The logstash Book

Log management made easy

James Turnbull

The Logstash Book

James Turnbull

November 5, 2016

Version: v5.0.0a (1216eaa)

Website: The Logstash Book

ISBN 978-0-9888202-1-0

Contents

Foreword

Who is this book for?

This book is designed for SysAdmins, operations staff, developers and DevOps who are interested in deploying a log management solution using the open source tool Logstash.

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.

NOTE This book focuses on Logstash version 5.0.0 and later. It might work for earlier versions of Logstash but is not recommended.

Credits and Acknowledgments

- Jordan Sissel for writing Logstash and for all his assistance during the writing process.
- Rashid Khan for writing Kibana.
- Dean Wilson for his feedback on the book.

- Aaron Mildenstein for his Apache to JSON logging posts here and here.
- R.I. Pienaar for his excellent documentation on message queuing.
- The fine folks in the Freenode #logstash channel for being so helpful as I peppered them with questions, and
- Ruth Brown for only saying ”Another book? WTF?” once, proof reading the book, making the cover page and for being awesome.

Technical Reviewers

Jan-Piet Mens

Jan-Piet Mens is an independent Unix/Linux consultant and sysadmin who's worked with Unix-systems since 1985. JP does odd bits of coding, and has architected infrastructure at major customers throughout Europe. One of his specialities is the Domain Name System and as such, he authored the book *Alternative DNS Servers* as well as a variety of other technical publications.

Paul Stack

Paul Stack is a London based developer. He has a passion for continuous integration and continuous delivery and why they should be part of what developers do on a day to day basis. He believes that reliably delivering software is just as important as its development. He talks at conferences all over the world on this subject. Paul's passion for continuous delivery has led him to start working closer with operations staff and has led him to technologies like Logstash, Puppet and Chef.

Technical Illustrator

Royce Gilbert has over 30 years experience in CAD design, computer support, network technologies, project management, business systems analysis for major Fortune 500 companies such as; Enron, Compaq, Koch Industries and Amoco Corp. He is currently employed as a Systems/Business Analyst at Kansas State University in Manhattan, KS. In his spare time he does Freelance Art and Technical Illustration as sole proprietor of Royce Art. He and his wife of 38 years are living in and restoring a 127 year old stone house nestled in the Flinthills of Kansas.

Author

James is an author and open-source geek. His most recent books are The Terraform Book about infrastructure management tool Terraform, The Art of Monitoring about monitoring, The Docker Book about Docker, 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.

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

Conventions in the book

This is an `inline code statement`.

This is a code block:

Listing 1: A sample code block

```
This is a code block
```

Long code strings are broken with ↩.

Code and Examples

You can find all the code and examples from the book on the website or you can check out the Git repo.

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 here.

Trademarks

Kibana and Logstash are trademarks of Elasticsearch BV. Elasticsearch is a registered trademark of Elasticsearch BV.

Version

This is version v5.0.0a (1216eaa) of The Logstash Book.

Copyright

ISBN 978-0-9888202-1-0

Version: v5.0.0a (1216eaa)

Chapter 1

Introduction or Why Should I Bother?

Log management is often considered both a painful exercise and a dark art. Indeed, understanding good log management tends to be a slow and evolutionary process. In response to issues and problems, new SysAdmins are told: "Go look at the logs." A combination of `cat`, `tail` and `grep` (and often `sed`, `awk` or `perl` too) become their tools of choice to diagnose and identify problems in log and event data. They quickly become experts at command line and regular expression kung-fu: searching, parsing, stripping, manipulating and extracting data from a humble log event. It's a powerful and practical set of skills that strongly I recommend all SysAdmins learn.

Sadly, this solution does not scale. In most cases you have more than one host and multiple sources of log files. You may have tens, hundreds or even thousands of hosts. You run numerous, inter-connected applications and services across multiple locations and fabrics, both physically, virtually and in the cloud. In this world it quickly becomes apparent that logs from any one application, service or host are not enough to diagnose complex multi-tier issues.

To address this gap your log environment must evolve to become centralized. The

tools of choice expand to include configuring applications to centrally log and services like `rsyslog` and `syslog-ng` to centrally deliver Syslog output. Events start flowing in and log servers to hold this data are built, consuming larger and larger amounts of storage.

But we're not done yet. The problem then turns from one of too little information to one of too much information and too little context. You have millions or billions of lines of logs to sift through. Those logs are produced in different timezones, formats and sometimes even in different languages. It becomes increasingly hard to sort through the growing streams of log data to find the data you need and harder again to correlate that data with other relevant events. Your growing collection of log events then becomes more of a burden than a benefit.

To solve this new issue you have to extend and expand your log management solution to include better parsing of logs, more elegant storage of logs (as flat files just don't cut it) and the addition of searching and indexing technology. What started as a simple `grep` through log files has become a major project in its own right. A project that has seen multiple investment iterations in several solutions (or multiple solutions and their integration) with a commensurate cost in effort and expense.

There is a better way.

Introducing Logstash and the Elastic Stack

Instead of walking this path, with the high cost of investment and the potential of evolutionary dead ends, you can start with Logstash. Logstash provides an integrated framework for log collection, centralization, parsing, storage and search.

Logstash is free and open source (Apache 2.0 licensed) and originally developed by American developer, Jordan Sissel and now from the team from Elastic. It's easy to set up, performant, scalable and easy to extend.

