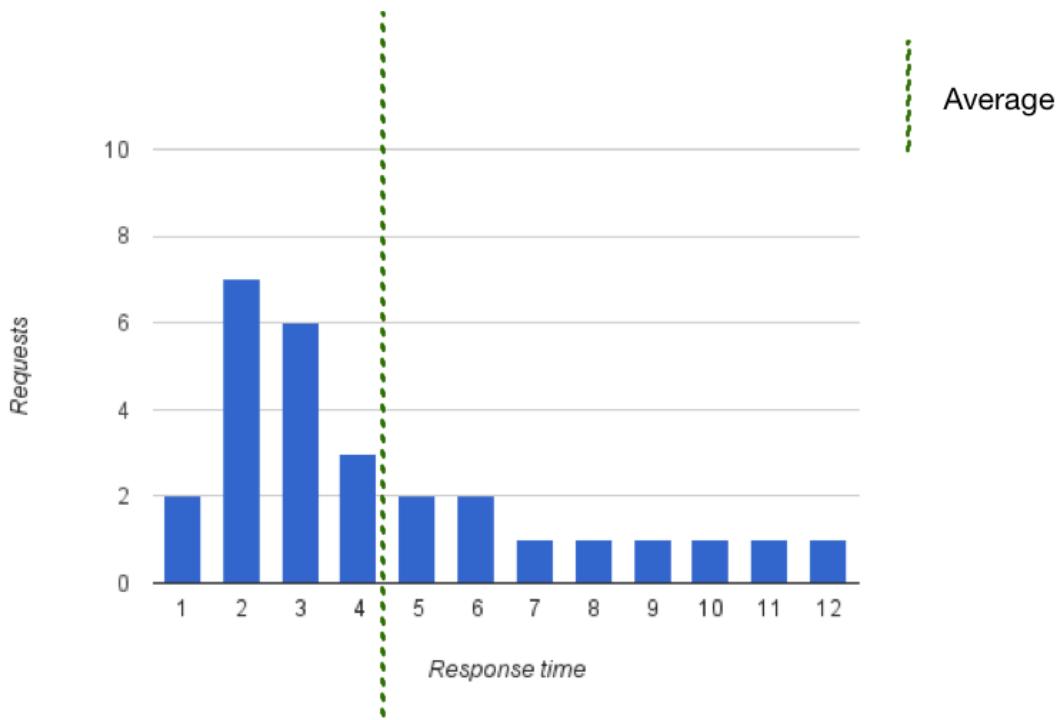


Figure 2.9: Response time average

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.

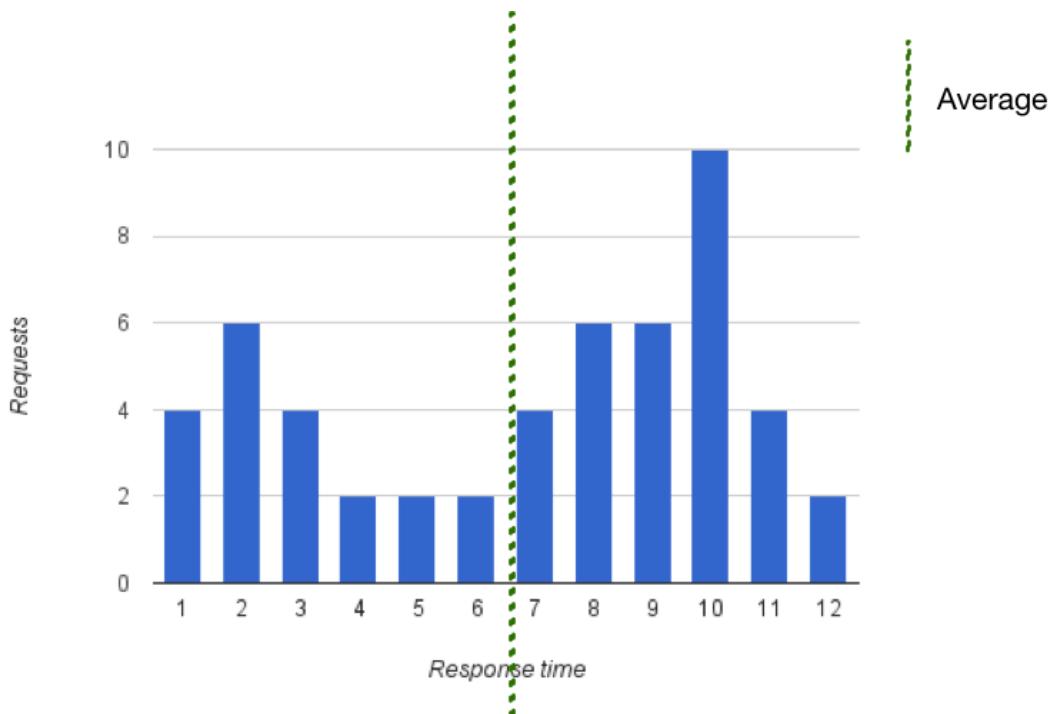


Figure 2.10: Response time average Mk II

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 our users are experiencing it.

Median

At this point you might be wondering about using the median. The median is the dead center of our values: exactly 50% of values are below it, and 50% are above it. If there are 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.

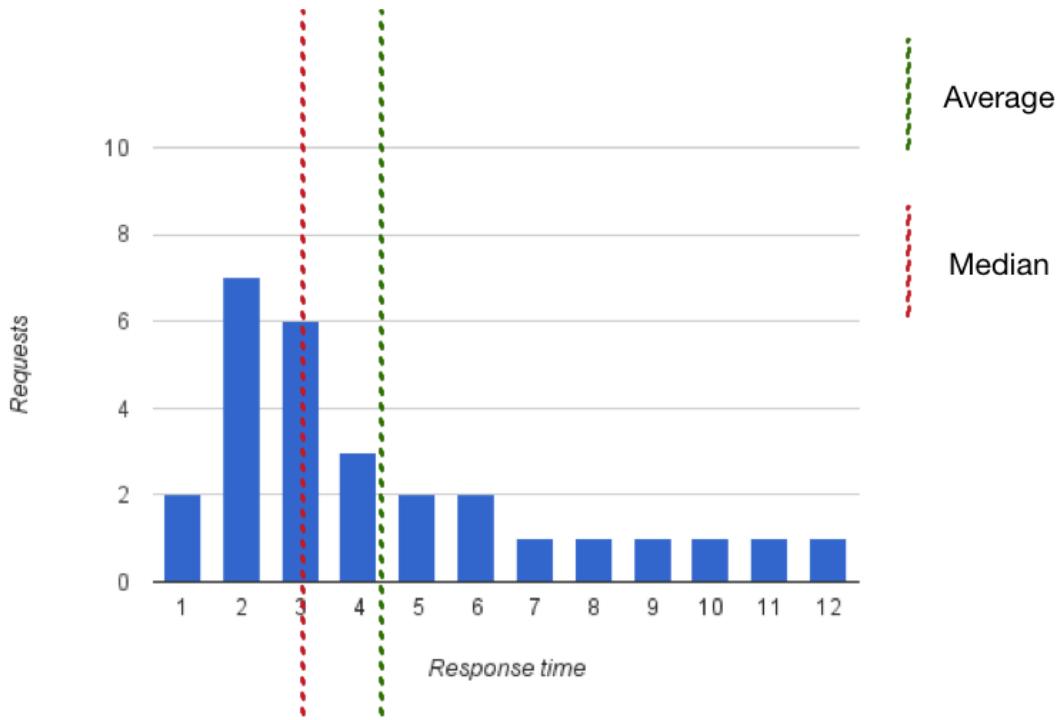


Figure 2.11: Response time average and median

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.

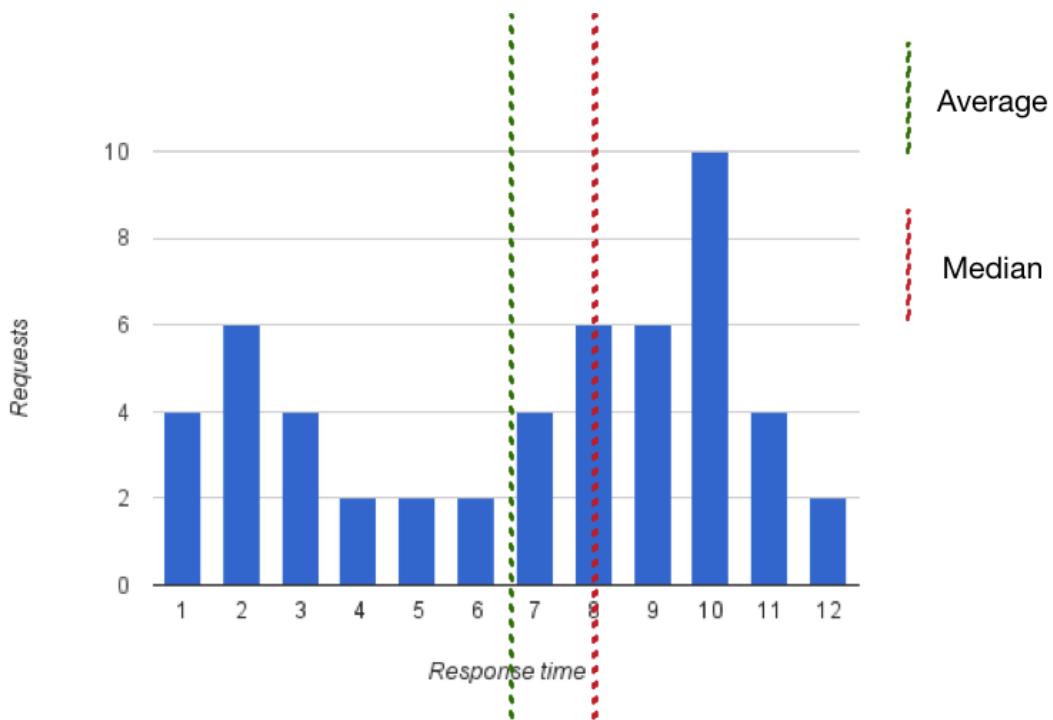


Figure 2.12: Response time average and median Mk II

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 σ or *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. Indeed, in a normal distribution there's a simple way of articulating the distribution: the [empirical rule](#). Within the rule, one standard deviation or 1 to -1 will represent 68.27% of all transactions on either side of the mean, two standard deviations or 2 to -2 would be 95.45%, and three standard deviations will represent 99.73% of all transactions.

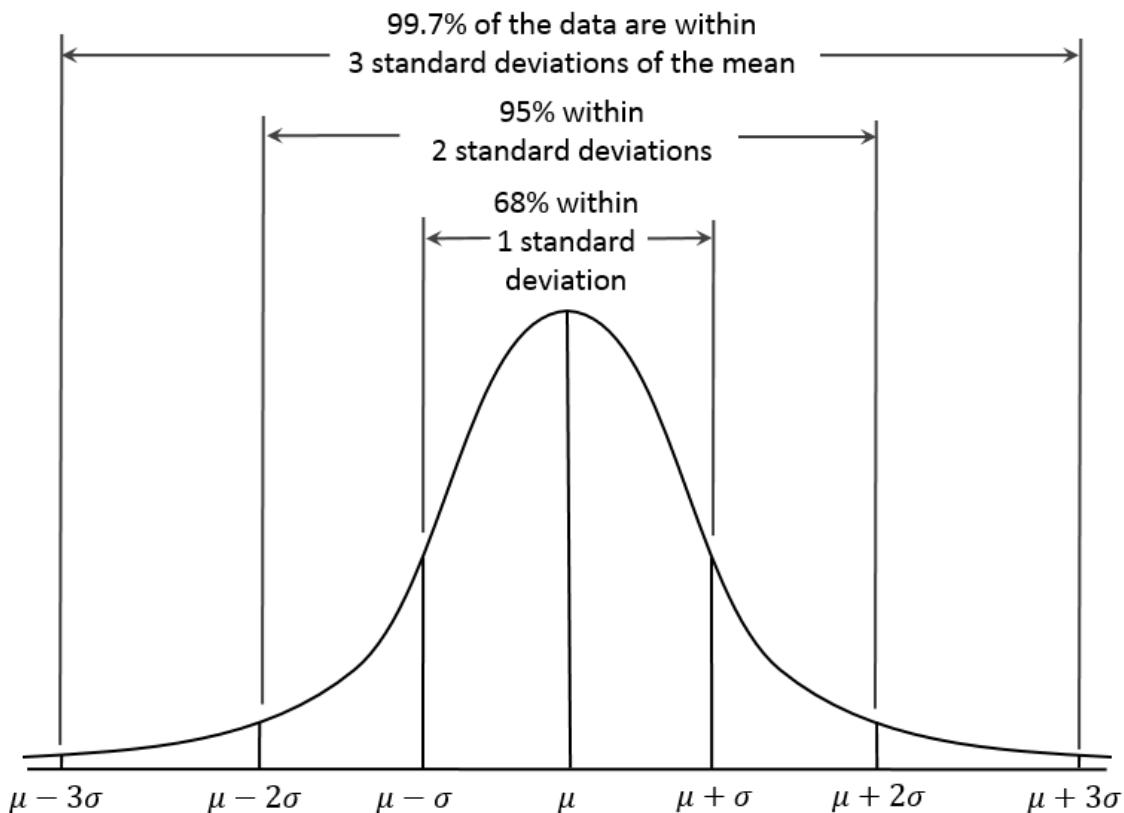


Figure 2.13: The empirical rule

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, will change that.

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% of values fall below and 50% 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% of transactions were completed in 10 milliseconds or less, and 1% of transactions took more than 10 milliseconds.

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

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.

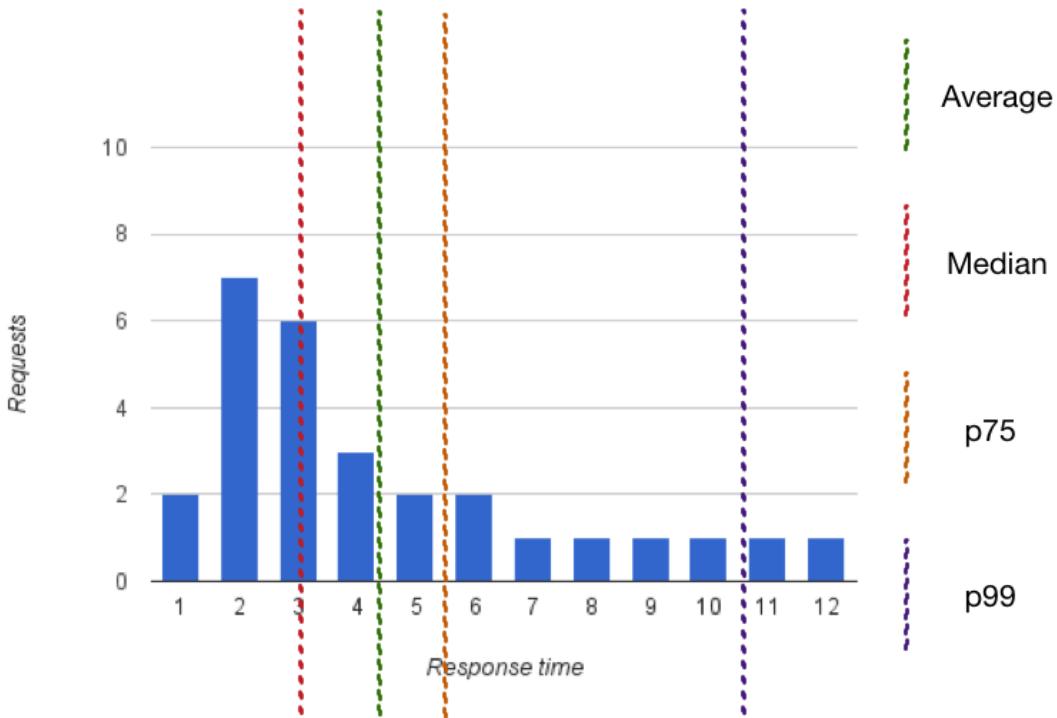


Figure 2.14: Response time average, median, and percentiles

We see that the 75th percentile is 5.5 seconds. That indicates that 75% completed in 5.5 seconds, and 25% of them were slower. 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% of users had request times of less than 10.74 seconds, and 1% 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% of users getting 10.74 second response times or better and 1% being slower 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.

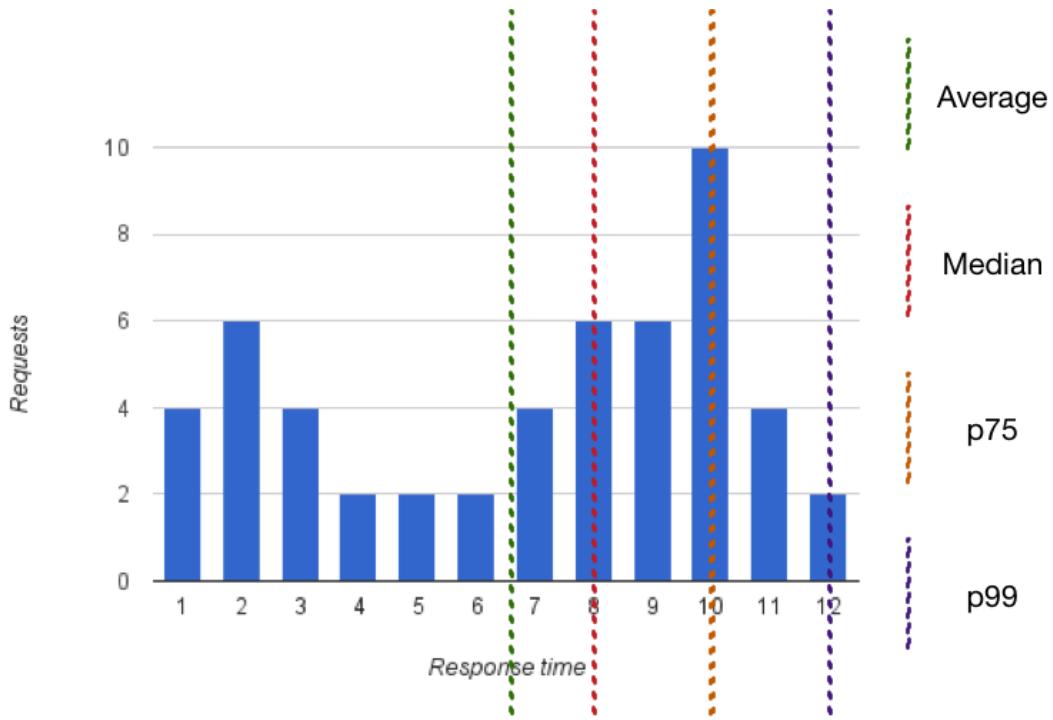


Figure 2.15: Response time average, median, and percentiles Mk II

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. Our recommendation is to graph several combinations of metrics to get a clear picture of the data. For example, when measuring latency it's often best to display a graph that shows:

- The 50th percentile (or median).
- The 95th and 99th percentiles.
- 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.

Collecting data for our monitoring framework

In our framework we're going to focus on agent-based collection of data. We're going to prefer the running of local agents on hosts, and focus on the instrumentation of applications and services. Wherever possible each host or service will be self-contained and responsible for emitting its own monitoring data. We'll locally configure collection and the destination of our data.

In keeping with our push-based architecture we'll try to avoid remote checks of hosts and services. With a few exceptions—like external monitoring of hosts and applications—that we'll discuss in Chapter 9, we'll rarely poll hosts, services, and applications from remote pollers or monitoring stations.

Our data collection will include a mix of data:

- Resource information, like consumption of CPU or memory
- Performance information, like latency and application throughput
- Business and user-experience metrics, like volumes or the amounts of transactions or numbers of failed logins
- Log data from hosts, services, and applications

We'll use much of the data and observations we collect directly as metrics. In some cases we'll also convert observations in the form of events into metrics.

Overhead and the observer effect

One thing to consider when thinking about data collection is that the process of collecting the data can also impact the values being collected. In normal operation many of the methods we use to collect data will consume some of the resources we're monitoring. Sometimes this overhead becomes excessive and can actually influence the state of what you're monitoring, or worse, trigger notifications and outages. This is often called [the observer effect](#), derived from [the related physics concept](#). The methods we're going to use will focus on making that overhead as small as possible, but you should remain conscious of the effect. Granular collections—for example, hitting an HTTP site or probing an API endpoint aggressively—could result in your monitoring check being a measurable percentage of the service's capacity.

Summary

In this chapter we've articulated the framework we're going to build to monitor our environment. We've talked about the push versus pull architecture and the focus on events and metrics. We've also discussed why we've chosen that architecture and what's wrong with some of the monitoring alternatives out there. We then walked through an introduction to some of the monitoring and metrics principles we're going to use throughout the book.

In the next chapter we launch our monitoring framework with the introduction of our event routing engine.

Chapter 3

Managing events and metrics with Riemann

In Chapter 2 we talked about events, metrics, and logs, and how we're going to use them. In this chapter we're going to build the base of our monitoring framework: the routing engine we described that will input and process those events, metrics, and logs.

Our design for an event router is:

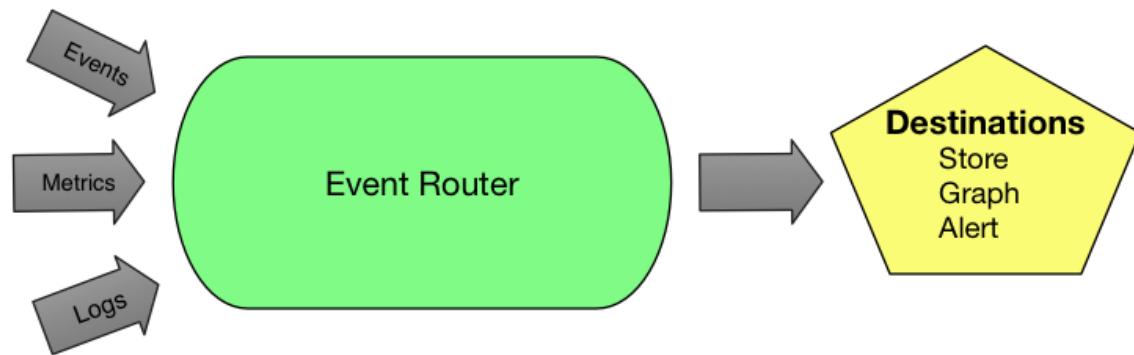


Figure 3.1: Event Routing

With this design we want our routing engine to:

- Receive our events and metrics (we'll talk more about logs in Chapter 8) including scaling as our environment grows.
- Maintain sufficient state to allow us to do event matching; provide context for notifications and for checks based on trending.
- Munge events including extracting metrics from events.
- Categorize and route data to be stored, graphed, alerted on, or sent to any other potential destinations.

To achieve these objectives we're going to look at a tool called [Riemann](#). Riemann is an event-based tool for monitoring distributed systems. It works in a push model, receiving events rather than polling for them. We're going to use Riemann as our routing engine. Our hosts, services, and applications will send their events into Riemann, and Riemann will make the necessary decisions about those events.

Introducing Riemann

If only I had the theorems! Then I should find the proofs easily enough.

— Bernard Riemann

So why Riemann? [Riemann](#) is a monitoring tool that aggregates events from hosts and applications and can feed them into a stream processing language to be manipulated, summarized, or actioned. The idea behind Riemann is to make monitoring and measuring events an easy default.

Riemann can also track the state of incoming events and allows us to build checks that take advantage of sequences or combinations of events. It provides notifications, the ability to send events onto other services and into storage, and a variety of other integrations.

Overall, Riemann has functionality that addresses all of our objectives. It is fast and highly configurable. Throughput depends on what you do with each event, but stock Riemann on commodity x86 hardware can handle millions of events per second at sub-millisecond latencies.

Riemann is [open source](#) and licensed with the [Eclipse Public license](#). It is primarily authored by [Kyle Kingsbury](#) aka Aphyr. Riemann is written in Clojure and runs on top of the [JVM](#).

Riemann architecture and implementation

In the Introduction we talked a little about the topology of our Example.com environment with Production A, Production B, and Mission Control environments. We're going to deploy Riemann servers in each environment. We're also going to deploy downstream servers in the Mission Control environment to allow us to roll up events and "monitor the monitors." To do this we'll send Riemann's own status events downstream, and we'll setup notifications if these events stop flowing.

In Chapter 4 we're going to pair Riemann with Graphite, a real-time time-series data graphing engine, that can store and graph metrics generated from events. We'll also introduce Grafana, a tool to visualize the data we're collecting.

In summary we will:

- Install Riemann servers in the Production A and Production B environments.
- Install Graphite and Grafana servers in the Production A and Production B environments (coming up in Chapter 4).
- Configure the downstream Riemann, Graphite, and Grafana servers in our Mission Control environment.

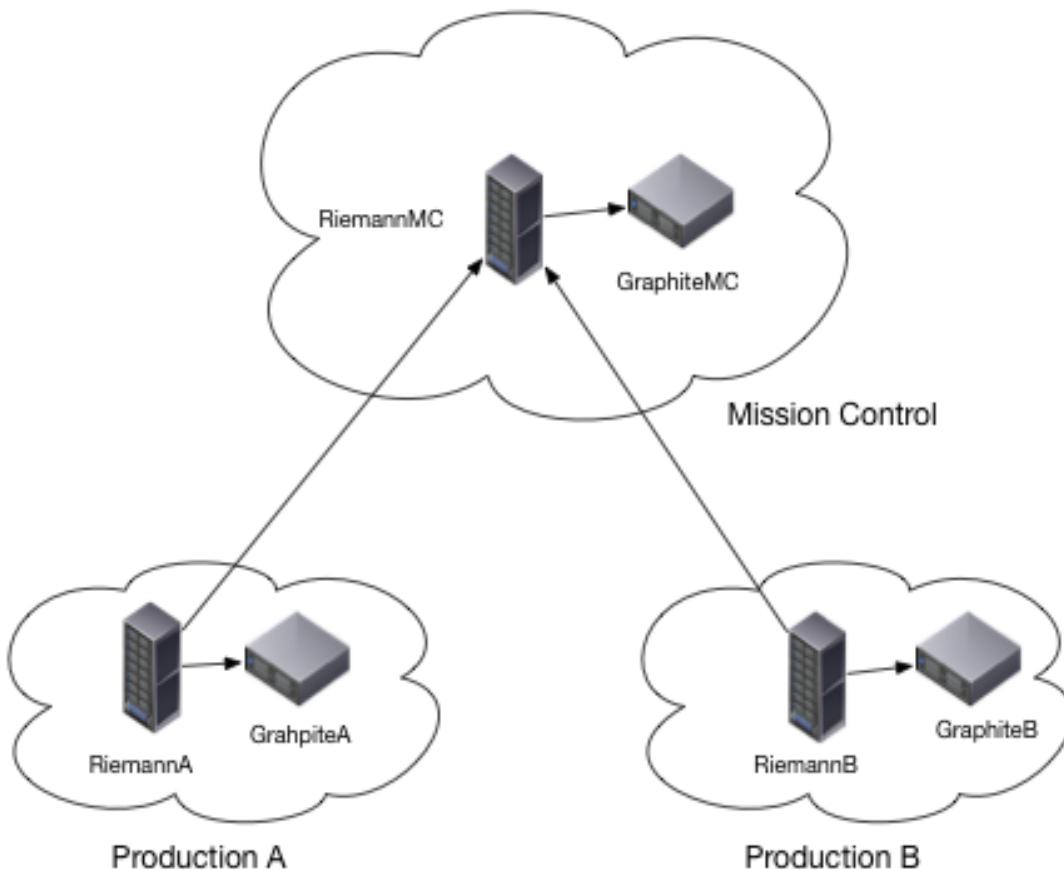


Figure 3.2: Metrics Architecture

Let's take a look at installing Riemann now.

Installing Riemann

We're going to install Riemann onto three hosts:

- An Ubuntu 14.04 host in Production A with a hostname of `riemannA.example.com` and an IP address of 10.0.0.110
- A Red Hat Enterprise Linux (RHEL) 7.0 host in Production B with a hostname of `riemannB.example.com` and an IP address of 10.0.0.120.

- An Ubuntu 14.04 host in Mission Control with a hostname of `riemannmc.example.com` and an IP address of 10.0.0.100.

NOTE Here, and throughout the book, we're going to assume you've configured DNS and local host resolution to find and resolve these hosts.

We're going to conduct a manual installation of the required prerequisites and packages to give you an understanding of how Riemann works, but in the real world we'd use a configuration management tool.

Installing Riemann on Ubuntu

We're going to do our first Riemann installation on the `riemann.a.example.com` host which is an Ubuntu 14.04 host.

Prerequisites for Ubuntu

First, we'll need Java to run Riemann itself. For Java we're going to use the default OpenJDK available on Ubuntu.

Listing 3.1: Installing Java on Ubuntu

```
$ sudo apt-get -y install default-jre
```

Then let's check Java is installed correctly.

Listing 3.2: Checking Java is installed on Ubuntu

```
$ java -version
java version "1.7.0_65"
OpenJDK Runtime Environment (IcedTea 2.5.3) (7u71-2.5.3-0ubuntu0
.14.04.1)
OpenJDK 64-Bit Server VM (build 24.65-b04, mixed mode)
```

Installing the Riemann package on Ubuntu

Now we're going to install Riemann itself. We're going to use [the Riemann project's own DEB packages](#). Also available are RPM packages and tarballs. I am going to do a manual install so you can see the steps involved, but you could just as easily use the configuration management content above.

Let's grab the DEB package of the current release. You should check the Riemann site for the latest version and update this command to get that package.

Listing 3.3: Fetching the Riemann DEB package

```
$ wget https://aphyr.com/riemann/riemann_0.2.11_all.deb
```

And then install it via the `dpkg` command.

Listing 3.4: Installing the Riemann package on Ubuntu

```
$ sudo dpkg -i riemann_0.2.11_all.deb
```

The Riemann DEB package installs the `riemann` binary and supporting files, service management, and a default configuration file.

We'd then repeat this installation for the second Ubuntu host, `riemannmc.example.com`.

Installing Riemann on Red Hat

We're going to do our second installation on the `riemannb.example.com` host, which is a Red Hat Enterprise Linux (RHEL) 7 host.

Prerequisites for Red Hat

Again we need to have Java installed. Let's install the package we require.

Listing 3.5: Installing Java and prerequisites on RHEL

```
$ sudo yum install -y java-1.7.0-openjdk
```

TIP On newer Red Hat and family versions the `yum` command has been replaced with the `dnf` command. The syntax is otherwise unchanged.

Here we've installed Java. Let's now test Java on this host.

Listing 3.6: Checking Java is installed on Red Hat

```
$ java -version
java version "1.7.0_75"
OpenJDK Runtime Environment (rhel-2.5.4.2.el7_0-x86_64 u75-b13)
OpenJDK 64-Bit Server VM (build 24.75-b04, mixed mode)
```

Installing the Riemann package on Red Hat

Now we're going to install Riemann itself. We're going to use [the Riemann project's own RPM packages](#). Again we're going to do a manual install but we could easily use configuration management.

Let's grab the RPM package of the current release. Check the Riemann site for the latest version and update this command to get that package.

Listing 3.7: Fetching the Riemann RPM package

```
$ wget https://aphyr.com/riemann/riemann-0.2.11-1.noarch.rpm
```

Then install it via the `rpm` command.

Listing 3.8: Installing the Riemann package on RHEL

```
$ sudo rpm -Uvh riemann-0.2.11-1.noarch.rpm
```

The Riemann RPM package installs the `riemann` binary and supporting files, service management, and a default configuration file.

Installing Riemann via configuration management

You could also install Riemann via a variety of configuration management tools like Puppet or Chef or via Docker or Vagrant.

You can find a Chef cookbook for Riemann at:

- <https://github.com/hudl/riemann-cookbook>

You can find a Puppet module for Riemann at:

- <https://forge.puppetlabs.com/garethr/riemann>

You can find an Ansible role for Riemann at:

- <https://github.com/dhruvbansal/riemann-server-ansible-role>

You can find Docker images for Riemann at:

- <https://hub.docker.com/search/?q=riemann>

You can find a Vagrant configuration for Riemann at:

- <https://github.com/garethr/riemann-vagrant>

Running Riemann

Now that we've installed Riemann on our hosts, we'll run it. Riemann can run interactively via the command line or as a daemon. If we're running it as a daemon we use the service management commands:

Listing 3.9: Starting and stopping Riemann

```
$ sudo service riemann start  
$ sudo service riemann stop  
..
```

We can also run Riemann interactively using the `riemann` binary. To do this we need to specify a configuration file. Conveniently the installation process has added one at `/etc/riemann/riemann.config`.

Listing 3.10: Running Riemann interactively

```
$ sudo riemann /etc/riemann/riemann.config  
loading bin  
INFO [2014-12-21 18:13:21,841] main - riemann.bin - PID 18754  
INFO [2014-12-21 18:13:22,056] clojure-agent-send-off-pool-2 -  
riemann.transport.websockets - Websockets server 127.0.0.1 5556  
online  
INFO [2014-12-21 18:13:22,091] clojure-agent-send-off-pool-4 -  
riemann.transport.tcp - TCP server 127.0.0.1 5555 online  
INFO [2014-12-21 18:13:22,099] clojure-agent-send-off-pool-3 -  
riemann.transport.udp - UDP server 127.0.0.1 5555 16384 online  
INFO [2014-12-21 18:13:22,102] main - riemann.core - Hyperspace  
core online
```

We see that Riemann has been started and a couple of servers have also been started: a WebSockets server on port 5556, and TCP and UDP servers on port 5555. By default Riemann binds to `localhost`.

NOTE Don't use UDP to send events to Riemann. UDP has no guarantee of delivery, ordering, or duplicate protection. You will lose events and data.

The default configuration logs to `/var/log/riemann/riemann.log` and you can also follow the daemon's activity there.

You can stop the interactive Riemann server with a `Ctrl-C` on the command line.

NOTE The Riemann packages also add a `riemann` user and group that Riemann runs by default.

Installing Riemann's supporting tools

Lastly, let's install a final supporting piece on all our Riemann hosts: the Riemann tools.

The Riemann tools are a collection of small programs that can be used to submit events to Riemann. They include tools for monitoring web services, local hosts, applications, and databases. We're going to use these tools for some local testing. You can see the repository for the Riemann tool on GitHub [here](#).

NOTE In Chapter 5 we'll explore host monitoring using collectd.

To install the tools we need to install Ruby and a compiler on our host. We can remove it afterward if we're concerned it's a security risk on the host. On Ubuntu

we would install:

Listing 3.11: Installing supporting tools prerequisites on Ubuntu

```
$ sudo apt-get -y install ruby ruby-dev build-essential zlib1g-dev
```

Or on Red Hat distributions we would install:

Listing 3.12: Installing supporting tools prerequisites on RHEL

```
$ sudo yum install -y ruby ruby-devel gcc libxml2-devel
```

We're going to install the supporting tools via Ruby Gems.

Listing 3.13: Installing Riemann's supporting tools

```
$ sudo gem install --no-ri --no-rdoc riemann-tools
```

TIP After installation you can remove the build tools we installed if required. Don't remove Ruby—you'll need that to run the supporting tools.

There's also a Riemann dashboard available that is provided by the `theriemann-dash` gem. It's basic and you can find its source code [on GitHub](#). We're not going to use it in the book, but it's useful for creating graphs and viewing events locally on the Riemann host—for example, when quickly doing diagnostics or checking out state

and status. You can find out more in [the Riemann Dashboard documentation](#).

Configuring Riemann

Riemann is configured using a Clojure-based domain-specific language, or DSL, configuration file. This means your configuration file is actually processed as a Clojure program. To process events and send notifications and metrics you'll be writing Clojure. Don't panic—you won't need to become a full-fledged Clojure developer to use Riemann. We'll teach you what you need to know. Riemann also comes with a lot of helpers and shortcuts that make it easier to write Clojure to do what we need to process our events.

Learning some Clojure

Your first step in learning how to configure Riemann is learning a little bit of Clojure, just enough to get started and build our first few monitoring checks. Later in the book we'll introduce you to further concepts and syntax that you'll find useful. To start learning, please flip over to [Appendix A - An Introduction to Clojure and Functional Programming](#) at the back of the book. Alternatively, Kyle Kingsbury, the author of Riemann, has written an excellent series called [Clojure from the ground up](#) that should greatly help you understand Clojure.

TIP We **strongly** recommend you read the appendix before continuing. It'll help you understand how Riemann's configuration DSL works and help you get started using it.

Riemann's base configuration

Now that we've installed Riemann let's look at how to configure it. The package installation installs a default configuration file at `/etc/riemann/riemann.config`. We're going to replace that file with a new initial configuration.

To do this edit the `/etc/riemann/riemann.config` file and add the following content.

Listing 3.14: New `/etc/riemann/riemann.config` configuration file

```
(logging/init {:file "/var/log/riemann/riemann.log"})

(let [host "127.0.0.1"]
  (tcp-server {:host host})
  (udp-server {:host host})
  (ws-server {:host host}))

(periodically-expire 5)

(let [index (index)]
  (streams
    (default :ttl 60
      index

      #(info %))))
```

TIP Any line or string prefixed with a ; is a comment.

We see the file is broken into a few stanzas. The first stanza sets up Riemann's logging to a file: `/var/log/riemann/riemann.log`.

Listing 3.15: Riemann logging stanza

```
(logging/init {:file "/var/log/riemann/riemann.log"})
```

TIP I strongly recommend you manage your Riemann configuration file with a configuration management tool and/or version control. Also useful is if your editor supports syntax highlighting and validation. Clojure uses a lot of braces, brackets, and parentheses and ensuring they are in order and matched can be tricky.

In this case we're calling a function, `logging/init`. We've specified the namespace of the function, `logging`, and the name of the function, `init`, and then any subsequent arguments. Namespaces are a way of organizing code in Clojure and we'll talk more about them later in this chapter. In this case our argument is a `map`. Our map contains any options we want to pass to our `logging/init` function.

Inside our map we've specified a single option, `:file`, with a value of `/var/log/riemann/riemann.log`. The `:file` option is a [Clojure keyword](#). A keyword is a label, much like a Ruby symbol, and it's commonly used inside collections like maps to mark the key in a key/value pair.

In summary we're calling the `logging/init` function and passing it a map, in this case containing only one option: the name of the file in which to write our logs.

The second stanza controls Riemann's interfaces. Riemann generally listens on TCP, UDP, and a WebSockets interface. By default, the TCP, UDP, and WebSockets interfaces are bound to the `127.0.0.1` or `localhost`.

- TCP is on port 5555.
- UDP is on port 5555.
- WebSockets is on port 5556.

We see that the definition of our interface configuration is inside a stanza starting with `let`. We're going to see `let` quite a bit in our configuration. The `let` expression creates lexically scoped immutable aliases for values. Or, in more simple terms, it defines a meaning for a symbol or symbols within a specific expression.

What does this mean? Let's look at our interface configuration.

Listing 3.16: The let form

```
(let [host "127.0.0.1"]
  (tcp-server {:host host})
  (udp-server {:host host})
  (ws-server {:host host}))
```

The `let` expression takes a `vector` of one or more bindings. Bindings are pairs of symbols and values that are bound for that expression. [Symbols are pointers to values](#)—for example, the symbol `host` has a value of `127.0.0.1`. The binding is only valid locally to our `let` expression. This is useful because it allows you to do things like override existing values of symbols inside the current expression.

We use these bindings in the subsequent expressions. In our interface example we're saying:

“Let the symbol `host` be `127.0.0.1` and then call the `tcp-server`, `udp-server`, and `ws-server` functions with that symbol as the value of the `:host` option.”

This sets the host interface of the TCP, UDP, and WebSockets servers to `127.0.0.1`.

A `let` binding is lexically scoped, i.e., limited in scope to the expression itself. Outside of this expression the `host` symbol would be undefined. The `host` symbol

is also immutable inside the expression in which it is defined. You cannot change the value of `host` inside this expression. This is an excellent example of why functional programming is useful. Nothing inside the expression can change the value (state) of `host`, which ensures that every time the expression is evaluated the same result will be achieved.

The `let` expression is useful for configuring Riemann because it is simple, readable, and—because of the clear scope and immutable state— reloads cleanly when we want to change configuration.

For our purposes having Riemann bound to only the `localhost` isn't overly useful. Let's make a quick change here to bind these servers to all available interfaces.

Listing 3.17: Exposing Riemann on all interfaces

```
(let [host "0.0.0.0"]
  (tcp-server {:host host})
  (udp-server {:host host})
  (ws-server {:host host}))
```

We've updated the value of the `host` symbol from `127.0.0.1` to `0.0.0.0`. This means if one of your interfaces is on the Internet then your Riemann server is now on the Internet and accessible to everyone. If you need more security you can also [configure Riemann with TLS](#).

We could also adjust the ports being used by Riemann by adding the `:port` argument to our map.

Listing 3.18: Changing the Riemann port

```
(let [host "0.0.0.0"
      tcp-port 5555]
  (tcp-server {:host host :port tcp-port}))
```

In addition to the other servers, Riemann also has a built-in REPL server you can use to test your Riemann configuration. You can add it to the servers you've enabled by adding `(repl-server {:host host})` to your server configuration. You may want to bind it to the localhost as `(repl-server {:host "127.0.0.1"})` to prevent inappropriate access. You can connect to it using the `lein` binary from inside a checkout of the Riemann source code.

TIP We talk about REPL servers in Appendix A. They are useful for testing with Riemann and Clojure.

Listing 3.19: Connecting to the Riemann REPL server

```
$ git clone git://github.com/riemann/riemann.git
$ cd riemann
$ lein repl :connect 127.0.0.1:5557
```

To make these changes we need to reload or restart Riemann. If we're reloading Riemann then it will respond to `SIGHUP`, using `kill -HUP <Riemann PID>` or from inside the REPL server.

Listing 3.20: SIGHUP from the Riemann REPL server

```
user=> (riemann.bin/reload!)
```

We strongly recommend using **SIGHUP** to reload the configuration. Riemann has hot loading of configuration and in most cases this will allow your event flow to be less impacted. You'll then see a message about reloading the config in the Riemann log file. You could also use the service management tools on your host.

Listing 3.21: Restarting Riemann

```
$ sudo service riemann reload
```

TIP If you make a configuration mistake or a syntax error then Riemann will continue running with the old config and won't apply the new one. It'll also log an error message detailing the issue so you can fix it up.

The next stanza configures indexing and streams. Both of these topics need special attention because they are at the heart of what makes Riemann so powerful. Let's look at each of these concepts now.

Events, streams, and the index

Riemann is an event processing engine. There are three concepts we need to understand if we're going to make use of Riemann: events, streams, and the index. Let's start by looking at events.

Events

The event is the base construct of Riemann. Events flow into Riemann and can be processed, counted, collected, manipulated, or exported to other systems. A Riemann event is a [struct](#) that Riemann treats as an immutable map.

Here's an example of a Riemann event.

Listing 3.22: Example Riemann event

```
{:host riemann, :service riemann streams rate, :state ok,
:description nil, :metric 0.0, :tags [riemann],
:time 355740372471/250, :ttl 20}
```

Each event generally contains the following fields.

Field	Description
host	A hostname, e.g. <code>riemann</code> .
service	The service, e.g. <code>riemann streams rate</code> .
state	A string describing state, e.g. <code>ok</code> , <code>warning</code> , <code>critical</code> .
time	The time of the event in Unix epoch seconds.
description	Freeform description of the event.
tags	Freeform list of tags.
metric	A number associated with this event, e.g. the number of reqs/sec.
ttl	A floating-point time in seconds, for which this event is valid.

A Riemann event can also be supplemented with optional custom fields. You can configure additional fields when you create the event or you can add additional fields to the event as it is being processed—for example, you could add a field containing a summary or derived metrics to an event.

Inside our Riemann configuration we'll generally refer to an event field using keywords. Remember that keywords are often used to identify the key in a key/value pair in a map and that our event is an immutable map. We identify keywords by their `:` prefix. So, the host field would be referenced as `:host`.

The next layer above events are streams.

Streams

Each arriving event is added to one or more streams. You define streams in the `(streams` section of your Riemann configuration. Streams are functions you can pass events to for aggregation, modification, or escalation. Streams can also have child streams that they can pass events to. This allows for filtering or partitioning of the event stream, such as by only selecting events from specific hosts or services.

Listing 3.23: Child streams example

```
(streams
  (childstream
    (childstream)))
```

You can think of streams like plumbing in the real world. Events enter the plumbing system, flow through pipes and tunnels, collect in tanks and dams, and are filtered by grates and drains.

You can have as many streams as you like and Riemann provides a powerful stream processing language that allows you to select the events relevant to a specific stream. For example, you could select events from a specific host or service that meets some other criteria.

Like your plumbing though, streams are designed for events to flow through them and for limited or no state to be retained. For many purposes, however, we do need to retain some state. To manage this state Riemann has the index.

The Riemann index

The index is a table of the current state of all services being tracked by Riemann. You tell Riemann to specifically index events that you wish to track. Riemann creates a new service for each indexed event by mapping its `:host` and `:service` fields. The index then retains the most recent event for that service. You can think about the index as Riemann's worldview and source of truth for state. You can query the index from streams or even from external services.

We saw in our event definition above that each event can contain a TTL or Time-to-Live field. This field measures the amount of time for which an event is valid. Events in the index longer than their TTL are expired and deleted. For each expiration a new event is created for the indexed service with its `:state` field set to `expired`. The new event is then injected back into the stream.

Let's take a closer look at this. Here's an example event:

Listing 3.24: Example Apache Riemann event

```
{:host www, :service apache connections, :state nil, :description  
nil, :metric 100.0, :tags [www], :time 466741572492, :ttl 20}
```

It's from a host called `www` and is for a service called `apache connections`. It has a TTL of 20 seconds. If we index this event then Riemann will create a service by mapping `www` and `apache connections`. If events keep coming into Riemann then the index will track the latest event from this service. If the events stop flowing then sometime after 20 seconds have passed the event will be expired in the index. A new event will be generated for this service with a `:state` of `expired`, like so:

Listing 3.25: Example expired Apache Riemann event

```
{:host www, :service apache connections, :state expired, :  
description nil, :metric 100.0, :time 466741573456, :ttl 20}
```

This event will then be injected back into streams where we can make use of it. This behavior is going to be pretty useful to us as we use Riemann for monitoring our applications and services. Instead of polling or checking for failed services, we'll monitor for services whose events have expired.

Configuring events, streams, and the index

Now that we know a bit more about Riemann let's take another look at the second half of our default configuration which contains our streams and an index configuration.

Listing 3.26: More of our default riemann.config configuration file

```
(periodically-expire 5)  
  
(let [index (index)]  
  (streams  
    (default :ttl 60  
      index  
  
      #(info %))))
```

The first function in our configuration, **(periodically-expire 5)**, removes any events that have expired from the index. It's an event reaper that runs every five

seconds and acts on any events with expired TTL by deleting them from the index. For every event reaped a new event is created for that indexed host and service. That event is then put onto the stream with a `:state` of `expired`.

By default, Riemann copies the `:host` and `:service` fields to the expired event. You can control what other fields from events are also copied onto expired events by passing the `:keep-keys` option to the `periodically-expire` function. For example, we'd like to add the `:tags` field to expired events.

Listing 3.27: Copying more keys into expired events

```
(periodically-expire 5 {:keep-keys [:host :service :tags]})
```

This will copy the `:host`, `:service`, and `:tags` fields from the event being expired into the new event being injected into the stream.

The `let` expression that follows our `(periodically-expire)` function defines a new symbol called `index`. This symbol has a value of `index`, which is the function that sends events to Riemann's index. We're going to use this symbol to tell Riemann when to index specific events.

The `let` expression also wraps our streams. Next inside our `let` expression (note the bracket is not closed yet) we've specified that what follows are streams. We've done this using the `streams` function. Each stream is a Clojure function that takes an event. The `streams` function means "here is a list of functions that you should call when new events arrive".

The first thing we've done inside our streams is to set a default TTL for our events of 60 seconds. We've done this using the `default` function. The `default` function takes a field from an event and allows you to specify a default value for that field.

Listing 3.28: Using the Riemann default function

```
(default :field default_value)
```

This TTL will determine how long an event will be valid within the index. In this case, after 60 seconds, events which do not already have a TTL will be expired.

Next the configuration calls our `index` symbol. This means all incoming events will be automatically added to Riemann's index.

The last item in our configuration prints any events to our log file.

Listing 3.29: Logging to the Riemann log file

```
#(info %)
```

The `info` function writes our event and some logging data to the `/var/log/riemann/riemann.log` log file and to `STDOUT`. You'll see events like the following in the log file when Riemann is running:

Listing 3.30: A Riemann log event

```
INFO [2015-03-22 21:40:37,287] Thread-5 - riemann.config - #
  riemann.codec.Event{:host riemann, :service riemann streams
    rate, :state nil, :description nil, :metric 7.739079374131467,
    :tags [riemann], :time 1427060437213/1000, :ttl 20}
```

Also available is the `#{warn %}` function that emits events with a level of `WARN` rather than `INFO`.

You can adjust this logging output to log additional information, such as by adding a prefix for debugging.

Listing 3.31: Adding a prefix to Riemann logs entries

```
#(info "prefix" %)
```

This will prefix any log entries with the word `prefix`. You can also limit the log output to specific fields in an event, for example:

Listing 3.32: Limiting Riemann log entries

```
#(info (:host %) (:service %))
```

This will only send the contents of the `:host` and `:service` fields to the log file.

Listing 3.33: A filtered Riemann log event

```
INFO [2015-03-22 21:55:35,172] Thread-6 - riemann.config -
riemann riemann streams rate
```

If you just want to print to `STDOUT` because, for example, you're running Riemann interactively to test something, you can use the `prn` function.

Listing 3.34: The Riemann prn function

```
; Print event to stdout  
prn  
  
; Print "output", then the event  
#(prn "Output: " %)
```

Now we need to reload Riemann to enable our new configuration.

Listing 3.35: Reloading Riemann to enable our new configuration

```
$ sudo service riemann reload
```

You should start to see events in your `/var/log/riemann/riemann.log` file. These events are Riemann's own internal status reporting.

Listing 3.36: Riemann internal events

```
INFO [2015-02-03 06:04:50,031] Thread-7 - riemann.config - #
  riemann.codec.Event{:host riemann, :service riemann streams
    rate, :state nil, :description nil, :metric 0.0, :tags [riemann
  ], :time 355740372471/250, :ttl 20}
INFO [2015-02-03 06:04:50,034] Thread-7 - riemann.config - #
  riemann.codec.Event{:host riemann, :service riemann streams
    latency 0.0, :state nil, :description nil, :metric nil, :tags [
      riemann], :time 355740372471/250, :ttl 20}
INFO [2015-02-03 06:04:50,035] Thread-7 - riemann.config - #
  riemann.codec.Event{:host riemann, :service riemann streams
    latency 0.5, :state nil, :description nil, :metric nil, :tags [
      riemann], :time 355740372471/250, :ttl 20}
.
.
```

They include the rates and latency of your streams and TCP, UDP, and WebSockets servers. We can use this data to report on the state of Riemann.

Sending our first event to Riemann

Let's test that Riemann is receiving events from external sources too. You can send data to Riemann in a number of ways, including via its own set of tools and a variety of client native language bindings.

NOTE You can find a full list of the clients [on the Riemann website](#).

The set of tools are written in Ruby and available via the `riemann-tools` gem we installed earlier. Each tool ships as a separate binary, and you can see a list of the available tools [in the `riemann-tools` repository on GitHub](#). They include basic health checks, web services like Apache and Nginx, Cloud services like AWS, and more.

TIP We can also query and send events to Riemann using the Riemann C client. You can install it on Ubuntu and Red Hat via the `riemann-c-client` package and use it via the `riemann-client` binary.

The easiest of these tools to test with is `riemann-health`. It sends CPU, memory, and load statistics to Riemann. Open up a new terminal and launch it now on a Riemann server.

Listing 3.37: The `riemann-health` command

```
$ riemann-health
```

You can either run it locally on the same host where you're running Riemann, or you can run it on a remote server and point it at a Riemann server using the `--host` flag.

Listing 3.38: The `riemann-health --host` option

```
$ riemann-health --host riemann.example.com
```

The `riemann-health` command will now start emitting events into Riemann's TCP

server on port **5555**, or you can specify an alternate port with the **--port** flag. In our current configuration we're sending all incoming events into Riemann's log file via the **#{info %}** function. Let's look at our incoming data in the Riemann log file: **/var/log/riemann/riemann.log**.

Listing 3.39: Our incoming Riemann data

```
$ tail -f /var/log/riemann/riemann.log
INFO [2015-12-23 17:23:47,050] pool-1-thread-16 - riemann.config
- #riemann.codec.Event{:host riemann.example.com, :service
disk /, :state ok, :description 11% used, :metric 0.11, :tags
nil, :time 1419373427, :ttl 60.0}
INFO [2015-12-23 17:23:47,055] pool-1-thread-18 - riemann.config
- #riemann.codec.Event{:host riemann.example.com, :service
load, :state ok, :description 1-minute load average/core is
0.11, :metric 0.11, :tags nil, :time 1419373427, :ttl 60.0}
. . .
```

Here we see two events, one for disk space and another for load. Let's look at the event itself a bit more closely.

Listing 3.40: A Riemann-health disk event

```
{:host riemann.example.com, :service disk /, :state ok,
:description 11% used, :metric 0.11, :tags nil,
:time 1419373427, :ttl 10.0}
```

Here we have an event from the host **riemann.example.com** from the service **disk /**. This measures the disk space used on the root or **/** filesystem. It has a state

of `ok`, a description that tells us the percentage of disk space consumed, and an associated metric with that percentage as a float, as well as the time the event was recorded. Lastly, the event has a Time-to-Live or TTL that controls for how long the event is valid.

Now we know our Riemann server is working and can receive events.

NOTE We're going to collect and monitor a lot more of these host-level events and metrics in Chapters 5 and 6.

Creating our first Riemann monitoring check

Now we'll get to the core of why we're here. We're going to build a monitoring check using one of our `riemann-health` events. Let's open up our `/etc/riemann/riemann.config` configuration file and add our first check.

Listing 3.41: Our first monitoring check

```
(let [index (index)]
  (streams
    (default :ttl 60
      index

      ;#(info %)
      (where (and (service "disk /") (> metric 0.10))
        #(info "Disk space on / is over 10%!" %)))
```

Here we've added some new Clojure code for our first check. You'll note we've

used a ; to comment out the `#(info %)` function. This stops Riemann from emitting every event to the log file. Next we've specified a new stream called `where`. The `where` stream selects events based on criteria then passes them to a child stream where you can do something else with the event. In our `where` stream we are matching on two criteria, combined with a Boolean `and` statement:

- The `:service` field of the event is `disk /`.
- The `:metric` field of the event is greater than `0.10` or 10%.

If these two criteria match then the matched event is sent to the `#(info %)` function, so it can then be sent to the log file. We've also added a prefix to our log message detailing why the event has been matched and outputted.

Let's look at what a matched and outputted event would look like in our `/var/log/riemann/riemann.log` log file.

Listing 3.42: Our prefixed warning event

```
Disk space on / is over 10%! #riemann.codec.Event{:host riemann,
  :service disk /, :state ok, :description 24% used, :metric
  0.24, :tags nil, :time 1449184188, :ttl 60.0}
```

We've stripped off some initial boilerplate from the event but you can see that our event has disk space usage of 24% and hence been filtered by our `where` stream. It's been prefixed with our helpful explanatory message and then printed to the log file.

Congratulations—you've just built a Riemann monitoring check! Yes, this check is basic, doesn't go anywhere terribly useful, and isn't telling us anything critical about our environment. But it starts to show us what we can do with Riemann. We'll see many more complex checks in later chapters as we build out our monitoring environment. Now let's explore some more ways we can filter events.

An interlude into Riemann filtering

Before we go on, and because filtering is going to be key to how we manage events inside Riemann, let's look at some more examples of how to use filtering streams.

In our first example we're going to match events using regular expressions. It's a common use case and a good starting point.

Listing 3.43: Using the where stream with a regular expression

```
(where (service #"^nginx"))
```

In Clojure, regular expressions are prefixed with `#` and wrapped in double quotes. Here the `where` stream matches all events where the `:service` field starts with `nginx`.

We can also use Boolean operators to match events.

Listing 3.44: Using the where stream with booleans

```
(where (and (tagged "www") (state "ok"))))
```

In this case our `where` stream matches events that are tagged with `www` and have a `:state` of `ok`. This example also combines the `where` stream and a new stream called `tagged`.

The `tagged` stream selects all events where the `:tags` field contains `www`. The `tagged` stream is shorthand for the `tagged-all` function which matches all tags specified. There's another function called `tagged-any` that allows you to match any one of a series of tags.

Listing 3.45: The tagged-any stream

```
(tagged-any ["www" "app1"]  
#(info %))
```

Here we've used the **tagged-any** stream to match any event which has either the **www** or the **app1** tag, and then send the event to be logged.

We can also combine tags, booleans, and regular expressions to do complex matching.

Listing 3.46: Using the where stream for complex matches

```
(where (and (tagged "www") (state "ok") (service #^apache*)))
```

Here we've matched events that are tagged with **www**, have a state of **ok**, and are from services starting with **apache**.

Up until now we've matched events using “standard” fields like **:service** and **:state**. You'll note we've referred to these fields by their names, minus the **:**. This is some useful syntactic sugar provided by the **where** filtering stream. Any references to the “standard” fields like **:service**, **:host**, **:tags**, **:metric**, **:description**, **:time**, and **:ttl** can use these name shortcuts. If, however, you want to refer to an optional field then you need to refer to it like so:

Listing 3.47: Referring to an optional example field in Riemann

```
(:field_name event)
```

We prefix the field name with a colon and tell Riemann it belongs to `event`, which is Riemann's shorthand for the event being processed. For example, with the field named `type` it would look like:

Listing 3.48: Referring to an optional field in Riemann

```
(:type event)
```

Let's see this in action.

Listing 3.49: The optional type field in Riemann

```
(where (and (tagged "www") (= (:type event) "load")))
```

Here the `where` stream matches all events tagged with `www` and the value of the `:type` field is `load`. You can see that the second `:type` field match does it using a combination of an operator, the name of the field, and a value, constructed like so:

Listing 3.50: Referring to an optional field in Riemann

```
(operator (:field_name event) value)
```

Using this operator-field-value syntax we can also do math operations to match events.

Listing 3.51: Using the where stream with math

```
(where (and (tagged "www") (>= (* metric 10) 5)))
```

Here we've matched all events with the `www` tag and those in which the value of the `:metric` field multiplied by `10` is greater than or equal to `5`. You can use the normal collection of operators: greater than, equal to, less than, and so on in these statements.

We also use a similar syntax to do a range query.

Listing 3.52: Using the where stream for a range query

```
(where (and (tagged "www") (< 5 metric 10)))
```

Here we've matched all events tagged with `www` and those where the `:metric` field has a value between `5` and `10`.

TIP You can learn more about event filtering on the [Riemann website](#).

Connecting Riemann servers

Now that we know our individual Riemann servers are working, let's hook them together. In our architecture we have two upstream Riemann servers: `riemann.example.com` and `riemannb.example.com`. We also have a downstream Riemann server `riemannmc.example.com`. We're assuming you've used the steps above to

install Riemann onto each of these hosts.

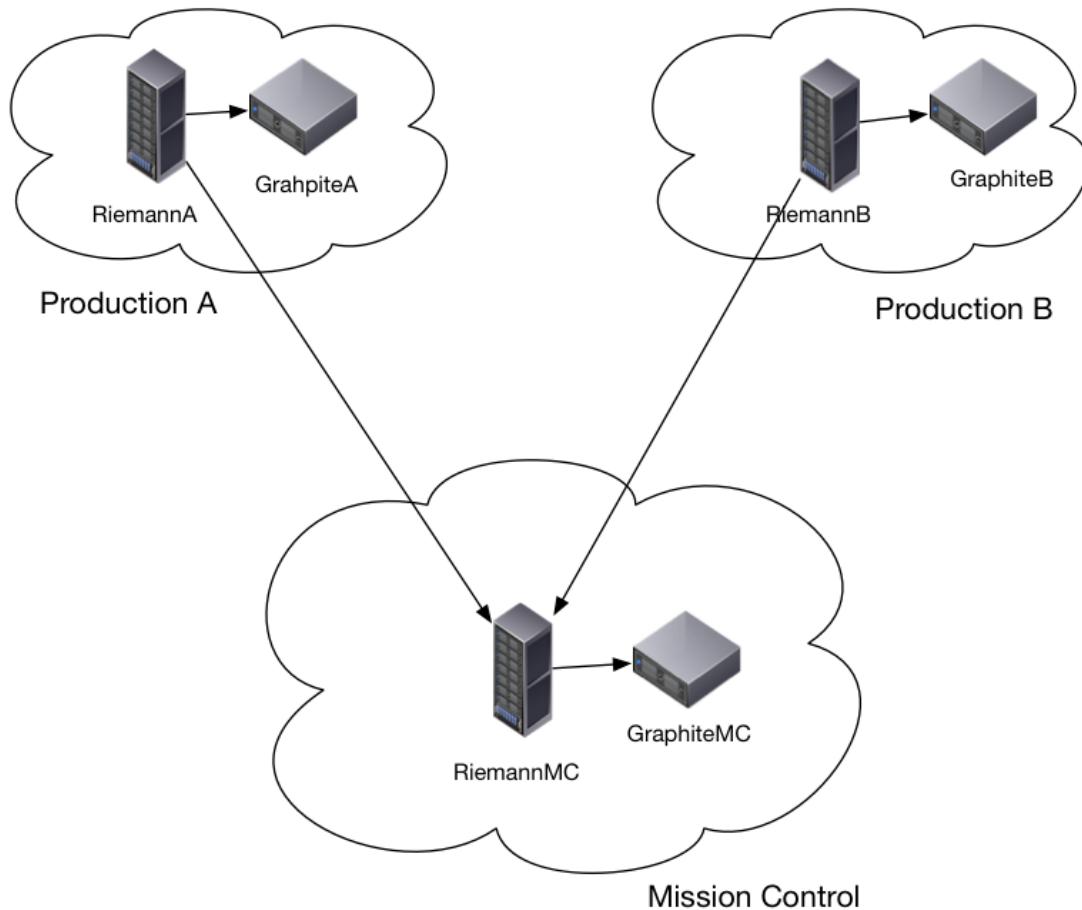


Figure 3.3: Connecting Riemann servers

In our architecture we want to send escalations and status updates for our production environments downstream to our Mission Control environment, riemannmc.example.com. But initially we're just going to send information about Riemann itself. This will allow us to detect if our upstream Riemann servers are operational. We're going to connect the upstream servers to the downstream server and then test our connection.

Configuring the upstream Riemann servers

First, we need to define our downstream Riemann server to our upstream servers. We do this by first specifying a new stream called `async-queue!`. The `async-queue!` stream creates an asynchronous `thread pool` queue. That queue accepts events and passes those events to child streams via that thread pool asynchronously.

So why pass the events asynchronously? Well Riemann is fast but anytime a stream connects to an external service there is a risk that your processing will be blocked waiting for that external service. With an asynchronous queue we tell Riemann to queue events and return to the mainline without blocking. In this case we're then going to use Riemann's own TCP client as one of those asynchronous streams to send events to our downstream server.

WARNING Asynchronous streams look like an easy fix for connecting to potentially blocking services but they come with some caveats. You should carefully read the [documentation](#) before using them wildly.

Now let's look at an updated configuration file.

Listing 3.53: Updated Riemann configuration

```
(logging/init {:file "/var/log/riemann/riemann.log"})

(require 'riemann.client)

. . .

(let [index (index)
      downstream (batch 100 1/10
                        (async-queue! :agg { :queue-size     1e3
                                              :core-pool-size 4
                                              :max-pool-size  32})
                        (forward
                          (riemann.client/tcp-client :host "riemannmc")))]

  (streams
    (default :ttl 60
              index

              #(info %)

              (where (service #^riemann.*")
                     downstream))))
```

You can see we've added some new configuration that adds the Riemann client. The `require` function is much like Ruby's `require` method. It loads additional code we might need to do certain actions.

Listing 3.54: Requiring the Riemann client

```
(require 'riemann.client)
```

Here our `require` loads the Riemann TCP client. We're going to use this client to send events downstream.

NOTE We're going to talk about `require` in a lot more detail a little later in this chapter.

Next, we've added another binding to our `let` expression (remember, a binding is a symbol-value pair). This configures the destination and process for sending events.

Listing 3.55: Added downstream binding to Riemann

```
(let [index (index)
      downstream (batch 100 1/10
                      (async-queue! :agg { :queue-size     1e3
                                             :core-pool-size 4
                                             :max-pool-size 32})
                      (forward
                        (riemann.client/tcp-client :host "riemannmc")))]
  . . .
  )
```

Our binding defines a symbol called `downstream`. The value of this symbol is a series of streams. Our first stream is called `batch`. This batches up events to send. Each batch is sent when 100 events or 1/10th of a second has passed. The `batch` stream passes the events into our `async-queue!`, which we've called `:agg` (for aggregation).

We've defined a `queue-size` for our `async-queue!` in exponential notation, `1e3` or 1000. We set `core-pool-size` (the number of threads in the pool) to 4 and `max-pool-size` (the maximum number of threads in the pool) to 32. This should generally work for most scenarios.

NOTE You can read a bit more about Java-based ThreadPooling [here](#). This provides some useful information about the interaction of queue and pool sizing.

Our queue takes the incoming event batches and passes them to another child stream, `forward`. The `forward` stream sends events through a Riemann client—here the TCP client—to our `riemannmc` host.

Listing 3.56: Riemann client forwarding configuration

```
(riemann.client/tcp-client :host "riemannmc")
```

NOTE This assumes you've configured DNS, added the various Riemann servers to `/etc/hosts`, or provided some other way for Riemann to resolve the `riemannmc` hostname.

This is a complex configuration so let's walk through the whole process to make sure it's clear.

1. We define the `downstream` symbol. When that symbol is referenced events are passed into it.
2. Events first go into the `batch` stream. Every 100 events or 1/10th of a second, events are batched and sent on.
3. The batched events are passed to the `async-queue!` stream.
4. The `async-queue!` stream passes the events to the `forward` stream which sends them to the `riemannmc` server.

Lastly, we've added a `where` stream to select the events we want to send.

Listing 3.57: The where filtering stream for forwards

```
(where (service #"^riemann.*")  
      downstream)
```

As we discovered earlier, the `where` filtering stream selects events based on specific criteria—for example, from a particular host or service—via a regular expression or from the result of executing a function of some kind.

Here our `where` stream selects any events that match the regular expression `^ riemann.*`. This is any events whose `:service` field starts with `riemann..`. These events are then passed to the `downstream` symbol which forwards them to the `riemannmc` server.

We then add the configuration we created to both the `riemann` and `riemannb` servers.

Configuring the downstream Riemann server

Now let's look at the configuration on our downstream `riemannmc` server.

Listing 3.58: The downstream `riemannmc` server

```
    . . .

(let [index (index)]
  ; Inbound events will be passed to these streams:
  (streams
    (default :ttl 60
      ; Index all events immediately.
      index

      #(info %))))
```

You can see it's basically the Riemann configuration we've just created except that it lacks our `downstream` sending configuration.

NOTE We've included all example configuration and code in the book [on GitHub](#).

Enabling the send of our Riemann events downstream

We now restart Riemann on all our servers to enable the sending of our events downstream.

Listing 3.59: Restarting Riemann to enable forwarding

```
riemann$ sudo service riemann restart  
riemannb$ sudo service riemann restart  
riemannmc$ sudo service riemann restart
```

We should now see some new events for our queue in the `/var/log/riemann/riemann.log` log file on our upstream servers.

Listing 3.60: Riemann agg events on riemannna or riemannb

```
INFO [2015-02-03 15:29:10,911] Thread-7 - riemann.config - #  
    riemann.codec.Event{:host riemannna, :service riemann executor  
        agg accepted rate, :state ok, :description nil, :metric  
        250/2507, :tags nil, :time 711497675449/500, :ttl 20}  
INFO [2015-02-03 15:29:10,911] Thread-7 - riemann.config - #  
    riemann.codec.Event{:host riemannb, :service riemann executor  
        agg completed rate, :state ok, :description nil, :metric  
        250/2507, :tags nil, :time 711497675449/500, :ttl 20}  
INFO [2015-02-03 15:29:10,911] Thread-7 - riemann.config - #  
    riemann.codec.Event{:host riemannna, :service riemann executor  
        agg rejected rate, :state ok, :description nil, :metric ON, :  
        tags nil, :time 711497675449/500, :ttl 20}  
    . . .
```

These are useful to allow us to track the state, performance, and status of our forwarding.

We'll also start to see events on our downstream server: `riemannmc`.

You'll note we don't have to change anything on the `riemannmc` server. It's already set up to receive events. If we look in the `/var/log/riemann/riemann.log` log file we'll find events from `riemannmc` and `riemannnb` as well as the local events from `riemannmc`.

Listing 3.61: Combined events from upstream and downstream

```
INFO [2015-02-03 08:35:58,507] Thread-6 - riemann.config - #
  riemann.codec.Event{:host riemannMC, :service riemann server ws
    0.0.0.0:5556 in latency 0.999, :state ok, :description nil,
    :metric nil, :tags nil, :time 1422970558489/1000, :ttl 20}
  .
  .
  .
INFO [2015-02-03 08:36:01,495] defaultEventExecutorGroup-2-1 -
  riemann.config - #riemann.codec.Event{:host riemannnb.
  lovedthanlost.net, :service riemann streams rate, :state nil,
  :description nil, :metric 3.9884721385215447, :tags [riemann], :
  time 1422970561, :ttl 20.0}
  .
  .
  .
INFO [2015-02-03 08:36:14,314] defaultEventExecutorGroup-2-1 -
  riemann.config - #riemann.codec.Event{:host riemannna, :service
  riemann streams latency 0.5, :state nil, :description nil,
  :metric 0.222681, :tags [riemann], :time 1422970574, :ttl 20.0}
  .
  .
```

Here we see events from all three servers indicating that they are successfully connected.

NOTE We've included this example configuration with all the required files in the book's code [on GitHub](#).

Alerting on the upstream Riemann servers

So now the downstream `riemannmc` Riemann server knows about the upstream `riemannna` and `riemannnb` servers. But we also want to know when something goes wrong with those upstream servers. To do that we're going to take advantage of Riemann's index.

Remember the index is a table of the current state of all services being tracked by Riemann. Each event you tell Riemann to index is added as a service mapped by its `:host` and `:service` fields. The index retains the most recent event for that service. Each indexed event has a Time-to-Live or TTL. The TTL can be set by the event's `:ttl` field, or if no TTL is present then a default can be specified. Our default TTL is 60 seconds set using the `default` function in our Riemann configuration.

If a service fails it stops submitting events to Riemann. In our Riemann configuration we have a `periodically-expire` function that runs every 5 seconds. This is the event reaper for the index. It checks the TTL of events in the index and expires and deletes events from the index if their TTL is expired. When an indexed event is expired a new event is created for the indexed service with a `:state` of `expired` and sent back to the stream.

We then monitor the stream for events with a `:state` of `expired`, which means the service isn't reporting, and notify on those.

Let's create some configuration on the `riemannmc` server to catch expired events from the `riemannna` and `riemannnb` servers. This means that if these servers stop sending events, we'll get notified.

Take a look at our `riemannmc` Riemann configuration now. Our `/etc/riemann/riemann.config` file looks like:

Listing 3.62: The downstream riemannmc server configuration

```
(logging/init {:file "/var/log/riemann/riemann.log"})

(let [host "0.0.0.0"]
  (repl-server {:host "127.0.0.1"})
  (tcp-server {:host host})
  (udp-server {:host host})
  (ws-server {:host host}))

(periodically-expire 10 {:keep-keys [:host :service :tags, :state
, :description, :metric]})

(let [index (index)]
  ; Inbound events will be passed to these streams:
  (streams
    (default :ttl 60
      ; Index all events immediately.
      index

      #(info %))))
```

It's similar to our upstream Riemann servers, minus the configuration that sends events downstream.

We're going to add some configuration to:

1. Identify expired events.
2. Select only the Riemann-specific events.
3. Email a notification on these Riemann-specific expired events.

Let's look at this additional configuration. First we're going to configure a notification mechanism. We're going to start with something simple: email. If we detect an expired Riemann event, we'll send an email notification.

To do this we need to configure the `mailer` plugin. The `mailer` plugin allows us to send email from Riemann. We're going to configure the `mailer` plugin in a namespace. Namespaces are a way of organizing code and functions. You can consider namespaces a more advanced method of organizing and including streams into our Riemann configuration.

Riemann uses Clojure's built-in `namespacing` to do this. In Clojure it's generally recommended you use a carefully defined namespace that won't overlap with anything else. In most cases this is a format like:

Listing 3.63: Clojure namespace format

```
[organization].[library|app].[group-of-functions]
```

In our case we're going to use `examplecom` as our organization. You might use `mycorpname` or a department name or something similar. We're then going to call our library `etc` because that's broadly its function: useful functions we're going to regularly use in our Riemann configuration. Finally, we're going to call the group of function: `email`. Literally `examplecom/etc/email` or Example.com Etc Email.

Our namespace also shapes where we store our code on our Riemann server. Riemann expects to find the code in a directory structure matching the namespace, underneath the `/etc/riemann` directory. So let's start by creating that directory structure.

Listing 3.64: Creating the examplecom/etc namespace path

```
$ sudo mkdir -p /etc/riemann/examplecom/etc
```

Let's then create a file to hold our Riemann code and functions.

Listing 3.65: Creating the email.clj file

```
$ sudo touch /etc/riemann/examplecom/etc/email.clj
```

We can see our directory structure and file follows the namespace:

`examplecom/etc/email`

Our Riemann code and functions will go inside the `email.clj` file.

NOTE `.clj` is the extension for a Clojure code file.

Let's look at the code in this file to get started.

Listing 3.66: Requiring the Riemann functions

```
(ns examplecom/etc/email
  (:require [riemann.email :refer :all]))

(def email (mailer {:from "riemann@example.com"}))
```

Firstly, we declare a name using the `ns` function. The `ns` function creates a new namespace. We've called our namespace: `examplecom.etc.email`. This name is how we'll reference our namespace in our Riemann configuration.

TIP Namespaces are the standard Clojure of organizing code, libraries and functions. You can read more about namespacing in the [Riemann HOWTO](#).

Next we've specified an argument, `:require`, to be passed into our namespace. The `:require` statement is closely related to the `require` function we used earlier to include Riemann's TCP client. It performs the same function inside a namespace. The `:require` argument here includes the `riemann.email` library of functions. The `:require` ensures that the namespace to be required exists, is ready to be used and evaluates the corresponding namespace. By evaluating the namespace, it also defines and makes available any functions in that namespace.

Inside our `examplecom.etc.email` namespace we can now refer to functions inside the `riemann.email` namespace, like so:

`riemann.email/mail`

Referring to functions with the fully-qualified namespace is a little unwieldy though. We've added an argument to our `:require` directive called `:refer`. The `:refer` function means you don't have to use fully qualified names to reference the functions inside a namespace. You can refer one, many or all functions in a namespace. Here we've used the `:all` option to refer all functions inside `riemann.email`.

Alternatively, if you only want to use one or more functions from that namespace you can specify only those.

Listing 3.67: Referring functions

```
(ns examplecom/etc/email
  (:require [riemann/email :refer [mailer]]))
```

```
  .  .  .
```

This would only refer the `mailer` function from the `riemann.email` namespace.

We can now reference a function inside our namespace without needing to prefix it with: `riemann.email`.

NOTE Be careful with this. If you define a symbol with the same name inside two namespaces and refer both of them then you will get a conflict and Riemann will fail to start. You cannot have `examplecom/etc/email/foo` and `examplecom/etc/fish/foo` defined and referred.

Now let's look at our `mailer` configuration in this file.

Listing 3.68: Configuring email notifications in Riemann

```
(def email (mailer {:from "riemann@example.com"}))
```

You can see we've used the `def` statement. As we learned in Appendix A, the `def` statement declares a symbol and a `var`.

In our case we're giving the `email` symbol a value of `mailer`. This tells Riemann that anywhere we specify `email` it should call the `mailer` function from the

`riemann.email` namespace. As we referred to `:all` functions in this namespace we don't have to prefix `mailer` with `riemann.email`.

The `mailer` function sends email. We've also specified one option for the `mailer` function: `:from`, which controls the source email address for any emails from Riemann.

The `mailer` function uses a standard Clojure email library called `Postal` under the covers to send email. By default it uses the local `sendmail` binary but it can also be configured to use an SMTP server.

TIP A quick way to add the `sendmail` binary and set up local mail sending is to install the `mailutils` package on Ubuntu, or the `mailx` package on Red Hat and related distributions.

Listing 3.69: Configuring an SMTP server for Postal

```
(def email (mailer {:host "smtp.example.com"
                     :user "james"
                     :pass "password"
                     :from "riemann@example.com"}))
```

TIP We should add this `email.clj` file and its configuration to all our Riemann hosts. It's going to be useful in future chapters to send notifications.

Now we've got a way to notify on our event, let's add that capability to our Rie-

mann configuration. To use our new `email` function we need to tell Riemann about it in our `riemann.config` file. To do this we again use the `require` function but slightly differently this time.

Listing 3.70: Adding our email function to Riemann

```
(logging/init {:file "/var/log/riemann/riemann.log"})  
  
(require 'riemann.client)  
(require '[examplecom.etc.email :refer :all])  
  
....
```

We've added a new `require` to our `riemann.config` file below our `riemann.client`. It includes the `examplecom.etc.email` namespace and uses the `:refer :all` directive to refer all functions in that namespace.

On startup, Riemann automatically searches for our namespace underneath the `/etc/riemann` directory, evaluates the namespace and the functions in it. In our case this will cause Riemann to search for a directory:

`/etc/riemann/examplecom/etc/`

And then load the file at:

`/etc/riemann/examplecom/etc/email.clj`

The `:refer` then allows us to reference the `email` function inside our Riemann configuration, without needing to prefix it with the `examplecom.etc.email` namespace.

We can now specify a stream to grab the relevant events.

Listing 3.71: Our expired Riemann event filter stream

```
(expired
  (where (service #"riemann.*")
    (email "james@example.com")))
```

Here we've used the `expired` filtering stream. The `expired` stream matches events expired from the index. You can think about it like a `where` stream with the match on the `:state` field of `expired` preconfigured.

We've used a `where` stream to do a regular expression match for any services that start with `riemann..`. Any events matched by the `where` stream will be passed to the `email` var and mailed via the `mailer` function to `james@example.com`.

To activate this new `email` function we now need to reload or restart Riemann. This will load, evaluate and refer the `examplecom.etc.email` namespace.

So if we were to stop Riemann on the `riemannna` host now, after the TTL had expired, then the Riemann index on the `riemannmc` host would generate some events like this:

Listing 3.72: The riemannna expired streams event

```
{:ttl 60, :time 713456271631/500, :state expired, :service
riemann streams latency 0.999, :host riemannna, :tags [riemann]}
```

We see the event has `:state` of `expired` and contains the `:service`, `:host`, and `:tags` fields, all of which were copied to the new expired event when the event reaper ran over the index. The event also contains the default TTL of 60 seconds, and the time the event was generated.

This matched event would then be passed to the `email` var, which would trigger an email to be sent. That email will look something like this:

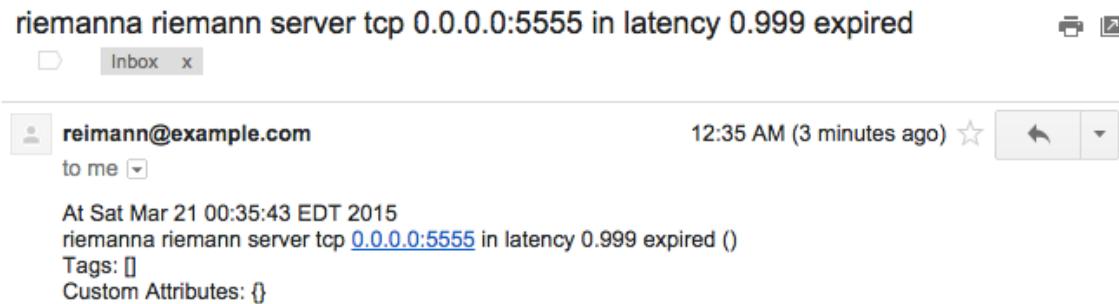


Figure 3.4: Email notification

It's great we're going to be getting emails when the Riemann server fails... But there's a problem. We'll get an email notification for every Riemann metric that is being collected—yes, one for each event. This could mean a lot of emails letting us know that one of our Riemann servers is down.

Throttling Riemann events

To prevent this flood of emails we're going to add a throttle to our notifications. Let's add some throttling to our configuration.

Listing 3.73: Our throttled expired Riemann event filter

```
(expired
  (where (service #"^riemann.*")
    (throttle 1 600
      (email "james@example.com"))))
```

You'll see we've added a new stream called `throttle`. The `throttle` allows a number of events through and then ignores any others for a period of time.

Listing 3.74: The throttle stream

```
(throttle 1 600)
```

This throttle works by allowing one event through and then dropping and ignoring all other events for 600 seconds or 10 minutes. It's a pretty crude mechanism but works for services where we only care about a limited number of notifications.

Rolling up Riemann events

An alternative to `throttle` is the `rollup` stream. The `rollup` stream will allow a few events to pass through, then it will collect and hold the others. It holds those events for a period of time and then sends a summary of them.

Listing 3.75: A rollup of our expired Riemann events

```
(expired
  (where (service #"riemann.*"))
  (rollup 5 3600
    (email "james@example.com")))
```

The `rollup` stream here will only send five emails per 3,600 seconds (one hour). You'll get four emails immediately and then, after an hour, you'll get a final email with a summary of any other events received during that hour period.

WARNING The `rollup` stream accumulates events in memory. The more events there are, the more memory is consumed. You should be careful to ensure

it doesn't exhaust memory on the host holding the rolled-up events.

Alternatives to email notifications

Obviously email is not always an ideal notification mechanism—we all already have folders full of emails—so Riemann provides a wide variety of other mechanisms including [PagerDuty](#) and a variety of chat applications like [Slack](#).

We'll talk a lot more about these mechanisms and about notifications in Chapter 10.

NOTE We've included the `riemannmc` configuration in the book's code [here](#).

Testing your Riemann configuration

One of the advantages of Riemann's configuration being a Clojure program is that you can write tests to confirm that your configuration is working correctly.

To test Riemann events, we tap into points where we'd like to observe events and then write tests that confirm the right behavior is occurring and the right events are being seen or generated. In production Riemann will ignore these taps so you don't experience a performance hit from adding them.

Let's look at an example. In our current `riemannmc` configuration we accept incoming events and index them immediately.

Listing 3.76: Revisting our riemannmc configuration

```
(logging/init {:file "/var/log/riemann/riemann.log"})

(let [host "0.0.0.0"]
  (repl-server {:host "127.0.0.1"})
  (tcp-server {:host host})
  (udp-server {:host host})
  (ws-server {:host host}))

(periodically-expire 10 {:keep-keys [:host :service :tags, :state
, :description, :metric]})

(let [index (index)]
  (streams
    (default :ttl 60
      ; Index all events immediately.
      index)))
```

So our test is going to confirm that events coming into the stream are being indexed. To support this test we need to wrap our `index` stream in a `riemann.test/tap` function. This will allow us to watch the events being indexed.

Listing 3.77: Adding a tap to our riemannmc index

```
(logging/init {:file "/var/log/riemann/riemann.log"})

(let [host "0.0.0.0"]
  (repl-server {:host "127.0.0.1"})
  (tcp-server {:host host})
  (udp-server {:host host})
  (ws-server {:host host}))

(periodically-expire 10 {:keep-keys [:host :service :tags, :state
, :description, :metric]})

(let [index (tap :index (index))]

  (streams
    (default :ttl 60
      ; Index all events immediately.
      index)))
```

You see that we've added a tap around our `index`. We've called that tap `:index`.

We then create one or more tests that sample incoming events to the `index`. We do this by defining a `tests` stream and using the standard Clojure test function: `deftest`.

Listing 3.78: Adding tests to our riemannmc configuration

```
(logging/init {:file "/var/log/riemann/riemann.log"})

(let [host "0.0.0.0"]
  (repl-server {:host "127.0.0.1"})
  (tcp-server {:host host})
  (udp-server {:host host})
  (ws-server {:host host}))

(periodically-expire 10 {:keep-keys [:host :service :tags, :state
, :description, :metric]})

(let [index (tap :index (index))]

  (streams
    (default :ttl 60
      ; Index all events immediately.
      index)))

  (tests
    (deftest index-test
      (is (= (inject! [{:service "test"
                        :time    1}])
            {:index [{:service "test"
                      :time    1
                      :ttl     60}]})}))
```

We see a new set of **tests** streams in our configuration now. We've defined one

test, `index-test`. The test is a simple process:

- Inject an event into the stream.
- Monitor the `:index` tap and see if that event arrives.

The `inject!` function injects an event into the stream. Here we're injecting a single event with a `:service` field set to `test` and a `:time` field of `1`. We're then asking if the `:index` tap sees a like event. We've added the `:ttl` field of `60` because the `default` function sets that when we index an event. Without it our test will fail because the event we're watching for will not match the event we've injected.

We then run our tests using the `riemann` binary. Riemann must be stopped when we run the tests because our whole configuration, including components like interface bindings, is run when the tests are run.

Listing 3.79: Running the Riemann tests

```
$ sudo riemann test riemann.config
INFO [2015-07-15 17:40:01,236] main - riemann.repl - REPL server
{:port 5557, :host 127.0.0.1} online

Testing riemann.config-test

Ran 1 tests containing 1 assertions.
0 failures, 0 errors.
```

We see that our single test, containing a single assertion, has been run and has passed successfully. This means we've injected an event into the stream, watched the tap, and then seen the corresponding event appear in the tap.

TIP You can read more about the testing of your configuration in the [Riemann documentation](#).

Validating Riemann configuration

Riemann configuration looks pretty scary at first. But to make it easier to build your Riemann configuration there is a syntax checker available. To use it you need to download and build it. This assumes you have Java, Git, and Leiningen installed.

Listing 3.80: Download the Riemann syntax checker

```
$ git clone https://github.com/samn/riemann-syntax-check.git
```

Then build the syntax checker as a Jar file.

Listing 3.81: Build the Riemann syntax checker

```
$ cd riemann-syntax-check  
$ lein uberjar  
...
```

You should see some dependencies downloaded and some Jar files created. You can then use one of these Jar files to syntax check a Riemann configuration.

Listing 3.82: Syntax check a Riemann configuration

```
$ cd /etc/riemann/  
$ java -jar riemann-syntax-check-0.2.0-standalone.jar riemann.  
config
```

Here we're syntax checking the `riemann.config` configuration file. Any errors or issues will be reported so you can fix them before reloading or restarting Riemann.

Performance, scaling, and making Riemann highly available

The performance of Riemann is largely memory-bound. The only disk IO it uses is logging. You should ensure any hosts running Riemann have generous allocations of memory.

You can configure how much additional memory goes to the Riemann process by configuring the Java heap size options that are passed when Riemann is launched. To do this we add any required options to the `/etc/default/riemann` file on Ubuntu and `/etc/sysconfig/riemann` file on Red Hat and similar distributions. In both files, uncomment the line starting with `EXTRA_JAVA_OPTS` and add the `-Xms` and `-Xmx` flags with an appropriate amount of additional memory. These flags control the initial and maximum Java heap size. Like so:

Listing 3.83: Configuring additional RAM

```
EXTRA_JAVA_OPTS="-Xms4096m -Xmx4096m"
```

Oracle recommends that you set the initial and maximum heap size to the same value to minimize garbage collections.

NOTE You can read a bit more about the available `-X` options in the [Java documentation](#).

You can also view the JVM tuning options in the [Debian and RPM packages](#) in the [`AGGRESSIVE_OPTS`](#) environment variable for other ideas on how to tune Riemann.

At this time there isn't an out-of-the-box solution for high availability with Riemann. Riemann servers don't come with a highly available, fail-over solution or configuration. This isn't all that dire though. Riemann doesn't maintain much state—it's mostly being used as a routing engine for events and metrics. Losing that capability isn't great, but because of the statelessness of Riemann it's easy to rebuild or add a new Riemann host to replace any unavailable hosts.

Alternatively you could run one or more Riemann hosts in a warm standby and replace missing hosts with new ones relatively quickly. You could also supplement this with a shared proxy like HAProxy and add multiple Riemann servers to your back-end host pool. Then if a failure occurred on one host another would be available.

For scaling it's also not yet possible to automatically scale and distribute Riemann workloads in any sort of cluster or distributed manner. For scaling Riemann it is largely a matter of horizontal scaling by adding Riemann servers, perhaps distributed by application, rack, stack, data center, or site. Again using a solution like a shared proxy to distribute events and metrics to a series of back-end Riemann hosts depending on their source is also an option.

Or you could also create a sharded configuration. This configuration would run two or more Riemann servers in parallel. Every event would be assigned a shard

ID from the Riemann server that received it. They would then be consolidated downstream, filtering out the shard ID and using a merge algorithm for each set of events.

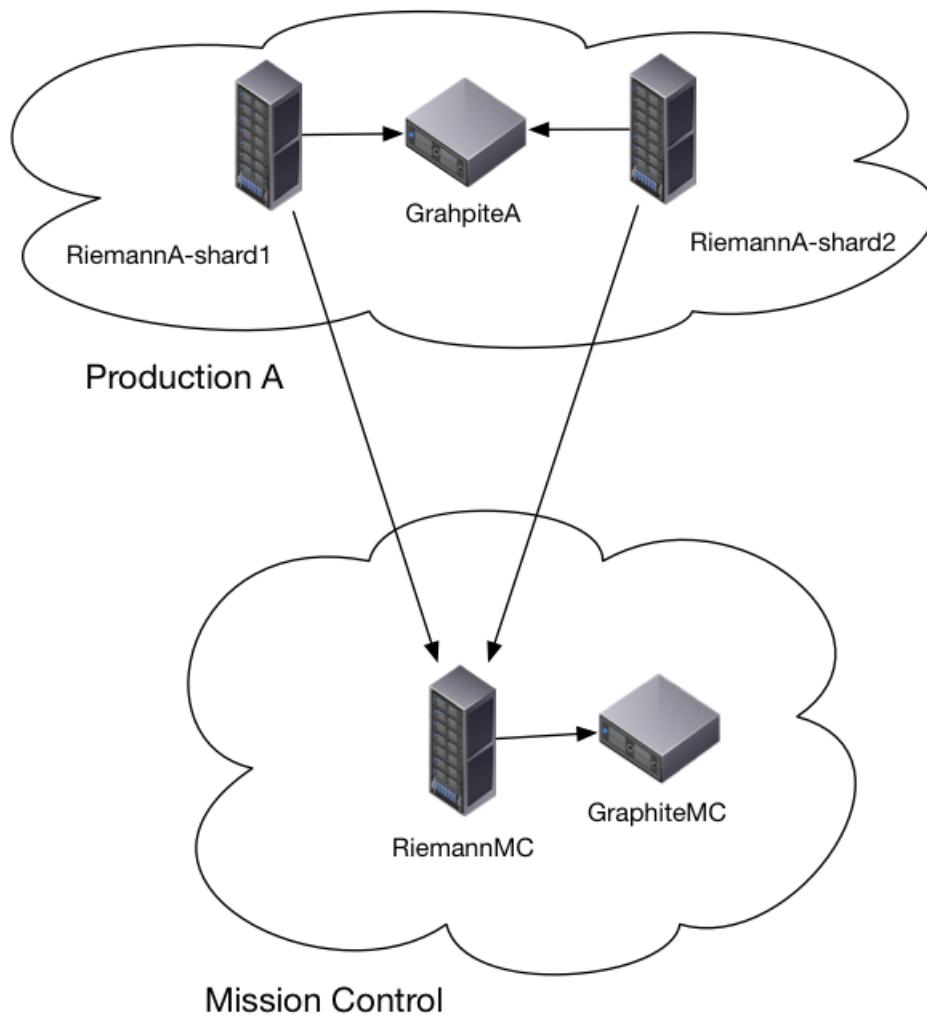


Figure 3.5: Sharded Riemann

This book does not currently cover either high availability or scaling in any depth, but will be updated if better native solutions become available.

TIP There's also [a plugin to enable JMX-based monitoring](#) of Riemann. You can see read more about using JMX to monitor the JVM in Chapters 8 and 12.

Alternatives to Riemann

There are a few event-based or message queue systems that work like Riemann. This is not a definitive list but it's a sampling of the more interesting tools you could use if the choices in the book aren't suitable or to your taste.

- [Apache Samza](#) and [Apache Kafka](#) — A distributed real-time computation system and a publish-subscribe messaging system with a cluster-centric design.
- [Prometheus](#) — Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud.
- [Heron](#) - Heron is realtime analytics platform developed by Twitter.
- [Bosun](#) — An open-source, MIT-licensed, monitoring and alerting system built by Stack Exchange.
- [Anthracite](#) — An event and change logging and management application.
- The [ELK Stack](#) — Elasticsearch, Logstash, Kibana. A powerful logging tool that can also be adapted to handle events and metrics. We'll look at it more closely in Chapter 8.
- [Heka](#) — Another logging tool, this one released by the Mozilla team. Sadly, [no longer maintained](#). There is a potential successor called [Hindsight](#).
- [Godot](#) — An event streamer modeled on Riemann but rewritten in node.js. Instead of Clojure, it's written in Javascript. It's somewhat slower than Riemann but shares many concepts.
- [Ganglia](#) — A monitoring tool with a focus on clusters and grids.
- [Munin](#) — A popular metric and monitoring tool that uses [RRDTool](#).
- [Snap](#) - Open source by Intel, Snap is a telemetry collection and processing tool. It also supports publishing data to Riemann.

Summary

In this chapter we've learned how to install Riemann. We installed Riemann on three hosts: `riemannna` and `riemannnb` in our production data centers, and `riemannmc` in our Mission Control environment. We've also had an introduction to configuring and running it.

We've also connected our upstream Riemann production servers with the downstream Riemann server in Mission Control. We then set up notifications in the downstream Riemann server to detect if there are any issues with the upstream servers.

These Riemann servers will form the basis of our monitoring framework: event-centric routing engines that will allow us to collect, process, and send events and metrics. We'll track the state of our hosts, services, and applications centrally in the Riemann index, notify on any critical or important events (or the absence of those events), and then graph any related metrics.

In the next chapter we're going to continue our monitoring build by installing and configuring the graphing engine Graphite in our environment. We'll then send some initial metrics from Riemann to Graphite.

Chapter 4

Introducing Graphite and Grafana

In the last chapter we installed Riemann and configured Riemann servers in each of our environments. Riemann provides the central destination and routing engine for our events and metrics. In the next layer of our monitoring framework we're going to provide a destination to store our time-series metrics after they've been received by Riemann. We're also going to look at tools to visualize those metrics so they provide us with insight into how our hosts, services, and applications are functioning.

The design for a metrics storage and visualization solution is:

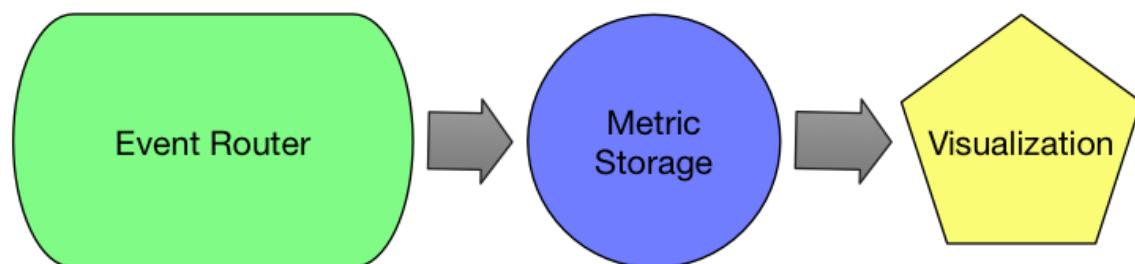


Figure 4.1: Metrics processing and visualization

With this design we want our graph storage and visualization to be able to:

- Receive and process our metric data efficiently.
- Scale as we grow our environment.
- Provide scalable, efficient storage of metrics data.
- Provide a mechanism to elegantly degrade the granularity of metric data, storing at a high granularity for recent metrics and degrading for older metrics.

To provide the storage and visualization capabilities we're going to install and configure a tool called [Graphite](#).

Introducing Graphite

Graphite is an engine that stores time-series data and then can render graphs from that data using an API. Graphite is licensed under the Apache 2.0 license and is written in Python.

Graphite is simple and fast. It may not be the most modern time-series database—it relies on flat files, for example, rather than more modern database-style implementations—but it is well tested and reliable. It comes with a wide collection of functions for manipulating, summarizing, and munging data for visualization. It also has a large and active community of users and a solid group of developers.

To get metrics into Graphite we're going to configure Riemann to send metrics to it. We'll also begin to look at Graphite's graphing and dashboard capabilities using a dashboard called [Grafana](#) so we can display that metric data.

TIP This book provides an introduction to Graphite but it is a hugely powerful tool with lots of capabilities. If you're interested in becoming a Graphite expert we strongly recommend [Jason Dixon's amazing “Monitoring with Graphite” book](#).

Graphite is made up of three components:

Carbon

Carbon is a collection of event-driven daemons that listen on network ports. There are two major daemons we'll be using in the book: `carbon-cache` and `carbon-relay`. The `carbon-cache` daemon listens, receives, and writes time-series data to storage. The `carbon-relay` daemon listens, receives, and forwards time-series data to `carbon-cache` servers.

The two Carbon daemons, `carbon-relay` and `carbon-cache`, allow us to scale and cluster Carbon across individual and multiple hosts. For example, you might have a host behind a load balancer running multiple `carbon-relay` daemons that receives events and then forwards them to another host running multiple `carbon-cache` daemons that ingest and process our data.

In our initial Graphite configuration we're going to run everything on a single host but we'll learn more about scaling Carbon later on.

The Carbon daemons expect a stream of time-series data: essentially a metric, a value, and a timestamp. Metrics can be anything that comes in the form of a measurable output: systems metrics, applications metrics, business metrics. After Carbon receives metrics, it periodically flushes them to a storage database called Whisper.

Whisper

Whisper is a lightweight, flat-file database format for storing time-series data. It is not a daemon or a service; it is a database file format. Carbon writes metric data to the disk in the Whisper format. Each unique metric type is stored in a fixed-size file with a `.wsp` suffix, for example:

Listing 4.1: The Whisper file layout

```
....carbon/whisper/HostA/memory-used.wsp  
....carbon/whisper/HostB/memory-free.wsp  
....carbon/whisper/HostB/memory-used.wsp
```

The size of each database file is determined by the number of data points stored.

As this is data written to disk this is the usual bottleneck for scaling Graphite servers. To counteract this you want fast local disks. For example, many people use hosts with SSD disks to store metrics data. You can also [cluster Graphite servers](#), which we'll talk more about later in this chapter.

Graphite Web, Graphite-API, and Grafana

The last component of Graphite is Graphite Web, a Django-based web UI that can query the Carbon daemon and read Whisper data to return metrics data. It can be used to compose graphs directly, or it can provide a RESTful API that can be used by third-party tools to extract metrics to compose graphs. The API can return data or rendered output like `.png` images.

We're not, however, going to use Graphite Web in our deployment. This is because Graphite Web can be problematic to install and configure. Additionally, the interface is somewhat dated and hard to use.

Instead we're going to use a more modern console called [Grafana](#). Grafana is an open-source metrics dashboard that supports Graphite, InfluxDB, and OpenTSDB. It's Apache 2.0 licensed. When using Graphite, Grafana runs on top of the Graphite Web API. So rather than install the full Graphite Web component we're going to install an [API integration](#) and then the Grafana dashboard on top of that API.

Graphite architecture

Here's the overall architecture: Carbon receives incoming metrics and writes them into the Whisper format on the disk, then the Graphite API makes those metrics available to Grafana.

We're going to install Graphite, the Graphite-API, and Grafana. Then we'll configure and run each component.

Installing Graphite

We're going to install Graphite onto three hosts:

- An Ubuntu 14.04 host in Production A with a hostname of `graphitea.example.com` and an IP address of 10.0.0.210.
- A Red Hat Enterprise Linux (RHEL) 7.0 host in Production B with a hostname of `graphiteb.example.com` and an IP address of 10.0.0.220.
- An Ubuntu 14.04 host in Mission Control with a hostname of `graphitemc.example.com` and an IP address of 10.0.0.200.

This is in line with the architecture we established in Chapter 3.

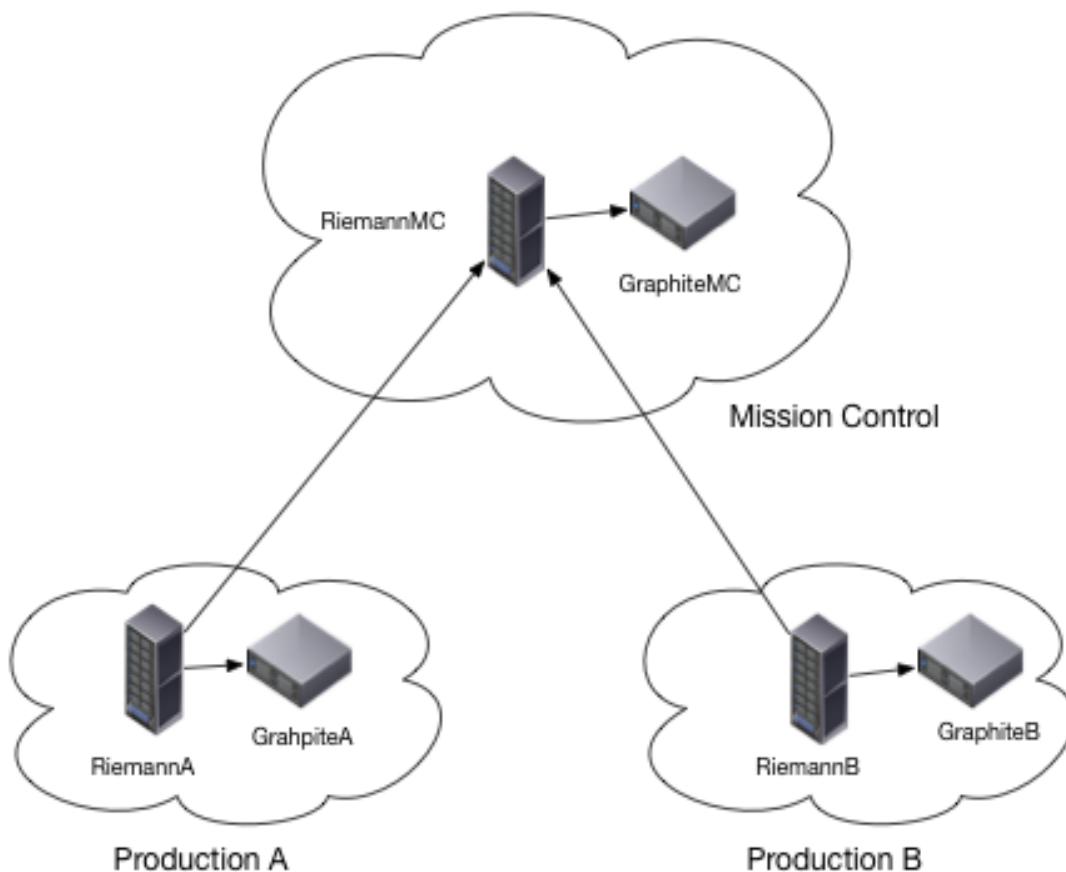


Figure 4.2: Metrics Architecture

We're going to install Graphite on our servers with one **carbon-relay** daemon to receive all our incoming events and two **carbon-cache** daemons to process the events.

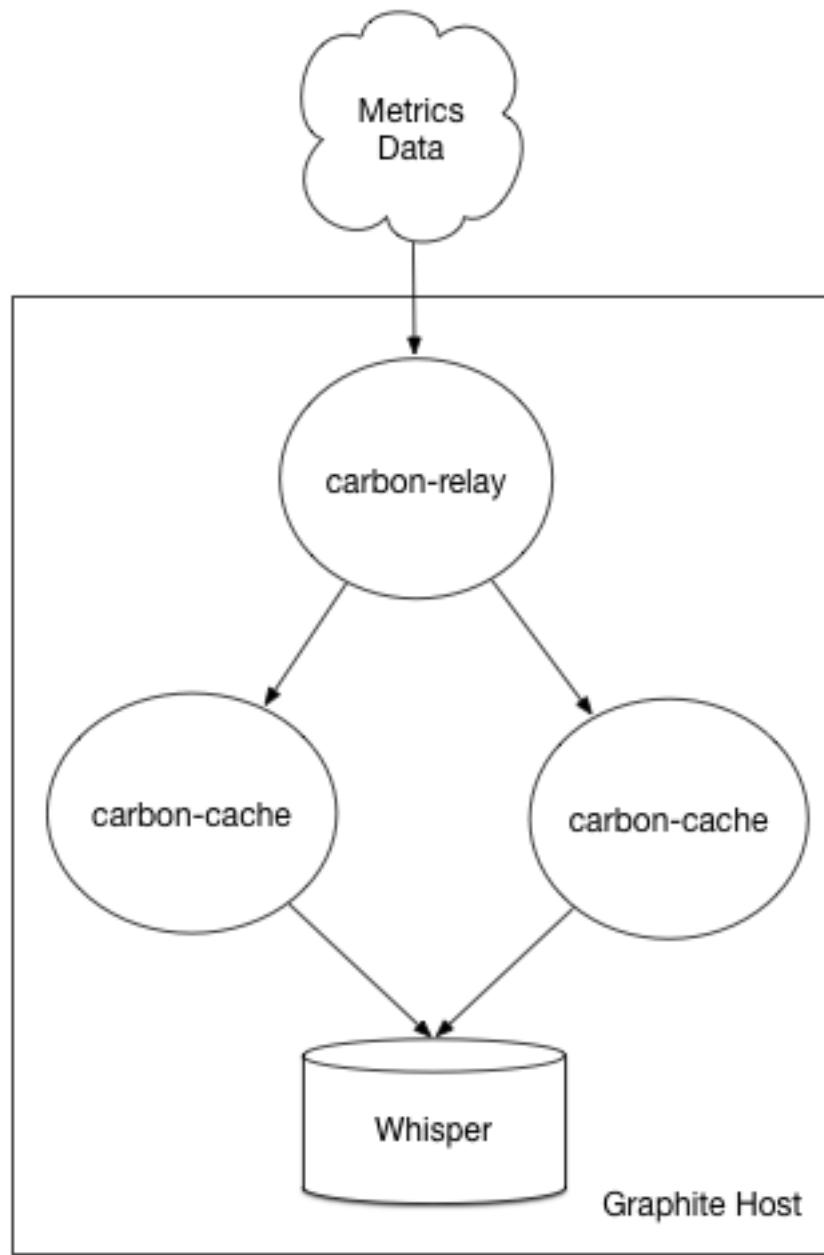


Figure 4.3: Graphite Architecture

This will set us up to be able to scale our Graphite installation by adding **carbon-cache** daemons on our Graphite host, if needed. It'll also make it easier to dis-

tribute our Graphite installation onto multiple hosts if we need to scale that far. To do this we're going to conduct [a manual installation of the required prerequisites](#) and packages to give you an understanding of how Graphite works, but in the real world we'd use a configuration management tool.

Installing Graphite on Ubuntu

On an Ubuntu 14.04 or later host Graphite is available from APT packages.

Let's install the packages we need.

Listing 4.2: Installing the Graphite package on Ubuntu

```
$ sudo apt-get update  
$ sudo apt-get -y install graphite-carbon
```

First we've updated our APT package cache, and then we've installed the **graphite-carbon** package. The **graphite-carbon** package contains the Carbon storage engine.

During installation you'll be asked whether your graph database should be removed if you uninstall Graphite. Answer **No** to ensure your graph data is preserved.

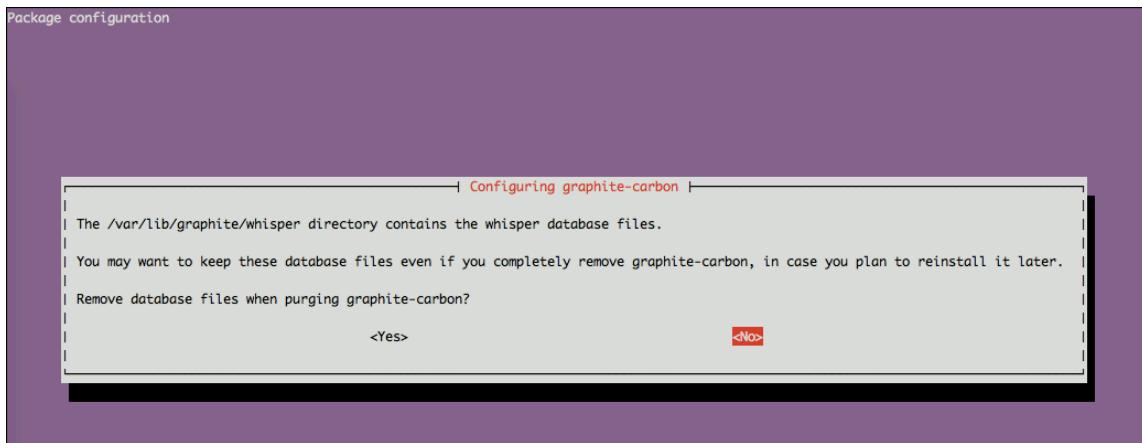


Figure 4.4: Graphite on Ubuntu installation prompt

And we're done.

Installing Graphite on Red Hat

Installing Graphite on Red Hat (and related distributions like CentOS and Fedora) is a bit trickier since not all the packages we need are available from the core package repositories. We'll need to get them from the “Extra Packages for Enterprise Linux” or EPEL repository.

Installing prerequisites on Red Hat

Let's add the EPEL repository now.

Listing 4.3: Adding the EPEL repository for Riemann

```
$ sudo yum install -y epel-release
```

Now let's install the prerequisite packages we require.

Listing 4.4: Installing Graphite prerequisite packages on Red Hat

```
$ sudo yum install -y python-setuptools
```

Here we've installed Python's setup tools, which provide some Python packaging support tools.

Installing Graphite on Red Hat

Now that we have these prerequisite packages, it's time to install Graphite itself.

Listing 4.5: Installing Graphite packages on Red Hat

```
$ sudo yum install -y python-whisper python-carbon
```

TIP On newer Red Hat and family versions, the `yum` command has been replaced with the `dnf` command. The syntax is otherwise unchanged.

Here we've installed `python-whisper` and `python-carbon` to provide Whisper and Carbon respectively.

Final installation steps on Red Hat

On Red Hat we're going to make some small changes to the default package installation to provide some consistency across our hosts and simplify our configuration. If all your hosts running Graphite are running Red Hat you can probably skip this

section and just note the changes in the configuration options in the *Configuring Carbon* section.

Here are our steps:

- Create a new group and user called `_graphite`.
- Move the default path of `/var/lib/carbon/` to `/var/lib/graphite`.
- Change the ownership of the `/var/log/carbon` directory.
- Remove the now unused `carbon` user.

Let's start with creating the new group and user.

Listing 4.6: Creating new Graphite user and group on Red Hat

```
$ sudo groupadd _graphite
$ sudo useradd -c "Carbon daemons" -g _graphite -d /var/lib/
graphite -M -s /sbin/nologin _graphite
```

Next let's move and change the ownership of the `/var/lib/carbon` directory.

Listing 4.7: Moving the `/var/lib/carbon` directory

```
$ sudo mv /var/lib/carbon /var/lib/graphite
$ sudo chown -R _graphite:_graphite /var/lib/graphite
```

Next, let's change the ownership of the `/var/log/carbon` directory.

Listing 4.8: Changing the ownership of /var/log/carbon

```
$ sudo chown -R _graphite:_graphite /var/log/carbon
```

Lastly, let's remove the **carbon** user.

Listing 4.9: Removing the carbon user

```
$ sudo userdel carbon
```

TIP This would all be better managed via a configuration management tool or in a Docker container. This walkthrough should just give you some insight into the changes we're making as we try to make Graphite consistent across hosts.

Installing Graphite-API

Now that we've installed Graphite we'll add the API layer that allows Grafana to connect to Graphite. The [Graphite API](#) is an open-source (Apache 2.0 licensed) add-on for Graphite.

Installing Graphite-API on Ubuntu

On Ubuntu (and Debian) Graphite-API is available as a community package. You can install it from the [exoscale community package repository](#) on the Package-Cloud site. First let's add the repositories APT key.

NOTE If you're not comfortable installing this from PackageCloud then you can make use of the pip-based installation instructions in the Red Hat installation section.

Listing 4.10: Adding the Graphite-API PackageCloud key

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

Then add a new APT repository listing.

Listing 4.11: Adding the PackageCloud exoscale repository listing

```
$ sudo sh -c "echo deb https://packagecloud.io/exoscale/community
/ubuntu/ trusty main > /etc/apt/sources.list.d/
exoscale_community.list"
```

We can now update APT. We may need to add the `apt-transport-https` package as well.

Listing 4.12: Updating APT for Graphite-API

```
$ sudo apt-get install apt-transport-https
$ sudo apt-get update
```

Then we install the `graphite-api` package.

Listing 4.13: Installing the graphite-api package on Ubuntu

```
$ sudo apt-get install graphite-api
```

Installing Graphite-API on Red Hat

Unfortunately there are not yet any native packages available for Red Hat, so we have to install Graphite-API via Python pip. This means we'll have to compile the Graphite-API package ourselves.

First, we need to install some prerequisite packages.

Listing 4.14: Install Graphite-API prerequisite packages on Red Hat

```
$ sudo yum install -y python-pip gcc libffi-devel cairo-devel  
libtool libyaml-devel python-devel
```

This installs a compiler, some development libraries, and the Python `pip` binary command. We'll then use the `pip` command to update some packages and install Graphite-API. We're going to upgrade some local Python libraries and then install some new packages.

Listing 4.15: Installing Graphite-API via pip

```
$ sudo pip install -U six pyparsing websocket urllib3  
$ sudo pip install graphite-api gunicorn
```

This will upgrade some Graphite-API prerequisites and then install the `graphite-api` and `gunicorn` pip packages. We'll use the `gunicorn` package to deploy Graphite-API later in this chapter.

TIP There's also a [Puppet module](#) to install Graphite-API on Red Hat.

Installing Grafana

Lastly, we're going to install Grafana. There are native packages available for both Ubuntu and Red Hat.

Installing Grafana on Ubuntu

We can use Grafana's package repository to install Grafana on Ubuntu. To do so, we'll need to add another repository listing.

Listing 4.16: Adding the Grafana repository listing

```
$ sudo sh -c "echo deb https://packagecloud.io/grafana/stable/
  debian/ wheezy main > /etc/apt/sources.list.d/
  packagecloud_grafana.list"
```

Then we'll add the Package Cloud key. This allows us to install their signed packages.

Listing 4.17: Adding the PackageCloud key

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

Finally, we'll update our Apt repositories and install the `grafana` package. (We may also need the `apt-transport-https` package, so we're going to install that too).

Listing 4.18: Installing the Grafana package

```
$ sudo apt-get update  
$ sudo apt-get install -y apt-transport-https grafana
```

Installing Grafana on Red Hat

To install on Red Hat we can use Grafana's package repository. To do this we'll add another Yum repository listing.

Create a new Yum repository definition at `/etc/yum.repos.d/grafana.repo`.

Listing 4.19: Creating the Grafana Yum repository

```
$ sudo touch /etc/yum.repos.d/grafana.repo
```

Then populate the `/etc/yum.repos.d/grafana.repo` file with the following:

Listing 4.20: Yum repository definition for Grafana

```
[grafana]
name=grafana
baseurl=https://packagecloud.io/grafana/stable/el/6/$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
```

NOTE Repace the 6 above with your Red Hat version, for example 7 for RHEL 7.

Now install Grafana via the `yum` command.

Listing 4.21: Installing Grafana via Yum

```
$ sudo yum install grafana
```

TIP You may be prompted to accept GPG keys from PackageCloud during the process.

Installing Graphite and Grafana via configuration management

There are a variety of options for installing Graphite, Graphite-API, and Grafana via configuration management.

You can find Chef cookbooks for:

- Graphite at <https://github.com/hw-cookbooks/graphite>.
- Graphite-API at <https://supermarket.chef.io/cookbooks/graphite-api>.
- Grafana at <https://supermarket.chef.io/cookbooks/grafana>.

You can find Puppet modules for:

- Graphite at <https://forge.puppetlabs.com/garethr/graphite>.
- Graphite-API at <https://forge.puppetlabs.com/stevenmerrill/graphiteapi>.
- Grafana at <https://forge.puppetlabs.com/modules?utf-8=%E2%9C%93&sort=rank&q=grafana>.

You can find Ansible roles for:

- Graphite at <https://galaxy.ansible.com/list#/roles/391>.
- Graphite-API at <https://galaxy.ansible.com/list#/roles/1944>.
- Grafana at <https://galaxy.ansible.com/list#/roles/3563>.

You can find Docker images for:

- Graphite at <https://hub.docker.com/search/?q=graphite>.
- Graphite-API at <https://hub.docker.com/r/brutasse/graphite-api/>.
- Grafana at <https://hub.docker.com/search/?q=grafana>.

You can find a Vagrant-based configuration tool for Graphite called Synthesize at <https://github.com/obfuscuity/synthesize>.

Configuring Graphite and Carbon

Next we need to configure Graphite and Carbon. Both Ubuntu and Red Hat install a default configuration file for Carbon. That file is `/etc/carbon/carbon.conf`. That example file is highly useful as it documents each option available to Carbon. We're going to replace it, however, with our own file to configure Carbon.

Listing 4.22: Editing the Carbon settings

```
$ sudo vi /etc/carbon/carbon.conf
```

The Carbon configuration file is broken into “ini” file style sections managing each daemon: `[cache]` for the `carbon-cache` daemon and `[relay]` for the `carbon-relay` daemon.

NOTE There's another section `[aggregator]` for the carbon-aggregation daemon. It's a **buffering and metric manipulation** daemon we're not going to use in this book.

Let's look at our `carbon.conf` configuration file now. It's a bit big so we're going to break it into pieces to review. You can find the full file in [the book's source code on GitHub](#).

Let's look at each section in turn. The first section, `[cache]`, configures our `carbon-cache` daemon. The first collection of configuration options controls where Carbon expects to find various components.

Listing 4.23: Directory configuration for Carbon

```
STORAGE_DIR      = /var/lib/graphite/
CONF_DIR        = /etc/carbon/
LOG_DIR         = /var/log/carbon/
PID_DIR         = /var/run/
LOCAL_DATA_DIR = /var/lib/graphite/whisper/
```

You can see we've configured several directory locations. Of particular note is the `/var/log/carbon` directory where you'll find all of Carbon's log files.

NOTE If you installed on Red Hat we've potentially made some consistency changes during installation, such as changing the Carbon user to `_graphite`. This is where you would update Carbon if you've chosen to change anything.

Next, we've set the Carbon user and log rotation settings.

Listing 4.24: Setting the Carbon user and log rotation

```
USER = _graphite
ENABLE_LOGROTATION = True
LOG_UPDATES = False
LOG_CACHE_HITS = False
```

We've set the `USER` to `_graphite` for the `carbon-cache` daemon. You'll see we've set the user again in the `[relay]` section. You need to set the user for each Carbon

daemon you are running.

We've also enabled log rotation for our `carbon-cache` daemon and disabled logging of updates and cache hits. We don't need to know every time a metric is updated—that can even cause contention if we're writing logs to the same filesystem as our metrics.

Now let's look at the `carbon-cache` daemon's networking and port configuration.

Listing 4.25: Configuring the Carbon Cache daemon

```
LINE_RECEIVER_INTERFACE = 127.0.0.1
PICKLE_RECEIVER_INTERFACE = 127.0.0.1
CACHE_QUERY_INTERFACE = 127.0.0.1

[cache:1]
LINE_RECEIVER_PORT = 2013
PICKLE_RECEIVER_PORT = 2014
CACHE_QUERY_PORT = 7012

[cache:2]
LINE_RECEIVER_PORT = 2023
PICKLE_RECEIVER_PORT = 2024
CACHE_QUERY_PORT = 7022
```

Note that we've first defined three interfaces.

- `LINE_RECEIVER_INTERFACE`
- `PICKLE_RECEIVER_INTERFACE`
- `CACHE_QUERY_INTERFACE`

We've set each interface to bind to `127.0.0.1` or `localhost`. These daemons will

be fed by the `carbon-relay` daemon, so they only need to be bound locally and not exposed. Each interface performs a different function.

The first two interfaces, `LINE_RECEIVER_INTERFACE` and `PICKLE_RECEIVER_INTERFACE`, handle Carbon's two protocols: line or plaintext and pickle.

The **plaintext protocol** is basic and takes single metrics in the following form:

Listing 4.26: The Carbon plaintext protocol

```
<metric path> <metric value> <metric timestamp>
```

For example:

Listing 4.27: The Carbon plaintext protocol metric example

```
riemann.cpu.usage 88 1423507423
```

Carbon will translate this into a form that Whisper can then write to disk. The protocol is so simple and plaintext that you can use Netcat, `nc`, to send a sample metric. For example:

Listing 4.28: Sending a plaintext metric via Netcat to Graphite

```
PORT=2003
SERVER=graphitea.example.com
echo "riemann.someservice.somemetric 12 `date +%s`" | nc -c ${
    SERVER} ${PORT}
```

The **pickle protocol** is a much more efficient variant of the plaintext protocol.

Instead of receiving single metrics, it supports batches of metrics. The pickled data takes the form of a list of multi-level tuples.

Listing 4.29: The Graphite pickle protocol

```
[(path, (timestamp, value)), ...]
```

Each tuple contains the measurement for a single metric name. The measurement is nested as a second tuple containing the timestamp and value. Your client or application then accumulates an appropriately sized list of tuples and sends them to the pickle receiver as a packet with a header attached.

The last interface, the `CACHE_QUERY_INTERFACE`, is the interface on which `carbon-cache` will listen for incoming queries from the Graphite Web interface. Sometimes the web interface wants data that is still in the cache rather than in Whisper files. This interface allows the Graphite Web interface to query that data.

Next we see our two `carbon-cache` instances: `[cache:1]` and `[cache:2]`. A `carbon-cache` instance is configured by suffixing the name of the daemon, `cache`, with an instance name.

Listing 4.30: Carbon Cache instance definitions

```
[cache:name_of_instance]
```

We're naming our instances numerically but you could also use letters or other names. In this case we're using numerical naming because it makes starting and managing them easier from service management as we'll see later in the chapter.

We've already defined interface bindings for each `carbon-cache` daemon. Inside each instance we define the ports each instance will be bound on. So for instance `1`, the plaintext protocol is bound on port 2013, the pickle protocol on port 2014,

and the cache query on port 7012. For instance 2, the plaintext protocol is bound on port 2023, the pickle protocol on port 2024, and the cache query on port 7022.

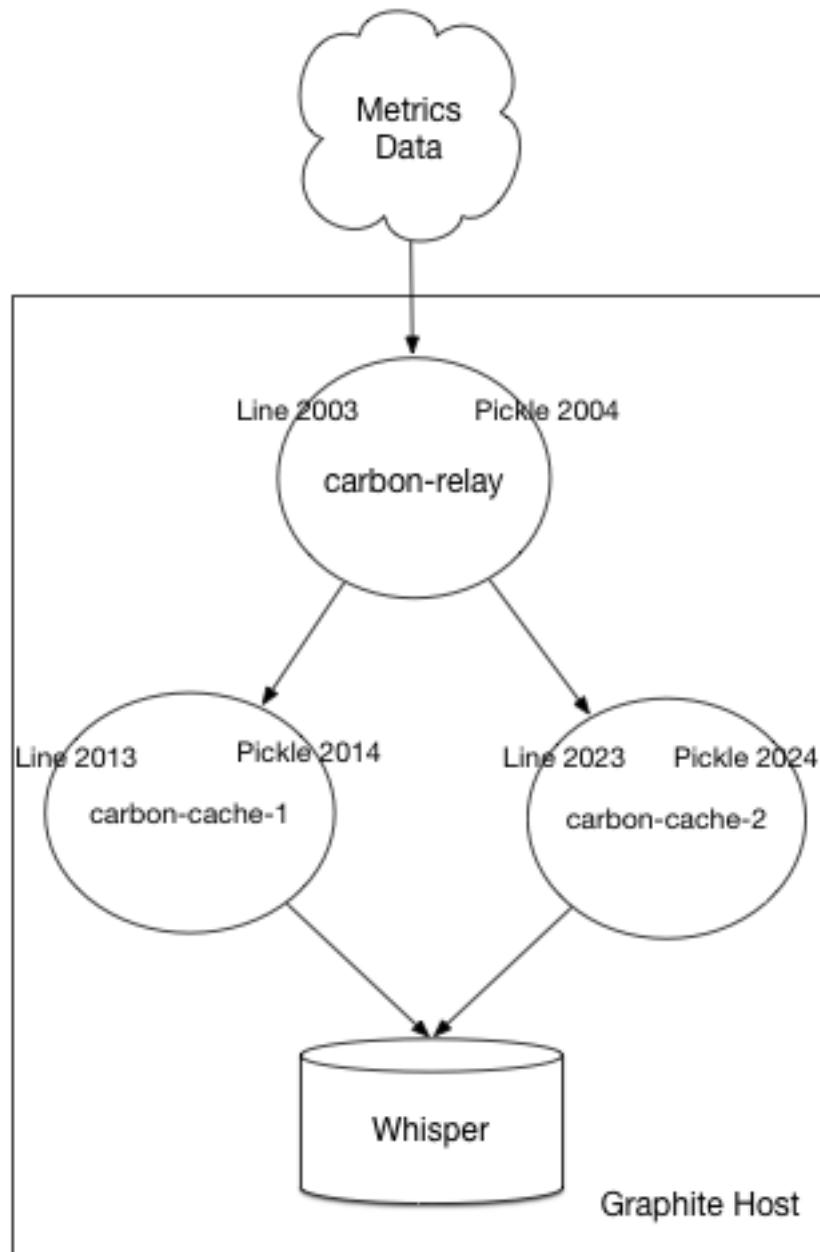


Figure 4.5: Carbon daemon ports and architecture

You can see how we've distributed the ports. If we wanted to add another instance we'd specify:

Listing 4.31: A third carbon-cache instance

```
[cache:3]
LINE_RECEIVER_PORT = 2033
PICKLE_RECEIVER_PORT = 2034
CACHE_QUERY_PORT = 7032
```

And so on.

This configuration ties into our [carbon-relay](#) configuration:

Listing 4.32: The carbon-relay daemon configuration

```
[relay]
USER = _graphite
LINE_RECEIVER_INTERFACE = 10.0.0.210
LINE_RECEIVER_PORT = 2003
PICKLE_RECEIVER_INTERFACE = 10.0.0.210
PICKLE_RECEIVER_PORT = 2004
RELAY_METHOD = consistent-hashing
REPLICATION_FACTOR = 1
DESTINATIONS = 127.0.0.1:2014:1, 127.0.0.1:2024:2
```

We've now created a `[relay]` configuration block for our [carbon-relay](#) daemon. We've specified the `_graphite` user, and we've defined the plaintext and pickle protocol interfaces to bind to our external network interface—here we're using [graphitea](#)'s IP address of 10.0.0.210. We've further bound the plaintext protocol

to port 2003 and the pickle protocol to 2004. Carbon will receive metrics upon this interface and these ports.

The `carbon-relay` then uses the `RELAY_METHOD` and `REPLICATION_FACTOR` to determine how it will distribute those metrics. The `RELAY_METHOD` specifies to which `carbon-cache` instances specific metrics will be distributed. This can be a [rules-based configuration](#), if we, for example, specify the distribution by name or group of metrics. Or it can be a consistent hash, where metrics are evenly distributed between `carbon-cache` daemons. We've chosen the latter, [consistent-hashing](#) method.

The `REPLICATION_FACTOR` is used to control how many replicas of metrics are sent to each destination. In this case we're sending one version of each metric. We could set this to `2` or higher to send replicas of each metric to multiple destinations to provide metric redundancy. We'll see an example of this later in this chapter.

The last configuration option, `DESTINATIONS`, tells the `carbon-relay` daemon where to send metrics. In this case we've specified the pickle protocol ports of each of our `carbon-cache` instances: `127.0.0.1:2014:1` for instance `1`, and `127.0.0.1:2024:2` for instance `2`. If we added further instances here we'd add more entries. For example, if we were to add instance `3` above, the line would become:

Listing 4.33: Adding a third instance destination to carbon-relay

```
DESTINATIONS = 127.0.0.1:2014:1, 127.0.0.1:2024:2,  
          127.0.0.1:2034:3
```

TIP A (highly) simplistic rule for scaling the `carbon-cache` and `carbon-relay` daemons is one daemon per CPU core.

In our current configuration a metric would be received on our Graphite host on either port 2003 (plaintext) or port 2004 (pickle) by the `carbon-relay` daemon. That daemon would pass the metric to one of the `carbon-cache` daemons, determined by Carbon’s consistent hashing, on the `localhost` on either port 2014 or port 2024. The `carbon-cache` daemon will in turn write the metric in Whisper format to the disk.

Configuring Carbon’s metric retention

Next we’re going to configure the density of Carbon’s metric collection—essentially how long metrics should be stored and how detailed those metrics should be. In the metrics world we call that resolution.

Metric resolution is critical to good monitoring. For effective monitoring it’s important to collect the right information at the right frequency. Many traditional monitoring and metrics systems collect metrics at low resolution—for example, one data point every minute or even five minutes. This presents some big problems if we want to use this data to work out what’s going on. A single data point collected over a long period, especially for a volatile or variable service like CPU, memory, or metrics like transaction counts, isn’t likely to capture the scope of what’s going on with that component. This can result in a false sense of the state of that component.

In our case, we’re going to collect data at a high resolution, generally a minimum of one data point every two to three seconds. This should give us a clear picture of the state of most of the components we’re collecting metrics on.

Naturally there is a downside to collecting at high resolutions: performance and storage. The more metrics you collect, the more frequently you collect data points, the more time this all takes, and the more storage this consumes. Shortly we’ll talk more about metrics resolution and configuring resolutions for specific types of metrics, and we’ll provide you with some calculations to help you understand the performance and storage costs of higher resolutions.

In the meantime, though, we're going to configure Carbon to provide a higher resolution by default.

Remember that your resolution decision is fairly permanent for your metrics. If you change the resolution, you'll be unable to use the previously collected data for comparisons. This is because it's generally fairly hard to compare metrics collected at different resolutions. Additionally, you'll either need to [resize the Whisper files](#) or delete and recreate them. This is a non-trivial exercise—choose your resolution with care!

In Carbon, metric resolution is configured in a file called `/etc/carbon/storage-schemas.conf`. Let's look at this file now.

Listing 4.34: Configuring Carbon storage schemas

```
# Schema definitions for Whisper files. Entries are scanned in
# order,
# and first match wins. This file is scanned for changes every 60
# seconds.
#
# [name]
# pattern = regex
# retentions = timePerPoint:timeToStore, timePerPoint:
#               timeToStore, ...

# Carbon's internal metrics. This entry should match what is
# specified in
# CARBON_METRIC_PREFIX and CARBON_METRIC_INTERVAL settings
[carbon]
pattern = ^carbon\.
retentions = 60:90d

[default_1min_for_1day]
pattern = .*
retentions = 60s:1d
```

A storage schema specifies how long Carbon should keep metrics and at what resolution they should be kept. A schema entry lists metrics by name and then specifies one or more retention period for metrics that match.

Listing 4.35: Carbon storage schema

```
[name]
pattern = regex_pattern
retentions = periods
```

A storage schema entry consists of a name wrapped in square brackets, which uniquely identifies the specific schema. It has a regular expression **pattern** that matches specific metrics by their names. Also, a schema needs **retentions**. These specify the resolutions at which Carbon will store each individual metric and for how long it will store at that resolution. You can specify one or more retentions in the form of:

Listing 4.36: Sample retention periods for Graphite

```
sample_time:retention_period
```

Let's look at the entries in our sample configuration file. The first entry, **[carbon]**, manages Carbon's own metrics. A regular expression pattern is matched to find these: any metric starting with **carbon**. The metrics retention is then set for the Carbon metrics with the **retentions** entry.

For the Carbon metrics, a data point is created every 60 seconds and kept for 90 days: **60:90d**. This means each data point represents 60 seconds, and we want to keep enough data points for 90 days worth of data.

All other metrics in the default configuration use the **[default_1min_for_1day]** schema; the pattern matches **.*** or all events. In this schema, Graphite creates data points every 60 seconds and keeps enough data to represent one day. That's a pretty low resolution by most standards, and Riemann processes events

much more quickly. So we're going to create a new schema and remove the [[default_1min_for_1day](#)] schema.

Listing 4.37: Creating a new default Graphite schema

```
[default]
pattern = .*
retentions = 1s:24h, 10s:7d, 1m:30d, 10m:2y
```

We've called our new schema [\[default\]](#). Its [pattern](#) again matches `.*` or all events. This new schema, however, includes multiple retentions. Multiple retentions allow graceful downsampling of historical data, saving you disk and performance. Our first retention, [1s:24h](#) creates data points every one second and keeps enough data for 24 hours. Our next retention, [10s:7d](#), retains 10-second data points for seven days and so on, right down to keeping 10-minute data points for two years.

To do the down sample from [1s:24h](#) to [10s:7d](#), Graphite gathers all of the data from the past 10 seconds (this should be 10 data points, one generated every one second). It then averages the data points to aggregate them and retains this new data point for seven days. By default, each retention averages the total as it down samples, so you should generally be able to determine metrics totals by reversing the average.

You can also configure Graphite to use methods other than averaging to aggregate the data points, including: min, max, sum, and last. This is done by configuring a [/etc/carbon/storage-aggregation.conf](#) file. You can use this sample file as a reference:

```
/usr/share/doc/graphite-carbon/examples/storage-aggregation.conf.
example
```

We're not going to use these aggregations right now, but there is an annoyingly

frequent log message that appears in our Carbon logs, `/var/log/carbon/console.log`:

Listing 4.38: Annoying Graphite log message

```
/etc/carbon/storage-aggregation.conf not found, ignoring.
```

Creating an empty `/etc/carbon/storage-aggregation.conf` file stops the message so let's do that now.

Listing 4.39: Creating an empty storage aggregation file

```
$ touch /etc/carbon/storage-aggregation.conf
```

Estimating Graphite storage

Whisper database files are fixed size—they don't grow like a normal relational database. Each Whisper file is created when a new metric is received. The size is calculated by the resolution and retention period of the metric being stored. A Whisper database file never becomes smaller or larger on its own.

As Whisper files are fixed size, we can easily predict our storage requirements. Let's look at an example. Firstly, each Graphite metric data point is 12 bytes in length. Our resolution and retention configuration will tell us how many data points will be stored. Let's quickly look back at that.

Listing 4.40: Creating a new default Graphite schema

```
[default]
pattern = .*
retentions = 1s:24h, 10s:7d, 1m:30d, 10m:2y
```

So if we're storing MetricA at one second intervals for 24 hours, we can easily calculate this as 86,400 data points (there are usually 86,400 seconds in a day). At 12 bytes a data point, this is 1,036,800 bytes or ~1.0368Mb.

That's only one of the pieces of our retention though. We'd have to calculate each to get the full size of the Whisper database file. [J. Javier Maestro](#) has written a Whisper calculator that makes life much easier. The `whisper-calculator.py` is a Python application that you can feed a retention period and it'll return a Whisper database file size.

Let's grab the calculator now.

Listing 4.41: Downloading the calculator

```
$ cd ~
$ wget https://gist.githubusercontent.com/jjmaestro/5774063/raw/9
b615fa9a4666529c264af738ffa34ecc0298bd6/whisper-calculator.py
```

Now we try it out.

Listing 4.42: The `whisper-calculator.py`

```
$ python ./whisper-calculator.py 1s:24h,10s:7d,1m:30d,10m:2y  
1s:24h,10s:7d,1m:30d,10m:2y >> 3542464 bytes
```

Here we provided our retention period, and the calculator returned a size of 3,542,464 bytes—so for each metric we’re collecting we’ll need ~3.5Mb in space. We can then work out a total capacity requirement, factoring in number of hosts and metrics. For example, for 100 hosts, each with 100 metrics, we’d need 35,424Mb or 35.4Gb.

NOTE There’s also a useful command shipped with Graphite called `whisper-info` that you can use on an existing Whisper file to return details of what data points are being stored and their sizes.

Carbon and Graphite service management

Now we need to set up Carbon and Graphite so that they’ll run via service management.

Service management on Ubuntu

We need to replace the standard Ubuntu init scripts installed by the Graphite package for `carbon-cache` and add a new init script for the `carbon-relay` daemon. This will allow us to run the multiple `carbon-cache` daemons as we configured above as well as a `carbon-relay` daemon.

We've prepared a `carbon-cache` init script you can download from [GitHub](#). It's based on the existing `carbon-cache` init script with modifications to run multiple daemons. The control of how many daemons is provided by an environment variable set in the `/etc/default/graphite-carbon` file. We've also provided an example `/etc/default/graphite-carbon` file.

Let's get started.

Download the init script.

Listing 4.43: Download the Carbon Cache init script on Ubuntu

```
$ wget https://raw.githubusercontent.com/jamtur01/aom-code/master  
/4/graphite/carbon-cache-ubuntu.init
```

Copy the init script into `/etc/init.d/` and set its permissions.

Listing 4.44: Install the Carbon Cache init script on Ubuntu

```
$ sudo cp carbon-cache-ubuntu.init /etc/init.d/carbon-cache  
$ sudo chmod 0755 /etc/init.d/carbon-cache
```

Enable the daemon.

Listing 4.45: Enable the Carbon Cache init script on Ubuntu

```
$ sudo update-rc.d carbon-cache defaults
```

We also need to configure an init script for our `carbon-relay` daemon. Repeat the steps here.

Listing 4.46: Install the Carbon relay init script on Ubuntu

```
$ wget https://raw.githubusercontent.com/jamtur01/aom-code/master  
/4/graphite/carbon-relay-ubuntu.init  
$ sudo cp carbon-relay-ubuntu.init /etc/init.d/carbon-relay  
$ sudo chmod 0755 /etc/init.d/carbon-relay  
$ sudo update-rc.d carbon-relay defaults
```

Finally, let's configure Carbon to run by default by editing the `/etc/default/graphite-carbon` file.

Listing 4.47: Enable Graphite at startup on Ubuntu

```
$ sudo vi /etc/default/graphite-carbon
```

Change the value of `CARBON_CACHE_ENABLED=false` to `CARBON_CACHE_ENABLED=true`. This will tell the `carbon-cache` to run at boot time.

We're also going to configure how many `carbon-relay` and `carbon-cache` daemons we want to run inside this file. To do this we'll use two variables: `RELAY_INSTANCES` and `CACHE_INSTANCES`. These variables will be used inside our `carbon-relay` and `carbon-cache` init scripts to launch the specified number of daemons. This will allow us to quickly scale daemons. Set these variables to `1` and `2` respectively.

Your final `/etc/default/graphite-carbon` file will look like:

Listing 4.48: The /etc/default/graphite-carbon file

```
CARBON_CACHE_ENABLED=true  
RELAY_INSTANCES=1  
CACHE_INSTANCES=2
```

Let's start the `carbon-cache` and `carbon-relay` daemons.

Listing 4.49: Starting the Carbon daemons on Ubuntu

```
$ sudo service carbon-relay start  
$ sudo service carbon-cache start
```

You should now be able to check the `/var/log/carbon/console.log` file to see if the daemons are running and Carbon is up.

NOTE We've included all example configuration including service management scripts in the book [on GitHub](#).

Soon we'll connect Riemann to Carbon and you'll be able to see metrics flowing into Graphite.

Service management on Red Hat

On Red Hat we use systemd for service management. We're going to create systemd unit files for both the `carbon-cache` and `carbon-relay` daemons. We need to run two instances of the `carbon-cache` and a single instance of the `carbon-`

`relay` daemon. We're going to take advantage of `systemd's instantiated services` to create a single unit file with multiple instances.

We'll start with the `carbon-cache` daemon. Let's create a file called:

`/lib/systemd/system/carbon-cache@.service`

TIP The @ indicates we're going to use instantiated services inside our unit file.

Populate that file like so:

Listing 4.50: systemd unit file for the Carbon Cache daemon

```
[Unit]
Description=carbon-cache instance %i (graphite)

[Service]
ExecStartPre=/bin/rm -f /var/run/carbon-cache-%i.pid
ExecStart=/usr/bin/carbon-cache --config=/etc/carbon/carbon.conf
--pidfile=/var/run/carbon-cache-%i.pid --logdir=/var/log/carbon
/ --instance=%i start
Type=forking
PIDFile=/var/run/carbon-cache-%i.pid

[Install]
WantedBy=multi-user.target
```

Our unit file runs instances of the `carbon-cache` daemon. In our case, each instance will correlate with the cache instances we configured earlier: 1 and 2. In

the unit file this is represented by the `%i` variable. When the unit file is run this will be replaced with the instance number we'll pass to it.

Let's enable and start each `carbon-cache` daemon.

Listing 4.51: Enabling and starting the systemd Carbon Cache daemons

```
$ sudo systemctl enable carbon-cache@1.service
$ sudo systemctl enable carbon-cache@2.service
$ sudo systemctl start carbon-cache@1.service
$ sudo systemctl start carbon-cache@2.service
```

These commands will enable two instances of the `carbon-cache` daemon, `carbon-cache@1` and `carbon-cache@2`. We've also started each instance of the `carbon-cache` daemon.

Now let's create a unit file for the `carbon-relay` daemon. We'll call it `/lib/systemd/system/carbon-relay@.service`, and we'll populate it.

Listing 4.52: systemd unit file for the Carbon relay daemon

```
[Unit]
Description=carbon-relay instance %i (graphite)

[Service]
ExecStartPre=/bin/rm -f /var/run/carbon-relay-%i.pid
ExecStart=/usr/bin/carbon-relay --config=/etc/carbon/carbon.conf
--pidfile=/var/run/carbon-relay-%i.pid --logdir=/var/log/carbon
/ --instance=%i start
Type=forking
PIDFile=/var/run/carbon-relay-%i.pid

[Install]
WantedBy=multi-user.target
```

Let's enable and start the `carbon-relay` daemon.

Listing 4.53: Enabling and starting the systemd Carbon relay daemon

```
$ sudo systemctl enable carbon-relay@1.service
$ sudo systemctl start carbon-relay@1.service
```

And let's delete the old systemd unit files.

Listing 4.54: Remove the old systemd unit files for Carbon

```
$ sudo rm -f /lib/systemd/system/carbon-relay.service  
$ sudo rm -f /lib/systemd/system/carbon-cache.service
```

You should now be able to check the `/var/log/carbon/console.log` file to see if the daemons are running and Carbon is up.

NOTE We've included all example configuration including service management scripts in the book [on GitHub](#).

Shortly we'll connect Riemann to Carbon and you'll be able to see metrics flowing into Graphite.

Configuring Graphite-API

The default configuration file for Graphite-API is located at `/etc/graphite-api.yaml`. It's installed automatically on Ubuntu but on Red Hat we'll need to create the file. The configuration file uses the YAML format.

On both distributions we're going to use our own configuration file. Populate the `/etc/graphite-api.yaml` file with the following content:

Listing 4.55: The /etc/graphite-api.yaml file

```
search_index: /var/lib/graphite/api_search_index
finders:
  - graphite_api.finders.whisper.WhisperFinder
functions:
  - graphite_api.functions.SeriesFunctions
  - graphite_api.functions.PieFunctions
whisper:
  directories:
    - /var/lib/graphite/whisper
carbon:
  hosts:
    - 127.0.0.1:7012
    - 127.0.0.1:7022
  timeout: 1
  retry_delay: 15
  carbon_prefix: carbon
  replication_factor: 1
  time_zone: UTC
```

TIP You can find a full description of Graphite-API's configuration in [their documentation](#).

Let's examine what this does. The first option, `search_index`, is the location of a search index file that Graphite-API will use to index metrics. We've put it in our `/var/lib/graphite` directory. It needs to be writeable by the Graphite-API.

Let's create it now and change its ownership to the `_graphite` user.

Listing 4.56: Creating the `/var/lib/graphite/api_search_index` file

```
$ sudo touch /var/lib/graphite/api_search_index
$ sudo chown _graphite:_graphite /var/lib/graphite/
api_search_index
```

The next two sets of options, `finders` and `functions`, provide plugins to connect and manipulate various API functions. The `finders` are plugins that specify places where Graphite-API can find Graphite data. In our case we're only defining one finder: `graphite_api.finders.whisper.WhisperFinder`. This finder locates and loads Whisper files on the local filesystem. If you were using another source of Graphite data—for example, a third-party data store—[there are other finders you could use](#).

Functions are used to transform, combine, and perform computations on data pulled from Graphite. In this case Grafana makes calls to retrieve data from Graphite that include some functions it expects the Graphite API to understand. [These functions are provided by Graphite-API](#).

The `whisper` option is used by the `graphite_api.finders.whisper.WhisperFinder` finder to specify the location of the Whisper files we want Graphite-API to query. This is a list of the directories where we can find Whisper files—for us, that'll be `/var/lib/graphite/whisper`.

Sometimes the data you want might be inside Carbon's cache. The `carbon` option allows Graphite-API to query that cache. We specify the `hosts` option with a list of Carbon Cache daemons and ports. Remember, we've already bound our two Carbon Cache daemons to `127.0.0.1` or localhost. The port specified in the Carbon query port is defined by the `CACHE_QUERY_PORT` for each Cache daemon in the Carbon configuration we defined above. In this case, our first cache is on port

7012 and our second on port 7022.

The last option, `time_zone`, is the time zone of the Graphite data we're pulling. It's crucial this matches the time zone of the host. If your time zone is set wrong you can expect to see some odd results in your data as time series become confused. We've set it to `UTC` time, and we'll see later in this chapter how to ensure our host and other services match and stay accurate.

Service management for Graphite-API

On Ubuntu we don't need to configure any service management as the package we installed comes with an init script. You can use it to both start and stop Graphite-API.

Listing 4.57: Restarting the Graphite-API on Ubuntu

```
$ sudo service graphite-api start
```

Graphite-API will now be bound to port 8888 on all interfaces on the host. You can adjust the configuration in the `/etc/init.d/graphite-api` init script to bind it to other interfaces or to the local host.

WARNING There is a minor bug in the init script of the 1.0.1 release of the Graphite API. On line 47 of the init script at `/etc/init.d/graphite-api` you may need to change the variable `$PID_FILE` to `$PIDFILE`. This is fixed in later releases.

On Red Hat, though, we need to create an init script to run the Graphite-API daemon. We're going to create a systemd script to run Graphite-API.

Let's create a file called `/lib/systemd/system/graphite-api.service`. We're going to populate that file with the following configuration:

Listing 4.58: Systemd script for Graphite-API

```
[Unit]
Description=graphite-api (graphite)

[Service]
ExecStartPre=/bin/rm -f /var/run/graphite-api.pid
ExecStart=/usr/bin/gunicorn --pid /var/run/graphite-api.pid -b
    0.0.0.0:8888 --daemon graphite_api.app:app
Type=forking
PIDFile=/var/run/graphite-api.pid

[Install]
WantedBy=multi-user.target
```

NOTE We've included all example configuration including service management scripts in the book [on GitHub](#).

This runs Graphite-API through the `gunicorn` daemon and binds it to all interfaces on port `8888`. You could also pass `gunicorn` the `-w` flag to specify the number of workers for the server. A good rule of thumb for the number of workers is two times the number of cores on the host plus one.

Listing 4.59: Gunicorn worker calculations

```
(2 x number_of_cores) + 1
```

TIP For more information on Gunicorn workers you can read [this useful FAQ entry](#).

We can then enable and start the Graphite-API service.

Listing 4.60: Enabling and starting the systemd Graphite-API daemons

```
$ sudo systemctl enable graphite-api.service  
$ sudo systemctl start graphite-api.service
```

Testing the Graphite-API

To test that the API is working, we query a URL. On your Graphite hosts use a web browser to browse to `http://hostname:8888/render?target=test`. For example, on our `graphitea` host this would be:

`http://graphitea.example.com:8888/render?target=test`

You should see an empty graph with a label of `No data`.

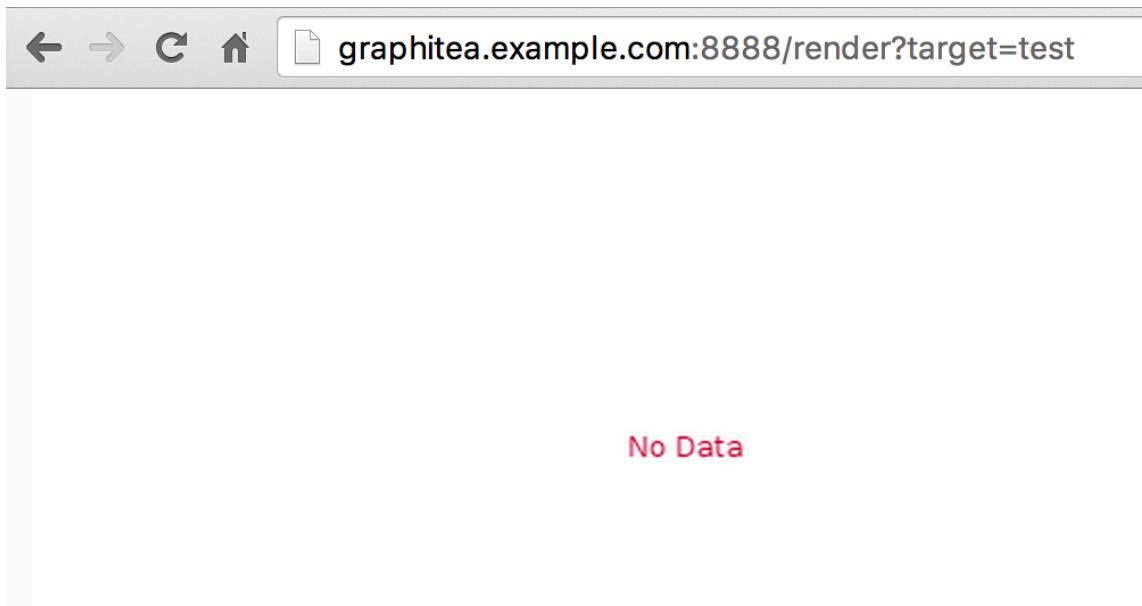


Figure 4.6: Testing Graphite-API

Don’t panic about the “No data” message. This isn’t an error—we just haven’t selected any specific data.

Configuring Grafana

There are two places where we can configure Grafana: a local configuration file `/etc/grafana/grafana.ini` and the Grafana web interface. To access that we need to start the Grafana web service, so let’s do that first with the `service` binary.

Listing 4.61: Starting the Grafana Server

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

This will work on both Ubuntu and Red Hat. Grafana is a Go-based web service that runs on port `3000` by default. Once it’s running you can browse to it using

your web browser.

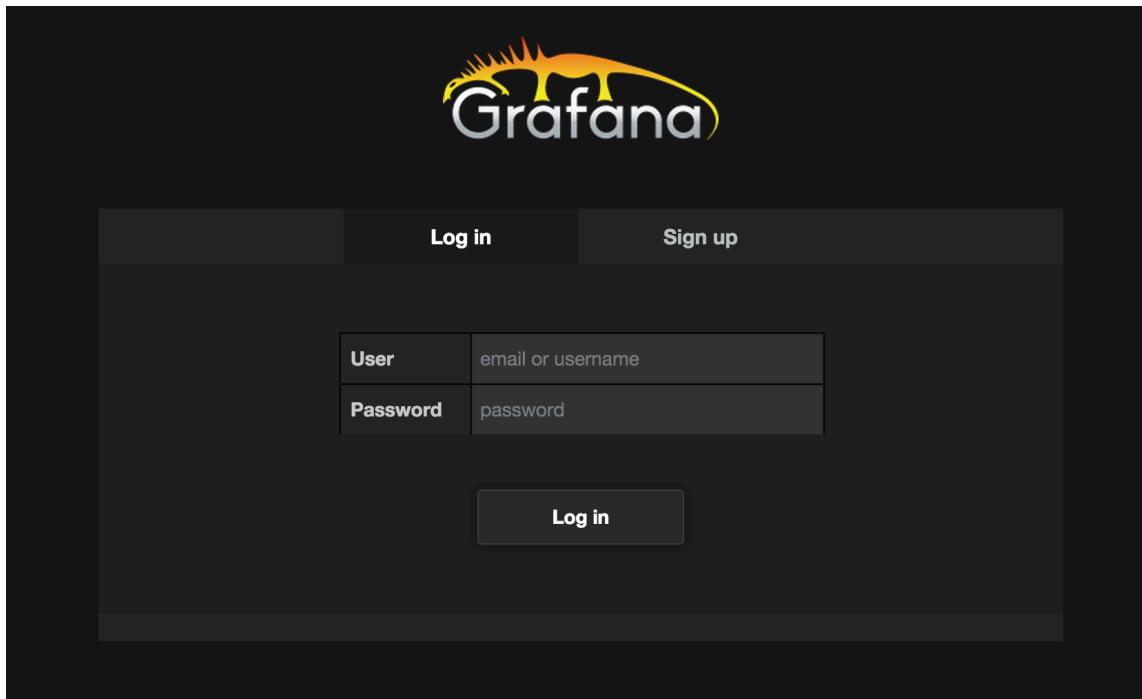


Figure 4.7: The Grafana Console login

You'll see a login screen initially. The default username and password is `admin` and `admin`. You can control this by updating [the \[security\] section](#) of the `/etc/grafana-server.ini` 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.

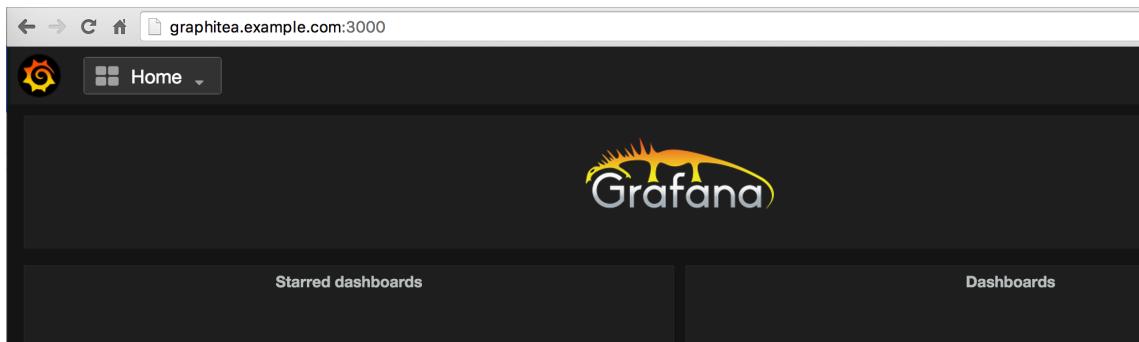


Figure 4.8: The Grafana Console

Next, we need to connect Grafana to our Graphite data via Graphite-API. Remember Graphite-API is running on our Graphite server on port `8888`, so on `graphitea` we find it at `http://graphitea.example.com:8888`. We can use the `curl` command to test this is working.

Listing 4.62: Curling the Graphite-API

```
$ curl http://graphitea.example.com:8888
```

We need to add this as a data source to Grafana by clicking the Grafana logo in the top left of the console to open the menu.

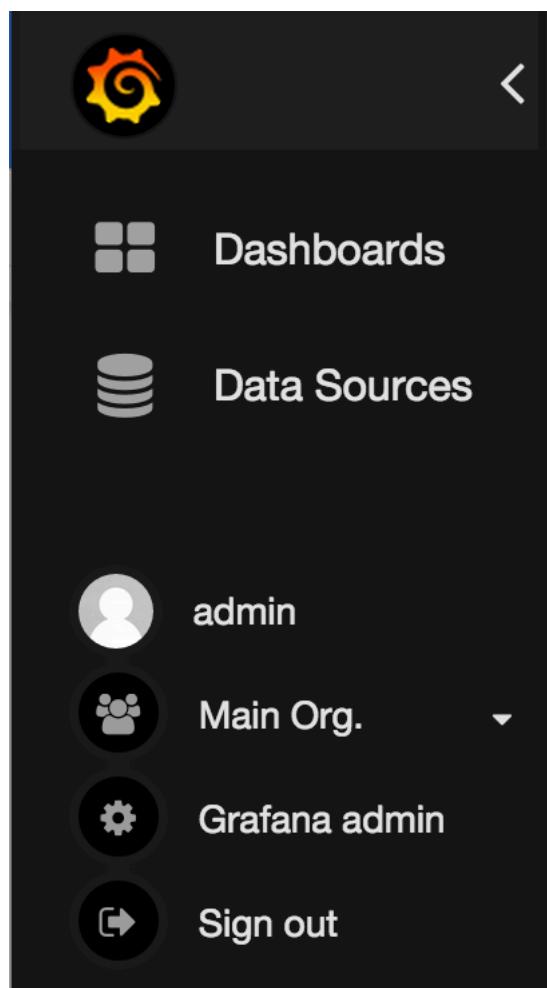


Figure 4.9: The Grafana Console menu

Then we click on the **Data Sources** link, where we'll find an empty list of data sources.

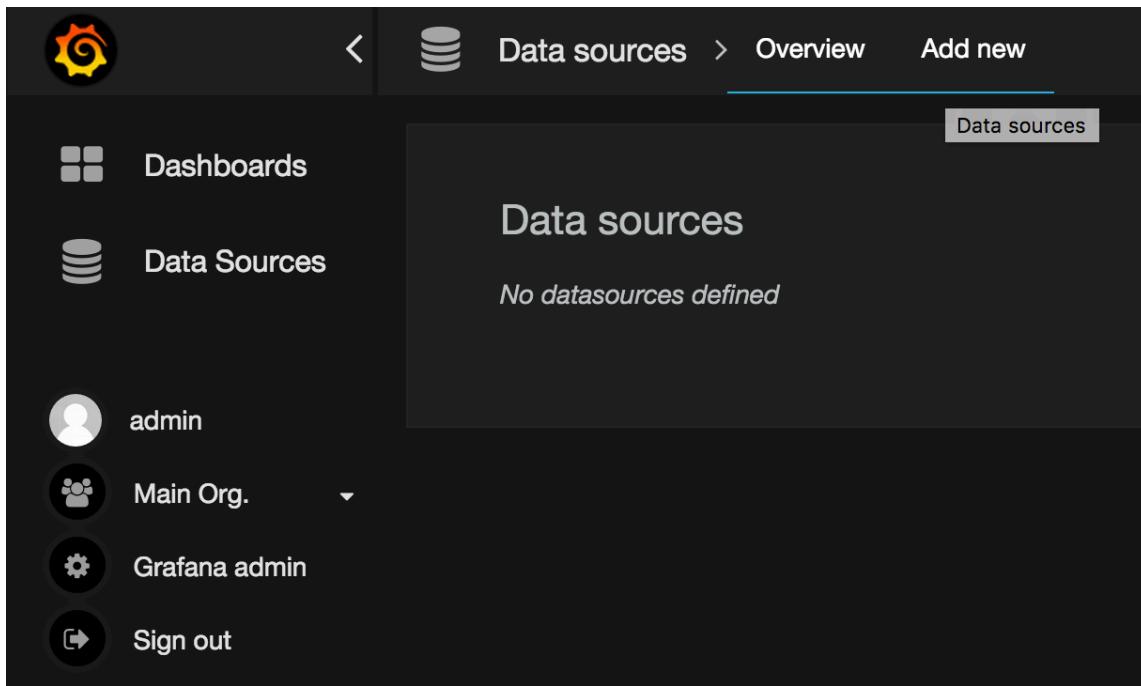


Figure 4.10: The Grafana Data Sources menu

Click on the **Add new** link to add a new 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 **Graphite**. 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 **Graphite**.

Next we need to specify the HTTP settings for our data source. This is the URL of the Graphite-API. 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 the Graphite-API. 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.

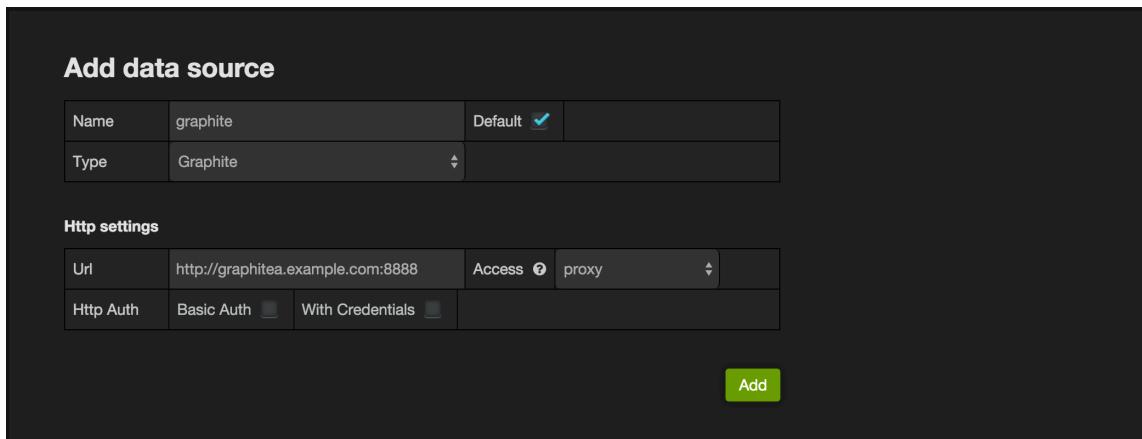


Figure 4.11: Adding a Grafana data source

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. Click the **Test Connection** button to ensure this new connection is working.



Figure 4.12: Testing the Grafana connection

Then click the **Dashboards** to return to the main console view.

As we don't yet have any data in Graphite it would be a bit hard to create graphs right now. Let's connect Riemann and Graphite to get that data flowing, and then we'll come back to Grafana to create some initial graphs.

Configuring Riemann for Graphite

Now that we've got Graphite set up we can start to direct metric events from Riemann to it. To do this we're going to add a Graphite output to Riemann. Riemann has a [native plugin to send events to Graphite](#). We're going to include this plugin in our `/etc/riemann/examplecom/etc` directory as a configuration snippet. This will keep our configuration neat.

Let's create a file to hold our new Graphite plugin snippet.

Listing 4.63: Creating the Riemann Graphite snippet file

```
$ sudo touch /etc/riemann/examplecom/etc/graphite.clj
```

Now let's populate this file.

Listing 4.64: The graphite.clj file

```
(ns examplecom.etc.graphite
  (:require [riemann.graphite :refer :all]
            [riemann.config :refer :all]))

(defn add-environment-to-graphite [event] (str "productiona.hosts"
  .", (riemann.graphite/graphite-path-percentiles event)))

(def graph (async-queue! :graphite {:queue-size 1000}
  (graphite {:host "graphitea" :path add-environment-to
  -graphite})))
```

We first create a namespace, `examplecom.etc.graphite` and then `require` Rie-

mann's Graphite library, `riemann.graphite`. We also require `riemann.config` to make `async-queue!` available in this namespace.

You can see we've added a new function, `add-environment-to-graphite`, and a var called `graph` to Riemann.

The function `add-environment-to-graphite` adds a prefix to any metrics sent to Graphite. Graphite's metric path format is usually in the form of something like:

Listing 4.65: Graphite metrics format

```
prefix.hostname.service.type.of.metric
```

The `add-environment-to-graphite` function prefixes this path on our `riemannna.example.com` host with the string `productiona.hosts..`. This will result in a new Graphite metrics path like so:

Listing 4.66: New Graphite metrics format

```
productiona.hosts.hostname.service.type.of.metric
```

This adds the environment that our host or service runs in to our metrics and helps us namespace those metrics in Graphite. On `riemannb.example.com` we'd add `productionb.hosts.` instead.

TIP Using a configuration management tool you could template our Riemann Graphite configuration to select the right environment depending on which Riemann host you are running on. This would remove the need to hardcode any values. If you didn't want to prefix values you could just omit the function and the `:path` option.

Listing 4.67: The add-environment-to-graphite function

```
(defn add-environment-to-graphite [event] (str "productiona.hosts  
.", (riemann.graphite/graphite-path-percentiles event)))
```

Let's break this function down so we understand how it works. We've named the function `add-environment-to-graphite`, and we're passing in a single argument, `event`. The `event` argument is Riemann's shorthand for the event we're processing, in this case an event we're sending to Graphite.

We then combine the `productiona.hosts.` string with the Riemann Graphite plugin's existing path construction function, `riemann.graphite/graphite-path-percentiles`. The `riemann.graphite/graphite-path-percentiles` function creates a Graphite path for each metric by taking the reversed, fully qualified domain name followed by the service, converting any spaces to dots. The function also converts trailing decimals like `0.95` to `95`.

TIP You can read about the Riemann Graphite plugin's path construction in [the Riemann Graphite documentation](#).

If we then turn to our `graph` var we see how this is combined.

Listing 4.68: The graph var

```
(def graph (async-queue! :graphite {:queue-size 1000}
  (graphite {:host "graphitea" :path add-environment-to
    -graphite})))
```

The `graph` var defines a connection to our Graphite server. We've again used the `async-queue!` stream we saw in Chapter 3. The `async-queue!` stream creates an asynchronous `thread pool` queue. In this case we're using the `async-queue!` to ensure that any issues with our Graphite server do not block Riemann's normal processing. We've called the queue `graphite` and specified a queue size of `1000` events.

Inside our asynchronous queue we've specified the `graphite` plugin and configured it. We've specified the hostname of our Graphite server using the `:host` parameter. Here on `riemannna` we've specified `graphitea`. On `riemannnb` we'd specify `graphiteb`. And on `riemannmc` we'd specify `graphitemc`.

NOTE By default the Graphite plugin uses TCP to send the events but you can also use UDP. However we strongly recommend you don't. UDP isn't a safe protocol for your metrics as it has no guarantees of delivery.

We've also specified the `:path` parameter to tell the Graphite plugin how to construct the metric name, in this case by calling the `add-environment-to-graphite` function we've just created.

NOTE This assumes you've configured DNS or added the various Graphite

servers to `/etc/hosts` or provided some other way for Riemann to resolve the hostnames.

Now that we've added our Graphite plugin, let's go back to the `/etc/riemann/riemann.config` file. We'll use the `graph` var in our streams to send events to Graphite.

Listing 4.69: Riemann Graphite configuration for riemannna

```
(require '[examplecom.etc.graphite :refer :all])  
  
.  
  
(let [index (index  
        downstream (batch 100 1/10  
            (async-queue! :agg { :queue-size      1000  
                                :core-pool-size 4  
                                :max-pool-size 32})  
            (forward  
                (riemann.client/tcp-client :host "riemannmc")))]  
  
(streams  
    (default :ttl 60  
        index  
  
        #(info %)  
  
        (where (service #^"^riemann.*")  
            graph  
  
            downstream))))
```

We first `require` our `examplecom.etc.graphite` function. Then, inside our streams, we use the `graph` var to send events through to Graphite. In our initial configuration we've added the `graph` var inside the `where` stream that sends Riemann events downstream to the `riemannmc` Riemann server.

This will take any Riemann-specific events and send them to Graphite to be used as metrics. Let's restart or reload Riemann to start to send our events to Graphite.

Listing 4.70: Reloading Riemann to enable Graphite

```
$ sudo service riemann reload
```

In our `/var/log/riemann/riemann.log` log file we should see a connection established to our Graphite server.

Listing 4.71: Riemann connecting to Graphite

```
... clojure-agent-send-off-pool-3 - riemann.graphite - Connecting  
to  {:port 2003, :host graphitea}  
... clojure-agent-send-off-pool-1 - riemann.graphite - Connected
```

On our Graphite server, `graphitea`, we should see our Riemann metrics start to be created in the `/var/log/carbon/creates.log` log file.

Listing 4.72: Metrics being created on graphitea

```
14/02/2015 15:28:21 :: new metric productiona.hosts.riemann.  
riemann.streams.latency.95 matched schema default  
14/02/2015 15:28:21 :: new metric productiona.hosts.riemann.  
riemann.streams.latency.95 matched aggregation schema default  
14/02/2015 15:28:21 :: creating database file /var/lib/graphite/  
whisper/productionahosts/riemannstreams/latency/95.  
wsp (archive=[(10, 360), (60, 10080), (900, 2880), (3600,  
17520)] xff=None agg=None)
```

We see that the incoming metrics have been matched against the `default` storage schema that we configured earlier and that each metric is in the form of:

Listing 4.73: metric name formatting

```
productiona.hosts.host.service.type.of.metric
```

Or in the case of a specific metric:

Listing 4.74: The Riemann streams latency path

```
productiona.hosts.riemann.riemann.streams.latency.95
```

Graphite calls this the metric path. The metric path is like a branching tree, with each branch separated by a period.

Listing 4.75: The metric path tree

```
productiona.  
hosts.  
riemann.  
riemann.  
streams.  
latency.  
95
```

Additionally, for each metric, Whisper files are created:

Listing 4.76: Our Riemann Whisper files

```
/var/lib/graphite/whisper/productiona/hosts/riemann/riemann/  
streams/latency/95.wsp
```

You can explore the `/var/lib/graphite/whisper/` directory and see all of our Riemann metrics in the `productiona/hosts/riemann` directory, and all of Carbon's own metrics in the `carbon` directory.

After the initial creation we see each data point added in the `/var/log/carbon/updates.log` log file.

Listing 4.77: Graphite datapoints in the updates log

```
14/02/2015 15:29:22 :: wrote 1 datapoints for productiona.hosts.  
riemann.riemann.streams.latency.0 in 0.00030 seconds  
14/02/2015 15:29:22 :: wrote 1 datapoints for productiona.hosts.  
riemann.riemann.streams.latency.5 in 0.00030 seconds
```

TIP These logs will get auto-rotated daily.

A brief introduction to Grafana

Now that we've got some data in Grafana, let's look at creating our first graph. Browse back to our Grafana console and log in (remember, unless you changed it, the username and password are both **admin**).

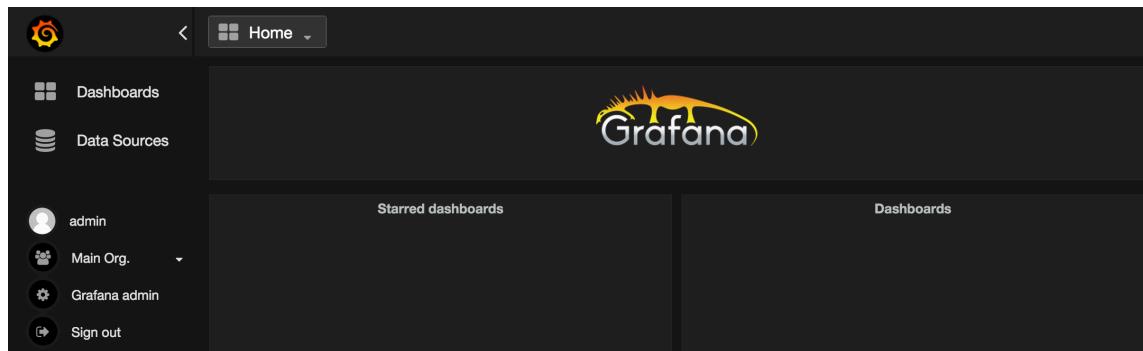


Figure 4.13: The Grafana console again

We're first going to create a new dashboard. Click on the **Home** button to open the menu, and then click the **+ New** button to create a new dashboard.

A new dashboard, titled **New Dashboard**, will be created. You'll see a menu next to the dashboard.

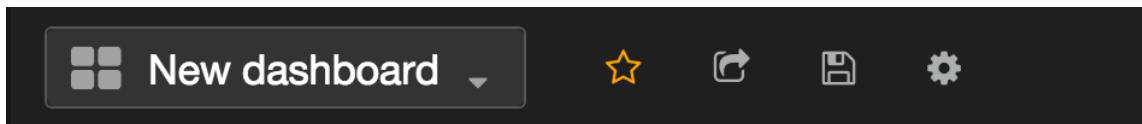


Figure 4.14: The Grafana Dashboard menu

Click on the last circular button to open the **Settings** menu. Inside this menu click on the **Settings** link. This will open the **Settings** sub-menu, where we name and configure our new Dashboard.

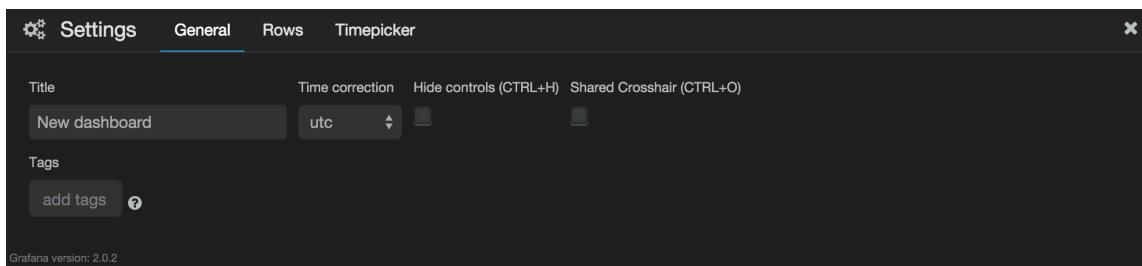


Figure 4.15: The Grafana Settings menu

In the **Title** box we're going to call our dashboard **Riemann**. Update the field and then click the save button in the dashboard menu to save the new dashboard.

Each dashboard is made up of rows, and each row consists of one or more panels. This creates a grid-style dashboard. Inside each row, panels can consist of text, single metrics, graphs, or even lists of other dashboards.

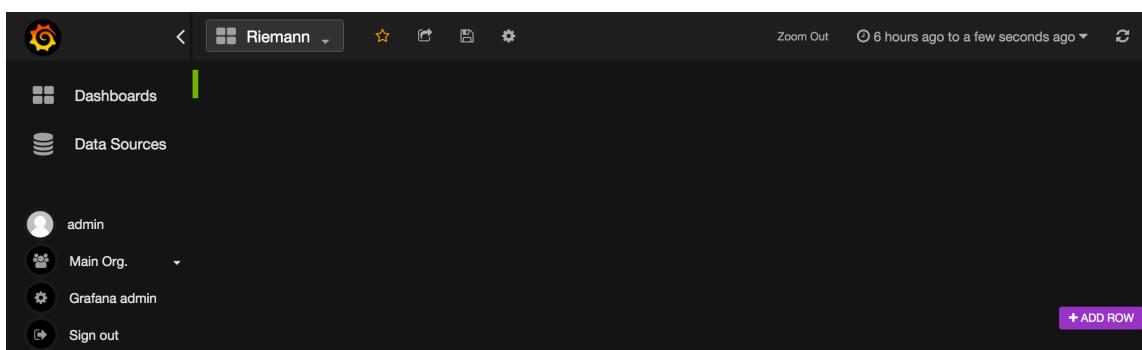


Figure 4.16: The Riemann dashboard menu

Each new dashboard comes with a single auto-created row. You can edit that row by clicking the green bar that opens the Row menu. There are options that allow you to control the location, height, and content of the row. We're going to click the [Add Panel](#) link and then the [Graph](#) link to add our first graph.

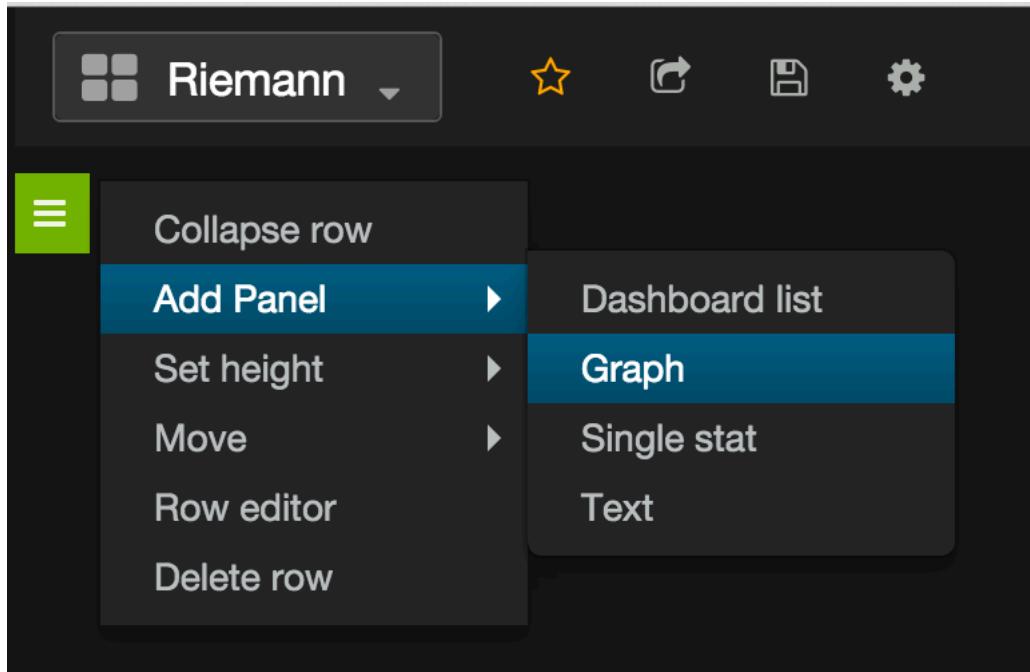


Figure 4.17: Adding our first graph

Our new graph will be the length of the row (the default), have a title of [Panel Title](#), and will be empty because we've not yet specified any data to be graphed.



Figure 4.18: Our first graph

To edit the graph, click on the **Panel Title** title. When the edit menu pops up, click **edit**. This will bring up the graph editing controls.

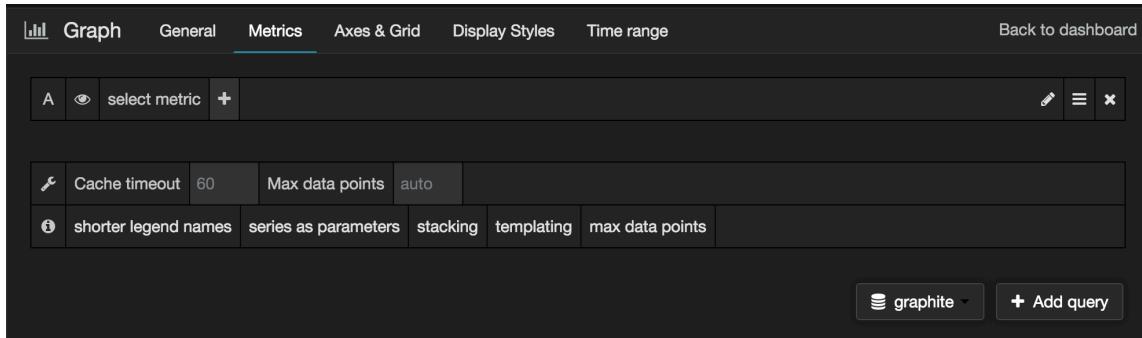


Figure 4.19: The graph editing controls

There's a menu bar at the top of the controls. The first entry, **General**, controls the base configuration and name of our graph. Let's click on that now. On this screen we see the title of the graph. Let's edit it to **Riemann Rate**. We can set the **Span** (how much of the row our graph panel will take up) and **Height**. We can also decorate our graph with a link, perhaps drilling down into another dashboard or even a destination like another service or view.

The other menu items control:

- **Metrics** — The data the graph will display.
- **Axes & Grid** — The display, content, and legend of the graph's axes and grid.
- **Display Styles** — The look and feel of the graph.
- **Time range** — The span of time and any time shift you want to apply.

TIP You can return to the dashboard screen at any time using the Back to dashboard link. If you try to browse away from the dashboard without saving your changes you'll be prompted to discard or save them.

From the **Metrics** menu we can add metrics and data to our graph. We're going to build a graph that shows us the rate of events flowing through our **riemann** Riemann server. To do this we need to select our metrics. There are two ways we can do this:

- Select the metric via the **select metric** box.
- Create a metric via the edit menu (the drop-down icon next to the metrics box).

The first is to click the **select metric** box. This will trigger a query through Grafana to our Graphite-API and into our Graphite data. A list of potential metrics will be returned. The list will be returned as individual metrics broken down by the metric path. We saw that Graphite uses a path-style format to construct metrics. For example, the Riemann rate metric has a path of:

Listing 4.78: The Riemann rate path

```
productiona.hosts.riemann.riemann.streams.rate
```

This will correlate with a file on our **graphitea** host at:

```
/var/lib/graphite/whisper/productionahosts/riemann/riemannstreams/  
rate.wsp
```

Grafana will return the first element of every metric in the path in the **select metric** box, in our case:

Listing 4.79: The first element of the metric path

```
*  
productiona  
carbon
```

This indicates that our Graphite server is storing a series of metrics starting with **productiona** and some starting with **carbon** (these metrics are automatically created by Carbon to help keep track of the Carbon daemons' performance). The ***** is a glob that selects all metrics. This is useful when we might want to say select all hosts, all environments, or the like. If we were to select the ***** here, Grafana would match both **productiona** and **carbon**.

To proceed we select the path element we want (or the ***** glob). In this case we want to select **productiona**, which is the environment we're managing and was created by the **add-environment-to-graphite** function we defined in Riemann earlier.

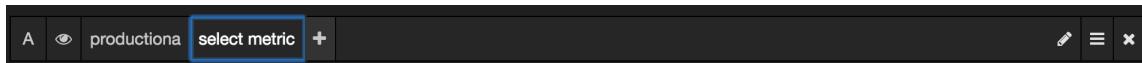


Figure 4.20: Selecting metrics

You can see that it has selected the first element and created a new **select metric** box. If we click this box we'll see the next element in the path and so on until we drill down to the metric we want to graph. When we get down to the last element the metric will be selected and graphed.



Figure 4.21: Our graphed Riemann rate metric

Here we see our graphed `productiona.hosts.riemann.riemann.streams.rate` metric.

Alternatively we can add the metric directly by clicking on the dropdown icon (at the end of the metrics query box) and selecting `Switch editor mode`.

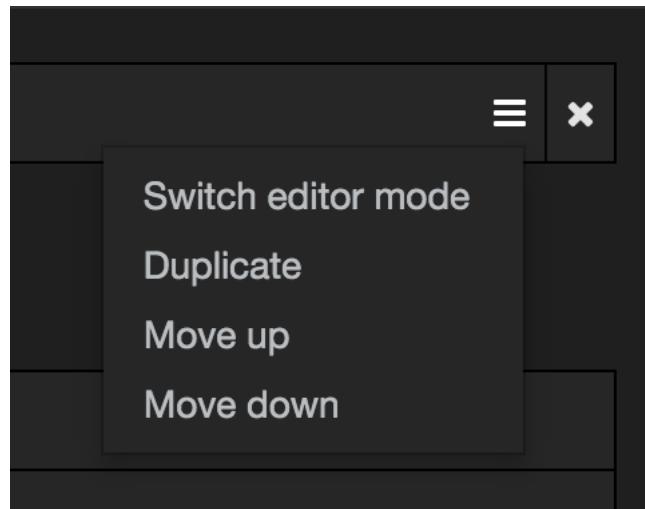


Figure 4.22: The metric dropdown menu

Click this icon and an empty query box will appear. Add your metric, with its full path, to the box and then click outside it to specify the metric.

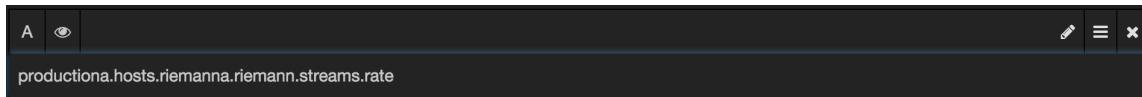


Figure 4.23: Our graphed Riemann rate metric via the edit box

If we then click [Back to dashboard](#) and click the “Save” icon in the top dashboard menu, this will save the new graphs to the dashboard.



Figure 4.24: Our graph and dashboard

This is a quick introduction to Grafana. We’ll see more about how to create graphs and dashboards in subsequent chapters.

Some of the more useful Grafana features worth exploring include:

- [Template and variable-driven](#) dashboards and graphs.
- [Scripted](#) graphs and dashboards.
- [Sharing](#) dashboards and graphs.
- [Playlist](#) dashboards.

TIP You can also find a [Getting Started guide](#), some [useful screencasts](#), and a [reference manual](#) for Grafana that can help you get started.

Graphite and Carbon Redundancy

If we want to provide more redundancy in our metrics storage we can adjust our existing configuration to send our metrics to two hosts. To do this we use the `carbon-relay` daemon. In this approach we'd add a stand-alone `carbon-relay` host and a second graphite server identical to the host we've just created. This stand-alone `carbon-relay` would send a copy of each metric to both hosts.

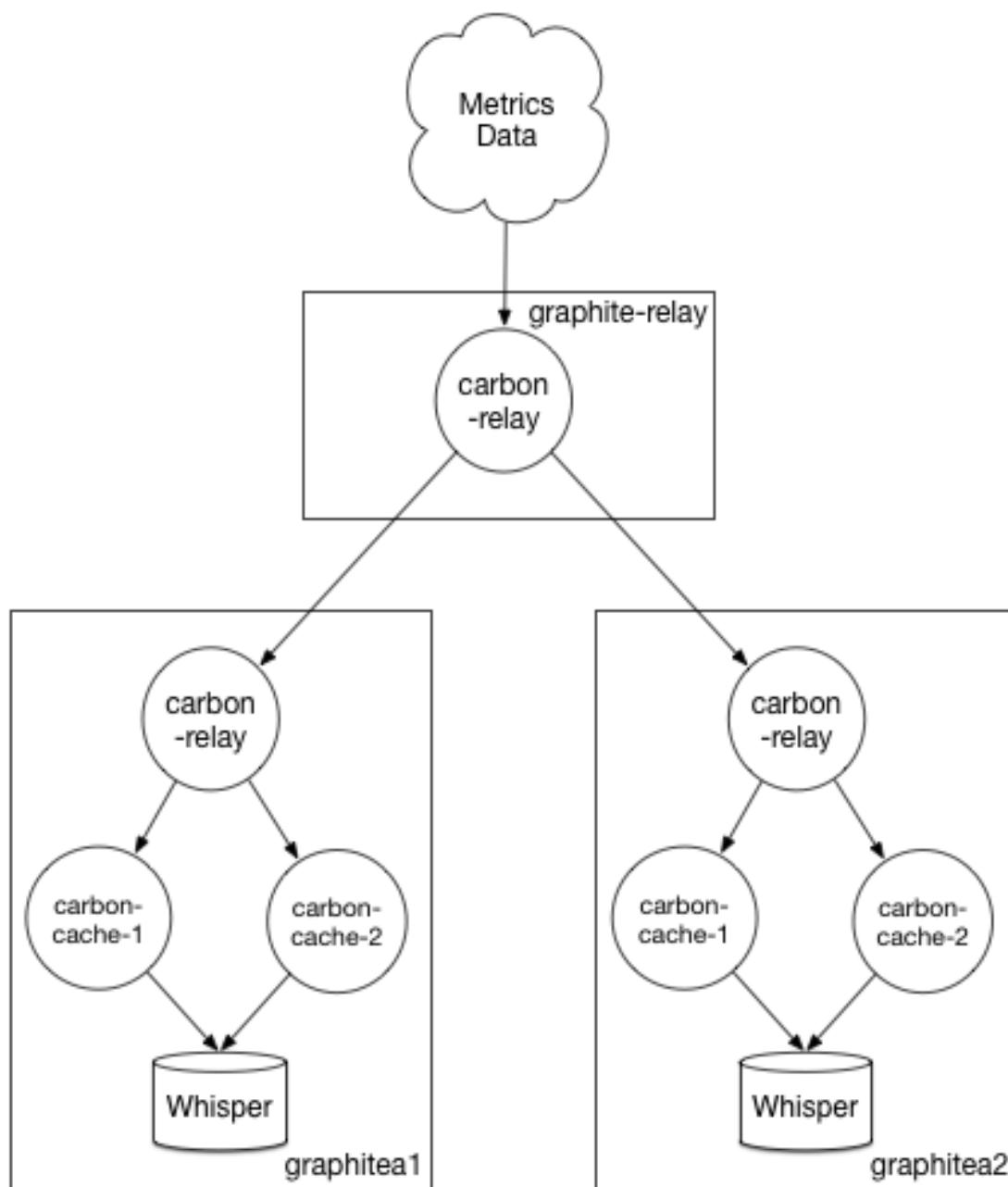


Figure 4.25: A redundant Graphite architecture

Let's look at this in some more detail. First, let's rename our existing **graphitea**

host to `graphitea1`. This let's folks know that we're going to have more than one Graphite server.

Then we configure two new hosts. The first will be an exact replica of our `graphitea1` host. Let's call it `graphitea2` with an IP address of `10.0.0.211`. We'll use the installation instructions from earlier in this chapter to install and configure Graphite and Carbon on the `graphitea2` host.

We'll call the second host `graphite-relay`. We'll follow the above instructions to install Graphite and Carbon but we'd abbreviate them a little because we're not going to be running any caches or web clients on this host.

For example, on Ubuntu, we'd only install the following packages:

Listing 4.80: Installing the Graphite packages on our relay-only host

```
$ sudo apt-get update  
$ sudo apt-get -y install graphite-carbon
```

This is because we only need the Carbon daemons, and more specifically only the `carbon-relay` daemon, rather than a full suite of Graphite and Carbon components.

Our `carbon.conf` would also be abbreviated if we're only running the `carbon-relay` daemon. Let's create that now in `/etc/carbon/carbon.conf`.

Listing 4.81: The carbon-relay only host

```
[relay]
USER = _graphite
LINE_RECEIVER_INTERFACE = 0.0.0.0
LINE_RECEIVER_PORT = 2003
PICKLE_RECEIVER_INTERFACE = 0.0.0.0
PICKLE_RECEIVER_PORT = 2004
RELAY_METHOD = consistent-hashing
REPLICATION_FACTOR = 2
DESTINATIONS = 10.0.0.210:2004, 10.0.0.211:2004
```

We see that the `carbon-relay` daemon is configured similarly to the `carbon-relay` daemons on our `graphitea1` and `graphitea2` hosts. There are a couple of important differences though. We've bound the `LINE_RECEIVER_INTERFACE` interface to all interfaces, we could also bind it just to a specific instance. More importantly, the `graphite-relay` host will write to two `DESTINATIONS`: our original `graphitea1` host on IP address `10.0.0.210` and our new host `graphitea2` on IP address `10.0.0.211`, both using the Pickle receiver port. This is instead of writing to any local `carbon-cache` instances.

You can see that we've again specified the `RELAY_METHOD` as `consistent-hashing` which creates a hash ring of potential destinations and sends metrics to them. This ensures our metrics are balanced. However, it doesn't ensure that our metrics will be duplicated across the two destination hosts. To do that we've adjusted the `REPLICATION_FACTOR` option to `2`. This option tells Carbon how many copies of metrics to distribute. In this case we want two copies, one copy of each metric for each destination, effectively mirroring our metrics.

We'd then configure our `carbon-relay`'s service management and ensure our daemon was running.

On Riemann, we'd update the `graph` var in the `/etc/riemann/examplecom/etc/graphite.clj` file to point to the new `graphite-relay` host.

Listing 4.82: The updated graph var

```
(def graph (graphite {:host "graphite-relay" :path add-
  environment-to-graphite}))
```

Now when we restart Riemann it will write metrics to the `graphite-relay` host. The `graphite-relay` host will write a copy of each metric to the `graphitea1` and `graphitea2` hosts. You should soon be able to check that an identical set of Whisper files exists on both hosts.

NOTE Don't panic if it takes a few minutes for all metrics to appear on both the `graphitea1` and `graphitea2` hosts.

There are some caveats with the resilience of this approach. The first is that the `carbon-relay` host is a single point of failure. If this host stops then metrics stop going through to Graphite (this is also true of Riemann in our current architecture). You can mitigate this with something like [HAProxy](#) if you wish to remove that single point of failure. There's an example of this configuration in [this gist by Jason Dixon](#), and another [sample HAProxy configuration is available on GitHub](#).

Additionally, if you lose either the `graphitea1` or `graphitea2` host and then recover you'll have a gap in the metrics on the host that was lost. There are, however, some useful tools for managing Carbon clusters including [Carbonate](#) which allow you to manage, balance, re-sync, and redistribute metrics and complete other tedious cluster management tasks. This can help you recover from this or

similar issues..

TIP You should also consider backing up your metrics data in the event you need to recover a Graphite host.

It's also important to note that this is just one potential method for clustering Graphite and Carbon. There are a variety of others with varying levels of resilience.

Here are some useful blog posts containing examples of Graphite clustering:

- [Clustering Graphite](#)
- [The architecture of clustering Graphite](#)
- [DynDNS's Graphite clustering configuration including HAProxy](#)

Time and time zones

The last thing we need to care about for our Riemann and Graphite servers is time. As you'd imagine, any event system is heavily dependent on the time being correct and consistent. Clock skew between your hosts will cause issues correlating, indexing, and matching events.

We're going to install a Network Time Protocol or NTP client and use it to get accurate time for all our hosts. We're also going to set our hosts to the UTC time zone. This ensures our Riemann and Graphite hosts are in the same time zone.

You can manage time on your hosts by installing an NTP client manually via a package or by using configuration management. We're going to see both options.

Managing time manually

You can install the NTP packages and set your time zone manually on both Ubuntu and Red Hat.

On Ubuntu

We're going to configure NTP and set our time zone on Ubuntu.

Setting our time zone on Ubuntu

To set our time zone we're going to use the `dpkg-reconfigure` command.

Listing 4.83: Setting the time zone on Ubuntu

```
$ sudo dpkg-reconfigure tzdata
```

A configuration screen will appear. Select `None of the above` and press `Enter`.

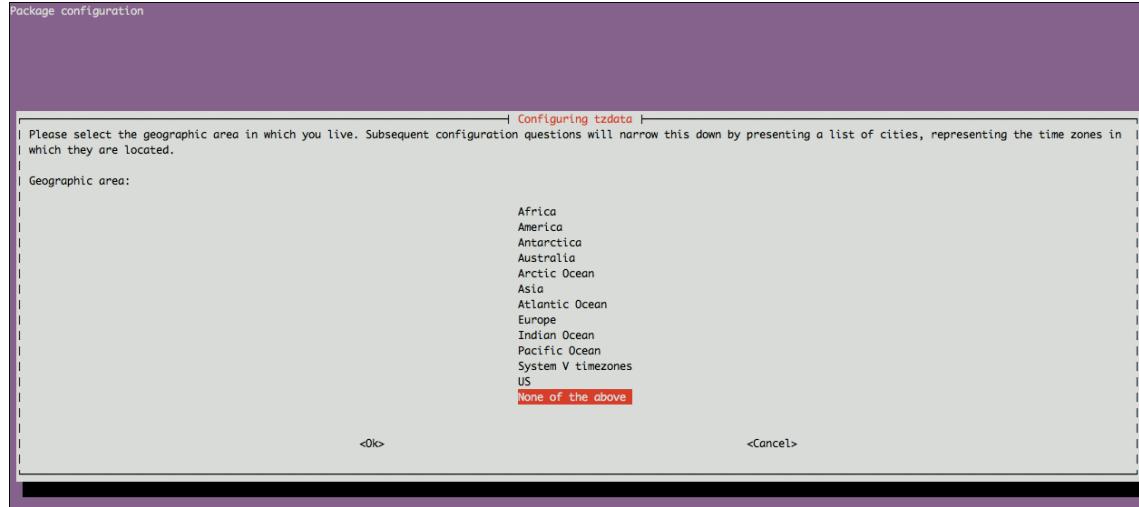


Figure 4.26: Configuring time zone on Ubuntu

On the following screen select **UTC** and hit **Enter** again.



Figure 4.27: Selecting the UTC time zone on Ubuntu

This will select the **Etc/UTC** time zone and put your host into UTC time.

Listing 4.84: The time zone configuration output on Ubuntu

```
Current default time zone: 'Etc/UTC'  
Local time is now:     Mon Feb 16 23:18:09 UTC 2015.  
Universal Time is now: Mon Feb 16 23:18:09 UTC 2015.
```

TIP On Ubuntu 14.04 and later there's also the `timedatectl` command. You could use it like so: `timedatectl set-timezone Etc/UTC`.

Installing NTP on Ubuntu

On Ubuntu we want to install the NTP client and supporting packages.

Listing 4.85: Installing NTP on Ubuntu

```
$ sudo apt-get -y install ntp
```

We then do an initial time synchronization.

Listing 4.86: Initial time sync on Ubuntu

```
$ sudo ntpdate -s ntp.ubuntu.com
```

The NTP daemon will be started automatically as part of the package installation process.

On Red Hat

We're going to configure NTP and set our time zone on Red Hat.

Setting our time zone on Red Hat

On Red Hat we're going to set our time zone with the new `timedatectl` command. This command was added in Red Hat 7 and later releases.

TIP On earlier Red Hat releases see <http://thornelabs.net/2013/04/25/rhel-6-manually-change-time-zone.html>.

The `timedatectl` command can be used with the `set-timezone` options to specify a specific time zone. We're going to set it to `Etc/UTC` or UTC time.

Listing 4.87: Running the timedatectl command on Red Hat

```
$ sudo timedatectl set-timezone Etc/UTC
```

We won't see any output from the command, but we can check it's worked by running the `timedatectl` command on its own.

Listing 4.88: Checking the time zone on Red Hat

```
Local time: Mon 2015-02-16 23:28:47 UTC
Universal time: Mon 2015-02-16 23:28:47 UTC
RTC time: Mon 2015-02-16 23:28:47
Timezone: Etc/UTC (UTC, +0000)
NTP enabled: no
NTP synchronized: no
RTC in local TZ: no
DST active: n/a
```

Installing NTP on Red Hat

On Red Hat we want to install the NTP client and supporting packages.

Listing 4.89: Installing NTP on Red Hat

```
$ sudo yum -y install ntp ntpdate ntp-doc
```

Then enable the NTP service.

Listing 4.90: Enabling the NTP service on Red Hat

```
$ sudo systemctl enable ntpd.service
```

And do an initial time synchronization.

Listing 4.91: Initial time sync on Red Hat

```
$ sudo ntpdate pool.ntp.org
16 Feb 23:26:47 ntpdate[31954]: step time server 199.102.46.73
    offset 0.910678 sec
```

Start the NTP service itself.

Listing 4.92: Starting the NTP service on Red Hat

```
$ sudo service ntpd start
```

Finally, let's ensure everything knows about our NTP server by again running the `timedatectl` command, this time with the `set-ntp` flag to turn on NTP.

Listing 4.93: Enabling NTP with timedatectl

```
$ sudo timedatectl set-ntp true
```

If we run `timedatectl` again we should see NTP is enabled and working.

Listing 4.94: Checking NTP is enabled with timedatectl

```
$ sudo timedatectl
    Local time: Mon 2015-02-16 23:30:31 UTC
    Universal time: Mon 2015-02-16 23:30:31 UTC
        RTC time: Mon 2015-02-16 23:30:31
      Timezone: Etc/UTC (UTC, +0000)
     NTP enabled: yes
    NTP synchronized: yes
      RTC in local TZ: no
       DST active: n/a
```

Managing Time via configuration management

There are a variety of options for managing the time on your hosts via configuration management.

You can find a Chef cookbook for NTP at:

- <https://supermarket.chef.io/cookbooks/ntp>.

You can find a Puppet module for NTP at:

- <https://forge.puppetlabs.com/puppetlabs/ntp>.

You can find an Ansible role for NTP at:

- <https://galaxy.ansible.com/list#/roles/464>.

TIP We recommend using one of these to manage the time on all your hosts rather than configuring time manually.

Checking the time status

We then check the status of our NTP service using the `ntpq` command or NTP query program..

Listing 4.95: Checking the NTP time status

```
$ sudo ntpq -p
```

A list will be generated showing that we've connected to several NTP hosts. It includes the `when` column that tells us the time in seconds since we last synched with each of them for updated time.

NOTE It's also possible to report on NTP's statistics via Riemann or [collectd](#) using the [NTPd plugin](#).

Alternatives to Graphite and Grafana

There are numerous alternatives to Graphite, both commercial and open source. This is not a definitive list but more a sampling of interesting tools you could use if the choices in this book aren't suitable or to your taste.

Commercial tools

A number of commercial Software-as-a-Service (SaaS) products exist including:

- [Circonus](#)
- [Geckoboard](#)
- [Leftronic](#)
- [Librato](#)
- [New Relic](#)
- [SignalFx](#)

NOTE Some of these SaaS tools may do more than just metrics and may have monitoring, application performance management, and alerting capabilities.

Open-source tools

Open-source products. Some tools include collection and storage but others, like D3, only provide visualization and need to be combined with appropriate collection mechanisms.

Storage

- [Druid](#) — Druid is a distributed, real-time analytics data store.
- [OpenTSDB](#) — OpenTSDB is a distributed metrics store that uses Hadoop and HBase.

Visualization and analysis

- [D3](#) — A Javascript library for data visualization.
- [Graphene](#) — A dashboard for Graphite data.
- [Rickshaw](#) — A Javascript library for data visualization.
- [Tessera](#) — An alternative dashboard for Graphite.
- [Facette](#) — A multi-data source dashboard written in Go.
- [Dusk](#) — A hot spot detection dashboard for Graphite that uses D3.
- [Graph Explorer](#) — A Graphite dashboard written by the team at Vimeo.
- [Giraffe](#) — A Graphite dashboard.

There's also a great list of tools that integrate or are related to Graphite at <http://graphite.readthedocs.org/en/latest/tools.html>.

Whisper alternatives

An alternative to addressing concerns about Whisper performance is to replace Whisper entirely. There are a couple of alternatives to using Whisper:

- InfluxDB
- Cyanite

We're not going to cover either of them in much detail but it's important to know they exist.

InfluxDB

The first of these options is [InfluxDB](#). InfluxDB is a time-series database that's written in Go. InfluxDB has some embryonic support for ingesting Graphite metrics and a new clustering system designed to make it more resilient. It has some integration with Graphite Web, Grafana, and similar web interfaces. InfluxDB is a relatively new tool—all its features may not be available, and it may not be production ready yet.

NOTE There is also a [Riemann-to-InfluxDB plugin available](#).

Cyanite

Another option is to use a [Cassandra](#) database using an integration called [Cyanite](#). Cyanite is written in Clojure and exposes a Carbon daemon that allows you to send metrics to a Cassandra cluster. Given the complexity of Cassandra, however, it's not a drop-in replacement.

Summary

In this chapter we learned how to install, configure, and run Graphite and Grafana. We gathered events in Riemann, and we've sent those events in the form of metrics from Riemann to Graphite. We also created our first Grafana dashboard and added our first graph to that dashboard.

We configured NTP to manage the time on our Riemann and Graphite hosts. With NTP, we can ensure the time stamps of our events are consistent between Riemann and Graphite. To help with this we also configured all the hosts' time zones to be

UTC time.

In the next chapters we're going to add more components to our monitoring framework: host-level metrics, logging, and better notifications. We're then going to see how we can use that framework to start monitoring a series of components, for both hosts and applications. We're going to send the metrics and events from the components we're monitoring to Riemann, store them in Graphite, and graph them in Grafana. In the next chapter we'll start with collecting and graphing host-based metrics.

Chapter 5

Host monitoring

In the last two chapters we installed and configured Riemann and Graphite. We got an introduction to Riemann and how it manages and indexes events. We also saw how to integrate Riemann servers in multiple environments. Finally, we saw how to send events from Riemann to Graphite and then graph them in Grafana.

In this chapter we're going to add another building block to our monitoring framework by collecting host-based data and sending it to Riemann. Host-based monitoring provides the basic information about our hosts and their performance. We can then collect and combine this data with the application data that we'll learn to collect in Chapter 9.

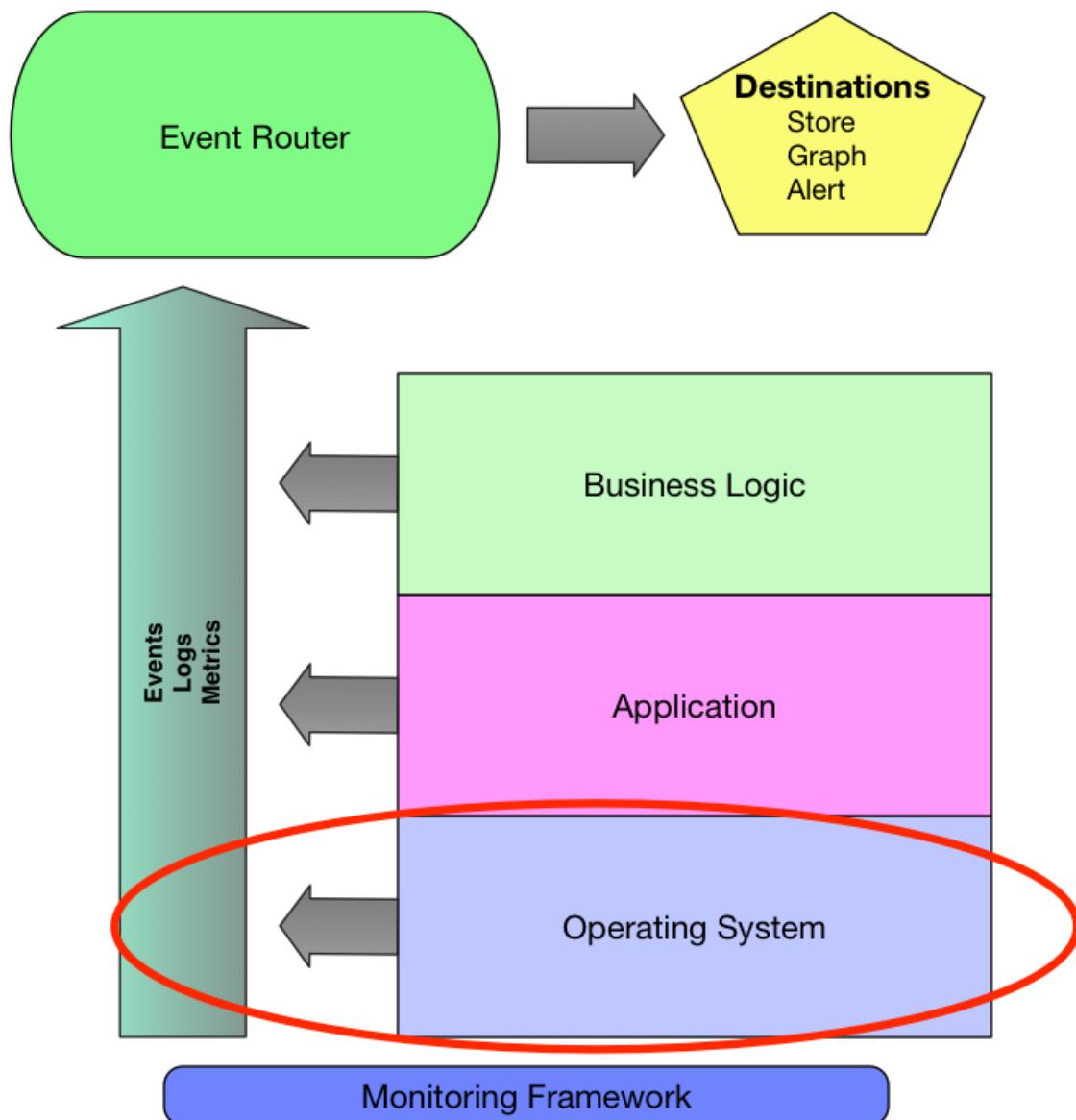


Figure 5.1: Collecting host-based data

NOTE We'll also look at container monitoring, focusing on Docker containers in Chapter 7.

We'll aim to implement host monitoring that:

- Is a scalable and high-performing solution. Our collection needs to be lightweight and not interfere with the actual running of our hosts and applications (refer to the observer effect we talked about in Chapter 2).
- Can ship data quickly and avoid queuing important information that we need.
- Has a flexible monitoring interface that can collect a wide variety of data “out of the box” but also allows us to collect custom data that is less common or unique to our environment.
- Accommodates our push-versus-pull architecture.

To satisfy these needs we're going to look at a tool called [collectd](#).

Introducing collectd

The collectd daemon, which acts as a monitoring collection agent, will do the local monitoring and send the data to Riemann. It will run locally on our hosts and selectively monitor and collect data from a variety of components.

We've chosen collectd because it is high performance and reliable. The collectd daemon has been around for about ten years, it's written in C for performance, and is well tested and widely used. It's also open source and licensed under a mix of the MIT and GPLv2 licenses.

WARNING This book assumes you're running collectd 5.5 or later. Earlier versions may work with this configuration but some components may not behave exactly as described.

The daemon uses a modular design: a central core and an integrated plugin system. The core of collectd is small and provides basic filtering and routing for the collected data. The collection, storage, and transmission of data is handled by plugins that can be enabled individually.

For the collection of data, collectd uses plugins called read plugins. They can collect information like CPU performance, memory, or application-specific metrics. This data is then passed into collectd's core functions.

The data can then be filtered or routed to write plugins. The write plugins allow data to be stored locally—for example, writing to files—or to send that data across the network to other services. In our case, they'll allow us to send that data to Riemann.

There are [a large collection of default plugins](#) shipped with collectd, as well as a variety of community-contributed and developed plugins you can add to it. The collectd daemon also supports running plugins you have written yourself.

This diagram shows our collectd architecture.

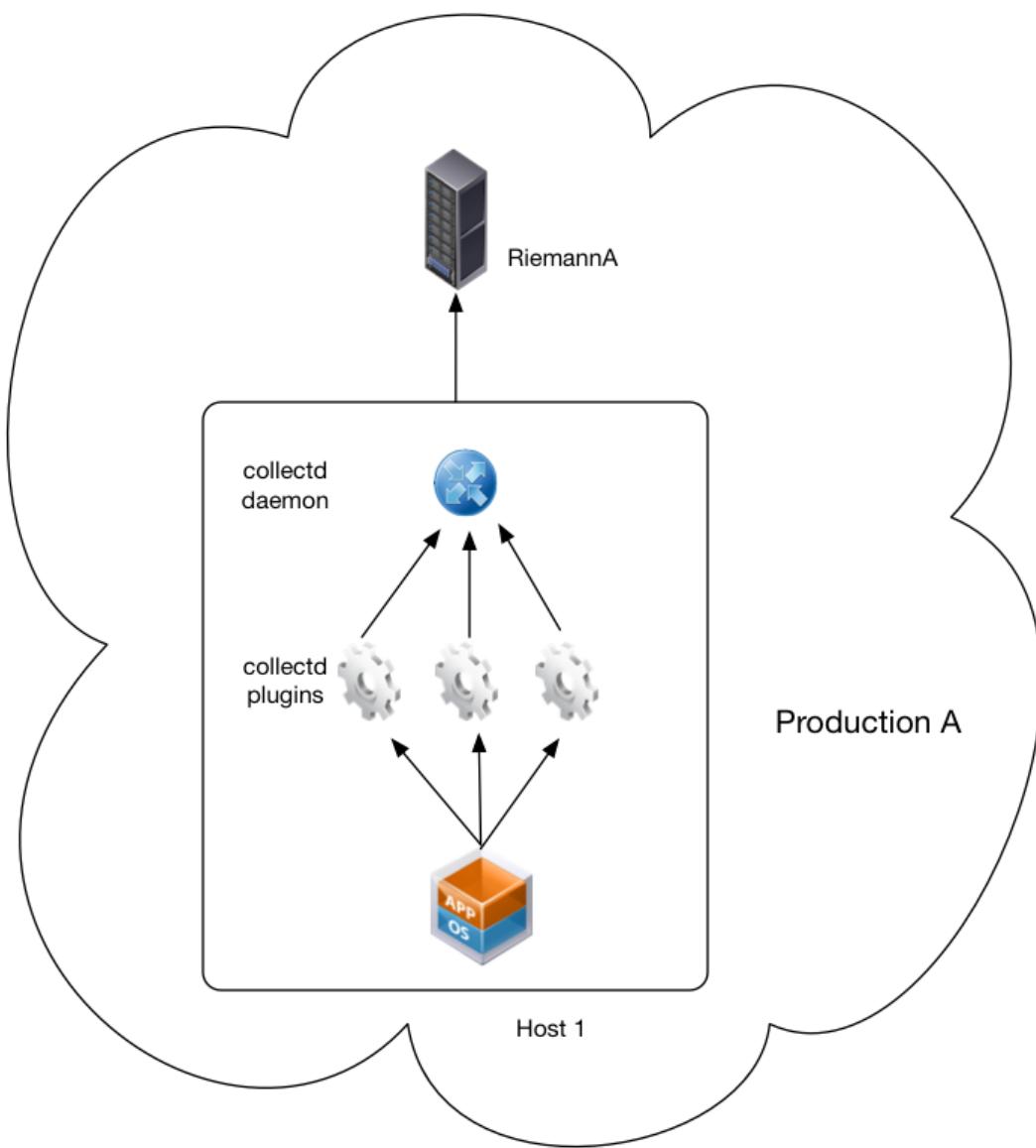


Figure 5.2: Our collectd architecture

Our applications and services are connected via collectd's read plugins. These read plugins send events to a write plugin that will send events onto Riemann.

In this chapter we're focusing on using collectd for host-based monitoring, but

we're also going to use it later in the book for collecting service and application data. Using collectd will allow us to run a single agent locally on our host and use it to send our data into our Riemann event router.

What host components should we monitor?

Before we jump into installing and configuring collectd, let's discuss what we might like to collect on our hosts. We're going to focus on collecting basic data that will show us the core performance of our hosts. We're going to configure a generic collection of monitoring data on all our hosts, and we'll add additional monitoring for specific use cases. For example, we would install our base monitoring on all hosts but add specific monitoring for a database or application server.

Our basic set of monitoring will include:

- CPU - Shows us how our host is running workloads.
- Memory - Show us how much memory is being used and is available on our hosts.
- Load - The “system load.” A rough overview of host utilization, defined as the number of runnable tasks in the run-queue in one, five, and fifteen-minute averages.
- Swap - Shows us how much swap is being used and is available on our hosts.
- Processes - Monitor both specific processes and processes counts, and identify “[zombie](#)” processes.
- Disk - Shows us how much disk space is being used and is available on our filesystems.
- Network - Shows us the basic performance of our interfaces and networks, including errors and traffic.

This base set of data will allow us to identify host performance issues or provide sufficient supplemental data for fault diagnosis of application issues.

At this point some of you may be wondering, “But didn’t we say earlier that we should focus on application and business events and metrics?” We’re indeed going to focus on those application and business events and metrics, but they aren’t the full story. When we have a fault or need to further diagnose a performance issue we often need to drill down to more granular data. The data we’re gathering here will supplement our application and business events and metrics and allow us to diagnose and identify system-level issues that cause application problems.

NOTE Two topics we’re not covering directly in the book are monitoring of Microsoft Windows and monitoring of non-host devices: networking equipment, storage devices, data center equipment. We will discuss some options for this later in the chapter.

Installing collectd

We’ll install collectd on every host and configure it to collect our metrics and send them to Riemann. We’ll walk through the installation process on both Ubuntu and Red Hat family operating systems and provide you with appropriate configuration management resources to do the installation automatically.

NOTE It’s important to make a (slightly pedantic) distinction about polling versus pushing here. Technically a client like collectd is polling on the local host, but it is polling locally and then pushing the events to Riemann. This local polling scales differently from centrally scheduled and initiated polling—for example, from a monitoring server to many hosts and services.

Installing collectd on Ubuntu

We'll install version 5.5 of collectd on Ubuntu from the collectd project's own Landscape repositories and via the `apt-get` command.

Add the collectd repository configuration to your host:

Listing 5.1: Installing collectd on Ubuntu

```
$ sudo add-apt-repository ppa:collectd/collectd-5.5
```

Then update and install collectd.

Listing 5.2: Installing collectd on Ubuntu

```
$ sudo apt-get update  
$ sudo apt-get -y install collectd
```

The collectd daemon and its associated dependencies will now be installed. We'll test it is installed and working by running the `collectd` binary.

Listing 5.3: Testing collectd on Ubuntu

```
$ collectd -h  
Usage: collectd [OPTIONS]  
  
...  
...
```

The `-h` flag will output collectd's flags and command line help.

Installing collectd on Red Hat

We'll install collectd on Red Hat using the EPEL repository.

Let's add the EPEL repository now.

Listing 5.4: Adding the EPEL repository for Riemann

```
$ sudo yum install -y epel-release
```

Now we'll use the `yum` command to install the `collectd` package. We'll also install the `write_riemann` plugin, which we'll use to send our events to Riemann, and the `protobuf-c` package it requires.

Listing 5.5: Installing collectd on Red Hat

```
$ sudo yum install collectd protobuf-c collectd-write_riemann
```

TIP On newer Red Hat and family versions the `yum` command has been replaced with the `dnf` command. The syntax is otherwise unchanged.

The `collectd` daemon and its associated dependencies will now be installed. We'll test it is installed by running the `collectd` binary.

Listing 5.6: Testing collectd on Red Hat

```
$ collectd -h
Usage: collectd [OPTIONS]
      . . .
```

The `-h` flag will output collectd's flags and command line help.

Installing collectd via configuration management

There are a variety of options for installing collectd on your hosts via configuration management.

You can find a Chef cookbook for collectd at:

- <https://supermarket.chef.io/cookbooks?q=collectd>.

You can find Puppet modules for collectd at:

- <https://forge.puppetlabs.com/modules?sort=rank&q=collectd>.

You can find an Ansible role for collectd at:

- <https://galaxy.ansible.com/list#/roles/350>.

You can also find Docker containers at:

- <https://hub.docker.com/search?q=collectd>.

And a Vagrant image at:

- <https://github.com/httpdss/collectd-web-vagrant>.

TIP We recommend using one of these methods to manage collectd on your hosts rather than configuring collectd manually.

Configuring collectd

Now that we have the collectd daemon installed we need to configure it. We're going to enable some plugins to gather our data and then configure a plugin to send the events to Riemann.

The collectd daemon is configured using a configuration file located at:

- `/etc/collectd/collectd.conf` on Ubuntu.
- `/etc/collectd.conf` on Red Hat.

The `collectd` package on both distributions installs a default configuration file. We're not going to use that file—instead, we're going to build our own configuration.

The collectd configuration is divided into three main concerns:

- Global settings for the daemon.
- Plugin loading.
- Plugin configuration.

Let's look at our initial configuration file.

Listing 5.7: Our initial collectd configuration

```
# Global settings
Interval 2
CheckThresholds true
WriteQueueLimitHigh 5000
WriteQueueLimitLow 5000

# Plugin loading
LoadPlugin logfile
LoadPlugin threshold

# Plugin Configuration
<Plugin "logfile">
    LogLevel "info"
    File "/var/log/collectd.log"
    Timestamp true
</Plugin>

Include "/etc/collectd.d/*.conf"
```

TIP The collectd configuration files lend themselves to being turned into configuration management templates. Indeed, many of the collectd modules for various configuration management tools include templating for collectd configuration.

First, we need to set two global configuration options in our collectd configuration. The first configuration option, **Interval**, sets the collectd daemon's check

interval. This is the resolution at which collectd collects data. It defaults to 10 seconds. We're going to make our resolution more granular and move it to two second intervals. This should either match or be slightly larger than the lowest retention period set in Graphite to ensure the collection periods sync up correctly. Remember that in Chapter 4 we installed Graphite and configured its retentions in the `/etc/carbon/storage-schemas.conf` configuration file.

TIP We could set the `Interval` to one second but Riemann's one second precision sometimes means duplicate metrics are generated when Riemann rounds the time to the nearest second.

Listing 5.8: The Graphite storage schema

```
[default]
pattern = .*
retentions = 1s:24h, 10s:7d, 1m:30d, 10m:2y
```

You can see our shortest resolution in the storage schema is one second. Our collectd `Interval` should be the same or more than that resolution.

WARNING As we discussed in Chapter 4, if you change the resolution of your collection, then it will be difficult to compare data collected during periods with the previous resolution.

The second global option, `CheckThresholds`, can be set to `true` or `false`, and con-

trols how state is set for collected data. The collectd daemon can check collected data for thresholds, marking data with metrics that exceed those thresholds as being in a warning or failed state. We're not going to set any specific thresholds right now, but we're turning on the functionality because it creates a default state—shown in the `:state` field in a Riemann event—of `ok` that we can later use in Riemann.

The `WriteQueueLimitHigh` and `WriteQueueLimitLow` options control the queue size of write plugins. This protects us from memory issues if a write plugin is slow, for example if the network times out. You can set the limits using the `WriteQueueLimitHigh` and `WriteQueueLimitLow` options. Each option takes a number, which is the number of metrics in the queue. If there are more metrics than the value of the `WriteQueueLimitHigh` options then any new metrics will be dropped. If there are less than `WriteQueueLimitLow` metrics in the queue, all new metrics will be enqueued.

If the number of metrics currently in the queue is between the two thresholds the metric is potentially dropped, with a probability that is proportional to the number of metrics in the queue. This is a bit random for us so we set both options to `5000`. This sets an absolute threshold. If more than `5000` metrics are in the queue then incoming metrics will be dropped. This protects the memory on the host running collectd.

Next, we'll enable some plugins for collectd. Use the `LoadPlugin` command to load each plugin.

Listing 5.9: Loading collectd plugins

```
LoadPlugin logfile

<Plugin "logfile">
  LogLevel "info"
  File "/var/log/collectd.log"
  Timestamp true
</Plugin>

LoadPlugin threshold
```

The first plugin, `logfile`, tells collectd to log its output to a log file.

We've configured the plugin using a `<Plugin>` block. Each `<Plugin>` block specifies the plugin it is providing configuration for, here `<Plugin "logfile">`. It also contains a series of configuration options for that plugin. In this case, we've specified the `LogLevel` option, which tells collectd how verbose to make our logging. We've chosen a middle ground of `info`, which logs some operational and error information, but skips a lot of debugging output. We've also specified the `File` option to tell where collectd to log this information, choosing `/var/log/collectd.log`.

TIP We can also log in Logstash's JSON format using the [log_logstash plugin](#). See Chapter 8 for more information on logging and Logstash.

Lastly, we've set the `Timestamp` option to `true` which adds a timestamp to any log output collectd generates.

TIP We've specified the `logfile` plugin and its configuration first in the `collectd.conf` file so that if something goes wrong with `collectd` we're likely to catch it in the log.

We'll also need to load the `threshold` plugin. The `threshold` plugin is linked to the `CheckThresholds` setting we configured in the global options above. This plugin [enables collectd's threshold checking logic](#). We're going to use it to help identify when things go wrong on our host.

Finally, we'll set one last global configuration option, `Include "/etc/collectd.d/*.conf"`. The `Include` option specifies a directory to hold additional `collectd` configuration. Any file ending in `.conf` will be added to our `collectd` configuration. We're going to use this capability to make our configuration easier to manage. With snippets included via the `Include` directory, we can easily manage `collectd` with configuration management tools like Puppet, Chef, or Ansible.

You'll note that we've put the `Include` option at the end of the configuration file. This is because `collectd` loads configuration in a top down order. We want our included files to be done last.

Let's create the directory for the files we wish to include now (it already exists on some distributions).

Listing 5.10: Creating the /etc/collectd.d directory

```
$ sudo mkdir /etc/collectd.d
```

TIP You can find a list of other global configuration options on the [collectd wiki](#).

Loading and configuring collectd plugins for monitoring

We're now going to load and configure a series of plugins to collect data. We're going to make use of the following plugins:

- `cpu` - The CPU plugin collects the amount of time spent by the CPU in various states.
- `df` - The DF plugin collects filesystem usage information, so named because it returns similar data to the `df` command.
- `load` - The Load plugin collects the system load.
- `interface` - The Interface plugin collects network interface statistics.
- `protocols` - The Protocols plugin records information about network protocol performance and data on the host, including TCP, UDP, and ICMP traffic.
- `memory` - The Memory plugin collects physical memory utilization.
- `processes` - The Processes plugin collects the number of processes, grouped by state: running, sleeping, zombie, etc.
- `swap` - The Swap plugin collects the amount of memory currently written to swap.

These plugins provide data for the basic state of most Linux hosts. We're going to configure each plugin in a separate file and store them in the `/etc/collectd.d/` directory we've specified as the value of the `Include` option. This separation allows us to individually manage each plugin and lends itself to management with a configuration management tool like Puppet, Chef, or Ansible.

Now let's configure each plugin.

The cpu plugin

The first plugin we're going to configure is the `cpu` plugin. The `cpu` plugin collects CPU performance metrics on our hosts. By default, the `cpu` plugin emits CPU metrics in [Jiffies](#): the number of ticks since the host booted. We're going to also send something a bit more useful: percentages.

First, we're going to create a file to hold our plugin configuration. We'll put it into the `/etc/collectd.d` directory. As a result of the [Include](#) option in the `collectd.conf` configuration file, collectd will load this file automatically.

Listing 5.11: Creating the `/etc/collectd.d/cpu.conf` file

```
$ sudo touch /etc/collectd.d/cpu.conf
```

Let's now populate this file with our configuration.

Listing 5.12: Configuring the `cpu` plugin

```
LoadPlugin cpu
<Plugin cpu>
  ValuesPercentage true
  ReportByCpu false
</Plugin>
```

We load the plugin using the `LoadPlugin` option. We then specify a `<Plugin>` block to configure our plugin. We've specified two options. The first, `ValuesPercentage`, set to `true`, will tell collectd to also send all CPU metrics as percentages. The second option, `ReportByCpu`, set to `false`, will aggregate all CPU cores on the host into a single metric. We like to do this for simplicity's sake. If you'd prefer your

host to report CPU performance per core, you can change this to `true`.

The memory plugin

Next let's configure the `memory` plugin. This plugin collects information on how much RAM is free and used on our host. By default the `memory` plugin returns metrics in bytes used or free. Often this isn't useful because we don't know how much memory a specific host might have. If we return the value as a percentage instead, we can more easily use this metric to determine if we need to take some action on our host. So we're going to do the same percentage conversion for our memory metrics.

Let's start by creating a file to hold the configuration for the `memory` plugin.

Listing 5.13: Creating the `/etc/collectd.d/memory.conf` file

```
$ sudo touch /etc/collectd.d/memory.conf
```

Now let's populate this file with our configuration.

Listing 5.14: Configuring the Memory plugin

```
LoadPlugin memory
<Plugin memory>
    ValuesPercentage true
</Plugin>
```

We first load the plugin and then add a `<Plugin>` block to configure it. We set the `ValuesPercentage` option to `true` to make the `memory` plugin also emit metrics as percentages.

NOTE It'll continue to emit metrics in bytes, too, in a separate metric so we can still use these if required.

The df plugin

The `df` plugin collects disk space metrics, including space used, on mount points and devices. Like the `memory` plugin, it outputs in bytes by default—so let's configure it to emit metrics in percentages, instead. That will make it easier to determine if a mount or device has a disk space issue.

Let's start by creating a configuration file.

```
Listing 5.15: Creating the /etc/collectd.d/df.conf file
```

```
$ sudo touch /etc/collectd.d/df.conf
```

And now we'll populate this file with our configuration.

```
Listing 5.16: Configuring the df plugin
```

```
LoadPlugin df
<Plugin df>
  MountPoint "/"
  ValuesPercentage true
</Plugin>
```

We first load the plugin and then configure it. The `ValuesPercentage` option tells

the `df` plugin to also emit metrics with percentage values. We've also specified a mount point we'd like to monitor via the `MountPoint` option. This option allows us to specify which mount points to collect disk space metrics on. We've only specified the `/` ("root") mount point. If we wanted or needed to, we could add additional mount points to the configuration now. To monitor a mount point called `/data` we would add:

Listing 5.17: Configuring another mount point for the df plugin

```
LoadPlugin df
<Plugin df>
  MountPoint "/"
  MountPoint "/data"
  ValuesPercentage true
</Plugin>
```

We can also specify devices using the `Dev` option.

Listing 5.18: Configuring the df plugin to monitor a device

```
LoadPlugin df
<Plugin df>
  MountPoint "/"
  Dev "/dev/hda1"
  ValuesPercentage true
</Plugin>
```

Or we can monitor all filesystems and mount points on the host.

Listing 5.19: Configuring the df plugin to monitor everything

```
<Plugin df>
  IgnoreSelected true
  ValuesPercentage true
</Plugin>
```

The `IgnoreSelected` option, when set to `true`, tells the `df` plugin to ignore all configured mounts points or devices and monitor every mount point and device.

The swap plugin

Let's now configure the `swap` plugin. This plugin collects a variety of metrics on the state of swap on your hosts. Like other plugins we've seen that it returns metric values measuring swap used in bytes. We again want to make those easier to consume by emitting them as percentage values.

Let's first create a file to hold its configuration.

Listing 5.20: Creating the /etc/collectd.d/swap.conf file

```
$ sudo touch /etc/collectd.d/swap.conf
```

Now we'll populate this file with our configuration.

Listing 5.21: Configuring the swap plugin

```
LoadPlugin swap
<Plugin swap>
  ValuesPercentage true
</Plugin>
```

We've only specified one option, `ValuesPercentage`, and set it to `true`. This will cause the `swap` plugin to emit metrics using percentage values.

The interface plugin

Now let's configure our `interface` plugin. The `interface` plugin collects data on our network interfaces and their performance.

Listing 5.22: Creating the /etc/collectd.d/interface.conf file

```
$ sudo touch /etc/collectd.d/interface.conf
```

We're not going to add any configuration to the `interface` plugin by default so we're just going to load it.

Listing 5.23: The interface.conf file

```
LoadPlugin interface
```

Without configuration, the `interface` plugin collects metrics on all interfaces on the host. If we wanted to limit this to one or more interfaces, instead, that can be

specified via configuration using the `Interface` option.

Listing 5.24: Configuring the interface plugin to only monitor one interface

```
<Plugin interface>
  Interface "eth0"
</Plugin>
```

Other times we might want to exclude specific interfaces—for example, the loopback interface—from our monitoring.

Listing 5.25: Ignoring interfaces

```
<Plugin "interface">
  Interface "lo"
  IgnoreSelected true
</Plugin>
```

Here we've specified the loopback, `lo`, interface with the `Interface` option. We've added the `IgnoreSelected` option and set it to `true`. This will tell the `interface` plugin to monitor all interfaces except the `lo` interface.

The protocols plugin

Also collecting data about our host's network performance is the `protocols` plugin. The `protocols` plugin collects data on our network protocols running on the host and their performance.

Listing 5.26: Creating the /etc/collectd.d/protocols.conf file

```
$ sudo touch /etc/collectd.d/protocols.conf
```

We're not going to add any configuration to the `protocols` plugin by default so we're just going to load it.

Listing 5.27: The protocols.conf file

```
LoadPlugin protocols
```

The load plugin

We're not going to add any configuration for the next plugin, `load`. We're just going to load it. This plugin collects load metrics on our host.

Let's create a file for the `load` plugin.

Listing 5.28: Creating the /etc/collectd.d/load.conf file

```
$ sudo touch /etc/collectd.d/load.conf
```

And configure it to load the `load` plugin.

Listing 5.29: The load.conf file

```
LoadPlugin load
```

The processes plugin

Lastly, we're going to configure the `processes` plugin. The `processes` plugin monitors both processes broadly on the host—for example, the number of active and zombie processes—but it can also be focused on individual processes to get more details about them. We're going to create a file to hold our `processes` plugin configuration.

Listing 5.30: Creating the processes.conf configuration file

```
$ sudo vi /etc/collectd.d/processes.conf
```

We're then going to specifically monitor the `collectd` process itself and set a **default threshold for any monitored processes**. The `processes` plugin provides insight into the performance of specific processes. We'll also use it to make sure specific processes are running—for example, ensuring `collectd` is running.

Listing 5.31: The processes.conf file

```
LoadPlugin processes
<Plugin "processes">
    Process "collectd"
</Plugin>

<Plugin "threshold">
    <Plugin "processes">
        <Type "ps_count">
            DataSource "processes"
            FailureMin 1
        </Type>
    </Plugin>
</Plugin>
```

In some cases you might want to monitor a process (or any other plugin) at a longer interval than the global default of 2 seconds. You can override the global **Interval** by setting a new interval when loading the plugin. To do this we convert the **LoadPlugin** directive into a block like so:

Listing 5.32: Overriding the global interval

```
<LoadPlugin processes>
    Interval 10
</LoadPlugin>
```

TIP You can see the other options you can configure per-plugin in the LoadPlugin configuration section in the [collectd configuration documentation](#).

Here inside our new `<LoadPlugin>` block we've specified a new `Interval` directive with a collection period of `10` seconds.

We continue by configuring the `processes` plugin inside the `<Plugin>` block. It can take two configuration options: `Process` and `ProcessMatch`. The `Process` option matches a specific process by name, for example the `collectd` process as we've defined here. The `ProcessMatch` option matches one or more processes via a regular expression. Each regular expression is tied to a label, like so:

Listing 5.33: The ProcessMatch option

ProcessMatch label regex

Let's take a look at `ProcessMatch` in action.

Listing 5.34: Using the ProcessMatch option

```
<Plugin "processes">
    ProcessMatch "carbon-cache" "python.+carbon-cache"
    ProcessMatch "carbon-relay" "python.+carbon-relay"
</Plugin>
```

This example would match all of the `carbon-cache` and `carbon-relay` processes we configured in Chapter 4 under labels. For example, the regular expression `python.+carbon-cache` would match all Carbon Cache daemons running on a host and group them under the label `carbon-cache`.

In another example, if we wished to monitor the Riemann server running on our `riemann`, `riemannb`, and `riemannmc` hosts, we'd configure the following `ProcessMatch` regular expression.

Listing 5.35: Using the ProcessMatch option for Riemann

```
<Plugin "processes">
  ProcessMatch "riemann" "\briemann.jar:\sriemann.bin\b"
</Plugin>
```

If we run a `ps` command on one of the Riemann hosts to check if monitoring for `\briemann.jar:\sriemann.bin\b` will find the Riemann process we'll see:

Listing 5.36: Checking for Riemann in the ps command

```
$ ps aux | grep '\briemann.jar:\sriemann.bin\b'
riemann 14998 33.8 28.4 6854420 2329212 ? Sl Apr21 5330:50
  java -Xmx4096m -Xms4096m -XX:NewRatio=1 -XX:PermSize=128m -XX:
    MaxPermSize=256m -server -XX:+ResizeTLAB -XX:+
    UseConcMarkSweepGC -XX:+CMSConcurrentMTEnabled -XX:+
    CMSClassUnloadingEnabled -XX:+UseParNewGC -XX:-
    OmitStackTraceInFastThrow -cp /usr/share/riemann/riemann.jar:
      riemann.bin start /etc/riemann/riemann.config
```

We see here that our `grep` matches `\briemann.jar:\sriemann.bin\b`, hence collectd will be able to monitor the Riemann server process.

The output from the `processes` plugin for specific processes provides us with detailed information on:

- Resident Segment Size (RSS), or the amount of physical memory used by the process.
- CPU user and system time consumed by the processes.
- The number of threads used.
- The number of major and minor page faults.
- The number of processes by that name that are running.

Now let's look at how we're going to configure some process thresholds using some of this data.

The processes plugin and thresholds

The last metric, the number of processes by that name, is useful for determining whether the process itself has failed. We're going to use it to monitor processes to determine if they are running. This brings us to the second piece of configuration in our `processes.conf` file.

Listing 5.37: The processes threshold

```
 . . .
<Plugin "threshold">
    <Plugin "processes">
        <Type "ps_count">
            DataSource "processes"
            FailureMin 1
        </Type>
    </Plugin>
</Plugin>
```

This new `<Plugin>` block configures the `threshold` plugin we loaded as part of our global configuration. You can specify one or more thresholds for every plugin

you use. If a threshold is breached then collectd will generate failure events—for example, if we were monitoring the RSyslog daemon and collectd stopped detecting the process, we'd receive a notification much like:

Listing 5.38: A collectd notification

```
[2016-01-07 04:51:15] Notification: severity = FAILURE, host =
graphitea, plugin = processes, plugin_instance = rsyslogd, type
= ps_count, message = Host graphitea, plugin processes (
instance rsyslogd) type ps_count: Data source "processes" is
currently 0.000000. That is below the failure threshold of
1.000000.
```

The plugin we're going to use for sending events to Riemann is also aware of these thresholds and will send them onto Riemann where we can use them to detect services that have failed.

To configure our threshold, we must first define which collectd plugin the threshold will apply to using a new `<Plugin>` block. In this case we're defining thresholds for the `processes` plugin. We then need to tell the `threshold` plugin which metric generated by this plugin to use for our threshold. For the `processes` plugin this metric is called `ps_count`, which is the number of processes running under a specific name. We define the metric to be used with the `<Type>` block.

Inside the `<Type>` block we specify two options: `DataSource` and `FailureMin`. The `DataSource` option references the type of data we're collecting. Each collectd metric can be made up of multiple types of data. To keep track of these data types, the collectd daemon has a central registry of all data known to it, called the type database. It's located in a database file called `types.db`, which was created when we installed collectd. On most distributions it's in the `/usr/share/collectd/` directory. If we peek inside the `types.db` file to find the `ps_count` metric we

will see:

Listing 5.39: The ps_count metric

```
ps_count      processes:GAUGE:0:1000000, threads:GAUGE:0:1000000
```

Note that the `ps_count` metric is made up of two data sources, both gauges (see Chapter 2 for information on what a gauge is):

- `processes` - which can range from `0` to `1000000`.
- `threads` - which can range from `0` to `1000000`.

This means `ps_count` can be used to either track the number of processes or the number of threads. In our case we care about how many processes are running.

The second option, `FailureMin`, is the threshold itself. The `FailureMin` option specifies the minimum number of processes required before triggering a failure event. We've specified `1`. So, if there is `1` process running, collectd will be happy. If the process count drops below `1`, then a failure event will be generated and sent on to Riemann.

The `threshold` plugin supports several types of threshold:

- `FailureMin` - Generates a failure event if the metric falls below the minimum.
- `WarningMin` - Generates a warning event if the metric falls below the minimum.
- `FailureMax` - Fails if the metric exceeds the maximum.
- `WarningMax` - Warns if the metric exceeds the maximum.

If you combine failure and warning thresholds you can create dual-tier event notifications, for example:

Listing 5.40: Specifying a warning and a failure

```
FailureMin 1  
WarningMin 3
```

Here, if the metric drops below **3** then a warning event is triggered. If it drops below **1** then a failure event is triggered. We use this to vary the response to a threshold breach—for example, a warning might not merit immediate action but the failure would.

If the threshold breach is corrected then collectd will generate a “things are okay” event, letting us know that things are back to normal. This will also be sent to Riemann and we could use it to automatically resolve an incident or generate a notification.

The threshold we’ve just defined is global for the **processes** plugin. This means any process we watch with the **processes** plugin will be expected to have a minimum of one running process. Currently, we’re only managing one process, **collectd**; if it drops to zero, it likely means collectd has failed. But if we were to define other processes to watch, which we’ll do in subsequent chapters, then collectd would trigger events for these if the number of processes was to drop below **1**.

But sometimes normal operation for a service will be to have more than one running process. To address this we customize thresholds for specific processes. Let’s say normal operation for a specific process meant two or more processes should be running, for example:

Listing 5.41: Matching multiple thresholds

```
<Plugin "processes">
    ProcessMatch "carbon-cache" "python.+carbon-cache"
    ProcessMatch "carbon-relay" "python.+carbon-relay"
</Plugin>

<Plugin "threshold">
    <Plugin "processes">
        Instance "carbon-cache"
        <Type "ps_count">
            DataSource "processes"
            WarningMin 2
            FailureMin 1
        </Type>
        Instance "carbon-relay"
        <Type "ps_count">
            DataSource "processes"
            FailureMin 1
        </Type>
    </Plugin>
</Plugin>
```

NOTE In the source code for the book we've included collectd process monitoring configuration for Graphite and Riemann in [the code for those chapters](#).

Here we have our `carbon-cache` and `carbon-relay` process matching configura-

tion. We know that there should be two `carbon-cache` processes, and we configure the `threshold` plugin to check that. We've added a new directive, `Instance`, which we've set to the `ProcessMatch` label, `carbon-cache`. The `Instance` directive tells the `threshold` plugin that this configuration is specifically for monitoring our `carbon-cache` processes.

We've also set both the `WarningMin` and `FailureMin` thresholds. If the number of `carbon-cache` processes drops below `2` then a warning event will trigger. If the number drops below `1` then a failure event will trigger.

TIP There are a number of other useful configuration items we can set for the `threshold` plugin. You can read about them on the collectd wiki's [threshold configuration page](#).

If we were to stop the `carbon-cache` daemon now on one of our Graphite hosts, we'd see the following event in the `/var/log/collectd.log` log file.

Listing 5.42: The `carbon-cache` collect failure event

```
[2015-12-25 00:09:08] Notification: severity = FAILURE, host =
graphitea, plugin = processes, plugin_instance = carbon-cache,
type = ps_count, message = Host graphitea, plugin processes (
instance carbon-cache) type ps_count: Data source "processes"
is currently 0.000000. That is below the failure threshold of
2.000000.
```

We could then use this event in Riemann to trigger a notification and let us know something was wrong. We'll see how to do that later in this chapter. First though, we need our metrics to be sent onto our last plugin, `write_riemann`, which does

the actual sending of data to Riemann.

NOTE We've also added a similar configuration for our carbon-relay process.

Writing to Riemann

Next, we need to configure the `write_riemann` plugin. This is the plugin that will write our metrics to Riemann. We're going to create our `write_riemann` configuration in a file in the `/etc/collectd.d/` directory.

First, let's create a file to hold the plugin's configuration.

Listing 5.43: The `write_riemann.conf` configuration file

```
$ sudo touch /etc/collectd.d/write_riemann.conf
```

Now let's load and configure the plugin.

Listing 5.44: Configuring the write_riemann plugin

```
LoadPlugin write_riemann
<Plugin "write_riemann">
    <Node "riemann">
        Host "riemann.example.com"
        Port "5555"
        Protocol TCP
        CheckThresholds true
        StoreRates false
        TTLFactor 30.0
    </Node>
    Tag "collectd"
</Plugin>
```

We first use the `LoadPlugin` directive to load the `write_riemann` plugin. Then, inside the `<Plugin>` block, we specify `<Node>` blocks. Each `<Node>` block has a name and specifies a connection to a Riemann instance. We've called our only node `riemann`. Each node must have a unique name.

Each connection is configured with the `Host`, `Port`, and `Protocol` options. In this case we're installing on a host in the Production A environment, and we're connecting to the corresponding Riemann server in that environment: `riemann.example.com`. We're using the default port of `5555` and the TCP protocol. To ensure your collectd events reach Riemann you'll need to ensure that TCP port `5555` is open between your hosts and the Riemann server. We're instead going to use TCP rather than UDP to provide a stronger guarantee of delivery.

NOTE Don't use UDP to send events to Riemann. UDP has no guarantee of

delivery, ordering, or duplicate protection. You will lose events and data.

The `CheckThresholds` option configures Riemann to use any thresholds we might set using the `thresholds` plugin we enabled earlier.

Setting the `StoreRates` option to `false` configures the plugin to not convert any counters to rates. The default, `true`, turns any counters into rates rather than incremental integers. As we're going to use counters a lot (see especially Chapter 9 on application metrics) we want to use them rather than rates.

The last option in the `Node` block is `TTLFactor`. This contributes to setting the default TTL set on events sent to Riemann. It's a factor in the TTL calculation:

$$\text{Interval} \times \text{TTLFactor} = \text{EventTTL}$$

This takes the `Interval` we set earlier, in our case `2`, and the `TTLFactor` and multiples them, producing the value of the `:ttl` field in the Riemann event that will get set when events are sent to Riemann. Our calculation would look like:

$$2 \times 30.0 = 60$$

This sets the value of the `:ttl` field to `60` seconds, matching the default TTL we set using the `default` function in Chapter 3. This means collectd events will have a time to live of 60 seconds if we choose to index them in Riemann.

If we want to send metrics to two Riemann instances we'll add a second `Node` block, like so:

Listing 5.45: Configuring a second Node block

```
<Plugin "write_riemann">
    <Node "riemann">
        Host "riemann.example.com"
        Port "5555"
        Protocol TCP
        CheckThresholds true
        TTLFactor 30.0
    </Node>
    <Node "riemannb">
        Host "riemannb.example.com"
        Port "5555"
        Protocol TCP
        CheckThresholds true
        TTLFactor 30.0
    </Node>
    Tag "collectd"
</Plugin>
```

You can see we've added a second `<Node>` block for the `riemannb.example.com` server. This potentially allows us to send data to multiple Riemann hosts, creating some redundancy in delivering events.

This is also how we could handle sharding and partitioning if we needed to scale our Riemann deployment. Hosts can be divided into classes and categories and directed at specific Riemann servers. For example, we could create servers like `riemann1` or `riemann2`, etc. With a configuration management tool it is easy to configure collectd (or other collectors) to use a specific Riemann server.

Lastly, we've also specified the `Tag` option. The `Tag` option adds a string as a tag

to our Riemann events, populating the `:tags` field of an event with that string. You can specify multiple tags by specifying multiple `Tag` options.

NOTE You can find the full `write_riemann` documentation at [the collectd wiki](#).

Finishing up

Now that we have a basic collectd configuration we can proceed to add that configuration to all of our hosts. The best way to do this—indeed, the best way to install and manage collectd overall—is to use a configuration management tool like Puppet, Chef, or Ansible. The collectd configuration file and the per-plugin configuration files we’ve just created lend themselves to management with most configuration management tools’ template engines. This makes it centrally managed and easier to update across many hosts.

Enabling and running collectd

Now that we have configured the collectd daemon we enable it to run at boot and start it. On Ubuntu we enable collectd and start it like so:

Listing 5.46: Enabling and starting collectd on Ubuntu

```
$ sudo update-rc.d collectd defaults  
$ sudo service collectd start
```

On Red Hat we enable collectd and start it like so:

Listing 5.47: Enabling and starting collectd on Red Hat

```
$ sudo systemctl enable collectd  
$ sudo service collectd start
```

We then see in the `/var/log/collectd.log` log file if collectd is running.

Listing 5.48: The collectd log file

```
[2015-08-18 20:46:01] Initialization complete, entering read-loop
```

The collectd events

Once the collectd daemon is configured and running, events should start streaming into Riemann. Let's log in to `riemann.example.com` and look in the `/var/log/riemann/riemann.log` log file to see what some of those events look like. Remember, our Riemann instance is currently configured to output all events to a log file:

Listing 5.49: Our current riemann.config

```
(streams
  (default :ttl 60
    ; Index all events immediately.
    index

    ; Send all events to the log file.
    #(info %)

  (where (service #"riemann.*")
    graph

    downstream))))
```

The `#(info %)` function will send all incoming events to the log file. Let's look in the log file now to see events generated by collectd from a selection of plugins.

Listing 5.50: The collectd events in Riemann

```
{
  :host tornado-web1, :service df-root/df_complex-free, :state ok,
  :description nil, :metric 2.7197804544E10, :tags [collectd], :
  time 1425240301, :ttl 60.0, :ds_index 0, :ds_name value, :
  ds_type gauge, :type_instance free, :type df_complex, :
  plugin_instance root, :plugin df}

  {:host tornado-web1, :service load/load/shortterm, :state ok, :
  description nil, :metric 0.05, :tags [collectd], :time
  1425240301, :ttl 60.0, :ds_index 0, :ds_name shortterm, :
  ds_type gauge, :type load, :plugin load}

  {:host tornado-web1, :service memory/memory-used, :state ok, :
  description nil, :metric 4.5678592E7, :tags [collectd], :time
  1425240301, :ttl 60.0, :ds_index 0, :ds_name value, :ds_type
  gauge, :type_instance used, :type memory, :plugin memory}
}
```

Let's look at some of the fields of the events that we're going to work with in Riemann.

Field	Description
host	The hostname, e.g. <code>tornado-web1</code> .
service	The service or metric.
state	A string describing state, e.g. <code>ok</code> , <code>warning</code> , <code>critical</code> .
time	The time of the event.
tags	Tags from the <code>Tag</code> fields in the <code>write_riemann</code> plugin.
metric	The value of the collectd metric.
ttl	Calculated from the <code>Interval</code> and <code>TTLFactor</code> .
type	The type of data. For example, CPU data.
type_instance	A sub-type of data. For example, idle CPU time.
plugin	The collectd plugin that generated the event.

Field	Description
ds_name	The DataSource name from the <code>types.db</code> file.
ds_type	The type of DataSource. For example, a gauge.

The collectd daemon has constructed events that contain a `:host` field with the host name we're collecting from. In our case all three events are from the `tornado-web1` host. We also have a `:service` field containing the name of each metric collectd is sending.

Importantly, when we send metrics to Graphite, it will construct the metric name from a combination of these two fields. So the combination of `:host tornado-web1` and `:service memory/memory-used` becomes `tornado-web1.memory.memory-used` in Graphite. Or `:host tornado-web1` and `:service load/load/shortterm` becomes `tornado-web1.load.load.shortterm`.

Our collectd events also have a `:state` field. The value of this field is controlled by the threshold checking we enabled with the `threshold` plugin above. If no specific threshold has been set or triggered for a collectd metric then the `:state` field will default to a value of `ok`.

Also familiar from Chapter 3 should be the `:ttl` field which sets the time to live for Riemann events in the index. Here the `:ttl` is 60 seconds, controlled by multiplying our `Interval` and `TTLFactor` values as we discovered when we configured collectd.

The `:description` field is empty, but we get some descriptive data from the `:type`, `:type_instance`, and `:plugin` fields which tell us what type of metric it is, the more granular type instance, and the name of the plugin that collected it, respectively. For example, one of our events is generated from the `memory` plugin with a `:type` of `memory` and a `:type_instance` of `used`. Combined, these provide a description of what data we're specifically collecting.

Also available is the `:ds_name` field which is a short name or title for the data source. Next is the `:ds_type` field which specifies the data source type. This tells

us what type of data this source is—in the case of all these events, a type of `gauge`. The collectd daemon has four data types:

- Gauges — A value that is simply stored. This is generally for values that can increase and decrease.
- Derive — These values assume any change in value is interesting, the derivative.
- Counter — Like the derive data source type, but if a new value is smaller than the previous value, collectd assumes the counter has rolled.
- Absolute — This is intended for counters that are reset upon reading. We won't see any absolute data sources in this chapter.

TIP You can find more details on collectd data sources on [the collectd wiki](#), and you can find the full list of default data types in the `types.db` file in `/usr/share/collectd/` directory.

Lastly, the `:metric` and the `:time` fields hold the actual value of the metric and the time it was collected, respectively.

Sending our collectd events to Graphite

Now that we've got events flowing from collectd to Riemann, we want to send them the step further and onto Graphite so we can graph them. We're going to use a `tagged stream` to select the collectd events. The `tagged` stream selects all events with a specific tag. Our collectd events have acquired a tag, `collectd`, via the `Tag` directive in the `write_riemann` configuration. We can match on this tag and then send the events to Graphite using the `graph` var we created in Chapter 4.

Let's see our new streams.

Listing 5.51: Sending collectd metrics to Graphite

```
    . . .
    (tagged "collectd"
        graph)
    . . .
```

Our `tagged` stream grabs all events tagged with `collectd`. We then added the `graph` var we created in Chapter 4 so that all these events will be sent on to Graphite.

Refactoring the collectd metric names

Now let's see what the metrics arriving at Graphite look like? If we look at the `/var/log/carbon/creates.log` log file on our `graphitea.example.com` host we should see our metrics being updated. Most of them look much like this:

Listing 5.52: The collectd metrics in Graphite

```
 . . .
productiona.hosts.tornado-web1.interface-eth0/if_packets/rx
productiona.hosts.tornado-web1.interface-eth0/if_errors/rx
productiona.hosts.tornado-web1.cpu-percent/cpu-steal
productiona.hosts.tornado-web1.df-root/df_complex-used
productiona.hosts.tornado-web1.interface-lo/if_packets/rx
productiona.hosts.tornado-web1.processes/ps_state-paging
productiona.hosts.tornado-web1.processes/ps_state-zombies
productiona.hosts.tornado-api1.df-dev/df_complex-used
productiona.hosts.tornado-api1.df-run-shm/df_complex-reserved
productiona.hosts.tornado-api1.df-dev/df_complex-reserved
productiona.hosts.tornado-api1.interface-eth0/if_octets/rx
productiona.hosts.tornado-api1.df-run-lock/df_complex-reserved
productiona.hosts.tornado-api1.interface-lo/if_octets/rx
productiona.hosts.tornado-api1.df-run-lock/df_complex-free
productiona.hosts.tornado-api1.df-run/df_complex-reserved
productiona.hosts.tornado-api1.processes/ps_state-blocked
 . . .
```

We see that our `productiona.hosts.` has been appended to the metric name thanks to the configuration we added in Chapter 4. We also see the name of the host from which the metrics are being collected, here `tornado-web1` and `app2-api`. Each metric name is a varying combination of the collectd plugin data structure: plugin name, plugin instance, and type. For example:

Listing 5.53: A processes metric

```
productiona.hosts.tornado-api1.processes/ps_state-blocked
```

This metric combines the `processes` plugin name; the process state, `ps_state`; and the type, `blocked`.

You can see that a number of the metrics have fairly complex naming patterns. To make it easier to use them we're going to try to simplify that a little. This is entirely optional but it is something we do to make building graphs a bit faster and easier. If you don't see the need for any refactoring you can skip to the next section.

There are several places we could adjust the names of our metrics:

- We can use [the Chains construct](#) in the collectd daemon.
- We can rewrite [the rules inside Carbon](#) using the Carbon Aggregation daemon.
- We can rewrite the metric events inside Riemann.

We're going to use the last option, Riemann, because it's a nice central collection point for metrics. We're going to make use of a neat bit of code written by [Pierre-Yves Ritschard](#), a well-known member of the collectd and Riemann communities. The code takes the incoming collectd metrics and rewrites their `:service` fields to make them easier to understand.

Let's first create a file to hold our new code on our Riemann server.

Listing 5.54: Creating the service rewrite rules

```
$ sudo touch /etc/riemann/examplecom/etc/collectd.clj
```

Here we've created `/etc/riemann/examplecom/etc/collectd.clj`. Let's populate this file with our borrowed code.

Listing 5.55: Rewriting services inside Riemann

```
(ns examplecom.etc.collectd
  (:require [clojure.tools.logging :refer :all]
            [riemann.streams :refer :all]
            [clojure.string :as str]))


(def default-services
  [{:service #^load/load/(.*)$" :rewrite "load $1"}
   {:service #^swap/percent-(.*)$" :rewrite "swap $1"}
   {:service #^memory/percent-(.*)$" :rewrite "memory $1"}
   {:service #^processes/ps_state-(.*)$" :rewrite "processes $1"
    }
   {:service #^cpu/percent-(.*)$" :rewrite "cpu $1"}
   {:service #^df-(.*)/(df_complex|percent_bytes)-(.*)$" :
    rewrite "df $1 $2 $3"}
   {:service #^interface-(.*)/if_(errors|packets|octets)/(tx|rx)
    $" :rewrite "nic $1 $3 $2"}])


(defn rewrite-service-with
  [rules]
  (let [matcher (fn [s1 s2] (if (string? s1) (= s1 s2) (re-find
  s1 s2)))]
  (fn [{:keys [service] :as event}]
    (or
      (first
        (for [{:keys [rewrite] :as rule} rules
              :when (matcher (:service rule) service)]
          (assoc event :service
            (if (string? (:service rule))
                (str/replace service (:service rule) rewrite)))
        )))
      event)))))

(defn rewrite-service
  (rewrite-service-with default-services))
```

This looks complex but what's happening is actually pretty simple. We first define a namespace: `examplecom.etc.collectd`. We then require three libraries. The first is Clojure's string functions from `clojure.string`. When we require this namespace we've used a new directive called `:as`. This creates an alias for the namespace, here `str`. This allows us to refer to functions inside the library as by this alias: `str/replace`. If we remember Chapter 3, `refer` lets you use names from other namespaces without having to fully qualify them, and `:as` lets you use a shorter name for a namespace when you're writing out a fully qualified name.

We also require the `clojure.tools.logging` namespace which provides access to some logging functions, for example the `info` function we used earlier in the book. Lastly, we require the `riemann.streams` namespace which provides access to Riemann's streams, for example the `where` stream.

Next we've created a var called `default-services` with the `def` statement. This is a series of regular expression maps. Each line does a rewrite of a specific `:service` field like so:

Listing 5.56: The service rewrite line

```
{:service #"\^cpu/percent-(.*)$" :rewrite "cpu $1"}
```

We first specify the content of the `:service` field that we want to match, here that's `#"\^cpu/percent-(.*)$"`. This will grab all of the collectd CPU metrics. We capture the final portion of the metric namespace. In the next element, `:rewrite`, we specify how the final metrics will look—here that's `cpu $1`, with `$1` being the output we captured in our initial regular expression. What's this actually look like? Well, the `:service` field of one of our CPU metrics is: `cpu-percent/cpu-steal`. Our regular expression will match the `steal` portion of the current metric name and rewrite the service to: `cpu steal`. Graphite's metric names treat spaces as dots, so when it hits Graphite that will be converted into the full path of:

```
productiona.hosts.tornado-web1.cpu.steal
```

The rules in our code right now handle the basic set of metrics we're collecting in this chapter. You can easily add new rules to cover other metrics you're collecting.

Next is a new function called `rewrite-service-with`. This contains the magic that actually does the rewrite. It takes a list of rules, examines incoming events, grabs any events with a `:service` field that matches any of our rules, and then rewrites the `:service` field using the Clojure string function `replace`. It's reasonably complex but you're not likely to ever need to change it. We're not going to step through it in any detail.

NOTE If you have any questions you can find the original code on GitHub at <https://github.com/pyr/riemann-extra>.

Lastly, we got a final var called `rewrite-service`. This is what actually runs the `rewrite-service-with` function and passes it the rules in the `default-services` var.

Let's rewrite our original `tagged` filter in `/etc/riemann/riemann.config` to send events through our rewrite function. Here's our original stanza.

Listing 5.57: Original collectd to graph where filter

```
(tagged "collectd"  
      graph)
```

Let's replace it with:

Listing 5.58: Updated collectd to graph stream

```
(require '[examplecom.etc.collectd :refer :all])  
.  
.  
.  
(tagged "collectd"  
  (smap rewrite-service graph))
```

We first add a `require` to load the functions in our

`examplecom/etc/collectd`

namespace. We're still using the same `tagged` filter to grab all events tagged with `collectd`, but we're passing them to a new stream called `smap`. The `smap` stream is a streaming map. It's highly useful for transforming events. In this case the `smap` stream says: “send any incoming events into the `rewrite-service` var, process them, and then send them onto the `graph` var”.

This will turn a metric like:

Listing 5.59: Original metric

```
productiona.hosts.tornado-web1.load/load/shortterm
```

Into:

Listing 5.60: Rewritten metric

```
productiona.hosts.tornado-web1.load.shortterm
```

TIP When Graphite sees spaces or slashes it converts them into periods.

This makes the overall metric much easier to parse and use, especially when we start to build graphs and dashboards.

NOTE We've included all example configuration and code in the book [on GitHub](#).

Summary

In this chapter we saw how to collect the base level of data across our hosts: CPU, memory, disk, and related data. To do this we installed and configured collectd. We configured collectd to use plugins to collect a wide variety of data on our hosts and direct them to Riemann for any processing and checks, and to forward them onto Graphite for longer-term storage.

In the next chapter we'll look at making use of our collectd metrics in Riemann and in Graphite and Grafana.

Chapter 6

Using collectd events in Riemann

In Chapter 5 we set up monitoring and events in collectd and sent those events into Riemann. Now let's see how we might use those events for monitoring. We're going to look at several examples of checks we could create using our collectd events.

- We're going to check collectd or other processes are running.
- We're going to replicate some traditional monitoring checks for CPU, memory and disk, to help the transition from that more traditional monitoring model.
- We're going to learn how to create better checks with more sophisticated logic.
- We're going to create a host-centric Grafana dashboard displaying our collectd metrics as graphs.

NOTE This section builds upon our earlier look at statistical techniques in Chapter 2. Like there, this book isn't going to teach you statistics, but it's going to mention, where relevant, what sort of analysis is best for certain types of metrics.

If you need to learn some statistics (and you do!) there are numerous good books available that can get you started.

Checking processes are running

The first check we're going to create is for ensuring processes are running. In Chapter 5 we configured the `processes` plugin. At the time, we configured it to monitor our `collectd` process, and we looked at examples of how to monitor Riemann and Graphite's Carbon daemons.

The `processes` plugin generates a series of metrics for the processes it is monitoring, including the `ps_count` metric which counts the number of processes of that name that are running. When we configured the `processes` plugin we also configured a threshold for that metric: a global minimum of `1` process needs to be running for any process being monitored. We're going to make use of the notifications produced when this threshold is breached to monitor running processes on our hosts.

But what if collectd fails? Then what sends the notifications? And will we ever receive one for collectd? This is a potential issue, but one we can work around by using *both* the metric and its existence to measure availability.

To do this we're going to add three streams:

- A wrapper that uses the `tagged` stream to match collectd events,
- A stream that matches the threshold notification, and
- A stream that matches expired events.

TIP You can learn more about event filtering in Chapter 3 and on the [Riemann](#)

website.

That way we catch all variations of process failures and metrics disappearing from the Riemann index when they are not sent.

Listing 6.1: Detecting collectd down or expired

```
....  
  
(tagged "collectd"  
  (tagged "notification"  
    (by [:host :service]  
      (changed :state {:init "ok"})  
      (adjust [:service clojure.string/replace #"\^processes-  
        (.* )\ps_count\$" "$1"]  
        (email "james@example.com")))))  
  
(where (and (expired? event)  
            (service #"\^processes-.+\ps_count\processes")))  
  (adjust [:service clojure.string/replace #"\^processes-(.* )\/  
    ps_count\processes\$" "$1"]  
    (email "james@example.com"))))  
  
....
```

You can see we've used a `tagged` stream to match all events with `collectd` in the `:tag` field. This wraps around both our other streams and narrows our events down to those coming from collectd.

We then create a second `tagged` stream that matches events tagged with

notification. These are our threshold notifications. When they reach Riemann they look like:

Listing 6.2: A collectd notification event

```
{:host graphitea, :service processes-rsyslogd/ps_count, :state
critical, :description Host graphitea, plugin processes (
instance rsyslogd) type ps_count: Data source "processes" is
currently 0.000000. That is below the failure threshold of
1.000000., :metric 0.0, :tags [notification collectd], :time
1451577254, :ttl 60, :DataSource processes, :type ps_count,
plugin_instance rsyslogd, :plugin processes}
```

We see that they are tagged with **notification**, and when reporting a failure they have a **:state** of **critical** and a useful description.

We then use a **by** stream to split events by field. This creates a new child stream each time a unique field is encountered—here we’re splitting on the **:host** and **:service** fields. The **by** stream will create a child stream for each unique host and service it encounters. This is useful when we want to run *many copies* of a particular stream, like we’re doing here so we get a distinct host and service to track independently. If we didn’t do this, we’d trigger notifications when, for example, **servicea** reports **ok** but **serviceb** reports **critical**.

Next, we’re going to take advantage of Riemann’s built-in change detection. Change detection works by identifying when an event differs from a previous event. We’ve used the **changed** var which performs our actual detection. We’ve specified we’re doing detection on the **:state** field. We’ve also given it a single argument, **{:init "ok"}**. This controls a base state for detection, and assumes that the default, initial value of the **:state** field in an event will be **ok**. This also protects us from false positives when Riemann first starts and has no history of

previous states.

TIP Another useful function to consider if you're worried about your state changes being triggered by spikes or flapping (where services rapidly change state) is `stable`. The `stable` var tries to only notify if an event is consistently in a specific state.

But Riemann also comes with a neat shortcut, `changed-state`, that builds on the `changed` var by breaking events into distinct streams. This replaces the need to specify a `by` stream.

Let's update our example to use our new var:

Listing 6.3: Detecting changed state

```
(tagged "notification"
  (changed-state {:init "ok"}
    (adjust [:service clojure.string/replace #"\^processes-(.*)
      \/ps_count\$" "$1"]
      (email "james@example.com"))))
```

We removed the need to specify a `by` stream, or the specified field we're monitoring for change—the `changed-state` var defaults to using the `:state` field. Now if the `processes` plugin detects any failures in a process being monitored it'll generate an event with a `:state` of `critical` and trigger our change detection and a notification.

What's also useful about Riemann's state detection is that it works both ways—for example, when an event is returned to normal then Riemann detects this change

in state too. For example, if your event has a `:state` of `critical`, and a new event for that host and service arrives with a `:state` of `ok`, then Riemann will trigger a state change. You can use this to resolve a triggered notification if the host or service recovers.

TIP We'll go over using state detection in more depth in Chapter 10.

We then adjusted the contents of the `:service` field to make it easier to see what service is impacted. Since our only thresholds are being triggered by the `processes` plugin we can be specific about the expected content of the `:service` field.

Listing 6.4: Adjusting the `:service` field

```
(adjust [:service clojure.string/replace #"\^processes-(.*)\\/"
          ps_count$" "$1"])
```

To do this we extract the name of the process being monitored using Riemann's `adjust function` and a `clojure.string` operation called `replace`. It uses a regular expression to match and capture the process and replace the contents of the `:service` field with the regular expression capture.

This would change the `:service` field in the `ps_count` metric, for the `rsyslogd` service for example, from `processes-rsyslogd/ps_count` to `rsyslogd`.

Lastly, any matching events will then be emailed to `james@example.com`, letting us know if any process has failed.

Our third `where` stream checks for when collectd fails to identify that a process has failed—for example, when collectd itself has potentially failed.

Listing 6.5: Finding expired ps_count events

```
(where (and (expired? event)
             (service #"\^processes-.+\/ps_count\/processes"))
        (adjust [:service clojure.string/replace #"\^processes-(.*)\/"
                 ps_count\/processes$" "$1"])
        (email "james@example.com")))
```

The `where` stream matches two conditions. The first condition uses the `expired?` var to check if an event is expired from the index. This will match on any event with a `:state` field of `expired`. Our second condition is a regular expression on the `:service` field to identify the `ps_count/processes` metric. This will identify if any `ps_count/processes` metric disappears from the index, likely letting us know that a process is no longer reporting.

You'll note this is a different service from our notification check. This is an idiosyncrasy of collectd. In the notification event the `:service` field is truncated to:

`processes-rsyslogd/ps_count`

But the metric itself is called:

`processes-rsyslogd/ps_count/processes`

So for our check for expired events we use the full metric name and, as we did in the previous check, we adjust the `:service` field to make our notification more elegant. We then pass our updated event to the `email` var to send an email to `james@example.com`.

Now let's see both checks in the context of the larger Riemann configuration.

Listing 6.6: Added collectd monitoring to our Riemann configuration

```
    . . .

(streams
  (default :ttl 60
    (where (not (tagged "notification")))
      index)

  (tagged "collectd"
    (tagged "notification"
      (changed-state {:init "ok"}
        (adjust [:service clojure.string/replace #"\^processes
          -(.*)\ps_count\$" "$1"]
          (email "james@example.com"))))

  (where (and (expired? event)
    (service #"\^processes-.+\ps_count\processes")
    )
    (adjust [:service clojure.string/replace #"\^processes-
      (.*)\ps_count\processes\$" "$1"]
      (email "james@example.com")))

  . . .
```

We see our full, current Riemann configuration here. Note that we've added the new streams to the file. It's the combination of both of these checks that will catch most failure cases and will form the core of any process monitoring we'll do in

our framework.

We also made one other change: we wrapped our initial `index` var in a `where` stream. This `where` stream matches events tagged with `notification` and skips indexing them. This is because notifications are singletons. We don't need to track their state in the index because they aren't a constant—they represent transitory events rather than the “whole of environment” picture the other events in our index represent.

TIP We're using the `email` notifications to `james@example.com` as a stub. These are obviously not very sophisticated, lack a lot of useful context and don't scale. In Chapter 10 we'll enhance these notifications and identify some new potential destinations for our notifications.

Other actions and enhancements

We can do more with our collectd notifications and thresholds. We can set further thresholds inside collectd and consume the events generated in Riemann. However, the power and usefulness of configuring and managing thresholds and checks in Riemann generally outweighs collectd's nascent thresholding capabilities. Beyond our use of them with the `processes` plugin, we're not going to make use of other notification events.

On the Riemann side, though, we can certainly take other actions and refine our notifications. In Chapter 10 we'll explore the default notifications, such as our currently defined `email` var, and look at making them more useful, with added context. We can also keep track of notifications and `measure exception rates` to determine the most problematic hosts and services. We'll discuss this too in Chapter 10.

We can also trigger actions based on collectd notifications. For example, we can attempt to restart a stopped service, re-deploy code, trigger a configuration management run. We would do this by shelling out with Clojure’s `sh`, or a library like `Conch` directly, or with tools like `MCollective`, `Fabric`, or `Ansible`.

Replicating some classic monitoring

Now let’s look at how we might replicate some of the typical host monitoring checks most of us have likely used in the past. We’re not doing this as a recommendation but more as an acknowledgement that changing the way you monitor is often an evolution, not a revolution. We’re not going to include every one of these examples, but we’ll show you how transitioning to a new monitoring framework doesn’t have to mean a complete and immediate cutover.

Let’s take some basic CPU, memory, and disk monitoring examples, starting with CPU monitoring. Here we use one of the collectd CPU metrics: `cpu/percent-user`.

Listing 6.7: The `cpu/percent` metric

```
{:host tornado-web1, :service cpu/percent-user, :state ok, :  
  description nil, :metric 98.981873111782477, :tags [collectd],  
  :time 1452374243, :ttl 60.0, :ds_index 0, :ds_name value, :  
  ds_type gauge, :type_instance user, :type percent, :plugin cpu}
```

This metric tracks the percentage of CPU consumed by user processes on our hosts. It is generated using the `cpu` plugin we configured earlier in the chapter. Let’s see our check in action.

Listing 6.8: Basic CPU monitoring

```
(where (and (service "cpu/percent-user") (>= metric 80.0))
      (email "james@example.com")
    )
```

Here we've got a `where` stream that selects based on two criteria. The first matches if the `:service` field has a value of `cpu/percent-user`. The second criteria matches on the `:metric` field if it is greater than or equal to `80.0` or 80%.

If both criteria match then an email is sent to `james@example.com`. We can do a similar match for memory monitoring using the `memory/percent-used` metric. This metric shows the percentage memory used on a host and is provided by the `memory` plugin.

Listing 6.9: Basic Memory monitoring

```
(where (and (service "memory/percent-used") (>= metric 80.0))
      (email "james@example.com")
    )
```

Here we send an email if the `memory/percent-used` metric goes above `80.0` or 80%.

Finally, we do the same process with disk monitoring by looking at any filesystems we're watching. To do this we use the `df` plugin's metrics. The `df` plugin generates the percentage of storage-used metrics for every filesystem you've configured it to watch. We use a regular expression to match every filesystem or notify on a specific filesystem.

Let's look at using a regular expression. The `df` metrics look like `df-*filesystem`

*/percent_bytes-used. Here's one now:

Listing 6.10: The df plugin percent_bytes-used metric

```
{:host tornado-web1, :service df-root/percent_bytes-free, :ttl  
60, :time 359630164907/250, :metric 96.41158294677734, :  
description nil, :state ok, :tags [collectd]}
```

This is the `df` plugin metric for monitoring the percent space used of the `root` filesystem. Now let's create a check to match these metrics.

Listing 6.11: Basic disk monitoring

```
(where (and (service #^df-(.+)/percent_bytes-used) (>= metric  
90.0))  
      (email "james@example.com"))  
)
```

We've specified a `where` stream that matches on the `:service` field using a regular expression to grab all monitored filesystems, grabbing the `root` filesystem from our example metric. We've made it so if our monitored filesystem exceeds `90.0` or 90% capacity then an email will be generated.

But, as we learned in Chapter 2, these somewhat arbitrary thresholds are fragile. Is a match on our statically and arbitrarily defined threshold a spike or a sustained performance issue? This is where Riemann really starts coming into its own. With the full power of Clojure underneath it, we create better, more sophisticated monitoring checks.

Better monitoring through smarter data

We're going to make better use of our metric data through two mechanisms:

- Better data granularity
- More sophisticated check functions

First, we're collecting observations at a reasonably high granularity, generally one to two seconds. This allows us to develop checks that look at data over a longer period of time rather than at a single point in time. So, for example, instead of every event arriving in Riemann and the metric value being checked for a threshold, we instead collect the metric values for a period of time, perform an appropriate calculation on them to combine them in an intelligent manner, and then check them against that data.

Second, Riemann provides a powerful set of functions that we'll use to replace our arbitrary thresholds. These include some of the methods—like percentiles—we discussed in Chapter 2 for summarizing our metric data.

Building a median-based check

Let's revisit our CPU percentage check using some of Riemann's capabilities. Instead of checking one metric at a single point in time, we're going to take a period of time. We're then going to take all the metrics inside this period and calculate a better metric from them. We're going to start with calculating the median, or 50th, percentile from our metric data. We've chosen the median because it can handle outliers and clusters of values better than the mean. A cluster of high values or several outliers can significantly influence the mean. In these instances the median tends to provide a more representative sample of the central tendency of your data. This is a useful first step in seeing how to create checks with Riemann.

Listing 6.12: Median CPU monitoring over ten seconds

```
(where (service "cpu/percent-user")
  (by :host
    (fixed-time-window 10
      (smap folds/median
        (where (> metric 80.0)
          (email "james@example.com"))))))
```

In our new check our `where` stream matches on a `:service` of `cpu/percent-user`. We then use `aby` stream to split events by field. This creates a new child stream each time a unique field is encountered. Here we're splitting on the `:host` field.

Events matching the service are copied via the `by` stream and then sent into a new stream: `fixed-time-window`. The `fixed-time-window` stream records a fixed window of events within the last n seconds, here `10` seconds. At the conclusion of every time period, it emits a `vector` of the events in the window to any child streams.

TIP In addition to `fixed-time-window`, Riemann also has some other useful related streams. These include `moving-event-window`, a sliding window of the last few events; `moving-time-window`, a sliding window in time; and `fixed-event-window`, which provides similar capabilities but for fixed windows of events. You can read about each of these in the [Riemann stream API documentation](#).

In our case the vector of events is sent to a child stream: `folds/median`. Folds are how Riemann reduces collections of events. You can perform a number of fold

operations including [mean](#), [median](#), [maximum](#), or [count](#).

TIP You can find the full list of fold operations in the [Riemann fold API documentation](#).

The `folds/median` stream converts the values of the metrics in our 10-second period into a median value. It then passes our new event with our median metric to another `where` stream. That `where` stream is our new threshold. If our calculated median event is over `80.0`, or 80%, then a notification will be emailed to `james@example.com`.

Using percentiles for host-based checks

But what if our data isn't a normal distribution? In Chapter 2 we discovered that percentiles are a good solution for using this sort of data for monitoring. Conveniently, Riemann allows us to calculate percentiles for our window of metrics. Let's update our code to emit multiple percentiles: the 50th (the median as above) as well as the 95th and 99th percentiles. We'll also output the max to cover off any extreme outliers.

Listing 6.13: Percentile CPU monitoring over ten seconds

```
(where (service "cpu/percent-user")
  (by :host
    (percentiles 10 [0.5 0.95 0.99 1]
      (smap rewrite-service graph)

      (where (and (service "cpu/percent-user 0.99") (> metric
        80.0))
        (email "james@example.com")))))
```

We've again used our `where` and `by` streams to watch the metric we want and split the streams by the `:host` field.

We've then used a new stream called `percentiles` that calculates percentiles and emits new events for each percentile calculated. In our check, the `percentiles` stream creates a new event for every percentile we want to measure over a specified period; here that's `10` seconds. We've specified each percentile as a vector: `[0.5 0.95 0.99 1]`. This will create new events for the 50th, 95th, and 99th percentiles, as well as `1`, which is the maximum value. Each new event has the specific percentile suffixed to the `:service` field. For example:

Listing 6.14: The percentile events

```
{:host graphitea, :service cpu/percent-user 0.5, :state ok, :  
  description nil, :metric 32.48730964467005, :tags [collectd], :  
  time 1440989473, :ttl 60.0, :ds_index 0, :ds_name value, :  
  ds_type gauge, :type_instance user, :type percent, :plugin cpu}  
{:host graphitea, :service cpu/percent-user 0.95, :state ok, :  
  description nil, :metric 35.025380710659896, :tags [collectd], :  
  time 1440989474, :ttl 60.0, :ds_index 0, :ds_name value, :  
  ds_type gauge, :type_instance user, :type percent, :plugin cpu}  
{:host graphitea, :service cpu/percent-user 0.99, :state ok, :  
  description nil, :metric 33.015781291354883, :tags [collectd], :  
  time 1440989474, :ttl 60.0, :ds_index 0, :ds_name value, :  
  ds_type gauge, :type_instance user, :type percent, :plugin cpu}  
{:host graphitea, :service cpu/percent-user 1, :state ok, :  
  description nil, :metric 38.5050505050505, :tags [collectd], :  
  time 1440989475, :ttl 60.0, :ds_index 0, :ds_name value, :  
  ds_type gauge, :type_instance user, :type percent, :plugin cpu}
```

We see four new events have been created, each with a new service—for example, `cpu/percent-user 0.99` for the event measuring the 99th percentile. The calculated percentile value is contained in the `:metric` field. So in our sample events 99% of our values are less than `33.015781291354883` or 33%.

Our code wraps up by sending our new metrics to Graphite to be graphed, via an `smap` to the `rewrite-service` function and then onto the `graph` var. We're also creating a monitoring check that selects the 99th percentile and sends an email notification for any host where the CPU user metric exceeds `80.0` or 80%.

This means we now have:

- New CPU metrics, measured in percentiles and the max value, that we can graph for each host. We can use the same method for any other metrics we'd like to work with.
- A notification if user CPU in the 99th percentile on a host exceeds our threshold over the specified window of events.

This provides a much more elegant monitoring check than a stock threshold—plus we get new metrics that we can graph and visualize!

Creating check abstractions

With many events and checks you can see how your Riemann configuration might get quite complex and extensive. We've also seen that the checks we're building are broadly similar in structure. We alleviate this potential complexity by creating monitoring abstractions in the form of custom check functions.

Our threshold check

Let's look at an example of this now. We'll first create a file to hold our custom functions. We're going to create it in the `/etc/riemann/examplecom/etc/` directory to ensure it is loaded when Riemann starts.

Listing 6.15: Creating a file to hold our custom check functions

```
$ sudo touch /etc/riemann/examplecom/etc/checks.clj
```

Now let's create a custom function that replicates our threshold check.

Listing 6.16: Our first custom check function

```
(ns examplecom.etc.checks
  (:require [clojure.tools.logging :refer :all]
            [riemann.streams :refer :all]))

(defn set-state [warning critical]
  (fn [event]
    (assoc event :state
           (condp < (:metric event)
             critical "critical"
             warning "warning"
             "ok"))))

(defn check_threshold [srv window func warning critical &
                      children]
  (where (service srv)
    (fixed-time-window window
      (smap func
        (where (< warning metric)
          (smap (set-state warning critical)
            (fn [event]
              (call-rescue event children)))))))
```

We first create a new namespace: `examplecom.etc.checks`. We then `require` two Riemann libraries: `clojure.tools.logging` that contains the `info` function and related logging actions and the `riemann.streams` library that contains Riemann's stream functions, like `where`.

Here we've defined two new functions: `set_state` and `check_threshold`. The

`set_state` function allows us to set the `:state` field of an event in response to a specific threshold. This allows us to create two-tier notifications, such as a warning event and a critical event. It takes two parameters: a `warning` and a `critical` threshold. Inside the `set_state` function we run each incoming event through the `assoc` function. When applied to a map, the `assoc` function returns a new map that potentially re-maps fields. In this case we apply a conditional, `condp`, to the `:state` field in the event map. The conditional has three clauses based on the value of the `:metric` field:

- If the value of the `:metric` field exceeds the `critical` threshold then set the `:state` field of the event to `critical`.
- If the value of the `:metric` field exceeds the `warning` threshold then set the `:state` field of the event to `warning`.
- If the value of the `:metric` field is below both thresholds then set the `:state` field of the event to `ok`, which it should be by default.

The second function, `check_threshold`, is our check abstraction. It takes six parameters:

- `srv` — The service we wish to check.
- `window` — The time window we wish to monitor.
- `func` — The fold function we wish to use.
- `critical` and `warning` — The warning and critical thresholds we wish to check.
- `children` — An optional argument (indicated by the `&` symbol). These are any child streams we might want to pass to the function—for example, sending our event as an email notification.

Inside our function we use the value of the `srv` parameter to match a specific metric. We then use the `window` parameter to specify the window of time from which we want to take events. We pass our vector of events into the `smap` stream and apply the specific fold function we wish—for example, `folds/median`—to the events.

A `where` stream then matches the resulting event metric against our `warning` threshold. If it matches, it's passed to our `set_state` function which sets the `:state` based on which threshold—warning or critical—has been breached.

Lastly we have an `fn` defined function that takes our final event and uses the `call-rescue` function to pass it to any defined children, such as sending it to Graphite or triggering a notification. What does this mean? Well, we know that Riemann events travel through streams. We've also learned that streams can have one or more children. For example:

Listing 6.17: A child stream explained

```
(where (service "cpu/percent-used")
  #(info %))
```

Here `#{(info %)}` is a child stream of our `where` stream. In our case we're saying, “If there are any child streams defined to our function, then pass any matching events to them.” The `call-rescue` var calls each child stream in turn with an event and will rescue any errors and log any failures.

So if we look at an example check we will see:

Listing 6.18: Using the check_threshold function

```
(by :host
  (check_threshold "cpu/percent-user" 10 folds/median 80.0 90.0
    (email "james@example.com"))
  (check_threshold "memory/percent-used" 10 folds/median 80.0
    90.0
    (email "james@example.com")))
  . . .
```

Here we've again split our events by `:host` using the `by` stream.. We've then specified the `check_threshold` function and passed it parameters. We've specified the `cpu-percent-user` service, a `10` second time window, the `folds/median` function, a warning threshold of `80.0` or 80%, and a critical threshold of 90%. We've also specified a child stream: `(email "james@example.com")`. If an event matches our threshold then an email will be generated to `james@example.com`. We've created a second threshold check for the `memory/percent-used` metric, and we can add further checks for any other metrics we'd like to monitor.

Our percentiles check

We create a similar abstraction for our `percentiles` check. Add the following code to our `/etc/riemann/examplecom/etc/checks.clj` file.

Listing 6.19: Percentile CPU monitoring over ten seconds

```
(defn check_percentiles [srv window & children]
  (where (service srv)
    (percentiles window [0.5 0.95 0.99 1]
      (fn [event]
        (call-rescue event children))))
```

We've defined a new custom function called `check_percentiles`. Our new function has three parameters:

- The `srv` parameter takes the name of the service we wish to monitor.
- The `window` parameter specifies the period for which we'll collect events.
- The `& children` parameter handles any child streams to which we might want to pass our events.

We use a `where` stream and the `srv` parameter to match the events we want and pass them to the `percentiles` function. Our `window` parameter provides the timespan for collecting events. We then pass a vector of these events to the `percentiles` function. We've defaulted to producing the 50th, 95th, and 99th percentile results, and the maximum value. This will produce a new event for each percentile we want calculated. We then call the same function, `call-rescue`, that we used in our previous custom function to pass our events to any child streams.

Let's see how we'd use our new `check_percentiles` function.

Listing 6.20: Using the check_percentiles function

```
(by :host
  (check_percentiles "cpu/percent-user" 10
    (smap rewrite-service graph)

    (where (and (service "cpu/percent-user 0.99") (> metric 80.0)
      )
    (email "james@example.com"))))
```

We again split our streams by `:host` and then call the `check_percentiles` function with the `cpu/percent-user` as our service and `10` seconds as our window. We then passed two child streams in this function. The first child stream is `(smap rewrite-service graph)`, for sending our new events to Graphite. The second stream is a `where` stream that watches for any newly created 99th-percentile events, and those with a metric value of `80.0` or greater. If any event is matched then an email notification will be generated.

TIP You can be smarter again about these checks by having them pull thresholds or time windows dynamically from a service discovery tool such as Zookeeper. This avoids us having to hardcode them into the code and allows us to be selective per host or per service—for example, having all database hosts use a threshold of 80% whilst app servers use 90%. You can find an example of how to do this with Zookeeper [in a blog post I wrote last year](#).

Organizing our checks

Let's look at how we might organize a series of checks, such as our updated monitoring of CPU, memory, and disk. We are going to bundle together multiple `check_percentiles` functions inside a `tagged` stream.

Listing 6.21: A set of monitoring checks

```
(tagged "collectd"
  (by :host
    (check_percentiles "cpu/percent-user" 10
      (smap rewrite-service graph))
    (check_percentiles "memory/percent-used" 10
      (smap rewrite-service graph))
    (check_percentiles #"\^df-(\.*)/percent_bytes-used" 10
      (smap rewrite-service graph))))
```

We first specify a `tagged` stream that selects only the events tagged with `collectd`. This tag is added by the `write_riemann` plugin configuration by the `Tag` directive. We then use the `by` stream to split our events by the `:host` field. Each `check_percentiles` function will match on a different metric `:service` field:

- `cpu/percent-user`
- `memory/memory-used`
- `#"\^df-(\.*)/percent_bytes-used"`

Each check will collect 10 seconds worth of events and calculate our percentiles and maximum values from them. These new events will be passed to any child streams defined.

In this case we're only defining one child stream for each check: `(smap rewrite-service graph)` to send the events to Graphite. These functions would generate

four metrics from our events. For example, for the `memory/memory-used` event on the `graphitea` host we'd get:

- `productiona.hosts.graphitea.memory.used.5`
- `productiona.hosts.graphitea.memory.used.95`
- `productiona.hosts.graphitea.memory.used.99`
- `productiona.hosts.graphitea.memory.used.1`

Plus the original percentage:

- `productiona.hosts.graphitea.memory.used`

We could also use any of these metrics as a threshold or just visualize them in Grafana.

NOTE You can find other examples of how to use Riemann to perform a variety of monitoring tasks in the [Riemann HOWTO](#).

Graphing collectd metrics with Grafana

Our collectd events and check outputs are now flowing into Graphite, and we can now use them to construct a new dashboard in Grafana. To start we're going to create some CPU and memory graphs, and then we'll explore some of the other available metrics. These are example graphs we're creating. They'll expand our knowledge of Grafana and working with metrics. You should consider the sort of graphs you need for your environment based on your requirements.

Creating the Hosts dashboard

We'll start by creating a new dashboard. First, let's log in to our Grafana dashboard. For example, on our `graphitea` host we'd browse to the URL:

`http://graphitea.example.com:3000`

Now we can log into Grafana using our username and password or the default `admin / admin`.

Next, we click on the `Home` button to open our list of dashboards.

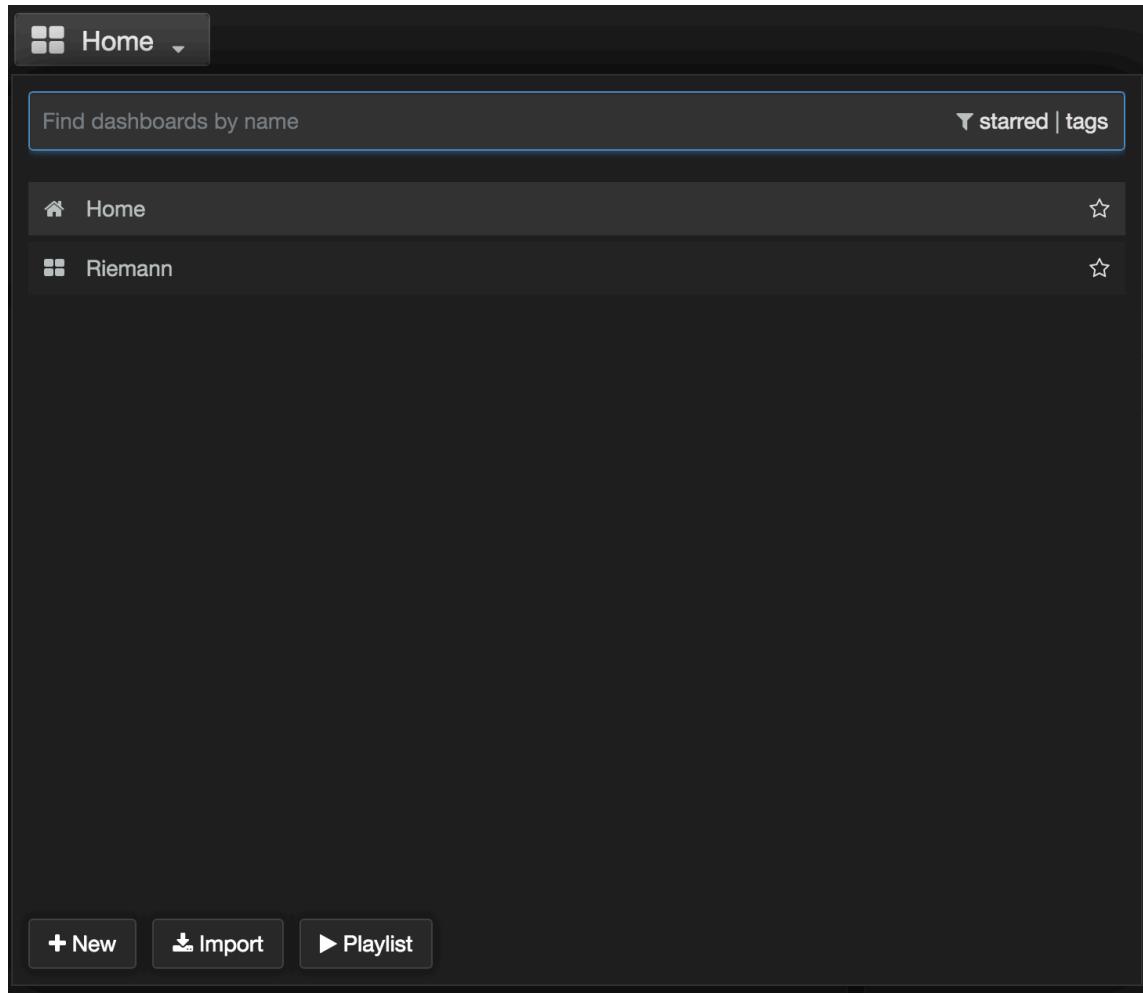


Figure 6.1: Creating a new host dashboard

We click the **+New** button to create a new dashboard. When the new dashboard opens we click on the Settings circle and select **Settings** to name our dashboard.

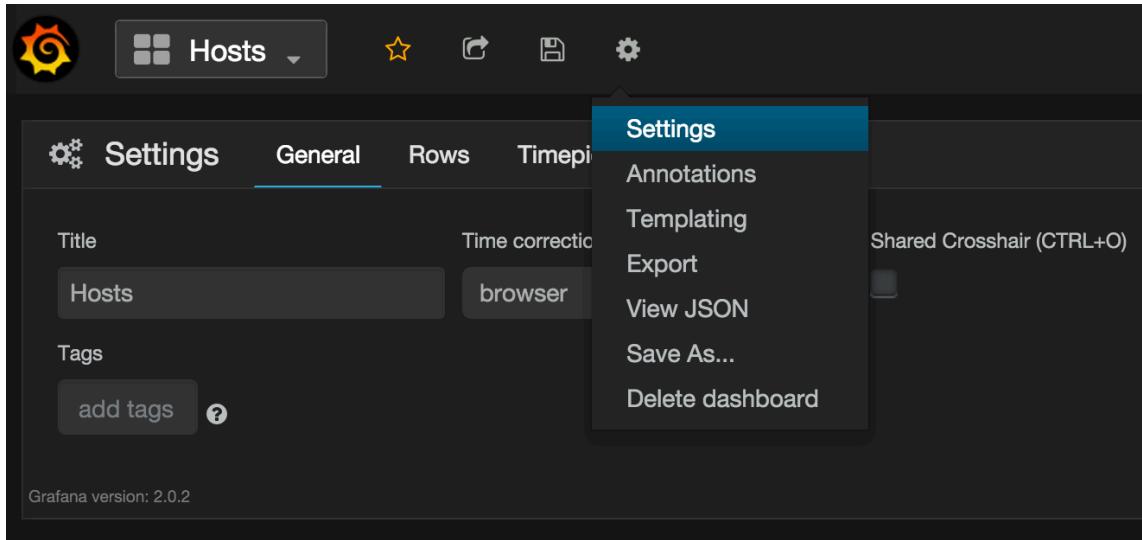


Figure 6.2: Naming our new host dashboard

Update the **Title** field with the name of the new dashboard. We're going to call ours **Hosts**. Close the **Settings** box and click the **Save** button to save the dashboard.

Creating our first host graph

Let's create our first graph by clicking on the green row bar and selecting **Add Panel -> Graph**.

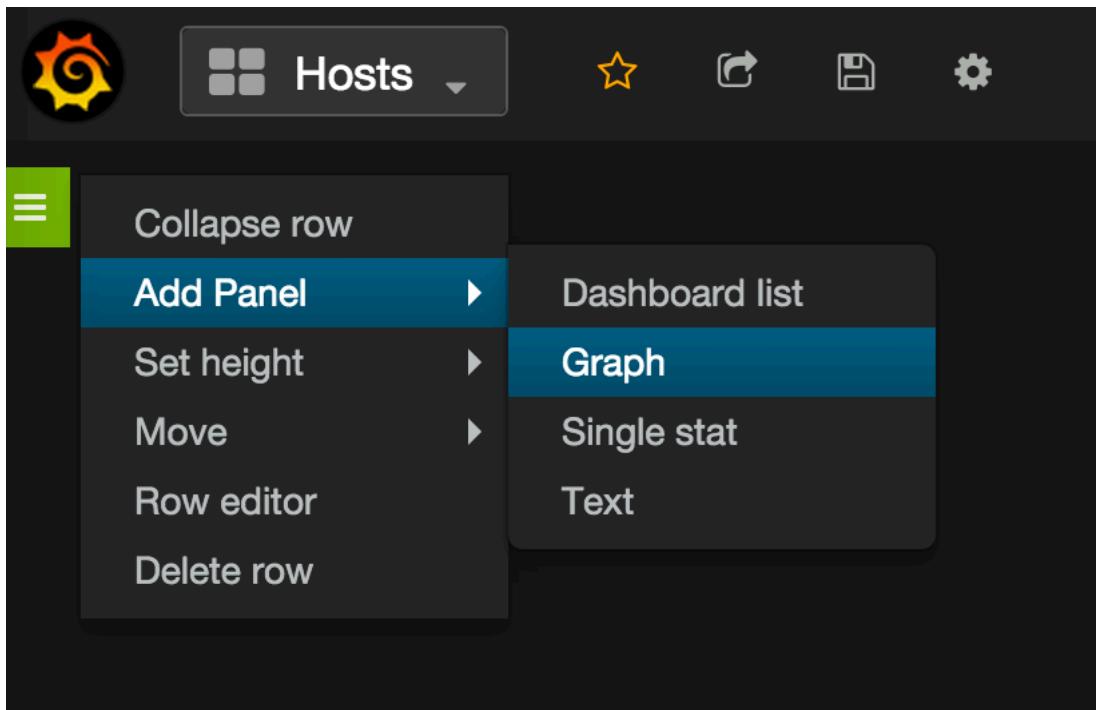


Figure 6.3: Creating our first host graph

Click on the **Panel Title** link to bring up the edit box, then click **Edit** to start building our graph.

Let's first give our graph a name by clicking the **General** tab and adding a title for the graph in the **Title** box. We're going to call this graph **CPU Usage**.

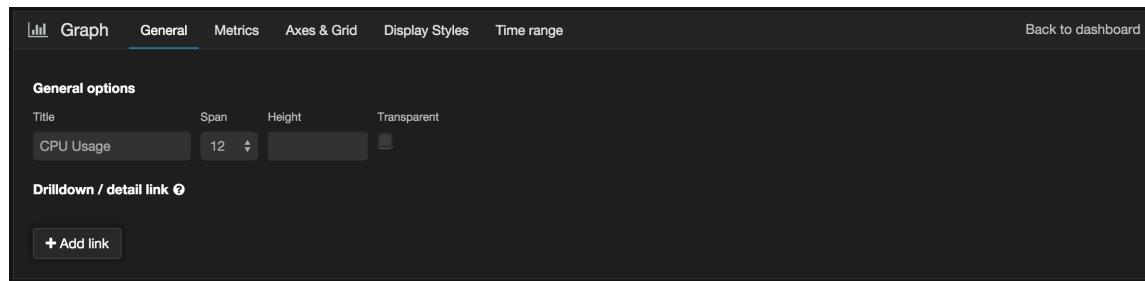


Figure 6.4: Naming our CPU Usage graph

The first graph we're going to create monitors CPU on all our hosts and displays it in graph form. To construct this graph we're not going to select a single, specific

metric. Instead we’re going to construct an algorithm that will use several metrics to display CPU usage across all our hosts. We’re going to sum the `user` and `system` CPU usage metrics and display the resulting sum per host, so we have a combined usage value for each host.

To do this we select the `Metrics` tab then the dropdown icon and we select `Switch editor mode` to specify our graph definition. We then add the following to the input:

Listing 6.22: CPU Usage graph definition

```
groupByNode(productiona.hosts.*.cpu.{user,system},2,'sumSeries')
```

When we looked at Grafana in Chapter 4 we specified a graph by using a single metric. This example shows us the next level of using Grafana: using functions to produce a graph. Functions are used to transform, combine, and perform computations on metrics. Graphite functions are defined roughly like:

Listing 6.23: Graphite functions

```
function(series_list_of_metrics,parameter_foo,parameter_bar)
```

In this case we’re using a function called `groupByNode`. The `groupByNode` or “group by node” function takes what Graphite calls a “series list.” A series list is essentially the name of a metric or series of metrics. We’re not just selecting a single metric—we’re selecting a series of metrics like so:

Listing 6.24: A Graphite series list

```
productiona.hosts.*.cpu.{user,system}
```

We select the metrics in two ways:

- Via a wildcard, *
- Via a set of path names in curly braces: { }

Our wildcard selects the host portion of the metric name, selecting all hosts recording this metric. Our curly braces enclose both the `user` and `system` paths, selecting every metric with either name.

We pass our series list of metrics to the function as a callback. Our function takes two parameters:

- An identifier for the element we're grouping by, here `2`.
- The operation to perform on the series list, `sumSeries`.

Our first parameter, `2`, is the common element from the metric that we're going to group our nodes by. The `2` is a 0-indexed count from the start of the metric name. In our metric that element matches the hostname wildcard: `productiona.hosts.*`.

Listing 6.25: The 0-indexed metric name

```
productiona . hosts . *
    0      1      2
```

This tells the `groupByNode` function to group by the hostname from the metric. So our callback series will look like:

Listing 6.26: The `groupByNode` callback

```
sumSeries(productiona.hosts.riemannna.cpu.{user,system})  
sumSeries(productiona.hosts.graphitea.cpu.{user,system})  
sumSeries(productiona.hosts.tornado-web1.cpu.{user,system})
```

The second parameter, `sumSeries`, refers to the `sumSeries` function. This function sums metrics and returns the result. Here the `sumSeries` function will return:

Listing 6.27: The `sumSeries` function output

```
productiona.hosts.riemannna.cpu.user+system  
productiona.hosts.graphitea.cpu.user+system  
... .
```

The `sumSeries` function will add the values of the `productiona.hosts.riemannna.cpu.user` and the `productiona.hosts.riemannna.cpu.system` together to provide a single metric, grouped by the host, that will show CPU usage of the combined `user` and `system` metrics.

Once we've created the function and series list we click away from the input box, and Grafana will apply the specified definition. We'll then see our CPU Usage graph.

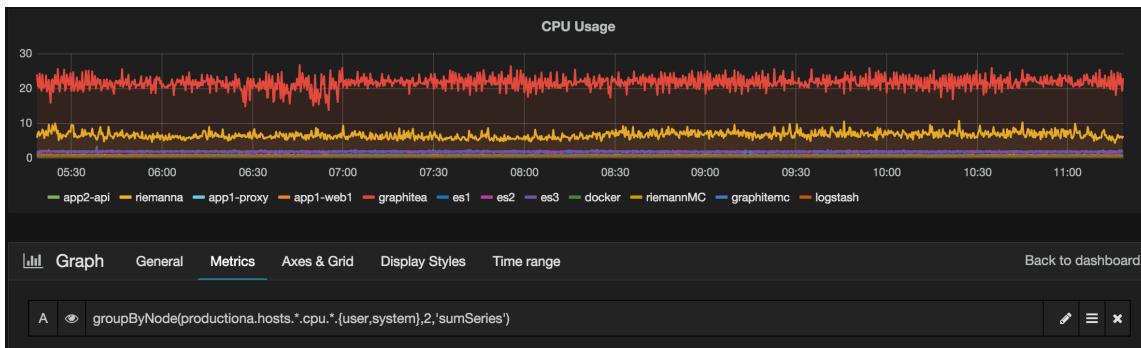


Figure 6.5: Our final CPU Usage graph

If we then click `Back to dashboard` and then the `Save dashboard` icon, we'll have saved our graph onto the dashboard.

We can then use the time resolution controls to specify the time period we're interested in and configure how often we'd like our graph's data to refresh using the `Auto-Refresh` controls.

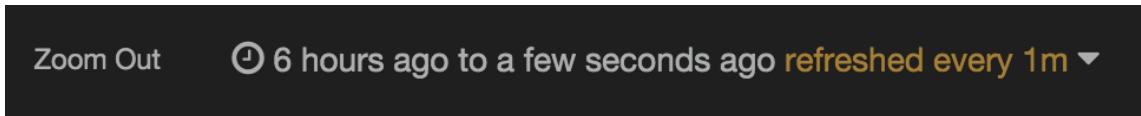


Figure 6.6: The time-series resolution controls

TIP You can find a full list of Graphite functions [in the documentation](#).

Creating a memory graph

We'll use the same principles to create another graph, this one showing memory usage. For this graph we're going to use the `productiona.hosts.*.memory.used` metric. As it is a complete metric, expressed as a percentage, we don't need to

sum any data. We do, however, want to use another function: `aliasByNode`. The `aliasByNode` function tells Grafana to name each metric we want to graph using an element of our metric path rather than the whole metric path. In this case we want to name each metric using the host name of each host being monitored.

Let's create our memory graph by clicking on the green row bar and selecting `Add Panel -> Graph`. Again we click on the `Panel Title` link to bring up the edit box and then click `Edit` to start building our graph. We give our graph a name by clicking the `General` tab and adding a title for the graph in the `Title` box. We're going to call this graph `Memory Usage`.

We then select the `Metrics` tab, then the dropdown icon, and select `Switch editor mode` to specify our graph definition. We add the following to the input:

Listing 6.28: Memory Usage graph definition

```
aliasByNode(productiona.hosts.*.memory.used,2)
```

Here we have the `productiona.hosts.*.memory.used` metric. Note the wildcard after the `productiona.hosts.` element. This tells us to select all hosts sending this metric. The metric is then wrapped in the `aliasByNode` function. This uses the specified parameter, here `2`, as the name of each metric. In our zero counted view of the metric name this is the third field and hostname element we've just wild-carded. So, if we were to receive a metric entitled `productiona.hosts.riemann.memory.used`, this graph would refer to that metric as `riemann`.

Click `Back to dashboard` to save our metric input, and then click `Save dashboard` to save our new memory graph.

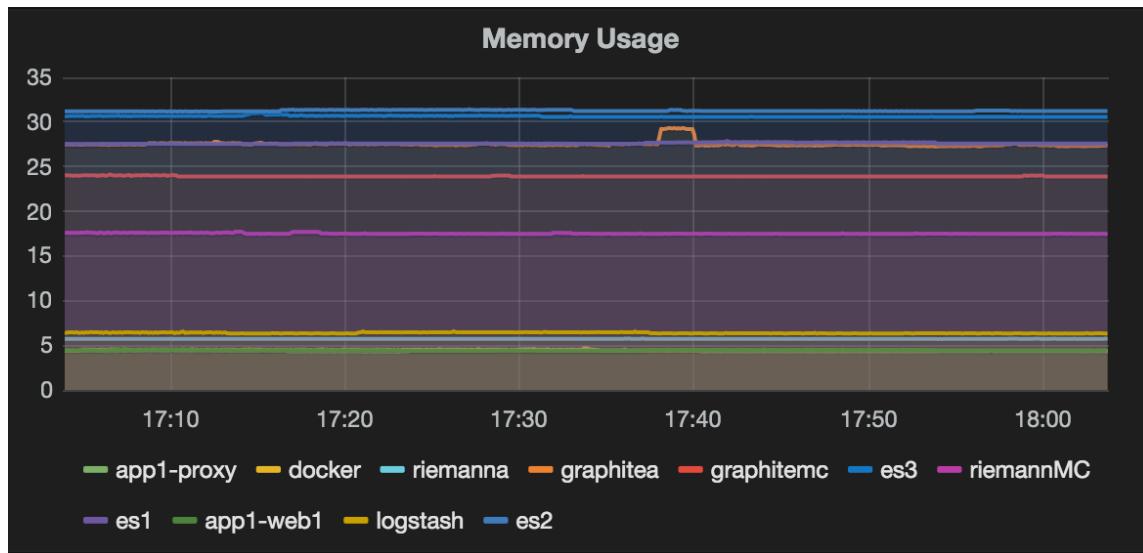


Figure 6.7: The Memory usage graph

Single host graphs

We're also going to create a host-centric graph for a single host—for example, making use of the CPU percentile metrics we created earlier. Let's build a CPU utilization graph for our `graphitea` host. We'll start by creating a new graph and calling it `graphitea CPU`.

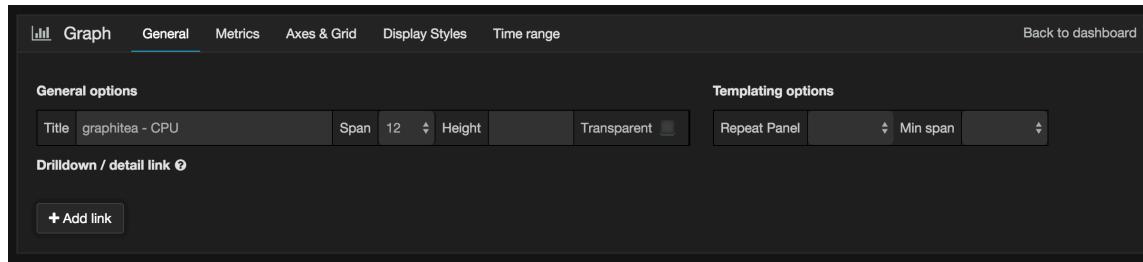


Figure 6.8: The graphitea CPU graph

We'll then configure a metric name in the `Metrics` tab. We're going to select multiple metrics for a single host using a wildcard, `*`, in our metric path. We're

again going to use the `aliasByNode` function to update the legend of our metrics. Instead of a single path entry as the alias, though, we're selecting two elements: the host name, `graphitea`, and the specific percentile value of each metric.

```
aliasByNode(productiona.hosts.graphitea.cpu.percent-user.*,2,5)
```

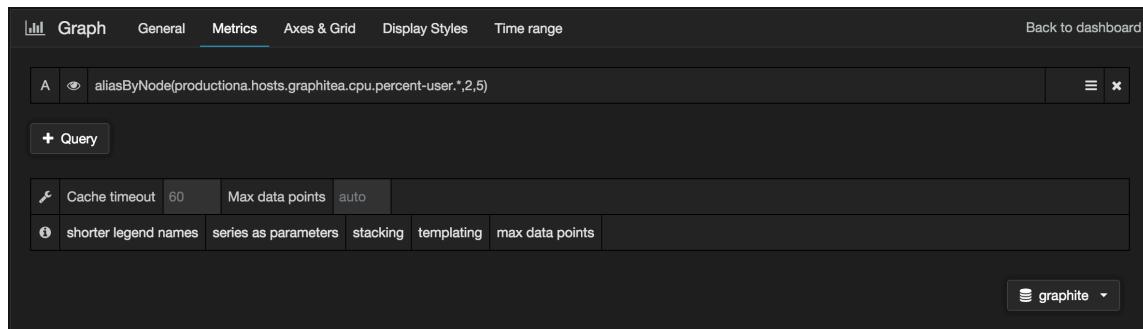


Figure 6.9: The single host Metrics tab

This will turn the metric legend into `graphitea.5`, `graphitea.95`, `graphitea.99`, etc.

If we then save our resulting graph we see CPU performance measured in the median, 95th, and 99th percentile, as well as the maximum value.

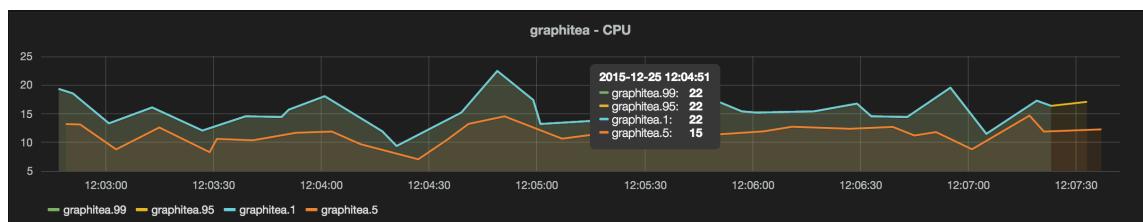


Figure 6.10: The single host CPU graph

Additional graphs

From here we could easily create graphs from other metrics. Here are some additional graphs we could create from the data we're gathering.

Metric	Formula
Disk used on the / (root) partition	<code>aliasByNode(productiona.hosts.*.df.root.percent_bytes.used,2)</code>
Load average	<code>aliasByNode(productiona.hosts.*.load.shortterm,2)</code>
Zombie processes	<code>aliasByNode(productiona.hosts.*.processes.zombies,2)</code>

TIP The [Graphite functions documentation](#) has many examples of metrics—especially collectd metrics—and how you might use them.

This collection of metric formulae would then result in a dashboard that shows the high-level current state of host-based metrics—like CPU, memory, and disk—in our environment.

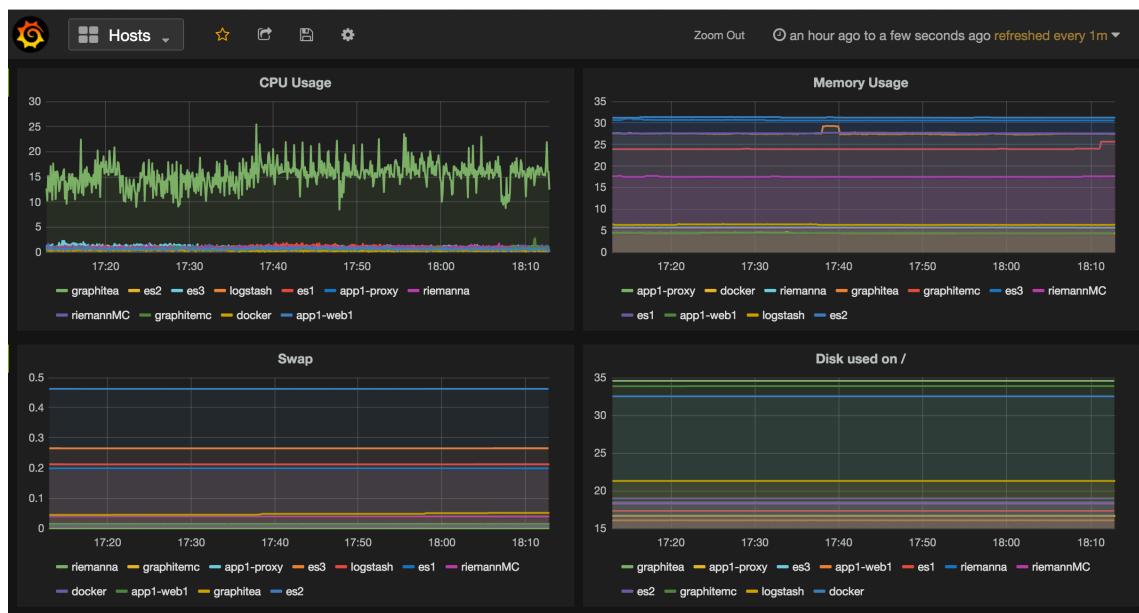


Figure 6.11: A Hosts Dashboard

Network, device, and Microsoft Windows monitoring

One of the areas we’re not covering in any detail in this book is monitoring network devices, data center equipment, or Microsoft Windows. However, none of the solutions in the book actively preclude doing this, and many are amenable to supporting this additional monitoring.

We can use collectd to monitor network devices remotely, such as using the [ping](#), [curl](#), or [SNMP](#) plugins to monitor devices we can’t run a collectd agent on.

Many devices support collection via an API—for example, see the [NetApp](#) plugin for monitoring NetApp appliances. We’ll also see more examples of scraping data from endpoints or via an API in Chapters 7 and 9.

Additionally, in Chapter 8 we talk about logging, and most of these kinds of devices generate Syslog-style log output. We could configure them to output to our logging environment to supplement existing monitoring.

Lastly, for Microsoft Windows, there are several collectd compatible agents like [CollectM](#) and [SSC Serv](#).

Alternatives to collectd

There are a number of commercial and open-source alternatives to collectd. This is not a definitive list but a sampling of some of the more interesting tools that you can use if collectd isn’t suitable or to your taste.

Commercial tools

- [New Relic](#) — Commercial Application Performance Management tool. It also includes some support for host, service, and availability monitoring.

- [Circonus](#) — Commercial Software-as-a-Service (SaaS) monitoring solution.
- [DataDog](#) — A SaaS console and collection center for monitoring and metrics data.
- [Librato](#) — A SaaS console and collection center, primarily focused on metrics data.

NOTE Some of these SaaS tools may do more than just collection and can include application monitoring, application performance management, and notification.

Open source

- Square's [Cube](#) and [Cubism](#) — Cube allows the collection of time-series events and Cubism is a [D3](#)-based visualization tool for those collected metrics.
- [Ganglia](#) — A monitoring tool with a focus on clusters and grids.
- [Munin](#) — A popular metric and monitoring tool that uses [RRDTool](#).
- [StatsD](#) — A network daemon that listens for statistics sent over the network and sends aggregates to one or more pluggable back-end services, including Graphite. We'll see more of StatsD in Chapter 9 of the book.
- [Diamond](#) — An open-source metrics collector originally written by Brightcove but now maintained by a wider community.
- [Fullerite](#) — An open-source metrics collector written by the Yelp Engineering team. It's written in Go and designed for large scale metrics collection.
- [PCP](#) and [Vector](#) — Used by Netflix this combination provides high resolution on host performance metrics suitable for diagnostics.
- [sumd](#) — A lightweight Python collector that allows you to run processes, for example Nagios plugins, locally and send the results to Riemann.

NOTE There's also some overlap with these tools and the collection and graphing tools we looked at in Chapters 3 and 4.

Summary

In our Riemann configuration we saw how we can make use of this data to monitor our hosts and their components, and how we can notify on specific events or thresholds.

We also saw how to create a basic host-centric dashboard and related graphs in Grafana.

In the next chapter we'll look at another kind of host: containers.

Chapter 7

Containers: another kind of host

With the rise of container virtualization it's also important to understand how to monitor metrics on containers. Containers are a form of operating system-level virtualization, most clearly epitomized by [Docker](#). Rather than running a full scale hypervisor like a traditional virtual machine they use kernel features like namespaces and [cgroups](#) to create small and lightweight compute instances.

NOTE We're going to focus on monitoring Docker-specific containers in this chapter. There are other emerging container platforms but none with the depth of adoption. If that changes the chapter will be updated to include them.

We're going to discuss the challenges of monitoring lightweight and fast-moving containers and how to make use of Docker's API to monitor metrics. We'll build on the work we did in Chapter 5 and use a collectd plugin to query statistics from that API and return system metrics, like CPU and memory, for each container running on our Docker daemon. In Chapter 8 we'll look at integrating Docker containers' log output into our monitoring environment.

NOTE This section assumes you've got Docker installed and running on a host. It doesn't show you how to install or manage Docker. If you're interested in that then we've got a book for you that's all about it: [The Docker Book](#).

Challenges with container monitoring

We've talked a little about how host-centric monitoring has changed as a result of faster-moving infrastructure, like the use of Cloud and virtualized hosts. This often means that the lifespan of a host is much shorter than it was in the past. With containers, which are designed to be lightweight and short lived, this is further exacerbated. Further, with containers being so lightweight, we often want to ensure capacity is focused on performing their designated purpose, rather than consumed by overhead from monitoring. This diagram articulates the changes that have occurred in compute architecture.

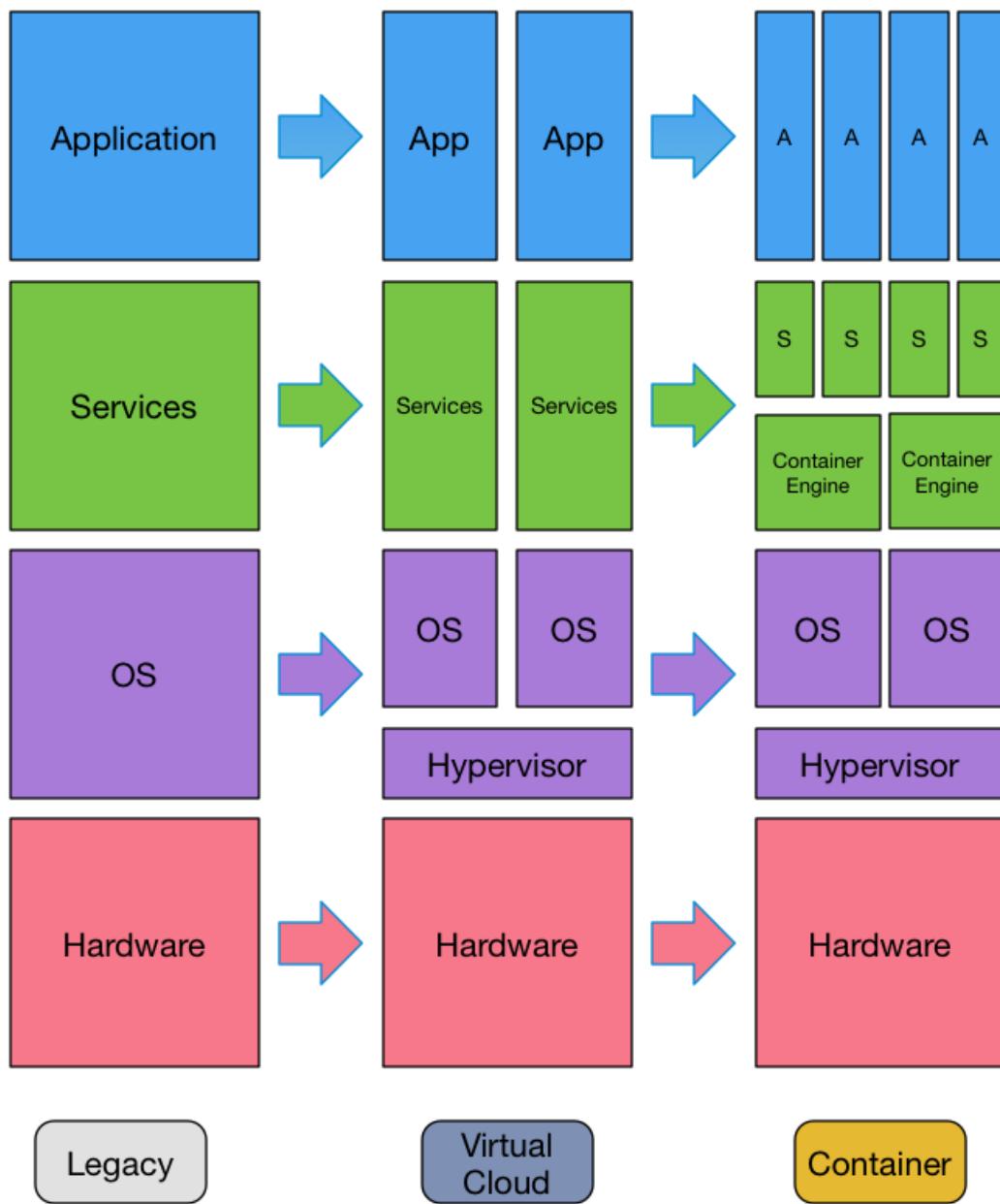


Figure 7.1: Host evolution

NOTE This diagram is based on one published in [this DataDog blog post](#). The

original image is copyright DataDog and modified with their permission.

The far left column is legacy applications. Generally legacy applications were run on a series of single physical hosts. This host was long lived and hence easy to monitor: there was a single set of metrics that was tracked over time. This was the typical host architecture of about 15–20 years ago.

The middle column shows the change to hosts introduced by virtualization, largely VMWare-driven, and the Cloud, largely the result of Amazon Web Services. Here two or more virtualized or cloud instances were running on top of a single hardware host. Correspondingly, operational and monitoring complexity increased as the number of hosts increased and the volume of metrics created was multiplied. Additionally, the lifespan of those instances tended to be shorter than those of single physical hosts. This model started to appear about 10 years ago.

The far right column represents a container-centric host architecture. Here we have a hardware host running a virtual environment (or Cloud instances), on top of which we're running a container service. In this world operational complexity is high. The application architecture can require understanding the interconnection between multiple containers, instances, and hosts. Added to this, [the lifespan of a container might be in seconds or minutes](#). This makes the traditional monitoring techniques used for a single host or instance problematic.

From a monitoring perspective there are three major issues with this new host model:

- Convergence and dynamism.
- Performance
- Metric volume

Let's first talk about convergence and dynamism. The speed and limited lifespan means a lot of churn in your monitoring configuration: hosts appearing and disappearing quickly. Sometimes a host will even appear and disappear before your

monitoring environment is aware of it. In many monitoring environments your configuration is applied after the installation of the host or service, either manually or via a configuration management tool like Puppet or Chef. In either case there is some delay between a host or service being active and it being monitored, both in terms of checks of its status and collection of any performance data. With containers, that delay could mean that the host or service will live a shorter period than the time required to converge your monitoring configuration.

There is one principal strategy we're going to employ to address our convergence challenge: the push-based architecture of our framework. If a host or service appears it can be preconfigured to emit events and metrics as soon as it appears rather than having to wait for it to be discovered or configured and then monitored. If something occurs to our container or the service or applications inside it, we'll have a record of it and the ability to notify. We'll make use of both the collectd infrastructure we created in Chapter 5 and logging to collect the relevant data. (More on the logging setup in Chapter 8.)

TIP A good rule of thumb is Your monitoring system should be at least as dynamic as your infrastructure.

Second, for performance reasons, we want to ensure whatever monitoring we do on the host is as low impact as possible. This generally means avoiding approaches like installing agents or running additional processes inside the container. We're going to focus on approaches that require as little overhead as possible, such as focusing monitoring on extracting information from the Docker daemon and API layer rather than from containers directly.

Our final issue is the metric volume produced by the container-based model. If we compare our typical metric volume:

Physical × Virtual × Services × Apps = Metrics

Versus our container metric volume:

Physical × Virtual × Containers × Services × Apps = Metrics

We see that adding the container layer adds a significant multiplier to the number of metrics being generated. There's not much we can do about the incoming volume of metrics beyond setting up our Graphite environment to be as optimized as possible. We can, however, consider retention and storage of metrics, especially ones generated by short-lived containers. Later in the chapter we're going to discuss setting a different retention for Docker metrics and cleaning up metrics that have not been updated recently.

Monitoring Docker containers

We're going to use our existing collectd infrastructure to monitor Docker and Docker containers. There isn't a default plugin for Docker in collectd yet, but we can take advantage of collectd's plugin framework and drop in an open-source plugin to do the monitoring.

Let's start by looking at Docker's monitoring capabilities to understand what's available to us and how this plugin gets its data. In the case of Docker, a daemon running on the host creates and manages any containers. Users interact with this daemon via a command line tool called `docker` or via an API that allows a user to manage and query data from the Docker daemon. Let's look at what data we get out of Docker containers using the binary and the API.

The `docker` binary can reveal the performance statistics of running containers. It has a sub-command called `stats` that takes the name or ID of one or more containers. The command returns a live stream of container resource information.

Listing 7.1: The docker stats command

```
$ docker stats web_1 db_1
CONTAINER CPU % MEM USAGE / LIMIT      MEM % NET I/O
      BLOCK I/O
web_1      0.00% 26.6 MB / 1.042 GB  2.55% 3.48 kB / 1.084 kB
          12.75 MB / 0 B
db_1       0.13% 13.46 MB / 1.042 GB 1.29% 1.732 kB / 2.922 kB
          6.828 MB / 0 B
```

Here we've returned the statistics of the `web_1` and `db_1` containers. We see the CPU and memory used as well as some network and block I/O usage data.

There's also a Docker API endpoint called `stats` that returns more detailed data for a container. Let's use the `curl` binary to query the Docker API for some container statistics.

TIP You can read more about the API in [the Docker API reference](#).

Listing 7.2: Querying the Docker API stats endpoint

```
$ curl http://127.0.0.1:2375/containers/bb82f35bab39/stats?stream=false | python -mjson.tool

{
    "blkio_stats": {
        "io_merged_recursive": [],
        "io_queue_recursive": [],
        "io_service_bytes_recursive": [
            {
                "major": 253,
                "minor": 0,
                "op": "Read",
                "value": 28672
            },
            {
                "major": 253,
                "minor": 0,
                "op": "Write",
                "value": 0
            },
            ...
        ]
    }
}
```

Here we see a much fuller set of metrics returned by the Docker API. We've curled the `stats` endpoint for a specific container, here identified by its container ID, `bb82f35bab39`. It's these metrics that we're going to extract via our Docker collectd plugin and send onto Riemann.

Docker collectd plugin

We're going to make use of a [collectd plugin specifically designed to collect Docker container metrics](#). We're going to install it on one of our Docker hosts, originally named `docker1.example.com`. We've also installed and configured collectd on that host as per the instructions specified in Chapter 5.

Our plugin uses [the Docker stats API](#) to collect statistics on the containers running on the Docker daemon.

The following container stats are reported for each container:

- Network bandwidth
- Memory usage
- CPU usage
- Block IO

The Docker collectd plugin is a third-party plugin that doesn't ship with the core collectd product. Third-party plugins for collectd can be written in a variety of languages. They can be written in C and run directly by collectd, or they can be written in other languages and executed using helper plugins.

These helper plugins include the [Exec plugin](#) which executes scripts and returns anything they produce to `STDOUT` and onto collectd. Another is the [Python plugin](#) which specifically executes Python-based scripts and returns their output to collectd. The [Exec](#) and [Python](#) plugins are installed by default on both Ubuntu and Red Hat distributions when you install collectd.

The [Python](#) plugin is a bit more elegant than the [Exec](#) plugin. The [Exec](#) plugin starts a new process for each run of a plugin, shelling out each time. This consumes a reasonable amount of resources and isn't overly optimal. The [Python](#) plugin provides an interpreter that is run by the collectd daemon and doesn't spawn repeated external processes. But as a result the [Python](#) plugin requires somewhat more sophisticated development effort than the [Exec](#) plugin.

TIP There is also a [Perl](#) and a [Java](#) plugin that provide similar capabilities for plugins written in Perl and Java.

Installing the Docker collectd plugin

In this case our new Docker collectd plugin is written in Python. We're going to show you how to do a manual installation, but the whole process would be much better managed using a configuration management tool.

We're going to store our new plugin with some of our existing collectd plugins in the `/usr/lib/collectd/` directory.

TIP If you've installed collectd then this directory already exists on Ubuntu, Debian, and most Red Hat family distributions.

We download a Zip file containing the latest version of the plugin to this directory.

Listing 7.3: Downloading the Docker collectd plugin

```
$ cd /usr/lib/collectd/  
$ sudo wget https://github.com/jamtur01/docker-collectd-plugin/  
archive/master.zip
```

Then we unzip it.

TIP You might need the `unzip` command on your host. If so, you can install it via the `unzip` package.

Listing 7.4: Unzip the master.zip file

```
$ sudo unzip master.zip
Archive: master.zip
4f9dd778f30ba82cdcf27653d219dfde4232c925
  creating: docker-collectd-plugin-master/
  extracting: docker-collectd-plugin-master/.gitignore
  inflating: docker-collectd-plugin-master/LICENSE
  ...
  
```

Now let's rename our directory and clean up our Zip file.

Listing 7.5: Rename the Docker collectd plugin directory

```
$ sudo mv docker-collectd-plugin-master docker
$ sudo rm master.zip
```

This will result in our Docker collectd plugin being installed in the:

`/usr/lib/collectd/docker`

directory.

The plugin has some Python library prerequisites that we also need to install. We do this via the Python `pip` command. If you don't have the command on your host you can install it via the `python-pip` package on Ubuntu natively. On Red Hat you will need to enable the EPEL package library before installing the `python-pip`

package.

Listing 7.6: Installing the docker-collectd prerequisites via pip

```
$ cd /usr/lib/collectd/docker  
$ sudo pip install -r requirements.txt
```

This will install some additional Python libraries including the Docker Python API library that the plugin uses to connect to the Docker API.

Configuring the Docker collectd plugin

Now that we've installed the plugin we need to configure it. Let's add a file to hold our configuration.

Listing 7.7: Creating the /etc/collectd.d/docker.conf file

```
$ sudo touch /etc/collectd.d/docker.conf
```

Let's now populate this file with our configuration.

Listing 7.8: Configuring the Docker collectd plugin

```
TypesDB "/usr/lib/collectd/docker/dockerplugin.db"
LoadPlugin python

<Plugin python>
    ModulePath "/usr/lib/collectd/docker"
    Import "dockerplugin"

    <Module dockerplugin>
        BaseURL "unix://var/run/docker.sock"
        Timeout 3
    </Module>
</Plugin>

<Plugin "processes">
    Process "docker"
</Plugin>
```

This collectd configuration is a little more complex than what we've previously seen.

The first option, **TypesDB**, extends collectd's base list of metrics. The collectd daemon has a **types database**, which can generally be found at [/usr/share/collectd/types.db](#), that defines each metric collectd can generate by name and type. The **TypesDB** allows you to specify a file that includes new metrics that will be appended to the core list of metrics.

Due to a quirk in collectd, when we specify custom metrics in a new types database we need to explicitly specify the default types database. To do this we need to add the following line to our [/etc/collectd/collectd.conf](#) on the host running

Docker:

Listing 7.9: Adding the TypesDB default configuration

```
TypesDB "/usr/share/collectd/types.db"
```

The second option, `LoadPlugin`, loads the `python` plugin we're using to execute our Docker collectd plugin.

We then use the `<Plugin>` block to tell collectd where to find the Docker collectd plugin and how to load it. We specify the path to the plugin and we use the `Import` command to import the specific plugin file, `dockerplugin`. This is the name of the file that contains the Python code that gets the Docker container metrics (we drop the `.py` extension).

Inside this block we then specify a `<Module>` block to configure the Docker collectd plugin itself. We use the `BaseUrl` option to specify the path to the Docker daemon's socket—by default it is usually at `/var/run/docker.sock`. We specify the `Timeout` option to control when to time out requests to the Docker daemon.

We also add some configuration for the `processes` plugin to enable monitoring of the Docker daemon.

Listing 7.10: Monitoring the Docker daemon

```
<Plugin "processes">
    Process "docker"
</Plugin>
```

This builds on the configuration from Chapter 5 and will ensure we're notified when the Docker daemon itself is not running.

We then restart collectd to enable the plugin and our process monitoring.

Listing 7.11: Restarting collectd to enable the Docker plugin

```
$ sudo service collectd restart
```

You should see some log entries in the `/var/log/collectd.log` log file to reflect the Docker collectd plugin being started.

Listing 7.12: The Docker collectd plugin log output

```
[2015-08-25 06:43:22] Collecting stats about Docker containers  
from unix://var/run/docker.sock (API version 1.20; timeout: 3s)  
  
[2015-08-25 06:43:22] Initialization complete, entering read-loop
```

Now our events should be collected from the Docker stats API by the plugin and passed through to Riemann for processing.

Processing Docker collectd statistics with Riemann

Our Docker collectd events are a little different from our other collectd events—they are generated from containers run by our Docker daemon on a host. Let's create a new Docker container and start to monitor it to see how this works. We're going to assume you already have Docker installed.

We're going to run a new container that runs a Redis database. Let's start our container now.

Listing 7.13: Starting our Redis container

```
$ docker run --name redis -d redis
Unable to find image 'redis:latest' locally
latest: Pulling from library/redis
...
Status: Downloaded newer image for redis:latest
341163d2a74f3c3c8c7719bc5e226554d684f1d82f181a6689d99dc257167960
```

We can see we've launched a new container in daemonized mode, the `-d` flag, with a name of `redis` and using the image `redis`. Our launch has downloaded the image, and we've run a container from the image with an ID of:

`341163d2a74f3c3c8c7719bc5e226554d684f1d82f181a6689d99dc257167960`

Let's see a bit more about that container.

Listing 7.14: Running docker ps

```
$ docker ps
CONTAINER ID IMAGE COMMAND                  CREATED      STATUS
PORTS      NAMES
341163d2a74f redis " /entrypoint.sh redis"  2 mins ago Up 2 mins
6379/tcp   redis
```

With the container running the Docker collectd plugin, which is querying the Docker API endpoint at `containers/341163d2a74f/stats`, via the Docker Unix socket. Let's look at one of the events from our Docker collectd plugin to see what

this means.

Inside our `riemann.config` we can capture just our Docker collectd events by using a `where` filtering stream.

Listing 7.15: A where filter for our Docker collectd events

```
(where (= (:plugin event) "docker")
  #(info %))
```

Here we've selected all events with a `:plugin` field value of `docker` and told Riemann to pass them to the log file. If we reload or restart Riemann we'll start to see events like the following in the log file.

Listing 7.16: A Docker collectd Riemann event

```
{:host docker1, :service docker-redis/cpu.percent, :state ok, :
  description nil, :metric 0.06, :tags [collectd], :time
  1440563513, :ttl 60.0, :ds_index 0, :ds_name value, :ds_type
  gauge, :type cpu.percent, :plugin_instance redis, :plugin
  docker}
```

We see that the event has the `:host` field set to `docker1`. That's the name of the host running the Docker daemon. The `:service` field has a value of `docker-redis /cpu.percent`. The first half of the `:service` field, before the `/`, is a combination of `docker-` and the name of a container, `redis` in this case. The second half is the name of the metric being sent—here that's `cpu.percent` or CPU percentage. This is constructed by the Docker collectd plugin.

NOTE Our event has also gotten the default TTL of 60 seconds provided by the default function we defined in Chapter 3.

This isn't overly useful because if we want to do something with this event, such as send it to Graphite, any metric path would be constructed using the `:host` and `:service` pair. Every Docker container would appear as a sub-metric underneath our `docker1` host rather than a host in its own right. The prefix in the `:service` field can also make parsing metrics confusing.

Additionally, if we want to notify about a specific Docker container, ensuring the `:host` and `:service` fields are set correctly will make that much easier and make it so we don't need to built custom logic to cater for Docker containers.

To counteract all this, and make each Docker container a first-class citizen in our metrics, we need to update the `:host` and `:service` fields with the name of the container and the name of the metric being collected.

We're also going to move any Docker containers to their own metric namespace. Remember the `productiona.hosts` prefix we added in Chapter 4? We're also going to update that prefix to be `productiona.docker` for any Docker containers.

Thankfully this will be pretty easy. We see that the host and service information is contained in two other fields in the event. The container name is specified in the `:plugin_instance` field and the metric name in the `:type` field. We just need to apply the values in these fields to our `:host` and `:service` fields.

To do this we're first going to need to exclude these events from being graphed with our other collected events so we don't end up with duplicate events being graphed. To exclude these events let's update the `where` stream we created for our previous collectd events.

Listing 7.17: The updated collectd Graphite where stream

```
(tagged "collectd"
  (where (not (= (:plugin event) "docker"))
    (smap rewrite-service graph))

  (where (= (:plugin event) "docker")
    (smap #(assoc % :host (:plugin_instance %)
      :service (cond-> (str (:type %)) (:type_instance %) (str "." (:type_instance %))))
    (smap rewrite-service graph))))
```

We're still grabbing all the events tagged with `collectd`. Inside the original `tagged` stream we're now further selecting events with two additional `where` streams.

The first `where` stream grabs any events not from our Docker plugin. These events are passed to an `smap` function, have their metric paths rewritten by the `rewrite-service` function, and then are passed to the `graph` var to be graphed in Graphite.

The second `where` stream selects only the Docker plugin events, taking all events with the `:plugin` field value set to `docker`. Inside that stream we've again used an `smap` stream. Remember that the `smap stream` is a streaming map. It takes events, applies a function, and then passes them on. Let's look at what happens in some more detail.

Listing 7.18: The Docker collectd plugin smap and graph

```
(where (= (:plugin event) "docker")
  (smap #(assoc % :host (:plugin_instance %)
                :service (cond-> (str (:type %)) (:type_instance %) (str "." (:type_instance %)))
                )))
  (smap rewrite-service graph)))
```

Our `smap` takes the events and passes them to the `assoc` function. The `assoc` function remaps some fields in our event. We first replace the value of the `:host` field with the value of `:plugin_instance` field. This replaces our existing `:host` field value of `docker` with a value of `redis`, the container name.

We next replace the value of the `:service` field by combining the `:type` and optionally the `:type_instance` field. We have to do this because the collectd plugin breaks up the type of metric between the `:type` and `:type_instance` fields. But several metrics don't define a `:type_instance` field—so to construct the `:service` field we need to specify the `:type` field and optionally the `:type_instance`, if it exists, separated with a `..`. To do this we use the `cond->` and `str` functions. The `cond->` function takes an expression and a set of test pairs. The `str` function returns a string. Combined we test if one or both fields is present and combine them into a string with a `.` separator if both are present. Otherwise we only write the field that is present—by default, that's `:type`.

Our new event will look like this for an event with only a `:type` field:

Listing 7.19: The updated Docker collectd :type Riemann event

```
{:host redis, :service cpu.percent, :state ok, :description nil,  
:metric 0.06, :tags [collectd], :time 1440563513, :ttl 60.0, :  
ds_index 0, :ds_name value, :ds_type gauge, :type cpu.percent,  
:plugin_instance redis, :plugin docker}
```

We see that the `:host` field is now set to `:redis` and the `:service` field is now set to `cpu.percent` which will provide a more simple, clear metric to use in Riemann and Graphite.

Alternatively, an event with both a `:type` and a `:type_instance` field will look like this:

Listing 7.20: The updated Docker collectd :type_instance Riemann event

```
{:host redis, :service memory.stats.total_pggout, :state ok, :  
description nil, :metric 2133.0, :tags [collectd], :time  
1440563513, :ttl 60.0, :ds_index 0, :ds_name value, :ds_type  
gauge, :type_instance total_pggout, :type memory.stats, :  
plugin_instance redis, :plugin docker}
```

The last thing our `smap` stream does is pass the event into (`smap rewrite-service graph`) to be sent onto Graphite and processed. This will create a new host in Graphite for each Docker container running on the Docker daemon and will start populating metrics data for that host.

With this in mind we want to make our next change: updating the metric name prefix. In Chapter 4 we configured the `graph` var to prefix our metric names with the environment and a type of metric, in our current case: `productiona.hosts`.

This was done by adding a function to the `graph` var's `:path` option. Let's revisit the code in:

`/etc/riemann/examplecom/etc/graphite.clj`

And specifically the `add-environment-to-graphite` function.

Listing 7.21: The graph var from Chapter 4

```
(defn add-environment-to-graphite [event] (str "productiona.hosts
  .", (riemann.graphite/graphite-path-percentiles event)))

(def graph (async-queue! :graphite {:queue-size 1000}
  (graphite {:host "graphitea" :path add-environment-to-graphite
    })))
```

We see that the `add-environment-to-graphite` function prefixes any metric name with `productiona.hosts`. Let's update this code to add a new prefix, `productiona.docker`, to any Docker container metrics. This will group all of our Docker containers in their own metric name.

Listing 7.22: The updated graph var

```
(defn add-environment-to-graphite [event]
  (if (= (:plugin event) "docker")
    (str "productiona.docker.", (riemann.graphite/graphite-path-
      percentiles event))
    (str "productiona.hosts.", (riemann.graphite/graphite-path-
      percentiles event)))))

(def graph (async-queue! :graphite {:queue-size 1000}
  (graphite {:host "graphitea" :path add-environment-
    to-graphite})))
```

We see that we've updated our `add-environment-to-graphite` function to add a conditional clause. We've used Clojure's `if` function to test if the `:plugin` field of the `event` is set to `docker`. The `if` conditional takes a test and if true returns one result and if false returns the next. In our `add-environment-to-graphite` function if the test is met then the metric name will be prefixed with `productiona.docker`. If not, the metric name will be set to `productiona.hosts`, our previous default.

Now our metrics will be sent through to Graphite with an appropriate host, service, and path set, for example `productiona.docker.redis.cpu.percent`.

Adding metadata to our Docker events

Our rewrite of the `:host` and `:service` fields helps us in many cases with our Docker containers. But it doesn't address a common Docker use case: multi-container services. A common deployment model for Docker containers is to run many containers as part of a clustered service—for example, a cluster of web servers or a cluster of caching servers.

We're going to address this concern by adding metadata to our Docker containers and then consuming that metadata inside Riemann. The preferred method of adding metadata to a Docker container or image is [via a label](#). Docker labels can be added to the Docker daemon or to a Docker image using the `LABEL` instruction in your [Dockerfile](#).

Listing 7.23: A Dockerfile label

```
...  
LABEL com.example.version="2.7.3"  
...
```

They can also be added at runtime using the `--label` flag on the `docker run` command.

Listing 7.24: Adding a label at runtime

```
$ sudo docker run --name daemon_dave --label com.example.  
application="tornado" -d run
```

Docker recommends that label keys be specified with namespaces that use reverse domain notation, for example:

`com.example.label`

This reduces the risk of name collisions. The Docker team recommends some other best practices for label keys:

- Keys should only consist of lower-cased alphanumeric characters, dots, and dashes (for example, [a-z, 0-9, - and .]).
- Keys should start and end with an alphanumeric character.

- Keys may not contain consecutive dots or dashes.

Let's recreate our `redis` container with a label.

Listing 7.25: Rerunning the redis container

```
$ sudo docker run --name redis --label com.example.application="tornado" -d redis
```

Here we've created a new `redis` container with the label of:

`com.example.application="tornado"`

We inspect our container with the `docker inspect` command and see the label.

Listing 7.26: Inspecting the redis container

```
$ sudo docker inspect -f "{{json .Config.Labels }}" redis
{"com.example.application": "tornado"}
```

We've inspected our `redis` container with the `-f` flag to select only the `"{{json .Config.Labels }}"` section of our container configuration. Our single label has been returned:

`"com.example.application": "tornado"`

Our collectd plugin detects the presence of any labels and adds them to every event collected by the plugin. Unfortunately, collectd's event format is not readily extendable so it adds the labels to the `:plugin_instance` field as a suffix to the existing content, like so:

`:plugin_instance "redis[com.example.application=tornado]"`

NOTE Even worse than the lack of extendability, collectd's existing fields have a 64-character limit. This means one or more labels that total more than 64 characters will be truncated.

The collectd Docker plugin does this by creating a formatted dictionary of comma-separated key/value pairs.

Listing 7.27: The Docker collectd plugin format

```
def _d(d):
    """Formats a dictionary of key/value pairs as a comma-
    delimited list of
    key=value tokens."""
    return ','.join(['='.join(p) for p in d.items()])
```

NOTE I've added this capability in a fork of the Docker collectd plugin. It's in a [pull request](#) on [the upstream plugin](#).

These events are then sent to Riemann. We'll need to update our Docker event processing to extract this data. To do this, we're going to create new functions in the top of our existing:

`/etc/riemann/examplecom/etc/collect.clj`

file. Let's start with two functions to extract any labels from the `:plugin_instance` field and convert them into a map. The first function will do the label extraction and we've called it: `docker-attribute-map`.

Listing 7.28: The plugin-map function

```
(ns examplecom/etc/collectd
  (:require [clojure.tools.logging :refer :all]
            [riemann.streams :refer :all]
            [clojure.string :as str]
            [clojure.walk :as walk]))


(defn docker-attribute-map
  "Parses labels from collectd plugin_instance"
  [attributes]
  (let [instance (str/split (str/replace attributes #"^.*\[([.*)\]"
                                             "$ \"$1\") #\",")]
    (walk/keywordize-keys (into {} (for [pair instance] (apply
      hash-map (str/split pair #\"=)))))))
```

Our existing `collectd.clj` file contains existing `require` statements for the `clojure.string`, `riemann.streams`, and `clojure.tools.logging` libraries. We've now added an additional `require` for the `clojure.walk` library. This library contains a function called `keywordize-keys` that converts map keys from strings to keywords that we're going to use shortly to convert our Docker label keys from strings.

Next, we define the `docker-attribute-map` function. It takes an argument called `attributes` which is the `:plugin_instance` field. We then start the function with a `let` statement that creates a new var called `instance`. It uses a regular expression to extract the labels contained within the field. It extracts the content between the `[]` and then splits the results on any commas.

We then use this new var inside our next line which loops through the `instance` var, extracts each pair of key/values, splits them on `=`, and inserts them into a

map. We wrap the final result with `clojure.walk's keywordize-keys` function which will turn our key into a keyword. We end up with a map like:

Listing 7.29: The label map

```
{:com.example.application "tornado"}
```

Now let's add a second function, below our `docker-attribute-map` function. We'll call this function `docker-attributes`. This function's job is to check if our event has any attributes and if it does send it off to be mapped. We do this because not every Docker event will have attributes added and we only want to process those that do.

Listing 7.30: The docker-attributes function

```
(defn docker-attributes
  [{:keys [plugin_instance] :as event}]
  (if-let [attributes (re-find #".*\[.*\]$" plugin_instance)]
    (merge event (docker-attribute-map attributes))
    event))
```

Our `docker-attributes` function takes an argument of the `:plugin_instance` field. Our argument uses a new Clojure construct called `destructuring`. Destructuring is a way to extract values from a data structure and bind them to symbols without having to explicitly traverse the whole structure. They are commonly used with vectors, maps and with function arguments, as we've done here.

Our argument contains a keyword, `:keys`, that is a shortcut for extracting a value, and the `:as` keyword which allows us to assign that value to a symbol.

Listing 7.31: Destructuring keys

```
[{:keys [plugin_instance] :as event}]
```

So here we're expecting that our `docker-attributes` function will take an event as an argument. We'll extract the contents of the `:plugin_instance` field from that event and assign it to a symbol of `event`.

TIP We've already seen one kind of destructuring earlier in the book when we used `&` to specify optional arguments in Clojure functions.

We then use a variant of the `let` expression called `if-let`. The `if-let` expression is a conditional binding.

Listing 7.32: The if-let binding

```
(if-let [attributes (re-find #".*\[.*\]$" plugin_instance)] ...)
```

The `if-let` expression will test if the `:plugin_instance` field contains attributes after the container name, for example:

```
redis[com.example.application=tornado]
```

If the test is true, then it evaluates a condition with binding bound to the value of the test. If it is false then it evaluates an `else` condition. Our conditions are:

Listing 7.33: The if-let conditions

```
(merge event (docker-attribute-map attributes))  
event))
```

The first condition is used if the test, whether the field contains attributes, is true. We `merge` our existing event with the map of attributes created by the `docker-attribute-map` function.. This extracts the attributes from the `:plugin_instance` field and creates them as new fields in the event, so that:

```
{:plugin_instance redis[com.example.application=tornado]}
```

Will become:

```
{:com.example.application tornado}
```

Our second condition is our `else` condition, if the result of the test is false. This condition returns the existing `event`, unchanged.

Finally, let's add a third function to replicate our existing event parsing and rewriting of the `:host` and `:service` fields. We'll call this function `docker-parse-service-host` and add it below our `docker-attribute-map` and `docker-attributes` functions in the `collectd.clj` file.

Listing 7.34: The docker-parse-service-host function

```
(defn parse-docker-service-host
  [{:keys [type type_instance plugin_instance] :as event}]
  (let [host (re-find #"\w+\.?\w+\.?\w+" (:plugin_instance event))
        service (cond-> (str (:type event)) (:type_instance event)
                           (str "." (:type_instance event)))]
    (assoc event :service service :host host)))
```

We again use destructuring to extract the values of the `:type`, `:type_instance`, and `:plugin_instance` fields and assign them to a symbol called `event`. We then bind some vars using `let`. We use the same code we developed earlier in the chapter to rewrite our `:host` and `:service` fields. We then use the `assoc` function to update the fields with their new content..

To make use of these functions let's now update the code that parses our Docker events in `riemann.config`.

Listing 7.35: Processing our Docker events

```
(where (= (:plugin event) "docker")
      (smap (comp parse-docker-service-host docker-attributes rewrite
                  -service) graph))
```

We still select our Docker events with a `where` stream. We then pass our events into an `smap`. We use a new function: `comp`, which is short for composition. The `comp` function returns a composition of all the listed functions. So our `smap` sends the event through the `parse-docker-service-host` function, the `docker-attributes`

function and the `rewrite-service` function. The combined event, which should now be totally processed, is then sent onto the `graph` var and then to Graphite.

Now a Docker event with a label will look like:

Listing 7.36: A rewritten Docker event Mark II

```
{:host redis, :service memory.stats.hierarchical_memory_limit, :  
  state ok, :description nil, :metric 1.8446744073709552E19, :  
  tags [collectd], :time 1458310314, :ttl 60.0, :com.example.  
  application tornado, :ds_index 0, :ds_name value, :ds_type  
  gauge, :type_instance hierarchical_memory_limit, :type memory.  
  stats, :plugin_instance redis[com.example.application=tornado],  
  :plugin docker}
```

Our `:host` and `:service` has been rewritten to make our metrics more parseable. We also see a new field parsed from our label:

`:com.example.application tornado`

We can now make use of that field (or fields from other labels) in checks. For example, we can group checks by all the containers that specify a given label. We can also use that label or others to bundle together metrics from specific applications in Graphite. We can do this by further updating our metric name rewriting. Let's go back to our `add-environment-to-graphite` function in the:

`/etc/riemann/examplecom/etc/graphite.clj`

file.

Let's make a change to use the contents of the `com.example.application` field in our metric name.

Listing 7.37: Updated our Graphite metric name write

```
(defn add-environment-to-graphite [event]
  (condp = (:plugin event)
    "docker"
    (if (:com.example.application event)
        (str "productiona.docker.", (:com.example.application
          event), ".", (riemann.graphite/graphite-path-
          percentiles event))
        (str "productiona.docker.", (riemann.graphite/graphite-
          path-percentiles event)))
    (str "productiona.hosts.", (riemann.graphite/graphite-path-
      percentiles event))))
```

We've updated our `condp` conditional to include a further `if` conditional. We check if the `com.example.application` field exists and, if it does, then we adjust our Graphite metric name to:

`productiona.docker.tornado.container.metric.name`

Assuming the value of the `com.example.application` field is `tornado`.

If the container is stopped or killed then the data will stop being sent but any existing data will still be retained in Graphite. Let's look at how we could clean up some metrics.

Specifying different resolution for Docker metrics

One of the things we may consider is a shorter retention period for our more numerous Docker metrics. We do that by configuring a Carbon Storage schema

specifically for Docker metrics. Remember in Chapter 4 that we defined a **default** schema in the `/etc/carbon/storage-schemas.conf` file.

Listing 7.38: The existing Carbon schemas

```
[carbon]
pattern = ^carbon\.
retentions = 60:90d

[default]
pattern =.*/
retentions = 1s:24h, 10s:7d, 1m:30d, 10m:1y, 1h:2y
```

We specify an additional schema specifically for Docker. Carbon Storage schemas are executed in a top-down order in the `storage-schema.conf` file. The first schema that matches a metric is what is applied. Let's add that new schema.

Listing 7.39: A new storage schema for Docker events

```
[carbon]
pattern = ^carbon\.
retentions = 60:90d

[docker]
pattern = ^production.\.docker\.
retentions = 1s:24h, 10s:7d, 1m:30d

[default]
pattern = .*
retentions = 1s:24h, 10s:7d, 1m:30d, 10m:1y, 1h:2y
```

We've inserted a new schema called **[docker]** into the file. It has a pattern of **^production.\.docker\.** which will match all incoming Docker metrics to our potential production environments, for example **productiona**, **productionb**, etc. Unlike our **[default]** schema, the retentions are shorter. We keep Docker metrics at a high resolution: one second for 24 hours, 10 seconds for seven days, and finally one minute for 30 days, but that's it. Beyond that we don't keep any metrics. This means our Docker metrics files should be smaller. Let's calculate that now using [J. Javier Maestro's Whisper calculator](#).

Listing 7.40: Calculating our new Docker metric file size

```
$ python ./whisper-calculator.py 1s:24h,10s:7d,1m:30d
1s:24h,10s:7d,1m:30d >> 2281012 bytes
```

We see that each Docker metric Whisper file will take up **2281012** bytes. A saving of **1261452** bytes on our **[default]** schema.

To enable the new schema we'll need to restart the Carbon Cache daemons. We'll also need to delete any existing Docker metric Whisper files. Changes in schema are not automatically propagated to the Whisper files. If you don't want to delete the files you can also use the [Whisper resize utility](#).

Cleaning up old Graphite Docker metrics

The state of the art for managing Graphite metric files is not advanced. Since Graphite stores metrics in files, the fastest and easiest way to manage expired or no-longer-updated metrics is to manage the files. We do that with some **find** commands.

Listing 7.41: A find command to clean up old Graphite metrics

```
$ find /var/lib/graphite/whisper -type f -mtime +10 -name *.wsp  
-delete; find /var/lib/graphite/whisper -depth -type d -empty -  
delete
```

Here we find all Whisper files last updated at least 10 days ago and delete them. The second **find** command deletes any empty directories left behind. If we want to limit the search to just our Docker files, we can update the command to target specific paths, such as all the Docker metrics in the **productiona** environment.

Listing 7.42: A find command to clean up Docker specific Graphite metrics

```
$ find /var/lib/graphite/whisper/productiona/docker -type f -  
mtime +10 -name *.wsp -delete; find /var/lib/graphite/whisper/  
productiona/docker -depth -type d -empty -delete
```

This command narrows the `find` results to a sub-directory.

Using Docker metrics for monitoring

Now that we have container metrics flowing into Riemann we can make use of them for monitoring. Many of the same checks we looked at in Chapters 5 and 6 are equally appropriate for Docker and will carry over without requiring any reworking.

Firstly, we're monitoring the Docker daemon process, `docker`, itself. Earlier in this chapter we enabled the `processes` plugin to monitor the Docker daemon process. As a result of the threshold configuration we set up in Chapter 6, if the process drops below the `1` process threshold we set, a notification will be triggered.

Secondly, the metrics-based checks we built in Chapter 6 will also work. Let's look at how we could use the `check_percentiles` function we created in Chapter 6 on Docker containers to monitor some general container metrics.

Listing 7.43: Using the check_percentiles function for Docker

```
(where (= (:plugin event) "docker")
      (smap #(assoc % :host (:plugin_instance %)
                    :service (cond-> (str (:type %)) (:type_instance %) (str "." (:type_instance %)))
                    )))
  (by :host
    (sdo
      (check_percentiles "cpu.percent" 10
        (smap rewrite-service graph)
        (where (and (service "cpu.percent 0.99") (> metric
          80.0))
          (email "james@example.com"))))
      (check_percentiles "memory.percent" 10
        (smap rewrite-service graph)
        (where (and (service "memory.percent 0.99") (> metric
          80.0))
          (email "james@example.com"))))))
```

Here we're selecting all events from the collectd `docker` plugin using a `where` stream. We've then used `smap` function to rewrite the `:host` and `:service` fields as we did earlier in the chapter.

We split the events via [the `by` function](#), and via the `:host` field, into a separate stream for each container. We then pass the rewritten events to [the `sdo` stream](#). The `sdo` stream takes events and sends them to all child streams specified beneath it—here that's two `check_percentiles` functions. We're looking specifically for the `cpu.percent` and `memory.percent` services and specifying a `10` second window. Remember, `check_percentiles` runs the `percentile` function over the incoming

events and generates the median, 95th, 99th, and max percentiles of the metric over the time window specified.

Inside this check we're first sending our new percentile metrics to Graphite via an `smap`. We're then using a `where` stream matching on one of the new percentile event—for example, on `cpu.percent 0.99` which has a `:metric` value of `80.0` or 80%. Any matching event that exceeds the threshold is then emailed as a notification.

From here we can add more checks, in line with our other host and service-based checks, to extend our monitoring framework to Docker containers.

Other container monitoring tools

Given Docker's youth, the monitoring ecosystem around it is relatively immature. Most of the major SaaS-based monitoring tools, such as [New Relic](#) and [DataDog](#), have added Docker integrations with varying levels of depth.

There are not many standalone open-source tools available yet, but there are some notable initial entrants.

- [cAdvisor](#) — A Google-developed tool that analyzes the resource usage and performance characteristics of running containers. Can run as a container on a Docker server.
- [Heapster](#) — If you're using Kubernetes, Heapster does resource analysis and monitoring of container clusters. It also has a native Riemann output.
- [collectd cgroups](#) — Docker uses cgroups or control groups to help allocate resources like CPU. You can use the collectd cgroups plugin to gather this information.

Summary

In this chapter we've looked at Docker and monitoring containers. We've explored how Docker generates statistics via its binary and API. We also installed a plugin for collectd that enables us to collect those statistics, turn them into metrics, and send them to Riemann.

Inside Riemann we've munged these metrics to make them easier to use and then sent them onto Graphite. We've also seen how we use typical host-based checks with our Docker containers to detect issues and send notifications.

In the next chapter we're going to add the next layer of our monitoring framework, a logging system, to the environment. As part of that chapter we'll also see how to collect logs from Docker.

Chapter 8

Logs and logging

In Chapters 5, 6, and 7 we looked at host and container-based monitoring. In this chapter we're going to add the next layer of our framework: logging. While our hosts, services, and applications generate crucial metrics and events, they also often generate logs that can tell us useful things about their state and status. Additionally, our logs are incredibly useful for diagnostic purposes when investigating an issue or incident. We're going to capture these logs, send them onto a central store, and make use of them to detect issues and provide diagnostic assistance. We're also going to generate metrics and graphs from our logs.

We're going to build a log management platform to complement the other components of our monitoring framework. We'll collect and send some of our logs to Riemann and Graphite, to be recorded as metrics. We'll also see how to integrate Docker containers into our logging.

In considering a log management solution, we need to choose something that:

- Provides lightweight log collection that does not interfere with the actual running of our hosts and applications (refer to the observer effect we talked about in Chapter 2). It should ship data quickly and avoid queuing important information that we need.

- Is a scalable and high-performing platform capable of processing large volumes of logs and storing them for an appropriate retention period to provide diagnostic value.
- Is able to parse and manipulate logs. The varying formats of logs require the ability to normalize aspects like time and date, tags, hosts, and other information.
- Can integrate with everything else we've built, especially to pass events and metrics into Riemann and vice versa.

To address these requirements we're going to introduce you to the Elasticsearch-Logstash-Kibana or ELK stack.

Introducing Elasticsearch, Logstash, and Kibana

So why did we choose ELK? ELK is fast, easy to set up, and is modular and flexible. It allows collection from many sources and can transform and normalize our logs. It also comes with a powerful, searchable storage system and an excellent visualization interface.

The ELK stack is made up of three components:

- Elasticsearch — A document search store.
- Logstash — A log-routing and management engine.
- Kibana — A web-based dashboard and visualization tool.

The first component, [Logstash](#), provides an integrated framework for log collection, centralization, parsing, storage, and search.

Logstash is free and open source ([Apache 2.0](#) licensed), and developed by developer, [Jordan Sissel](#) and the team from [Elastic](#). Logstash is written in JRuby and runs on top of the Java Virtual Machine (JVM). It's easy to set up, high performing, scalable, and easy to extend.

Logstash has a wide variety of input mechanisms: it can take inputs from TCP/UDP, files, Unix Syslog, Microsoft Windows EventLogs, **STDIN**, and a variety of other sources. As a result there's likely little in your environment from which you can't extract logs that can be sent to Logstash.

When those logs hit the Logstash server, there is a large collection of filters that allow for modification, manipulation, and transformation of those events. Information can be extracted from logs to give them context or normalize them across multiple sources.

Finally, when outputting data, Logstash supports a huge range of destinations, including TCP/UDP, email, files, HTTP, Nagios, and a wide variety of network and online services, including Riemann. Most usefully though, Logstash integrates with the search tool, [Elasticsearch](#).

Elasticsearch is a powerful text-indexing and search tool. As the Elastic team puts it: “Elasticsearch is a response to the claim: ‘Search is hard.’” Elasticsearch is easy to set up, has search and index data available RESTfully as JSON over HTTP, and is easy to scale and extend. It’s released under the Apache 2.0 license and is built on top of Apache’s Lucene project.

The best metaphor for Elasticsearch is the index of a book. You flip to the back of the book, look up a word, and then find the reference to a page. That means that rather than searching text strings directly, it creates an index from incoming text and performs searches on the index, not the content. The result? It’s fast.

NOTE This is a simplified explanation. See the [Elasticsearch site](#) for more information and exposition.

The last piece of the stack is [Kibana](#). Kibana is a dashboard and visualization interface that attaches to Elasticsearch. It is primarily used as an interface for

Logstash events, but can query any data stored in Elasticsearch. Kibana can create graphs and dashboards. It's also licensed under the Apache 2.0 license. It comes with its own web server and can be run on any host that can connect to our Elasticsearch back end.

We're going to install, configure, and deploy each of these components. We'll then configure our hosts to send logs through to our ELK stack. We'll also send some events and derived metrics from the ELK stack onto our Riemann installation.

TIP If Logstash and logging interests you, or if this chapter lacks sufficient detail for you, we have another whole book on the topic: [The Logstash Book](#).

Logstash architecture

Logstash is written in JRuby and runs in a Java Virtual Machine (JVM). Its architecture is message based and simple. In our Logstash logging framework there are four elements:

- Shipping — Sends logs from our host to a Logstash server.
- Indexing — Receives logging events and indexes them.
- Storage — Elasticsearch servers store and make logs searchable.
- Visualization — Kibana allows us to build graphs and dashboards.

We're going to install each component starting with the Indexing and Storage components, and then we'll connect our hosts to Logstash. In each environment we're going to run a Logstash server and an Elasticsearch cluster. We're going to install Logstash on a host aptly named `logstash.example.com` and create a three-node Elasticsearch cluster to store our logs: `esa1.example.com`, `esa2.example.com`

, and esa3.example.com. This would be our Production A Logstash environment. We'd create duplicates in Production B, etc.

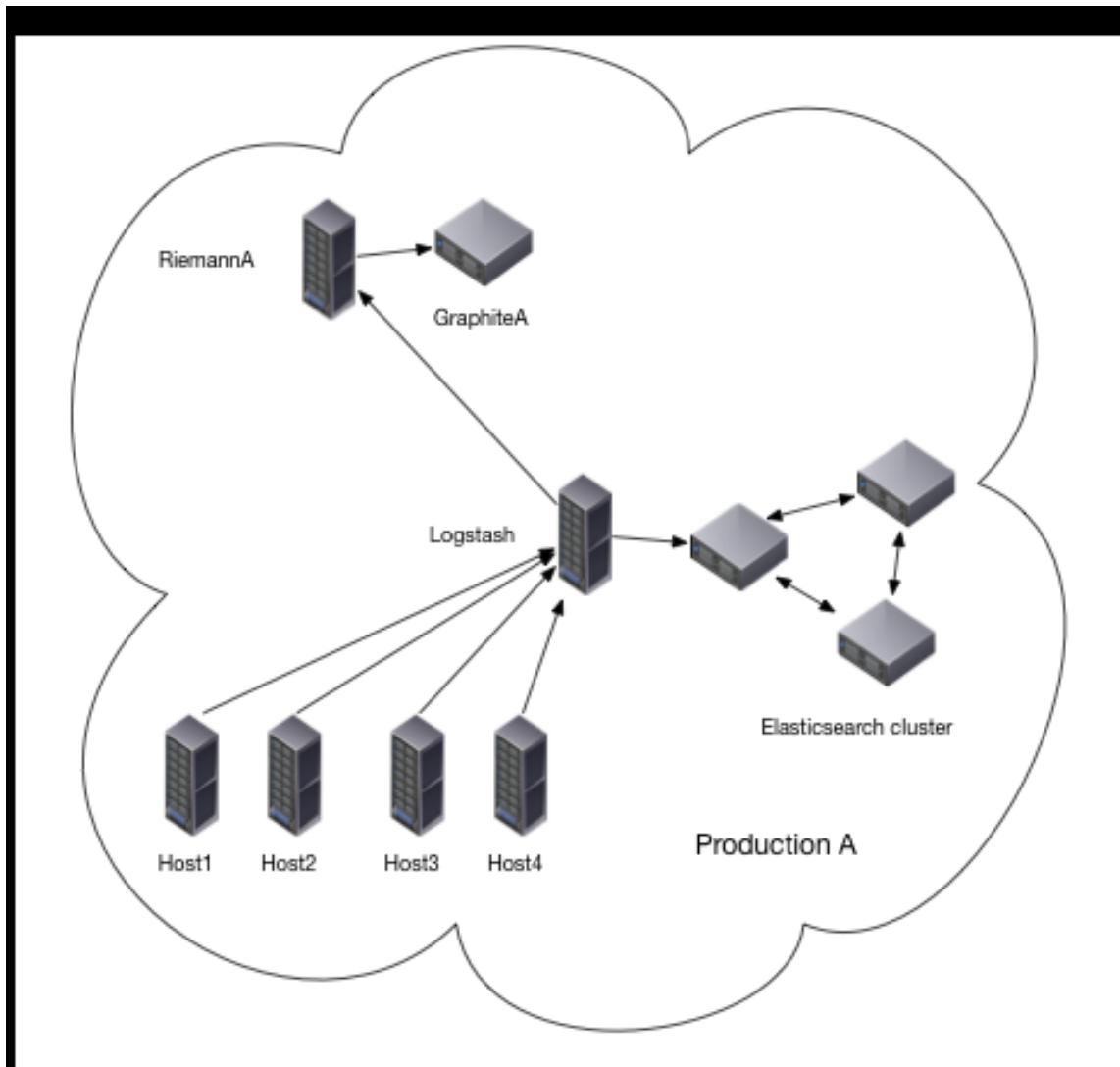


Figure 8.1: Logstash Architecture

NOTE We'll also install collectd on every host using the instructions from Chapter 5. This will allow us to monitor our monitoring.

Installing Logstash

We'll start by installing Logstash on our `logstasha.example.com` host, but to do so, we need to install prerequisites for Logstash. Logstash's only prerequisite is Java. Let's install it.

On Debian & Ubuntu

We install Java via the `apt-get` command:

Listing 8.1: Installing Java on Debian and Ubuntu

```
$ sudo apt-get -y install default-jre
```

On Red Hat

We install Java via the `yum` command:

Listing 8.2: Installing Java on Red Hat

```
$ sudo yum install java-1.7.0-openjdk
```

TIP On newer Red Hat and family versions the `yum` command has been replaced with the `dnf` command. The syntax is otherwise unchanged.

Testing Java is installed

We then test that Java is installed via the `java` binary:

Listing 8.3: Testing Java is installed

```
$ java -version
java version "1.7.0_09"
OpenJDK Runtime Environment (IcedTea7 2.3.3)(7u9-2.3.3-0ubuntu1
~12.04.1)
OpenJDK Client VM (build 23.2-b09, mixed mode, sharing)
```

Installing the Logstash package

Once we have Java installed we install the Logstash package.

On Ubuntu and Debian

For Ubuntu and Debian systems we first need to add the Elastic.co public key, like so:

Listing 8.4: Getting the Logstash public key on Ubuntu

```
$ wget -qO - https://packages.elastic.co/GPG-KEY-elasticsearch |
  sudo apt-key add -
```

We add the following Apt configuration so we can find the Logstash repository.

Listing 8.5: Adding the Logstash packages

```
$ echo "deb http://packages.elastic.co/logstash/2.2/debian stable  
main" | sudo tee -a /etc/apt/sources.list.d/logstash.list
```

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

Listing 8.6: Updating Apt and installing the Logstash package

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

On Red Hat

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

Listing 8.7: Getting the Logstash public key on Red Hat

```
$ sudo rpm --import https://packages.elastic.co/GPG-KEY-  
elasticsearch
```

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

Listing 8.8: The Logstash Yum configuration

```
[logstash-2.2]
name=Logstash repository for 2.2.x packages
baseurl=http://packages.elastic.co/logstash/2.2/centos
gpgcheck=1
gpgkey=http://packages.elastic.co/GPG-KEY-elasticsearch
enabled=1
```

Now we install Logstash using the `yum` command.

Listing 8.9: Installing Logstash on Red Hat

```
$ sudo yum install logstash
```

Installing Logstash via configuration management

We could also install Logstash via a variety of configuration management tools like Puppet or Chef or via Docker or Vagrant.

You can find a Chef cookbook for Logstash:

- <https://github.com/lusis/chef-logstash>

You can find a Puppet module for Logstash:

- <https://github.com/elastic/puppet-logstash>

You can find an Ansible role for Logstash:

- <https://github.com/valentinogagliardi/logstash-role>

You can find Docker images for Logstash:

- https://hub.docker.com/_/logstash/

You can find a Vagrant configuration for Logstash:

- <https://github.com/paulczar/vagrant-logstash>

Testing Logstash is installed

Logstash is installed into the `/opt/logstash` directory. Inside that directory is the `/bin/logstash` binary, which runs Logstash. The package also installs basic service management including a `logstash` service. Let's test Logstash is installed and working using the binary.

Listing 8.10: Testing Logstash is installed

```
$ /opt/logstash/bin/logstash --version
logstash 2.2.2
```

Configuring Logstash

Now let's add some basic configuration to Logstash to get started. We're going to create a `logstash.conf` file in the `/etc/logstash/conf.d/` directory to hold that configuration.

Listing 8.11: Creating the logstash.conf file

```
$ sudo vi /etc/logstash/conf.d/logstash.conf
```

Let's populate that file now.

Listing 8.12: The logstash.conf configuration file

```
input {  
    stdin {}  
}  
  
output {  
    stdout {  
        codec => rubydebug  
    }  
}
```

Our `logstash.conf` file contains two configuration blocks, one called `input` and one called `output`. These are two of three types of plugin components in Logstash that we can configure. The third type is `filter`. Each type configures a different portion of the Logstash server:

- inputs — How events get into Logstash.
- filters — How events in Logstash can be manipulated.
- outputs — How events from Logstash can be output.

In the Logstash world, events enter via inputs; they are manipulated, mutated, or changed in filters; and they exit Logstash via outputs.

Inside each component's block we can specify and configure plugins. For example, in the `input` block above we've defined the `stdin` plugin which enables event input from `STDIN`. In the `output` block we've configured its opposite: the `stdout` plugin, which outputs events to `STDOUT`. For this plugin we've added a configuration option, `codec`, with a value of `rubydebug`. This outputs each event as a JSON hash, in the style of Ruby's own debug output (hence the name).

Let's test this now by running Logstash interactively.

Listing 8.13: Testing Logstash interactively

```
$ sudo /opt/logstash/bin/logstash -v -f /etc/logstash/conf.d/
  logstash.conf
Pipeline started {:level=>:info}
testing
{
  "message" => "testing",
  "@version" => "1",
  "@timestamp" => "2015-04-22T02:15:15.107Z",
  "host" => "logstasha"
}
```

We've run the `logstash` binary and passed it two flags. The `-v` flag runs Logstash in its verbose mode. The `-f` flag tells Logstash where to load the configuration file. Our configuration file configures Logstash to receive input from `STDIN` and then outputs it to `STDOUT`. We've typed `testing` onto the command line and Logstash has processed it into a Logstash event.

Another useful flag is `--configtest`. You can pass this flag to test the validity of a Logstash configuration.

Listing 8.14: Testing Logstash configuration validity

```
$ sudo /opt/logstash/bin/logstash --configtest -f /etc/logstash/  
conf.d/logstash.conf
```

Our event has been printed as a hash. Indeed it's represented internally in Logstash as a JSON hash.

Listing 8.15: Our first Logstash event

```
{  
    "message" => "testing",  
    "@version" => "1",  
    "@timestamp" => "2015-04-22T02:15:15.107Z",  
    "host" => "logstash"  
}
```

The event has a set of fields and a format. Logstash calls these formats **codecs**. There are a variety of codecs that Logstash supports.

- plain — Events are recorded as plain text and any parsing is done using **filter** plugins.
- json — Events are assumed to be JSON. Logstash tries to parse an event's contents into fields itself with that assumption.

We're going to focus on the **json** format as it's the easiest way to work with Logstash events and show how they can be used. The format is made up of a number of elements. A basic event has the following elements:

- `@timestamp` — An ISO8601 timestamp.
- `message` — The event’s message. Here, that’s `testing`, as that’s what we put into `STDIN`.
- `@version` — The version of the event format. This current version is `1`.

Additionally many of the plugins we’ll use add additional fields. For example, the `stdin` plugin we’ve just used adds a field called `host` which specifies the host that generated the event. Other plugins—for example, the `file` input plugin which collects events from files—add fields like `path`, which reports the path of the file from which we’re collecting. We can also add elements like custom fields, tags, and other context to events.

TIP Running interactively we can stop Logstash using the Ctrl-C key combination.

Now that we’ve generated an event with Logstash we need to be able to send and store that event—and future events. To do that we need to install and configure Elasticsearch.

Installing Elasticsearch

We’re going to install [Elasticsearch](#) to provide our search capabilities. We’re going to do this in three hosts in each environment. This allows us to build a cluster of nodes to provide some resiliency for our log storage. We’ll learn a bit more about how Elasticsearch’s clustering works later in this chapter.

Like Logstash, Elasticsearch’s only prerequisite is Java. Let’s install that and then jump straight into installing Elasticsearch itself. We need to repeat these instructions, or preferably use a configuration management tool, on all of our Elastic-

search hosts.

TIP For [security reasons](#), you should ensure you install an Elasticsearch version later than 1.4.4.

On Debian and Ubuntu

We install Java via the `apt-get` command.

Listing 8.16: Installing Java on Debian and Ubuntu

```
$ sudo apt-get -y install default-jre
```

We then test that Java is installed via the `java` binary:

Listing 8.17: Testing Java is installed

```
$ java -version
java version "1.7.0_09"
OpenJDK Runtime Environment (IcedTea7 2.3.3)(7u9-2.3.3-0ubuntu1
~12.04.1)
OpenJDK Client VM (build 23.2-b09, mixed mode, sharing)
```

Next we install Elasticsearch itself. The Elastic.co team provides tar balls, RPMs, and DEB packages. You can find the [Elasticsearch download page here](#).

We're going to install Elasticsearch via packages using Elastic.co's package repos-

itory.

For Ubuntu and Debian, we first need to install the Elastic.co public key.

Listing 8.18: Installing the public key for Elasticsearch

```
$ wget -qO - https://packages.elastic.co/GPG-KEY-elasticsearch |  
  sudo apt-key add -
```

Then add the following repository to our Apt configuration.

Listing 8.19: Adding the Elasticsearch Apt repository

```
$ sudo sh -c "echo deb http://packages.elastic.co/elasticsearch  
/2.x/debian stable main > /etc/apt/sources.list.d/elasticsearch  
.list"
```

Next we update our Apt repositories and install the `elasticsearch` package.

Listing 8.20: Installing Elasticsearch on Ubuntu

```
$ sudo apt-get update  
$ sudo apt-get -y install elasticsearch
```

On Red Hat

On Red Hat, we install Java via the `yum` command.

Listing 8.21: Installing Java on Red Hat for Elasticsearch

```
$ sudo yum install java-1.7.0-openjdk
```

Then we need to add the Elastic.co public key.

Listing 8.22: Adding the Elasticsearch public key on Red Hat

```
$ sudo rpm --import https://packages.elastic.co/GPG-KEY-elasticsearch
```

Now add the Elasticsearch package repository to our Yum repositories. Create a file called **elasticsearch.repo** in the **/etc/yum.repos.d/** directory.

Listing 8.23: Adding the Yum repo for Elasticsearch

```
$ sudo vi /etc/yum.repos.d/elasticsearch.repo
```

Now populate that file.

Listing 8.24: The Yum repo for Elasticsearch

```
[elasticsearch-2.x]
name=Elasticsearch repository for 2.x packages
baseurl=http://packages.elastic.co/elasticsearch/2.x/centos
gpgcheck=1
gpgkey=http://packages.elastic.co/GPG-KEY-elasticsearch
enabled=1
```

Now we install the `elasticsearch` package via the `yum` command.

Listing 8.25: Installing Elasticsearch on Red Hat

```
$ sudo yum -y install elasticsearch
```

Installing Elasticsearch via configuration management

We could also install Elasticsearch via a variety of configuration management tools like Puppet or Chef or via Docker or Vagrant.

You can find a Chef cookbook for Elasticsearch:

- <https://github.com/elastic/cookbook-elasticsearch>.

You can find a Puppet module for Elasticsearch:

- <https://forge.puppetlabs.com/elasticsearch/elasticsearch>.

You can find an Ansible role for Elasticsearch:

- <https://galaxy.ansible.com/list#/roles/1450>.

You can find Docker images for Elasticsearch:

- https://hub.docker.com/_/elasticsearch/.

You can find a Vagrant configuration for Elasticsearch:

- <https://github.com/comperiosearch/vagrant-elk-box>.

Testing Elasticsearch is installed

We test that Elasticsearch is installed by running the `elasticsearch` binary. The package installs the binary to `/usr/share/elasticsearch` in the `bin` directory. Let's run that binary now.

Listing 8.26: Testing the elasticsearch binary

```
$ /usr/share/elasticsearch/bin/elasticsearch --version
Version: 2.2.0, Build: 8ff36d1/2016-01-27T13:32:39Z, JVM: 1.7.0
_79
```

The `--version` flag returns the Elasticsearch version.

Determining Elasticsearch is running

You can tell if Elasticsearch is running by browsing to port `9200` on your host, for example:

Listing 8.27: Checking Elasticsearch is running

```
http://es1.example.com:9200
```

NOTE You may need to start Elasticsearch first, for example `sudo service elasticsearch start`.

This should return some status information that looks like:

Listing 8.28: Elasticsearch status information

```
{
  "name" : "es1",
  "version" : {
    "number" : "2.2.0",
    "build_hash" : "8ff36d139e16f8720f2947ef62c8167a888992fe",
    "build_timestamp" : "2016-01-27T13:32:39Z",
    "build_snapshot" : false,
    "lucene_version" : "5.4.1"
  },
  "tagline" : "You Know, for Search"
}
```

You can also browse to a more detailed status page:

Listing 8.29: Elasticsearch stats page

```
http://es1.example.com:9200/_stats?pretty=true
```

This will return a page that contains a variety of information about the statistics and status of your Elasticsearch server.

TIP You can find more extensive documentation for Elasticsearch [on the Elastic documentation site](#).

Configuring our Elasticsearch cluster and nodes

Our next step is to turn our individual Elasticsearch nodes into a cluster. Each Elasticsearch node is started with a default cluster name, `elasticsearch`, and a random, allegedly amusing node name—for example, “Franz Kafka” or “Spider-Ham.” A new random node name is selected each time Elasticsearch is restarted. New Elasticsearch nodes join any cluster with the same cluster name they have defined. To do this, Elasticsearch makes use of either unicast or multicast discovery to find other nodes.

To enable our own cluster we need to update the defined cluster name in Elasticsearch’s configuration. We’re going to customize our cluster and node names to ensure we have unique names and make sure our nodes join the right cluster.

To do this we need to edit the `/etc/elasticsearch/elasticsearch.yml` file. This is Elasticsearch’s [YAML-based](#) configuration file. We look for the following entries inside the file:

Listing 8.30: Initial cluster and node names

```
# cluster.name: elasticsearch
# node.name: "Franz Kafka"
```

We're going to uncomment and change both the cluster and node name, and configure networking and clustering. We're going to choose a cluster name of **productiona** for the environment our cluster is running in. In Production B we'd choose **productionb**, in Mission Control we'd choose **missioncontrol**, and so on. This is just a way of labelling the cluster and identifying what data is in it. We then choose a node name matching our node's host name.

Listing 8.31: New cluster and node names

```
cluster.name: productiona
node.name: esa1
network.host: [ _local_, _non_loopback:ipv4_ ]
discovery.zen.ping.unicast.hosts: ["esa1.example.com", "esa2.
example.com", "esa3.example.com"]
```

We've also specified the **network.host** option. This controls where Elasticsearch will be bound. In this case we're binding to the local host and the first non-loopback IPv4 interface.

We're using unicast discovery to connect our Elasticsearch cluster members. We've listed each cluster member by host name (DNS will be needed to resolve them) in an array. For the details of this, look at the Discovery section of the **/etc/elasticsearch/elasticsearch.yml** configuration file. The file is well commented and self-explanatory.

TIP You can also read about Elasticsearch discovery in [the Zen discovery guide on the Elasticsearch site](#).

Elasticsearch clusters have four types of nodes:

- Master-eligible — Nodes that are able to become master nodes and control a cluster.
- Data node — Data nodes hold data and perform data-related operations such as CRUD, search, and aggregations.
- Client node — Does not hold data and can not become the master node. It behaves as a “smart router” and is used to forward cluster-level requests to the master node, and data-related requests to the appropriate data nodes.
- Tribe node — A tribe node is a special type of client node that can connect to multiple clusters and perform search and other operations across all connected clusters.

We’re only going to look at master and data nodes. By default, a freshly installed node is potentially both a master-eligible node and a data node. In our initial configuration we’re going to leave our nodes in their mixed master-eligible and data mode. This means a master will be automatically elected when the cluster is started.

But indexing and searching your data is performance-intensive work. As you expand your cluster this can cause issues on your master node that could impact your cluster’s functionality. To ensure that the master node is stable in a bigger cluster, consider splitting the roles between dedicated master-eligible nodes and dedicated data nodes.

These decisions are controlled by the `node.master` and `node.data` configuration options in the `/etc/elasticsearch/elasticsearch.conf` file. So, if we wished to

configure one of our cluster members as the master and have it not store data we could do this:

Listing 8.32: Configuring our Elasticsearch cluster

```
cluster.name: productiona
node.name: esa1
node.master: true
node.data: false
```

We would reverse this configuration to specify a data-alone node.

NOTE You can read more about Elasticsearch nodes [here](#).

We need to restart Elasticsearch to reconfigure it.

Listing 8.33: Restarting Elasticsearch to enable clustering

```
$ sudo /etc/init.d/elasticsearch restart
```

We would now configure the remaining nodes: `esa2` and `esa3`.

Adding a cluster management plugin

Elasticsearch has a plugin system. Plugins provide additional capabilities for Elasticsearch servers including additional management capabilities. One of the most useful management plugins is [Elasticsearch-head](#) written by Ben Birch. It's a web-

based cluster management front-end. Let's add that plugin now.

TIP You can read more about Elasticsearch plugins [here](#).

To add plugins we use the `plugin` binary from the `/usr/share/elasticsearch/bin` directory. Let's do that now on the `es1.example.com` host.

Listing 8.34: Installing an Elasticsearch plugin

```
es1$ sudo /usr/share/elasticsearch/bin/plugin install mobz/
      elasticsearch-head
```

Here we've used the `install` flag and pointed the `plugin` at `mobz/elasticsearch-head` which is a combination of a plugin author namespace, `mobz`, and a plugin name, `elasticsearch-head`. This will install and enable the plugin.

Plugins are generally available in the Elasticsearch server via URLs with a specific URL path, `_plugins`, reserved for them. To access the Elasticsearch-head plugin we browse to:

`http://es1.example.com:9200/_plugin/head/`

We should see an interface much like:



Figure 8.2: The Elasticsearch Head plugin

NOTE Your browser may not look exactly like this screen as cluster names, sizes, and status will vary.

The plugin shows the list of current cluster nodes and their status. Particularly useful for our current situation is the **Info** dropdown in the top right of the screen. Selecting the **Cluster Health** dropdown will show the health of the cluster.

Time and time zone

Like our Riemann and Graphite hosts, we need to ensure the time and time zone is right on our Logstash and Elasticsearch hosts. This ensures events have consistent timestamps across all our hosts. To do this, follow the steps we documented in Chapter 4 when we installed Graphite.

We recommend setting Logstash and Elasticsearch hosts to UTC time so they'll match the Riemann and Graphite hosts.

Integrating Logstash and Elasticsearch

Now that we've got both Logstash and Elasticsearch installed and running let's hook them together. To do this we're going to send some sample events through Logstash into our Elasticsearch cluster.

To provide these sample events we're going to read our local Syslog logs and send them through to Elasticsearch. As our `logstasha.example.com` host is running Ubuntu, we first need to add the `logstash` user to the `adm` group. This gives our local Logstash server the right access to read our Syslog files.

Listing 8.35: Granting access to logs on Ubuntu

```
$ sudo useradd -G adm logstash
```

For any file added, Logstash will need to have permission to read it. We'll need to grant the user and group with which we're running Logstash those permissions. On CentOS, we would adjust the permissions with the `setacl` command.

Listing 8.36: Granting access to logs on CentOS

```
$ sudo setacl -m g:logstash:r /var/log/messages /var/log/secure
```

This will grant permissions to the `/var/log/messages` and `/var/log/secure` files.

Now we need to tell Logstash to use an `input` plugin called `file` to read events from files, and an `output` plugin called `elasticsearch` to send events through to our Elasticsearch cluster. Let's update our `/etc/logstash/conf.d/logstash.conf` file with this new configuration.

Listing 8.37: The updated logstash.conf configuration file

```
input {
    stdin { }
    file {
        path => [ "/var/log/syslog", "/var/log/auth.log" ]
        type => "syslog"
    }
}

output {
    stdout { }
    elasticsearch {
        sniffing => true
        hosts => "es01.example.com"
    }
}
```

Note that we've added a new input plugin, `file`, and a new output plugin called `elasticsearch`. The `file` input plugin reads log entries from files. The `file` plugin does some pretty useful things:

- It automatically detects new files matching our collection criteria.
- It can handle file rotation, such as if were to run `logrotate` to rotate log files.
- It keeps track of where it is up to in a file. Specifically this will load any new events from the point at which Logstash last processed an event. Any new files start from the bottom of the file.

TIP See the [sinceDb option of file plugin](#) for more information.

We've specified an array of files from which we'll collect events in the `path` option. In our case, we've selected two files containing Syslog output: `/var/log/auth.log` and `/var/log/syslog`.

NOTE These file choices assume an Ubuntu or Debian host. On Red Hat we'd change these to `/var/log/messages` and `/var/log/secure`.

The `path` option also allows us to specify globbing. For example, we could collect events from all `*.log` files in the `/var/log/` directory:

Listing 8.38: File input globbing

```
path => [ "/var/log/*.log" ]
```

Or even a recursive glob like:

Listing 8.39: File recursive globbing

```
path => [ "/var/log/**/*log" ]
```

TIP You can find more options of the `file` plugin [here](#).

The last attribute, `type`, is a global attribute we can set for plugins to identify the type of events being generated. In this case we've set it to `syslog`. The `type` attribute can be used to identify events that you might want to filter, process, notify on, or otherwise manage in your Logstash configuration. In this case we're going to mark these events as being of type `syslog` so we can use some filter plugins to process them into more usable events. We'll see how to do that shortly.

The `elasticsearch` output plugin sends events from Logstash onto an Elasticsearch server or cluster. The plugin has two configuration options: `sniffing` and `hosts`. The `hosts` option takes the name of an Elasticsearch host—in our case `es1`. The `sniffing` option then connects to our Elasticsearch node and asks about the other nodes in the cluster. It automatically adds any nodes it finds to the `hosts` option.

To enable this we need to restart Logstash.

Listing 8.40: Restarting Logstash to enable file monitoring

```
$ sudo service logstash restart
```

Let's now send some events to our new cluster. Most Unix and Unix-like platforms come with a handy utility called `logger`. It generates Syslog messages that allow us to easily test if our Syslog configuration is working. It can be used like so:

Listing 8.41: Testing with logger

```
$ logger "This is a syslog message"
```

TIP You can see full options to change the facility and priority of logger messages

here.

This will generate messages that will end up in our `/var/log/syslog` file, be picked up by the `file` plugin, be processed into Logstash, and then will be sent onto Elasticsearch.

What happens inside Logstash?

When an event is collected by Logstash it is turned into a native JSON-based event like the one we saw earlier in this chapter. Let's look at our Syslog event in this form.

Listing 8.42: A syslog log event

```
{  
    "message" => "Apr 29 17:29:10 logstash root: This is a  
                  syslog message",  
    "@version" => "1",  
    "@timestamp" => "2015-04-29T21:29:10.830Z",  
    "host" => "logstasha",  
    "type" => "syslog",  
    "path" => "/var/log/syslog"  
}
```

Our plugin has grabbed our message from the file Syslog wrote it to and used it as the value of our `message` field. We also see the version of the event schema, a timestamp (this is when the event was received by Logstash), the host that generated it, and the type we assigned earlier, `syslog`. The plugin has also added a new field, `path`, which tells Logstash the file from which the event was plucked.

This is a useful event but we can make it even more useful by parsing it and extracting some more data using some of Logstash's filter plugins. Let's look at an updated configuration that does this.

Listing 8.43: Adding our first Logstash filters

```
input {
    stdin { }
    file {
        path => [ "/var/log/syslog", "/var/log/auth.log" ]
        type => "syslog"
    }
}
filter {
    if [type] == "syslog" {
        grok {
            match => { "message" => "%{SYLOGTIMESTAMP:syslog_timestamp
} %{SYLOGHOST:syslog_hostname} %{DATA:syslog_program
}(?:\[%{POSINT:syslog_pid}\])?: %{GREEDYDATA:
syslog_message}" }
            remove_field => ["message"]
        }
        syslog_pri { }
        date {
            match => [ "syslog_timestamp", "MMM d HH:mm:ss", "MMM dd
HH:mm:ss" ]
        }
    }
}
output {
    stdout { }
    elasticsearch {
        sniffing => true
        hosts => "es01.example.com"
}
```

You can see we've added a new block called `filter`. This section contains filter plugins that allow Logstash to parse, manipulate, or change events. In this case we're parsing Syslog logs.

First, we're taking advantage of the `syslog` type we set earlier. This, combined with a conditional `if` statement, allows us to only select those events with a `type` field (to reference it, the field name is wrapped in [] block brackets) value of `syslog`.

Listing 8.44: Logstash conditional selection

```
filter {
  if [type] == "syslog" {
    . . .
  }
}
```

TIP You can read more about [conditionals in the Logstash documentation](#).

Inside our conditional we specify some plugins that will inspect and parse our Syslog logs.

Listing 8.45: Our Syslog parsing filters

```
grok {  
  match => { "message" => "%{SYSLOGTIMESTAMP:syslog_timestamp}  
    %{SYSLOGHOST:syslog_hostname} %{DATA:syslog_program}(?:\[ %{  
      POSINT:syslog_pid}\])?: %{GREEDYDATA:syslog_message}" }  
  remove_field => [ "message" ]  
}  
syslog_pri { }  
date {  
  match => [ "syslog_timestamp", "MMM d HH:mm:ss", "MMM dd HH:  
    mm:ss" ]  
}
```

Here we've specified three plugins: `grok`, `syslog_pri`, and `date`.

The `grok` filter

The `grok` filter is one of Logstash's most commonly used plugins. It provides the ability to match field contents with regular expressions and extract useful information, usually into new fields.

Listing 8.46: The grok filter

```
grok {  
    match => { "message" => "%{SYSLOGTIMESTAMP:syslog_timestamp}  
        %{SYSLOGHOST:syslog_hostname} %{DATA:syslog_program}(?:\[%"  
            POSINT:syslog_pid\]\])?: %{GREEDYDATA:syslog_message}" }  
    remove_field => [ "message" ]  
}
```

Inside the `grok` filter we specify a `match` attribute. This combines the name of the field we wish to parse—here we’re parsing the `message` field—and the regular expression we’re using to parse that field. Logstash tries to make regular expression matching really easy by providing a set of pre-baked expressions that match common log forms. This includes a set of expressions that match various components of Syslog log entries. Logstash calls these patterns.

In our regular expression these patterns are indicated by the capitalized declarations wrapped in `%{ }` like `%{SYSLOGTIMESTAMP:syslog_timestamp}`. When Logstash sees a pattern, it knows to look it up and expand it into a regular expression. Here `SYSLOGTIMESTAMP` translates into the expression: `%{MONTH} +%{MONTHDAY} %{TIME}`. This two is a series of patterns, each of which Logstash would expand into a lower-level regular expression. For example, the `%{MONTH}` pattern expands into:

Listing 8.47: The MONTH pattern

```
MONTH \b(?:Jan(?:uary)?|Feb(?:ruary)?|Mar(?:ch)?|Apr(?:il)?|May|  
Jun(?:e)?|Jul(?:y)?|Aug(?:ust)?|Sep(?:tember)?|Oct(?:ober)?|Nov  
(?:ember)?|Dec(?:ember)?)\b
```

You can find these default patterns and others, which ship with Logstash, on GitHub [here](#).

We take these patterns, as we have in our `match` attribute, and combine them to parse a specific field. Note that our patterns have two pieces: `%{SYSLOGTIMESTAMP:syslog_timestamp}`. After our pattern we've specified a colon and `syslog_timestamp`. This syntax allows us to capture any data we've matched from our pattern and store it in a new field. Here we've captured any timestamp matched by the `SYSLOGTIMESTAMP` pattern and added a new field to our event called `syslog_timestamp`. We've also done this for the hostname, the program, the process ID, and the Syslog message itself.

This turns our original event...

Listing 8.48: Our original syslog log event

```
{  
    "message" => "Apr 29 17:29:10 logstash root: This is a  
                  syslog message",  
    "@version" => "1",  
    "@timestamp" => "2015-04-29T21:29:10.830Z",  
    "host" => "logstasha",  
    "type" => "syslog",  
    "path" => "/var/log/syslog"  
}
```

...Into a much more useful parsed event:

Listing 8.49: Our grok-parsed event

```
{  
    "@version" => "1",  
    "@timestamp" => "2015-04-29T17:29:10.000Z",  
    "type" => "syslog",  
    "host" => "logstasha",  
    "path" => "/var/log/syslog",  
    "syslog_timestamp" => "Apr 29 17:29:10",  
    "syslog_hostname" => "logstasha",  
    "syslog_program" => "root",  
    "syslog_message" => "This is a syslog message"  
}
```

We see new custom fields—`syslog_timestamp`, `syslog_hostname`, `syslog_program`, and `syslog_message`—have all been created. This makes our event eminently more searchable and useful because we can now identify particular programs, hosts, and messages much more easily.

Our `message` field is also gone. This is done by the `remove_field` attribute in our `grok` filter. We've done this because once the `message` field is parsed then we don't need it anymore and storing isn't valuable. It's a useful cleanup—we recommend you consider it.

If the idea of creating these matches is a bit daunting (who likes regular expressions!) then here are some useful tools that can help you construct `grok` matches:

- The Grok Constructor <http://grokconstructor.appspot.com/>.
- The Grok Debugger <https://grokdebug.herokuapp.com/>.

The Syslog priority filter

The next filter, `syslog_pri`, processes the [Syslog priority/severity and facility information](#) from our Syslog event. Syslog facilities and priorities tell us the source of our event. For example, the `auth` facility is generally reserved for authentication events and the severity, such as `info` for informational and `crit` for critical.

The plugin doesn't take any options and further manipulates our event to become:

Listing 8.50: The syslog_pri parsed event

```
{  
    "message" => "Apr 29 17:29:10 logstash root: This is a syslog  
    message",  
    "@version" => "1",  
    "@timestamp" => "2015-04-29T17:29:10.000Z",  
    "type" => "syslog",  
    "host" => "logstash",  
    "path" => "/var/log/syslog",  
    "syslog_timestamp" => "Apr 29 17:29:10",  
    "syslog_hostname" => "logstash",  
    "syslog_program" => "root",  
    "syslog_message" => "This is a syslog message",  
    "syslog_severity_code" => 5,  
    "syslog_facility_code" => 1,  
    "syslog_facility" => "user-level",  
    "syslog_severity" => "notice"  
}
```

We see that four new custom fields have been added, two showing the severity and facility code, and two more fields showing the matching textual description.

The date filter

The last plugin, **date**, helps us process dates inside events. By default, Logstash uses the time an event is received as the timestamp of that event. Depending on circumstances, this time could vary from the time the event was actually generated. The **date** plugin helps address this variance. It's used to select a timestamp from a specified field, parse the date format, and then use that timestamp as the

time of the event.

Listing 8.51: The date filter

```
date {  
    match => [ "syslog_timestamp", "MMM d HH:mm:ss", "MMM dd HH:mm  
              :ss" ]  
}
```

To use the plugin, we select a field to parse and use as our event's timestamp—in this case, `syslog_timestamp`. We specify this as the value of the `match` attribute. We also provide one or more potential date formats to match against our timestamp field, here `MMM d HH:mm:ss` and `MMM dd HH:mm:ss`. Between these two formats they should match most Syslog date formats.

When the event hits the `date` plugin, the `syslog_timestamp` field will be parsed for a usable timestamp, and if successful, the `@timestamp` of the event will be replaced with this timestamp. This ensures Logstash has the correct date the event was generated rather than the date Logstash received the event.

NOTE This is a basic demonstration of the power of Logstash's filtering. There are a lot more filtering plugins available. We'll see some more of them in subsequent chapters but there is also a full list [on the Elastic site](#).

What happens inside Elasticsearch?

After the event is filtered it reaches the `output` block and is processed by the `elasticsearch` plugin. This plugin takes the event, indexes it, converts it into a

form Elasticsearch can process, and then sends it into the cluster. After the event exits Logstash it is received by Elasticsearch and put into an index. Under the covers, Elasticsearch uses [Apache Lucene](#) to create this index. Each index is a logical namespace; in Logstash's case the default indexes are named for the day the events are received. For example:

Listing 8.52: A Logstash index

```
logstash-2015.12.31
```

As we've seen, each Logstash event is made up of fields, and these fields become documents inside that index. To compare Elasticsearch to a relational database: an index is a table, a document is a table row, and a field is a table column. Like a relational database you can define a schema too. Elasticsearch calls these schemas "mappings."

TIP You can read more about mappings in [the Elasticsearch documentation](#).

Like a schema, mapping declares what data and data type fields documents contain, any constraints that are present, what the unique and primary keys are, and how to index and search each field. Unlike a schema, though, we can also specify Elasticsearch settings.

To see the mapping that's currently applied on your Elasticsearch server, use the `curl` command.

Listing 8.53: Showing the current Elasticsearch mapping

```
$ curl http://es1.example.com:9200/_template/logstash?pretty
```

You can also see mappings applied to specific indexes like so:

Listing 8.54: Showing index-specific mappings

```
$ curl http://es1.example.com:9200/logstash-2015.12.31/_mapping?  
pretty
```

NOTE This default mapping has been provided by Logstash so it's broadly optimized for managing Logstash events.

Indexes are stored in Lucene instances called “shards.” There are two types of shards: primary and replica. Primary shards are where your documents are stored. Each new index automatically creates five primary shards. This is a default setting, and the number of primary shards can be adjusted when the index is created but not after it is created. Once the index has been created the number of primary shards cannot be changed.

TIP A useful tool for managing indexes is [Curator](#). Curator helps you set good retention policies on your indexes and clean up old indexes when required.

Replica shards are copies of the primary shards that exist for two purposes:

- To protect your data
- To make your searches faster

Each primary shard will have one replica by default but more if required. Unlike primary shards, this can be changed dynamically to scale out or make an index more resilient. Elasticsearch will distribute these shards across the available nodes and ensure primary and replica shards for an index are not present on the same node.

Shards are stored on our Elasticsearch nodes. Each node is automatically part of an Elasticsearch cluster, even if it's a cluster of one. Elasticsearch automatically distributes shards amongst all nodes in the cluster. It can move shards automatically from one node to another in the case of node failure or when new nodes are added. One of the great things about Elasticsearch's clustering is that it is relatively simple and pain free. Nodes, as long as they can communicate, generally join their specified cluster cleanly and automatically. Data is mostly managed automatically on the cluster and protected from node failures, assuming sufficient nodes are present. As a result, we're not going to do any tweaking of our clustering or cluster performance.

You can view your shards and replicas as data flows into Elasticsearch using the Elasticsearch-head plugin we installed earlier.

TIP You can read more about Elasticsearch clustering [in the documentation](#).

Installing Kibana

Now that we have data flowing into Elasticsearch we need to be able to review and search that data and potentially create visualizations from it. This is where the last piece of the ELK stack comes in: [Kibana](#). Kibana is a dashboard for Elasticsearch-based data written in Javascript. Like the rest of the ELK stack, it is open source and licensed under the Apache 2.0 license.

Kibana runs its own web server and can be installed on any host that can connect to our Elasticsearch cluster. Let's download and install it on our [logstash.example.com](#) host.

Kibana is available as a download from [the Elastic website](#) and as packages for Ubuntu and Red Hat distributions.

We're now ready to install Kibana. To do so we need to add the Kibana APT repository to our host.

Listing 8.55: Adding the Kibana APT repository

```
$ sudo sh -c "echo 'deb http://packages.elastic.co/kibana/4.4/
debian stable main' > /etc/apt/sources.list.d/kibana.list"
```

TIP If we were running on Red Hat or a derivative we would install the appropriate Yum repository. See [this documentation for details](#).

We then run an [apt-get update](#) to refresh our package list.

Listing 8.56: Updating the package list for Kibana

```
$ sudo apt-get update
```

And finally we install Kibana itself.

Listing 8.57: Installing Kibana via apt-get

```
$ sudo apt-get install kibana
```

Now we just need to ensure Kibana starts when our host boots.

Listing 8.58: Starting Kibana at boot

```
$ sudo update-rc.d kibana defaults 95 10
```

Kibana is now installed.

TIP The packages install Kibana into the /opt directory, /opt/kibana.

Configuring Kibana

To configure Kibana we use the `kibana.yml` file in the `/opt/kibana/config` directory. This is a well-commented, YAML-based configuration file. The major items

we might want to configure are the interface and port we want to bind Kibana to and the Elasticsearch cluster we wish to use for queries. All of those settings are at the top of our configuration file.

Listing 8.59: The Kibana kibana.yml configuration file

```
# Kibana is served by a back-end server. This controls which port
# to use.
server.port: 5601

# The host to bind the server to.
server.host: "0.0.0.0"

# The Elasticsearch instance to use for all your queries.
elasticsearch.url: "http://localhost:9200"

...
```

Here we see that our Kibana console will be bound on port **5601** on all interfaces. The console will point to an Elasticsearch server located at **http://localhost:9200** by default. We need to change this to point at a host in our Elasticsearch cluster. Let's change it to:

Listing 8.60: Updating the Elasticsearch instance

```
# The Elasticsearch instance to use for all your queries.
elasticsearch.url: "http://es1.example.com:9200"

...
```

The configuration file also contains other settings, including the ability to setup TLS-secured access to Kibana and to load plugins to enhance or provide additional functionality to the dashboard.

Running Kibana

To run Kibana we use the `kibana` service.

Listing 8.61: Running Kibana

```
$ sudo service kibana start
```

This will run the Kibana console as a service on port 5601 of our `logstash.example.com` host. We browse to Kibana by opening a browser and connecting to `http://logstash.example.com:5601`.

When we launch Kibana we'll be prompted to specify what Elasticsearch indexes to search on and configure a default timestamp field.

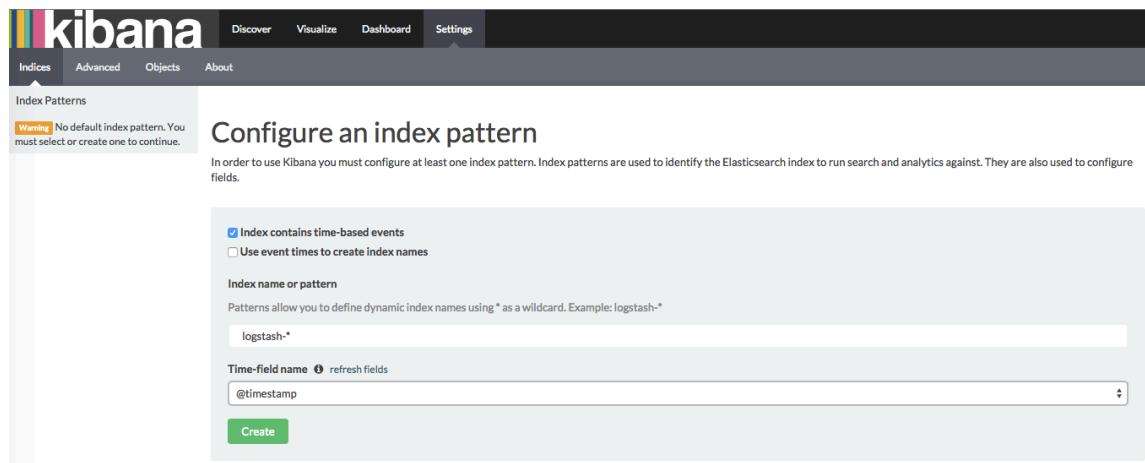


Figure 8.3: Configuring Kibana

The default index specification is `logstash-*`, which is a wildcard match for all our Logstash-delivered Elasticsearch indexes. We've also selected the `@timestamp` field to be used as Kibana's global time filter.

Now click the **Create** button. This will complete Kibana's configuration.

NOTE Kibana will create a special index called `.kibana` to hold its configuration. Don't delete this index or you'll need to reconfigure Kibana.

Next click **Discover** on the top menu bar to be taken to Kibana's basic interface. The **Discover** interface lists all events received and all the fields available to us. It includes a historical graph of all events received and a listing of each individual event sorted by the latest event received.

It also includes a query engine that allows us to select subsets of events.

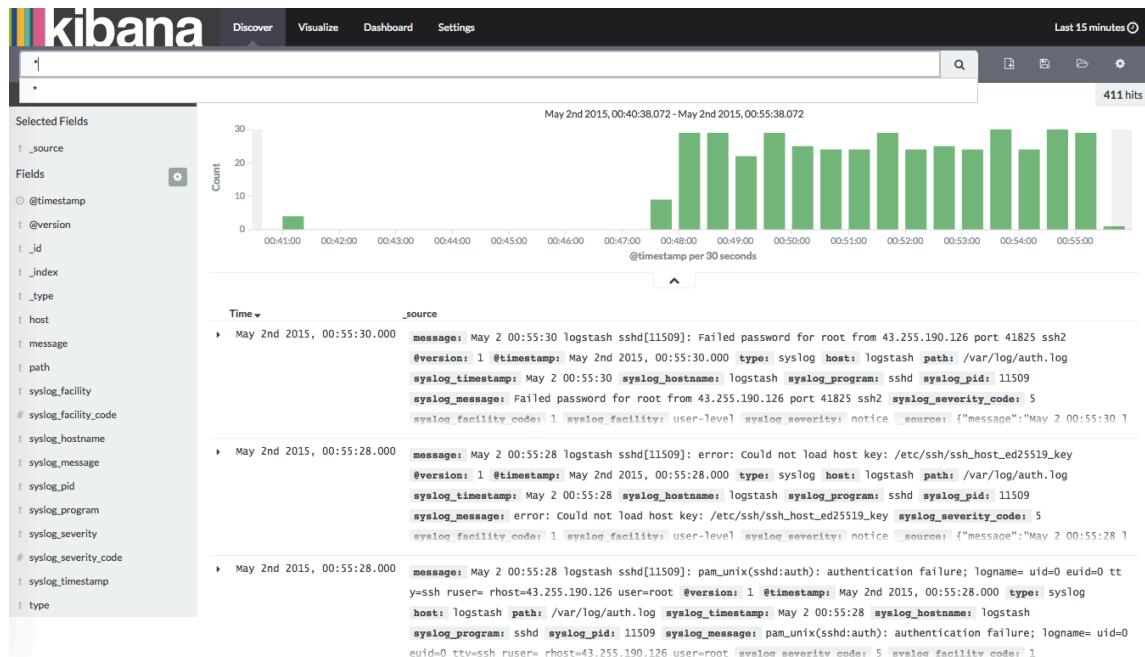


Figure 8.4: The Kibana Dashboard

Using Kibana

We're not going to show you how to do much with Kibana itself, primarily because web consoles change frequently and that sort of information dates quickly. Thankfully, there are some really good sources of documentation that can help.

The [Kibana User Guide](#) at the Elastic.co site is excellent and provides a detailed walk through of using Kibana to search, visualize, and build dashboards for logs.

There are also some great blog posts on Kibana [in the Digital Ocean documentation](#) and [in this blog post from Tim Roes](#).

Connecting our hosts to Logstash via Syslog

Now that we've gotten our ELK stack set up, we can start using it. To do this we need to configure hosts in our environment to connect to Logstash and send logs. But we're not going to install Logstash on any of our hosts. It's a JVM-based application and a little too performance heavy to run on application and service hosts. Instead we're going to make use of Syslog on these hosts and configure our central Logstash server to receive these events.

Configuring Logstash

Earlier in the chapter we configured Logstash to parse Syslog logs, albeit logs collected via the `file` input plugin. Now let's add support for receiving them via the network. To do this, we'll revisit our Logstash configuration and add a couple of new plugins to it. Let's open our `logstash.conf` on the `logstasha.example.com` host.

Listing 8.62: Adding Syslog receivers

```
input {
    tcp {
        port => 5514
        type => "syslog"
    }
    udp {
        port => 5514
        type => "syslog"
    }
}
filter {
    if [type] == "syslog" {
        grok {
            match => { "message" => "%{SYLOGTIMESTAMP:syslog_timestamp
                } %{SYLOGHOST:syslog_hostname} %{DATA:syslog_program
                }(?:\[%{POSINT:syslog_pid}\])?: %{GREEDYDATA:
                syslog_message}" }
        }
        syslog_pri { }
        date {
            match => [ "syslog_timestamp", "MMM d HH:mm:ss", "MMM dd
                HH:mm:ss" ]
        }
    }
}
output {
    elasticsearch {
        hosts => [ "es1.example.com", "esa2.example.com", "esa3.
            example.com" ]
    }
}
```

You can see we've added two new input plugins to our `input` block: `tcp` and `udp`. These are inputs that open ports on our Logstash host to receive events. In this case we're opening TCP and UDP ports `5514` on our host. We're opening a high port, `5514`, because Logstash isn't running as the `root` user and so we can't bind Syslog's traditional port: `514`.

NOTE In most Syslog implementations port 514 is the default port used by Syslog to send events between hosts.

Now we restart Logstash to enable these new plugins.

Listing 8.63: Restarting Logstash to enable new inputs

```
$ sudo service logstash restart
```

We confirm the plugins have bound correctly via the `netstat` command.

Listing 8.64: Running netstat

```
$ sudo netstat -an | grep '5514'  
tcp    0      0 0.0.0.0:5514    0.0.0.0:*      LISTEN  
udp    0      0 0.0.0.0:5514    0.0.0.0:*
```

NOTE If you find that Logstash has unexpectedly only bound to IPv6 ports, you might need to tweak your Logstash Java configuration slightly. Open the `/etc/default/logstash` file and uncomment and update this line `LS_JAVA_-`

OPTS="-Djava.net.preferIPv4Stack=true".

This will now allow us to send Syslog logs from the other hosts in our Production A environment. Let's set that up now.

A quick introduction to Syslog

Syslog is one of [the original standards](#) for computer logging. It was designed by Eric Allman as part of Sendmail and has grown to support logging from a variety of platforms and applications. It has become the default mechanism for logging on Unix and Unix-like systems and is heavily used by applications, non-host devices like printers, and networking devices like routers, switches, and firewalls.

As a result of its ubiquity on these types of platforms, it is a commonly used means to centralize logs from disparate sources. Each message generated by Syslog (and there are variations between platforms) is roughly structured like so:

Listing 8.65: A Syslog message

```
Dec 31 14:29:31 logstash systemd-logind[2113]: New session 31581
of user bob.
```

A Syslog message will contain a timestamp, the host that generated the message (here **logstash**), the process and process ID (PID) that generated the message, and the content of the message.

Messages also have metadata attached to them in the form of facilities and severities. Messages refer to a facility like:

- AUTH
- KERN

- MAIL
- etc...

The facility specifies the type of message generated. Messages from the **AUTH** facility usually relate to security or authorization, from the **KERN** facility they're usually kernel messages, and if from the **MAIL** facility it's an indication it was generated by a mail subsystem or application. There are a wide variety of facilities including custom facilities, prefixed with **LOCAL** and a digit—**LOCAL0** to **LOCAL7**—that you can use for your own messages.

Messages also have a severity assigned, for example, **EMERGENCY**, **ALERT**, and **CRITICAL**, ranging down to **NOTICE**, **INFO**, and **DEBUG**.

TIP You can find more details on Syslog [here](#).

Configuring Syslog

Syslog is installed and running on most Linux hosts by default. Most modern Linux distributions use a Syslog daemon called RSyslog. The [RSyslog daemon](#) has become popular on many distributions. Indeed it has become the default Syslog daemon on recent versions of Ubuntu, CentOS, Fedora, Debian, openSuSE, and others. It can process log files, handle local Syslog, and it comes with a modular plugin system.

TIP In addition to supporting Syslog output, Logstash also supports the RSyslog-specific [RELP](#) protocol.

To connect RSyslog to Logstash we're going to configure RSyslog to forward all logs to Logstash. RSyslog is configured in two places. The basic configuration is in the file `/etc/rsyslog.conf`. Most distributions also have a `/etc/rsyslog.d` directory in which we place configuration snippets.

To configure forwarding we're going to create a snippet inside the `/etc/rsyslog.d` directory. First let's create a file:

Listing 8.66: Creating an RSyslog snippet for forwarding

```
$ sudo vi /etc/rsyslog.d/30-forwarding.conf
```

RSyslog loads configuration files in alphanumeric order, so we'd normally prefix any file with a number, such as `30-`, to help specify a load order. On Ubuntu, for example, the default RSyslog configuration is loaded in `50-default.conf` in the `/etc/rsyslog.d` directory. We've specified a prefix of `30-` so our configuration loads first.

Let's populate this file right now.

Listing 8.67: Configuring RSyslog for Logstash

```
*.* @@logstash.example.com:5514
```

NOTE If you specify the hostname, here `logstash.example.com`, your host will need to be able to resolve it via DNS. For performance reasons, it may better to specify an IP address here.

This tells RSyslog to send all messages, indicated by: `*.*`, to the host `logstash.example.com`. The `*.*` indicates all facilities and priorities. You can use specific facilities or priorities instead, if you wish. For example:

Listing 8.68: Specifying RSyslog facilities or priorities

```
mail.* @@logstash.example.com:5514  
*.emerg @@logstash.example.com:5514
```

The first line would send all `mail` facility messages, and the second would send all messages of `emerg` priority.

The `@@` tells RSyslog to use TCP to send the messages. Specifying a single `@` uses UDP as a transport.

TIP We strongly recommend using the more reliable and resilient TCP protocol to send your Syslog messages. Unlike UDP it has guarantees of delivery.

We then restart the RSyslog daemon, like so:

Listing 8.69: Restarting RSyslog

```
$ sudo service rsyslog restart
```

Our host will now be sending all the messages collected by RSyslog to our Logstash server.

The RSyslog imfile module

In addition to logs generated via Syslog, we can collect logs from specific files. One of RSyslog's modules provides another method of sending log entries from RSyslog. You can use the [imfile module](#) to send the contents of files on the host via Syslog. The [imfile](#) module works much like Logstash's [file](#) input, supports file rotation, and tracks the currently processed entry in the file.

TIP If you don't want to use RSyslog to collect log entries from files, Elastic has released a series of collection agents called beats. Specifically there is a beat called [Filebeat](#) for collecting log entries from files. Beats are also available for collecting wire data from Windows Event Logs and a variety of other sources.

To send a specific file via RSyslog we need to enable the [imfile](#) module and then specify the file to be processed. For example, to collect events from the [/var/log/riemann/riemannlog](#) log file, we would create a snippet in [/etc/rsyslog.d/](#).

Listing 8.70: Collecting the riemann.log file

```
$ sudo vi /etc/rsyslog.d/riemann.conf
```

We'd populate this file like so:

Listing 8.71: Monitoring files with the imfile module

```
module(load="imfile" PollingInterval="10")

input(type="imfile"
      File="/var/log/riemann/riemann.log"
      Tag="riemann:")
```

The first line loads the `imfile` module and sets the polling interval for events to 10 seconds. It only needs to be specified once in the configuration and can be adjusted for more frequent updates, if required.

The next block specifies the file from which to collect events. It has a `type` of `imfile`, telling RSyslog to use the `imfile` module. The `File` attribute specifies the name of the file to poll. The `File` attribute also supports wildcards.

Listing 8.72: Monitoring files with an imfile wildcard

```
input(type="imfile"
      File="/var/log/riemann/*.log"
      Tag="riemann:")
```

This would collect all events from all files in the `/var/log/riemann` directory with a suffix of `.log`.

Lastly, the `Tag` attribute tags these messages in RSyslog with a tag of `riemann:`. Note the trailing `:` on the tag—this is important to ensure a tag is correctly parsed. You should always end a tag with a colon.

In another example we could add the `/var/log/collectd.log` log file we created in Chapter 5 to log collectd's events.

Listing 8.73: Monitoring the collectd log file

```
input(type="imfile"
      File="/var/log/collectd.log"
      Tag="collectd:")
```

Here we're again specifying the `imfile` type and the specific file, and we're adding a tag of `collectd:` to the log entries.

The `imfile` module also supports state management. RSyslog keeps track of how much of the file has been processed, as well as tracking changes like file rotations.

TIP You can find the full RSyslog documentation [on the RSyslog website](#).

Logging from Docker

In Chapter 7 we talked about metric generation from Docker. We can also extract logs from our Docker containers and daemon. The Docker daemon has a series of logging plugins, including plugins that write events to Syslog or to file on disk. We're going to take advantage of a combination of the Syslog infrastructure we just configured and the `syslog` Docker logging plugin to send events from Docker and containers to Logstash.

The Docker daemon provides two levels of logging configuration: globally at the daemon level, and locally at runtime when a container is created. Initially we're going to configure Docker to log globally, but we'll demonstrate how to log at runtime too. To configure the Docker daemon for logging, we need to pass the

daemon some arguments at startup.

Configuring the Docker Daemon for logging

On both the Ubuntu and the Red Hat family of distributions, the Docker daemon is configured by system configuration files. On Ubuntu we edit the `/etc/default/docker` file and on Red Hat the `/etc/sysconfig/docker` file. Both files have an environment variable called `DOCKER_OPTS` that controls the arguments passed to the `docker daemon` command executed when the Docker daemon is started.

TIP You may need to uncomment the variable in the file.

Let's update that variable to include a logging plugin and some configuration. Recalling that we configured a Logstash Syslog receiver on host `logstash.example.com` on port `5514`, let's configure that as our destination.

Listing 8.74: Configuring the `DOCKER_OPTS` option for logging

```
DOCKER_OPTS="--log-driver=syslog --log-opt tag=\"{{.Name}}/{{.ID}}\"
           --log-opt syslog-address=tcp://logstash.example.com:5514"
```

NOTE You might need to append this configuration to the `DOCKER_OPTS` variable if you already have existing configuration in there.

We've configured two different options. The first, `--log-driver`, controls which

logging driver plugin will be loaded. We've set it to `syslog`. We've also configured two `--log-opt` options.

WARNING Enabling the `syslog` Docker logging plugin will prevent you from being able to run the `docker logs` command locally on the Docker daemon, as all logs are being sent to Syslog.

The `--log-opt` flag can be specified more than once to configure elements of each logging plugin. In this case we've first specified the flag with the `tag` option. The `tag` option controls additional information about our containers that we might want to send to our logging service. In this case we add the container name and container ID to our log entries. This results in a log entry like this:

Listing 8.75: Docker logging tags

```
2015-11-28T20:24:04Z docker docker/container_name/5790672ab6a0  
[9103]: Hello from Docker.
```

It looks like a standard Syslog entry, except our Syslog program and PID have the name and ID of the container in between the program name, `docker`, and the process ID, `9103`. We'll parse out this information using the `grok` filter when it arrives at Logstash. You can add a variety of tags, including the name of the image the container is running from and any labels applied to the container. The `tags` field is a [Go template](#) and has access to everything in the [logging context](#).

For the second `--log-opt` flag we specified our logging destination using the `syslog-address` option. We configure a Syslog destination of:

`tcp://logstash.example.com:5514`

This points to the Logstash server listening on TCP port `5514` on the `logstash.example.com` host. We could also do UDP, if we wished, by prefixing the server address with `udp://`.

Any of these options can be specified with the `docker run` command. Using a different logging driver when you run a container from the command line will override the choice and configuration of the logging plugin enabled for the Docker daemon.

To enable our logging driver, we need to restart the Docker daemon.

Listing 8.76: Restarting Docker to enable logging

```
$ sudo service docker restart
```

Now the Docker daemon, or any Docker containers, will be sending events to the Logstash server. When those events land on the Logstash server they'll be grabbed and marked as type `syslog` and processed by the `grok` filter. Let's revisit that configuration.

Listing 8.77: Our Logstash Syslog receivers

```
input {
  tcp {
    port => 5514
    type => "syslog"
  }
  ...
}

filter {
  if [type] == "syslog" {
    grok {
      match => { "message" => "%{SYLOGTIMESTAMP:syslog_timestamp
} %{SYLOGHOST:syslog_hostname} %{DATA:syslog_program
}(?:\[ %{POSINT:syslog_pid}\])?: %{GREEDYDATA:
syslog_message}" }
      remove_field => ["message"]
    }
    syslog_pri { }
    date {
      match => [ "syslog_timestamp", "MMM d HH:mm:ss", "MMM dd
HH:mm:ss" ]
    }
  }
}
.
.
```

We see our `tcp` input on port `5514` which applies the `syslog` type to any received events. These events are then processed by our filters. But our current Syslog `grok`

filter is configured for standard Syslog lines, and we will quickly see that events from Docker aren't parsed correctly, resulting in events that look like:

Listing 8.78: Grok parse failed Docker event

```
{  
  "message" => "<30>2015-11-28T20:24:04Z docker docker/07  
  c15432c076[16829]: hello world",  
  "@version" => "1",  
  "@timestamp" => "2015-11-28T20:24:04.254Z",  
  "host" => "docker",  
  "type" => "syslog",  
  "tags" => [  
    [0] "_grokparsefailure"  
  ],  
  "syslog_severity_code" => 5,  
  "syslog_facility_code" => 1,  
  "syslog_facility" => "user-level",  
  "syslog_severity" => "notice"  
}
```

You can see our Docker log event in the `message` field, but the rest of our event hasn't fared well. Indeed, the attempt by the `grok` filter to parse it has failed. When this happens, the `grok` filter will add a tag to the event called `_grokparsefailure`.

NOTE We recommend you monitor for this tag in your Logstash logs as it'll let you know when your Logstash is receiving events and not correctly parsing them. Personally, we have a Kibana graph and a search specifically configured for this tag to let us know when it happens.

To fix this we need to update the regular expression that matches the `message` field in the `grok` filter. Let's first identify what is different about our Docker log message.

Listing 8.79: Docker log message

```
<30>2015-11-28T20:24:04Z docker docker/daemon_dave/07c15432c076  
[16829]: hello world
```

We see we're using a different timestamp from the default Syslog timestamp. Thankfully it's a standard ISO8601 timestamp and there's an existing Grok pattern for it. But we'll also need to update our `date` filter to reflect that. We'll do that next.

Note that the Syslog program and PID is prefixed with the Docker container's name and ID as we added using the Docker logging tag option, here `daemon_dave` and `07c15432c076`. We need to update our regular expression to handle that too.

Listing 8.80: An updated Docker grok regex

```
grok {  
    match => { "message" => "(?:%{SYSLOGTIMESTAMP:syslog_timestamp}  
    |%{TIMESTAMP_ISO8601:syslog_timestamp}) %{SYSLOGHOST:  
    syslog_hostname} %{DATA:syslog_program}(?:\%{DATA:  
    container_name}\%{DATA:container_id})?(?:\[%\{POSINT:  
    syslog_pid\]\])?: %{GREEDYDATA:syslog_message}" }  
    remove_field => [ "message" ]  
}
```

Now our `grok` match can take optional ISO8601 timestamps as well as our regular Syslog timestamps. We've also added two new optional fields, `container_name` and `container_id`, that are only populated if the message is a Docker log message. These fields will contain the Docker container name and ID.

TIP Remember the [Grok Debugger](#) is useful for parsing and creating grok regular expressions.

We also need to update the `date` filter to mention that we need to support an additional time format, ISO8601. Adding a new format is easy, especially when it is a known and named format.

Listing 8.81: Updated date filter for Docker

```
date {  
    match => [ "syslog_timestamp", "MMM d HH:mm:ss", "MMM dd  
              HH:mm:ss", "ISO8601" ]  
}
```

Here we've added the `ISO8601` format to the end of our list of supported formats for the `syslog_timestamp` field.

Now if we HUP or restart Logstash we should be able to parse our incoming Docker log messages.

Listing 8.82: Restarting Logstash to enable Docker logging

```
$ sudo service logstash restart
```

If we now see a similar Docker log message we'll see it's been properly parsed, and we'll have also gained some useful contextual information: the container name and ID.

Listing 8.83: A properly parsed Docker log message

```
{  
    "message" => "<30>2015-11-28T21:51:26Z docker  
                  docker/daemon_dave/d6cfce59a1d1[23338]: hello  
                  world",  
    "@version" => "1",  
    "@timestamp" => "2015-11-28T21:51:26.000Z",  
    "host" => "docker",  
    "type" => "syslog",  
    "syslog_timestamp" => "2015-11-28T21:51:26Z",  
    "syslog_hostname" => "docker",  
    "syslog_program" => "docker",  
    "container_name" => "daemon_dave",  
    "container_id" => "d6cfce59a1d1",  
    "syslog_pid" => "23338",  
    "syslog_message" => "hello world",  
    "syslog_severity_code" => 5,  
    "syslog_facility_code" => 1,  
    "syslog_facility" => "user-level",  
    "syslog_severity" => "notice"  
}
```

With this, our Logstash server now has our Docker daemon and container logs flowing in. This allows us to log events from the daemon and any running containers, and provides a diagnostic history useful for identifying and resolving faults.

Sending data from Logstash to Riemann

Now that we have some events flowing from our hosts into Logstash, let's see how to send events and metrics from Logstash to Riemann and then onto Graphite. This is useful for when we want to generate metrics from specific log events and emit them. Let's see how this works by creating a metric measuring event throughput in Logstash, the rate of events processed by Logstash.

First, let's install a new output plugin, `riemann`, on our Logstash host. The `riemann` output plugin is a community-maintained plugin to connect Logstash to Riemann. It doesn't ship with Logstash by default—we need to use the `plugin` command to install it.

Listing 8.84: Installing the Riemann Logstash output

```
$ sudo /opt/logstash/bin/plugin install logstash-output-riemann
Validating logstash-output-riemann
Installing logstash-output-riemann
Installation successful
```

TIP You can read more about working with external Logstash plugins in the [Elasticsearch Logstash plugin documentation](#).

Now let's take our existing Logstash configuration and add a new filter called `metrics` to it and our new `riemann` output.

Listing 8.85: The Logstash metrics filter

```
filter {  
    . . .  
  
    metrics {  
        meter => "events"  
        add_tag => "logstash_events"  
    }  
}  
  
output {  
    . . .  
  
    if "logstash_events" in [tags] {  
        riemann {  
            host => "riemann.example.com"  
            map_fields => true  
            riemann_event => {  
                "service" => "Logstash events"  
                "host" => "logstasha.example.com"  
            }  
        }  
    }  
}
```

The `metrics` filter creates metrics from Logstash events. It has two modes: `meter` and `timer`. In the `meter` mode it will count events and then emit a new event containing the total number of events and the short, mid, and long-term rates of

events. The default short, mid, and long-term measures are customizable but default to one minute, five minutes, and 15 minutes, much like Linux load averages.

In `timer` mode the `metrics` filter emits the same event but also includes some statistics on the field we specify. For example, we might be collecting the access logs from an Apache web server which includes HTML status codes and response times. You could create a metric from the response time, and the `metrics` filter could output rates as well as min, max, mean, standard deviations, and percentile output.

In this case our `meter => "events"` line above would emit the following event:

Listing 8.86: Logstash metrics event

```
{  
    "@version" => "1",  
    "@timestamp" => "2015-08-05T00:36:02.773Z",  
    "message" => "logstash",  
    "events.count" => 1066562,  
    "events.rate_1m" => 1222.4379898784412,  
    "events.rate_5m" => 1175.383369199709,  
    "events.rate_15m" => 766.9274163646223,  
    "tags" => [  
        [0] "logstash_events"  
    ]  
}
```

Our event contains the typical fields: `@timestamp`, `@version`, and a `message` of `logstash`. We also see four custom fields that the `metrics` filter has automatically created in the emitted event: `events.count`, `events.rate_1m`, `events.rate_5m`, and `events.rate_15m`. This is the count of events that have passed through the

`metrics` filter, plus the rate at one minute, five minute, and 15-minute intervals. The custom fields are prefixed with the value we specified for the `meter` option: `events`. This, plus a period, `.`, creates a name for the metric.

Here every event is passing through the `metrics` filter so we've effectively created statistical rates for Logstash's event processing. We could also use tags, the `type` field, or other selectors to grab specific events and create metrics from them.

It's also these metrics fields that we're going to pass to Riemann. Note that we've added a tag to our filter of `logstash_events`. This helps us identify specific events we want to send to Riemann in the `output` section.

In the `output` section we've matched all events with the tag `logstash_events`. Inside that match we've specified `the riemann output`.

Listing 8.87: The Riemann output

```
riemann {
    host          => "riemann.example.com"
    sender        => "%{syslog_hostname}"
    map_fields    => true
    riemann_event => {
        "service" => "Logstash events"
        "metric"   => "%{events.rate_1m}"
    }
}
```

The `host` option tells the output which Riemann server to send our events to, here `riemann.example.com`. The `sender` option controls the source host of the event—this will populate the `:host` field of our Riemann event. We're going to use the contents of the `syslog_hostname` field. The `%{syslog_hostname}` syntax is a [field reference](#). Field references allow us to refer to event fields in Logstash

configuration. The `map_fields` option tells the `riemann` output to map any custom fields in the Logstash event to custom fields in the corresponding Riemann event.

Finally, the `riemann_event` option allows us to map fields or create fields in the Riemann event. Here we've told the `riemann` output that the `:service` field on our Riemann event should specify `Logstash events`, and the `:metric` field should take its value from the `events.rate_1m` field.

This configuration will result in a Riemann event like so:

Listing 8.88: Logstash metric event in Rieman

```
{:host logstasha.example.com, :service Logstash events, :state ok
  , :description logstash, :metric 1199.8928828641517, :tags [
    logstash_events], :time 1438738018, :ttl 60, :events.rate_15m
  610.7294184757806, :events.rate_5m 1055.0701220952672, :events.
  rate_1m 1199.8928828641517, :events.count 768848, :message
  logstash}
```

We see the default Riemann fields, including our `:metric` field, but also some custom fields like `:events.rate_5m`. So, in addition to matching on the `:metric` field as we've done previously, we could match on one of these custom fields too.

TIP You can also output your metrics directly to Graphite if you would prefer using the [graphite output](#).

Sending data from Riemann to Logstash

In addition to sending metrics to Riemann from Logstash, we can do the reverse. If data coming into Riemann is useful for diagnostics purposes we can also send that data into Logstash. To make this easier, Riemann comes with a Logstash plugin we can enable on our Riemann server. Let's start by doing that.

We're going to add some configuration for the Logstash plugin, much like how we configured the Graphite plugin in Chapter 4. Let's create a file called `logstash.clj` in `/etc/riemann/examplecom/etc/` to hold that configuration.

Listing 8.89: The logstash.clj configuration file

```
(ns examplecom/etc/logstash
  (:require [riemann.logstash :refer :all]))

(def logstash (async-queue! :logstash {:queue-size 1000}
  (logstash {:host "logstash" :port 2003 :port-size
  20})))
```

We first add a namespace, `examplecom/etc/logstash`, and require the `riemann.logstash` library which contains Riemann's Logstash driver.

We've created a new var called `logstash`. Like our Graphite connection from Chapter 4, we've specified an `async-queue!` to ensure that sending events to Logstash isn't a blocking action for Riemann. We've called that queue `:logstash` and specified a queue size of `1000`. Inside our queue we've specified the `logstash` plugin with two options: `:host` and `:port`. We've set the `:host` option to `logstash.example.com`, the name of our Logstash server. We're going to assume our DNS will resolve this correctly. We've also specified a `:port` of `2003`. Lastly, we've specified a `:pool-size` of `25`. The `:pool-size` controls how many connections

Riemann will keep open to Logstash. If you experience issues with throughput from Riemann to Logstash you can experiment with increasing the pool size.

If we were to use the `logstash` var, much like the `graph` var we created in Chapter 4, then any events would be sent from Riemann to Logstash. Let's see that in action now.

Listing 8.90: Using the logstash var

```
(require '[examplecom.etc.logstash :refer :all])  
.  
.  
.  
(tagged "logstash_events"  
  logstash  
)
```

We've first required our `examplecom.etc.logstash` functions in `riemann.config`. Here, every event tagged with `logstash_events` is identified by a `where` stream and passed to the `logstash` var.

Now we need to configure the Logstash end of the connection. To do this we're going to add a new incoming TCP input running on port `2003`. Our incoming Riemann events are emitted as JSON hashes, so parsing them is easy using Logstash's built-in `json` codec.

Let's add some configuration to our `/etc/logstash/conf.d/logstash.conf` file.

Listing 8.91: Adding a Riemann receiver

```
input {  
    tcp {  
        port => 5514  
        type => "syslog"  
    }  
    tcp {  
        port => 2003  
        type => "riemann"  
        codec => "json"  
    }  
    . . .
```

We've added a new `tcp` input running on port `2003`. We set a `type` of `riemann` for any events received on this input, and we use the `json` codec to parse any incoming events into Logstash's message format from JSON. To enable our configuration we need to restart Logstash.

Listing 8.92: Restarting Logstash for our Riemann events

```
$ sudo service logstash restart
```

Now, if an event is sent via the `logstash` var, it'll appear in Logstash. Let's look at an example collectd event from Chapter 5 as it would be seen in Logstash.

Listing 8.93: A Riemann event in Logstash

```
{  
    "host" => "tornado-web1",  
    "service" => "cpu user",  
    "state" => "ok",  
    "description" => nil,  
    "metric" => 0.9950248756218906,  
    "tags" => [  
        [0] "collectd"  
    ],  
    "time" => "2015-12-09T12:43:58.000Z",  
    "ttl" => 60.0,  
    "source" => "tornado-web1",  
    "ds_index" => "0",  
    "ds_name" => "value",  
    "ds_type" => "gauge",  
    "type_instance" => "user",  
    "type" => "percent",  
    "plugin" => "cpu",  
    "@version" => "1",  
    "@timestamp" => "2015-12-09T12:44:08.444Z"  
}
```

Our collectd event has been sent from Riemann, converted from JSON into Logstash's event format, and here, outputted as debug. We wouldn't generally send our collectd events to Logstash—we're already graphing them in Graphite after all—but this shows how easy it is to send events from Riemann to Logstash.

WARNING You need to be careful not to create feedback loops between Riemann and Logstash. Ensure any events sent from Riemann to Logstash and vice versa are not looped between the two destinations

Scaling Elasticsearch and Logstash

In our current configuration, there's a single Logstash server in each environment, each receiving our events and passing them to a cluster of three Elasticsearch servers. This addresses our immediate needs for an environment, but at some point we may need to scale Logstash and Elasticsearch.

Scaling Logstash

For Logstash, we have a few major options to scale our current solution:

- Partitioning
- Load balancing
- Brokering

The first option is partitioning. Partitioning means that Logstash servers process events from a collection of hosts—for example, a Logstash server for a specific application stack or rack or environment. Our current configuration, a Logstash server per environment, is an example of partitioning.

Setting up partitioning is relatively easy if you're using a configuration management system to manage your environments. Hosts, services, and applications can be configured to send events to specific servers—for example, configuring RSyslog forwarding for a subset of hosts to a specific Logstash server.

We can also partition at the next level up. The most expensive performance costs on a Logstash server are processing events, such as **grok** filtering. So we configure a Logstash server and configure it to simply route events, for example:

Listing 8.94: A routing configuration for Logstash

```
input {
  tcp {
    port => 5514
    type => "syslog"
  }
  udp {
    port => 5514
    type => "syslog"
  }
}
output {
  if [host] == "tornado-web1" {
    tcp {
      host => "logstash1.example.com"
      port => 5514
    }
  }
  if [host] == "tornado-db" {
    tcp {
      host => "logstash2.example.com"
      port => 5514
    }
  }
}
```

Here we've configured a Logstash server to receive events using the `tcp` and `udp` input plugins on port `5514`. We've not specified any filter plugins; instead we've

passed our events straight to output plugins. In the output block we've specified `if` clauses to select events based on their source host and route them via the `tcp` output plugin to one of two new Logstash servers. Our events are then partitioned at that Logstash server, and specific processing would take place on the Logstash servers to which the events were forwarded.

Upstream, on the Logstash servers, we configure Logstash itself to ensure each server has the required and associated configuration to process events from that subset.

Our second method for scaling Logstash is load balancing. In this approach we leave our local configuration on hosts and services pointing at a DNS name, such as `logstasha.example.com`. We then run as many Logstash servers as required, each with identical configuration, behind a HAProxy cluster using something like `keepalived` to help manage the cluster. With identical configuration on each Logstash server, a round robin or least-connection algorithm on HAProxy will fairly distribute connections to the Logstash servers. Load balancing has more moving parts but is far simpler for ongoing maintenance, as the backends can be scaled without requiring configuration changes in the sender.

Lastly, we can also insert a broker between our collection agent and the Logstash server, or between the Logstash server and Elasticsearch. A broker is a layer that buffers and stores log entries prior to sending them between ELK components. It usually has greater capacity for receiving events and allows you to broker high volumes of events prior to them hitting a component. Example brokers could include [Redis](#) and [Kafka](#).

All approaches have pros and cons. Partitioning is a technically simple solution that can be managed simply and easily with configuration management. But it can, depending on the granularity of the partition, require a large number of hosts and potentially result in excess capacity being provisioned. Load balancing is a more technologically complex solution and requires more in-depth and dynamic configuration and operational management. It does, however, have the advantage of generally being able to grow as additional capacity is required.

Whatever solution we select, we'd then send all our events into our existing Elasticsearch cluster, scaling that cluster as needed.

Scaling Elasticsearch

For Elasticsearch, our scaling story is pretty simple. The cluster can be expanded with additional nodes, some nodes can be designated as masters, some clients designated as nodes, and others as pure data nodes to assist in optimizing performance.

The Elastic team offers some excellent resources that discuss scaling:

- The [Scaling Guide](#).
- Chartbeat's [useful guide on scaling Elasticsearch and Logstash](#).
- And Wilfred Hughes' [great post on taming Elasticsearch clusters](#) is hugely useful.

Each of these contains ideas for tuning and growing Elasticsearch clusters.

Monitoring our components

Finally we want to make sure all of the components we've just set up keep working, are efficient, and that we get told if any of them fail. Let's build some monitoring for each component.

Monitoring RSyslog

First we want to make sure RSyslog keeps running and reports to us when it fails. To do this we're going to make use of collectd's process monitoring. In Chapters 5 and 6 we saw how to use the [processes](#) plugin to monitor specific processes. Let's add some monitoring for RSyslog.

Let's create a file, `rsyslogd.conf`, in `/etc/collectd.d/` to hold our configuration.

Listing 8.95: Creating the rsyslogd.conf file

```
$ sudo vi /etc/collectd.d/rsyslogd.conf
```

Now let's populate that file.

Listing 8.96: The rsyslogd.conf file

```
LoadPlugin processes
<Plugin "processes">
    Process "rsyslogd"
</Plugin>
```

Note that we're monitoring the `rsyslogd` process, and events from this process will pass through to Riemann to be tracked. If the process drops below the one process threshold we configured in Chapter 6 then a failure event will be generated and we'll be notified.

Monitoring Logstash

As Logstash (and Elasticsearch) are both JVM-based applications, we can take advantage of the [Java Management Extensions](#) or JMX monitoring framework. JMX is a monitoring and measurement integration for Java. It's shipped with most Java distributions, and we can enable it on a case-by-case basis for specific applications.

TIP An open source alternative to JMX is [Kamon](#). Also available, for Java 7 and later, is [jstat](#), which you could use with a [simple Collectd plugin](#).

Configuring JMX for Logstash

Let's enable JMX for Logstash. We do that by updating the starting arguments for Logstash. On Ubuntu, we do this by editing the `/etc/default/logstash` file. On Red Hat and related distributions we'd update the `/etc/sysconfig/logstash` file.

In both cases we want to update the `LS_JAVA_OPTS` environment variable.

Listing 8.97: Updating JMX for the LS_JAVA_OPTS variable

```
LS_JAVA_OPTS="-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=8855 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false"
```

We've added the following options:

Listing 8.98: The JMX enabling options

```
-Dcom.sun.management.jmxremote  
-Dcom.sun.management.jmxremote.port=8855  
-Dcom.sun.management.jmxremote.authenticate=false  
-Dcom.sun.management.jmxremote.ssl=false
```

They enable JMX and configure it to report locally, bound to `localhost` on port `8855`. We've disabled authentication and SSL. If you'd like to use usernames and

passwords, you can find instructions [on the JMX site](#). We could also enable SSL, but it's probably not required when bound to a local host.

Now we restart Logstash to enable JMX.

Listing 8.99: Restarting Logstash to enable JMX

```
$ sudo service logstash restart
```

Configuring collectd for Logstash's JMX

We now need to configure a collectd plugin to scrape our JMX endpoint and retrieve the metrics. The collectd daemon comes with a Java plugin called [GenericJMX](#) to do this. We run the [GenericJMX](#) plugin by executing it with the [java](#) helper plugin, much like using the [python](#) helper plugin as we saw in Chapter 7.

Let's create a file to hold our plugin configuration.

Listing 8.100: The collectd JMX configuration file for Logstash

```
$ vi /etc/collectd.d/logstash_jmx.conf
```

Now let's populate that file. The whole configuration is a little bit large to paste here, but we've included a sampling to show you how it works. The whole file is available [here](#).

Listing 8.101: The logstash_jmx.conf file

```
LoadPlugin java
<Plugin "java">
  JVMARG "-Djava.class.path=/usr/share/collectd/java/collectd-api
          .jar:/usr/share/collectd/java/generic-jmx.jar"
  LoadPlugin "org.collectd.java.GenericJMX"

  <Plugin "GenericJMX">

    <MBean "memory">
      ObjectName "java.lang:type=Memory,*"
      InstancePrefix "java_memory"
      <Value>
        Type "memory"
        InstancePrefix "heap-"
        Table true
        Attribute "HeapMemoryUsage"
      </Value>

      <Value>
        Type "memory"
        InstancePrefix "nonheap-"
        Table true
        Attribute "NonHeapMemoryUsage"
      </Value>

    </MBean>
```

The first line loads the `java` plugin. This is the base plugin that enables all supporting Java code for collectd. We then configure the plugin operation inside a `<Plugin>` block. Inside this block we tell the `java` plugin where to find our `GenericJMX` plugin and then load that plugin.

We then have the `GenericJMX` plugin's configuration nested inside a second `<Plugin>` block.

We've specified two types of configuration for our `GenericJMX` plugin.

MBeans

MBeans or Managed Beans blocks that define a mapping of attributes to the types used by collectd to generate metrics. An MBean is a managed Java object. It's like a JavaBean component but for exposing a management interface through JMX. You can define MBeans for devices, applications or resources inside applications or the JVM. Each MBean exposes readable or writeable attributes, a set of operations and a description.

In our `GenericJMX` plugin configuration we've focused on general JVM-related metrics rather than application-specific attributes. We look at useful metrics for [managing the JVM's performance](#) like memory, garbage collection, invocations, and memory pools. Let's look at an example of an MBean.

Listing 8.102: An MBean

```
<MBean "memory-heap">
    ObjectName "java.lang:type=Memory"
    InstancePrefix "memory-heap"
    <Value>
        Type "memory"
        Table true
        Attribute "HeapMemoryUsage"
    </Value>
</MBean>
<MBean "memory-nonheap">
    ObjectName "java.lang:type=Memory"
    InstancePrefix "memory-nonheap"
    <Value>
        Type "memory"
        Table true
        Attribute "NonHeapMemoryUsage"
    </Value>
</MBean>
```

Our **MBean** block has a name, here **memory-heap**. This is the name of the MBean definition inside collectd. There is a further name called the **ObjectName** which is the name of the MBean inside JMX. In this case our MBean tracks memory state in the JVM.

We next define the **InstancePrefix**, which is an optional setting that prefixes our MBean values, so each value we define in this case will be prefixed with **memory-heap**. This helps us identify the source of individual values.

Next we define any values we're sourcing from the MBean. We enclose these in

<Value> blocks. Every MBean needs at least one <Value> block. The attributes in the <Value> block set the individual fields of our metrics.

We first set the **Type**, in our first value `memory`. This will help construct the metric type and name. The **Table** option specifies whether our value is a composite value or not. If `true`, then it assumes a composite value. Composite values are collections of data, each of which can be looked up by a key.

The last option, **Attribute**, is the name of the attribute inside our MBean from which we're going to read our value. In the case of our first value it'll read an attribute called `HeapMemoryUsage`.

So what metrics will we end up with from this <Value> block? In this case, collectd will lookup the `HeapMemoryUsage` attribute, which is a composite attribute made up of multiple values. It'll grab all of these values and prefix them with the `InstancePrefix` and the `Type` and generate some metrics much like:

Listing 8.103: GenericJMX Java memory heap metrics

```
GenericJMX-memory-heap/memory-committed  
GenericJMX-memory-heap/memory-init  
GenericJMX-memory-heap/memory-max  
GenericJMX-memory-heap/memory-used
```

Our collectd instance will then send these metrics onto Riemann. We can then use them to monitor the performance of Logstash. Specific metrics of interest that we might create graphs from are:

- Memory. The `memory-heap/memory-used` and `memory-nonheap/memory-used` metrics can help identify memory usage and leaks. Watch for growth here over time.
- Garbage collection. Monitoring the frequency of garbage collection and the length of time it took can indicate if there are memory leaks.

- Threads. Each thread consumes memory. A lot of threads opening or a lot of threads running can indicate memory bottlenecks or preempt OOM or “out of memory” errors.

TIP You can find a full list of the GenericJMX plugins settings in [the collectd plugin documentation](#).

Connections

The `<Connection>` blocks control where the plugin should look to find our JMX server.

Listing 8.104: The Connections block

```
<Connection>
    ServiceURL "service:jmx:rmi:///jndi/rmi://localhost:8855/
                  jmxrmi"
    Collect "classes"
    Collect "compilation"
    Collect "garbage_collector"
    Collect "memory"
    Collect "memory_pool"
    Collect "thread"
    Collect "thread-daemon"
</Connection>
```

Inside our `<Connection>` block we've defined the `ServiceUrl` that points to our

JMX instance running on `localhost` on port `8855`. We also specify a series of `Collect` options that tell collectd which MBeans to process for this connection. In this case we're retrieving all of the defined MBeans, including, for example, the `memory` MBean we just saw.

Enabling Logstash process monitoring

We also want to monitor the Logstash process. We do this using the same method and `processes` plugin as we used to monitor collectd and RSyslog.

First, let's create a file, `logstash.conf`, in `/etc/collectd.d/` to hold our configuration.

Listing 8.105: Creating the logstash.conf file

```
$ sudo vi /etc/collectd.d/logstash.conf
```

Now let's populate that file.

Listing 8.106: The logstash.conf file

```
LoadPlugin processes
<Plugin "processes">
    ProcessMatch "logstash" "logstash\runner.rb"
</Plugin>
```

You can see that we're monitoring the `logstash` process. If it fails our process threshold will trigger and failure will be generated and sent to Riemann. We'll then be notified that Logstash has failed.

Restarting collectd

Finally, to enable both our plugins, we restart the collectd daemon.

```
Listing 8.107: Restarting collectd to enable JMX
```

```
$ sudo service collectd restart
```

Monitoring Logstash with Riemann

Now we've got incoming metrics from our Logstash JVM instance and from our Logstash process. Our metrics will be tagged with `collectd` and Riemann will be automatically sending them through to Graphite. Once they are in Graphite we can construct appropriate graphs and add them to dashboards, etc.

Monitoring Elasticsearch

For monitoring Elasticsearch we've got a few different options, including monitoring the process itself. These include:

- The same JVM monitoring using JMX that we used for the Logstash application.
- The monitoring metrics exposed by Elasticsearch's statistics API endpoint.

We're going to add a new plugin to our existing installation to collect metrics from Elasticsearch. Let's start with grabbing that plugin. As we've learned, the collectd daemon comes with a few useful plugins that can help us run plugins written in a variety of languages. In this case our new plugin is written in Python so we'll again use the `Python` plugin to run it. Let's download the plugin.

Listing 8.108: Download the elasticsearch_collectd.py plugin

```
$ cd /usr/lib/collectd/
$ wget https://raw.githubusercontent.com/jamtur01/collectd-
  elasticsearch/master/elasticsearch_collectd.py
```

NOTE Credit for the original plugin goes to [SignalFuse](#) and [Jeremy Carroll](#).

Here we've changed into the `/usr/lib/collectd/` directory (on Red Hat this would be the `/usr/lib64/collectd/` directory) where collectd keeps all of our plugins. We've then downloaded the plugin from GitHub. You can take a look at its source. It defines a list of metrics and some configurable options. It also has a series of methods that read the [Elasticsearch stats API](#) and parse the metrics.

TIP You can learn about writing your own Python plugins on the [collectd Python man page](#).

Configuring the Elasticsearch collectd plugin

Now that we've installed the plugin we need to tell collectd about it and configure it. To do that we're going to add a configuration file for the plugin to the `/etc/collectd.d` directory called `elasticsearch.conf`.

Listing 8.109: Creating an elasticsearch.conf configuration file

```
$ sudo vi /etc/collectd.d/elasticsearch.conf
```

Now let's populate that file.

Listing 8.110: The elasticsearch.conf configuration file

```
<LoadPlugin "python">
    Globals true
</LoadPlugin>

<Plugin "python">
    ModulePath "/usr/lib/collectd/"

    Import "elasticsearch_collectd"

    <Module "elasticsearch_collectd">
        Verbose false
        Cluster "productiona"
    </Module>
</Plugin>
```

Our configuration file loads the `python` plugin. The `Globals true` line enables any Python standard libraries on your host so you can use them in your plugins.

Next we configure the `python` plugin. We tell collectd where to find our plugins, here `/usr/lib/collectd` (this would be `/usr/lib64/collectd` on Red Hat). We then import the `elasticsearch_collectd` plugin we just installed. This loads the

plugin into collectd.

Then we configure the plugin itself. We tell it to disable verbose logging via the `Verbose` option, and we tell it the Elasticsearch cluster name via the `Cluster` option, here `production`.

Enabling Elasticsearch process monitoring

We also want to monitor the Elasticsearch process. We again do this using the `processes` plugin.

We edit the file `elasticsearch.conf` in `/etc/collectd.d/` to add our configuration.

Listing 8.111: Editing the elasticsearch.conf file

```
$ sudo vi /etc/collectd.d/elasticsearch.conf
```

Now we add to the bottom of this file:

Listing 8.112: Updating the elasticsearch.conf file

```
LoadPlugin processes
<Plugin "processes">
    Process "elasticsearch"
</Plugin>
```

Note that we're monitoring the `elasticsearch` process and events from this process will pass through to Riemann. As with all of our other process monitoring, if Elasticsearch stops, we'll get a failure event and be notified.

Restarting collectd

Finally, to enable both our plugins, we restart the collectd daemon.

Listing 8.113: Restarting collectd to enable Elasticsearch monitoring

```
$ sudo service collectd restart
```

Now collectd should be scraping the Elasticsearch statistics API for metrics and watching the process for availability and performance.

Monitoring Elasticsearch with Riemann

Now we've got incoming metrics from our Logstash JVM instance and from our Logstash process. Our metrics will be tagged with `collectd`, and Riemann will be automatically sending them through to Graphite. Once they are in Graphite we can construct appropriate graphs, add them to dashboards, etc.

Inside Graphite you should now find a set of metrics that include most of the JVM-related metrics we added for Logstash, including memory and garbage collection. We also get a set of metrics for Elasticsearch including request latency and timing. We can combine these with our CPU and memory metrics for each node to produce a [definitive picture of how Elasticsearch is performing](#).

NOTE We've included all example configuration and code in the book [on GitHub](#).

Alternatives to Logstash

There are a wide variety log management solutions available, many of them boutique or designed for specific scenarios like security or network monitoring. As a result, we're only going to cover a couple of examples.

Splunk

[Splunk](#) is the major commercial log management product available. It has both on-premise and Cloud-based solutions.

Heka

[Heka](#) is an open-source log management solution. Designed by the team at Mozilla, it is a log and data collection tool written in Go. Like Logstash it provides a pluggable framework for collecting and processing logs. It also allows output to a variety of destinations, including Elasticsearch. Heka is currently [not maintained](#). There is a potential successor called [Hindsight](#).

Graylog

[Graylog](#) is another open-source log management solution. Like Logstash and Heka it provides a pluggable framework and has native collection tools for Linux and Windows.

mtail

[mtail](#) is a logfile metrics extraction program open sourced by Google, used for low overhead metrics extraction from applications that can't be linked with their varz metrics libraries. It uses small awk-like programs to describe patterns and actions to export the metrics.

Summary

In this chapter we've added a log management layer to our monitoring environment. We've done this by installing Logstash on a central host, an Elasticsearch cluster to hold our logs, and we've introduced you to the Kibana dashboard and console for visualizing events.

We've also integrated it with our Riemann solution to allow you to send notifications from Logstash events, capture metrics from your logs, or—in reverse—send events from Riemann into Logstash.

We've added support for collecting logs on your hosts and Docker containers via Syslog and by sending them to Logstash. This solution complements the host-based metrics we established in Chapters 5, 6, and 7, and we'll exercise it in subsequent chapters.

In the next chapter we're going to continue building our monitoring environment by looking at application metrics and events.

Chapter 9

Building Monitored Applications

We've been building our monitoring platform for the last few chapters. We've got our Riemann event engine and a way to visualize those events and metrics using Graphite and Grafana. We've also started to collect events and metrics to process and visualize. In the last three chapters we've gathered basic host metrics using collectd and set up the collection of logs from our hosts.

In this chapter we're going to extend our monitoring and collection to applications.

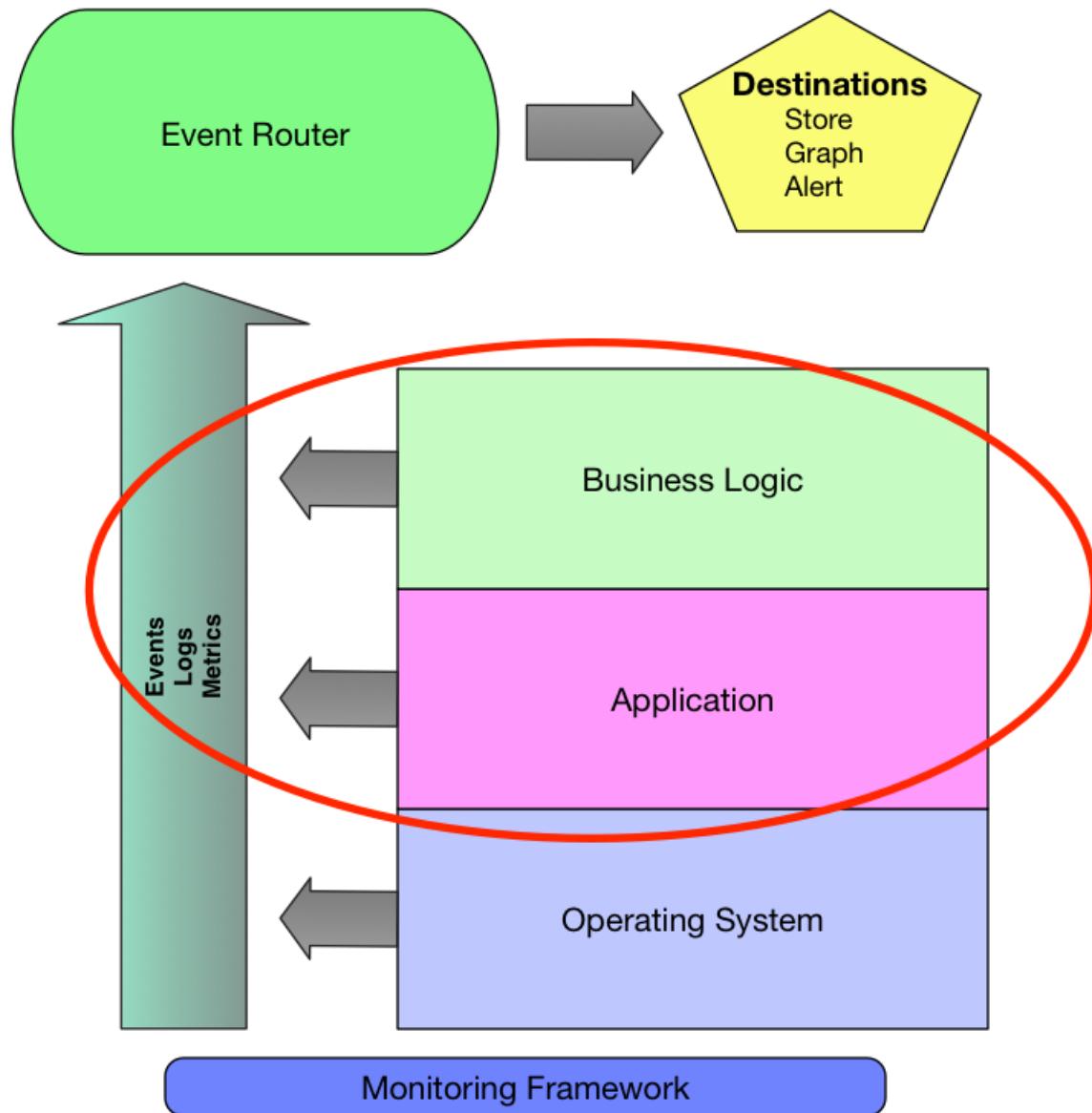


Figure 9.1: Application and business logic monitoring

We're going to focus on three approaches to monitoring applications:

- Emitting metrics by instrumenting code.
- Generating structured or semantic log events.

- Building health checks and endpoints.

We'll look at how to embed these methods in your applications and use the resulting measurements and metrics to analyze the performance of your applications.

Firstly, 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. Firstly, in any good application development methodology, it's a good idea to identify what you want to build before you build it. Monitoring is no different. Sadly there is a common anti-pattern in application development of considering monitoring and other operational functions like security as value-add components of your application rather than core features. Monitoring (and security!) are core functional features of your applications. So, 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, like tracking sales figures or the value of transactions.

A second common anti-pattern is not instrumenting enough. It's always recommended that you over-instrument your applications. One will often complain about having too little data but rarely worry about having too much.

Thirdly, if you use multiple environments—for example development, testing, staging, and production—then ensure your monitoring configuration provides

tags or identifiers so you know that the metric, log entry, or event 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 and log requests and responses, such as to specific web pages or API endpoints. If you're instrumenting an existing application then make a priority-driven list of specific pages or endpoints and instrument them in order of importance.
- Measure and log all 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 and log job scheduling, execution, and other periodic events like [cron jobs](#).
- Measure significant business and functional events, such as users being created or transactions like payments and sales.
- Measure methods and functions that read and write from databases and caches.

Instrument schemas

You should ensure that events and 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 specific event or metric is generated.

You should develop a schema for your metric names and log events (we'll talk more about structured and semantic logging later in this chapter). Let's look at an example:

Listing 9.1: An example metric name schema

```
location.application.environment.subsystem.function.actions
```

Here we've created a metric schema that should cover most of our typical applications and can easily be shortened to cater for other examples. Let's apply it to an application:

```
productiona.tornado.development.payments.collection.job.count
```

We've specified the data center where our application is located, here `productiona`. We then specify the name of the application. We've also listed the environment it is running in, e.g., `production`, `development`, and so on. Then a subsystem or component like `payments` or `user`, the function being measured, and beneath that, any methods or actions. For less complex applications we can shrink the path to suit.

Time and the observer effect

It's also important to ensure, as we discussed in Chapter 4, that the time of events is accurate, and that the time on the hosts that run your applications is accurate. You can use a service like NTP to do this as we described in Chapter 4. Also ensure that the time zone on your host is set to UTC for consistency across your hosts.

You should ensure your events have timestamps. If you create events and metrics that contain timestamps, please use standards. For example, the [ISO8601](#) standard provides dates and timestamps that are parseable by many tools.¹

Lastly, wherever possible, minimize the load on your application by logging events asynchronously. In [more than one case](#), outages have been caused or performance degraded by monitoring or metrics overloading an application. Also relevant here

¹Please don't invent your own timestamp format. Please.

is the observer effect we introduced in Chapter 2. If your monitoring consumes considerable CPU cycles or memory then it could impact the performance of your application or skew the results of any monitoring.

Now, let's look at each monitoring method in turn.

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 types of metrics:

- 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, that might be how many sales you made.

We're going to look at examples of both types of metrics in this chapter.

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 A good example of how to measure application throughput is [Brendan Gregg's USE Method](#).

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.

Monitoring patterns, or 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 and 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 where 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 also sometimes called the utility pattern—metrics logging as a 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 tools like [StatsD](#), Coda Hale's Java [Metrics](#) library, or directly in Riemann 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 pseudo Ruby-esque code to demonstrate, and we've assumed that we have already created a utility library called [Metric](#).

NOTE We'll see several functioning examples of this pattern later in this chapter.

Listing 9.2: 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}"
  Metric.increment "payment.country, #{user.country}"
  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 three metrics:

- 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.
- A `payment.country` metric — This metric records the country of origin of each payment.

Finally, we've sent an email using a second method, `send_payment_notification`, where we've incremented a fourth 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.

If your application emits logs, then identify what material they contain and see if you can use their contents to measure the behavior of the application. This is an ideal use for Logstash (which we introduced in Chapter 8) and filters like [grok](#). You can use filters, like the [metrics plugin](#), to dissect your log entries and extract data that you can map and send to Riemann (and from there to Graphite) or Elasticsearch to be indexed. 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.

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 to a statistics aggregation server called [StatsD](#).

NOTE An alternative to doing this yourself is to deploy an Application Per-

formance Management, or APM, tool like [New Relic](#) or [AppDynamics](#). These are usually plugins or add-ons to your application that collect and send application performance data from your application into a service that processes and displays it.

Introducing StatsD

StatsD is a daemon for aggregating statistics. It listens on a UDP or TCP port for incoming messages, parses them, and extracts any metrics contained in them. The original daemon was developed in NodeJS by Etsy as part of [their focus on measuring everything in their environment](#). That daemon flushed extracted metrics to Graphite directly, but later releases have pluggable back ends that allow you to send metrics to a variety of destinations.

The StatsD daemon has been rewritten numerous times in a variety of languages and has client support for a large number of languages and frameworks. Part of the reason for its popularity is the line protocol that it uses:

Listing 9.3: The StatsD protocol

```
<metricname>:<value>|<type>
```

A typical metric looks like:

Listing 9.4: A typical StatsD metric

```
productiona.tornado.production.payments.amount:1|c
```

Like Graphite (and because it was originally intended to output directly to

Graphite), StatsD uses periods `.` to divide metric namespaces or “buckets.” So our metric would be called:

```
productiona.tornado.production.payments.amount
```

After the `:` we specify the value of our metric, here `1`, and then, separated by a `|`, the type of metric. In this case, that’s `c` for counter. We could also specify `g` for gauge or a [variety of other metric types](#) including timers.

StatsD receives metrics and periodically aggregates them, then flushes them, by default every 10 seconds, to a back end. The exact aggregation behavior at flush depends on the type of metric. So let’s look at some of the available metric types.

Counters

We’ve already seen a counter.

Listing 9.5: A StatsD counter

```
productiona.tornado.production.payments:1|c
```

When this is triggered it would add `1` to the `productiona.tornado.production.payments` counter. In most StatsD implementations the client will send both the count and the rate. StatsD iterates over any counters it receives and increments them until flushed. So if you send seven increments of the `productiona.tornado.production.payments` counter, when StatsD flushes, generally every 10 seconds, it’ll set the counter to `7` and calculate a rate of `0.7` increments per second.

Timers

This is a StatsD timer. Timers can be units of time, bytes, numbers of items, or any other number collected.

Listing 9.6: A StatsD timer

```
productiona.tornado.production.payments.job.collection:10|ms|@0.1
```

You can see a metric name and the time (here in `ms` or milliseconds) of `10`. The last option is a sample rate. You can apply sampling to StatsD metrics for frequent events that you are worried will overload a server.. Here `0.1` is a sample rate of $1/10$. So 10% of events are sampled and StatsD will multiply the sampled data to give an estimate of 100% of the sample rate.

StatsD will also automatically calculate percentiles (you can configure what percentile it'll calculate but usually it defaults to `90`), average/mean, standard deviation, sum, and the lower and upper bounds of any timing data. It'll suffix the type of calculation to a new metric event, generally like so:

Listing 9.7: StatsD timer calculations

```
productiona.tornado.production.payments.job.collection-average  
productiona.tornado.production.payments.job.collection-upper  
productiona.tornado.production.payments.job.collection-lower  
productiona.tornado.production.payments.job.collection-sum  
productiona.tornado.production.payments.job.collection-count  
...
```

So if we were to send the following values to StatsD over the flush interval `112`, `133`, and `156`, then StatsD would calculate something like the following values:

Listing 9.8: StatsD timer calculations

```
productiona.tornado.production.payments.job.collection-average  
    133.66666666667  
productiona.tornado.production.payments.job.collection-upper 156  
productiona.tornado.production.payments.job.collection-lower 112  
productiona.tornado.production.payments.job.collection-sum 401  
productiona.tornado.production.payments.job.collection-count 3  
    . . .
```

If the client is configured to do so, then it will also calculate percentiles for each metric, resulting in metrics like:

Listing 9.9: StatsD timer percentile calculations

```
productiona.tornado.production.payments.job.collection_99
```

This would be the 99th percentile.

Gauges

We can also use StatsD to send gauge metrics, an arbitrary value at a point in time.

Listing 9.10: A StatsD gauge

```
productiona.tornado.production.payments.job.active_jobs:232|g
```

Here the metric is a gauge, **g**, and when flushed it'll send **232** as the value of the

metric. If StatsD receives multiple gauges during a flush interval it'll only send the last value. For example, if StatsD received 143, 567, and 232 as values of the: `productiona.tornado.production.payments.job.active_jobs`

gauge, then only the last value, 232, would be flushed to the back end. If the gauge isn't updated at the next flush it'll send the previous value—or you can configure StatsD to delete the value instead.

We can also change gauge values by prefixing a sign.

Listing 9.11: Changing StatsD gauge values

```
productiona.tornado.production.payments.job.active_jobs:232|g  
productiona.tornado.production.payments.job.active_jobs:-15|g  
productiona.tornado.production.payments.job.active_jobs:+4|g
```

So if our initial gauge is 232 then we'd subtract 15 from it and add 4 for a final value of 221.

Sets

StatsD also supports sets. (However this is a recent addition to StatsD—not all StatsD clients support them yet.) Sets allow you to track the number of unique elements belonging to a group, such as the number of unique users on a site.

Listing 9.12: A StatsD set example

```
productiona.tornado.production.payments.userid:3567|s
```

When flushed StatsD will push the number of unique elements in the set as a gauge value.

Our application collection architecture

Now that we've got an idea of how StatsD works, we'll use our previously installed collectd infrastructure to provide a StatsD server via a plugin we'll enable: `statsd`. The collectd `statsd` plugin provides a StatsD server and sends events to the `write_riemann` plugin, which in turn sends the events to Riemann. This means our events will flow directly from collectd to Riemann, and we'll see how we can make use of them when they arrive there. Then we'll see what to do with them when they are added to Graphite and made available to Grafana.

This diagram shows our application collection architecture.

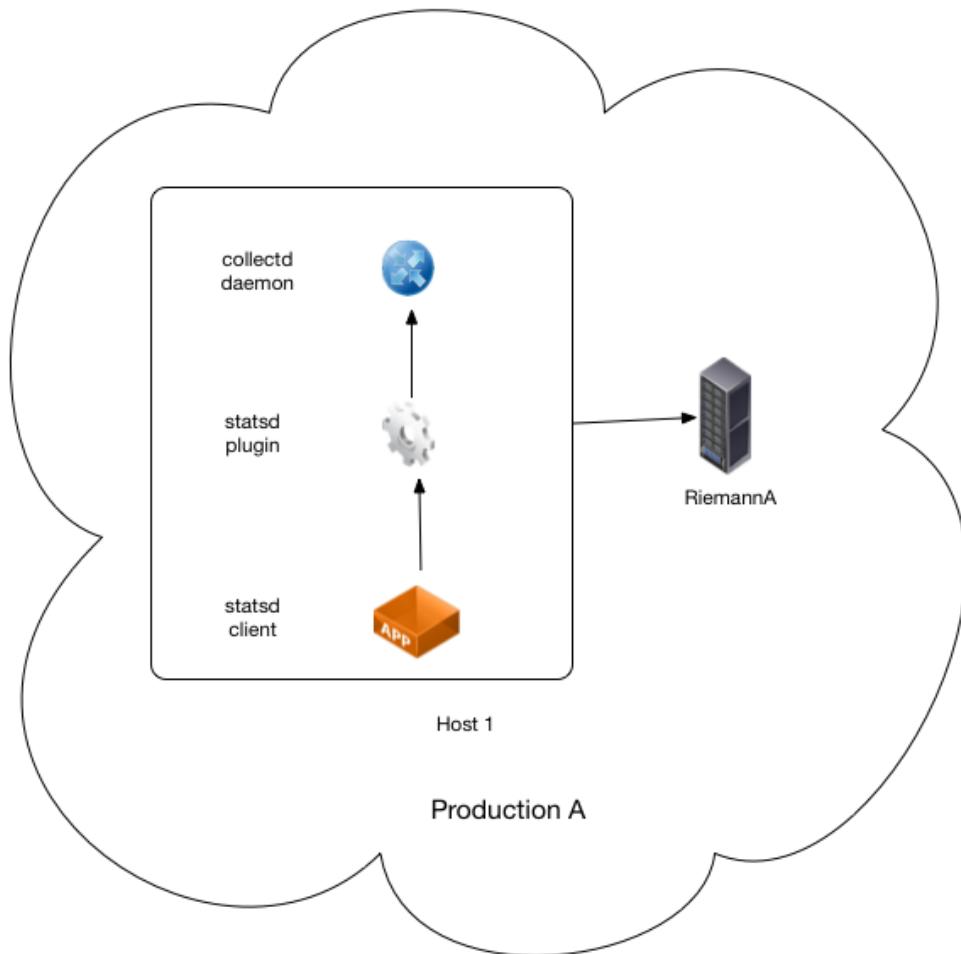


Figure 9.2: Our application monitoring architecture

Our application generates metrics and acts as a StatsD client. We'll configure it using the utility pattern suggested above.

Setting up the statsd plugin inside collectd

Let's start by setting up the **statsd** plugin in our collectd configuration. We create a new collectd configuration file, let's call it **statsd.conf**, in the **/etc/collectd.d/** directory and populate it.

Listing 9.13: The statsd.conf configuration file

```
LoadPlugin statsd

<Plugin statsd>
    Host "localhost"
    Port "8125"
    TimerPercentile 90
    TimerPercentile 99
    TimerLower true
    TimerUpper true
    TimerSum true
    TimerCount true
</Plugin>
```

We first load the **statsd** plugin. We then configure the plugin and set the **Host** and **Port** to which we're binding the StatsD server. Here we're binding it to the **localhost** interface on port **8125**.

We also configure a number of options that control StatsD's behavior. We mentioned earlier in the chapter that when using StatsD timer metrics we can configure a percentile calculation (amongst other calculations) of the timer value. Here we've specified the **TimerPercentile** option twice to configure two percentile values: the 90th and 99th percentiles. We've also used the **TimerLower**, **TimerUpper**, **TimerSum**, and **TimerCount** options to tell the **statsd** plugin to calculate the lower

and upper bounds as well as the sum and count.

We can also control the behavior of metrics over flush intervals. The default `statsd` plugin behavior is to continue to send metrics with default values. For example, the rate of counters and size of sets will be zero, timers will report `Nan`, and gauges will be unchanged. We could use the `DeleteCounters`, `DeleteTimers`, `DeleteGauges`, and `DeleteSets` boolean options to change this behavior to delete unchanged metrics. The default for all of these is `false`, meaning metrics are sent with their default behavior.

It's also important to note that the flush interval for the `statsd` plugin will be the check interval for collectd. In our case we set this to two seconds back in Chapter 5.

Tagging our application events

Our current collectd events are all tagged with `collectd` in the `:tags` field of Riemann events. This tag is added by the `Tag` directive in the `write_riemann` plugin's configuration.

Listing 9.14: Revisiting the `write_riemann` plugin

```
LoadPlugin write_riemann
<Plugin "write_riemann">
    <Node "riemann">
        Host "riemann.example.com"
        Port "5555"
        Protocol TCP
        CheckThresholds true
        StoreRates false
        TTLFactor 30.0
    </Node>
    Tag "collectd"
</Plugin>
```

Let's add a new tag to this configuration that will identify that this host and its services are related to a new application called `aom-rails`. We do this by adding a new `Tag` directive to our configuration.

NOTE We'll see more about this new application shortly.

Listing 9.15: Revisiting the write_riemann plugin

```
LoadPlugin write_riemann
<Plugin "write_riemann">
    .
    .
    Tag "collectd"
    Tag "aom-rails"
</Plugin>
```

TIP Another example of why managing collectd with a configuration management tool will make life a lot easier for you.

Our `:tags` field in Riemann will now be a vector containing:

```
:tags [collectd aom-rails]
```

We can use this new tag to better filter our events.

Let's restart collectd to enable the plugin and our new tags.

Listing 9.16: Restarting collectd to enable statsd

```
$ sudo service collectd restart
```

We now test that the StatsD server is working by sending it some sample metrics. We do this via the command line using the `nc` binary. The `nc`, or netcat, binary allows us to interact with arbitrary TCP and UDP services. Here we're going to send a raw StatsD metric string to our local StatsD server and then see if the

corresponding event appears in Riemann.

Listing 9.17: Testing the statsd collectd client

```
$ echo "foo:1|c" | nc -u -w0 localhost 8125
```

This will echo a counter called `foo` with a metric of `1` (remember `c` means counter) to our `statsd` collectd plugin on `localhost` at port `8125`. This metric will be processed by collectd and sent onto Riemann. If we look in Riemann we should see an event like:

Listing 9.18: Our first statsd event in Riemann

```
{:host tornado-web1, :service statsd/derive-foo, :state ok, :description nil, :metric 1, :tags [collectd aom-rails], :time 1446555358, :ttl 60.0, :ds_index 0, :ds_name value, :ds_type derive, :type_instance foo, :type derive, :plugin statsd}
```

Let's break our event down by field. We get a host, here `tornado-web1`, and a service, `statsd/derive-foo`. The `statsd/derive-` prefix comes from the `statsd` plugin. The `:type_instance` field contains the plain name of the metric, `foo`. We also get the `:plugin` field that contains the name of our `statsd` plugin. All of these are useful fields to filter in order to grab specific metrics. We can see that our `:tags` field has been updated to contain both `collectd` and `aom-rails`. Lastly, we have the `:metric` field which contains our current count, `1`. We can use this field in checks or send it to be graphed.

The collectd daemon will continue to emit each metric until it is restarted unless we use the `DeleteCounters` option. The metric value will remain unchanged unless no data is received for that metric. So, if timing a method and that method

wasn't executed, the `:metric` field will change to a value of `NaN` upon the next flush.

Now let's see StatsD monitoring an actual application.

Our sample Rails application

We've created a sample Rails application using [Rails Composer](#). We're going to call it `aom-rails` or The Art of Monitoring Rails application. The `aom-rails` application allows us to create and delete users and sign in to the application.

NOTE You can find the `aom-rails` application [on GitHub](#).

Say we want to get in early and instrument our application. We'll first need to add support for StatsD using [a Ruby-based client](#). The `statsd-ruby` gem allows us to create a StatsD client inside our application. We're going to do that initially by adding the `statsd-ruby` gem to our Rails application's [Gemfile](#).

Listing 9.19: The `aom-rails` Gemfile

```
source 'https://rubygems.org'  
ruby '2.2.2'  
gem 'rails', '4.2.4'  
.  
.  
.  
gem 'statsd-ruby'  
.
```

We then install the new gem using the `bundle` command.

Listing 9.20: Install statsd-ruby 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 statsd-ruby 1.2.1
.
.
```

We then test the **statsd-ruby** client using a Rails console. Let's launch one now using the **rails c** command.

Listing 9.21: Testing statsd-ruby with the Rails console

```
$ rails c
Loading development environment (Rails 4.2.4)
[1] pry(main)> $statsd = Statsd.new 'localhost', 8125
```

We've launched a Rails console and created a StatsD client using the code:

Listing 9.22: Our first Ruby-based StatsD client

```
$statsd = Statsd.new 'localhost', 8125
#<Statsd:0x007f4371f39dd0 @batch_size=10, @host="localhost",
@port=9125, @postfix=nil, @prefix=nil>
```

We've created a new variable called **\$statsd** which calls an instance of **Statsd**

and passes in two variables, `localhost` and `8125`, the host and port of the target StatsD server respectively. In our case this is the StatsD server provided locally by the collectd `statsd` plugin.

TIP If you use a DNS name for the host in your StatsD client then you should use a local DNS caching service like `nscd` to ensure DNS resolution doesn't impact the performance of your StatsD client. In our case we're connecting to our local `statsd` plugin from collectd.

We'll use this `$statsd` variable to send metrics to the StatsD server. Let's look at some examples.

There are three major methods we can use to create metrics:

Listing 9.23: The basic statsd-ruby methods

```
$statsd.increment 'app.counter'  
$statsd.timing 'app.timer', 30  
$statsd.gauge 'app.gauge', 100
```

The `increment` method increments a counter, here `app.counter`. The `timing` method creates a timer; in this case we're saying the `app.timer` took 30 seconds. The last method, `gauge`, creates a gauge with a value of `100`.

We're not going to manually create a client every time we want to log any metrics, so let's set up some utility code to do this for us. In the Ruby on Rails world, we'd generally add an initializer to create our StatsD client when Rails starts. Let's do that now by creating a `statsd.rb` file in the `config/initializers` directory containing:

Listing 9.24: The statsd-ruby configuration initializer

```
STATSD = Statsd.new("localhost", 8125)
STATSD.namespace = "aom-rails.#{Rails.env}"
```

Next we define a constant called `STATSD` by calling a new instance of `Statsd`. It uses the value of `localhost` for the host and `8125` as the port number. This will connect to our local `statsd` plugin running in collectd.

Then we set a namespace. The namespace sets a global prefix for any StatsD metrics. It allows us to identify what application our metric comes from. We've added a wrinkle and are using the `#{Rails.env}` method, which returns the Rails environment in which the application is running. In the Rails world, the “environment” allows Rails applications to be configured differently for production, development, etc. This means we can ensure metrics are coming from the production variant of the application rather than any other environment.

We could even configure StatsD to report somewhere else for non-production instances, for example:

Listing 9.25: The statsd-ruby configuration initializer

```
if %w(staging production).include?(Rails.env)
  STATSD = Statsd.new("localhost", 8125)
else
  STATSD = Statsd.new("statsd-dev.example.com", 8125)
end
STATSD.namespace = "aom-rails.#{Rails.env}"
```

Here we're specifying that if the `Rails.env` method returns either `production`

or `staging` then we use the local collectd plugin at `localhost`, or else we use a new host, `statsd-dev.example.com`, for reporting of metrics for any other environment.

So what do our metrics look like with a namespace? Let's create a new counter and take a look.

Listing 9.26: The statsd-ruby namespace method

```
STATSD.increment 'app.counter'
```

The namespace is prefixed in front of our metric name. So with a `namespace` of `aom-rails.#{Rails.env}`—assuming we're running in the `production` environment—and a metric name of `app.counter`, our new metric name would then become:

`aom-rails.production.app.counter`.

We then use the `STATSD` constant inside our code. Let's look at how we might add metrics. We'll start with a counter that increments when users are deleted.

Listing 9.27: Counter for aom-rails user deletions

```
def destroy
  user = User.find(params[:id])
  user.destroy
  STATSD.increment "user.deleted"
  redirect_to users_path, :notice => "User deleted."
end
```

This would create a counter called:

`aom-rails.production.user.deleted`

That counter would be incremented every time a user is deleted. Now, when a user is deleted, a counter will be incremented. The counter then travels through the `statsd` plugin in collectd, into the `write_riemann` plugin, and from there to our Riemann server. On the Riemann server we then see the new event:

Listing 9.28: The `aom-rails.production.user.deleted` event in Riemann

```
{:host tornado-web1, :service statsd/derive-aom-rails.production.  
user.deleted, :state ok, :description nil, :metric 1, :tags [  
collectd aom-rails], :time 1449374333, :ttl 60.0, :ds_index 0,  
:ds_name value, :ds_type derive, :type_instance aom-rails.  
production.user.deleted, :type derive, :plugin statsd}
```

We see that the event has a `:service` field of:

`statsd/derive-aom-rails.production.user.deleted`

As we learned earlier in the chapter, the `statsd/` prefix has been added by the `statsd` plugin in collectd. Remember, we can see the non-prefixed metric name in the `:type_instance` field. We also get the host `tornado-web1` that generated the event. Finally, in the `:metric` field we see the actual counter value, currently `1` for one user deleted since the counter started recording.

We could also create another counter for created users by adding it to the `User` model.

Listing 9.29: Counter for aom-rails user creation

```
class User < ActiveRecord::Base
  enum role: [:user, :vip, :admin]
  after_initialize :set_default_role, :if => :new_record?

  after_create do
    STATSD.increment "user.created"
  end
  . . .
end
```

Here we've used an ActiveRecord callback, `after_create`, to increment a counter, `aom-rails.production.user.created`, when a new user is created. This will generate an event much like our `aom-rails.production.user.deleted` counter.

Listing 9.30: The aom-rails.production.user.created event in Riemann

```
{:host tornado-web1, :service statsd/derive-aom-rails.production.
  user.created, :state ok, :description nil, :metric 1, :tags [
    collectd aom-rails], :time 1449386123, :ttl 60.0, :ds_index 0,
  :ds_name value, :ds_type derive, :type_instance aom-rails.
  production.user.created, :type derive, :plugin statsd}
```

We can also time specific events, functions, and methods, either by using the `timer` method or by wrapping the action in a block. Here we're instrumenting the `show` method to time how long it takes to find a user.

Listing 9.31: Time the User.find action

```
def show
  STATSD.time("user.find") do
    @user = User.find(params[:id])
  end
  unless current_user.admin?
    unless @user == current_user
      redirect_to :back, :alert => "Access denied."
    end
  end
end
```

This will create a timer event every time the `show` method is executed. When it reaches Riemann we'll see an event like:

Listing 9.32: The user.find timer in Riemann

```
{:host tornado-web1, :service statsd/latency-aom-rails.production
  .user.find-average, :state ok, :description nil, :metric
  0.003999999724328518, :tags [collectd aom-rails], :time
  1446653797, :ttl 60.0, :ds_index 0, :ds_name value, :ds_type
  gauge, :type_instance aom-rails.production.user.find-average, :
  type latency, :plugin statsd}
```

The collectd `statsd` plugin has generated a new event with a `:service` field of `statsd/latency-aom-rails.production.user.find-average`.

The `statsd/latency` prefix is added to denote the type of metric, and the `-average`

suffix indicates that it's the average time that the method took. Since we've configured the `statsd` plugin to produce a variety of other timing metrics we'd also get events for:

- 90th and 99th percentile.
- Upper and lower bounds.
- Sum.
- Count.

Here's the 99th percentile event.

Listing 9.33: The `find.user` 99th percentile

```
{:host tornado-web1, :service statsd/latency-aom-rails.production
  .user.find-percentile-99, :state ok, :description nil, :metric
  0.005849609151482582, :tags [collectd aom-rails], :time
  1449377543, :ttl 60.0, :ds_index 0, :ds_name value, :ds_type
  gauge, :type_instance aom-rails.production.user.find-percentile
  -99, :type latency, :plugin statsd}
```

NOTE You'll have a wide variety of applications you may 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 operational.

Using our metrics

Now our application has metrics being generated and we can make use of them in Riemann. There's a couple of things we could do here with events from our `aom-rails` events. The first is that we can detect if any events don't appear.

Listing 9.34: Detecting missing aom-rails events

```
(where (service #"^statsd\/.*-aom-rails.production")
  (expired
    (email "james@example.com")))
```

Here we've added some code that matches any incoming `aom-rails` metrics coming only from the `production` environment, identifies if any metric events are expired (i.e., haven't appeared in the index for 60 seconds), and then sends `james@example.com` an email notification.

We could also target a specific metric. Here we'll do that to check how our application is performing. Let's look at our `aom-rails.production.user.find-average` timer and create a check from it.

Listing 9.35: Checking the user.find timer

```
(where (and (service "statsd/derive-aom-rails.production.user.
  find-average") (> metric 0.5))
  (email "james@example.com")))
```

We've used a `where` stream to match on the average timer. If the metric exceeds `0.5` seconds then we'd send an email.

Finally, we want to send our metrics through to Graphite. Unlike previous met-

rics, though, these are application-centric rather than host-centric. Our typical Graphite host metrics have been written to a path like:

```
productiona.hosts.hostname.metric.name
```

Our new metrics are related to our `aom-rails` application, so it's preferable to bundle them together as an application rather than scatter them amongst the hosts that might make up the application. We'll write them to Graphite as:

```
productiona.aom-rails.hostname.metric.name
```

To do this we're going to revisit rewriting our Graphite paths. We need to make two changes to our configuration:

1. Rewrite the `:service` field to strip out the `statsd/` prefix from our metrics using our existing service rewriting functions.
2. Rewrite the Graphite metric name to put the application name first, again using previous functions we created.

Let's start with rewriting our `:service` field by editing the:

```
/etc/riemann/examplecom/etc/collectd.clj
```

File on our Riemann host and adding a new service rewrite to the `default-services` var.

Listing 9.36: Rewriting our statsd rules

```
(def default-services
  [{:service #"^load/load/(.*)$" :rewrite "load $1"}
   ...
   {:service #"^statsd\/(gauge|derive|latency)-(.*)$" :rewrite "
$2"}])
```

This will change events from the `statsd` plugin from:

`statsd/derive-aom-rails.production.user.find-average`

to:

`aom-rails.production.user.find-average`

This strips away the `statsd/` prefix and removes the metric type identifier as well.

Next we want to take this updated event and ensure its metric name is rewritten when we send it to Graphite. To do this we need to edit the `add-environment-to-graphite` function we created in Chapter 4 and updated in Chapter 7 when we added our Docker metrics. This is inside the `/etc/riemann/examplecom/etc/graphite.clj` file on our Riemann host.

Listing 9.37: Updated graphite.clj

```
(ns examplecom/etc/graphite
  (:require [clojure.string :as str]
            [riemann.config :refer :all]
            [riemann.graphite :refer :all]))


(defn graphite-path-statsd [event]
  (let [host (:host event)
        app (re-find #".*\?.\." (:service event))
        service (str/replace-first (:service event) #".*\?.\." ""))
        split-host (if host (str/split host #"\.") [])
        split-service (if service (str/split service #" ") [])]
    (str app, (str/join "." (concat (reverse split-host) split-
                                     service)))))

(defn add-environment-to-graphite [event]
  (condp = (:plugin event)
    "docker"
    (if (:com.example.application event)
        (str "productiona.docker.", (:com.example.application
                                     event), ".", (riemann.graphite/graphite-path-
                                     percentiles event))
        (str "productiona.docker.", (riemann.graphite/graphite-
                                     path-percentiles event)))
    "statsd" (str "productiona.", (graphite-path-statsd event))
    (str "productiona.hosts.", (riemann.graphite/graphite-path-
                                percentiles event))))


(def graph (async-queue! :graphite {:queue-size 1000}
  (graphite {:host "graphitea" :path add-environment-
Version: v1.0.3 (2c4a7d0) to-graphite})))
```

Here we've updated the `add-environment-to-graphite` function to select a metric name rewrite depending on the source of the event. We've used a `condp` function to select the event based on the contents of the `:plugin` field. The first condition controls metric name writes for our Docker events. The second matches on the `statsd` plugin's events and writes their path using the `graphite-path-statsd` function. Our last clause catches all other matches and uses the standard path rewrite we established for all other events with the `productiona.hosts.` prefix.

Let's look at the `graphite-path-statsd` function.

Listing 9.38: The `graphite-path-statsd` function

```
(defn graphite-path-statsd [event]
  (let [host (:host event)
        app (re-find #".*?\." (:service event))
        service (str/replace-first (:service event) #".*?\." ""))
        split-host (if host (str/split host #"\.") [])
        split-service (if service (str/split service #" ") []))
    (str app, (str/join "." (concat (reverse split-host) split-
        service)))))
```

The function takes an `event` as an argument and opens with a `let` instruction. Our `let` instruction sets several variables:

- `host` — From the `:host` field of the event.
- `app` — This pulls the application name from the `:service` field.
- `service` — The service itself, minus the application name.
- `split-host` — Handles fully-qualified host and domain names.
- `split-service` — Splits services on any spaces so we can rewrite those with periods in the next line.

The core of the function takes the variables we've created and combines them. We create a string for the path starting with the application name, the host name and potentially fully-qualified domain, and then the service, joined with periods.

If we go back to our rewritten event:

```
aom-rails.production.user.find-average
```

Then it'll get written into Graphite as:

```
production.aom-rails.tornado-web1.production.user.find-average
```

This groups all of our events together in Graphite by application name, then by host name, then environment, then service. You can obviously adjust this schema to suit your needs, perhaps writing them by application and then environment—`production`, etc.— rather than by host.

Logging

In addition to embedding metrics we can also add support for logging and log events to our applications. Metrics are useful for telling us the performance or keeping track of various pieces of state, but logs are often far more expressive. With logs we provide additional context or information about a situation, or highlight that something has occurred. An example of this is a stack trace generated when an error occurs. For diagnostic purposes log entries are hugely useful. So, to supplement our existing instrumentation, there are two ways of deducing application status and performance from logs:

- Creating structured log entries at strategic points of our application, in line with what we discussed in our application monitoring primer.
- Consuming existing log data, as we discussed in the external pattern, also introduced earlier in this chapter.

We're going to look at both methods, starting with adding our own log entries.

Adding our own structured log entries

Most logging mechanisms emit log entries that contain a string value and the message or description of the error. The classic example of this is Syslog, used by many hosts, services, and applications as a default logging format. A typical Syslog message looks like:

Listing 9.39: A typical syslog message

```
Dec 6 23:17:01 logstash CRON[5849]: (root) CMD (cd / && run-parts --report /etc/cron.hourly)
```

In addition to the payload, in this case a report on a Cron job, it has a timestamp and a source (the host `logstash`). While versatile and readable, the Syslog format is not ideal—it's basically one long string. This string is awesome from a human readability perspective—it's easy to glance at a Syslog string and know what's happened. But are we the target audience of a string-based message? Probably back in the day when we had a small volume of hosts and we were connecting to them to read the logs. Now we have a pool of hosts, services, and applications, and our log entries are centralized. That means there is now a machine that consumes the log message before we, the human audience, see it. And because of the eminently readable string format, that consumption is not easy.

That format means we're likely to be forced, as we saw in Chapter 8, to resort to regular expressions to parse it. In fact, probably more than one regular expression. Again Syslog is a good example. Implementations across platforms are sometimes subtly different, and this often means more than one regular expression needs to be implemented and then maintained. We also saw this in Chapter 8 when we added our Docker logs to our existing Syslog implementation. The additional overhead means it's much harder to extract the value—diagnostic or operational—from our log data.

There is, however, a better way of generating logs: structured logs (also known as semantic or typed logs). There's currently no standard for structured logging. There have been a few attempts to create one but nothing has yet gained traction. Still, we can describe the concept of structured logging. Instead of a string like our Syslog examples, structured logs try to preserve typed rich data rather than convert it. Let's look at an example of some code that produces an unstructured string:

NOTE There are some examples of attempts to formalize a structured logging format such as the [Common Event Expression](#) and [Project Lumberjack](#). None of them got much traction and are largely unmaintained.

Listing 9.40: Unstructured log message example

```
Logger.error("The system had a hiccup trying to create user" +  
    username)
```

Let's assume the user being created was `james@example.com`. This pseudo-code would generate a message like: `The system had a hiccup trying to create user james@example.com`. We'd then have to send that message somewhere, to Logstash for example, and then parse it into a useful form.

Alternatively, we can create a more structured message.

Listing 9.41: Structured log message example

```
Logger.error("user_creation_failed", user=username)
```

Note that in our structured message we've gotten a head start on any parsing. Assuming we send the log message in some encoded format, JSON for example or a binary format like protocol buffers, then we get an event name, `user_creation_failed`, and a variable, `user`, which contains the username of the user that we failed to create, or even a user object containing all the parameters of the user being created.

Let's look at what our JSON encoded event might look like:

Listing 9.42: JSON encoded event

```
[  
  {  
    "time": 1449454008,  
    "priority": "error",  
    "event": "user_creation_failed",  
    "user": "james@example.com"  
  }  
]
```

Instead of a string we've got a JSON array containing a structured log entry: a time, a priority, an event identifier, and some rich data from that event: the user that our application failed to create. We're logging a series of objects that are now easily consumed by a machine rather than a string we need to parse.

Adding structured logging to our sample application

Let's see how we might extend our sample application with some structured log events. Remember, it's a Ruby on Rails application that allows us to create and delete users and not much else. We're going to add two structured logging libraries—the first called [Lograge](#), and the second called [Logstash-logger](#)—to our application. The Lograge library formats Rails-style request logs into a structured format, by default JSON, but can also generate Logstash-structured events. The second library, Logstash-logger, allows us to hijack Rails' existing logging framework, emit much more structured events, then send them directly to Logstash. Let's install these now and see what some structured logging messages might look like.

We first need to add three gems, [lograge](#), [logstash-event](#), and [logstash-logger](#), to our application to enable our structured logging support.

The [lograge](#) gem enables Lograge's request log reformatting. The [logstash-event](#) gem allows Lograge to format requests into Logstash events. The [logstash-logger](#) gem allows you to output log events in Logstash's event format and enables a variety of potential logging destinations, including Logstash. We're going to start by adding the gems to our Rails application's [Gemfile](#).

Listing 9.43: Adding our logging gems to the aom-rails Gemfile

```
source 'https://rubygems.org'  
ruby '2.2.2'  
gem 'rails', '4.2.4'  
.  
.  
.  
gem 'lograge'  
gem 'logstash-event'  
gem 'logstash-logger'  
..
```

We then install the new gems using the `bundle` command.

Listing 9.44: Install the gems 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 lograge  
Installing logstash-event  
Installing logstash-logger  
. .
```

Next we need to enable all of our new logging components inside our Rails application's configuration. We're only going to enable each component for the `production` environment. To do this we add our configuration to the `config/`

environments/production.rb file.

Listing 9.45: Adding logging to the Rails production environment

```
Rails.application.configure do
  # Settings specified here will take precedence over those in
  # config/application.rb.

  ...
  config.log_level = :info
  config.lograge.enabled = true
  config.lograge.formatter = Lograge::Formatters::Logstash.new
  config.logger = LogStashLogger.new(type: :tcp, host: 'logstash.
    example.com', port: 2020)
end
```

Here we've configured four options. The first, `config.log_level`, is a Rails logging default for the log level. Here we're telling Rails to only log events of an `:info` level or higher; by default, Rails logs at a `:debug` level. The second option, `config.lograge.enabled`, turns on Lograge, taking over Rails' default logging for requests. The third option, `config.lograge.formatter`, controls the format in which those log events are emitted. Here we're using Logstash's event format. Lograge has [a series of other formats](#) available, including raw JSON. The last option, `config.logger`, takes over Rails' default logging with Logstash-logger. It creates a new instance of the `LogStashLogger` class that connects to our Logstash server, `logstash.example.com`, via TCP on port `2020`.

Let's look at the corresponding required configuration on our Logstash server. We need to add a new `tcp` input to receive our application events.

Listing 9.46: Adding a new TCP input to Logstash

```
input {
  tcp {
    port => 5514
    type => "syslog"
  }
  tcp {
    port => 2020
    type => "apps"
    codec => "json"
  }
  ...
}
```

We've added a new `tcp` input running on port `2020`. We set a `type` of `apps` for any events received on this input, and we use the `json` codec to parse any incoming events into Logstash's message format from JSON. To enable our configuration we would need to restart Logstash.

Listing 9.47: Restarting Logstash for our Application events

```
$ sudo service logstash restart
```

So what does this do for our sample application? Enabling Lograge will convert Rails' default request logs into something a lot more structured and a lot more useful. A traditional request log might look like:

Listing 9.48: Traditional Rails request logging

```
Started GET "/" for 127.0.0.1 at 2015-12-10 09:21:45 +0400
Processing by UsersController#index as HTML
  Rendered users/_user.html.erb (6.0ms)
Completed 200 OK in 79ms (Views: 78.8ms | ActiveRecord: 0.0ms)
```

With Lograge enabled the log request would appear more like:

Listing 9.49: A Lograge request log event

```
{
  "method": "GET",
  "path": "/users",
  "format": "html",
  "controller": "users",
  "action": "index",
  "status": 200,
  "duration": 189.35,
  "view": 186.35,
  "db": 0.92,
  "@timestamp": "2015-12-11T13:35:47.062+00:00",
  "@version": "1",
  "message": "[200] GET /users (users#index)",
  "severity": "INFO",
  "host": "tornado-web1",
  "type": "apps"
}
```

We see that the log event has been converted into a Logstash event. The original base message is now in the `message` field and each element of the request has been parsed into a field—for example, note that the request’s method is in the `method` field and the controller is in the `controller` field. Logstash-logger will send this structured event to our Logstash server where we can parse it, create metrics from it (we now have things like the HTTP status code and timings from the request), and store it in Elasticsearch where we can query it.

We can also send stand-alone log events using Logstash-logger’s override of Rails’ default `logger` method. Let’s specify a message that gets sent when we delete a user.

Listing 9.50: Logging deleted users

```
def destroy
  STATSD.time("find.user") do
    @user = User.find(params[:id])
  end
  @user.destroy
  STATSD.increment "user.deleted"
  logger.info message: 'user_deleted', user: @user
  redirect_to users_path, :notice => "User deleted."
end
```

Here we’ve added a `logger.info` call to the `destroy` method. We’ve passed it two arguments, `message` and `user`. The `message` argument will become the value of our `message` field in the Logstash event. The `user` field will also become a field containing the `@user` instance variable, which in turn contains the details of the user being deleted. Let’s look at an event that might be generated when we delete the user `james`.

Listing 9.51: A Logstash formatted event for a user deletion

```
{  
  "message": "user_deleted",  
  "@version": "1",  
  "severity": "INFO",  
  "host": "tornado-web1",  
  "type": "apps",  
  "user": {  
    "id": 6,  
    "email": "james@example.com",  
    "created_at": "2015-12-11T04:31:46.828Z",  
    "updated_at": "2015-12-11T04:32:18.340Z",  
    "name": "james",  
    "role": "user",  
    "invitation_token": null,  
    "invitation_created_at": null,  
    "invitation_sent_at": null,  
    "invitation_accepted_at": null,  
    "invitation_limit": null,  
    "invited_by_id": null,  
    "invited_by_type": null,  
    "invitations_count": 0  
  },  
  "@timestamp": "2015-12-11T13:35:50.070+00:00",  
  "severity": "INFO",  
  "host": "tornado-web1",  
  "type": "apps",  
  "user": {  
    "id": 6,  
    "email": "james@example.com",  
    "created_at": "2015-12-11T04:31:46.828Z",  
    "updated_at": "2015-12-11T04:32:18.340Z",  
    "name": "james",  
    "role": "user",  
    "invitation_token": null,  
    "invitation_created_at": null,  
    "invitation_sent_at": null,  
    "invitation_accepted_at": null,  
    "invitation_limit": null,  
    "invited_by_id": null,  
    "invited_by_type": null,  
    "invitations_count": 0  
  }  
}
```

We see our event is in Logstash format with our `user_deleted` message and the contents of the `@user` instance variable structured as fields of a `user` hash. When

a user is deleted this event will be passed to Logstash and could then be processed and stored or even sent onto Riemann. There are more than enough details to help us diagnose issues and track events.

TIP You can see some more usage examples for generating log events with Logstash-logger in [the Github documentation](#).

This is an example of how structured logging can make monitoring applications so much easier. The basic principles articulated here can be applied in a variety of languages and frameworks.

Structured logging libraries

Just to get you started, here are some structured logging libraries and integrations for a variety of languages and frameworks. You should be able to find others by searching online.

Java

The Java community has the powerful and venerable [Log4j](#). It's hugely configurable and flexible.

Go

Golang has [Logrus](#), which extends the standard logger library with structured data.

Clojure

Clojure has a couple of good structured logging implementations, one from [Puppet Labs](#) and the other [clj-log](#).

Ruby and Rails

We've already seen [Lograge](#) for Ruby and Rails. Other examples include [Semantic Logger](#) and [ruby-cabin](#).

Python

Python has [Structlog](#), which augments the existing Logger methods with structured data.

Javascript and Node.JS

Javascript (and Node) has an implementation of .Net's Serilog called [Structured Log](#). Another example is [Bunyan](#).

.Net

The .Net framework has [Serilog](#).

PHP

PHP has [Monolog](#).

Perl

Perl has a Log4j-esque clone called [Log4perl](#).

Working with your existing logs

Sometimes we aren't able to rewrite our application to make use of structured logging techniques. In these cases we have to work with the existing logs our application is generating. Much like the Syslog parsing we did in Chapter 8, we can make use of Logstash's plugins to extract meaning from our applications logs. Let's look at some sample application logs that we might want to parse.

Listing 9.52: Custom application logs

```
04-Feb-2016-215959 app=brewstersmillions subsystem=payments
  Payment to James failed for $12.23 on 02/04/2016 Transaction ID
    A092356
04-Feb-2016-220114 app=brewstersmillions subsystem=payments
  Payment to Alice succeeded for $843.16 on 02/04/2016
  Transaction ID D651290
04-Feb-2016-220116 app=brewstersmillions subsystem=collections
  Invoice to Frank for $1093.43 was posted on 02/04/2016
  Transaction ID P735101
04-Feb-2016-220118 app=brewstersmillions subsystem=payments
  Payment from Bob succeeded for $188.67 on 02/04/2016
  Transaction ID D651291
```

We see that these application logs are somewhat contradictory in format. They contain an unusual timestamp, several different types of logging including key-value pairs, strings, another date, and a transaction ID. We're going to assume they are being written to our Logstash server. We'll start with a **tcp** input on our Logstash server to receive those events.

Listing 9.53: Adding a TCP input for our applications

```
input {  
  tcp {  
    port => 2030  
    type => "brewstersmillions"  
    codec => "plain"  
  }  
  . . .
```

Here we've added our `tcp` input on port `2030` with a `type` of `brewstersmillions` to mark our application's events. We've also specified a `codec` of `plain`, as our events are plain text strings.

Now Logstash will receive our log events. When they are received they'll be in the form of an event. An example:

Listing 9.54: An unprocessed custom Logstash formatted event

```
{  
  "message": "04-Feb-2016-215959 app=brewstersmillions subsystem=  
  payments Payment to James failed for $12.23 on 02/04/2016  
  Transaction ID A092356",  
  "@timestamp": "2015-12-13T09:23:51.070+00:00",  
  "@version": "1",  
  "host": "tornado-web1",  
  "type": "brewstersmillions"  
}
```

This unparsed event is not useful, so we need to parse our log events using a filter. The perfect filter is the `grok` plugin. We can use it to match elements of the `message` field and make our event more usable. Let's look at how we might do this.

Listing 9.55: Adding a grok filter for our applications

```
filter {
  if [type] == "brewstersmillions" {
    grok {
      patterns_dir => "/etc/logstash/patterns"
      match => { "message" => "%{APP_TIMESTAMP:app_timestamp} app
= %{WORD:app_name} subsystem=%{WORD:subsystem} %{WORD:
transaction_type} (to|from) %{WORD:user} %{WORD:status}
for \$%{NUMBER:amount} on %{DATE_US:transaction_date}
Transaction ID %{WORD:transaction_id}" }
    }
  }
}
```

We see inside the `filter` block that we're using a conditional to match on any events with a type of `brewstersmillions`. These events are passed to the `grok` filter. We've used a new option, `patterns_dir`, in our filter. The `patterns_dir` option specifies the location of additional, custom patterns we can use to parse log events. Let's create this directory before we continue.

Listing 9.56: Creating the new patterns directory

```
$ sudo mkdir -p /etc/logstash/patterns
```

This creates the `/etc/logstash/patterns` directory. Any files inside this directory will be loaded and parsed for `grok` patterns when Logstash starts. They will then be available to use when parsing log events.

Let's create our first custom pattern, `APP_TIMESTAMP`, which will match the unusual timestamp of our application logs.

Listing 9.57: The `/etc/logstash/patterns/app` file

```
APP_TIMESTAMP %{MONTHDAY}-%{MONTH}-%{YEAR}-%{HOUR}%{MINUTE}%{SECOND}
```

A `grok` pattern has a capitalized name, here `APP_TIMESTAMP`, and then a regular expression. In this case our custom pattern combines [several other grok patterns](#) from the set that ships with Logstash. Here we've combined a series of patterns to match our log event's timestamp. This could be, for example, `04-Feb-2016-215959`.

We then see the `APP_TIMESTAMP` pattern being used in our `grok` regular expression, which is matching on the `message` field.

Listing 9.58: Using the APP_TIMESTAMP pattern

```
" %{APP_TIMESTAMP:app_timestamp} app=%{WORD:app_name} subsystem=%{WORD:subsystem} %{WORD:transaction_type} (to|from) %{WORD:user} %{WORD:status} for \$%{NUMBER:amount} on %{DATE_US:transaction_date} Transaction ID %{WORD:transaction_id}"
```

Our `APP_TIMESTAMP` pattern will assign the value of the regular expression match to a new field called `app_timestamp`. We then use a series of other patterns to extract specific data from the `message` and assign it to new fields. Ultimately,

when the `grok` filter is complete, we should see an event much like:

Listing 9.59: A processed custom Logstash formatted event

```
{  
  "message": "04-Feb-2016-215959 app=brewstersmillions subsystem=  
    payments Payment to James failed for $12.23 on 02/04/2016  
    Transaction ID A092356",  
  "@timestamp": "2015-12-13T09:23:51.070+00:00",  
  "app_timestamp": "04-Feb-2016-215959",  
  "app_name": "brewstersmillions",  
  "subsystem": "payments",  
  "transaction_type": "Payment",  
  "user": "James",  
  "status": "failed",  
  "amount": "12.23",  
  "transaction_date": "02/04/2016",  
  "transaction_id": "A092356",  
  "@version": "1",  
  "host": "tornado-web1",  
  "type": "brewstersmillions"  
}
```

Using Logstash and our `grok` filter, we've turned a custom application log message into structured data. From here we could:

- Send transaction amounts to Riemann and Graphite.
- Send failed transactions to Riemann. For example, we could create an event containing the error—perhaps tagged with the application, the route, or class—with any relevant stack trace or error output as the description.

- Graph failed and successful transactions in Kibana or Grafana.
- Count specific transaction types as metrics.
- Use the event data for audit and diagnostic purposes.

Or we could perform a wide variety of other processing actions.

Health checks, endpoints, and external monitoring

One of the popular ways to monitor applications is an endpoint that displays state and/or metrics, that can be monitored or scraped by a pull-based system of some kind. This endpoint could potentially be the application itself, for example polling for a HTTP 200 status code on a website or API endpoint. Or it could be an endpoint that returns metric data, for example a JSON-based collection of metrics.

This is commonly used, and required, for:

- Gathering metrics, but more often state, for example: “Is the site up?” This is done via monitoring the external surface of a service or application.
- Services that you can’t instrument internally or provided without internal instrumentation.

The former type of external monitoring is frequently done using a SaaS-based service of some kind, for example:

- [Pingdom](#),
 - [Idera](#) (formerly CopperEgg),
 - [Ruxit](#), or
 - [New Relic](#).
-

NOTE This is a list of some of the available services, not an endorsement of any specific service.

These external services are useful because our current monitoring assumes that a check from inside our network takes the same route and is subject to the same conditions as an incoming request from an external location. In many cases traffic conditions, security controls, or network configuration mean that breakages or issues won't be identified from an internal source. Additionally, it's often hard to ascertain if performance issues are related to latency inside our network or latency the customer is subject to en route to our application.

Most of these services provide notification and reporting in addition to monitoring. This is also useful in providing a secondary channel to identify outages if our internal monitoring system fails to detect an issue or is part of a broader failure in our environment.

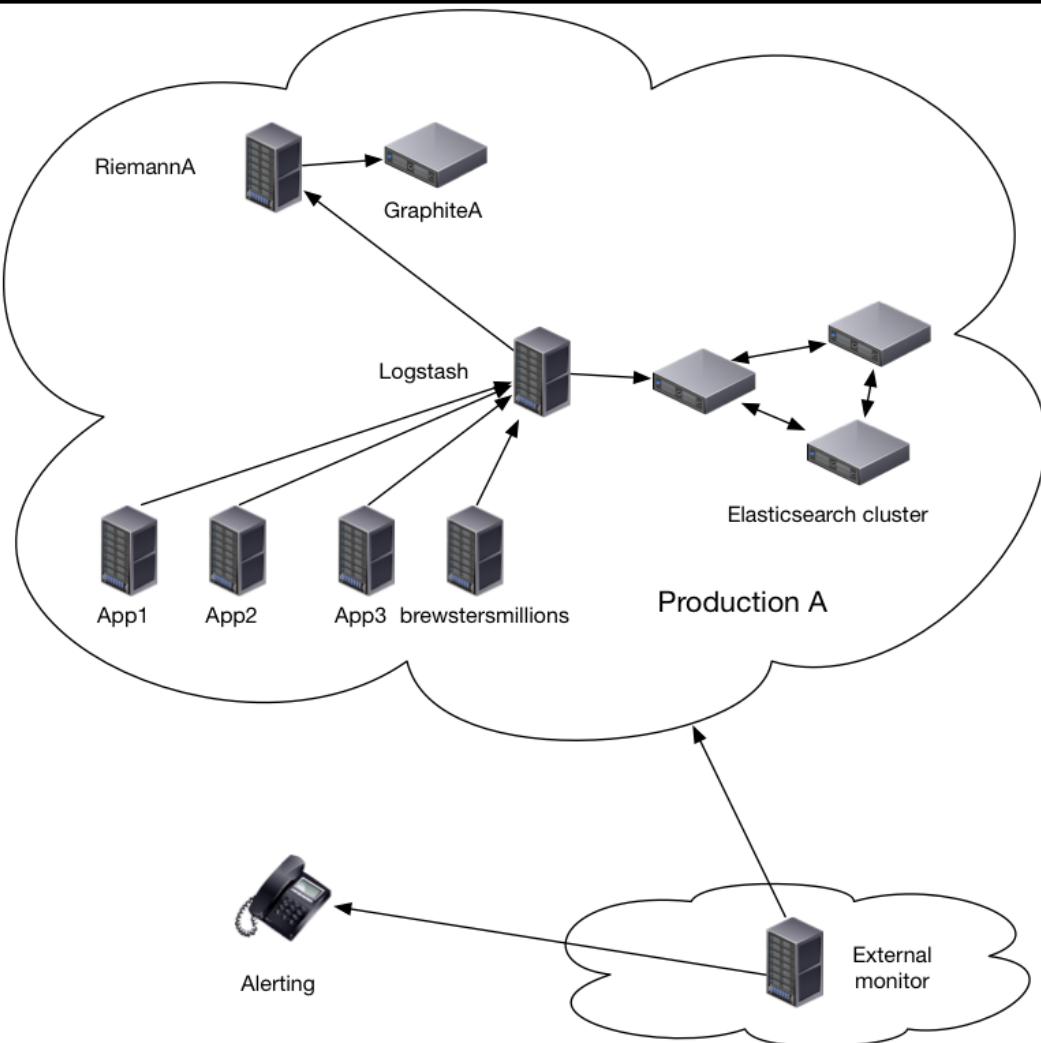


Figure 9.3: External monitoring

You could potentially do your own external monitoring as well. You could set up a host in another data center or Cloud provider and use it to query your applications and services. Perhaps useful here, collectd provides a [ping plugin](#) that you can use to make ICMP queries of hosts. You could also supplement that with some of the plugins we'll talk about in the next section.

Checking an internal endpoint

We don't always need to use an external polling system to query an application's health check endpoint. We can also make use of collectd to grab the contents of an application's health check endpoint via a plugin locally. The collectd framework has several plugins that can be useful here:

- `curl_json` — Connects to a JSON-enabled HTTP or HTTPS endpoint and returns the contents in JSON.
- `curl_xml` — Uses `libcurl` and `libxml2` to curl XML-based endpoints and return the contents in XML.
- `curl` — CURLs HTTP/HTTPS endpoints using `libcurl` and returns the contents in HTML.

Adding an endpoint route to our sample application

Let's see an example of this in action using our sample application. First, we add an `api` route to our Rails application in `config/routes.rb`.

Listing 9.60: Adding an API route to config/routes.rb

```
Rails.application.routes.draw do
  .
  .
  resources :api, only: [:index]
end
```

Then we create a controller for that `api` route in the `app/controllers/api_controller.rb`.

Listing 9.61: The api_controller.rb controller

```
class ApiController < ApplicationController
  protect_from_forgery with: :null_session
  respond_to :json

  def index
    users = { "users": User.count }

    render(json: users)
  end
end
```

We create a basic controller with a single action, `index`, that returns the user count in a JSON field called `users`.

Listing 9.62: The /api endpoint

```
{"users":1}
```

We could easily add additional data to the `api` controller that we might want to expose for consumption by collectd.

Configuring collectd to scrape our endpoint

Now let's configure the local collectd instance to scrape the endpoint and return the data to collectd and hence onto Riemann. To do this we're going to use the `curl_json` plugin.

To configure it we add a configuration file, `curl_json.conf`, to our `/etc/collectd.d` directory. Let's populate that file.

Listing 9.63: The curl.json.conf file

```
LoadPlugin curl_json

<Plugin curl_json>
    <URL "http://localhost/api">
        Instance "aom-rails"
        <Key "*">
            Type "count"
        </Key>
    </URL>
</Plugin>
```

We first load the `curl_json` plugin with the `LoadPlugin` directive. We then configure it in the `Plugin` block. Next we add `URL` blocks for each endpoint we wish to scrape. In this case we're assuming our sample application is bound on the `localhost` interface, and we've specified the path to the `/api` controller.

By default, the plugin will check the endpoint using the value of the global `Interval` setting, in our case every two seconds. But we could override that locally here by specifying an `Interval` directive inside our `URL` blocks.

Inside our `URL` block we've specified the `Instance` directive. The directive sets the value of the `:plugin_instance` field in our event. We'd set it to the name of our application, here `aom-rails`. Next, we specify a `Key` block. The `Key` block controls what piece of JSON we're going to scrape from the endpoint. It matches the key of a JSON hash or the index of a JSON array. We can specify as many `Key` blocks as we need. A wildcard, `*`, can also be used to grab all of the keys, as we've done

here.

Inside a `Key` block we can also specify the `Type` directive, which controls what type of data is in our key. These can be custom types or drawn from the `types.db` file (see `/usr/share/collectd/types.db` on most distributions). We've specified `count`, which tells collectd that our event will be a counter. We can also specify another `Instance` directive. This would control the value of the `:type_instance` field in our event. If we don't specify this directive, the field will default to the value of the key or index being collected.

We'll need to restart collectd for our new check to be enabled.

Listing 9.64: Restarting collectd for curl_json

```
$ sudo service collectd restart
```

Now let's look at what an event from the `curl_json` plugin might look like:

Listing 9.65: A curl_json event from our sample application

```
{:host tornado-web1, :service curl_json-aom-rails/count-users, :  
state ok, :description nil, :metric 1.0, :tags [collectd], :  
time 1450555094, :ttl 60.0, :ds_index 0, :ds_name value, :  
ds_type gauge, :type_instance users, :type count, :  
plugin_instance aom-rails, :plugin curl_json}
```

We see the `tornado-web1` host, and the `:service` field is set to a value of `curl_json-app/count-user_count`. This is constructed from the name of the plugin, `curl_json`, the value of the `Instance` directive in the `curl_json` configuration, and the combination of the `Type` and the `Instance` directives inside the `Key` block. Since we haven't specified an `Instance` directive, we get the JSON

key, here `users`. The `:metric` field contains the value of the `users` hash, in this case `1` user. The `:type_instance` field contains the key of the JSON, `users`, and the `:plugin_instance` field contains the `aom-rails` value from the `Instance` directive in the `URL` block.

This metric will now be automatically graphed in Graphite, and we could also use it for checks inside Riemann itself.

Deployments

Lastly, one of the key things we want to know about applications is when they change—or more specifically, when they are deployed. If we’re tracking metrics and events from applications then it’s important to understand when something happens that could change those metrics. This means we want to be informed when updates are made or changes are deployed to those applications.

Traditionally application deployment has been done by tools. In the Ruby world, for example, that’s via Rake tasks or tools like [Capistrano](#). Other people use configuration management tools like Puppet, Chef, or Ansible. Still others use scripts or tools written in languages ranging from shell to C.

To track our deployments we need to notify Riemann that a deployment is taking place. We’d generally do that at the end of a deployment by creating a metric to track. We use a variant of the schema we introduced earlier in the chapter to see what a metric might look like for a deployment.

Listing 9.66: An example code deployment metric

```
productiona.tornado.production-web1.aom-rails.code_deploy
```

Here we have a metric called `code_deploy` for the `production` release of the `aom-rails` application on the `tornado-web1` host in the `productiona` environment. We

would send a metric, with a `1` or even the SHA of a release as a value, and the time of a successful deployment.

Adding deployment notifications to our sample application

Let's see an example of how we might do this with our sample application. We're going to create a deployment Rake task for our application and add support for sending a notification of that deployment to Riemann. We're then going to see how to add that metric to a graph in Grafana.

To send our deployment events we're going to use Riemann's [native Ruby client](#). We'll first add this gem to our Rails' application's [Gemfile](#).

TIP In addition to the Ruby client, there are [a whole series of other clients](#) you can use to replicate this model in other frameworks and languages.

Listing 9.67: Adding the Riemann client to the aom-rails Gemfile

```
source 'https://rubygems.org'  
ruby '2.2.2'  
gem 'rails', '4.2.4'  
.  
.  
.  
gem 'riemann-client'  
.
```

We then install the new gem using the `bundle` command.

Listing 9.68: Install the gem 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 riemann-client 0.2.6
.
```

We next add a Rake task to run our actual deployment. Let's create one in the `lib/tasks` directory. Any tasks located in here are automatically loaded when we run the `rake` command. We'll call our task file `deploy.rake`.

Listing 9.69: The deploy Rake task

```
require 'riemann/client'
namespace :deploy do
  desc "Deploy the aom-rails application"
  task :release do
    # Your deployment code would go here
    Rake::Task["deploy:notify"].invoke
  end

  desc "Notify Riemann of a deployment"
  task :notify do
    c = Riemann::Client.new host: 'riemann.example.com', port:
      5555, timeout: 5

    c << {
      service: ".aom-rails.#{Rails.env}.code_deploy",
      metric: 1,
      description: "Application aom-rails deployed",
      tags: "deployment",
      time: Time.now.to_i
    }
  end
end
```

We first require the `riemann/client` from the `riemann-client` gem. We then create a namespace called `deploy` and create two tasks, `release` and `notify`, in that namespace. Since we're not actually deploying, we're going to skip the deployment code. Instead our task only does one thing: invokes our second Rake task,

`deploy:notify`, which performs the actual notification. We create a Riemann client, `c`, that connects to `riemann.example.com` on port `5555`. This is the default network port Riemann receives events on that we configured in Chapter 3. The Riemann client defaults to TCP connectivity, and we've also set a timeout of five seconds.

We then use that `c` Riemann client to send an event. That event has a `:service` field with a value of `aom-rails.#{Rails.env}.code_deploy`. The `Rails.env` method will be replaced with current Rails environment, `production`, `development`, etc. The final `:service` field will look something like `.aom-rails.production.code_deploy`. We've specified a `:metric` field value of `1` and then set the `:description` field. We've also added a tag to our event; every deployment event will be tagged with `deployment`. We've specified the `:time` field using the Ruby method `Time.now.to_i`, which returns the current time.

This will send our event to Riemann where we can see it. A typical event would look like:

Listing 9.70: A sample deployment event

```
{:host tornado-web1, :service aom-rails.production.code_deploy, :  
state ok, :description Application aom-rails deployed, :metric  
1, :tags [deployment], :time 1450585652, :ttl 60}
```

We can now make use of this event in Riemann and Graphite.

Working with our deployment events

Now that our deployment events are hitting Riemann we need to make use of them. The first thing we need to do is ensure they are graphed. We're going to add a `where` stream to select all `deployment` tagged events and send them to the `graph` var.

Listing 9.71: Graphing our deployment events

```
(tagged "deployment"
graph
)
```

If we reload Riemann our deployment events will be sent to Graphite.

Listing 9.72: Reloading Riemann to graph deployment events

```
$ sudo service riemann reload
```

Now let's create a new dashboard for our `aom-rails` application. To do this we sign in to Grafana, select the `Home` button, and click `+ New` to create a new dashboard.

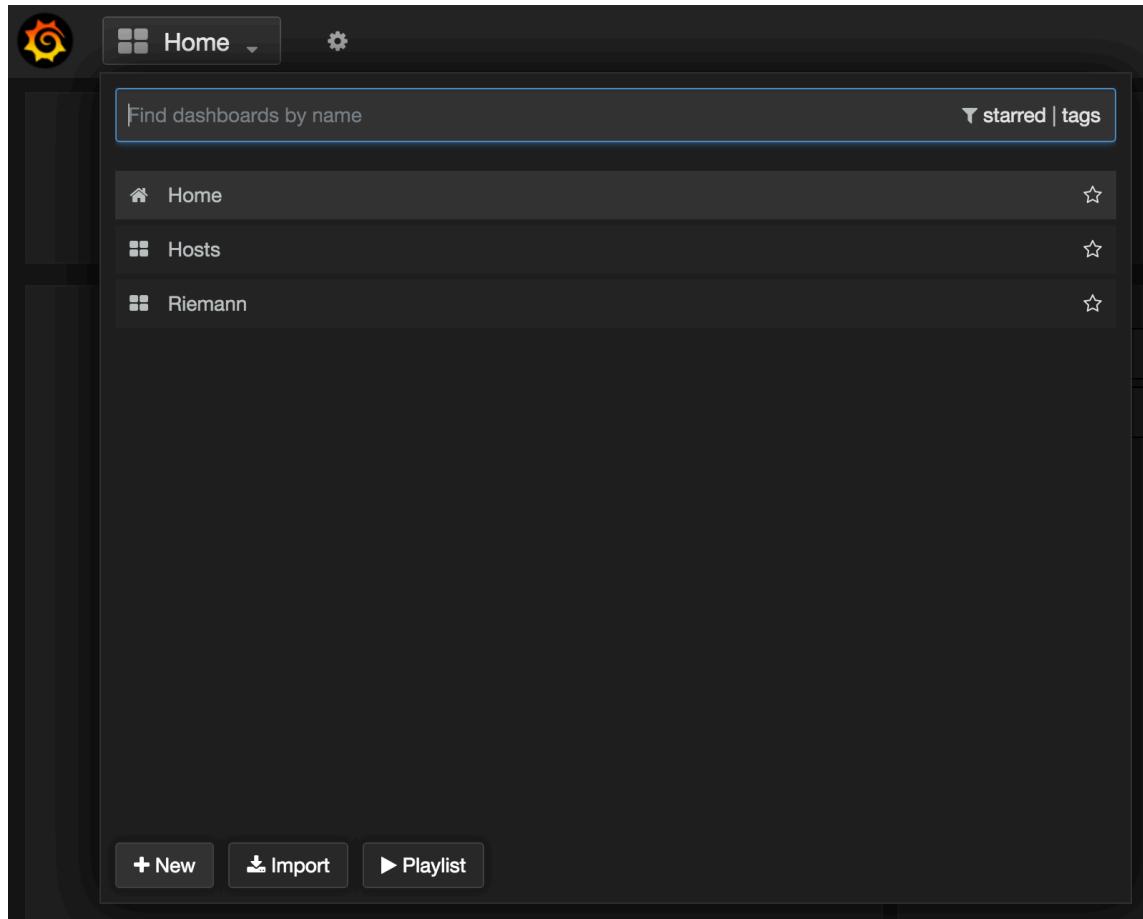


Figure 9.4: Creating the aom-rails dashboard

Once the new dashboard has been created we need to name it. To do this we click the `Manage dashboard` button then select `Settings`. Inside the `Settings` tab we need to update the `Title` field in the `Dashboard Info` box. We're going to call our dashboard `aom-rails`.

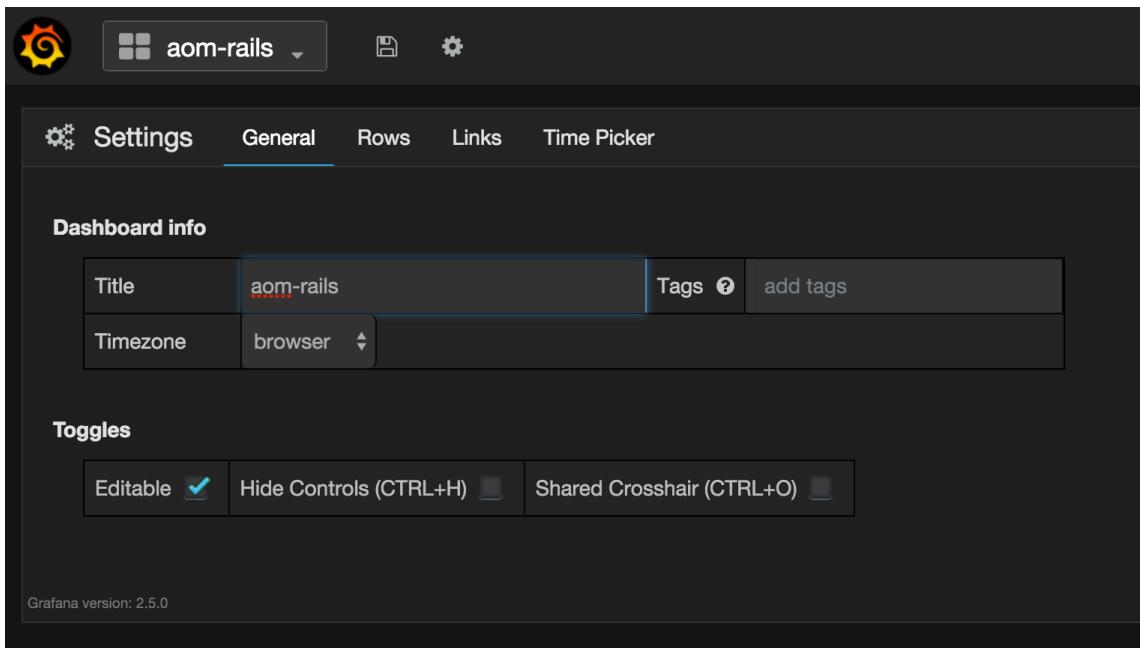


Figure 9.5: Naming the aom-rails dashboard

Then we click the **Save dashboard** button.

Next, we're going to create a [Grafana Annotation](#). Annotations are Grafana's approach to decorating graphs with specific events. We're going to create this Annotation from our `aom-rails.production.code_deploy` metric. We sent that metric to Riemann and then onto Graphite. When we sent it to Graphite it was prefixed with the environment and type of metric, `productiona.hosts`, and the host that generated it. In Graphite, our final metric from our sample application will be called:

```
productiona.hosts.tornado.production-web1.aom-rails.code_deploy
```

To create the Annotation, we again select the **Manage dashboard** button and then the Annotations menu item.

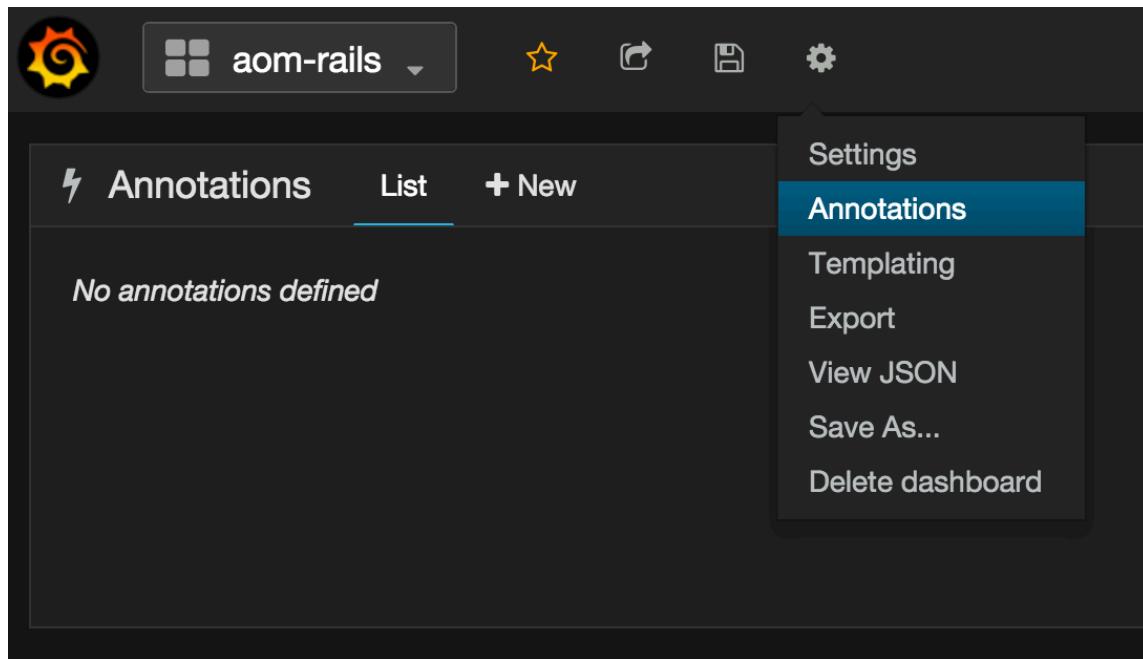


Figure 9.6: Creating an Annotation

To create a new Annotation, we click the **+ New** button.

We first need to give the Annotation a name. We're going to call ours **aom-rails deployed**. We then need to select a data source for our Annotation. In our case this is the default **graphite**. We can also configure the color and style of the annotation here.

We then define the source of the Annotation. We're going to use this metric:

```
productiona.hosts.tornado.production-web1.aom-rails.code_deploy
```

(We could also specify wildcards, for example:

```
productiona.*.*.aom-rails.production.code_deploy
```

if we wanted to select all metrics that match these paths.)

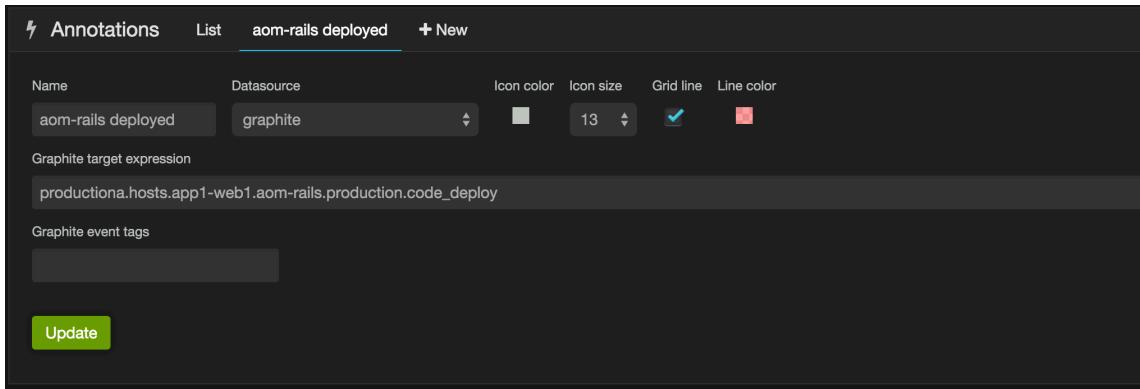


Figure 9.7: Configuring our Annotation

We then click the **Update** button to save our Annotation. We'll close the Annotations box by clicking on the **X** symbol.

Now there will be an Annotation at the top of our dashboard.

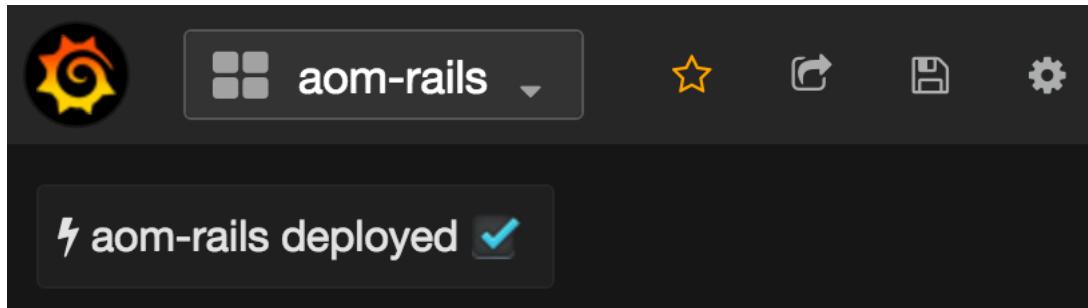


Figure 9.8: The Annotation in the dashboard

If we click on the tick mark next to a specific Annotation and un-tick it, that Annotation will no longer show up on any graphs.

Now, if we create new graphs in this dashboard, we'll get an Annotation on the graphs any time we deploy our **aom-rails** application.

TIP In the Graphite world we'd use the [drawAsInfinite](#) function to create these

kinds of Annotations.

Let's create a graph using the metric we created earlier by scraping our `/api` endpoint:

```
productiona.hosts.tornado-web1.curl_json-app.count-user_count
```

Let's also run our Rake task and deploy our application. I've created that graph below and you can see it here:

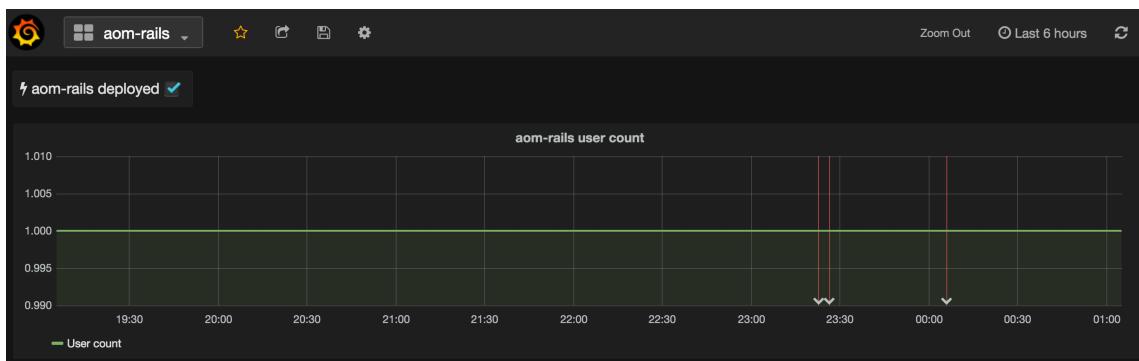


Figure 9.9: Annotated user count graph

The Annotations are marked on the graph in the form of the three red lines, each ending in white arrows. They mark the points in time that our Rake deployment task was run. If our metric was to change as the result of one of those deploys, we'd now identify the connection between the time of the deployment and the change in the metric.

Tracing

Before we move on from this chapter it would be remiss not to mention another diagnostic tool available for application monitoring: tracing, and more specifically, distributed tracing. This is a technique most useful for identifying performance

problems and latency in distributed systems or microservices architectures. We're not going to cover it in the book but you should be aware of it if you go down the microservices path.

A good example of a distributed tracing tool is Twitter's [Zipkin](#) project. Zipkin is in turn based on Google's [Dapper paper on tracing](#).

Summary

In this chapter we explored several ways to monitor and instrument our applications and their workflow, including understanding where to place our application monitoring.

We learned about building our own metrics into our applications and services, and we used StatsD as an example of how to transmit useful metrics.

We also saw the usefulness of structured logging, how to generate our own logs, and how to parse existing logs using Logstash.

We discovered how to extend our internal monitoring to external locations and how to generate and work with health checks and endpoints.

Finally, we discovered how to instrument the lifecycle of our applications and decorate our graphs with Annotations letting us know when our applications are deployed and changed.

In the next chapter we conclude our framework with a chapter on notifications.

Chapter 10

Notifications

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

— “War Games,” 1983

In the Introduction and Chapter 2 we discussed some of the challenges of notifications. The primary purpose of this chapter is to make notifications useful: to send them at the right time and put useful information in them.

Sending too many notifications is the monitoring equivalent of “[the boy who cried wolf](#)”. Recipients will become numb to notifications and tune them out. Crucial notifications are often buried in the flood of unimportant updates.

And even if a notification surfaces, it is not always useful. In Chapter 2 we saw a stock Nagios notification.

Listing 10.1: Sample Nagios notification redux

```
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, 9% disk space free on a 1Gb partition is different from 9% disk free on a 1Tb disk. Can we ignore or mute this notification or do we need to act now?

With this example in mind, we're going to level up our notifications. We're going to focus on four key objectives:

1. We'll add appropriate context to notifications to make them immediately useful.
2. We'll handle maintenance and downtime.
3. We'll also do something useful with non-critical notifications that'll help us identify patterns and trends.

Our current notifications

Up until now our notifications have largely consisted of sending emails and managing multiple notifications via throttling or roll ups inside Riemann itself. This isn't a flexible approach. Additionally, our current email notifications are the default notifications we configured in Chapter 3. Let's convert our existing notifications to be more of a framework with more context attached to our notifications and with the option to target a variety of destinations.

We're going to get some of our additional context by making use of Riemann's index. Remember in Chapter 3 we learned that events that are sent to the index are stored and indexed according to a map of their host and service. These events stay live inside the index until their TTL expires and then they spawn an **expired** event.

Using a Riemann function we query the index for events currently live inside it. We're going to find any events we think might be related to an event which has generated a notification and then output those related events as part of the notification.

Updating expired event configuration

Since we're also going to be using **expired** events, let's revisit our expiration configuration in `/etc/riemann/riemann.config`.

Listing 10.2: Our Riemann expiration configuration

```
(periodically-expire 5 {:keep-keys [:host :service :tags]})
```

We see that the event reaper runs across the index every five seconds, and that we copy the `:host`, `:service`, and `:tags` keys to our expired events. There are other

fields that might be useful to us for our notifications so let's copy those into the new `expired` event too.

Listing 10.3: Our Riemann expiration configuration

```
(periodically-expire 5 {:keep-keys [:host :service :tags, :description, :metric]})
```

Here we've added the `:description` and `:metric` fields to our `expired` event fields.

Upgrading our email notifications

We want to next look at our existing `email.clj` configuration in the `/etc/riemann/examplecom/etc/` directory.

Listing 10.4: The original email.clj

```
(def email (mailer {:from "reimann@example.com"}))
```

The `email` function uses the default formatting we saw in Chapter 3. We can see that [formatting in the `email` function's source code](#).

This will produce email notifications like so:

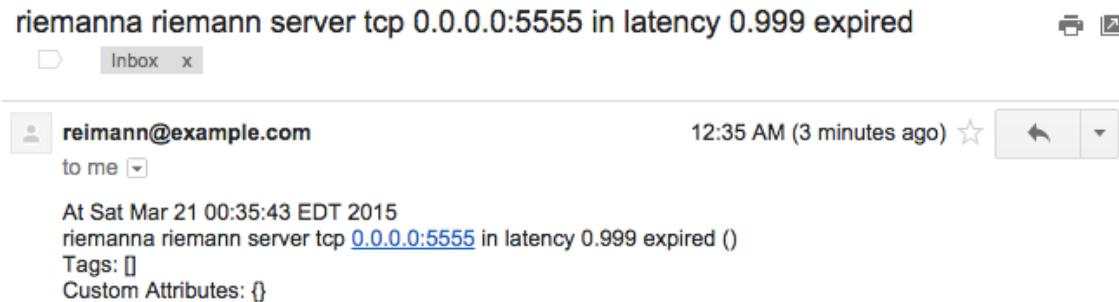


Figure 10.1: A Riemann email notification

Let's expand the `email` function by adding some new formatting to our email notifications. To provide this additional formatting and context we'll use the `:subject` and `:body` options of the `mailer` function. These allow us to specify a function that will format the incoming events any way we like. Let's add some formatting now.

NOTE You can find the code for our new email [notifications on GitHub](#). You'll note in that code that the order of our functions inside the file is basically the reverse of how we've explored them here. In Clojure, the file is parsed from the top down. So we need to define a function before we use it. This can be worked around using the `declare` function.

Listing 10.5: The :format and :body options

```
(ns examplecom/etc/email
  (:require [clojure.string :as str]
            [riemann/email :refer :all]))

(def email (mailer {:from "reimann@example.com"
                    :subject (fn [events] (format-subject events))
                    }
                    :body (fn [events] (format-body events))
                    )))
```

We've added a requirement, `clojure.string`, which we've required as `str`. The `clojure.string` library has some useful functions that will allow us to manipulate and format strings to make our output look better.

We've also added two options to the `mailer` function.

Listing 10.6: The new mailer options

```
    ...
    :subject (fn [events] (format-subject events))
    :body (fn [events] (format-body events))
    ...
```

These options take a function which accepts incoming events and then formats them for the subject line and body of our emails respectively. In our case we've created two new functions, `format-subject` and `format-body`, to do the formatting for us. Let's look at each new function now starting with `format-subject`.

Formatting the email subject

Listing 10.7: The format-subject function

```
(defn format-subject
  "Format the email subject"
  [events]
  (apply format "Service %s is in state %s on host %s" (str/join
    ", " (map :service events)) (str/join ", " (map :state events
    )) (map :host events)))
```

Our `format-subject` function has an argument of `events` for the incoming events. It then uses the `apply` and `format` functions to create a subject line. We have to use the `apply` function and `map` on the various fields because the `mailer` function assumes the `events` argument is a sequence of events rather than a single event.

Listing 10.8: The new subject line

```
(apply format "Service %s is in state %s on host %s" (str/join ",
  " (map :service events)) (str/join ", " (map :state events)) (
  map :host events))
```

This will result in a subject line something like:

Listing 10.9: The new subject line applied

```
Service collectd is in state critical on host tornado-web1
```

This subject line is service-centric. If we want to craft subject lines for other circumstances then we could update the `format-subject` function to accommodate this. We could select formatting for events with specific tags or the contents of specific fields.

TIP If you're interested in learning more about working with Clojure strings then [this guide from the Clojure documentation](#) will help.

Formatting the email body

The next function, `format-body`, is a bit more complex and has some supporting variables and functions. It's also in this function that we're going to add our extra context by searching the Riemann index for related or useful events and outputting them with our notification.

Listing 10.10: The format-body function

```
(defn format-body
  "Format the email body"
  [events]
  (str/join "\n\n\n"
    (map
      (fn [event]
        (str
          header
          "Time:\t\t" (riemann.common/time-at (:time event))
          "\n"
          "Host:\t\t" (:host event) "\n"
          "Service:\t\t" (:service event) "\n"
          "State:\t\t" (:state event) "\n"
          "Metric:\t\t" (if (ratio? (:metric event))
            (double (:metric event))
            (:metric event)) "\n"
          "Tags:\t\t[" (str/join ", " (:tags event)) "] \n"
          "\n"
          "Description:\t\t" (:description event)
          "\n\n"
          (if-not (riemann.streams/expired? event)
            (context event)
            footer)))
      events))))
```

TIP The \t and \n symbols represent tabs and new lines respectively.

There is a lot going on here so let's break it down. The `format-body` function takes a sequence of events as an argument and then outputs some string content for each event. It wraps the `events` argument inside a `join` with three new lines. We then `map` the `events` argument to produce a single `event` and pass that to a function that will construct our output.

Our function first uses the `header` variable. This is boilerplate header text provided by this variable.

Listing 10.11: The header variables

```
(def header "Monitoring notification from Riemann!\n\n")
```

We could adjust this to say whatever we feel is an appropriate opener for our notification email.

We'll then pull our `event` to pieces to construct some notification content.

Listing 10.12: The format-body function

```
"Time:\t\t" (riemann.common/time-at (:time event)) "\n"
"Host:\t\t" (:host event) "\n"
"Service:\t\t" (:service event) "\n"
"State:\t\t" (:state event) "\n"
"Metric:\t\t" (if (ratio? (:metric event))
  (double (:metric event))
  (:metric event)) "\n"
"Tags:\t\t[" (str/join ", " (:tags event)) "] \n"
"\n"
"Description:\t\t" (:description event)
"\n\n"
(context event)
footer))
```

We first use a function from the `riemann.common` namespace, `time-at`. The `time-at` function converts a Unix epoch timestamp into a more human, readable date. We're using it to convert our event's epoch time into something a bit easier to understand. For example, converting `1458385820` in the `:time` field of an event into: `Sat Mar 19 18:59:48 UTC 2016`.

Next we return the host, the name of the service, and the state of the service (taken from the `:host`, `:service`, and `:state` fields respectively). We also return the metric from the `:metric` field. We check if the `:metric` is a `ratio`. If it is, we coerce it into a double using the `double` function.

We then `join` any tags on the event into a list and return the `:description` of the event.

Next comes the most complex part of our formatting: adding useful context to the

notification. We have a `context` function that looks up contextual events in the Riemann index.

Let's take a look at the `context` function.

Listing 10.13: The context function

```
(defn context
  "Add some contextual event data"
  [event]
  (str
    "Host context:\n"
    "  CPU Utilization:\t"(round (+ (:metric (lookup (:host event
      ) "cpu/percent-system")) (:metric (lookup (:host event) "
      cpu/percent-user")))) "%\n"
    "  Memory Used:\t"(round (:metric (lookup (:host event) "
      memory/percent-used))) "%\n"
    "  Disk(root) %:\t\t"(round (:metric (lookup (:host event) "
      df-root/percent_bytes-used))) "% used"
    "  ("(round (byte-to-gb (:metric (lookup (:host event) "df-
      root/df_complex-used")))) " GB used of "
    (round (+ (byte-to-gb (:metric (lookup (:host event) "df-root/
      df_complex-used"))))
    (byte-to-gb (:metric (lookup (:host event) "df-root/
      df_complex-free"))))
    (byte-to-gb (:metric (lookup (:host event) "df-root/
      df_complex-reserved")))) "GB)\n\n"))
```

The `context` function takes an argument of the `event` we're notifying on. It then creates a string consisting of several pieces of contextual information about the host that generated the event, such as the percentage of system and user CPU

usage. It gets this data by looking up events in the Riemann index using our `lookup` function. Let's explore that now.

Listing 10.14: The `lookup` function

```
(defn lookup
  "Lookup events in the index"
  [host service]
  (riemann.index/lookup (:index @riemann.config/core) host
    service))
```

The `lookup` function takes two arguments: a `host`, which is derived from the `event`'s `:host` field, and a `service`, which is the name of the service that contains the metric we want—for example, `memory/percent-used` for the memory used on the host. It uses the `lookup` function from `riemann.index` to search the index for an event matching the host/service pair we specify. It then returns this event to the `context` function.

TIP In addition to the `riemann.index/lookup` function, there's also a `riemann.index/search` function that allows you to query multiple events. You can find an example [here](#).

Inside the `context` function, we extract the `:metric` value from the new event we pulled from the index and put it into the `round` function.

The `round` function formats our metric to two decimal places.

Listing 10.15: The round function

```
(defn round
  "Round numbers to 2 decimal places"
  [metric]
  (clojure.pprint/cl-format nil "~,2f" metric))
```

We've also used another function: `bytes-to-gb`. In our context we show both the percentage used and the GBs used and free on the `root` mount. The `bytes-to-gb` function converts the bytes reported by collectd's `df` plugin metrics into gigabytes.

Listing 10.16: The byte-to-gb function

```
(defn byte-to-gb [bytes] (/ bytes (* 1024.0 1024.0 1024.0)))
```

Once the `context` function has looked up all the events, it returns the string to the `format-body` function to be included in our email message body.

The `format-body` function finishes by returning the contents of the `footer` var, another piece of boilerplate text.

Listing 10.17: The footer variable

```
(def footer "This is an automated Riemann notification. Please do
not reply.")
```

Put together, this results in an email body something like:

Listing 10.18: The new formatted email notification

```
Monitoring notification from Riemann!
```

```
Time:      Mon Dec 28 16:32:37 UTC 2015
```

```
Host:      graphitea
```

```
Service:   rsyslogd
```

```
State:    critical
```

```
Metric:   0.0
```

```
Tags:      [notification, collectd]
```

```
Description:      Host graphitea, plugin processes (instance  
rsyslogd) type ps_count: Data source "processes" is currently  
0.000000. That is below the failure threshold of 1.000000.
```

```
Host context:
```

```
CPU Utilization: 16.84%
```

```
Memory Used:     8.46%
```

```
Disk(root):      32.02% used. (18.85GB used of 58.93GB)
```

```
This is an automated Riemann notification. Please do not reply.
```

TIP We could also create an HTML-based email [as you can see in this blog post](#) using template tools like [Selmer](#). We could also use many of these concepts for non-email notifications.

Adding graphs to notifications

Just quoting a few metrics in the email will be insufficient context in a lot of cases. How about we directly link to the source data for a specific host in our notification? We can include a link to some of the graphs specifically relevant to a host using Grafana's [dashboard scripting](#) capabilities.

A dashboard script is a JavaScript application that loads specific dashboards and graphs based on query arguments sent to Grafana. We can then browse via a specific URL. For example, there is a sample scripted dashboard in the `scripted.js` file, installed when we installed Grafana. View it by browsing to:

`http://graphitea.example.com:3000/dashboard/script/scripted.js`

This scripted dashboard generates graphs that are customizable via query arguments passed to the URL. There are several examples available to review in the `/usr/share/grafana/public/dashboards` directory on the host on which we installed Grafana.

Let's create a scripted dashboard of our own and then insert a link to it in our Riemann notifications. First, though, we need to create our own JavaScript application in the `/usr/share/grafana/public/dashboards` directory. We'll do that now.

Listing 10.19: Creating a new scripted dashboard

```
$ sudo touch /usr/share/grafana/public/dashboards/riemann.js
```

Our objective with this dashboard is to show a small number of critical status graphs that will give us an idea of the host's health. We're going to show graphs including:

1. CPU.

2. Memory.
3. Load.
4. Swap.
5. Disk usage.

We could easily add other graphs or create dashboards for specific applications or groups of hosts. Scripted dashboards are powerful, and there's all sorts of ways to build them. We've built a sample dashboard application that shows one approach, but you can also refer to the example dashboards shipped with Grafana. We've included additional examples later in this chapter.

Our dashboard, contained in the `riemann.js` file, is a little large to show in its entirety here, but let's step through the functionality. Broadly, there are four major sections of a Grafana scripted dashboard.

NOTE You can find the full application [in the book's source code on GitHub](#).

1. Defining a connection to our data source.
2. Defining our query parameters.
3. Defining our graph panels and rows.
4. Rendering the dashboard.

We'll look at extracts from each component in the context of the larger application.

Defining our data source

Let's start by looking at defining our data source. Our `riemann.js` dashboard will use the Graphite-API we installed in Chapter 4 to query our Graphite metrics directly. We're going to define a variable to hold our Graphite connection, originally named `graphite`.

Listing 10.20: Querying the Graphite-API

```
var graphite = 'http://graphitea.example.com:8888';
```

Here we're connecting to our `graphitea.example.com` host on port `8888`. This is where the Graphite-API is running. You can update this or template the file with a configuration management tool to insert the appropriate host name for your environment.

We'll need to make a small change to our Graphite-API configuration file to allow connections directly to it. The Graphite-API relies on [Cross-Origin Resource Sharing](#), or CORS, to ensure only the correct hosts connect to it. We can define approved hosts in the Graphite-API configuration. To do this we update the `/etc/graphite-api.yaml` configuration file on our Graphite hosts.

Listing 10.21: Adding CORS controls to Graphite-API

```
...
allowed_origins:
  - graphitea.example.com:3000
```

Note that we've added a new configuration directive, `allowed_origins`, to our configuration file. This directive allows us to list the hosts that are allowed to connect to the Graphite-API. We've specified our Grafana server, `graphitea.example.com`, running on port `3000`. We'll need to restart Graphite-API to enable this.

Listing 10.22: Restarting Graphite-API for CORS

```
$ sudo service graphite-api restart
```

NOTE We don't need to do this for Grafana's own data source configuration because we proxied the connection in Chapter 4.

Defining our query parameters

Next we define query parameters. These are URL parameters that we'll pass so we're able to query specific hosts, time frames, or other attributes. Grafana will pass this query to Graphite to retrieve our metric data and render our graphs. We'll define defaults for each parameter, too, so that our graphs will load, even without query parameters.

Listing 10.23: Our riemann.js query parameters

```
var arg_host  = 'graphitea';
var arg_span  = 4;
var arg_from  = '6h';
var arg_env   = 'productiona';
var arg_stack = 'hosts';

if (!_.isUndefined(ARGs.span)) {
    arg_span = ARGs.span;           // graph width
}
if (!_.isUndefined(ARGs.from)) {
    arg_from = ARGs.from;          // show data from 'x' hours
    until now
}
if (!_.isUndefined(ARGs.host)) {
    arg_host = ARGs.host;          // host name
}
if (!_.isUndefined(ARGs.env)) {
    arg_env = ARGs.env;            // environment
}
if (!_.isUndefined(ARGs.stack)) {
    arg_env = ARGs.stack;          // stack
}
```

We're specified four query parameters:

1. **span** — The span of our graphs, defaults to **4**.
2. **from** — The age of the data to show, defaults to six hours.
3. **host** — The specific host to query, defaults to **graphitea**.

4. `env` — The environment to query, defaults to `production`.
5. `stack` — The stack, a normal host or a Docker container. Defaults to normal hosts.

The `span` and `from` are fairly self-explanatory. The `host`, `env`, and `stack` parameters help us construct the queries we're going to run to retrieve our metric data from Graphite. They'll also be used to decorate and configure our dashboard. Let's look at an example.

To specify a host with a query parameter we'd browse to:

```
http://graphitea.example.com:3000/riemann.js?host=riemann
```

Grafana will pass the specific host, here `riemann`, to `riemann.js`, which in turn will construct a query to return all the relevant metrics. We could specify a Docker container in a similar way by adding the `stack` parameter to our URL parameters.

```
http://.../riemann.js?host=dockercontainer?stack=docker
```

This would specify a host of `dockercontainer` and update our query to search the `docker` metric namespace.

To run these queries, the `riemann.js` dashboard constructs a query for Graphite using our parameters and some defaults we've defined for our arguments. It uses two variables, `prefix` and `arg_filter`, for this.

Listing 10.24: Our query construction variables

```
var prefix = arg_env + '.' + arg_stack + '.';
var arg_filter = prefix + arg_host;
```

The `prefix` variable contains the first part of our metric name. Remember, our Graphite metrics are constructed like so:

```
environment.stack.host.metric.value
```

The `prefix` variable constructs the `environment.stack` portion of that query using the `arg_env` and `arg_stack` arguments populated from our query parameters or defaults. We then combine this with the hostname, set via the `arg_host` variable. This forms the metric name that we wish to query, using the `query` function like this:

Listing 10.25: The `find_filter_values` query function

```
function find_filter_values(query) {
  var search_url = graphite + '/metrics/find/?query=' + query;
  var res = [];
  var req = new XMLHttpRequest();
  req.open('GET', search_url, false);
  req.send(null);
  var obj = JSON.parse(req.responseText);
  var key;
  for (key in obj) {
    if (obj.hasOwnProperty(key)) {
      if (obj[key].hasOwnProperty("text")) {
        res.push(obj[key].text);
      }
    }
  }
  return res;
}
```

This query function uses the connection information we defined in the `graphite` variable plus a Graphite-API call to the `/metrics/find/` API endpoint. The `query` variable would contain the metric name to search for, for example, `productiona.hosts.graphitea`.

Listing 10.26: The Graphite-API metrics find query

```
http://graphitea.example.com:8888/metrics/find/?query=productiona  
.hosts.graphitea
```

This would return all metrics in this path—every metric underneath:

`productiona.hosts.graphitea`

We can then use these metrics to populate specific graphs.

Defining our graph panels and rows

We then define a series of functions to create rows of graphs and individual graph panels inside those rows. These are functions that feed the configuration for a row or a graph to Grafana to be rendered. Let's look at the function for our memory graph.

Listing 10.27: The panel_memory function

```
function panel_memory(title, prefix) {
    return {
        title: title,
        type: 'graphite',
        span: arg_span,
        y_formats: ["none"],
        grid: {max: null, min: 0},
        lines: true,
        fill: 2,
        linewidth: 1,
        stack: true,
        tooltip: {
            value_type: 'individual',
            shared: true
        },
        nullPointMode: "null",
        targets: [
            { "target": "aliasByNode(" + prefix + "[[host]].memory.used
                ,4)" }
        ],
        aliasColors: {
            "used": "#ff6666",
        }
    };
}
```

Here we've defined a new graph with a series of attributes. The attributes cor-

respond with the configuration options for a graph panel. The key attribute is **targets**, which are the specific metrics to display, here:

Listing 10.28: The targets attribute

```
targets: [
  { "target": "aliasByNode(" + prefix + "[[host]].memory.used,4)"
  }
],
```

We construct the metric to display using a combination of Graphite functions, our query, and Grafana's template capability. If we were to introspect these variables we'd get:

```
aliasByNode(productiona.hosts.graphitea.memory.used,4)
```

We first see the Graphite **aliasByNode** function, which we introduced in Chapter 6; it creates an alias from some element on the metric name. We combine that with the **prefix** variable we constructed earlier. We then see a Grafana template variable, **[[host]]**. This is populated inside our dashboard application from the **arg_host** option. When the graph is rendered, **[[host]]** will be replaced with the specific host we're querying. Lastly, we see the tail end of our metric name, **memory.used**, which returns the percentage memory used on the host. We're aliasing by the fourth element from a zero count in the metric name **used**.

We can then add our graph panel to a row. Remember, Grafana dashboards are made up of rows that can contain a series of different components. We define a new function for each row.

Listing 10.29: The row_cpu_memory function

```
function row_cpu_memory(title, prefix) {
    return {
        title: title,
        height: '250px',
        collapse: false,
        panels: [
            panel_cpu('CPU %', prefix),
            panel_memory('Memory', prefix),
            panel_loadavg('Load avg', prefix)
        ]
    };
}
```

In this row we've defined three graph panels: `panel_memory`, which we've just seen, as well as `panel_cpu` and `panel_loadavg`, which you can look at in the [book's source code](#); the latter two define CPU and load graph panels respectively.

Rendering the dashboard

Lastly, our application contains a callback function that defines some attributes, like our `[[host]]` template variable, the dashboard title, and other items, and then renders the dashboard. If we run the default scripted dashboard, we'll see something like this:

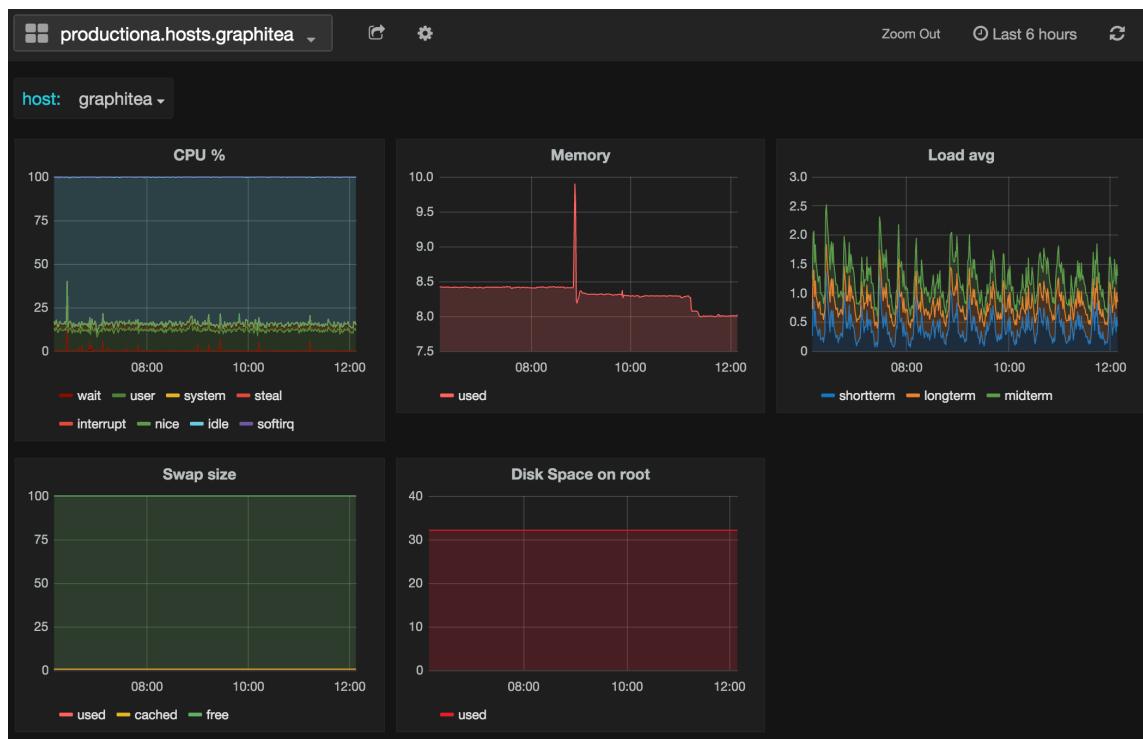


Figure 10.2: Our scripted dashboard

We see two rows of graphs, each containing the five types of data we defined in our graph panels. Now how do we add this to our notification?

NOTE Our scripted dashboard was modified from an example created by [Bimlendu Mishra](#).

Adding our dashboard to the Riemann notification

We've already got our `context` function in our `email.clj` file. Let's extend this slightly to add a link to our dashboard.

Listing 10.30: Adding a dashboard to our context function

```
(defn context
  "Add some contextual event data"
  [event]
  (str
    "Host context:\n"
    "  CPU Utilization:\t"(round (+ (:metric (lookup (:host event
      ) "cpu/percent-system")) (:metric (lookup (:host event) "
      cpu/percent-user")))) "%\n"
    "  Memory Used:\t"(round (:metric (lookup (:host event) "
      memory/percent-used))) "%\n"
    "  Disk(root):\t\t"(round (:metric (lookup (:host event) "df-
      root/percent_bytes-used")))% used.\n\n"
    "Grafana Dashboard:\n\n" "http://graphitea.example.com:3000/
      dashboard/script/riemann.js?host=\"(:host event)\"\n"))
  
```

Note that we've added the following lines to our `context` function.

Listing 10.31: A dashboard in context

```
"Grafana Dashboard:\n\n"
"  http://graphitea.example.com:3000/dashboard/script/riemann.js?
  host=\"(:host event)\"\n\n"
  
```

This will add a URL for our scripted dashboard and insert the relevant hostname from the `:host` event field. When we get our notification now it'll look like:

Listing 10.32: Our new graph-enhanced notification

Monitoring notification from Riemann!

Time: Wed Dec 30 17:22:56 UTC 2015

Host: graphitea

Service: rsyslogd

State: critical

Metric: 0.0

Tags: [notification, collectd]

Description: Host graphitea, plugin processes (instance rsyslogd) type ps_count: Data source "processes" is currently 0.000000. That is below the failure threshold of 1.000000.

Host context:

CPU Utilization: 16.27%

Memory Used: 8.18%

Disk(root): 32.13% used.

Grafana Dashboard:

[http://graphitea.example.com:3000/dashboard/script/riemann.js?
host=graphitea](http://graphitea.example.com:3000/dashboard/script/riemann.js?host=graphitea)

This is an automated Riemann notification. Please do not reply.

When we get our notification we can then click on the embedded link in the email and open up our dashboard to get some further context on the host with the issue.

Some sample scripted dashboards

To help you build different dashboards we've collected a few examples of scripted Grafana dashboards.

- A [sample dashboard for Graphite and InfluxDB](#).
- Two other [dashboard examples](#).
- Some [utility methods for building scripted dashboards](#).
- A [template for building a scripted dashboard](#).

Other context

This is not the limit of the possible context we could add to a notification. Some further ideas to consider:

- Longer-term metrics, using the Graphite API to populate the notification with more historic metric data.
- Application or business-specific metrics in addition to host-centric metrics. Remember, you can retrieve anything in the index or in Graphite.
- Links to runbooks, documentation, or a wiki.
- Graphs embedded directly into the notification. You can [render graphs using the Graphite API](#).
- Decorating notifications with contextual configuration information about hosts and services. For example, pulling from a configuration management store like PuppetDB, from a distributed configuration tool [like Zookeeper](#), or from a traditional CMDB.
- Shell-out from Riemann to gather additional context. For example, if a notification was a disk space notification, then your first step on that host would typically be to run the `df -h` command. You could have Riemann run that command on the impacted host via a tool like [MCollective](#), [PSSH](#), or [Fabric](#), and add the resulting output to the notification.

TIP Of interest in this domain is [Nagios Herald](#) which aims to decorate Nagios notifications. It provides some excellent [examples and ideas](#) for adding appropriate context to notifications.

Adding Slack as a destination

Even though we've improved our email notifications, only having a single type of notification is a little limiting. Let's add another type of notification to Riemann: [Slack](#) notifications. Slack is a SaaS-based team communication service with the concept of channels and private messages. Notifications to services like Slack, where you are notifying a channel or room of recipients, can be useful for sharing quick real-time notifications of situations that require "in the moment" reactions.

TIP This is just an example of adding chat notifications to Riemann. You aren't limited to using Slack—Riemann comes with plugins for other services like Campfire and Hipchat—but you can learn how to use Slack [through their documentation](#).

We're going to start by creating a new function to send notifications to Slack. Let's create this function now in a new file, [`slack.clj`](#), to be included in our Riemann configuration.

Listing 10.33: Adding the slack.clj file

```
$ touch /etc/riemann/examplecom/etc/slack.clj
```

Now let's populate this file.

Listing 10.34: The slack notification configuration

```
(ns examplecom.etc.slack
  (:require [riemann.slack :refer :all]))

(def credentials {:account "examplecom", :token "123ABC123ABC"})

(defn slack-format
  "Format our Slack message"
  [event]
  (str "Service " (:service event) " on host " (:host event) " is
       in state " (:state event) ".\n"
       "See http://graphitea.example.com:3000/dashboard/script/
       riemann.js?host="(:host event)))

(defn slacker
  "Send notifications to Slack"
  [& {:keys [recipient]
       :or {recipient "#monitoring"}}]
  (slack credentials {:username "Riemann bot"
                     :channel recipient
                     :formatter (fn [e] { :text (slack-format e)
                                       } )
                     :icon ":smile:}))
```

We've first required the `riemann.slack` functions which contain Riemann's slack connector.

TIP We could also connect to other services like [Campfire](#) or [Hipchat](#).

Next we've used the `def` statement to define a var called `credentials` that contains our Slack security credentials. To connect to Slack we need the name of our Slack organization, here `examplecom`, and a token. The Slack connection uses [Incoming Webhooks](#) to receive Riemann notifications. To use it, we've defined a Slack Incoming Webhook for our Riemann monitoring notifications.

We got back a URL when we created the Webhook:

Listing 10.35: The Slack Webhook URL

```
https://hooks.slack.com/services/T037T0K31/B048W3VQY/123ABC123ABC
```

The token is the last piece of your Slack Webhook URL, here `123ABC123ABC`.

Next, we've defined another function called `slack-format` that will format our actual Slack notification.

Listing 10.36: The slack-format function

```
(defn slack-format
  "Format our Slack message"
  [event]
  (str "Service " (:service event) " on host " (:host event) " is
       in state " (:state event) ".\n"
       "See http://graphitea.example.com:3000/dashboard/script/
          riemann.js?host="(:host event)))
```

Our function constructs a message to be sent. It takes an argument of a standard

Riemann event. It then uses the `str` function, which turns everything after it into a string. In this string we extract the values of the `:service`, `:host`, and `:state` fields from the `event` to create a notification like:

Listing 10.37: The Slack notification

```
Service rsyslog on host graphitea is in state critical.  
See http://graphitea.example.com:3000/dashboard/script/riemann.js  
?host=graphitea
```

We'll see how to apply this function shortly.

Lastly, we've defined another function called `slacker` that will actually send events to our Slack instance. The `slacker` function has an argument construction we've not seen before.

Listing 10.38: Our default argument

```
(defn slacker  
  "Send notifications to Slack"  
  [& {:keys [recipient]  
       :or {recipient "#monitoring"}}]  
  ...)
```

Here we've specified what looks like an optional argument for the function, represented by the `&`, but is instead a default argument. We use the `:keys` and `:or` options to specify an argument of `recipient`, which will be the Slack room to which we send our notification. The default for this is `#monitoring`. We're saying to Clojure: run this function with whatever argument we specify for `recipient` or default, using `#monitoring` as the recipient room.

TIP You can read more about functions and their arguments in the [Clojure function documentation](#).

Our function then calls the `slack` plugin, passes in the required `credentials` variable, and configures the details of our notification. We can configure a variety of options, in addition to the recipient, including:

- The `:username` of the bot that'll announce the Riemann events.
- The `:icon` that defines the icon the bot uses.

We've also defined an option called `:formatter`.

Listing 10.39: The Slack formatter option

```
:formatter (fn [e] { :text (slack-format e) } )
```

The `:formatter` option is useful and allows us to structure the actual message our Slack channel receives. This is much like the formatting we've done for email subjects and bodies. In this case we've passed our `slack-format` function to the formatter `:text` option with a variable of `e`, which is shorthand for the event. The `:text` option formats the Slack message itself. We can also add attachments or more complex formatting to the message, as you can see in [the Slack plugin source code](#).

We can use our new destination like so:

Listing 10.40: Using the Slack notification

```
(slacker "#operations")
```

Or we can just skip the recipient if we want it to default to our `#monitoring` room.

Listing 10.41: Using the Slack notification default

```
(slacker)
```

Adding PagerDuty as a destination

Let's add a final type of notification using a service called [PagerDuty](#). PagerDuty is a commercial notification platform. It allows you to create users, teams, and on-call schedules and escalations. You can then send notifications to the platform that can be routed to those teams via a variety of mechanisms: email, SMS, or even via a voice call.

TIP You can learn how to use PagerDuty [using their documentation](#).

We're choosing PagerDuty as it's a good example of these types of services. If PagerDuty isn't a fit for you, there are several other commercial alternatives supported in Riemann including [VictorOps](#) and [OpsGenie](#).

Let's start by creating a new function to send notifications to PagerDuty. We add our new function to a file, `pagerduty.clj`, in the `/etc/riemann/examplecom/etc/`

directory, to be included in our Riemann configuration.

Listing 10.42: Adding the pagerduty.clj file

```
$ touch /etc/riemann/examplecom/etc/pagerduty.clj
```

Now let's populate this file.

Listing 10.43: The pagerduty.clj file

```
(ns examplecom/etc/pagerduty
  (:require [riemann.pagerduty :refer :all]
            [riemann.streams :refer :all]))


(defn pd-format
  [event]
  {:incident_key (str (:host event) " " (:service event))
   :description (str "Host: " (:host event) " "
                      (:service event) " is "
                      (:state event) " (" 
                      (:metric event) ")")
   :details (assoc event :graphs (str "http://graphitea.example.
                                         com:3000/dashboard/script/riemann.js?host="(:host event))))}

(def pd (pagerduty { service-key: "ABC123ABC" :formatter pd-
  format }))

(defn page
  []
  (changed-state {:init "ok"}
    (where (state "ok"))
    (:resolve pd)
    (else (:trigger pd)))))
```

In our `pagerduty.clj` file, we first add a namespace, `examplecom/etc/pagerduty`, and require the `riemann.pagerduty` library.

We then define a function called `pd-format`. This function formats our PagerDuty

message. We don't need to specify a function for this, there's a default format, but we want to customize our message a bit. The function takes the `event` as a parameter and needs to return a map containing:

- The `incident_key` — PagerDuty's incident key identifies a specific incident. We use the host and service pairing.
- The `description` — A short description of the problem.
- The `details` — The full event.

It's here in this function that we can tweak our PagerDuty format. In our case we've only made one change.

Listing 10.44: The `:details` key with graph URL

```
:details (assoc event :graphs (str "http://graphitea.example.com  
:3000/dashboard/script/riemann.js?host=" (:host event)))
```

We've used the `assoc` function to add a new entry to our `event` map. The new entry is called `:graphs` and contains the URL to our scripted Grafana dashboard. If we wished, we could also use something like our `context` function from our email notifications to add context to our PagerDuty notifications.

We next define a var called `pd` which configures the `pagerduty` plugin. It has two arguments: our PagerDuty service key, here represented by the fake key `ABC123ABC`, and the name of our formatter function, `pd-format`. To create a service key, [add an API service to PagerDuty](#) and select a type of `Riemann`. Then take the resulting service key and add it to your Riemann configuration.

We also define a function called `page` to execute our adapter. Inside our `page` function we've used the `changed-state` var we introduced in Chapter 6. This detects changes in the value of the `:state` field of an event. It also implies a `by`

stream that splits events on the `:host` and `:service` fields, generating an event for every host and service pair.

We've specified an initial state assumption of `ok`. The initial state allows us to start from a known state when Riemann first sees a service. With this configuration Riemann assumes that a new service it sees with a `:state` of `ok` is in its normal state, and to not take any action.

Inside the `pagerduty` adapter there are three actions we can take with an incoming event. These actions relate to PagerDuty's incident management process.

- `:trigger` — Trigger an incident.
- `:acknowledge` — Acknowledge an incident exists but don't resolve it.
- `:resolve` — Resolve an incident.

In our case, we want any event with a `:state` of `ok` to resolve an incident. We also want any event whose `:state` is not `ok` to trigger an incident. We've done this with a `where` stream matching the `:state` field of the event.

NOTE We haven't used the `:acknowledge` action at all.

We then use our `page` destination in a notification.

Listing 10.45: Using our PagerDuty notification

```
(require '[examplecom.etc.pagerduty :refer :all])

. . .

(tagged "notification"
  (changed-state {:init "ok"})
  (adjust [:service clojure.string/replace #"\^processes-(\.*)\/"
            ps_count$ "$1"]
    (page))))
```

We first need to `require` the `Pagerduty` function in `examplecom.etc.pagerduty`.

Here we've used our `page` function in our `notification` tagged events to trigger or resolve PagerDuty incidents. If a process failure is detected then a PagerDuty incident will be triggered.

The screenshot shows the PagerDuty incident management interface. At the top, there are two tabs: 'Your open incidents' (1 triggered, 0 acknowledged) and 'All open incidents' (1 triggered, 0 acknowledged). Below these are filters for 'Open 1', 'Triggered 1', 'Acknowledged 0', 'Resolved', and 'Any Status'. A button 'Assigned to me' is also present. The main table lists one incident: 'High' urgency, incident number 188, created on Jan 13, 2016 at 01:17 AM, with the details 'Host: graphitea rsyslogd is critical (0.0)'. It is assigned to 'Riemann' and 'James Turnbull' and is in a 'Triggered' state. Buttons for 'Acknowledge', 'Reassign', and 'Resolve' are available for this incident. At the bottom, there are additional filters and a page navigation section.

Figure 10.3: A PagerDuty incident

Alternatively, if the monitored process recovers, then the notification will be re-

solved.

Maintenance and downtime

One of the other aspects of notifications we should consider is when not to send them. On many occasions you don't want notifications to trigger because you're performing maintenance on a host or service. In these cases you know a service might fail or be deliberately stopped—you don't need a notification to be sent.

With Riemann we solve this by injecting maintenance events. A maintenance event is a normal Riemann event that we identify by host, service, or a specific tag. The event will have an infinite TTL or time to live. If we want to start a maintenance window we send Riemann one of these maintenance events with a `:state` of `active`. If we want to end the maintenance window we send another event with a `:state` of anything but `active`.

To check for maintenance events we're going to build a check that will execute before notifications. The check will search the Riemann index for any maintenance events. If it finds an event that matches the host and service that triggered the notification then it will check the event's `:state`. For any events that have a `:state` of `active` it will abort the notification.

Let's start with the function that will perform that check. We're going to create a new file:

`/etc/riemann/examplecom/etc/maintenance.clj`

We'll populate that file with this function.

Listing 10.46: The maintenance-mode? function

```
(ns examplecom.etc.maintenance
  (:require [riemann.streams :refer :all]))  
  
(defn maintenance-mode?
  "Is it currently in maintenance mode?"
  [event]
  (->> '(and (= host (:host event))
              (= service (:service event))
              (= (:type event) \"maintenance-mode\")))
        (riemann.index/search (:index @core))
        first
        :state
        (= "active")))
```

We create a namespace: `examplecom.etc.maintenance` and required the `riemann.streams` library to provide access to Riemann's streams.

This new function, called `maintenance-mode?`, takes a single argument: an event. It then uses a macro, `->>`. The `->>` macro rearranges the series of expressions, reversing the order and running through the forms. You can think about this expression as being:

Listing 10.47: The `->` expression expanded

```
(= "active" (:state (first (riemann.index/search (:index (clojure.core/deref core)) (quote (and (= host (:host event)) (= service (:service event)) (= (:type event) "maintenance-mode"))))))
```

This series of expressions means: “If the `:state` field of the first event returned by a search in the index is `active`, has a matching host and service, and a custom field of `:type` with a value of `maintenance-mode`.“

The index search is being done using the `riemann.index/search` function we discussed earlier in the chapter.

We then wrap our notifications with the new function.

Listing 10.48: Checking maintenance prior to notifications

```
(tagged "notification"
  (where (not (maintenance-mode? event)))
  (changed-state {:init "ok"}
    (adjust [:service clojure.string/replace #"\^processes-(.*)"
      \/ps_count\$" "$1"]
    (page))))
```

Let’s schedule some maintenance. We can do this a number of different ways, such as by manually submitting an event using one of the Riemann clients like that Riemann Ruby client. We’ll install the Ruby client now.

Listing 10.49: Installing the Riemann Ruby client

```
$ sudo gem install riemann-client
```

We'll use the `irb` shell to send a manual event.

Listing 10.50: Sending a Riemann maintenance event

```
$ irb
irb(main):001:0> require 'riemann/client'
=> true
irb(main):002:0> client = Riemann::Client.new host: 'riemann.
example.com', port: 5555, timeout: 5
irb(main):003:0> client << {service: "nginx", host: "tornado-web1",
  type: "maintenance-mode", state: "active", ttl: Float::
INFINITY}
=> nil
irb(main):003:0>
```

We require the `riemann/client` and then create a `client` that connects to our Riemann server. We then send a manual event with a `:service` of `nginx` for the relevant host, `tornado-web1`, and with a custom field of `:type` set to a value of `maintenance-mode`. The `:state` field will be set to `active` and the TTL of the event is forever. If a notification were to now trigger on the `tornado-web1` host, then Riemann would detect the active maintenance event and not send the notification.

If the maintenance window was over we could disable it like so:

Listing 10.51: Disabling the maintenance window

```
irb(main):002:0> client << {service: "nginx", host: "tornado-web1", type: "maintenance-mode", state: "inactive", ttl: Float::INFINITY}  
=> nil
```

Using the Riemann client directly is a little clumsy so we've actually written a tool to help automate this process. It's a [Ruby gem called maintainer](#). It can be installed via the `gem` command.

Listing 10.52: Installing the maintainer gem

```
$ sudo gem install maintainer
```

We then use it like so:

Listing 10.53: Using the maintainer gem to generate an active event

```
$ maintainer --host riemann.example.com --event-service nginx
```

This will generate a maintenance event for the current host (or you can specify a specific host with the `--event-host` flag). The event will look something like:

Listing 10.54: The maintainer active event

```
{:host tornado-web1, :service nginx, :state
active, :description Maintenance is active, :metric nil, :tags
nil,
:time 1457278453, :type maintenance-mode, :ttl Infinity}
```

We can also disable maintenance events like so:

Listing 10.55: Using the maintainer gem to disable maintenance

```
$ maintainer --host riemann.example.com --event-service nginx --
event-state inactive
```

Which will generate an event like this:

Listing 10.56: The maintainer active event

```
{:host tornado-web1, :service nginx, :state
inactive, :description Maintenance is inactive, :metric nil, :
tags nil,
:time 1457278453, :type maintenance-mode, :ttl Infinity}
```

We can then wrap this binary inside a configuration management tool or a Cron job or whatever else triggers our maintenance windows.

Learning from your notifications

We've done our best to uplift the quality of our notifications in this chapter. We've also created a fairly small collection of checks through the book. In the next chapters we'll put all of this together and you'll see what the volume of checks and notifications might be for an application stack, and hence be able to extrapolate an overall volume for your environment.

With the full set of checks and notifications in your environment you're likely to find that notifications generally fall into two categories:

- Notifications that are useful and that you need to action.
- Notifications that you delete.

This is also an evolutionary process rather than an immediate one. A notification that was useful one day may become an “oh, that again ... *delete*” notification a month later. Other times a notification is for a host, service, or application that is no longer important or mission critical but still exists. Or, more insidious, a notification is for a component or aspect of a host, service, or application that is still important but doesn't rise to the level of immediate action. These are the sort of notifications we tend to address by placing a ticket in a backlog or putting them aside for a rainy day when we might have time to take a look.

There are several approaches we can take to these notifications. There's the obvious: deletion. We could also turn off notifications for these hosts, services, or applications. We could even let them pile up. Most of us have had, on occasion, a mail filtering rule that places spurious notifications from monitoring tools into a specific folder.

The approach we're going to take, though, is to make a rule that any notification we don't action we count. We then use these spurious notifications to detect trends and highlight potential issues in our applications and infrastructure.

To do this we're going to create a new function called `count-notifications`. Let's create a file to hold our new counting function.

Listing 10.57: Adding the count-notifications.clj file

```
$ touch /etc/riemann/examplecom/etc/count-notifications.clj
```

Now let's populate this file.

Listing 10.58: Adding the count-notifications function

```
(ns examplecom.etc.count-notifications
  (:require [riemann.streams :refer :all]))


(defn count-notifications
  "Counts notifications"
  [& children]
  (adjust [:service #(str % ".alert_rate")]
    (tag "notification-rate"
      (rate 5
        (fn [event]
          (call-rescue event children))))))
```

We have created a namespace `examplecom.etc.count-notifications` and required the Riemann streams. You can see we've defined a new function: `count-notifications`. It doesn't have any arguments, except any child streams passed to it (we're going to graph our notification rate as you'll see shortly). We first adjust any `:service` field passed to our function to suffix `.alert_rate` to the end of the service. We also use the `tag` var to add a tag, `notification-rate`, to

our `:tags` field. This will make it easier for us to work with our new event.

Next, we use the `rate` var to calculate a rate from our event. The `rate` var takes the sum of the `:metric` over a specified interval, here `5` seconds, and divides by the interval size. This calculates a rate that's emitted as an event every interval seconds. The rate starts as soon as an event is received, and stops when the most recent event expires.

So, in our case, we sum the `:metric` over five seconds, divide by five to calculate the rate, and then emit an event with the `:service` name suffixed by `.rate` every five seconds.

Let's see how we might use our new function. Let's say we have a check like so:

Listing 10.59: A memory usage check

```
(by :host
  (where (and (service "memory/percent-used") (>= metric 80.0))
    (email "james@example.com")))
```

This check uses the `by` stream to split events by `:host` and then checks the `memory/percent-used` metric from collectd. It notifies via email if the memory usage metric exceeds 80%. Let's say we get a lot of these notifications and they sit unactioned in an email folder. Rather than deleting the check or suppressing the notification, we update it to use our new `count-notifications` function. Here's our updated check.

Listing 10.60: A memory usage check notification count

```
(by :host
  (where (and (service "memory/percent-used") (>= metric 80.0))
    (count-notifications (smap rewrite-service graph))))
```

Note we've replaced the `email` function with `count-notifications` and directed the output into a child stream, `(smap rewrite-service graph)`, which will write our new event to Graphite. Now when a notification is triggered we'll see a new metric being created. For example, for the `graphitea` host and taking the rewrite rules we established in Chapter 6 into consideration:

- `productiona.hosts.graphitea.memory.used.alert_rate`

This would be the rate of notifications over the five-second interval. We then use this rate, either in a graph or by creating another check that monitors a rate threshold. Let's see how we might do this:

Listing 10.61: A memory usage check reinjection

```
(by :host
  (where (and (service "memory/percent-used") (>= metric 80.0))
    (count-notifications reinject (smap rewrite-service graph)))

  (tagged "notification-rate"
    (by :service
      (where (>= metric 100)
        (with { :state "warning", :description "Notification count
          warning" }
          (email "james@example.com")))))
```

We pretty much see our previous memory usage check with one addition: the `reinject` var. The `reinject` var sends events back into the Riemann core. Rejected events flow through all top-level streams, as if they've just arrived from the network.

We see below our check that we've created another check. This one uses the `tagged` stream to select any events, including our reinjected events, with a tag of `notification-rate`. We then use a `by` stream to split our events by the `:service` field and a `where` stream to match any event with a `:metric` value equal to or higher than `100`. This would match any service with a notification rate of 100 notifications in five seconds, almost certainly signaling some kind of issue.

Next, we use the `with` stream to create a new copy of our event with new values for certain fields. Here we change the `:state` field to a value of `warning` and add a `:description` field of `Notification count warning`. The new event is then emailed to `james@example.com`. Our event would end up something like this:

Listing 10.62: Our notification count warning notification

Monitoring notification from Riemann!

Time: Wed Jan 06 01:05:12 UTC 2016
Host: graphiteb
Service: memory/percent-used.alert_rate
State: warning
Metric: 100.986363814988014
Tags: [notification-rate, collectd]

Description: Notification count warning

Host context:

CPU Utilization: 78.50%
Memory Used: 88.31%
Disk(root): 78.49% used.

Grafana Dashboard:

[http://graphitea.example.com:3000/dashboard/script/riemann.js?
host=graphiteb](http://graphitea.example.com:3000/dashboard/script/riemann.js?host=graphiteb)

This is an automated Riemann notification. Please do not reply.

Now we get an notification when the rate of notifications themselves is abnormal or exceeds a specified threshold.

TIP Also of interest in this arena is [OpsWeekly](#). It's a tool written by the Etsy team that acts as a living life-cycle-reporting and State of the Union view of your operational environment.

Other alerting tools

- [Flapjack](#) - A monitoring notification router.
- [Prometheus Alertmanager](#) - Part of the Prometheus monitoring framework.
Can be used as a standalone alert manager.

Summary

In this chapter we've extended our email notifications to provide additional context and information. We've used data from the Riemann index to give the reader of a notification some idea of the current state of the environment around that host or service. We've also added the ability to reference external data from a variety of sources, such as by presenting a custom dashboard for a host or service.

In addition, we've added new destinations that are useful for different types of notifications. Notifications to destinations like Slack can be useful for sharing quick real-time notifications of situations that require "in the moment" reactions. We can also make use of commercial notification services like PagerDuty, whose ability to manage teams of users, on-call schedules, and escalations make it easier to manage accountability for notifications.

We've also added a method for handling maintenance and downtime on our hosts and services and proposed a method by which we can learn from alerts we don't plan to action.

In the next chapter, we're going to take all of the elements of our framework

introduced in the book and demonstrate how we manage, measure, and notify on an entire application stack: from host to business metrics.

Chapter 11

Monitoring Tornado: a capstone

In the previous chapters of the book we built the base framework for monitoring. We started with our Riemann server to receive and route our events, added Graphite servers to store metrics, and added Grafana to visualize them. We then installed collectd on our hosts (including Docker containers) to allow us to collect metrics and deliver them to Riemann. We also added support for collecting logs via Syslog and feeding them into the ELK stack. We looked at how we could instrument an application to better understand its state and performance. Lastly, we updated our notifications and added some new notification destinations. That's a lot of moving pieces!

You can see our current framework architecture here:

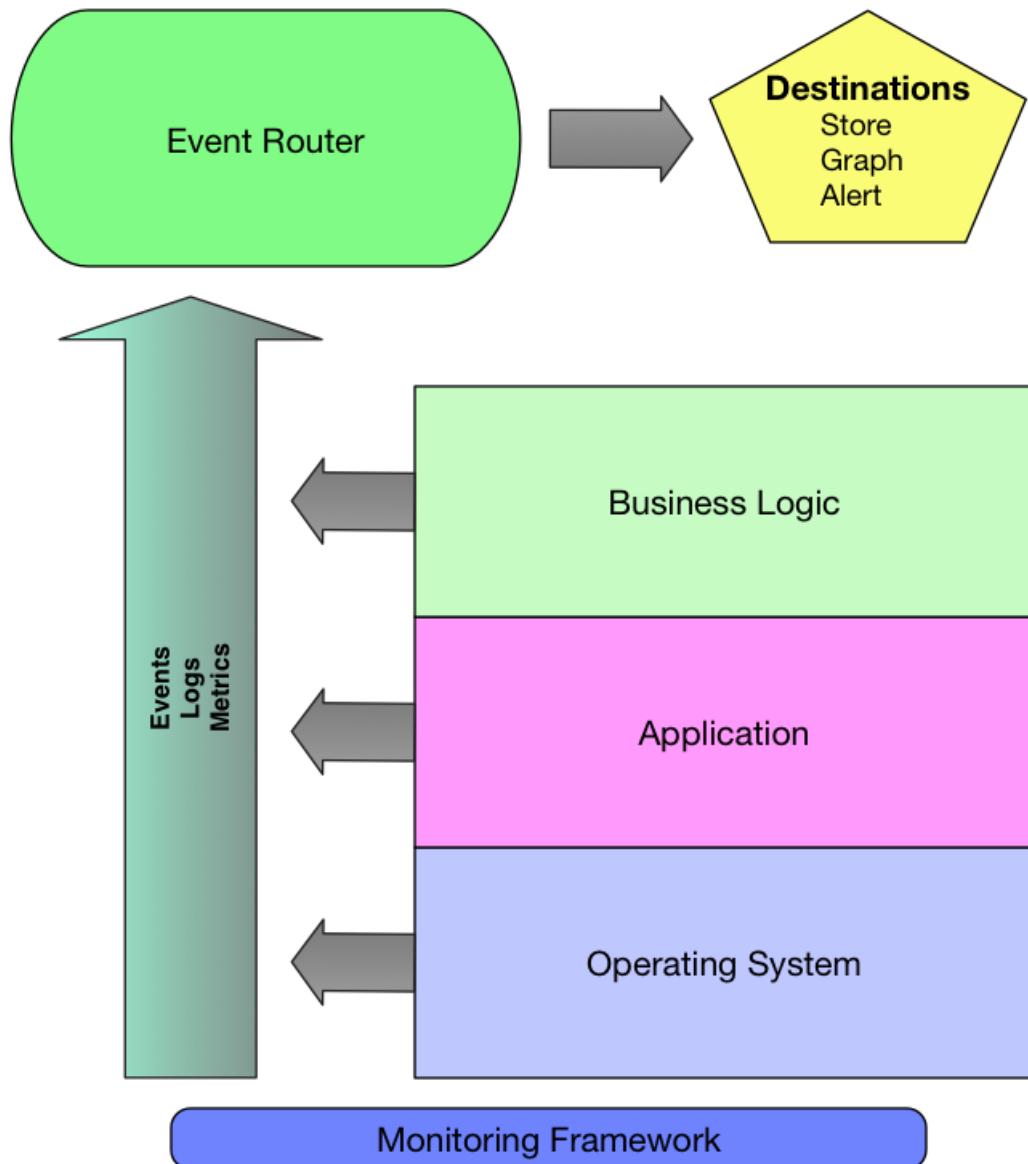


Figure 11.1: Current Monitoring Framework Architecture

In this and subsequent chapters, we're going to put all of those moving pieces together and monitor an application stack: from business metrics to application state to host metrics. The idea behind these chapters is not to definitively show you how to monitor specific tools or services, but rather to demonstrate potential

techniques based on the framework we've just built. We'll provide self-contained and adaptable monitoring techniques for different types of services that can easily be used to monitor similar services or tools. You can then take these techniques, potentially extend them, and use them across your environment to provide monitoring coverage.

To do this we're going to look at a multi-tier, multi-host, multi-service application and use it to show you how to combine everything we've learned in previous chapters. We're going to look at the specific concerns of the application that we're monitoring and demonstrate how to gain insights into those concerns.

The Tornado application

Our application is called Tornado. It's an order processing system that buys and sells items. The business owners of the Tornado application care:

- That items can be bought and sold with Tornado.
- That the Tornado user experience for buyers and sellers is acceptable.
- That they have visibility into the volume and value of items bought and sold.

The engineers who manage the Tornado application have built their concerns from the business owner's priorities:

- That the Tornado application is available, and that they are notified when the Tornado application is not available.
- That at least one Tornado web servers is available as much as possible.
- That there is a 5xx error rate less than 1% on the Tornado web servers.
- That the Tornado application 99th-percentile latency is 100 milliseconds or less.
- that 99th-percentile latency adding items to the Tornado DB is 3 milliseconds or less.

We'll also add some additional checks and metrics that demonstrate more of our monitoring framework.

Now let's take a look at the Tornado application's architecture.

Application architecture

You can see the architecture of the Tornado application here:

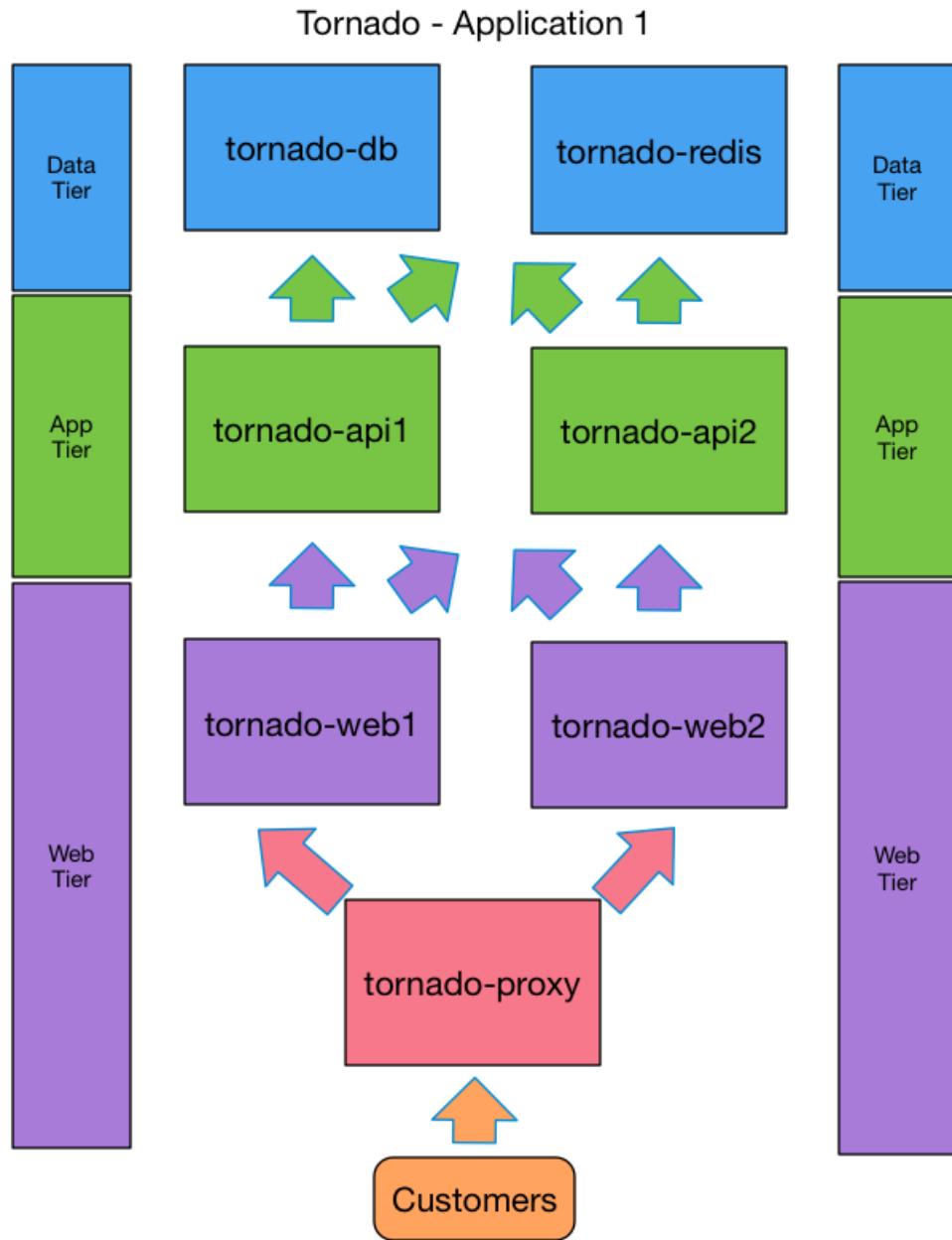


Figure 11.2: Tornado's architecture

The Tornado application is made up of three tiers.

Web tier

The Web tier is made up of three hosts: `tornado-proxy` running HA-Proxy and load balancing two hosts, `tornado-web1` and `tornado-web2`, running Nginx. We'll look at monitoring this tier in this chapter.

Application tier

The Application tier is made up of two hosts: `tornado-api1` and `tornado-api2`. Each host contains a Clojure-based API running on the JVM that is the back end to our hosted website. We'll look at monitoring this tier in Chapter 12.

Data tier

Our Data tier provides the database storage and data stores for our API. It's made up of two hosts: `tornado-db` running MySQL and `tornado-redis` running a Redis database. We'll look at monitoring this tier in Chapter 13.

Monitoring strategy

In the next three chapters we're going to walk through monitoring each tier, starting with the Web tier. We'll cover monitoring hosts, the services like web servers or databases on the hosts, as well as any application components on the hosts.

We'll assume each host has a collectd daemon installed and Syslog configured to deliver to Logstash as we demonstrated in Chapters 5, 6, and 8 respectively.

We're going to propose a small number of initial checks for our application, with a focus on application performance and state. We'll build these checks so that you can easily expand or duplicate the design we're proposing for other applications. These initial checks are just a selection of potential checks for the components in the Tornado application. Your environment may differ or your concerns may

vary. You should use the examples here as starting points rather than merely duplicating them.

TIP We're going to edit a lot of files and change a bunch of settings in the chapter. In your real world environment don't do it manually. Use a configuration management tool to manage your configuration files.

Tagging our Tornado events

In our first step, on all our hosts, let's tell Riemann that these events are for Tornado. In Chapter 5, when we installed the `write_riemann` plugin for collectd, we added some configuration.

Listing 11.1: The `write_riemann` configuration from Chapter 5

```
<Plugin "write_riemann">
  <Node "riemannna">
    Host "riemannna.example.com"
    Port "5555"
    Protocol TCP
    StoreRates true
  </Node>
  Tag "collectd"
</Plugin>
```

Note we specified a `Tag` option to add the tag `collectd` to our Riemann `:tags` field. Let's add another tag to identify the specific application being monitored

on these hosts.

Listing 11.2: Adding a new tag for Tornado

```
<Plugin "write_riemann">
    <Node "riemann">
        Host "riemann.example.com"
        Port "5555"
        Protocol TCP
        StoreRates true
    </Node>
    Tag "collectd"
    Tag "tornado"
</Plugin>
```

Here we've added the tag **tornado** to collectd. We could also add the tag to existing **Tag** option.

Listing 11.3: Adding to an existing Tag

```
Tag "collectd" "tornado"
```

We're going to use a second **Tag** option for ease of readability.

Now when events are generated by collectd and sent to Riemann their **:tags** field will look like:

Listing 11.4: The Riemann :tags field for Tornado

```
:tags [collectd tornado]
```

NOTE This is a host-level tag. You could specify more tags if there were other applications or components that you wished to tag on the host by adding more Tag options.

Monitoring Tornado — Web tier

Our first tier is the Web tier, a load-balanced website using HAProxy and Nginx. For Tornado’s Web tier, what do we care about? Let’s break down our concerns:

- That HAProxy and Nginx are running on our hosts.
- That the site is load balanced, our nodes are up, and that at least two Tornado web servers are available at all times.
- 5xx error rate less than 1% on the Tornado web servers.

We’ll start with configuring the monitoring for each component, HAProxy and then Nginx, and then we’ll look at using the events and metrics generated by that monitoring to address those concerns.

Monitoring HAProxy

The first service we’re going to monitor is [HAProxy](#). HAProxy provides high availability, load balancing, and proxying for TCP and HTTP-based applications. It’s

written in C and, despite its age, is still commonly used by a wide variety of sites and organizations.

In our case HAProxy is configured on the `tornado-proxy` host. We're going to monitor this instance.

HAProxy has several available methods for monitoring its state:

- It can create a Unix socket that can be queried for statistical data.
- It can create an administrative web page that contains statistics and state.
- It can generate logs.

We're going to use the first and last methods to monitor HAProxy: we're going to extract statistics and state from the local Unix socket, and we're going to collect HAProxy's logs and send them to Logstash.

The first method, the Unix socket, outputs a CSV dump of the current state of the running HAProxy daemon. This is essentially a CSV dump of the same data displayed in HAProxy's [statistics and management HTTP console](#).

HAProxy version 1.4.24, released 2013/06/17

Statistics Report for pid 27604

> General process information

General process information																																	
Statistics Report for pid 27604																																	
HAProxy version 1.4.24, released 2013/06/17																																	
Global																																	
pid = 27604 (process #1, nbproc = 1) uptime = 8d 21h3m21s system_uptime = 8d 21h3m21s maxconn = unlimited; ulimit-r = 4015 maxsock = 4015; maxconn = 2000; maxpipes = 0 current conn = 2; current pipes = 0/0 Running tasks: 1/4																																	
Up/Down status																																	
active UP backup UP active DOWN, going down backup DOWN, going up not checked active or backup DOWN for maintenance (MAINT)																																	
Display option:																																	
Halt DOWN servers Primary site Refresh now Updates (v1.4) CSV export Online manual																																	
External resources:																																	
Frontend																																	
state			Queue			Session rate			Sessions			Bytes			Denied			Errors															
Cur			Max			Cur			Max			Cur			Limit			Total															
Frontend			2			2			2			2,000			2			LbTot															
Backend			0			0			0			2,000			0			In															
																		Out															
																		Req															
																		Conn															
																		Req															
																		Retr															
																		Redis															
																		Status															
																		LastChk															
																		Wght															
																		Act															
																		Bck															
																		Chk															
																		Dwn															
																		Downtime															
																		Thrte															
Tornado WWW																																	
Frontend																																	
state			Queue			Session rate			Sessions			Bytes			Denied			Errors															
Cur			Max			Cur			Max			Cur			Limit			Total															
Frontend			0			7			0			5			2,000			196															
																		In															
																		Out															
																		Req															
																		Conn															
																		Req															
																		Retr															
																		Redis															
																		Status															
																		LastChk															
																		Wght															
																		Act															
																		Bck															
																		Chk															

From Chapter 5 we already have collectd installed and configured on this host, and it's sending host-based metrics and events to Riemann. We're going to take advantage of this installation to install a custom plugin that will monitor HAProxy.

Adding the statistics socket

Before we can monitor using our custom plugin we need to enable HAProxy's statistical Unix socket. We do this in HAProxy's configuration on `tornado-proxy`. Usually this is located in the `/etc/haproxy` directory. Let's take a look at the HAProxy configuration file, usually in the `/etc/haproxy/haproxy.cfg` file; for Tornado it's principally the `global` configuration section.

Listing 11.5: The HAProxy configuration

```
global
  log /dev/log    local0
  log /dev/log    local1 notice
  chroot /var/lib/haproxy
  user haproxy
  group haproxy
  daemon
```

The `global` block configures global options for the HAProxy daemon. We need to add a new option called `stats` to this block.

Listing 11.6: Adding the stats option to our HAProxy configuration

```
global
  log /dev/log  local0
  log /dev/log  local1 notice
  chroot /var/lib/haproxy
  user haproxy
  group haproxy
  daemon
  stats socket /var/run/haproxy.sock mode 600
  stats timeout 5s
```

You can see that we've added two lines starting with `stats`. The first line configures our Unix socket at `/var/run/haproxy.sock` and sets the permissions of the socket to `600`. The second line sets a timeout for queries to our socket of five seconds.

If we then restart HAProxy:

Listing 11.7: Restarting the HAProxy service

```
$ sudo service haproxy restart
```

We should see the socket available in the `/var/run` directory.

Listing 11.8: The /var/run/haproxy.sock socket

```
james@tornado-proxy:/var/run# ls -l /var/run/haproxy.sock
srw----- 1 root root 0 Mar 29 23:22 /var/run/haproxy.sock
```

Now we'll configure collectd to read from this socket.

Installing the HAProxy plugin

We saw in Chapter 5 that collectd uses read and write plugins to collect and write metrics. This includes community-contributed plugins that collectd can execute using helper plugins. We saw one of these when we enabled Docker monitoring in Chapter 7. In this case our [HAProxy plugin](#) is another Python plugin that we'll execute using the collectd [python](#) plugin.

Let's start by grabbing our plugin. We'll create a directory to hold it, [/usr/lib/collectd/haproxy](#), and then download the plugin into it.

Listing 11.9: Download the haproxy.py plugin

```
$ mkdir /usr/lib/collectd/haproxy
$ cd /usr/lib/collectd/haproxy
$ wget https://raw.githubusercontent.com/jamtur01/collectd-
haproxy/master/haproxy.py
```

NOTE Credit for the original plugin goes to [Michael Leinartas](#), [German Gutierrez](#), and [SignalFx](#).

Here we've created the directory, changed into the `/usr/lib/collectd/haproxy` directory (on Red Hat this would be the `/usr/lib64/collectd/haproxy` directory). We've then downloaded the plugin from GitHub. You can take a look at its source. It defines a list of metrics and some configurable options. It also has a series of methods that reads the socket and parses the metrics.

TIP You can learn about writing your own Python plugins on the [collectd Python man page](#).

Configuring the HAProxy plugin

Now that we've installed the plugin we need to tell collectd about it and configure it. To do that we're going to add a configuration file for the plugin to the `/etc/collectd.d` directory called `haproxy.conf`.

Listing 11.10: Creating a haproxy.conf configuration file

```
$ sudo vi /etc/collectd.d/haproxy.conf
```

Let's populate that file.

Listing 11.11: The haproxy.conf configuration file

```
<LoadPlugin python>
    Globals true
</LoadPlugin>

<Plugin python>
    ModulePath "/usr/lib/collectd/haproxy/"

    Import "haproxy"

    <Module haproxy>
        Socket "/var/run/haproxy.sock"
    </Module>
</Plugin>

LoadPlugin processes
<Plugin "processes">
    Process "haproxy"
</Plugin>
```

Our configuration file loads the `python` plugin. The `Globals true` line enables any Python standard libraries on our host so we can use them in our plugins.

Next we configure the `python` plugin. We tell collectd where to find our plugins, here `/usr/lib/collectd/haproxy` (again this would be `/usr/lib64/collectd/haproxy` on Red Hat). We then import the `haproxy` plugin we just installed. This loads the plugin into collectd.

Then we configure the plugin itself. We tell it where to find the HAProxy socket, `/var/run/haproxy.sock`, that we created earlier.

Lastly, we've configured the `processes` plugin that we added in Chapter 5. The `processes` plugin monitors processes in general on the host but can also be configured to drill down and monitor specific processes in more detail. In this case we've configured it to match the `haproxy` process.

Once we have our configuration in place we enable it by restarting collectd.

Listing 11.12: Restarting the collectd daemon for HAProxy

```
$ sudo service collectd restart
```

Adjusting the HAProxy metric names in Riemann

We should now see our HAProxy events coming through to our Riemann server. Let's look at one of them.

Listing 11.13: HAProxy events in Riemann

```
{:host tornado-proxy, :service haproxy/derive-backend.stats.  
  response_4xx, :state ok, :description nil, :metric 0, :tags [  
    collectd tornado], :time 1453656413, :ttl 60.0, :ds_index 0, :  
    ds_name value, :ds_type derive, :type_instance backend.stats.  
  response_4xx, :type derive, :plugin haproxy}
```

We see that the event is tagged with `collectd` and `tornado`. It's coming from our `tornado-proxy` host and, in this case, our `:service` field is:

`haproxy/derive-backend.stats.response_4xx`

Which translates into: 4xx HTTP response codes from the back end. With our current configuration this metric will be going onto Graphite and being graphed.

And, like our other collectd metrics, the metric name may be a little hard to parse. Let's use the collectd metric name rewriting we built in Chapter 6 to make this a little more palatable. Edit the `/etc/riemann/examplecom/etc/collectd.clj` file on our Riemann host and add a new service rewrite to the list.

Listing 11.14: Updating HAProxy metrics in Riemann

```
(def default-services
  [ {:service #"^load/load/(.*)$" :rewrite "load $1"}
    . . .
    {:service #"^haproxy/(gauge|derive)-(.*)$" :rewrite "haproxy
$2"}])
```

We've added an entry at the bottom of the `default-services` var: a rewrite of our HAProxy metric names. It uses a regular expression to capture the `gauge` or `derive` statements from the front of our metric and then rewrite the remaining path via a capture. Now the metric with the `:service` field of:

`haproxy/derive-backend.stats.response_4xx`

Will become:

`haproxy.backend.stats.response_4xx`

Which is much easier to navigate. We'll need to restart or reload Riemann to activate this updated rewriting.

Listing 11.15: Reloading Riemann for HAProxy rewrites

```
$ sudo service riemann reload
```

Collecting HAProxy logs

By default, HAProxy logs to Syslog. This is configured inside the `/etc/haproxy/haproxy.cfg` file on most distributions.

Listing 11.16: The HAProxy log configuration

```
global
    log /dev/log    local0
    log /dev/log    local1 notice
    . . .
```

We see that there is a `global` configuration option, `log`, which will send all events to `/dev/log`, which in turn will write them to Syslog.

We're going to make a slight adjustment to this configuration to help identify the Tornado application's specific events. To do this we're going to modify the Syslog program name sent by HAProxy. This normally defaults to `haproxy` but we're going to change that with the `log-tag` configuration directive in our `haproxy.cfg` file. Let's update that file now.

Listing 11.17: The updated HAProxy log configuration

```
global
    log /dev/log    local0
    log /dev/log    local1 notice
    log-tag tornado-haproxy
    . . .
```

Now we need to restart HAProxy to enable this change.

Listing 11.18: Restarting HAProxy to change log tags

```
$ sudo service haproxy restart
```

TIP HAProxy's log directive also allows us to send events directly via a Syslog daemon, for example: `log logstash.example.com:5514`. We could hence send the events, without going through the local Syslog, directly to Logstash if we wished.

After Chapter 8, RSyslog has been happily sending these log entries onto Logstash. On the Logstash end we're not currently doing anything special to parse them, so they are just being treated by `grok` as standard Syslog messages.

Let's change that by parsing those events in Logstash to extract the useful context from them. HAProxy has quite a complexly structured log format, called the [HTTP log format](#). Let's look at a typical HAProxy event.

Listing 11.19: HAProxy log entry

```
Jan 18 23:21:03 tornado-proxy tornado-haproxy[27604]:  
66.108.110.85:57896 [18/Jan/2016:23:21:03.239] tornado-www  
tornado-web/tornado-web2 218/0/1/1/220 200 447 - - -  
2/2/0/1/0 0/0 "GET / HTTP/1.1"
```

We see the event is prefixed with the typical components of a Syslog message: date, host, program (now our updated `tornado-haproxy` instead of the default `haproxy`), PID. It then contains, from HAProxy, HTTP connections or similar events such as

the source IP address and port, the front end and back end that responded to the request, HTTP status codes, and other data.

This event is then sent to Logstash, and we're going to parse out the components by adding a specific `grok` filter to our Logstash configuration. Let's do that now. On the `logstash` host let's open up our `/etc/logstash/conf.d/logstash.conf` configuration file and edit our `filter` section.

Listing 11.20: The HAProxy updated Logstash configuration

```

filter {
  if [type] == "syslog" {
    grok {
      match => { "message" => "(?:(?:SYLOGTIMESTAMP:
        syslog_timestamp}|%{TIMESTAMP_ISO8601:syslog_timestamp}
        %{SYLOGHOST:syslog_hostname} %{DATA:syslog_program
        }(?:\\/{:container_name}\\/{:container_id})
        (?:(?:\\[%{POSINT:syslog_pid}\\])?: %{GREEDYDATA:
        syslog_message}" )
      remove_field => ["message"]
    }
    syslog_pri { }
    date {
      match => [ "syslog_timestamp", "MMM d HH:mm:ss", "MMM dd
        HH:mm:ss", "ISO8601" ]
    }
    if [syslog_program] == "tornado-haproxy" {
      grok {
        match => ["syslog_message", "%{HAPROXYHTTPBASE}"]
        remove_field => ["syslog_message"]
        add_field => { "tags" => "tornado" }
      }
    }
  }
}

```

We see our existing `if` conditional that checks if an event `type` is `syslog` and then

parses out the Syslog message. This results in a new event with a number of fields prefixed with `syslog_`.

One of these fields, `syslog_message`, contains the event message minus the Syslog-prefixed components like date, program, and PID. We've added another `if` conditional inside our previous conditional because we want to take advantage of our existing Syslog parsing that selects events based on the contents of the `syslog_program` field created by our parsing. Here we're matching on `tornado -haproxy`. Any events that match are passed to our second `grok` filter. In this filter we're using the `match` attribute to match on our `syslog_message` field. To parse out and extract the components from the event, we're lucky that we have an existing `grok` pattern, installed with Logstash. That pattern, `HAPROXYHTTPBASE`, will parse the event and return a new event that will look something like this:

Listing 11.21: Our new parsed HAProxy event

```
{  
    . . .  
    "client_ip" => "66.108.110.85",  
    "client_port" => "57896",  
    "accept_date" => "18/Jan/2016:23:21:03.239",  
    "haproxy_monthday" => "18",  
    "frontend_name" => "tornado-www",  
    "backend_name" => "tornado-web",  
    "time_request" => "218",  
    "time_backend_connect" => "1",  
    "time_backend_response" => "1",  
    "time_duration" => "220",  
    "http_status_code" => "200",  
    "bytes_read" => "447",  
    "captured_request_cookie" => " - ",  
    "captured_response_cookie" => " - ",  
    "termination_state" => "----",  
    "actconn" => "2",  
    "retries" => "0",  
    "srv_queue" => "0",  
    "backend_queue" => "0",  
    "http_verb" => "GET",  
    "http_request" => "/",  
    "http_version" => "1.1"  
    "tags" => "tornado"  
}
```

NOTE Our grok filter will also remove the `syslog_message` field from the event, as it now serves no purpose and can be discarded to save space and tidy up after ourselves.

Note that we've added a field called `tags` with a value of `tornado`. This will be useful later when we want to identify Tornado events.

The updated event now contains much of the data we'd need to diagnose or identify any specific issues. From here the event will go into Elasticsearch to be indexed, and then would be available for searching or graphing in Kibana.

We can also use our Logstash HAProxy events in Riemann. Let's send our HAProxy events to Riemann using the `riemann` output we introduced in Chapter 8. To do this we need to construct the specific event we're sending to Riemann in our Logstash `output` section. Let's open our `/etc/logstash/conf.d/logstash.conf` configuration file on our Logstash server and the `riemann` output plugin.

Listing 11.22: Sending HAProxy events to Riemann

```
    . . .
output {
  if [syslog_program] == "tornado-haproxy" {
    riemann {
      host => "riemann"
      sender => "%{syslog_hostname}"
      map_fields => true
      riemann_event => {
        "service"      => "tornado.proxy.request"
        "metric"       => "%{time_duration}"
        "state"        => "ok"
      }
    }
  }
}

    . . .
}
```

We've selected any events with a `syslog_program` of `tornado-haproxy` using an `if` conditional. We're passing these events to the `riemann` plugin. We specify the target Riemann server, here `riemann`, and specify the `sender` directive to set the value of the `:host` field of our Riemann event. We're setting it to the `syslog_hostname` field of our Logstash event.

We've also used the `map_fields` directive, which automatically maps Logstash fields to Riemann fields of the same name. For example, the `http_status_code` field in Logstash would become the `:http_status_code` field in the Riemann event. This might be overkill in many cases: this maps all fields. We could in-

stead only set specific fields in the `riemann_event` directive. We use it here to set the `:service` field to `tornado.proxy.request` and map the `:metric` field to the `time_duration` field (notice we wrap fields in `%{field}` to identify them as part of an event). The `time_duration` field contains the full length of time a request took to process in HAProxy. We can then use this value to graph or notify on HAProxy response time. We also set the `:state` field to `ok`.

TIP You can find the full breakdown of the [HAProxy HTTP log format](#) in the HAProxy documentation.

If we now restart Logstash we should start to see HAProxy events in Riemann.

Listing 11.23: A HAProxy event in Riemann

```
{:host tornado-proxy1, :service tornado.proxy.request, :state ok,
  :description nil, :metric 1.0, :tags tornado, :time
1453995163, :ttl 60, :frontend_name tornado-www, :time_queue 0,
  :srvconn 1, :termination_state ----, :haproxy_month Jan, :
  http_version 1.1, :captured_response_cookie -, :
  syslog_severity_code 5, :client_port 56855, :bytes_read 447, :
  time_duration 1, :time_backend_response 1, :syslog_facility
user-level, :retries 0, :client_ip 85.17.156.11, :
  haproxy_monthday 28, :time_request 0, :haproxy_time 15:32:43, :
  type syslog, :actconn 1, :syslog_timestamp Jan 28 15:32:43, :
  port 58985, :srv_queue 0, :syslog_severity notice, :beconn 0, :
  backend_name tornado-web, :haproxy_hour 15, :http_request /, :
  accept_date 28/Jan/2016:15:32:43.241, :feconn 1, :
  captured_request_cookie -, :haproxy_year 2016, :
  syslog_facility_code 1, :syslog_pid 27604, :syslog_program
tornado-haproxy, :server_name tornado-web1, :haproxy_second 43,
  :http_status_code 200, :backend_queue 0, :haproxy_minute 32, :
  http_verb GET, :syslog_hostname tornado-proxy, :
  time_backend_connect 0, :haproxy_milliseconds 241}
```

As you can see, this is a LOT of fields. We can trim this down by only setting the fields we want in the `riemann_event` directive rather than using the `map_fields => true` directive. We then direct this event to be graphed.

Listing 11.24: Graphing our HAProxy events in Riemann

```
(where (service "tornado.proxy.request")
  (smap rewrite-services graph))
```

Monitoring Nginx

We've monitored HAProxy on the `tornado-proxy` host. Next we want to monitor Nginx on the `tornado-web1` and `tornado-web2` hosts. Similar to HAProxy we're going to monitor Nginx two ways:

- We'll collect metric and status data from our running Nginx daemons.
- We'll collect logs from Nginx and send them onto Logstash.

Like our `tornado-proxy` host we've already installed collectd and configured log collection on these hosts, and we're again going to make use of these existing installations.

Adding the Nginx status page

For collecting metrics from the Nginx daemons we make use of the Nginx status page. The Nginx status page provides metrics and state about the currently running Nginx daemons. We then have collectd use a plugin to gather data from that page.

The Nginx status page is provided by an optional module called:

`ngx_http_stub_status_module`

The status page is an HTTP page that contains data like the following:

Listing 11.25: The Nginx status page

```
Active connections: 291
server accepts handled requests
    16630948 16630948 31070465
Reading: 6 Writing: 179 Waiting: 106
```

The stub status module provides information on active connections, accepted and handled requests, and the volume of client requests. The module is generally compiled and available with Nginx on most distributions. You can confirm it's available for you like so:

Listing 11.26: Confirming the status module is available

```
$ 2>&1 nginx -V | tr -- - '\n' | grep status
http_stub_status_module
```

If you see a line, `http_stub_status_module`, then you have the `stub_status` module available.

You can enable it by specifying a location in your Nginx configuration like so:

Listing 11.27: The Nginx status page

```
location /nginx_status {  
    stub_status on;  
    access_log off;  
    allow 127.0.0.1;  
    deny all;  
}
```

Here we've defined a `location` block called `/nginx_status` (you can call it anything you like). We've enabled the status module with `stub_status on;`. We've also turned off access logging as we don't want our visits to the status page to generate unnecessary log entries. Lastly, we've added some access controls to only allow access from `127.0.0.1` to prevent people not on the host from querying the status page. The `deny all` blocks any other access. We could also enable [Nginx's basic authentication](#).

To enable the status page we need to reload or restart Nginx.

Listing 11.28: Restarting the Nginx service

```
$ sudo service nginx restart
```

We then view our status page on one of our web servers, for example `http://tornado-web1/nginx_status`, and we'll see metrics like this:

Listing 11.29: The Nginx status page

```
Active connections: 3
server accepts handled requests
    3 3 2
Reading: 0 Writing: 1 Waiting: 2
```

Enabling the Nginx collectd plugin

With our status page enabled we now configure collectd to scrape the page. We do this using the [Nginx plugin](#). The Nginx plugin is installed by default on both Ubuntu and Red Hat distributions when you install collectd. Let's load and configure the plugin now. To do that we're going to add a configuration file for the plugin to the `/etc/collectd.d` directory called `nginx.conf`.

Listing 11.30: Creating a nginx.conf configuration file

```
$ sudo vi /etc/collectd.d/nginx.conf
```

Now let's populate that file.

Listing 11.31: The nginx.conf collectd configuration

```
LoadPlugin nginx

<Plugin nginx>
    URL "http://127.0.0.1/nginx_status"
</Plugin>

LoadPlugin processes
<Plugin "processes">
    Process "nginx"
</Plugin>
```

Here we've loaded the `nginx` plugin. We've also configured it by specifying the URL at which it can find the Nginx status page. It should match the location we configured above.

We've also added our `processes` plugin configuration for the `nginx` process so we can drill down on it and use it to ensure Nginx is running.

Once we have our configuration in place we enable it by restarting collectd.

Listing 11.32: Restarting the collectd daemon for the Nginx plugin

```
$ sudo service collectd restart
```

We should now see Nginx events from our two web servers flowing into Riemann, for example:

Listing 11.33: An Nginx plugin event in Riemann

```
{:host tornado-web1, :service nginx/connections-accepted, :state
ok, :description nil, :metric 306, :tags [collectd tornado], :
time 1453658444, :ttl 60.0, :ds_index 0, :ds_name value, :
ds_type derive, :type_instance accepted, :type connections, :
plugin nginx}
```

This shows us the Nginx connections accepted from the `tornado-web1` host. Like with our HAProxy metrics, we could rewrite the path here. However it's usually fairly readable so we're going to skip that rewrite in this case.

TIP If you're using Apache you can take a similar approach with the `mod_status` module. You can find out more in the [Apache mod_status documentation](#). There's also a corresponding [Apache plugin for collectd](#).

Logging Nginx events

Like our HAProxy events, we also get [Nginx's log events](#) from our web servers. There are two types of standard Nginx logs:

- Access logs — Logs showing incoming requests.
- Error logs — Logs showing Nginx server and application errors.

Your access log files have a predefined format controlled in your Nginx configuration by the `log_format` directive. The default format is called `combined`. It

contains most of the information we'd likely need to debug issues: sources and destinations, request paths, HTTP status codes, and request sizes.

Error logs are controlled using the `error_log` directive. You can't adjust the format, but you can dial up and down the detail level of your logging.

Unlike HAProxy, Nginx doesn't log to Syslog by default. On most distributions, though, Nginx writes logs into the `/var/log/nginx` directory, specifically by placing access logs into the `/var/log/nginx/access.log` and Nginx's error logs into `/var/log/nginx/error.log`. Nginx can also be configured to output logs for specific sites or locations, and the examples here can be extended to support that.

TIP You could also configure [Nginx to log to Syslog](#) if you wish.

As Nginx's current logs aren't going into Syslog, we need to collect them via RSyslog's `imfile` module. We talked about the module in Chapter 8. The `imfile` module isn't loaded by default but we'll load and configure it by adding a new configuration file. We'll create a configuration to collect Nginx access and error logs now.

To start, we create a file in the `/etc/rsyslog.d/` directory to configure our log collection. RSyslog loads configuration files in alphanumeric order. We normally prefix any file with a number, `00-filename`, to help specify a load order. For example, we discover in Chapter 8 that the default RSyslog configuration on Ubuntu loads in a file called `50-default.conf`. Let's create and populate a file called `35-nginx.conf` to hold our Nginx configuration.

We first create the file.

Listing 11.34: Create the nginx log configuration file

```
$ sudo touch /etc/rsyslog.d/35-nginx.conf
```

Then populate it.

Listing 11.35: The RSyslog Nginx configuration

```
module(load="imfile" PollingInterval="10")

input(type="imfile"
      File="/var/log/nginx/access.log"
      StateFile="nginx_access"
      Tag="tornado-nginx-access:"
      Severity="info"
      Facility="local7")

input(type="imfile"
      File="/var/log/nginx/error.log"
      StateFile="nginx_error"
      Tag="tornado-nginx-error:"
      Severity="error"
      Facility="local7")
```

In our first step we load the `imfile` module and tell it to poll the listed files. RSyslog will poll any listed files once every 10 seconds, set by the `PollingInterval` directive.

Each file we wish to collect events from is then listed in an `input` block. The

`input` block has a `type`, here `imfile`, and a series of directives. The `File` directive specifies the file we're collecting log events from. It should be the fully qualified path. The `StateFile` is a file that keeps track of where RSyslog is in the file. It should have a unique name: if you name two state files with the same name, expect weird clashes of state to occur. You don't need to specify a path—RSyslog will stash it for you, usually in `/var/spool/rsyslog`.

Next, we specify a `Tag` directive that will label our events. We will make use of this in Logstash to filter our events. Note the `:` at the end of the tag. This is important to correctly parse the tag. Lastly, we specify the Syslog `Severity` and `Facility` of our events. This will control how Syslog will process the incoming events. We use a severity of `info` for our Nginx access logs and `error` for our Nginx error logs. We use a `Facility` of `local7`. The `local0` to `local7` facilities are reserved for user purposes to log specific daemons or applications. In most normal distributions, the `local0` to `local7` are generally not processed, so they won't generally be processed or written into a file in the `/var/log/` directory.

TIP This assumes an RSyslog version later than 6. Earlier versions used [a different configuration file format](#).

We next restart RSyslog on the `tornado-web1` and `tornado-web2` hosts.

Listing 11.36: Restarting the RSyslog service for Nginx

```
$ sudo service rsyslog restart
```

This will start our collection of Nginx access and error logs. They'll be shipped to Logstash as traditional Syslog events. A typical Nginx access log entry looks like:

Listing 11.37: A Nginx access log entry

```
45.55.148.142 - - [21/Jan/2016:05:01:44 +0000] "GET / HTTP/1.1"
200 219 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_3)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.111
Safari/537.36"
```

It contains the source IP, HTTP verb, path, status codes, and a variety of other useful information. Let's update our Logstash configuration to parse these events and extract some useful information.

Listing 11.38: Updated Nginx Logstash configuration

```
}

filter {
    if [type] == "syslog" {

        . . .

        if [syslog_program] == "tornado-haproxy" {
            grok {
                match => ["syslog_message", "%{HAPROXYHTTPBASE}"]
                remove_field => ["syslog_message"]
                add_field => { "tags" => "tornado" }
            }
        }

        if [syslog_program] == "tornado-nginx-access" {
            grok {
                patterns_dir => "/etc/logstash/patterns"
                match => { "syslog_message" => "%{NGINXACCESS}" }
                remove_field => ["syslog_message"]
                add_field => { "tags" => "tornado" }
            }
        }
    }
}
```

We've added another `if` conditional, again inside the parent conditional, to take advantage of the Syslog processing. We match on the `syslog_program` field and specifically look for the `tornado-nginx-access` as the Syslog program. This value

is provided by setting the `Tag` directive in our `imfile` configuration. Inside our `grok` filter we've specified a new attribute, `patterns_dir`. This specifies a directory in which we store patterns we've developed ourselves. We're doing this because, unlike HAProxy, there isn't a standard pattern for parsing Nginx access logs, and we'll need to create our own. Let's create that directory now on the `logstash` host.

Listing 11.39: Make the patterns directory

```
$ sudo mkdir /etc/logstash/patterns
```

Now let's create and populate a file called `nginx` in that directory with our Nginx access log pattern.

Listing 11.40: The nginx pattern

```
NGINXACCESS %{IPORHOST:remote_addr} - %{USERNAME:remote_user}
\[ %{HTTPDATE:time_local}\] "%{WORD:http_method} %{URIPATHPARAM:
http_request} HTTP/%{NUMBER:http_version}" %{INT:http_status}
%{INT:body_bytes_sent} %{QS:http_referer} %{QS:http_user_agent}
```

Patterns are listed in the file with a name, here `NGINXACCESS`, and then the pattern itself. Our pattern matches against our Nginx access log entries and pulls out useful data like the remote IP address, the method and request, and the status. It'll turn our Nginx access log into something like this:

Listing 11.41: The parsed Nginx access log event

```
{  
    "syslog_hostname" => "tornado-web2",  
    "syslog_program" => "tornado-nginx-access",  
    . . .  
    "remote_addr" => "45.55.148.142",  
    "remote_user" => "- ",  
    "time_local" => "21/Jan/2016:05:01:44 +0000",  
    "method" => "GET",  
    "request" => "/",  
    "httpversion" => "1.1",  
    "status" => "200",  
    "body_bytes_sent" => "219",  
    "http_referer" => "\"-\"",  
    "http_user_agent" => "\"Mozilla/5.0 (Macintosh; Intel  
    Mac OS X 10_11_3) AppleWebKit/537.36 (KHTML, like  
    Gecko) Chrome/47.0.2526.111 Safari/537.36\""  
    "tags" => "tornado"  
}
```

NOTE Our grok filter will also remove the `syslog_message` field from the event, as it now serves no purpose and can be discarded to save space and tidy up after ourselves.

We see that the updated event now contains much of the data we'd need to diagnose or identify any specific issues with Nginx. From here the event will go into

Elasticsearch to be indexed, and then would be available for searching or graphing in Kibana. Or, like our HAProxy events, we could send it through to Riemann. Here's an example configuration for the `riemann` output plugin.

Listing 11.42: Sending Nginx logs to Riemann

```
...
output {
    if [syslog_program] == "tornado-nginx-access" {
        riemann {
            host => "riemann"
            sender => "%{syslog_hostname}"
            map_fields => true
            riemann_event => {
                "service" => "tornado.web.request"
                "metric"  => "%{body_bytes_sent}"
                "state"   => "ok"
            }
        }
    }
}
...
}
```

Here we're selecting our Nginx log events based on a `syslog_program` value of `tornado-nginx-access`. We're passing them to our `riemann` server via the `host` directive. We've specified the source host for the Riemann `:host` field from the `syslog_hostname` field. We've used the `map_fields` directive to map all possible fields from the Nginx log entries, but we could instead specify selective fields in

the `riemann_event` directive. We've set the value of the Riemann `:service` field to `tornado.web.request` and the value of the `:metric` field to the `body_bytes_sent` field. This field contains the total bytes sent as part of the request. We've also set the `:state` field to `ok`.

If we now restart Logstash we should start to see HAProxy events in Riemann.

Listing 11.43: A Nginx event in Riemann

```
{:host tornado-web1, :service tornado.web.request, :state ok, :  
description nil, :metric nil, :tags tornado, :time 1454008963,  
:ttl 60, :http_version 1.1, :syslog_severity_code 5, :  
http_status 200, :http_user_agent "Pingdom.com_bot_version_1.4_  
(http://www.pingdom.com)", :syslog_facility user-level, :type  
syslog, :syslog_timestamp Jan 28 19:22:43, :port 57762, :  
syslog_severity notice, :http_request /, :remote_addr  
45.55.148.142, :http_method GET, :syslog_facility_code 1, :  
syslog_program tornado-nginx-access, :body_bytes_sent 219, :  
http_referer "-", :remote_user -, :syslog_hostname tornado-web1  
, :time_local 28/Jan/2016:19:22:43 +0000}
```

You can see this is a LOT of fields. We can trim this down by only setting the fields we want in the `riemann_event` directive. We then direct this event to be graphed by extending the `where` stream we used for our HAProxy events.

Listing 11.44: Graphing our HAProxy events in Riemann

```
(where (or (service "tornado.proxy.request") (service "tornado.  
web.request"))  
(smap rewrite-services graph))
```

Now Nginx events will be passed through Riemann into Graphite.

NOTE We're not grok'ing our Nginx error logs. They are most useful stored for diagnostics, and their format is highly variable. We're just storing them directly in Elasticsearch and making them available for search.

Addressing the Web tier monitoring concerns

Now that we have HAProxy and Nginx monitored and sending data, what can we do with that data? We identified a series of concerns for Tornado's Web tier earlier. Those concerns were:

- That HAProxy and Nginx are running on our hosts.
- That at least two Tornado web servers are available at all times.
- The 5xx error rate is less than 1% on the Tornado web servers.

Let's look at how we might address these concerns.

The first concern, that HAProxy and Nginx are running on our hosts, we get for free thanks to Chapters 5 and 6. In those chapters we configured the collectd **process** plugin to send notifications when any process being monitored falls below the

threshold, in our case a minimum of one process per host. We also get part of our second concern, that our nodes are up, as a result of process check. We could tweak this specifically to add thresholds for a specific process—for example, we could tweak the number of Nginx processes. Let's update our `/etc/collectd.d/nginx.conf` configuration file to do this now.

Listing 11.45: Matching multiple Nginx processes

```
    . . .

LoadPlugin processes
<Plugin "processes">
    Process "nginx"
</Plugin>

<Plugin "threshold">
    <Plugin "processes">
        Instance "nginx"
        <Type "ps_count">
            DataSource "processes"
            WarningMin 4
            FailureMin 1
        </Type>
    </Plugin>
</Plugin>
```

We've specified that a warning notification will be generated if the `nginx` process count drops below four, and to send a failure notification if the count drops below one process.

These notifications will be processed in Riemann via our existing logic. Remember

we have a wrapper around `collectd`-tagged events and inside that wrapper we further select all events tagged with `notification`. This code is below.

Listing 11.46: Added collectd monitoring to our Riemann configuration

```
    . . .
    (tagged "collectd"
        (tagged "notification"
            (changed-state {:init "ok"})
                (adjust [:service clojure.string/replace #"\^processes
                    -(.*)\/ps_count\$" "$1"]
                    (page))))
    . . .
    (where (and (expired? event)
        (service #"\^processes-.+\/ps_count\//processes
            "))
        (adjust [:service clojure.string/replace #"\^processes-
            (.*)\/ps_count\//processes\$" "$1"]
            (page)))
    . . .
```

Let's look at our notification logic. Events are checked for a change state, with a default of `ok`, and if they are in the new state then the `:service` field is adjusted to grab the process being monitored. The resulting event is paged out to the PagerDuty function, `page`, that we created in Chapter 10.

Note our `expired` event logic, which detects when the `processes` plugin metrics disappear from the Riemann index. We again adjust the event to grab the process name and page out.

TIP Remember, as with our default configuration, every metric sent from collectd will be sent to Graphite and graphed. We don't need to do anything explicit for that to happen.

Setting up the Tornado checks in Riemann

Now let's define some checks on our Tornado events. As we're conscious of keeping our configuration neat and self-contained we're going to define our new streams in a separate namespace and include it in our main configuration.

Let's start by building some basic logic for our event notifications. We're going to create a new file, `tornado.clj` in a new directory, `/etc/riemann/examplecom/app`. This file will hold the configuration for our Tornado monitoring. Now let's create our new directory and the `tornado.clj` file.

Listing 11.47: Creating the examplecom directory and tornado.clj file

```
$ sudo mkdir -p /etc/riemann/examplecom/app  
$ sudo touch /etc/riemann/examplecom/app/tornado.clj
```

We'll populate this file with some monitoring streams. We're going to create a core function to divide events into each tier. We'll call that function `checks`. We'll also create a function for each tier, starting with the Web tier, that will perform checks for hosts and services in that tier.

Let's look at those initial streams now. We're going to look at a subset of the file initially, but you can see the full file in [the book's source code](#).

Listing 11.48: The `tornado.clj` file

```
(ns examplecom.app.tornado
  "Monitoring streams for Tornado"
  (:require [riemann.config :refer :all]
            [clojure.tools.logging :refer :all]
            [riemann.folds :as folds]
            [riemann.streams :refer :all]))  
  
.  
  
(defn checks
  "Handles events for Tornado"
  []
  (let [web-tier-hosts #"tornado-(proxy|web1|web2)"
        app-tier-hosts #"tornado-(api1|api2)"
        db-tier-hosts #"tornado-(db|redis)"]  
  
  (splitp re-matches host
          web-tier-hosts (webtier)
          app-tier-hosts (apptier)
          db-tier-hosts (datatier)
          #(info "Catchall - no tier defined" (:host %) (:service %))))  
  )
```

You can see we've added a new function and new namespace called `examplecom.app.tornado`. We've called our library or application `app` because that's broadly its function: application monitoring. Finally, we've called the group of function: `tornado`. Literally Example.com Application Tornado. This matches the path we

created above.

TIP You can read more about Clojure namespaces in [this documentation](#).

Next, we've added a documentation string to describe what our namespace will do. In this case, that's providing streams for monitoring the Tornado application. Lastly, we've used the `require` function to include some Clojure and Riemann base functions. From Clojure we included the `clojure.tools.logging` library to be able to log events; the `info` function comes from this library. We also require the `riemann.config`, `riemann.folds` and `riemann.streams` libraries. The `riemann.streams` library contains the `expired` and `where` filtering streams. The `riemann.folds` library contains folds that we'll use later on.

Listing 11.49: Requiring the Riemann functions

```
...  
(:require [riemann.config :refer :all]  
          [clojure.tools.logging :refer :all]  
          [riemann.folds :as folds]  
          [riemann.streams :refer :all]))  
...
```

The `require` function here includes the `riemann.config` and `riemann.streams` libraries, and the `:refer :all` operation loads all functions from these libraries. We also add the `riemann.folds` library to give us access to Riemann's folds streams, and we use the `:as` option to alias the library to `folds`. This gives us access to

the functions we're going to need to work with our Tornado events inside the namespace.

We've defined a new function called `checks` using the `defn` statement. We saw the `defn` function in Appendix A. It combines the `def` statement, which defines variables, and the `fn` statement, which creates functions. It is used to create a function and bind it to a symbol. We then call this symbol to process events.

Inside this initial function we're going to create the first streams we'll use to process Tornado's events.

TIP You can read more about writing functions in Appendix A and the [Riemann HOWTO](#). Also useful is the explanation in the [Clojure from the ground up blog post on functions](#).

Listing 11.50: The `tornado checks` function

```
(defn checks
  "Handles events for Tornado"
  []
  (let [web-tier-hosts #"tornado-(proxy|web1|web2)"
        app-tier-hosts #"tornado-(api1|api2)"
        db-tier-hosts #"tornado-(db|redis)"]

    (splitp re-matches host
      web-tier-hosts (webtier)
      app-tier-hosts (apptier)
      db-tier-hosts (datatier)
      #(:info "Catchall - no tier defined" (:host %) (:service %))))
  )
)
```

Let's first look at the `checks` function. We've defined the arguments this function takes—in our case, none. We've added a documentation string to tell people what this function does.

Next, we've defined a `let` expression. Remember `let` bindings are lexically scoped, i.e., limited in scope to the expression itself. Outside of this expression, any symbols will be undefined. We've defined a series of symbols containing regular expressions that match the hosts that make up our Tornado application. We've broken them into three symbols: `web-tier-hosts`, `app-tier-hosts`, and `db-tier-hosts`. Each symbol contains a regular expression with a list of the hosts in each tier of the application.

TIP We could also query a data store or service discovery like Zookeeper or

PuppetDB for this application architecture.

Inside our `let` expression we only have one further stream: `splitp`. The `splitp` function is a Riemann stream that splits a stream into child streams using a binary predicate and an expression. In our case we're splitting the stream on the `:host` field using a regular expression match.

Listing 11.51: The splitp predicate and expression

```
(splitp re-matches host
```

The match uses the Clojure `re-matches` function, which returns matches based on a pattern. Under the `splitp` function we then specify clauses with test expressions, for example:

Listing 11.52: A splitp clause

```
web-tier-hosts (webtier)
```

Here we're passing a test expression of `web-tier-hosts`, a symbol containing a regular expression list of the hosts in a specific tier of the Tornado application. We defined the symbol in our `let` expression. If the symbol matches the host field, then the event will be sent to the `webtier` function. Our remaining clauses send events to two other functions: `apptier` and `datatier`.

We also need to specify a catchall clause. If we omit a catchall and an incoming event *doesn't* match any clause then Riemann will emit an error. In our case we're sending all events that didn't match our clauses to the log file with a message indicating that we should investigate why a stray event has slipped into our

streams.

Let's look at our `webtier` function next. It'll be defined in the `tornado.clj` file above our `checks` function.

The `webtier` function

The `webtier` function matches any events in our `web-tier-hosts` regular expression, i.e events from the `tornado-proxy`, `tornado-web1`, and `tornado-web2` hosts. These are our HAProxy and Nginx servers. Inside the `webtier` function we've defined two checks that tell us HAProxy and Nginx are functioning in line with expectations.

- The first check alerts us if HAProxy reports less than two back-end servers are available. This lets us know if HAProxy is functioning and can see our web servers.
- The second check reports if the percentage of 5xx HTTP status codes returned is higher than a threshold. This lets us know if our application is running correctly and not generating errors.

Let's look at each check and how it is constructed in turn.

Listing 11.53: The webtier function

```
(defn webtier
  "Checks for the Tornado Web tier"
  []
  (let [active_servers 2.0]
    (sdo
      (where (and (service "haproxy/gauge-backend.tornado-web."
                           active_servers)
                   (< metric active_servers))
              (adjust #(assoc % :service "tornado-web active servers"
                               :type_instance nil
                               :state (condp = (:metric %)
                                               0.0 "critical"
                                               1.0 "warning"
                                               2.0 "ok")))
              (changed :metric {:init active_servers}
                       (slacker))))
      (check_ratio "haproxy/derive-frontend.tornado-www."
                   response_5xx
                     "haproxy/derive-frontend.tornado-www."
                     request_total"
                     "haproxy.frontend.tornado-www.5
                     xx_error_percentage"
                     0.5 1
                  (sdo
                    (changed-state {:init "ok"}
                      (where (state "critical")
                        (page))
                      (where (state "warning"))
Version: v1.0.3 (2c4a7d0) (slacker)))
                     (smap rewrite-service graph)))))))
```

We've first defined a `let` expression to hold a variable: the number of active Nginx servers in our HAProxy configuration. We've hard-coded the number here but we could template it via a configuration management system or pull it from a data store like [Zookeeper](#).

Our first function inside `webtier` is the `sdo` var. The `sdo` var takes a list of functions and submits a copy of an event to each of them. This allows us to send our events to multiple child streams. Each of our checks will be a child stream. This makes structuring our checks easier and ensures that events will be distributed to each.

Our first check is in line with checks we've configured in earlier chapters.

Listing 11.54: Our first webtier check

```
(where (and (service "haproxy/gauge-backend.tornado-web.  
active_servers")  
            (< metric active_servers))  
      (adjust #(assoc % :service "tornado-web active servers"  
              :type_instance nil  
              :state (condp = (:metric %)  
                          0.0 "critical"  
                          1.0 "warning"  
                          2.0 "ok"))  
      (changed-state {:init "ok"}  
                    (slacker))))
```

We use a `where` stream with two conditions. The first is a service called:

`haproxy/gauge-backend.tornado-web.active_servers`

Which tells us how many back ends HAProxy thinks is active. The second is a `:metric` value of less than our `active_servers` variable. The `:service` is generated by our `haproxy` plugin in collectd, and we set the variable in the `let` expression.

We then use the `adjust` var combined with an `assoc` to update our event's fields: we change the service name to `tornado-web active servers`, we remove the `:type_instance` field to tidy up our events, and—importantly—we change the `:state` field. We use a `condp` conditional to set the value of the `:state` field according to the value of the `:metric` field. If no active servers are present, `0.0`, then the state will be `critical`; if only one server is active then the state will be `warning`; and if all servers are active then the state remains `okay`. We then pass this new event into the `changed-state` stream, with a default state of `ok` set. Here's an example of our new event.

Listing 11.55: The tornado checks and notifications function

```
{:host tornado-proxy, :service tornado-web active servers, :state
  ok, :description nil, :metric 2.0, :tags [collectd tornado], :
  time 1453671310, :ttl 60.0, :ds_index 0, :ds_name value, :
  ds_type gauge, :type gauge, :plugin haproxy}
```

If the event's state changes—for example, from `ok` to `warning`—then Riemann will send a notification to Slack.



Riemann bot BOT 10:02 PM

Service tornado-web active servers on host tornado-proxy is in state warning.

See <http://graphitea.example.com:3000/dashboard/script/riemann.js?host=tornado-proxy>

Service nginx on host tornado-web1 is in state critical.

See <http://graphitea.example.com:3000/dashboard/script/riemann.js?host=tornado-web1>

Figure 11.4: Tornado Slack notifications for Nginx and HAProxy

Our second check checks how healthy our application is by identifying the ratio of 5xx HTTP response codes as a percentage of normal requests. Let's look at it now.

Listing 11.56: Our second webtier check

```
(check_ratio "haproxy/derive-frontend.tornado-www.response_5xx"
             "haproxy/derive-frontend.tornado-www.request_total"
             "haproxy.frontend.tornado-www.5xx_error_percentage"
             0.5 1
  (sdo
    (changed-state {:init "ok"})
    (where (state "critical")
           (page))
    (where (state "warning")
           (slacker)))
  (smap rewrite-service graph)))))
```

The check uses a new check abstraction we've called `check_ratio`. We've added this new abstraction to the `checks.clj` file we created in Chapter 6. We'll take a look at it now and come back to our Tornado check.

Listing 11.57: The check_ratio function

```
(defn check_ratio [srv1 srv2 newsrv warning critical & children]
  "Checks the ratio between two events"
  (project [(:service srv1)
            (:service srv2)]
    (smap folds/quotient-sloppy
      (fn [event] (let [percenta (* (float (:metric event)) 100)
                        new-event (assoc event :metric percenta
                                          :service (str newsrv)
                                          :type_instance nil
                                          :state (condp < percenta
                                            critical "critical"
                                            warning "warning"
                                            "ok"))]
        (call-rescue new-event children))))))
```

We've defined a new function, `check_ratio`. The `check_ratio` function compares the ratio of two services. For example, in our Tornado case that's the ratio of 5xx errors to normal responses. It emits a new event containing the ratio as a percentage as the value of the `:metric` field.

The function takes five arguments and optional child streams.

- The first two arguments are `srv1` and `srv2` which are the `:service` fields of two events we wish to compare.
- The `newsrv` argument contains the value of the `:service` field of the new event measuring the ratio.
- The `warning` and `critical` arguments are thresholds that will change the `:state` of the new event to a value of either `warning` or `critical`.

Lastly, like the other checks we defined in `checks.clj`, our check takes child streams—for example, we could send our new event to Graphite to be stored.

Inside the `check_ratio` function we use a stream called `project` to compare our two events. The `project` stream takes a vector of predicate expressions, much like the expressions we'd use in a `where` stream. It then maintains a vector of the one most recent event that matches each predicate. Incoming events, if they match, replace existing events and the vector is passed to any child streams.

In our case our child stream is an `smap` that feeds our events into a fold: `quotient-sloppy`. The `quotient-sloppy` fold divides the `:metric` values of each event in the vector producing a new event. The `-sloppy` extension let's our quotient deal with nil values in metrics.

TIP If you know your metric values will never be nil then you can just use the folds/quotient fold.

We then feed our new event into a function to do some updates to its content.

Listing 11.58: Presenting our new check_ratio event

```
(fn [event] (let [percenta (* (float (:metric event)) 100)
                  new-event (assoc event :metric percenta
                                     :service (str newsrv)
                                     :type_instance nil
                                     :state (condp < percenta
                                                critical "critical"
                                                warning "warning"
                                                "ok"))]
  (call-rescue new-event children)))))
```

Inside our function, which takes an `event`, we define two symbols inside a `let` expression. The first symbol, `percenta`, takes our new event's `:metric` field and converts it into a percentage by multiplying it by `100`. The `quotient-sloppy` fold will also emit our event `:metric` in the form `srv1_value/srv2_value`—for example, `1/10`—so we'll need to coerce our `:metric` field into a `float` first. Our second symbol, `new-event`, uses `assoc` to take our event map and emit a new map with some updated fields. Firstly, we use the value of the `newsrv` argument as the new `:service` field. We then set the `:type_instance` field to `nil` to tidy our event a little. Lastly, we set the value of the `:state` field based on the value of the event's metric after we've calculated the percentage. If the percentage of events is above the `critical` argument threshold then the event will be set to a `:state` of `critical`, and so on.

Finally, our `check_ratio` passes our `new-event` symbol via the `call-rescue` function to any child streams we define.

NOTE We've customized our check to emit percentages. You could configure

it to emit the metric in a variety of formats.

Let's go back to our Tornado checks to see what this looks like:

Listing 11.59: Our second webtier check returned

```
(check_ratio "haproxy/derive-frontend.tornado-www.response_5xx"
             "haproxy/derive-frontend.tornado-www.request_total"
             "haproxy.frontend.tornado-www.5xx_error_percentage"
             0.5 1
(sdo
 (changed-state {:init "ok"})
 (where (state "critical")
        (page))
 (where (state "warning")
        (slacker)))
 (smap rewrite-service graph)))))
```

We see our check calls the `check_ratio` function with two services:

`haproxy/derive-frontend.tornado-www.response_5xx`

And

`haproxy/derive-frontend.tornado-www.request_total`

These are the metrics from the `haproxy` collectd plugin that record the 5xx errors recorded in the `tornado-www` front end and the total requests to this front end. We then pass in the name of the new event `:service` we'd like for our comparison event.

`haproxy.frontend.tornado-www.5xx_error_percentage`

Lastly, we pass into a `warning` and `critical` threshold of 0.5% and 1% respectively. Our `check_ratio` function will compare both metric values and spit out a new event.

Listing 11.60: The haproxy.frontend.tornado-www.5xx_error_percentage event

```
{:host tornado-proxy, :service haproxy.frontend.tornado-www.5xx_error_percentage, :state ok, :description nil, :metric 0.016398819570895284, :tags [collectd tornado], :time 1454109239, :ttl 60.0, :ds_index 0, :ds_name value, :ds_type derive, :type_instance nil, :type derive, :plugin haproxy}
```

Note the new event here with a `:metric` value of `0.016398819570895284`. This indicates a healthy 0.016% of 5xx errors of our total requests and returns a state of `ok`.

The last part of our check decides what to do with our new event. We've again used the `sdo` function to send events to multiple child streams. In this case we've matched in the `:state` of the event, and if it is `warning` (or 5xx responses greater than 0.5% of responses) we'd send a Slack notification. If it's `critical`, or 5xx errors greater than 1% of responses, then we'll send a notification to PagerDuty.

Adding Tornado checks to Riemann

Before we go any further and look at our other tiers, let's see how we'd use our new Tornado checks. To do this we need to add our new `examplecom.app.tornado` namespace to the top of our `riemann.config` configuration file. Let's update it now.

Listing 11.61: Added monitoring.services to Riemann configuration

```
(require 'riemann.client)

. . .

(require '[examplecom.app.tornado :as tornado])
```

We've used the `require` function to add our `examplecom.app.tornado` namespace to the configuration. We've included the quoted `examplecom.app.tornado` namespace with an alias of `tornado`.

TIP While there are solid Clojure reasons why the required namespace is quoted, those reasons are too detailed for the purposes of this book.

We've then referred to this namespace inside a `where` stream that matches `tornado` tagged events.

Listing 11.62: Sending our Tornado events to be checked

```
(tagged "tornado"
  (tornado/checks))
```

Now our Tornado events will flow into the `tornado` namespace and the `checks` function. And with that, our Web tier is being monitored.

Summary

In this chapter we've seen all the pieces of our monitoring framework operating together. We've started our monitoring with the Web tier. We've started monitoring a complex, multi-tier application in a simple and scalable manner. We've looked at ways to monitor the web components of the Tornado application, including HAProxy and Nginx, using collectd plugins. We've monitored the services themselves and their internals using plugins as well as by consuming their log output to produce useful metrics and diagnostic information. We've added a small set of checks in Riemann to notify us when issues occur.

In the next chapter we'll move on to the application tier.

Chapter 12

Monitoring Tornado: Application Tier

In this chapter we're going to monitor Tornado's Application tier. Our Application tier is made up of two API servers running [the Tornado API](#) on top of the Java Virtual Machine (JVM). For Tornado's application tier, what do we care about? Let's break down our concerns:

- That the Tornado API is running on our hosts.
- That the Tornado application 99th-percentile latency is 100 milliseconds or less.
- That the percentage of JVM heap memory used is under a threshold.

NOTE We've also instrumented the Tornado API application in Chapter 9 and our Tornado application servers will be emitting those metrics too.

We'll start with configuring the monitoring for each component, the underlying

JVM and then the Tornado API, and then we'll look at using the events and metrics generated by that monitoring to address our concerns.

Monitoring the Application tier JVM

We're going to start with monitoring the JVM on both API servers. We saw how to monitor JVM applications using the `GenericJMX` plugin executed by the `java` collectd plugin in Chapter 8 when we looked at monitoring Logstash. The `GenericJMX` plugin uses Java Management Extensions or JMX. JMX is a framework that provides monitoring and management capabilities for Java resources. These resources are represented by MBeans, or Managed Beans. The MBean represents a resource running in the JVM, such as an application.

In our case we're going to monitor a basic collection of JVM metrics including memory, memory pools, threads, and garbage collection. To enable the monitoring we need to take two steps:

- Enable JMX on our Tornado API.
- Enable and configure the `java` plugin in collectd.

Let's start with enabling JMX by starting our application with the appropriate flags. We would update our Tornado API to launch something like so:

Listing 12.1: Updating Tornado API to run JMX

```
java -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=8855 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false -jar tornado-api.jar
```

Here we've added the following flags:

Listing 12.2: The JMX enabling options

```
-Dcom.sun.management.jmxremote  
-Dcom.sun.management.jmxremote.port=8855  
-Dcom.sun.management.jmxremote.authenticate=false  
-Dcom.sun.management.jmxremote.ssl=false
```

They enable JMX and configure it to report locally bound to `localhost` on port `8855`. We've disabled authentication and SSL. If you'd like to use usernames and passwords you can find instructions [on the JMX site](#). We could also enable SSL, but it's probably not required when bound to a local host.

Configuring collectd for JMX

We then need to configure a collectd plugin to scrape our endpoint and retrieve the metrics. The collectd daemon comes with a Java-based plugin called [GenericJMX](#) which does exactly this. We run the `GenericJMX` plugin by executing it with the `java` helper plugin.

Let's create a file to hold our plugin configuration.

Listing 12.3: The collectd JMX configuration file for Tornado API

```
$ vi /etc/collectd.d/tornado-api.conf
```

Now let's populate that file. The whole configuration is a little bit large to paste into here, but we've included a sampling to show you how it works. The whole file is available [in the book's source code](#).

Listing 12.4: The tornado-api.conf file

```
LoadPlugin java
<Plugin "java">
JVMARG "-Djava.class.path=/usr/share/collectd/java/collectd-api.
jar:/usr/share/collectd/java/generic-jmx.jar"
LoadPlugin "org.collectd.java.GenericJMX"
<Plugin "GenericJMX">

. . .

<MBean "memory-heap">
ObjectName "java.lang:type=Memory"
InstancePrefix "memory-heap"
<Value>
Type "memory"
Table true
Attribute "HeapMemoryUsage"
</Value>
</MBean>

. . .

<Connection>
ServiceURL "service:jmx:rmi:///jndi/rmi://localhost:8855/
jmxrmi"
Collect "memory_pool"
Collect "memory-heap"
Collect "memory-nonheap"
Collect "gc-count"
Collect "gc-time"
Version: v1.0.3 (2c4a7d0)                                         635
Collect "thread"
Collect "thread-daemon"
</Connection>
</Plugin>
</Plugin>
```

The first line loads the `java` plugin. This is the base plugin that enables all supporting Java code for collectd. We then configure the plugin operation inside a `<Plugin>` block. Inside this block we tell the `java` plugin where to find our `GenericJMX` plugin and then load that plugin.

We then add a second `<Plugin>` block to configure the `GenericJMX` plugin itself. We then load each MBean we'd like to use to collect monitoring data.

We've specified two types of configuration for our `GenericJMX` plugin. As we learned in Chapter 8, MBeans are blocks that define a mapping of attributes to the types used by collectd to generate metrics. An MBean is a managed Java object. It's like a `JavaBean` component but for exposing a management interface through JMX. You can define MBeans for devices, applications, or resources inside applications or the JVM. Each MBean exposes readable or writeable attributes, a set of operations, and a description.

Listing 12.5: An MBean

```
<MBean "memory-heap">
  ObjectName "java.lang:type=Memory"
  InstancePrefix "memory-heap"
  <Value>
    Type "memory"
    Table true
    Attribute "HeapMemoryUsage"
  </Value>
</MBean>
```

Our `MBean` block has a name, here `memory-heap`. This is the name of the MBean definition inside collectd. Next we specify the `ObjectName`, which is the name of the MBean inside JMX. In this case our MBean tracks memory state in the JVM.

We next define the `InstancePrefix`, which is an optional setting that prefixes our MBean values, so each value we define in this case will be prefixed with `memory-heap`. This helps us identify the source of individual values.

Next we define any values we're sourcing from the MBean. We enclose these in `<Value>` blocks. Every MBean needs at least one `<Value>` block. The attributes in the `<Value>` block set the individual fields of our metrics.

We then set the `Type`. In the case of these values the type is both: `memory`. This will help construct the metric type and name. The `Table` option specifies whether our value is a composite value or not. Composite values are collections of data, each of which can be looked up by a key. If `true` then it assumes a composite value. The last option, `Attribute`, is the name of the attribute inside the MBean from which we're going to read our value. In the case of our first value it'll read an attribute called `HeapMemoryUsage`.

So what metrics will we end up with from this `<Value>` block? In this case collectd will look up the `HeapMemoryUsage` attribute, which is a composite attribute made up of multiple values. It'll grab all of these values and prefix them with the `InstancePrefix` and the `Type` and generate some metrics much like:

Listing 12.6: GenericJMX Java memory heap metrics

```
GenericJMX-memory-heap/memory-committed  
GenericJMX-memory-heap/memory-init  
GenericJMX-memory-heap/memory-max  
GenericJMX-memory-heap/memory-used
```

Our collectd instance will then send these metrics onto Riemann.

We then define a `<Connection>` block that controls where the plugin should look to find our JMX server inside the JVM.

Listing 12.7: The Connections block

```
<Connection>
  ServiceURL "service:jmx:rmi:///jndi/rmi://localhost:8855/
    jmxrmi"
  Collect "memory_pool"
  Collect "memory-heap"
  Collect "memory-nonheap"
  Collect "gc-count"
  Collect "gc-time"
  Collect "thread"
  Collect "thread-daemon"
</Connection>
```

Inside our `<Connection>` block we've defined the `ServiceUrl` that points to our JMX instance running on `localhost` on port `8855`. We also specify a series of `Collect` options that tell collectd which MBeans to process for this connection. In this case we're retrieving a whole series, including the `memory-heap` MBean we just saw.

TIP Also worth looking at is [riemann-jmx](#) which directly sends JMX events to Riemann.

Lastly, lets add the `processes` plugin and configure it to monitor for a process containing `-jar tornado-api` with a label of `tornado-api`.

Listing 12.8: Adding tornado-api process monitoring

```
LoadPlugin processes
<Plugin "processes">
  ProcessMatch "tornado-api" "-jar tornado-api"
</Plugin>
```

This links to the process monitoring threshold we set up in Chapter 5 and allows us to get notifications if the Tornado API stops running.

Collecting our Application tier JVM logs

Like our HAProxy and Nginx events, we also want to get our Tornado API logs. In our case the Tornado API uses a logging framework called [timbre](#), much like Java's [Log4j](#) framework. We've configured Timbre to log errors and requests to a file:

`/var/log/tornado-api.log`

We log events with a specific pattern, modeled roughly on a Syslog entry. As our application's current logs aren't going into Syslog, we need to collect them via RSyslog's [imfile](#) module. Let's create a configuration to collect our applications logs now.

To start, we create a file in the [/etc/rsyslog.d/](#) directory to configure our log collection. RSyslog loads configuration files in alphanumeric order by file name. We normally prefix any file with a number, [`00-filename`](#), to help specify a load order. Let's create and populate a file called [`35-tornado-api.conf`](#) to hold our Tornado API configuration.

Listing 12.9: The RSyslog tornado-api configuration

```
module(load="imfile" PollingInterval="10")

input(type="imfile"
      File="/var/log/tornado-api.log"
      StateFile="tornado_api"
      Tag="tornado-api:"
      Severity="info"
      Facility="local7")
```

In our first step we've loaded the `imfile` module and tell it to poll the listed files. Next, we've specified the `Tag` directive that will label our events and that we will use in Logstash to filter our events. Note the `:` at the end of the tag. This is important to correctly parse the tag. Lastly, we've specified the Syslog `Severity` and `Facility` of our events. This will control how the Syslog will process the incoming events. We'll use a severity of `info` for our application logs, and a `Facility of local7`.

TIP Remember from Chapter 11 that this assumes an RSyslog version later than 6. Earlier versions used [a different configuration file format](#).

We next restart RSyslog on the `tornado-api1` and `tornado-api2` hosts.

Listing 12.10: Restarting the RSyslog service for Tornado API

```
$ sudo service rsyslog restart
```

This will start our collection of our application logs. They'll be shipped to Logstash via the RSyslog forwarding configuration we built in Chapter 8. A typical log entry looks like:

Listing 12.11: A Timbre log entry from our Tornado API

```
16-02-04 21:20:45 tornado-api1 INFO [ring.logger.timbre] - nil
Finished :get /api for 66.108.110.85 in (956 ms) Status: 200
```

When it reaches Logstash it'll then be processed by our generic Syslog parsing and become an event much like this extract:

Listing 12.12: Our initially parsed Tornado API event

```
{  
    . . .  
  
    "syslog_timestamp" => "Feb  4 21:33:24",  
    "syslog_hostname" => "tornado-apil",  
    "syslog_program" => "tornado-api",  
    "syslog_message" => "16-02-04 21:33:23 tornado-apil INFO [ring.  
        logger.timbre] - nil Finished :get /api for 66.108.110.85 in  
        (102 ms) Status: 200",  
    "syslog_severity_code" => 5,  
    "syslog_facility_code" => 1,  
    "syslog_facility" => "user-level",  
    "syslog_severity" => "notice"  
}
```

We can then take the lessons of our HAProxy and Nginx events and parse our event further using a **grok** filter.

Listing 12.13: Updated Tornado API Logstash configuration

```
}

filter {
    if [type] == "syslog" {

        . . .

        if [syslog_program] == "tornado-nginx-access" {
            grok {
                patterns_dir => "/etc/logstash/patterns"
                match => { "syslog_message" => "%{NGINXACCESS}" }
                remove_field => ["syslog_message"]
                add_field => { "tags" => "tornado" }
            }
        }

        if [syslog_program] == "tornado-api" {
            grok {
                patterns_dir => "/etc/logstash/patterns"
                match => { "syslog_message" => "%{TORNADOAPI}" }
                remove_field => ["syslog_message"]
                add_field => { "tags" => "tornado" }
            }
        }
    }
}
```

We've added a new conditional block inside our Logstash configuration, selecting all events with a `syslog_program` of `tornado-api`. We've then used a `grok` filter

pattern, `TORNADOAPI`, to parse out our log messages further. This pattern will be loaded from our `/etc/logstash/patterns` directory on our `logstasha` host.

Now let's create and populate a file called `tornadoapi` in that directory with our Tornado API log pattern.

Listing 12.14: The tornado-api pattern

```
TORNADOAPI %{TIMESTAMP_ISO8601:app_timestamp} %{URIHOST:app_host}
  %{DATA:app_severity} %{SYSLOG5424SD} - nil %{DATA:
    app_request_state} \:%{DATA:app_verb} %{DATA:app_path} for %{
      URIHOST:app_source} (?::in \(%{INT:app_request_time:int} ms\)
        Status: %{INT:app_status_code:int}|%{GREEDYDATA:app_request})
```

Using this new pattern, we've extracted some new information from our output and put it into fields including the request path, HTTP verb, status code, and request time.

We've also added a field, `tags`, and set it to a value of `tornado`. The `tags` field will allow us to identify our `tornado` events inside Logstash and Riemann.

Now if we look at our event we'll see these new fields have been added.

Listing 12.15: Our updated Tornado API event

```
{  
    . . .  
  
    "app_timestamp" => "16-02-04 21:33:23",  
    "app_host" => "tornado-apil",  
    "app_severity" => "INFO",  
    "app_request_state" => "Finished",  
    "app_verb" => "get",  
    "app_path" => "/api",  
    "app_source" => "66.108.110.85",  
    "app_request_time" => 102,  
    "app_status_code" => 200  
    "tags" => "tornado"  
}
```

One of these fields is particularly interesting: `app_request_time`. This seems just like something we'd want to send to Riemann as an event and for which we'd want to create a metric. Let's do that now using our `riemann` output.

Listing 12.16: Sending Tornado API events to Riemann

```
output {
    . . .

    if [syslog_program] == "tornado-api" and [app_request_time] {
        riemann {
            host => "riemann"
            sender => "%{syslog_hostname}"
            map_fields => true
            riemann_event => {
                "service" => "tornado.api.request"
                "metric"  => "%{app_request_time}"
                "state"   => "ok"
            }
        }
    }
}
```

Here we've selected all events with a `syslog_program` field of `tornado-api` and with the `app_request_time` field (the conditional `if` without a condition means test for the presence of a field). We've added the `app_request_time` field condition because not all our log events have this field—only finished requests do—so we only want to send those events.

If we now restart Logstash we should be sending these events to Riemann. Let's look at one of those events inside Riemann.

Listing 12.17: The Tornado API request in Riemann

```
{:host tornado-api1, :service tornado.api.request, :state ok, :  
  description nil, :metric 52.0, :tags tornado, :time 1454684082,  
  :ttl 60, :syslog_severity_code 5, :app_request_time 52, :  
  app_host tornado-api1, :app_status_code 200, :syslog_facility  
  user-level, :type syslog, :syslog_timestamp Feb 5 14:54:42, :  
  app_request_state Finished, :port 48367, :syslog_severity  
  notice, :app_path /api, :app_severity INFO, :app_verb get, :  
  syslog_facility_code 1, :syslog_program tornado-api, :  
  syslog_hostname tornado-api1, :app_timestamp 16-02-05 14:54:42,  
  :app_source 198.179.69.250}
```

The event has a `:service` field of `tornado.api.request` and a `:metric` field value of `52` which is the response time of the Tornado API application.

Monitoring the Tornado API application

We're going to monitor our API application itself using two methods:

- We'll query the API to ensure it is returning data.
- We'll instrument our API application to emit events when it operates.

For the first method, we're going to use the `curl_json` to check the API is serving data.

Our Tornado API runs on the `tornado-api1` and `tornado-api2` hosts. It's a RESTful API that allows us to buy and sell items using `POST` and `DELETE` requests. It runs on port `8080` at the `/api` URL, for example:

`http://tornado-api:8080/api/`

NOTE You can find the source code for the Tornado API application in [the Tornado API repository](#).

We've seeded our API with a dummy item that we're going to query to ensure the API is correctly serving data. To do this we're going to request this dummy item by its item ID. The dummy item has an ID of:

`fffff3d1-835a-4f52-bf47-edc85345f4c5`

We query our dummy item using the `curl` command, for example:

Listing 12.18: Querying our dummy item

```
$ curl http://tornado-api:8080/api/fffff3d1-835a-4f52-bf47-
  edc85345f4c5
{
  "id": "fffff3d1-835a-4f52-bf47-edc85345f4c5",
  "title": "Dummy item",
  "text": "This is a dummy item.",
  "price": 666,
  "type": "stock"
}
```

This performs a `GET` request to return a JSON hash containing the item's `id`, `title`, `text`, `price`, and `type` fields. We're going to rely on this event's fields, in particular the `price` field, returning the correct results to help us determine that the API is working correctly.

We can create an event from this data using the `curl_json` plugin. We'll configure the plugin now. Let's add our configuration to the existing:

`/etc/collectd.d/tornado-api.conf`
configuration file.

Listing 12.19: Adding to the tornado-api.conf configuration file

```
LoadPlugin curl_json
<Plugin curl_json>
  <URL "http://tornado-api1:8080/api/fffff3d1-835a-4f52-bf47-
    edc85345f4c5">
    Instance "tornado-api"
    <Key "price">
      Type "gauge"
    </Key>
  </URL>
</Plugin>
```

We first load the `curl_json` plugin with the `LoadPlugin` directive. We then configure it in the `Plugin` block. We add a `URL` block for the Tornado API endpoint we wish to scrape. In this case we're querying `tornado-api`. You'd use your configuration management tool to template and update this for the relevant server name:

`http://tornado-api1:8080/api/fffff3d1-835a-4f52-bf47-edc85345f4c5`

By default, the plugin will check the endpoint using the value of the global `Interval` setting, in our case every two seconds.

Inside our `URL` block we've specified the `Instance` directive. The directive sets the value of the `:plugin_instance` field in our event. We'd set it to the name of our application, here `tornado-api`. Next, we specify a `Key` block. The `Key` block controls what piece of JSON we're going to scrape from the endpoint. It matches the `price` key of our JSON hash. Inside a `Key` block we also specify the `Type` directive, here a `gauge`, to indicate our `price` value will be recorded as a gauge.

We'll need to restart collectd for our new check to be enabled.

Listing 12.20: Restarting collectd for curl_json

```
$ sudo service collectd restart
```

Now let's look at what an event from the Tornado API might look like:

Listing 12.21: A curl_json event from the Tornado API

```
{:host tornado-api1, :service curl_json-tornado-api/gauge-price,
  :state ok, :description nil, :metric 666, :tags [collectd
tornado], :time 1457163125, :ttl 60.0, :ds_index 0, :ds_name
value, :ds_type gauge, :type_instance price, :type gauge, :
plugin_instance tornado-api, :plugin curl_json}
```

We see that our event's `:service` field has been constructed from our plugin name, the `Instance` directive, the `Type` directive, and the name of the JSON hash key we specified in the `Key` directive:

`curl_json-tornado-api/gauge-price`

Our `:metric` field has been populated with the value of the `price` key, `666`. We'll make use of this event in the next section to ensure our API is functional.

TIP We could also use collectd's threshold plugin here to set a threshold for this check that would generate a notification if the `price` field returned a different result or if the event did not return at all.

Lastly, in our setup of monitoring for the [Tornado API](#), we've also instrumented

the application to generate StatsD metrics for specific events. Our API specifically creates, updates, or deletes items in our environment based on sales. We've added a Clojure StatSD client to our application. We've then instrumented [the relevant API endpoints of our application](#).

Listing 12.22: Our instrumented Tornado API setup

```
(ns tornado-api.handler
  ...
  (:require [compojure.handler :as handler]
            [clj-statsd :as statsd]
  ...
  (def statsd-prefix "tornado.api.")
  ...)
```

We see our Tornado API requires the `clj-statsd` client. We've then defined a var called `statsd-prefix` with a value of `tornado.api`. We'll use this var to prefix all of our StatsD metrics.

We've instrumented three of our functions: `buy-item`, `update-item`, and `sell-item`. Let's look at each in turn.

Listing 12.23: Our instrumented Tornado API buy-item method

```
(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)
        (statsd/gauge (str statsd-prefix "item.bought.total")
          (item "price"))))
      (wcar* (car/ping)
        (car/set id (item "title"))))
      (get-item id))))
```

Each time these functions are called, a StatsD gauge or counter will be incremented. If, for example, we were to hit the `/api` API endpoint with a `POST` like so:

Listing 12.24: Hitting the buy-item API endpoint

```
$ curl -X POST -H "Content-Type: application/json" -d '{"title":"This is an item","text":"A Tornado application item","price":123.45}' http://tornado-api:8080/api
```

Then we'd generate a new metric, like so, and send it to Riemann:

`tornado.api.item.bought.total 123.45`

We've done the same with the `update-item` and `sell-item` methods.

Listing 12.25: Our instrumented Tornado API update and sell item methods

```
(defn update-item [id item]
  (sql/db-do-commands db-config
    (let [item (assoc item "id" id)]
      (sql/update! db-config :items ["id=?" id] item)
      (statsd/increment (str statsd-prefix "update.item")))
    )
  (get-item id))

(defn sell-item [id]
  (sql/db-do-commands db-config
    (let [item (get-item id)
          price (get-in item [:body :price])
          item_state (get-in item [:body :type])]
      (when-not (= item_state "sold")
        (sql/update! db-config :items { :type "sold"} ["id=?"
          " id"])
        (statsd/gauge (str statsd-prefix "item.sold.total")
          price)))
    )
  (get-item id))
```

To collect the metrics from this instrumentation we need to revisit setting up [the statsd plugin](#) for collectd that we explored in Chapter 9.

On each of our API servers we create a new collectd configuration file, let's call it `statsd.conf`, in the `/etc/collectd.d/` directory and populate it.

Listing 12.26: The statsd.conf configuration file

```
LoadPlugin statsd

<Plugin statsd>
    Host "localhost"
    Port "8125"
    TimerPercentile 90
    TimerPercentile 99
    TimerLower true
    TimerUpper true
    TimerSum true
    TimerCount true
</Plugin>
```

We first load the `statsd` plugin. We then configure the plugin and set the `Host` and `Port` to which we're binding the StatsD server. In our case we're binding it to the `localhost` interface on port `8125`.

Let's restart collectd to enable the plugin.

Listing 12.27: Restarting collectd to enable statsd

```
$ sudo service collectd restart
```

Now let's look at our Riemann server to see our StatsD metrics as events:

Listing 12.28: A Tornado API StatsD metric in Riemann

```
:host tornado-api2, :service statsd/gauge-tornado.api.item.bought
  .total, :state ok, :description nil, :metric 5, :tags [collectd
  tornado], :time 1454782475, :ttl 60.0, :ds_index 0, :ds_name
  value, :ds_type derive, :type_instance tornado.api.item.bought.
  total, :type derive, :plugin statsd}
```

We've generated an event showing the metric for the total of items created on our API endpoint.

We'll probably want to rewrite these events before they hit Graphite. To do this we again update our collectd rewrite rules inside the `/etc/riemann/examplecom/etc/collectd.clj` file.

Listing 12.29: Rewriting the Tornado API events

```
(def default-services
  [{:service #"^load/load/(.*)$" :rewrite "load $1"}

  . . .

  {:service #"^statsd\/(gauge|derive)-(.*)$" :rewrite "statsd $1
  $2"}])
```

Here we've added a new rewrite line that will update our Tornado API event's `:service` field before it is written to Graphite, for example from:

`statsd/gauge-tornado.api.item.bought.total`

To a slightly simpler:

`statsd.gauge.tornado.api.item.bought.total`

We can now more easily use these metrics in graphs or checks.

Addressing the Tornado Application tier monitoring concerns

Now that we have the Application tier monitored and sending data, what can we do with that data? We identified a series of concerns for Tornado's app tier earlier. Those concerns were:

- That the Tornado API is running on our hosts.
- That the Tornado application 99th-percentile latency is 100 milliseconds or less.
- That the percentage of JVM heap memory used is under a threshold.

Let's look at how we might address these concerns.

Thanks to the `processes` plugin, we get process monitoring and notifications resulting from the thresholds we configured in Chapters 5 and 6. We don't need to make any specific changes here, but we could tweak the threshold for processes depending on the number of API back ends running, if required.

We can also address our other concerns with checks we've previously created. Let's add a new function to our `tornado.clj` file to hold our Application tier checks. We'll call this function `apptier` and call it via our `splitp` var.

Listing 12.30: The tornado checks function

```
    . . .

  (splitp re-matches host
    web-tier-hosts (webtier)
    app-tier-hosts (apptier)
    db-tier-hosts  (datatier)
    #(info "Catchall - no tier defined" (:host %) (:service %)))
  )
```

The `apptier` function will hold the checks for our Application tier.

Listing 12.31: The apptier function

```
(defn apptier
  "Checks for the Tornado App Tier"
  []
  (sdo
    (where (service "curl_json-tornado-api/gauge-price"))
    (where (!= metric 666)
      (slacker))
    (expired
      (page)))
    (where (service #^tornado.api.))
    (smap rewrite-service graph))
  (check_ratio "GenericJMX-memory-heap/memory-used"
    "GenericJMX-memory-heap/memory-max"
    "jmx.memory-heap.percentage_used"
    80 90
    (alert_graph))
  (where (service "tornado.api.request")
    (with { :service "tornado.api.request.rate" :metric 1 }
      (rate 1
        (smap rewrite-service graph))))
  (check_percentiles "tornado.api.request" 10
    (smap rewrite-service graph)
    (where (and (service "tornado.api.request 0.99") (>= metric
      100.0))
      (changed-state { :init "ok"
        (slacker)))))))
```

Our first check uses the event generated from our dummy item by the `curl_json` plugin to confirm that our API service is functioning correctly. It matches on a `:service` of:

```
curl_json-tornado-api/gauge-price
```

And then performs two sub-checks. The first sub-check sends a Slack notification if the `:metric` value of the event is not `666`, the price of the dummy item. This will let us know if something isn't right with our API. The second sub-check pages us if this event expires from the index. If the event has expired from the index it likely means that the API can't be queried and we should investigate this issue.

Our next check isn't really a check. Instead we're taking any events whose `:service` starts with `tornado.api.` and sending them to Graphite. This will collect any API metrics, like `tornado.api.buy.item`, generated by our Tornado API and ensure they are available as graphs.

The following check uses the `check_ratio` abstraction we created in Chapter 11. We're taking two events, the JVM heap memory used and heap memory max, and comparing them. We generate a new event:

```
jmx.memory-heap.percentage_used
```

And check that event's `:metric` field against a warning and critical threshold. If the percentage of JVM heap memory used exceeds `80%` then a warning will be triggered to Slack, and if it hits `90%` a page will be generated. We've wrapped the notifications in a `changed-state` var to ensure we only notify when the state actually changes.

These are the same child streams we used in our previous `check_ratio` function in the `webtier` function. Rather than repeat the same child streams we've DRY'ed this up and created a function for this default set of child streams: `alert_graph`.

We've created the `alert_graph` function at the top of our `tornado.clj` file, above our `webtier` function.

Listing 12.32: DRY'ed child streams

```
(defn alert_graph
  []
  "Alert and graph on events"
  (sdo
    (changed-state {:init "ok"})
    (where (state "critical")
      (page))
    (where (state "warning")
      (slacker)))
    (smap rewrite-service graph)))
```

In the future, if we need to use the same set of child streams, we can make use of this `alert_graph` function. We can also go back and refactor our previous use of this pattern.

Our last check creates a new metric showing the rate of API requests every second.

Listing 12.33: Creating a rate

```
(where (service "tornado.api.request")
  (with { :service "tornado.api.request.rate" :metric 1 }
    (rate 1
      (smap rewrite-service graph))))
```

We start by selecting any `tornado.api.requests` events and, using `the with stream`, we create a new event with a `:metric` set to `1` and a service name of `tornado.api.request.rate`. We set the metric to `1` because we don't want to

count the request time; instead we want to count each instance of the event. We then pass this event to [the rate stream](#). The `rate` stream takes a timeframe, here 1 second, that we'll calculate the rate over. We'll then write that metric to Graphite via the `graph` var. That metric will look something like:

Listing 12.34: The `tornado.api.request` rate

```
{:host tornado-api1, :service tornado.api.request.rate, :state ok
, :description nil, :metric 7, :tags [tornado], :time
1455322655207/1000, :ttl 60 . . . }
```

We then graph the metric from each API server like so:

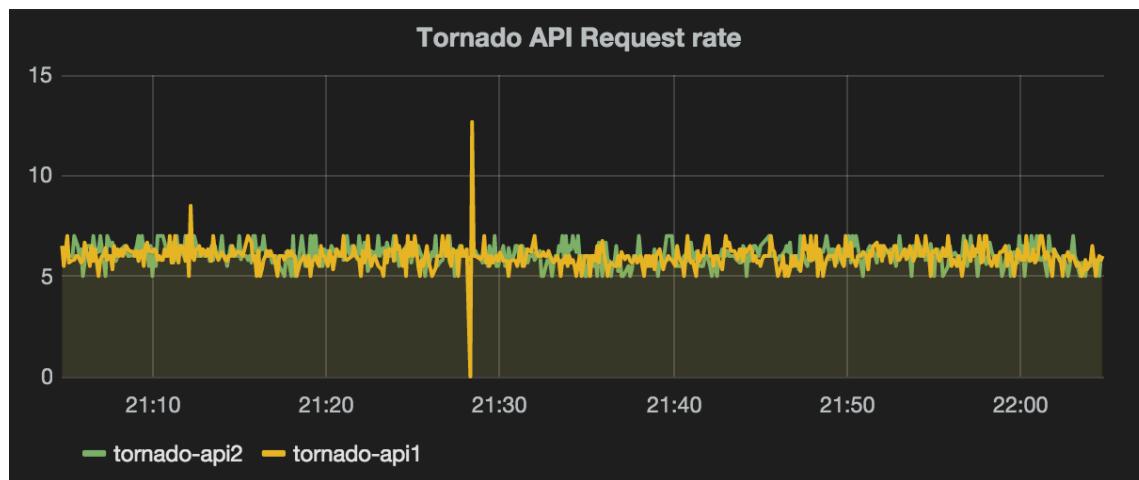


Figure 12.1: Graphing our Tornado API rate

This graph shows us the rate of API requests per second for both API servers.

Our last check uses the [check_percentiles](#) check abstraction we created in Chapter 6. It takes the `tornado.api.request` event, generated from the Tornado API's logs, parsed in Logstash and then sent to Riemann, and creates 0.5, 0.95, 0.99, and 1 (or max) percentiles from it over a 10-second window. We've then added two

child streams to the check. The first writes our new percentile events to Graphite so we can make graphs from them. The second checks if the 99th-percentile request time exceeds `100` and, if so, sends a Slack notification. We've wrapped our Slack notification in the `changed-state` var to ensure we only get notified when appropriate.

Our initial checks for the Application tier should ensure we know when the application services have an issue as well as provide a picture of their current performance.

Summary

In this chapter we've used similar techniques to Chapter 11 to monitor the services that make up our Application tier. We've exposed both business and application performance metrics. We've then built checks and thresholds for letting us know when our Tornado Application tier is available and when it is not performing in accordance with our expectations.

In the next chapter we'll conclude our monitoring of the Tornado application by looking at the Data tier. We'll also expose some of our metrics as visualizations for business owners, developers, and operations teams.

Chapter 13

Monitoring Tornado: Data tier

Our last tier, the Data tier, is made up of two servers, `tornado-db` running MySQL and `tornado-redis` running Redis. For Tornado's Data tier, what do we care about? Let's break down our concerns:

- That MySQL and Redis are running on their respective hosts.
- 99th-percentile latency adding items to the Tornado DB is three milliseconds or less.
- That maximum MySQL connections don't exceed 80% of the available connections.
- That we can measure the rate of aborted connections to MySQL.
- That we are generating application metrics from data inside our MySQL database.
- That we can measure key queries to ensure they are performing within expectations.

We'll start with configuring the monitoring for each component and the underlying database or data store, and then we'll look at using the events and metrics generated by that monitoring to address those concerns.

Monitoring the Data tier MySQL server

Let's start by monitoring our MySQL server. In our case we have a database called `items` running on our `tornado-db` host that we'd like to monitor. For MySQL monitoring the recommended approach is to look at the health and metrics of the MySQL server itself as well as the performance of application-specific queries. We're going to start with monitoring the MySQL server itself.

In Chapter 11 we saw a community-contributed plugin for monitoring HAProxy. We're going to use a similar plugin to monitor MySQL. In this case our new plugin is another Python-based plugin that we'll execute using the collectd `python` plugin. It connects to our MySQL server and runs some useful commands, like: `SHOW GLOBAL STATUS`, `SHOW GLOBAL VARIABLES`, and `SHOW PROCESSLIST`. It then converts the results of these commands into events and feeds them into collectd.

TIP We're focused on MySQL here but there are plugins available for a wide variety of databases and data stores.

Let's start by installing a prerequisite package, the Python MySQL bindings. On Ubuntu this is the `python-mysqldb` package.

Listing 13.1: Installing the python-mysqldb package

```
$ sudo apt-get -qqy install python-mysqldb
```

On Red Hat derivatives, this is the `mysql-python` package.

Listing 13.2: Installing the mysql-python package

```
$ sudo yum install mysql-python
```

Next we need to grab our plugin. We'll create a directory to hold it, `/usr/lib/collectd/mysql`, and then download the plugin into it.

Listing 13.3: Download the mysql.py plugin

```
$ mkdir /usr/lib/collectd/mysql
$ cd /usr/lib/collectd/mysql
$ wget https://raw.githubusercontent.com/jamtur01/collectd-python
-mysql/master/mysql.py
```

Here we've created the directory and changed into the `/usr/lib/collectd/mysql` directory (on Red Hat this would be the `/usr/lib64/collectd/mysql` directory). We've then downloaded the plugin from GitHub. You can take a look [at its source](#). It defines a list of metrics and some configurable options.

TIP You can learn about writing your own Python plugins on the [collectd Python man page](#).

Next we need to create a MySQL user through which the plugin can monitor our database. We do this using MySQL's standard user and privilege system. Let's create a user called `collectd`.

Listing 13.4: Creating a MySQL collect user

```
$ mysql -u root -p
Enter password: ****
mysql> CREATE USER 'collectd'@'localhost' IDENTIFIED BY '
    strongpassword';
Query OK, 0 rows affected (0.00 sec)
mysql> GRANT PROCESS,USAGE,SELECT ON *.* TO 'collectd'@'
    localhost';
Query OK, 0 rows affected (0.00 sec)
```

You can see we've used the `mysql` command to connect to our MySQL server. We've then created a new user, `collectd`, with a password of `strongpassword`. You should replace this with an appropriately strong password of your own. We've then granted the `USAGE`, `PROCESS`, and `SELECT` privileges to this new user. The `USAGE` level of privilege is the lowest level of MySQL access and just allows our user to interact with the target database. The `PROCESS` privilege allows us to view the state of the MySQL server processes. The `SELECT` permission will allow us to query specific databases for metrics, as we'll see later in this chapter. We've granted the access across all MySQL databases on the host, `*.*`. In your environment you could easily limit this to a subset of databases, as required.

Next we create a configuration file for the `mysql` plugin. Let's create it at: `/etc/collectd.d/mysql.conf`.

Now let's populate it.

Listing 13.5: The mysql.conf collectd configuration file

```
<LoadPlugin python>
    Globals true
</LoadPlugin>
<Plugin python>
    ModulePath "/usr/lib/collectd/mysql/"
</Plugin>

<Plugin python>
    Import mysql
    <Module mysql>
        Host "localhost"
        Port 3306
        User "collectd"
        Password "strongpassword"
    </Module>
</Plugin>

LoadPlugin processes
<Plugin "processes">
    Process "mysqld"
</Plugin>
```

We first load the `python` plugin and then turn on `Globals`, which provides our plugin access to local Python libraries. We then use a `<Plugin>` block to tell collectd where to find the `mysql` plugin and how to load it. We specify the path to the plugin, and we use the `Import` command to import the specific plugin file, `mysql`. This is the name of the file that contains the Python code, dropping the

.py extension.

Inside this block we then specify a <Module> block to configure the MySQL plugin itself. Inside this block we use the Host option to specify the database host, and Port to specify the port. We then specify the User and Password options for the username and password we just created and will use to connect to the MySQL server.

The plugin will connect to our database and run commands to return details of the database's state. This shows performance, command, handler, query, thread, and traffic statistics for the specific server.

TIP If you're interested in data from specific databases there's also [the DBI plugin](#) that we'll see shortly.

At the end of our configuration we've also loaded and configured the processes plugin to monitor the mysqld process to ensure we know when it's available.

If we now restart collectd, the plugin will connect to MySQL, collect, and return our events to Riemann. Let's look at an example event.

Listing 13.6: A typical MySQL event in Riemann

```
{:host tornado-db, :service mysql-innodb/counter-log_writes, :  
    state ok, :description nil, :metric 8, :tags [collectd tornado  
    ], :time 1455399827, :ttl 60.0, :ds_index 0, :ds_name value, :  
    ds_type counter, :type_instance log_writes, :type counter, :  
    plugin_instance innodb, :plugin mysql}
```

We see the event has a :service prefix of mysql-, the type of data it is—here

InnoDB statistics—as well as the data class, a counter, and finally the name of the specific metric being recorded. Given this complexity, we'll probably want to rewrite these events before they hit Graphite. To do this we again update our collectd rewrite rules inside the `/etc/riemann/examplecom/etc/collectd.clj` file.

Listing 13.7: Rewriting the MySQL events

```
(def default-services
  [{:service #"^load/load/(.*)$" :rewrite "load $1"}

  . . .

  {:service #"^mysql-(.*)\/(counter|gauge)-(.*)$" :rewrite "
    mysql $1 $3"}])
```

Here we've added a new rewrite line that will update our MySQL event's `:service` field before it is written to Graphite, for example from:

`mysql-innodb/counter-log_writes`

To a simpler:

`mysql.innodb.log_writes`

We can now more easily use these metrics in graphs or checks.

Using MySQL data for metrics

In addition to the state of our MySQL server we're going to use data inside our databases and tables as metrics. This allows us to expose application state and performance at multiple levels: inside our Web tier, inside the Application tier and our application itself, and inside the Data tier. To make use of our Tornado application data we're going to use another collectd plugin: `dbi`. The `dbi` plugin

connects to a database, runs queries to return data, and converts that data into a collectd metric.

TIP We're using the `dbi` plugin with MySQL but it also supports a wide variety of other databases.

Let's start with installing a prerequisite package that the `dbi` plugin needs, the `libdbi bindings`. On Ubuntu this is the `libdbd-mysql` package.

Listing 13.8: Installing the libdbd-mysql package

```
$ sudo apt-get -qqy install libdbd-mysql
```

On Red Hat derivatives, this is the `libdbi` package.

Listing 13.9: Installing the libdbi package

```
$ sudo yum install libdbi
```

Now let's look at querying some data. Inside our MySQL database we've got the `items` database that is being populated by our Tornado API servers. We're going to query that data to return some key metrics. To do this we create a configuration for the `dbi` plugin in our existing configuration file: `/etc/collectd.d/mysql.conf`

.

Listing 13.10: The dbi plugin configuration

```
LoadPlugin dbi
<Plugin dbi>
  <Query "get_item_count">
    Statement "SELECT COUNT(*) AS value FROM items;"
    MinVersion 50000
  <Result>
    Type "gauge"
    InstancePrefix "tornado_item_count"
    ValuesFrom "value"
  </Result>
</Query>
. .
<Database "items">
  Driver "mysql"
  DriverOption "host" "localhost"
  DriverOption "username" "collectd"
  DriverOption "password" "collectd"
  DriverOption "dbname" "items"
  SelectDB "items"
  Query "get_item_count"
. .
</Database>
</Plugin>
```

We've added a `LoadPlugin` statement for our `dbi` plugin and then defined its configuration in a `<Plugin>` block. The `dbi` plugin is configured with two distinct blocks: queries and databases. Queries define the SQL statement and data we'd

like to pull from our database. Databases configure connection details for each specific database and which queries apply to them.

We start with defining a query. Queries need to be defined before the database that will use them as the file is parsed top down. Each query consists of a `<Statement>`, the SQL statement we want to run. We've also specified the `MinVersion` directive, which specifies the minimum version of the server on which we're running the statement. Here, if the database server we connect to is an older version than MySQL 5.0 then the query will not be run. Our statement will do a count of items in the `items` table, returning the count aliased to a name of `value`.

We then use the result returned from the statement in the `<Result>` block to construct our metric. We've used three directives: `Type`, `InstancePrefix`, and `ValuesFrom`. The `Type` directive controls what type of metric we're constructing; in our case that's a `gauge`. The `InstancePrefix` provides a prefix to our metric name; here we're using `tornado_item_count`. Lastly, the `ValuesFrom` directive controls what value will be used as the metric value. In our statement we've aliased the count from the `items` table with an `AS` clause to `value`. The `value` variable will contain the total count of items, and we're going to assign it to the `ValuesFrom` directive.

We're also storing the price of each item we're creating in the database. Let's add another query to get that information and return a metric to track it.

Listing 13.11: Our price query

```
<Query "items_sold_total_price">
    Statement "SELECT SUM(price) AS total_price FROM items WHERE
        type = 'sold'"
    MinVersion 50000
    <Result>
        Type "gauge"
        InstancePrefix "items_sold_total_price"
        ValuesFrom "total_price"
    </Result>
</Query>
```

Here we're running a query that sums the total value of the `price` field and returns it as a new field called `total_price`. We've then created a new metric result with an `InstancePrefix` of `items_sold_total_price`; it will have a `:metric` value of the contents of the `total_price` field.

We've created a query to record the items bought as well, creating a metric called `items_bought_total_cost` that we haven't shown here. See the [complete file in the book's source code for details](#).

We could create a variety of other metrics from this data: total price of items added per day, total price of items deleted per day, and others.

TIP We could also add these metrics in the application code for the API but, as we have multiple API servers and the database represents a central source of truth, we're querying them here.

Our `<Database>` block is fairly self-explanatory. We define the host and port of our database server (ours is our local host) and the user and password we just created. Additionally, we specify the target database to which we want to connect. Lastly, we specify `Query` directives that tell the `dbi` plugin what queries to run on this database.

Listing 13.12: The Query directives

```
Query "get_item_count"
```

TIP You can find the `dbi` plugin's other configuration options in the [collectd documentation](#).

If we then restart the collectd service on our `tornado-db` host we'll start to see each of these queries executed and metrics being generated and sent to Riemann. Let's examine these events inside Riemann.

Listing 13.13: The items database events in Riemann

```
{:host tornado-db, :service dbi-items/gauge-tornado_item_count, :  
state ok, :description nil, :metric 484621.0, :tags [collectd  
tornado], :time 1455747174, :ttl 60.0, :ds_index 0, :ds_name  
value, :ds_type gauge, :type_instance tornado_item_count, :type  
gauge, :plugin_instance items, :plugin dbi}
```

Here we see an event from the `dbi` plugin with a `:service` of:

```
dbi-items/gauge-tornado_item_count
```

And `:metric` of `484621.0`, representing the total count of items in the `items` database.

As we did with our MySQL metrics, we want to rewrite these events before they hit Graphite. To do this we again update our collectd rewrite rules inside the `/etc/riemann/examplecom/etc/collectd.clj` file.

Listing 13.14: Rewriting the dbi events

```
(def default-services
  [{}{:service #"^load/load/(.*)$" :rewrite "load $1"}

  . . .

  {:service #"^dbi-(.*)>/(gauge|counter)-(.*)$" :rewrite "dbi $1
$3"}])
```

Here we've added a new rewrite line that will update our `dbi` event's `:service` field before it is written to Graphite, for example from:

```
dbi-items/gauge-tornado_item_count
```

To a simpler:

```
dbi.items.tornado_item_count
```

We can now more easily use these metrics in graphs or checks.

Query timing

Lastly, when monitoring a database, understanding the execution time of your queries is critical. This can be done in a number of different ways in the MySQL world:

- Via parsing [the slow query log](#) — Useful but hard to parse and, at high volume, adds considerable overhead to your host.
- Via network monitoring — Complex to set up and requires building custom tools or using a third-party platform.
- Via [the performance_schema database](#).

We're going to use the last method because it allows us to leverage our existing `dbi` plugin implementation to retrieve statistics from the `performance_schema` database.

The `performance_schema` database is a mechanism for monitoring MySQL server execution at a low level. It was introduced in MySQL 5.5.3. The `performance_schema` database includes a set of tables that give information on how statements performed during execution. The tables we're interested in contain current and historical data on statement execution. We're going to extract information about the queries our Tornado API is running from this table: [the events_statements_history_long table](#). The `events_statements_history_long` table was introduced in MySQL 5.6.3 and contains the last 10,000 statement events. This is the default that can be configured using the:

`performance_schema_events_statements_history_long_size`
[configuration option](#).

But before we get started we need to determine if the `performance_schema` database is enabled by checking the status of the `performance_schema` variable. We sign in to the MySQL console using the `mysql` command.

Listing 13.15: Checking the PERFORMANCE_SCHEMA

```
$ mysql -p

. . .

mysql> SHOW VARIABLES LIKE 'performance_schema';
+-----+-----+
* Variable_name      Value
+-----+-----+
* performance_schema | ON
+-----+-----+
1 row in set (0.00 sec)
```

If the `performance_schema` variable is set to `ON` then it is enabled. If it's `OFF` we can enable it in our MySQL configuration file. We'll also enable `some performance_schema consumers` to record our statements.

Listing 13.16: Enabling the PERFORMANCE_SCHEMA

```
[mysqld]
performance_schema=ON
performance-schema-consumer-events_statements_history=ON
performance-schema-consumer-events_statements_history_long=ON
```

We'll need to restart MySQL to enable it. If you find the variable doesn't exist, you may have an older version of MySQL that does not support the `performance_schema` database.

Now we're going to add another query to our `dbi` plugin configuration to return the execution time of one of our key Tornado API queries, in this case the `INSERT` statement that adds an item to our database. Let's do that now.

Listing 13.17: New query

```
    . . .

<Query "insert_query_time">
    Statement "SELECT MAX(thread_id), timer_wait/1000000000 AS
        exec_time_ms
    FROM events_statements_history_long
    WHERE digest_text = 'INSERT INTO `items` ( `title` , TEXT , `'
        price` , `id` ) VALUES ( ... )';"
    MinVersion 50000
    <Result>
        Type "gauge"
        InstancePrefix "insert_query_time"
        ValuesFrom "exec_time_ms"
    </Result>
</Query>

    . . .

<Database "performance_schema">
    Driver "mysql"
    DriverOption "host" "localhost"
    DriverOption "username" "collectd"
    DriverOption "password" "collectd"
    DriverOption "dbname" "performance_schema"
    Query "insert_query_time"
</Database>
</Plugin>
```

Our new `<Query>` block defines a new query we've called `insert_query_time`. It contains a statement that queries the `events_statements_history_long` table. We pull all statements from that table that match our Tornado API `INSERT` statement.

Listing 13.18: The Tornado API INSERT statement

```
INSERT INTO `items` ( `title` , TEXT , `price` , `id` ) VALUES  
(...)
```

We return the last thread that executed our query using the thread ID; this is the unique identifier of the instrumented thread. We then return the `timer_wait` field, which is the time the query took to execute in picoseconds. We've applied `timer_wait/1000000000` to change those picoseconds into milliseconds as the field `exec_time_ms`. We then use this `exec_time_ms` field in the `<Result>` block as the value of our `:metric` field and to construct an event for Riemann. We specify another `<Database>` block to execute our new query.

If we then restart collectd we'll be able to run this query and send a new metric to Riemann. Let's look at an example of this event now.

Listing 13.19: The INSERT event in Riemann

```
{:host tornado-db, :service dbi-performance_schema/gauge-  
    insert_query_time, :state ok, :description nil, :metric  
    0.370724, :tags [collectd tornado], :time 1455995585, :ttl  
    60.0, :ds_index 0, :ds_name value, :ds_type gauge, :  
    type_instance insert_query_time, :type gauge, :plugin_instance  
    performance_schema, :plugin dbi
```

Here we see our new event with a `:service` of:

`dbi-performance_schema/gauge-insert_query_time`

And a `:metric` value of `0.370724` milliseconds.

NOTE This whole section assumes you're running a flavor of MySQL 5.6 or later.

Monitoring the Data tier's Redis server

The last service we wish to monitor is Redis. We use another collectd plugin to do this. The `collect redis` plugin connects to your Redis instances using `the credis library` and returns usage statistics.

To configure the `redis` plugin we need to add a configuration file with the connection details of our Redis instance. Let's create `/etc/collectd.d/redis.conf` on our `tornado-redis` host now.

Listing 13.20: The redis.conf collectd configuration

```
LoadPlugin redis
<Plugin redis>
  <Node "tornado-redis">
    Host "localhost"
    Port "6379"
    Timeout 2
    Password "strongpassword"
  </Node>
</Plugin>

LoadPlugin processes
<Plugin "processes">
  Process "redis-server"
</Plugin>
```

We first load the `redis` plugin. We then configure a `<Plugin>` block, and inside that a `<Node>` block for each Redis instance we wish to monitor. Each `<Node>` block needs to be uniquely named and contains the connection settings for that Redis instance. We specify the host, port, a timeout, and, optionally, a password for the connection.

We've also included configuration for the `processes` plugin to monitor our `redis-server` process.

When we restart collectd we'll start collecting Redis metrics and reporting them to Riemann. We'll see metrics covering uptime, connections, memory, and operations. Each event's `:service` will be named for the name provided in the `<Node>` block and the name of the plugin. For example, for Redis memory usage you'll see an event like so:

Listing 13.21: A Riemann Redis event

```
{:host tornado-redis, :service redis-tornado-redis/memory, :state
  ok, :description nil, :metric 501248.0, :tags [collectd
  tornado], :time 1455055518, :ttl 60.0, :ds_index 0, :ds_name
  value, :ds_type gauge, :type memory, :plugin_instance tornado-
  redis, :plugin redis}
```

Like our MySQL events, let's rewrite these events before they hit Graphite. To do this we edit the `/etc/riemann/examplecom/etc/collectd.clj` file on our Riemann server.

Listing 13.22: Rewriting the Redis events

```
(def default-services
  [{:service #"^load/load/(.*)$" :rewrite "load $1"}
   .
   .
   .
   {:service #"^redis-(.*)$" :rewrite "redis $1"}])
```

This will turn an event like:

`redis-tornado-redis/memory`

Into:

`redis.tornado-redis.memory`

Making it easier to parse and graph.

NOTE We could also collect logs from both our MySQL and Redis services using the techniques we established in Chapters 11 and 12, but nothing we're currently monitoring for explicitly requires it.

Addressing the Tornado Data tier monitoring concerns

Now that we have the Data tier monitored and sending data, what can we do with that data? We identified a series of concerns for Tornado's Data tier earlier. Those concerns were:

- That MySQL and Redis are running on their respective hosts.
- 99th-percentile latency adding items to the Tornado DB is 3 milliseconds or less.
- That maximum MySQL connections don't exceed 80% of the available connections.
- That we can measure the rate of aborted connections to MySQL.
- That we're able to make use of the application metrics from the data inside our MySQL database.
- That we can measure key queries to ensure they are performing within expectations.

Let's look at how we might address these concerns.

Like our earlier Web and Application tier components, we get process monitoring from the `processes` plugin and notifications thanks to the thresholds we configured in Chapters 5 and 6. We don't need to make any specific changes here but we could tweak the threshold for processes depending on the number of Redis or MySQL instances running, if required.

We can address our other concerns by using checks we've previously created too. Let's add a new function to our `tornado.clj` file to hold our Data tier checks. We'll call this function `datatier` and call it via our `splitp` var.

Listing 13.23: The `tornado` checks function

```
    . . .

  (splitp re-matches host
    web-tier-hosts (webtier)
    app-tier-hosts (apptier)
    db-tier-hosts  (datatier)
    #(info "Catchall - no tier defined" (:host %) (:service %)))
  )
```

The `datatier` function will hold the checks for our Data tier. Let's look at our checks now.

Listing 13.24: The datatier function

```
(defn datatier
  "Check for the Tornado Data Tier"
  []
  (sdo
    (check_ratio "mysql-status/gauge-Max_used_connections"
                 "mysql-variables/gauge-max_connections"
                 "mysql.max_connection_percentage"
                 80 90
                 (alert_graph))
    (create_rate "mysql-status/counter-Aborted_connects" 5)
    (check_percentiles "dbi-performance_schema/gauge-
      insert_query_time" 10
      (smap rewrite-service graph)
      (where (and (service "dbi-performance_schema/gauge-
        insert_query_time 0.99") (>= metric 3.0))
              (changed-state { :init "ok"}
                            (slacker))))))
```

We've defined three checks. Our first check, using our `check_ratio` function, is to create a percentage metric for maximum used connections. We set a threshold generating a warning notification if the percentage exceeds 80%, and a critical notification if the percentage exceeds 90%. We use the generic `alert_graph` function to handle the notification and send the resulting percentage metric to Graphite.

Our second check creates a rate for MySQL aborted connections. As we have created a couple of rates now, let's create a new function to manage this process. We'll create a function called `create_rate` and add it to our `/etc/riemann/examplecom/etc/checks.clj` file. Let's look at this new function.

Listing 13.25: The `create_rate` function

```
(defn create_rate [srv window]
  (where (service srv)
    (with {:service (str srv " rate")})
    (rate window index (smap rewrite-service graph))))
```

Our `create_rate` function takes a service and a time window as parameters. It uses a `where` stream to filter events based on service. We then use a `with` stream to create a new copy of the event and adjust the `:service` field to append `rate` to the field. We then pass this new event to `the rate stream` where we calculate a rate using the `window` parameter and pass it into Graphite via `an smap`. We could add other child streams or checks to this function too.

Our last check uses one of the metrics we created with the `dbi` plugin. We're using our `check_percentiles` check to create percentiles from the:

`dbi-performance_schema/gauge-insert_query_time`

metric. We're creating percentiles over a 10 second window, writing them to Graphite, and then using the resulting 99th percentile.

Listing 13.26: The INSERT query check

```
(check_percentiles "dbi-performance_schema/gauge-
  insert_query_time" 10
  (smap rewrite-service graph)
  (where (and (service "dbi-performance_schema/gauge-
    insert_query_time 0.99") (>= metric 3.0))
  (changed-state { :init "ok"
    (slacker))))))
```

If the 99th percentile of the `INSERT` query time exceeds three milliseconds then we'll trigger a Slack notification.

This is a small set of the possible checks for our Data tier. You can easily expand on this to suit your needs.

The Tornado dashboard

With all of our metrics flowing into Graphite we can now build a dashboard. Let's build a Tornado-specific dashboard that we'd share with the team managing it. We want to construct an eclectic dashboard with some business metrics and some more technical visualizations.

We sign in to our Grafana server and create a new Dashboard called `Tornado`. We're going to create two initial panels with business metrics, both taking advantage of data we're pulling from our MySQL database with the `dbi` plugin. But rather than create a graph, we're going to create a single stat panel.

Chapter 13: Monitoring Tornado: Data tier

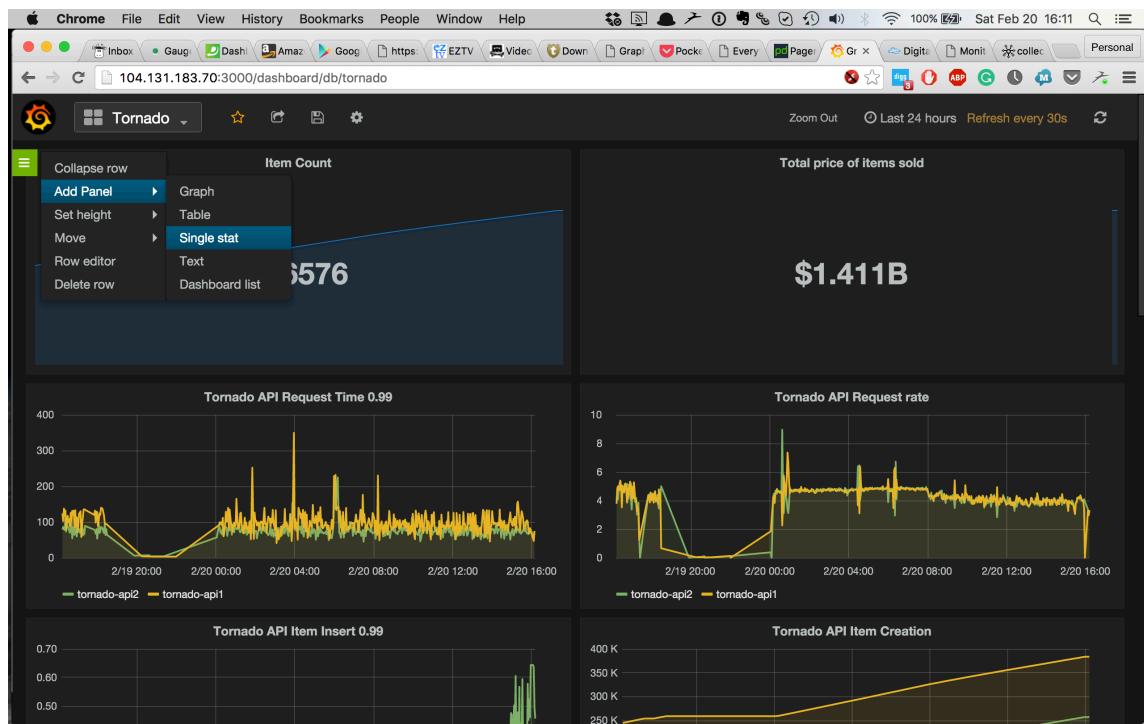


Figure 13.1: Adding a single stat panel

A single stat panel contains a single number or metric. We're going to create our first one with the current item count of the Tornado application.

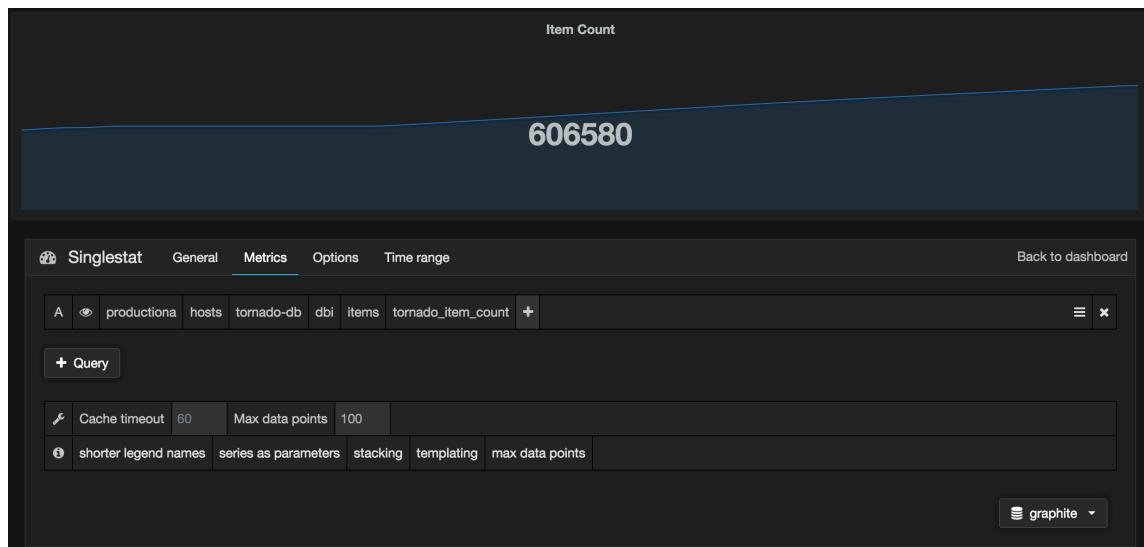


Figure 13.2: Tornado item count panel

We specify the metric name of our item count:

```
productiona.hosts.tornado-db.dbi.items.tornado_item_count
```

And we give the panel a title, `Item Count`, and then save it. We'll do the same for our total items sold price using the metric:

```
productiona.hosts.tornado-db.dbi.items.items_sold_total_price
```

And we give it a title of `Total price of items sold`. Then again for the metric:

```
productiona.hosts.tornado-db.dbi.items.items_bought_total_cost
```

And we give it a title of `Total cost of item bought`.

This will be the top row of our Tornado dashboard that will allow us to see the business metrics of our Tornado application.

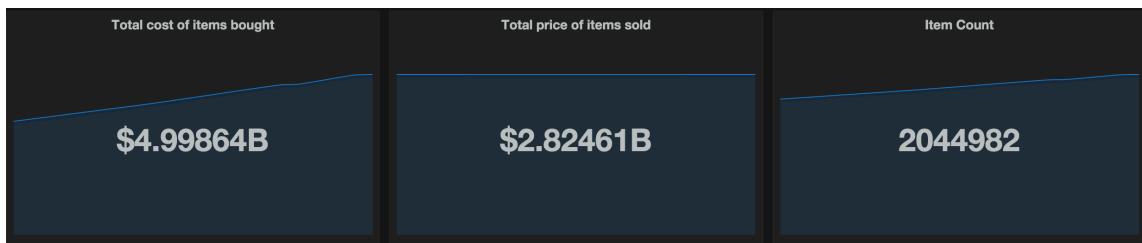


Figure 13.3: Tornado business metrics panels

We also want to create some graphs showing the current dollar value of items bought and sold. We can use a method like:

Listing 13.27: The current value of items bought

```
alias(sumSeriesWithWildcards(productiona.hosts.*.statsd.gauge.  
tornado.api.item.bought.total, 2), 'Tornado API servers')
```

Here we've used a wildcard, `*`, to select all of the Tornado API metrics that record items bought:

```
productiona.hosts.*.statsd.gauge.tornado.api.item.bought.total
```

We've used two Graphite functions: `alias` and `sumSeriesWithWildcards`. The first function, `alias`, wraps around the outside of our formula. It aliases the `tornado-api1` and `tornado-api2` servers into a single alias: `Tornado API servers`. The second function performs a sum of a series of metrics, in this case adding all the bought item metrics from the Tornado API servers.

We then use this formula on both the bought and sold items metrics to create two new graphs.

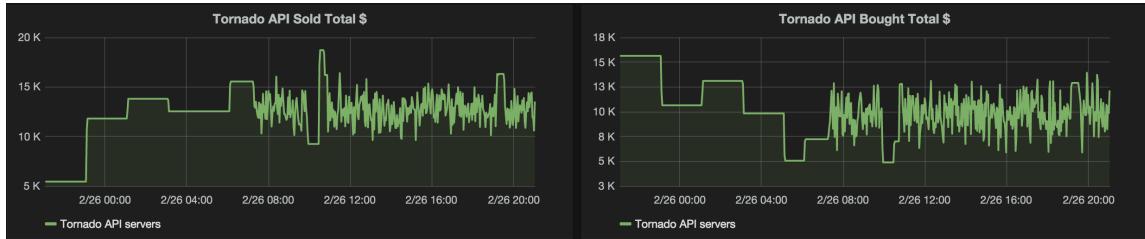


Figure 13.4: Tornado API bought and sold metrics panel

These new graphs show the total dollar value of all buying and selling transactions executed on Tornado API servers.

Let's add a few more graphs that address our developers' and operations team's needs. Let's start with the Tornado 5xx error percentage. Here we use the metric we created in Chapter 11:

```
tornado-proxy.haproxy.frontend.tornado-www.5xx_error_percentage
```

We want to alias it by node to see the host name.

Listing 13.28: The Tornado 5xx error percentage

```
aliasByNode(productiona.hosts.tornado-proxy.haproxy.frontend.  
tornado-www.5xx_error_percentage,2)
```

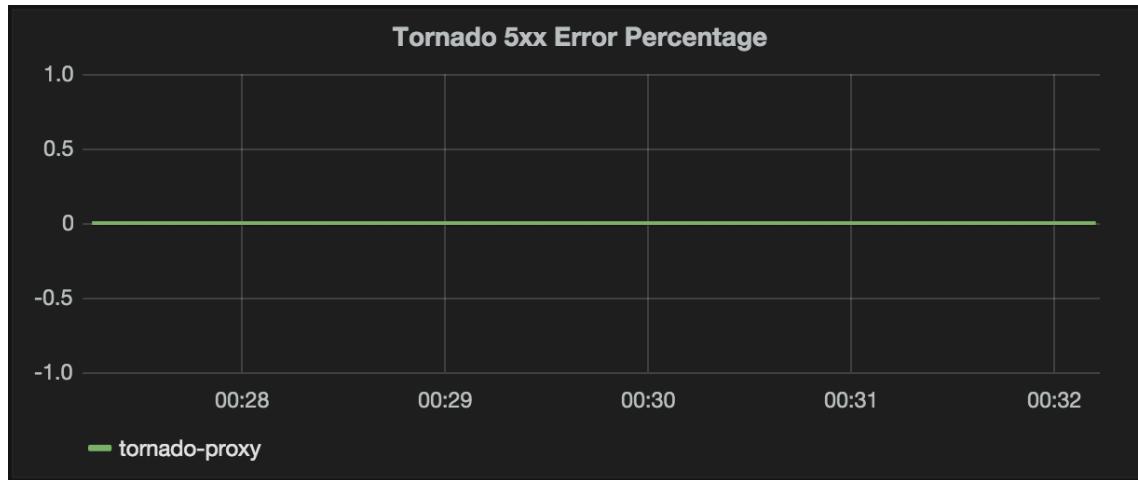


Figure 13.5: Tornado 5xx percentage error

Next let's create a graph for the insert query timing created from the `performance_schema` on `tornado-db`. We again alias by node to get the host name from the metric.

Listing 13.29: The MySQL database insertion query timing

```
aliasByNode(productiona.hosts.tornado-db.dbi.performance_schema.  
insert_query_time.99,2)
```

Then we create the graph.

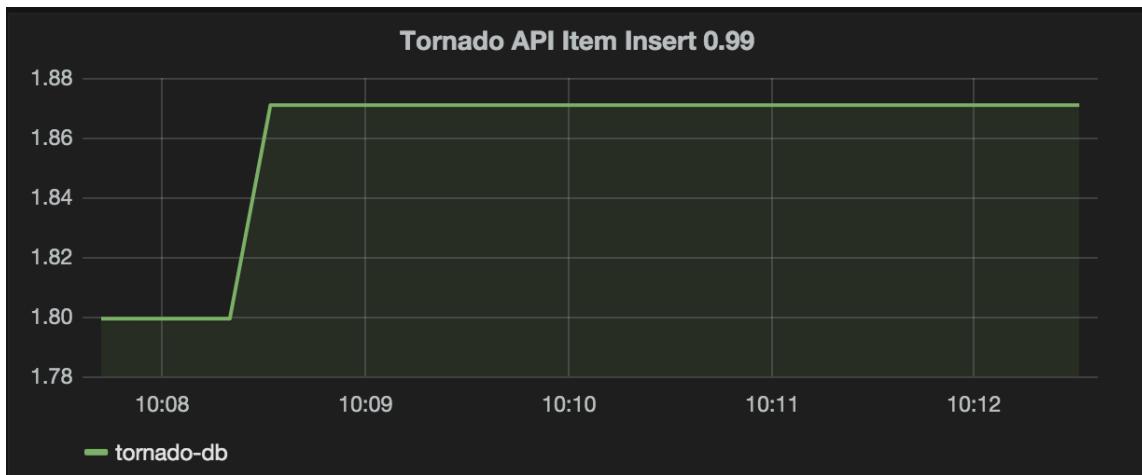


Figure 13.6: Tornado DB Insert query

The remaining graphs in the dashboard use the following configuration:

- Tornado API Requests Time 0.99

Listing 13.30: Tornado API Requests Time 0.99

```
aliasByNode(productiona.hosts.*.tornado.api.request.99,2)
```

- Tornado API Request Rate

Listing 13.31: Tornado API Request Rate

```
aliasByNode(productiona.hosts.*.tornado.api.request.rate,2)
```

- Tornado DB Tier CPU Usage

Listing 13.32: Tornado DB Tier CPU Usage

```
groupByNode(productiona.hosts.tornado-{redis,db}.cpu.{user,system},2,'sumSeries')
```

- Tornado App Tier CPU Usage

Listing 13.33: Tornado App Tier CPU Usage

```
groupByNode(productiona.hosts.tornado-{api1,api2}.cpu.{user,system},2,'sumSeries')
```

- Tornado Web Tier CPU Usage

Listing 13.34: Tornado Web Tier CPU Usage

```
groupByNode(productiona.hosts.tornado-{proxy,web1,web2}.cpu.{user,system},2,'sumSeries')
```

- Tornado Swap Used

Listing 13.35: Tornado Swap Used

```
aliasByNode(productiona.hosts.tornado-{proxy,web1,web2,api1,api2,redis,db}.swap.used, 2)
```

- Tornado Memory Used

Listing 13.36: Tornado Memory Used

```
aliasByNode(productiona.hosts.tornado-{proxy,web1,web2,api1,api2,  
redis,db}.memory.used,2)
```

- Tornado Load (shortterm)

Listing 13.37: Tornado Load (shortterm)

```
aliasByNode(productiona.hosts.tornado-{proxy,web1,web2,api1,api2,  
redis,db}.load.shortterm,2)
```

- Tornado Disk used on the / partition

Listing 13.38: Tornado Disk used on the / partition

```
aliasByNode(productiona.hosts.tornado-{proxy,web1,web2,api1,api2,  
redis,db}.df.root.percent_bytes.used, 2)
```

And that completes our Tornado operational dashboard. You can see that business, ops, and developers can all make use of the dashboard to gain insight into the current state of the Tornado application.

If you can want to look at it in more detail, you can find the JSON source code for our dashboard in [the book's source code](#).

Expanding monitoring beyond Tornado

One of the most useful things about Riemann is that much of our configuration and many of our checks are reusable across applications and services. If we had many iterations of the Tornado app we could easily update our `checks` function's `let` statement to select all of the relevant hosts. Let's assume our Tornado applications were numbered (and all Tornado-application events were tagged by collectd):

Listing 13.39: Expanding Tornado

```
(defn checks
  "Handles events for Tornado applications"
  []
  (let [web-tier-hosts #"(tornado(\d+)?-(proxy|web1|web2))"
        app-tier-hosts #"(tornado(\d+)?-(api1|api2))"
        db-tier-hosts #"(tornado(\d+)?-(db|redis))"]

    (splitp re-matches host
      web-tier-hosts (webtier)
      app-tier-hosts (apptier)
      db-tier-hosts (datatier)
      #(info "Catchall" (:host %)))))
```

This would match hosts from the `tornado-xxx`, `tornadol-xxx`, or `tornadol0-xxx` applications.

To be more sophisticated we could also replace the regular expression lookups for `web-tier-hosts` and the others with lookups of an external data source from a configuration management tool like PuppetDB or service discovery tools like Zookeeper or Consul.

Using this, and by modifying the regular expression, we could also match on events from a wide variety of applications. For example, we could create a generic set of three-tier web application checks by renaming our `tornado` namespace.

Listing 13.40: Renaming the Tornado namespace

```
(ns examplecom.app.webapps  
  "Monitoring streams for Web Applications"  
  ...)
```

We can then update the stream inside our `riemann.config` to select events from multiple applications.

Listing 13.41: Sending more than our Tornado events to be checked

```
(tagged-any ["tornado" "avalanche"]  
  (webapps/checks))
```

The `tagged-any` stream will match on events that have any of the tags listed; here events with either the `tornado` or the `avalanche` tags will be sent to the `application/checks` function. Assuming our host selection is also updated, the events will be routed to the relevant tier checks. We would then update any application-specific metric names in our streams to be generic.

This allows us reuse or build generic checks with our Riemann configuration.

Summary

In this chapter we monitored the Data tier of our Tornado application. We've again looked at both business and application performance metrics. We've con-

structured appropriate checks and thresholds for letting us know when the Data tier is available and when it is not performing in accordance with our expectations.

We've also exposed some of our metrics as visualizations in a dashboard for the business owners, as well as adding other graphs that meet the monitoring needs of our developers and operations teams.

Finally, combined, these last three chapters represent the capstone of the Art of Monitoring. They demonstrate how to combine the tools and techniques we've built in the preceding chapters to build a comprehensive monitoring framework. With these tools, techniques, and the real-world examples we've provided, you should be able to build and extend the framework to monitor your own environment.

Chapter A

An Introduction to Clojure and Functional Programming

Riemann is configured using a Clojure-based configuration file. This means your configuration file is actually processed as a Clojure program. So, to process events and send notifications and metrics, you'll be writing Clojure. Don't panic! You won't need to become a full-fledged Clojure developer to use Riemann. We'll teach you what you need to know in order to use Riemann. Additionally, Riemann comes with a lot of helpers and shortcuts that make it easier to write Clojure to do what we need to process our events.

Let's learn a bit more about Clojure and help you get started with Riemann. Clojure is a dynamic programming language that targets the Java Virtual Machine. It's a dialect of Lisp and is largely a [functional programming language](#).

[Functional programming](#) is a programming style that focuses on the evaluation of mathematical functions and steers away from changing state and mutable data. It's highly declarative, meaning you build programs from expressions that describe "what" a program should accomplish rather than "how" it accomplishes things.

NOTE Languages that describe more of the “how” are called imperative languages.

Examples of declarative programming languages include SQL, CSS, regular expressions, and configuration management languages like Puppet and Chef. Let’s look at a simple example.

Listing A.1: A declarative statement

```
SELECT user_id FROM users WHERE user_name = "Alice"
```

In this SQL query we’re asking for the `user_id` for `user_name` of `Alice` from the `users` table. The statement is asking a declarative “what” question. We don’t really care about the “how”; the database engine takes care of those details.

In addition to their declarative nature, functional programming languages try to eliminate all side effects from changing state. In a functional language, when you call a function its output value depends only on the inputs to the function. So if you repeatedly call function `f` with the same value for argument `x`, `f(x)`, it will produce the same result every time. This makes functional programs easy to understand, test, and predict. Functional programming languages call functions that operate like this “pure” functions.

The best way to get started with Clojure is to understand the basics of its syntax and types. Let’s get an introduction now.

WARNING This is going to be a high-level introduction to Clojure. It’s designed to give you the knowledge and recognition of various syntax and expressions to allow you to work with Riemann. We will not be teaching you how to

develop in Clojure in this book.

A brief introduction to Clojure

Let's step through the Clojure basic syntax and types. We'll also show you a tool called REPL that can help you test and build your Clojure snippets. REPL (short for read–eval–print loop) is an interactive programming shell that takes single expressions, evaluates them, and returns the results. It's a great way to get to know Clojure.

NOTE If you're from the Ruby world then REPL is just like `irb`. Or, in Python, it's like when you launch the `python` binary interactively.

We install REPL via a tool called [Leiningen](#). Leiningen is an automation tool for Clojure that helps you automate the build and management of Clojure projects.

Installing Leiningen

In order to install Leiningen we'll need to have Java installed on the host. The prerequisite Java packages we installed on Ubuntu and Red Hat for Reimann will be sufficient for Leiningen too.

We're going to download a Leiningen binary called `lein` to install it. Let's download that into a `bin` directory under our home directory.

Listing A.2: Getting lein

```
$ mkdir -p ~/bin  
$ cd ~/bin  
$ curl -olein https://raw.githubusercontent.com/technomancy/  
leiningen/stable/bin/lein  
$ chmod a+x lein  
$ export PATH=$PATH:$HOME/bin
```

Here we've created a new directory called `~/bin` and changed into it. We've then used the `curl` command to download the `lein` binary and the `chmod` command to make it executable. Lastly, we've added our `~/bin` directory to our path so that we can find the `lein` binary.

TIP The addition of the `/bin` directory assumes you're in a Bash shell. It's also temporary to your current shell. You'd need to add the path to your `.bashrc` or a similar setup for your shell.

Next we need to run `lein` to auto-install its supporting libraries.

Listing A.3: Auto-installing lein

```
$ lein  
...
```

This will download Leiningen's supporting Jar file.

Finally, we run REPL using the `lein repl` sub-command.

Listing A.4: Launching REPL

```
$ lein repl  
...  
user=>
```

This will download Clojure itself (in the form of its Jar file) and launch our interactive Clojure shell.

Clojure syntax and types

Let's use this interactive shell to look at some of the syntax and functions we've just learned about. We'll start by opening our shell.

Listing A.5: The REPL shell

```
user=>
```

Now we try a simple expression.

Listing A.6: Our first Clojure value

```
user=> nil  
nil
```

The `nil` expression is the simplest value in Clojure. It represents literally nothing.

We can also specify an integer value.

Listing A.7: Our first Clojure integer

```
user=> 1  
1
```

Or a string.

Listing A.8: Our first Clojure string

```
user=> "hello Ms Event"  
"hello Ms Event"
```

Or Boolean values.

Listing A.9: Our first Clojure Booleans

```
user=> true  
true  
user=> false  
false
```

Clojure functions

While interesting, these values aren't exciting on their own. To do some more interesting things we can use Clojure functions. A function is structured like this:

Listing A.10: The Clojure function syntax

```
(function argument argument)
```

TIP If you're used to the Ruby or Python world, a function is broadly the equivalent of a method.

Let's look at a function in action by doing something with some values: adding two integers together.

Listing A.11: Our first Clojure function

```
user=> (+ 1 1)  
2
```

In this case we've used the `+` function and added `1` and `1` together to get `2`.

But there's something about this structure that might look familiar to you if you've used other programming languages. Our function looks just like [a list](#). This is because it is! Our expression might add two numbers together, but it's also a list of three items in a valid list data structure.

NOTE Technically it's an [s-expression](#).

This is a feature of Clojure called [homoiconicity](#), sometimes described as: "code

is data, data is code.” This concept is inherited from Clojure’s parent language, [Lisp](#).

Homoiconicity means that the program’s structure is similar to its syntax. In this case Clojure programs are written in the form of lists. Hence you can gain insight into the program’s internal workings by reading its code. This also makes [metaprogramming](#) really easy because Clojure’s source code is a data structure and the language can treat it like one.

Now let’s look more closely at the `+` function. Each function is a symbol. A symbol is a bare string of characters, like `+` or `inc`. Symbols have short names and full names. The short name is used to refer to it locally—for example, `+`. The full name, or perhaps more accurately the fully qualified name, gives you a way to refer to the symbol unambiguously from anywhere. The fully qualified name of the `+` symbol is `clojure.core/+`. The `clojure.core` is the fundamental library of the Clojure language. We refer to `+` in its fully qualified form here:

Listing A.12: The fully qualified `+` function

```
user=> (clojure.core/+ 1 1)
2
```

Symbols refer to other things; generally they point to values. Think about them as a name or identifier that points to a concept: `+` is the name, “adding” is the concept. When Clojure encounters a symbol it evaluates it by looking up its meaning. If it can’t find a meaning it’ll generate an error message, for example:

Listing A.13: Unable to resolve symbol

```
user=> (bob 1 2)
CompilerException java.lang.RuntimeException: Unable to resolve
symbol: bob in this context, compiling:(NO_SOURCE_PATH:1:1)
```

Clojure also has a syntax for stopping that evaluation. This is called quoting, and it is achieved by prefixing the expression with a quotation mark: `'`.

Listing A.14: Quoting a symbol

```
user=> '(+ 1 1)
(+ 1 1)
```

This returns the symbol itself without evaluating it. This is important because often we want to do things, review things, or test things without evaluating.

For example, if we need to determine what type of thing something is in Clojure we use the `type` function and quote the function like so:

Listing A.15: The type function

```
user=> (type '+)
clojure.lang.Symbol
```

Here we see that `+` is a Clojure language symbol.

Lists

Clojure also has a variety of data structures. Especially useful to us will be collections. Collections are groups of values, for example, a list or a map.

Let's start by looking at lists. Lists are core to all Lisp-based languages (Lisp means "LISt Processing"). As we discovered above, Clojure programs are essentially lists. So we're going to see a lot of them!

Lists have zero or more elements and are wrapped in parentheses.

Listing A.16: A Clojure list

```
user=> '(a b c)
(a b c)
```

Here we've created a list containing the elements `a`, `b`, and `c`. We've quoted it because we don't want it evaluated. If we didn't quote it then evaluation would fail because none of the elements, `a`, `b`, etc., are defined. Let's see that now.

Listing A.17: An unquoted Clojure list

```
user=> (a b c)
CompilerException java.lang.RuntimeException: Unable to resolve
symbol: a in this context, compiling:(NO_SOURCE_PATH:1:1)
```

We can do a few neat things with lists, such as adding an element using the `conj` function.

Listing A.18: Adding an element to a list

```
user=> (conj '(a b c) 'd)  
(d a b c)
```

You can see we've added a new element, **d**, to the front of the list. Why the front? Because a list is really a [linked list](#) and focuses on providing immediate access to the first value in the list. Lists are most useful for small collections of elements and when you need to read elements in a linear fashion.

We can also return values from a list using a variety of functions.

Listing A.19: Working with lists

```
user=> (first '(a b c))  
a  
user=> (second '(a b c))  
b  
user=> (nth '(a b c) 2)  
c
```

Here we've pulled out the first element, second element, and using the **nth** function, the third element.

This last, **nth**, function shows us a multi-argument function. The first argument is the list, **'(a b c)**, and the second argument is the index value of the element we want to return, here **2**.

TIP Like most programming languages Clojure starts counting from 0.

We can also create a list with the `list` function.

Listing A.20: Creating a list

```
user=> (list 1 2 3)
(1 2 3)
```

Vectors

Another collection available to us is the vector. Vectors are like lists but they are optimized for random access to the elements by index. Vectors are created by adding zero or more elements inside square brackets.

Listing A.21: A Clojure vector

```
user=> '[a b c]
[a b c]
```

Like lists, we again use `conj` to add to a vector.

Listing A.22: Adding an element to a vector

```
user=> (conj '[a b c] 'd)
[a b c d]
```

You'll note the `d` element is added at the end because a vector isn't focused on sequential access like a list.

There are some other useful functions we can use on lists and vectors. For example, to get the last element in a list or vector:

Listing A.23: Getting the last element in a vector

```
user=> (last '[a b c d])
d
```

Or count the elements:

Listing A.24: Counting elements in a vector

```
user=> (count '[a b c d])
4
```

Because vectors are designed to look up elements by index, we can also use them directly as functions, for example:

Listing A.25: Using a vector as a function

```
user=> ([1 2 3] 1)
2
```

Here we've retrieved the value, `2`, at index `1`.

We can create a vector with the `vector` function or change another data structure into a vector with the `vec` function.

Listing A.26: Creating or converting vectors

```
user=> (vector 1 2 3)
[1 2 3]
user=> (vec (list 1 2 3))
[1 2 3]
```

Sets

There's a final collection related to lists and vectors called a set. Sets are unordered collections of values, prefixed with # and wrapped in curly braces, { }. They are most useful for collections of values where you want to check if a value or values are present.

Listing A.27: A Clojure set

```
user=> '#{a b c}
#{c a b}
```

You'll notice the set was returned in a different order. This is because sets are focused on presence lookups so order doesn't matter quite so much.

Like lists and vectors we use the `conj` function to add an element to a set.

Listing A.28: Adding to a set

```
user=> (conj '#{a b c} 'd)  
#{a c b d}
```

Sets can never contain an element more than once, so adding an element that's already present does nothing. You can remove elements with the `disj` function.

Listing A.29: Removing an element from a set

```
user=> (disj '#{a b c d} 'd)  
#{a c b}
```

The most common operation with a set is to check for the presence of a specific value. For this we use the `contains?` function.

Listing A.30: Checking for a value inside a set

```
user=> (contains? '#{a b c} 'c)  
true  
user=> (contains? '#{a b c} 'd)  
false
```

Like a vector, you can also use the set itself as a function. This returns the value if it is present, or `nil` if it is not.

Listing A.31: Using the set as a function

```
user=> (#{} a b c) 'c)
c
user=> (#{} a b c) 'd)
nil
```

You can make a set out of any other collection with the **set** function.

Listing A.32: Making a set

```
user=> (set [a b c])
#{a c b}
```

Here we've made a set out of a vector.

Maps

The last data structure we're going to look at is the map. Maps are key/value pairs enclosed in braces. You can think about them as being equivalent to a hash.

Listing A.33: A Clojure map

```
user=> { :a 1 :b 2}
{:a 1, :b 2}
```

Here we've defined a map with two key/value pairs: **:a 1** and **:b 2**.

You'll note each key is prefixed with a `:`. This denotes another type of Clojure syntax: the keyword. A keyword is much like a symbol, but instead of referencing another value it is merely a name or label. It's highly useful in data structures like maps to do lookups—you look up the keyword and return the value.

We can use the `get` function to retrieve a value.

Listing A.34: Getting a Clojure map value

```
(get {:a 1 :b 2} :a)  
1
```

Here we've specified the keyword `:a` and asked Clojure if it is inside our map. It's returned the value in the key/value pair, `1`.

If the key doesn't exist in the map then Clojure returns `nil`.

Listing A.35: Getting a missing Clojure map value

```
user=> (get {:a 1 :b 2} :c)  
nil
```

The `get` function can also take a default value to return instead of `nil` if the key doesn't exist in that map.

Listing A.36: Getting a default value from a map

```
user=> (get {:a 1 :b 2} :c :novalue)  
:novalue
```

We can use the map itself as a function, as well.

Listing A.37: Using a map as a function

```
user=> ({:a 1 :b 2} :a)  
1
```

And we can use keywords as functions to look themselves up in a map.

Listing A.38: Using a keyword as a function

```
user=> (:a {:a 1 :b 2})  
1
```

To add a key/value pair to a map we use the `assoc` function.

Listing A.39: Using assoc to add a key/value

```
user=> (assoc {:a 1 :b 2} :c 3)  
{:a 1, :b 2, :c 3}
```

If a key isn't present then `assoc` adds it. If the key is present then `assoc` replaces the value.

Listing A.40: Replacing a key/value with assoc

```
user=> (assoc {:a 1 :b 2} :b 3)
{:a 1, :b 3}
```

To remove a key we use the **dissoc** function.

Listing A.41: Removing a key/value with dissoc

```
user=> (dissoc {:a 1 :b 2} :b)
{:a 1}
```

NOTE If you've come from the Ruby or Python world, the terms list, set, vector, and map might be a little new. But the syntax probably looks familiar. You can think about lists, vectors, and sets as being similar to arrays, and maps being hashes.

Strings

We can also work with strings. Clojure lets you turn pretty much any value into a string using the **str** function.

Listing A.42: The str function

```
user=> (str "holiday")
"holiday"
```

The `str` function turns anything specified into a string. We can also use it to concatenate strings.

Listing A.43: Concatenating a string

```
user=> (str "james needs " 2 " holidays")
"james needs 2 holidays"
```

Creating our own functions

Up until now we've run functions as stand-alone expressions. For example, here's the `inc` function that increments arguments passed to it:

Listing A.44: The inc function again

```
user=> (inc 1)
2
```

This isn't overly practical except to demonstrate how a function works. If we want do more with Clojure we need to be able to define our own functions. To do this, Clojure provides a function called `fn`. Let's construct our first function.

Listing A.45: The fn function

```
user=> (fn [a] (+ a 1))
```

So what's going on here? We've used the `fn` function to create a new function. The `fn` function takes a vector as an argument. This vector contains any arguments being passed to our function. Then we specify the actual action our function is going to perform. In our case we're mimicking the behavior of the `inc` function. The function will take the value of `a` and add `1` to it.

If we run this code now nothing will happen because `a` is currently unbound—we haven't defined a value for it. Let's run our function now.

Listing A.46: Running our first fn function

```
user=> ((fn [a] (+ a 1)) 2)  
3
```

Here we've evaluated our function and passed in an argument of `2`. This is assigned to our `a` symbol and passed to the function. The function adds `a`, now set to `2`, and `1` and returns the resulting value: `3`.

There's also a shorthand for writing functions that we'll see occasionally in the book.

Listing A.47: The fn function shortcut

```
user=> #(+ % 1)
```

This shorthand function is the equivalent of `(fn [x] (+ x 1))` and we can call it to see the result.

Listing A.48: Calling the fn function shortcut

```
user=> (#(+ % 1) 2)
3
```

Creating variables

But we're still a step from a named function, and we're missing an important piece: how do we define our own variables to hold values? Clojure has a function called `def` that allows us to do this.

Listing A.49: Creating a var

```
user=> (def smoker "joker")
#'user/smoker
```

The `def` statement does two things:

- It creates a new type of object called a var. Vars, like symbols, are references to other values. You can see our new var `#'user/smoker` returned as output of the `def` function.
- It binds a symbol to that var. Here the symbol `smoker` is bound to a var with a value of the string `"joker"`.

When we evaluate a symbol pointing to a var it is replaced by the var's value. But because `def` also creates a symbol, we can refer to our var like that too.

Listing A.50: Evaluating a symbol

```
user=> user/smoker
"joker"
user=> smoker
"joker"
```

Where did this `user/` come from? It's a Clojure namespace. Namespaces are a way Clojure organizes code and program structure. In this case the REPL creates a namespace called `user/` by default. Remember, we learned earlier that a symbol has both a short name—for example, `smoker`—that can be used locally to refer to it, and a full name. That full name, here `user/smoker`, would be used to refer to this symbol from another namespace.

We'll talk more about namespaces and use them to organize our Riemann configuration in other parts of this book. If you'd like to read more about them, there is an excellent explanation [in this post](#).

We can also use the `type` function to see the type of value the symbol references.

Listing A.51: Using the type function on the symbol

```
user=> (type smoker)
java.lang.String
```

Here we see that the value `smoker` resolves to is a string.

Creating named functions

Now, with the combination of `def` and `fn`, we can create our own named functions.

Listing A.52: Creating our first named function

```
user=> (def grow (fn [number] (* number 2)))  
#'user/grow
```

Firstly, we've defined a var (and symbol) called `grow`. Inside that we've defined a function. Our function takes a single argument, `number`, and passes that number to the `*` function, the mathematical multiplication operator in Clojure, and multiplies it by `2`.

Let's call our function now.

Listing A.53: Calling our grow function

```
user=> (grow 10)  
20
```

Here we've called the `grow` function and passed it a value of `10`. The `grow` function multiplies that value and returns the result: `20`. Pretty awesome eh?

But the syntax is a little cumbersome. Thankfully Clojure offers a shortcut to creating a var and binding it to a function called `defn`. Let's rewrite our function using this form.

Listing A.54: Using the defn form

```
user=> (defn grow [number] (* number 2))  
#'user/grow
```

That's a little neater and easier to read. Now how about we add a second argument? Let's make both the number to be multiplied and the multiplier arguments.

Listing A.55: Adding a second argument

```
user=> (defn grow [number multiple] (* number multiple))  
#'user/grow
```

Let's call our `grow` function again.

Listing A.56: Calling our grow function again

```
user=> (grow 10)  
ArityException Wrong number of args (1) passed to: user/grow  
clojure.lang.AFn.throwArity (AFn.java:429)
```

Oops, not enough arguments! Let's add the second argument.

Listing A.57: Calling grow with our second argument

```
user=> (grow 10 4)  
40
```

We can also add a doc string to our function to help us articulate what it does.

Listing A.58: Adding a second argument

```
(defn grow
  "Multiplies numbers - can specify the number and multiplier"
  [number multiple]
  (* number multiple)
)
```

We can access a function's doc string using the `doc` function.

Listing A.59: Using the doc function

```
user=> (doc grow)
-----
user/grow
([number multiple])
Multiplies numbers - can specify the number and multiplier
nil
```

The `doc` function tells us the full name of the function, the arguments it accepts, and returns the docstring.

That's the end of our introduction. We'll see and learn more Clojure elsewhere in the book.

Learning more Clojure

We recommend trying to get an understanding of the basics of Clojure to get the most out of Riemann. If you'd like to start to learn a bit about Clojure, Kyle Kings-

bury's excellent [Clojure from the ground up](#) series is a great place to start. This section is much an abbreviated summary of that tutorial, and we can't thank Kyle enough for writing it. A reading of his tutorial will greatly add to the knowledge we've shared here. For the purposes of this book, we recommend at least a solid reading of the first three posts in the series:

- The [Welcome](#) post.
- The post on [Basic types](#).
- The post on [Functions](#).

Also useful to writing good code is the [Clojure Style Guide](#).

TIP If you're interested in learning a bit more about the basics of Clojure, another good resource is [Learn Clojure](#).

List of Figures

1 License	5
2 ISBN	5
1.1 Monitoring Maturity Model Distribution.	13
1.2 Traditional Monitoring	14
2.1 Monitoring framework	22
2.2 A sample plot	28
2.3 A sample gauge	29
2.4 A sample counter	30
2.5 A sample timer	30
2.6 A histogram example	32
2.7 An aggregated collection of metrics	33
2.8 The flaw of averages - Copyright Jeff Danzinger	44
2.9 Response time average	45
2.10 Response time average Mk II	46
2.11 Response time average and median	47
2.12 Response time average and median Mk II	48
2.13 The empirical rule	49
2.14 Response time average, median, and percentiles	51
2.15 Response time average, median, and percentiles Mk II	52
3.1 Event Routing	55
3.2 Metrics Architecture	58
3.3 Connecting Riemann servers	91

3.4 Email notification	109
3.5 Sharded Riemann	119
4.1 Metrics processing and visualization	122
4.2 Metrics Architecture	127
4.3 Graphite Architecture	128
4.4 Graphite on Ubuntu installation prompt	130
4.5 Carbon daemon ports and architecture	145
4.6 Testing Graphite-API	168
4.7 The Grafana Console login	169
4.8 The Grafana Console	170
4.9 The Grafana Console menu	171
4.10 The Grafana Data Sources menu	172
4.11 Adding a Grafana data source	173
4.12 Testing the Grafana connection	173
4.13 The Grafana console again	183
4.14 The Grafana Dashboard menu	184
4.15 The Grafana Settings menu	184
4.16 The Riemann dashboard menu	184
4.17 Adding our first graph	185
4.18 Our first graph	185
4.19 The graph editing controls	186
4.20 Selecting metrics	188
4.21 Our graphed Riemann rate metric	189
4.22 The metric dropdown menu	189
4.23 Our graphed Riemann rate metric via the edit box	190
4.24 Our graph and dashboard	190
4.25 A redundant Graphite architecture	192
4.26 Configuring time zone on Ubuntu	197
4.27 Selecting the UTC time zone on Ubuntu	198
5.1 Collecting host-based data	209

5.2 Our collectd architecture	212
6.1 Creating a new host dashboard	288
6.2 Naming our new host dashboard	289
6.3 Creating our first host graph	290
6.4 Naming our CPU Usage graph	290
6.5 Our final CPU Usage graph	294
6.6 The time-series resolution controls	294
6.7 The Memory usage graph	296
6.8 The graphitea CPU graph	296
6.9 The single host Metrics tab	297
6.10 The single host CPU graph	297
6.11 A Hosts Dashboard	298
7.1 Host evolution	304
8.1 Logstash Architecture	346
8.2 The Elasticsearch Head plugin	367
8.3 Configuring Kibana	389
8.4 The Kibana Dashboard	390
9.1 Application and business logic monitoring	441
9.2 Our application monitoring architecture	456
9.3 External monitoring	496
9.4 Creating the aom-rails dashboard	507
9.5 Naming the aom-rails dashboard	508
9.6 Creating an Annotation	509
9.7 Configuring our Annotation	510
9.8 The Annotation in the dashboard	510
9.9 Annotated user count graph	511
10.1 A Riemann email notification	517
10.2 Our scripted dashboard	539

10.3 A PagerDuty incident	554
11.1 Current Monitoring Framework Architecture	570
11.2 Tornado's architecture	573
11.3 The HAProxy HTTP console	578
11.4 Tornado Slack notifications for Nginx and HAProxy	623
12.1 Graphing our Tornado API rate	661
13.1 Adding a single stat panel	689
13.2 Tornado item count panel	689
13.3 Tornado business metrics panels	690
13.4 Tornado API bought and sold metrics panel	691
13.5 Tornado 5xx percentage error	692
13.6 Tornado DB Insert query	693

Listings

1 Sample code block	4
2.1 Sample Nagios notification	35
3.1 Installing Java on Ubuntu	59
3.2 Checking Java is installed on Ubuntu	60
3.3 Fetching the Riemann DEB package	60
3.4 Installing the Riemann package on Ubuntu	60
3.5 Installing Java and prerequisites on RHEL	61
3.6 Checking Java is installed on Red Hat	62
3.7 Fetching the Riemann RPM package	62
3.8 Installing the Riemann package on RHEL	62
3.9 Starting and stopping Riemann	64
3.10 Running Riemann interactively	64
3.11 Installing supporting tools prerequisites on Ubuntu	66
3.12 Installing supporting tools prerequisites on RHEL	66
3.13 Installing Riemann's supporting tools	66
3.14 New /etc/riemann/riemann.config configuration file	68
3.15 Riemann logging stanza	69
3.16 The let form	70
3.17 Exposing Riemann on all interfaces	71
3.18 Changing the Riemann port	72
3.19 Connecting to the Riemann REPL server	72
3.20 SIGHUP from the Riemann REPL server	73

3.21 Restarting Riemann	73
3.22 Example Riemann event	74
3.23 Child streams example	75
3.24 Example Apache Riemann event	76
3.25 Example expired Apache Riemann event	77
3.26 More of our default <code>riemann.config</code> configuration file	77
3.27 Copying more keys into expired events	78
3.28 Using the Riemann default function	79
3.29 Logging to the Riemann log file	79
3.30 A Riemann log event	79
3.31 Adding a prefix to Riemann logs entries	80
3.32 Limiting Riemann log entries	80
3.33 A filtered Riemann log event	80
3.34 The Riemann <code>prn</code> function	81
3.35 Reloading Riemann to enable our new configuration	81
3.36 Riemann internal events	82
3.37 The <code>riemann-health</code> command	83
3.38 The <code>riemann-health --host</code> option	83
3.39 Our incoming Riemann data	84
3.40 A Riemann-health disk event	84
3.41 Our first monitoring check	85
3.42 Our prefixed warning event	86
3.43 Using the <code>where</code> stream with a regular expression	87
3.44 Using the <code>where</code> stream with booleans	87
3.45 The tagged-any stream	88
3.46 Using the <code>where</code> stream for complex matches	88
3.47 Referring to an optional example field in Riemann	88
3.48 Referring to an optional field in Riemann	89
3.49 The optional type field in Riemann	89
3.50 Referring to an optional field in Riemann	89
3.51 Using the <code>where</code> stream with math	90

3.52 Using the where stream for a range query	90
3.53 Updated Riemann configuration	93
3.54 Requiring the Riemann client	94
3.55 Added downstream binding to Riemann	94
3.56 Riemann client forwarding configuration	95
3.57 The where filtering stream for forwards	96
3.58 The downstream riemannmc server	97
3.59 Restarting Riemann to enable forwarding	98
3.60 Riemann agg events on riemannmc or riemannb	98
3.61 Combined events from upstream and downstream	99
3.62 The downstream riemannmc server configuration	101
3.63 Clojure namespace format	102
3.64 Creating the examplecom.etc namespace path	103
3.65 Creating the email.clj file	103
3.66 Requiring the Riemann functions	103
3.67 Referring functions	105
3.68 Configuring email notifications in Riemann	105
3.69 Configuring an SMTP server for Postal	106
3.70 Adding our email function to Riemann	107
3.71 Our expired Riemann event filter stream	108
3.72 The riemannmc expired streams event	108
3.73 Our throttled expired Riemann event filter	109
3.74 The throttle stream	110
3.75 A rollup of our expired Riemann events	110
3.76 Revisiting our riemannmc configuration	112
3.77 Adding a tap to our riemannmc index	113
3.78 Adding tests to our riemannmc configuration	114
3.79 Running the Riemann tests	115
3.80 Download the Riemann syntax checker	116
3.81 Build the Riemann syntax checker	116
3.82 Syntax check a Riemann configuration	117

3.83 Configuring additional RAM	117
4.1 The Whisper file layout	125
4.2 Installing the Graphite package on Ubuntu	129
4.3 Adding the EPEL repository for Riemann	130
4.4 Installing Graphite prerequisite packages on Red Hat	131
4.5 Installing Graphite packages on Red Hat	131
4.6 Creating new Graphite user and group on Red Hat	132
4.7 Moving the /var/lib/carbon directory	132
4.8 Changing the ownership of /var/log/carbon	133
4.9 Removing the carbon user	133
4.10 Adding the Graphite-API PackageCloud key	134
4.11 Adding the PackageCloud exoscale repository listing	134
4.12 Updating APT for Graphite-API	134
4.13 Installing the graphite-api package on Ubuntu	135
4.14 Install Graphite-API prerequisite packages on Red Hat	135
4.15 Installing Graphite-API via pip	135
4.16 Adding the Grafana repository listing	136
4.17 Adding the PackageCloud key	137
4.18 Installing the Grafana package	137
4.19 Creating the Grafana Yum repository	137
4.20 Yum repository definition for Grafana	138
4.21 Installing Grafana via Yum	138
4.22 Editing the Carbon settings	140
4.23 Directory configuration for Carbon	141
4.24 Setting the Carbon user and log rotation	141
4.25 Configuring the Carbon Cache daemon	142
4.26 The Carbon plaintext protocol	143
4.27 The Carbon plaintext protocol metric example	143
4.28 Sending a plaintext metric via Netcat to Graphite	143
4.29 The Graphite pickle protocol	144
4.30 Carbon Cache instance definitions	144

4.31 A third carbon-cache instance	146
4.32 The carbon-relay daemon configuration	146
4.33 Adding a third instance destination to carbon-relay	147
4.34 Configuring Carbon storage schemas	150
4.35 Carbon storage schema	151
4.36 Sample retention periods for Graphite	151
4.37 Creating a new default Graphite schema	152
4.38 Annoying Graphite log message	153
4.39 Creating an empty storage aggregation file	153
4.40 Creating a new default Graphite schema	154
4.41 Downloading the calculator	154
4.42 The whisper-calculator.py	155
4.43 Download the Carbon Cache init script on Ubuntu	156
4.44 Install the Carbon Cache init script on Ubuntu	156
4.45 Enable the Carbon Cache init script on Ubuntu	156
4.46 Install the Carbon relay init script on Ubuntu	157
4.47 Enable Graphite at startup on Ubuntu	157
4.48 The /etc/default/graphite-carbon file	158
4.49 Starting the Carbon daemons on Ubuntu	158
4.50 systemd unit file for the Carbon Cache daemon	159
4.51 Enabling and starting the systemd Carbon Cache daemons	160
4.52 systemd unit file for the Carbon relay daemon	161
4.53 Enabling and starting the systemd Carbon relay daemon	161
4.54 Remove the old systemd unit files for Carbon	162
4.55 The /etc/graphite-api.yaml file	163
4.56 Creating the /var/lib/graphite/api_search_index file	164
4.57 Restarting the Graphite-API on Ubuntu	165
4.58 Systemd script for Graphite-API	166
4.59 Gunicorn worker calculations	167
4.60 Enabling and starting the systemd Graphite-API daemons	167
4.61 Starting the Grafana Server	168

4.62 Curling the Graphite-API	170
4.63 Creating the Riemann Graphite snippet file	174
4.64 The graphite.clj file	174
4.65 Graphite metrics format	175
4.66 New Graphite metrics format	175
4.67 The add-environment-to-graphite function	176
4.68 The graph var	177
4.69 Riemann Graphite configuration for riemann	179
4.70 Reloading Riemann to enable Graphite	180
4.71 Riemann connecting to Graphite	180
4.72 Metrics being created on graphitea	181
4.73 metric name formatting	181
4.74 The Riemann streams latency path	181
4.75 The metric path tree	182
4.76 Our Riemann Whisper files	182
4.77 Graphite datapoints in the updates log	183
4.78 The Riemann rate path	187
4.79 The first element of the metric path	188
4.80 Installing the Graphite packages on our relay-only host	193
4.81 The carbon-relay only host	194
4.82 The updated graph var	195
4.83 Setting the time zone on Ubuntu	197
4.84 The time zone configuration output on Ubuntu	198
4.85 Installing NTP on Ubuntu	199
4.86 Initial time sync on Ubuntu	199
4.87 Running the timedatectl command on Red Hat	200
4.88 Checking the time zone on Red Hat	200
4.89 Installing NTP on Red Hat	201
4.90 Enabling the NTP service on Red Hat	201
4.91 Initial time sync on Red Hat	201
4.92 Starting the NTP service on Red Hat	201

4.93 Enabling NTP with timedatectl	202
4.94 Checking NTP is enabled with timedatectl	202
4.95 Checking the NTP time status	203
5.1 Installing collectd on Ubuntu	215
5.2 Installing collectd on Ubuntu	215
5.3 Testing collectd on Ubuntu	215
5.4 Adding the EPEL repository for Riemann	216
5.5 Installing collectd on Red Hat	216
5.6 Testing collectd on Red Hat	217
5.7 Our initial collectd configuration	219
5.8 The Graphite storage schema	220
5.9 Loading collectd plugins	222
5.10 Creating the /etc/collectd.d directory	223
5.11 Creating the /etc/collectd.d/cpu.conf file	225
5.12 Configuring the cpu plugin	225
5.13 Creating the /etc/collectd.d/memory.conf file	226
5.14 Configuring the Memory plugin	226
5.15 Creating the /etc/collectd.d/df.conf file	227
5.16 Configuring the df plugin	227
5.17 Configuring another mount point for the df plugin	228
5.18 Configuring the df plugin to monitor a device	228
5.19 Configuring the df plugin to monitor everything	229
5.20 Creating the /etc/collectd.d/swap.conf file	229
5.21 Configuring the swap plugin	230
5.22 Creating the /etc/collectd.d/interface.conf file	230
5.23 The interface.conf file	230
5.24 Configuring the interface plugin to only monitor one interface	231
5.25 Ignoring interfaces	231
5.26 Creating the /etc/collectd.d/protocols.conf file	232
5.27 The protocols.conf file	232
5.28 Creating the /etc/collectd.d/load.conf file	232

5.29 The load.conf file	233
5.30 Creating the processes.conf configuration file	233
5.31 The processes.conf file	234
5.32 Overriding the global interval	234
5.33 The ProcessMatch option	235
5.34 Using the ProcessMatch option	235
5.35 Using the ProcessMatch option for Riemann	236
5.36 Checking for Riemann in the ps command	236
5.37 The processes threshold	237
5.38 A collectd notification	238
5.39 The ps_count metric	239
5.40 Specifying a warning and a failure	240
5.41 Matching multiple thresholds	241
5.42 The carbon-cache collect failure event	242
5.43 The write_riemann.conf configuration file	243
5.44 Configuring the write_riemann plugin	244
5.45 Configuring a second Node block	246
5.46 Enabling and starting collectd on Ubuntu	247
5.47 Enabling and starting collectd on Red Hat	248
5.48 The collectd log file	248
5.49 Our current riemann.config	249
5.50 The collectd events in Riemann	250
5.51 Sending collectd metrics to Graphite	253
5.52 The collectd metrics in Graphite	254
5.53 A processes metric	255
5.54 Creating the service rewrite rules	255
5.55 Rewriting services inside Riemann	257
5.56 The service rewrite line	258
5.57 Original collectd to graph where filter	259
5.58 Updated collectd to graph stream	260
5.59 Original metric	260

5.60 Rewritten metric	261
6.1 Detecting collectd down or expired	264
6.2 A collectd notification event	265
6.3 Detecting changed state	266
6.4 Adjusting the :service field	267
6.5 Finding expired ps_count events	268
6.6 Added collectd monitoring to our Riemann configuration	269
6.7 The cpu/percent metric	271
6.8 Basic CPU monitoring	272
6.9 Basic Memory monitoring	272
6.10 The df plugin percent_bytes-used metric	273
6.11 Basic disk monitoring	273
6.12 Median CPU monitoring over ten seconds	275
6.13 Percentile CPU monitoring over ten seconds	277
6.14 The percentile events	278
6.15 Creating a file to hold our custom check functions	279
6.16 Our first custom check function	280
6.17 A child stream explained	282
6.18 Using the check_threshold function	283
6.19 Percentile CPU monitoring over ten seconds	284
6.20 Using the check_percentiles function	285
6.21 A set of monitoring checks	286
6.22 CPU Usage graph definition	291
6.23 Graphite functions	291
6.24 A Graphite series list	292
6.25 The 0-indexed metric name	292
6.26 The groupByNode callback	293
6.27 The sumSeries function output	293
6.28 Memory Usage graph definition	295
7.1 The docker stats command	308
7.2 Querying the Docker API stats endpoint	309

7.3 Downloading the Docker collectd plugin	311
7.4 Unzip the master.zip file	312
7.5 Rename the Docker collectd plugin directory	312
7.6 Installing the docker-collectd prerequisites via pip	313
7.7 Creating the /etc/collectd.d/docker.conf file	313
7.8 Configuring the Docker collectd plugin	314
7.9 Adding the TypesDB default configuration	315
7.10 Monitoring the Docker daemon	315
7.11 Restarting collectd to enable the Docker plugin	316
7.12 The Docker collectd plugin log output	316
7.13 Starting our Redis container	317
7.14 Running docker ps	317
7.15 A where filter for our Docker collectd events	318
7.16 A Docker collectd Riemann event	318
7.17 The updated collectd Graphite where stream	320
7.18 The Docker collectd plugin smap and graph	321
7.19 The updated Docker collectd :type Riemann event	322
7.20 The updated Docker collectd :type_instance Riemann event	322
7.21 The graph var from Chapter 4	323
7.22 The updated graph var	324
7.23 A Dockerfile label	325
7.24 Adding a label at runtime	325
7.25 Rerunning the redis container	326
7.26 Inspecting the redis container	326
7.27 The Docker collectd plugin format	327
7.28 The plugin-map function	328
7.29 The label map	329
7.30 The docker-attributes function	329
7.31 Destructuring keys	330
7.32 The if-let binding	330
7.33 The if-let conditions	331

7.34 The docker-parse-service-host function	332
7.35 Processing our Docker events	332
7.36 A rewritten Docker event Mark II	333
7.37 Updated our Graphite metric name write	334
7.38 The existing Carbon schemas	335
7.39 A new storage schema for Docker events	336
7.40 Calculating our new Docker metric file size	336
7.41 A find command to clean up old Graphite metrics	337
7.42 A find command to clean up Docker specific Graphite metrics	338
7.43 Using the check_percentiles function for Docker	339
8.1 Installing Java on Debian and Ubuntu	347
8.2 Installing Java on Red Hat	347
8.3 Testing Java is installed	348
8.4 Getting the Logstash public key on Ubuntu	348
8.5 Adding the Logstash packages	349
8.6 Updating Apt and installing the Logstash package	349
8.7 Getting the Logstash public key on Red Hat	349
8.8 The Logstash Yum configuration	350
8.9 Installing Logstash on Red Hat	350
8.10 Testing Logstash is installed	351
8.11 Creating the logstash.conf file	352
8.12 The logstash.conf configuration file	352
8.13 Testing Logstash interactively	353
8.14 Testing Logstash configuration validity	354
8.15 Our first Logstash event	354
8.16 Installing Java on Debian and Ubuntu	356
8.17 Testing Java is installed	356
8.18 Installing the public key for Elasticsearch	357
8.19 Adding the Elasticsearch Apt repository	357
8.20 Installing Elasticsearch on Ubuntu	357
8.21 Installing Java on Red Hat for Elasticsearch	358

8.22 Adding the Elasticsearch public key on Red Hat	358
8.23 Adding the Yum repo for Elasticsearch	358
8.24 The Yum repo for Elasticsearch	359
8.25 Installing Elasticsearch on Red Hat	359
8.26 Testing the elasticsearch binary	360
8.27 Checking Elasticsearch is running	361
8.28 Elasticsearch status information	361
8.29 Elasticsearch stats page	362
8.30 Initial cluster and node names	363
8.31 New cluster and node names	363
8.32 Configuring our Elasticsearch cluster	365
8.33 Restarting Elasticsearch to enable clustering	365
8.34 Installing an Elasticsearch plugin	366
8.35 Granting access to logs on Ubuntu	368
8.36 Granting access to logs on CentOS	368
8.37 The updated logstash.conf configuration file	369
8.38 File input globbing	370
8.39 File recursive globbing	370
8.40 Restarting Logstash to enable file monitoring	371
8.41 Testing with logger	371
8.42 A syslog log event	372
8.43 Adding our first Logstash filters	374
8.44 Logstash conditional selection	375
8.45 Our Syslog parsing filters	376
8.46 The grok filter	377
8.47 The MONTH pattern	378
8.48 Our original syslog log event	379
8.49 Our grok-parsed event	379
8.50 The syslog_pri parsed event	381
8.51 The date filter	382
8.52 A Logstash index	383

8.53 Showing the current Elasticsearch mapping	384
8.54 Showing index-specific mappings	384
8.55 Adding the Kibana APT repository	386
8.56 Updating the package list for Kibana	387
8.57 Installing Kibana via apt-get	387
8.58 Starting Kibana at boot	387
8.59 The Kibana kibana.yml configuration file	388
8.60 Updating the Elasticsearch instance	388
8.61 Running Kibana	389
8.62 Adding Syslog receivers	392
8.63 Restarting Logstash to enable new inputs	393
8.64 Running netstat	393
8.65 A Syslog message	394
8.66 Creating an RSyslog snippet for forwarding	396
8.67 Configuring RSyslog for Logstash	396
8.68 Specifying RSyslog facilities or priorities	397
8.69 Restarting RSyslog	397
8.70 Collecting the riemann.log file	398
8.71 Monitoring files with the imfile module	399
8.72 Monitoring files with an imfile wildcard	399
8.73 Monitoring the collectd log file	400
8.74 Configuring the DOCKER_OPTS option for logging	401
8.75 Docker logging tags	402
8.76 Restarting Docker to enable logging	403
8.77 Our Logstash Syslog receivers	404
8.78 Grok parse failed Docker event	405
8.79 Docker log message	406
8.80 An updated Docker grok regex	407
8.81 Updated date filter for Docker	408
8.82 Restarting Logstash to enable Docker logging	408
8.83 A properly parsed Docker log message	409

8.84 Installing the Riemann Logstash output	410
8.85 The Logstash metrics filter	411
8.86 Logstash metrics event	412
8.87 The Riemann output	413
8.88 Logstash metric event in Rieman	414
8.89 The logstash.clj configuration file	415
8.90 Using the logstash var	416
8.91 Adding a Riemann receiver	417
8.92 Restarting Logstash for our Riemann events	417
8.93 A Riemann event in Logstash	418
8.94 A routing configuration for Logstash	421
8.95 Creating the rsyslogd.conf file	424
8.96 The rsyslogd.conf file	424
8.97 Updating JMX for the LS_JAVA_OPTS variable	425
8.98 The JMX enabling options	425
8.99 Restarting Logstash to enable JMX	426
8.100 The collectd JMX configuration file for Logstash	426
8.101 The logstash_jmx.conf file	427
8.102 An MBean	429
8.103 GenericJMX Java memory heap metrics	430
8.104 The Connections block	431
8.105 Creating the logstash.conf file	432
8.106 The logstash.conf file	432
8.107 Restarting collectd to enable JMX	433
8.108 Download the elasticsearch_collectd.py plugin	434
8.109 Creating an elasticsearch.conf configuration file	435
8.110 The elasticsearch.conf configuration file	435
8.111 Editing the elasticsearch.conf file	436
8.112 Updating the elasticsearch.conf file	436
8.113 Restarting collectd to enable Elasticsearch monitoring	437
9.1 An example metric name schema	444

9.2 A sample payments method	448
9.3 The StatsD protocol	450
9.4 A typical StatsD metric	450
9.5 A StatsD counter	451
9.6 A StatsD timer	452
9.7 StatsD timer calculations	452
9.8 StatsD timer calculations	453
9.9 StatsD timer percentile calculations	453
9.10 A StatsD gauge	453
9.11 Changing StatsD gauge values	454
9.12 A StatsD set example	454
9.13 The statsd.conf configuration file	457
9.14 Revisiting the write_riemann plugin	459
9.15 Revisiting the write_riemann plugin	460
9.16 Restarting collectd to enable statsd	460
9.17 Testing the statsd collectd client	461
9.18 Our first statsd event in Riemann	461
9.19 The aom-rails Gemfile	462
9.20 Install statsd-ruby with the bundle command	463
9.21 Testing statsd-ruby with the Rails console	463
9.22 Our first Ruby-based StatsD client	463
9.23 The basic statsd-ruby methods	464
9.24 The statsd-ruby configuration initializer	465
9.25 The statsd-ruby configuration initializer	465
9.26 The statsd-ruby namespace method	466
9.27 Counter for aom-rails user deletions	466
9.28 The aom-rails.production.user.deleted event in Riemann	467
9.29 Counter for aom-rails user creation	468
9.30 The aom-rails.production.user.created event in Riemann	468
9.31 Time the User.find action	469
9.32 The user.find timer in Riemann	469

9.33 The find.user 99th percentile	470
9.34 Detecting missing aom-rails events	471
9.35 Checking the user.find timer	471
9.36 Rewriting our statsd rules	472
9.37 Updated graphite.clj	474
9.38 The graphite-path-statsd function	475
9.39 A typical syslog message	477
9.40 Unstructured log message example	478
9.41 Structured log message example	479
9.42 JSON encoded event	479
9.43 Adding our logging gems to the aom-rails Gemfile	481
9.44 Install the gems with the bundle command	481
9.45 Adding logging to the Rails production environment	482
9.46 Adding a new TCP input to Logstash	483
9.47 Restarting Logstash for our Application events	483
9.48 Traditional Rails request logging	484
9.49 A Lograge request log event	484
9.50 Logging deleted users	485
9.51 A Logstash formatted event for a user deletion	486
9.52 Custom application logs	489
9.53 Adding a TCP input for our applications	490
9.54 An unprocessed custom Logstash formatted event	490
9.55 Adding a grok filter for our applications	491
9.56 Creating the new patterns directory	491
9.57 The /etc/logstash/patterns/app file	492
9.58 Using the APP_TIMESTAMP pattern	492
9.59 A processed custom Logstash formatted event	493
9.60 Adding an API route to config/routes.rb	497
9.61 The api_controller.rb controller	498
9.62 The /api endpoint	498
9.63 The curl_json.conf file	499

9.64 Restarting collectd for curl_json	500
9.65 A curl_json event from our sample application	500
9.66 An example code deployment metric	501
9.67 Adding the Riemann client to the aom-rails Gemfile	502
9.68 Install the gem with the bundle command	503
9.69 The deploy Rake task	504
9.70 A sample deployment event	505
9.71 Graphing our deployment events	506
9.72 Reloading Riemann to graph deployment events	506
10.1 Sample Nagios notification redux	514
10.2 Our Riemann expiration configuration	515
10.3 Our Riemann expiration configuration	516
10.4 The original email.clj	516
10.5 The :format and :body options	518
10.6 The new mailer options	518
10.7 The format-subject function	519
10.8 The new subject line	519
10.9 The new subject line applied	519
10.10 The format-body function	521
10.11 The header variables	522
10.12 The format-body function	523
10.13 The context function	524
10.14 The lookup function	525
10.15 The round function	526
10.16 The byte-to-gb function	526
10.17 The footer variable	526
10.18 The new formatted email notification	527
10.19 Creating a new scripted dashboard	528
10.20 Querying the Graphite-API	530
10.21 Adding CORS controls to Graphite-API	530
10.22 Restarting Graphite-API for CORS	531

10.23 Our riemann.js query parameters	532
10.24 Our query construction variables	533
10.25 The <code>find_filter_values</code> query function	534
10.26 The Graphite-API metrics find query	535
10.27 The <code>panel_memory</code> function	536
10.28 The <code>targets</code> attribute	537
10.29 The <code>row_cpu_memory</code> function	538
10.30 Adding a dashboard to our context function	540
10.31 A dashboard in context	540
10.32 Our new graph-enhanced notification	541
10.33 Adding the <code>slack.clj</code> file	544
10.34 The slack notification configuration	545
10.35 The Slack Webhook URL	546
10.36 The <code>slack-format</code> function	546
10.37 The Slack notification	547
10.38 Our default argument	547
10.39 The Slack formatter option	548
10.40 Using the Slack notification	549
10.41 Using the Slack notification default	549
10.42 Adding the <code>pagerduty.clj</code> file	550
10.43 The <code>pagerduty.clj</code> file	551
10.44 The <code>:details</code> key with graph URL	552
10.45 Using our PagerDuty notification	554
10.46 The <code>maintenance-mode?</code> function	556
10.47 The <code>--></code> expression expanded	557
10.48 Checking maintenance prior to notifications	557
10.49 Installing the Riemann Ruby client	558
10.50 Sending a Riemann maintenance event	558
10.51 Disabling the maintenance window	559
10.52 Installing the <code>maintainer</code> gem	559
10.53 Using the <code>maintainer</code> gem to generate an active event	559

10.54 The maintainer active event	560
10.55 Using the maintainer gem to disable maintenance	560
10.56 The maintainer active event	560
10.57 Adding the count-notifications.clj file	562
10.58 Adding the count-notifications function	562
10.59 A memory usage check	563
10.60 A memory usage check notification count	564
10.61 A memory usage check reinjection	565
10.62 Our notification count warning notification	566
11.1 The write_riemann configuration from Chapter 5	575
11.2 Adding a new tag for Tornado	576
11.3 Adding to an existing Tag	576
11.4 The Riemann :tags field for Tornado	577
11.5 The HAProxy configuration	579
11.6 Adding the stats option to our HAProxy configuration	580
11.7 Restarting the HAProxy service	580
11.8 The /var/run/haproxy.sock socket	581
11.9 Download the haproxy.py plugin	581
11.10 Creating a haproxy.conf configuration file	582
11.11 The haproxy.conf configuration file	583
11.12 Restarting the collectd daemon for HAProxy	584
11.13 HAProxy events in Riemann	584
11.14 Updating HAProxy metrics in Riemann	585
11.15 Reloading Riemann for HAProxy rewrites	585
11.16 The HAProxy log configuration	586
11.17 The updated HAProxy log configuration	586
11.18 Restarting HAProxy to change log tags	587
11.19 HAProxy log entry	587
11.20 The HAProxy updated Logstash configuration	589
11.21 Our new parsed HAProxy event	591
11.22 Sending HAProxy events to Riemann	593

11.23 A HAProxy event in Riemann	595
11.24 Graphing our HAProxy events in Riemann	596
11.25 The Nginx status page	597
11.26 Confirming the status module is available	597
11.27 The Nginx status page	598
11.28 Restarting the Nginx service	598
11.29 The Nginx status page	599
11.30 Creating a nginx.conf configuration file	599
11.31 The nginx.conf collectd configuration	600
11.32 Restarting the collectd daemon for the Nginx plugin	600
11.33 An Nginx plugin event in Riemann	601
11.34 Create the nginx log configuration file	603
11.35 The RSyslog Nginx configuration	603
11.36 Restarting the RSyslog service for Nginx	604
11.37 A Nginx access log entry	605
11.38 Updated Nginx Logstash configuration	606
11.39 Make the patterns directory	607
11.40 The nginx pattern	607
11.41 The parsed Nginx access log event	608
11.42 Sending Nginx logs to Riemann	609
11.43 A Nginx event in Riemann	610
11.44 Graphing our HAProxy events in Riemann	611
11.45 Matching multiple Nginx processes	612
11.46 Added collectd monitoring to our Riemann configuration	613
11.47 Creating the examplecom directory and tornado.clj file	614
11.48 The tornado.clj file	615
11.49 Requiring the Riemann functions	616
11.50 The tornado checks function	618
11.51 The splitp predicate and expression	619
11.52 A splitp clause	619
11.53 The webtier function	621

11.54 Our first webtier check	622
11.55 The tornado checks and notifications function	623
11.56 Our second webtier check	624
11.57 The check_ratio function	625
11.58 Presenting our new check_ratio event	627
11.59 Our second webtier check returned	628
11.60 The haproxy.frontend.tornado-www.5xx_error_percentage event .	629
11.61 Added monitoring.services to Riemann configuration	630
11.62 Sending our Tornado events to be checked	630
12.1 Updating Tornado API to run JMX	633
12.2 The JMX enabling options	634
12.3 The collectd JMX configuration file for Tornado API	634
12.4 The tornado-api.conf file	635
12.5 An MBean	636
12.6 GenericJMX Java memory heap metrics	637
12.7 The Connections block	638
12.8 Adding tornado-api process monitoring	639
12.9 The RSyslog tornado-api configuration	640
12.10 Restarting the RSyslog service for Tornado API	641
12.11 A Timbre log entry from our Tornado API	641
12.12 Our initially parsed Tornado API event	642
12.13 Updated Tornado API Logstash configuration	643
12.14 The tornado-api pattern	644
12.15 Our updated Tornado API event	645
12.16 Sending Tornado API events to Riemann	646
12.17 The Tornado API request in Riemann	647
12.18 Querying our dummy item	648
12.19 Adding to the tornado-api.conf configuration file	649
12.20 Restarting collectd for curl_json	650
12.21 A curl_json event from the Tornado API	650
12.22 Our instrumented Tornado API setup	651

12.23 Our instrumented Tornado API buy-item method	652
12.24 Hitting the buy-item API endpoint	652
12.25 Our instrumented Tornado API update and sell item methods . . .	653
12.26 The statsd.conf configuration file	654
12.27 Restarting collectd to enable statsd	654
12.28 A Tornado API StatsD metric in Riemann	655
12.29 Rewriting the Tornado API events	655
12.30 The tornado.checks function	657
12.31 The apptier function	658
12.32 DRY'ed child streams	660
12.33 Creating a rate	660
12.34 The tornado.api.request rate	661
13.1 Installing the python-mysqldb package	664
13.2 Installing the mysql-python package	665
13.3 Download the mysql.py plugin	665
13.4 Creating a MySQL collect user	666
13.5 The mysql.conf collectd configuration file	667
13.6 A typical MySQL event in Riemann	668
13.7 Rewriting the MySQL events	669
13.8 Installing the libdbd-mysql package	670
13.9 Installing the libdbi package	670
13.10 The dbi plugin configuration	671
13.11 Our price query	673
13.12 The Query directives	674
13.13 The items database events in Riemann	674
13.14 Rewriting the dbi events	675
13.15 Checking the PERFORMANCE_SCHEMA	677
13.16 Enabling the PERFORMANCE_SCHEMA	677
13.17 New query	679
13.18 The Tornado API INSERT statement	680
13.19 The INSERT event in Riemann	680

13.20 The redis.conf collectd configuration	682
13.21 A Riemann Redis event	683
13.22 Rewriting the Redis events	683
13.23 The tornado checks function	685
13.24 The datatier function	686
13.25 The create_rate function	687
13.26 The INSERT query check	688
13.27 The current value of items bought	690
13.28 The Tornado 5xx error percentage	692
13.29 The MySQL database insertion query timing	692
13.30 Tornado API Requests Time 0.99	693
13.31 Tornado API Request Rate	693
13.32 Tornado DB Tier CPU Usage	694
13.33 Tornado App Tier CPU Usage	694
13.34 Tornado Web Tier CPU Usage	694
13.35 Tornado Swap Used	694
13.36 Tornado Memory Used	695
13.37 Tornado Load (shortterm)	695
13.38 Tornado Disk used on the / partition	695
13.39 Expanding Tornado	696
13.40 Renaming the Tornado namespace	697
13.41 Sending more than our Tornado events to be checked	697
A.1 A declarative statement	700
A.2 Getting lein	702
A.3 Auto-installing lein	702
A.4 Launching REPL	703
A.5 The REPL shell	703
A.6 Our first Clojure value	703
A.7 Our first Clojure integer	704
A.8 Our first Clojure string	704
A.9 Our first Clojure Booleans	704

A.10 The Clojure function syntax	705
A.11 Our first Clojure function	705
A.12 The fully qualified + function	706
A.13 Unable to resolve symbol	707
A.14 Quoting a symbol	707
A.15 The type function	707
A.16 A Clojure list	708
A.17 An unquoted Clojure list	708
A.18 Adding an element to a list	709
A.19 Working with lists	709
A.20 Creating a list	710
A.21 A Clojure vector	710
A.22 Adding an element to a vector	710
A.23 Getting the last element in a vector	711
A.24 Counting elements in a vector	711
A.25 Using a vector as a function	711
A.26 Creating or converting vectors	712
A.27 A Clojure set	712
A.28 Adding to a set	713
A.29 Removing an element from a set	713
A.30 Checking for a value inside a set	713
A.31 Using the set as a function	714
A.32 Making a set	714
A.33 A Clojure map	714
A.34 Getting a Clojure map value	715
A.35 Getting a missing Clojure map value	715
A.36 Getting a default value from a map	715
A.37 Using a map as a function	716
A.38 Using a keyword as a function	716
A.39 Using assoc to add a key/value	716
A.40 Replacing a key/value with assoc	717

A.41 Removing a key/value with dissoc	717
A.42 The str function	718
A.43 Concatenating a string	718
A.44 The inc function again	718
A.45 The fn function	719
A.46 Running our first fn function	719
A.47 The fn function shortcut	719
A.48 Calling the fn function shortcut	720
A.49 Creating a var	720
A.50 Evaluating a symbol	721
A.51 Using the type function on the symbol	721
A.52 Creating our first named function	722
A.53 Calling our grow function	722
A.54 Using the defn form	722
A.55 Adding a second argument	723
A.56 Calling our grow function again	723
A.57 Calling grow with our second argument	723
A.58 Adding a second argument	724
A.59 Using the doc function	724

Index

- Annotations, 508
- Ansible, 139, 202, 217, 247, 271, 501
- Apache
 - mod_status, 601
- Apache Lucene, 344, 383, 384
- Apophenia, 36
- AppDynamics, 450
- Application architecture, 442
- Application deployment, 501
- Application logs, 491
- Application metrics, 440
- Application monitoring, 440, 442
- assoc, 281, 321
- Availability, 41
- Average, 31
- Averages, 43, 46
- Bell Curve, 43
- Blackbox, 23
- Borgmon, 24
- Business metrics, 440
- Campfire, 546
- Capistrano, 501
- Carbon, 124
- carbon-cache, 124
- carbon-relay, 124
- configuration, 140
- consistent hashing, 147
- metrics distribution, 147
- protocols, 143
- Redundancy, 191
- relay rules, 147
- RELAY_METHOD, 191
- REPLICATION_FACTOR, 191
- carbon-cache, 124
- carbon-relay, 124
- Carbonate, 196
- Cassandra, 206
- Chef, 63, 139, 202, 217, 247, 306, 350, 359, 501
- Circonus, 300
- Clock skew, 196, 367, 444
- Clojure, 19, 57, 67, 102, 699
 - *, 722
 - +, 705
 - apply, 519
 - assoc, 281, 321, 332, 552, 623, 716

cond->, 321
condp, 281, 475, 623
conj, 708, 710, 712
contains?, 713
count, 711
declare, 517
def, 105, 258, 546, 720
default arguments, 547
defn, 259, 722
deftest, 113
destructuring, 329
disj, 713
dissoc, 717
doc, 724
double, 523
first, 709
float, 627
fn, 282, 718
format, 519
function ordering, 517
functions, 548
get, 715
hash, 714
homoiconicity, 706
if-let, 330
info, 616
join, 522
last, 711
Leiningen, 701
let, 332, 622, 627
list, 705, 708
macro, 556
map, 519, 522, 714
merge, 331
namespaces, 104, 721
ns, 104
nth, 709
quoting, 707
ratio, 523
re-matches, 619
replace, 259, 267
require, 616, 630
second, 709
set, 712, 714
Shell, 271
StatsD, 651
str, 321, 547, 717
string, 518
strings, 520
style, 725
symbols, 706
type, 721
types, 703
var, 105, 720
vec, 711
vector, 710
walk, 328
CMDB, 542
collectd, 210, 211, 423, 497
-h, 215, 217
Apache, 601
cgroups, 340

CheckThresholds, 221
configuration, 223
configuration management, 247
cpu, 224, 225, 271
 ReportByCpu, 225
 ValuesPercentage, 225
curl, 497
curl_json, 647
curl_json, 497
curl_xml, 497
data sources, 252
dbi, 669
df, 224, 227, 272
 Dev, 228
 IgnoreSelected, 229
 MountPoint, 228
 ValuesPercentage, 228, 229
Elasticsearch, 434
Exec, 310
GenericJMX, 426, 634
help, 215
Include, 223–225
installation, 214
interface, 224, 231
 IgnoreSelected, 231
 Interface, 231
Interval, 220
Java, 311, 634
load, 224, 232
LoadPlugin, 235
logfile, 222
Logstash logging, 222
memory, 224, 226, 272
 ValuesPercentage, 226
MySQL, 665
Nginx, 599
nginx, 599
notifications, 613
Per-plugin intervals, 235
Perl, 311
ping, 496
plugins, 211
processes, 224, 233, 338, 432, 584,
 613
Process, 235
ProcessMatch, 235
protocols, 224, 232
Python, 310, 582
Redis, 681
Required version, 210
statsd, 455, 457, 653
 DeleteCounters, 458
 DeleteGauges, 458
 DeleteSets, 458
 DeleteTimers, 458
 flush interval, 458
 TimerCount, 458
 TimerLower, 458
 TimerPercentile, 458
 TimerSum, 458
 TimerUpper, 458
swap, 224, 229

ValuesPercentage, 229, 230
tags, 575
template configuration, 219
threshold, 222, 223
thresholds, 238
types.db, 314
write_riemann, 455
write_riemann, 244
 CheckThresholds, 245
 Multiple nodes, 246
 Node, 244
 Tag, 247
 TTLFactor, 245
Writing plugins, 434, 582, 665
CollectM, 299
condp, 281
Configuration Management, 460
Configuration management, 59, 60, 62, 63, 69, 129, 133, 139, 175, 202, 214, 217, 219, 223, 224, 246, 247, 271, 306, 311, 350, 356, 359, 419, 422, 501, 530, 542, 575, 622, 649, 700
Consul, 696
Container
 logging, 400
 monitoring, 302
Containers, 302
Count, 31
Counters, 30
redis, 681
Curator, 384
Cyanite, 206
D3, 205, 300
DataDog, 300
deftest, 113
Deployment, 501
DevOps, 13
Docker, 19, 63, 139, 202, 217, 302, 303, 350, 359
 /etc/default/docker, 401
 /etc/sysconfig/docker, 401
 API, 308
 cgroups, 340
 docker command, 308
 DOCKER_OPTS, 401
 labels, 325
 logging, 400
 tags, 402
 metadata, 325
Downtime scheduling, 560
Elasticsearch, 19, 343, 344, 355, 371, 449, 485
 –version, 360
 clustering, 362, 363
 Curator, 384
 DEB, 356
 discovery, 364
 document, 383
 index, 383
 introduction, 344

nodes, 385
packages, 356
Plugins, 365
replica, 385
RPM, 356
scaling, 423
shard, 384
 primary, 384
 replica, 384
status, 360
template, 383
unicast, 363
ELK, 19, 343
Endpoints, 442, 494
EPEL, 130
Error rates, 270
Events, 74
External monitoring, 494
Extra Packages for Enterprise Linux,
 130
Fabric, 271, 542
Filebeat, 398
Flapping, 266
Frequency distribution, 32
Functional programming, 699
Ganglia, 120, 300
Gauges, 29
GenericJMX, 426, 634
Git, 116
Grafana, 19, 57, 125, 287, 508
 allowed_origins, 530
 Annotations, 508
 dashboard scripting, 528
 Installation, 136
 Introduction, 183
 Playlists, 190
 scripted dashboard examples, 542
 Scripted dashboards, 190
 Sharing, 190
 Templating, 190
Granularity, 28
Graph, 29
Graphite, 19, 57, 123
 alias, 691
 aliasByNode, 295, 537
 API, 125, 542
 architecture, 126
 Carbon, 124, 140
 carbon.conf, 140
 changing resolution, 149
 Functions, 298
 Graphite Web, 125
 installation, 129
 line protocol, 143
 logging, 182
 metric format, 181
 metric path, 181
 Performance, 125
 pickle protocol, 143, 144
 plaintext, 143
 plaintext protocol, 143

protocols, 143
Redundancy, 191
Render graphs, 542
resolution, 148
retention, 149
Storage schemas, 149, 181, 220
sumSeriesWithWildcards, 691
Whisper, 124, 149, 220
Graphite Web, 125
Graphite-API, 125, 133
 Configuration, 162
 Documentation, 163
 finders, 164
 functions, 164
 Installation, 133
Graylog, 438
grok, 590

HAProxy, 195, 422, 577, 578
 HTTP log format, 587, 594
 logging, 578
 Logs, 587
 statistics, 578
Health checks, 442, 494
Heka, 438
Hindsight, 438
Hipchat, 546
Histogram, 32
Host monitoring, 208

index, 75
InfluxDB, 206
Instrumentation, 441
Java, 57, 347, 356, 357
 JVM, 345
Java Management Extensions, 424
JavaBean, 636
Jiffies, 225
JMX, 424, 633
Jordan Sissel, 343
JRuby, 345
JSON, 344, 354, 416, 483, 497
jstat, 425
JVM, 57
 Managed Bean, 636
 MBean, 636

Kamon, 425
keepalived, 422
Kibana, 19, 343, 344, 386, 485
 kibana.yml, 388

Latency, 53
Leiningen, 116, 701
libdbi, 670
Librato, 300
Log4j, 639
logger, 371
Logging, 342
Lograge, 480, 482
Logs, 26
Logstash, 19, 222, 343, 449, 578, 587,
 607

-f, 353
-v, 353
@timestamp, 355
@version, 355
.grokparsefailure, 405
architecture, 346
beats, 398
brokers, 422
codec, 354, 416
 json, 417
conditionals, 375
configtest, 353
configuration, 351
Custom grok patterns, 492
Custom logs, 491
date, 382
elasticsearch, 371
events, 372
field reference, 414
file, 371
filter
 date, 590
 grok, 491, 492, 588, 590, 607,
 642
 metrics, 412
 syslog_pri, 590
filtering, 375
filters, 352
Graphite, 414
grok, 376, 405, 449
 helpers, 380
patterns_dir, 492
grok patterns, 491, 492, 590
input
 tcp, 417, 482, 483, 489, 490
inputs, 352
installation, 348
introduction, 343
JSON, 416
JSON codec, 354
Load balancing, 422
message, 355
metrics, 449
Nginx, 602
output
 riemann, 410, 592, 610
 tcp, 422
outputs, 352
plain codec, 354
plugin, 410
plugins, 353
RELP, 395
Riemann, 413
riemann, 592, 610
Riemann output, 410
scaling, 419
sinceedb, 370
stdin plugin, 353
stdout plugin, 353
Syslog, 391
syslog, 393
syslog_pri, 380

udp, 393
 Logstash-logger, 480
 Logstash
 tcp, 393

 Maintainer, 559
 Maintenance, 560
 Managed Beans, 428, 636
 MBean, 633
 MBeans, 428, 636
 MCollective, 271, 542
 Mean, 43
 Median, 31, 47, 52
 Metric resolution, 148
 Metrics, 26, 447
 latency, 53
 Microsoft Windows, 214, 299
 Munin, 120, 300
 MySQL
 performance_schema, 676

 nc, 461
 netcat, 461
 Network monitoring, 214, 299
 New Relic, 299, 450
 Nginx, 596, 597, 602
 error_log, 602
 log_format, 601
 stub_status, 597
 Notifications, 33, 513
 Email, 516
 PagerDuty, 549

 Slack, 543
 NTP, 196, 367, 444
 installation, 199
 ntpq, 203

 Observations, 28
 Observer Effect, 54
 OpsGenie, 549
 OpsWeekly, 567

 PagerDuty, 111, 551
 Percentiles, 31, 50, 52, 276
 Plot, 29
 plugins
 file, 398
 Postal, 106
 Prometheus, 24
 Pull-based architecture, 23
 Puppet, 63, 139, 202, 217, 247, 306,
 350, 359, 501, 542
 PuppetDB, 542, 619, 696
 Push-based architecture, 23

 Rails, 462
 logger, 482
 logging, 480
 Lograge, 480
 metrics, 462
 StatsD, 462
 Rake, 501
 Rates of change, 31
 Redis, 317

Reimann
 REPL, 701
REPL, 701
Resolution, 28
Rieman
 email, 268
Riemann, 19, 56, 58, 410, 415
 /var/log/riemann/riemann.log, 79
 adjust, 267, 623
 AGGRESSIVE_OPTS, 118
 async-queue, 92, 177
 asynchronous streams, 92
 by, 265, 275, 283, 339, 553, 563, 565
 C client, 83
 call-rescue, 282, 284, 627
 changed, 266
 changed-state, 266, 552, 623
 client, 92
 Clients, 502
 Clojure DSL, 67
 configuration, 64, 68
 connecting servers, 91
 dashboard, 66
 default, 78, 100
 email, 102, 517
 event, 74
 events, 519
 expired, 268, 613
 expired stream, 108
 EXTRA_JAVA_OPTS, 117
failover, 118, 246
filtering streams, 87
fixed-time-window, 275
folds, 275
folds/count, 276
folds/maximum, 276
folds/mean, 276
folds/median, 275
forwarding events, 92
high availability, 118
HOWTO, 287
HUP, 73
index, 75, 78, 100, 525, 557
installation, 59
JMX, 120
Leiningen, 701
logging, 65, 69, 80, 81
logstash plugin, 415
lookup, 525
mailer, 102, 517
Measuring exceptions, 270
namespacing, 104
network, 69
ns, 104
PagerDuty, 111, 551
percentiles, 284
Performance, 117
ports, 69
Postal, 106
prn, 80
project, 626

quotient, 626
quotient-sloppy, 626
rate, 563, 661, 687
reinject, 565
REPL, 72
require, 93
rollup, 110
Ruby client, 502
scaling, 118, 246
sdo, 339, 622, 629
search, 525
Service discovery, 285
SIGHUP, 73
Slack, 111, 545
smap, 260, 278, 320, 332, 339, 340, 626
splitip, 619
SSL, 71
stable, 266
state, 238
Streams, 75
streams, 78
syntax checking, 116
tag, 460, 563
tagged, 87, 252, 253, 259, 263, 264, 286, 565
tagged-all, 87
tagged-any, 87, 697
tags, 247, 460, 575
tap, 113
TCP, 245
testing, 113
throttle, 109
TLS, 71
tools, 65
TTL, 76, 245
UDP, 245
where, 87, 96, 108, 270, 272, 332, 471, 553, 565, 626, 630, 687
with, 565, 660, 687
Zookeeper, 285
RSyslog, 395, 587
imfile, 398, 602
Selmer, 527
Semantic logging, 441, 478
sendmail, 106
Server monitoring, 208
Service discovery, 285
Slack, 111, 543
SNMP, 299
Splunk, 438
SSC Serv, 299
Standard Deviation, 31, 50
State detection, 266
StatsD, 300, 447, 449, 450, 651
 Counters, 451
 Gauges, 453
 Metric types, 451
 namespaces, 466
 Protocol, 450
 Ruby-client, 462

Sampling, 452
Sets, 454
Timers, 451
statsd-ruby, 462
Streams, 75
Structured logging, 441, 478, 480
Sum, 31
Syslog, 391, 394, 477, 590
 facility, 395
 severity, 395
Thresholds, 39, 42
Throttle, 109
timbre, 639
Time, 196, 367, 444
 ntpq, 203
Time series, 27
Time zone, 196, 367, 444
 Red Hat, 199
 Ubuntu, 197
time-at, 523
Timers, 30
Tracing, 512
Typed logging, 478

Vagrant, 63, 139, 202, 217, 350, 359
VictorOps, 549
Visualization, 36

Websockets, 69
Whisper, 124
 Alternatives, 205

Thanks! I hope you enjoyed the book.

© Copyright 2016 - James Turnbull <james@lovedthanlost.net>