This book focusses on Logstash but it is part of a broader stack, that's undergone

a few name changes, available from Elastic. The stack was originally nicknamed ELK, from the combined first letters of three components: Elasticsearch, Logstash and Kibana. Elasticsearch being a powerful, documentation-based search engine (which we'll see more of later in the book) that back-ends Logstash and Kibana, being the web interface to the Logstash data stored in Elasticsearch (although able to query a variety of otherdata too!). The ELK stack has recently been renamed to the Elastic Stack.

TIP Elastic also ships additional products as part of the Elastic Stack including security, analytics and even a hosted stack Elastic Stack product you can use.

Logstash design and architecture

Logstash has a threefold design:

1. Log input.

Logstash has a wide variety of input mechanisms: it can take inputs from TCP/UDP, files, Syslog, Microsoft Windows EventLogs, STDIN and a variety of other sources. Elastic also ships an open source collection of input tools called Beats, that can help you gather log data from disparate sources. As a result there's likely very little in your environment that you can't extract logs from and send them to Logstash.

2. Log filtering.

When those logs hit the Logstash server, there is a large collection of filters that allow you to modify, manipulate and transform those events. You can extract the

information you need from log events to give them context. Logstash makes it simple to query those events. It makes it easier to draw conclusions and make good decisions using your log data.

3. Log output.

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. You can integrate Logstash with metrics engines, alerting tools, graphing suites, storage destinations or easily build your own integration to destinations in your environment. Ultimately though, the vast majority of your data will end ip in Elasticsearch, where you can store, query and manage it.

NOTE We'll look at how to develop practical examples of each of these input, filter and output plugins in Chapter 9.

Logstash is written in JRuby and runs in a Java Virtual Machine (JVM). Its architecture is message-based and very simple. Rather than separate agents or servers, Logstash has a single agent that is configured to perform different functions in combination with other open source components.

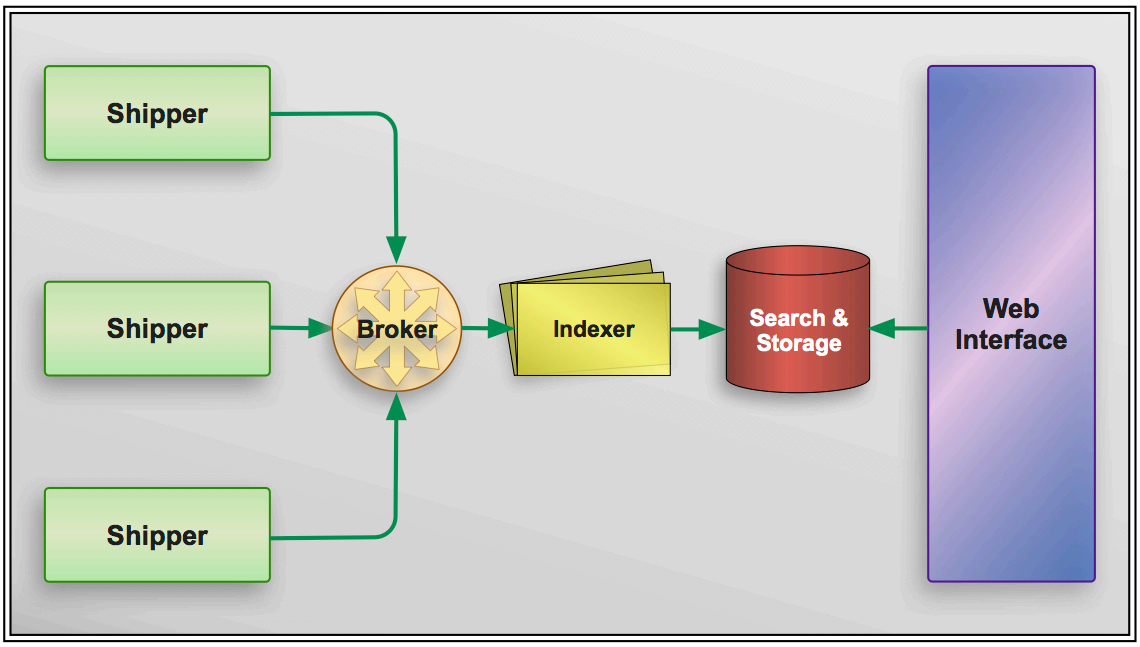

Figure 1.1: The Logstash Architecture

In the Logstash ecosystem there are four components:

- Shipper: Sends events to Logstash. Your remote agents will generally only run this component or use other tools to send events to a central Logstash server.
- Indexer: Receives and indexes the events.
- Search and Storage: Allows you to search and store events.
- Web Interface: A Web-based interface to Logstash data in Elasticsearch called Kibana.

Logstash servers run one or more of these components independently, which allows us to separate components and scale Logstash.

In most cases there will be three broad classes of host you will probably be running:

- Hosts running the Logstash agent as an event "shipper" that send your application, service and host logs to a central Logstash server. These hosts will only need the Logstash agent.
- Hosts running other agents or tools, for example using Beats or a Syslog server, to send events to a central Logstash server. We'll see more about Beats in Chapters 3 and 4.
- Central Logstash hosts running some combination of the Indexer, Search and Storage and Kibana Web Interface which receive, process and store your logs.

NOTE We'll look at scaling Logstash by running the Indexer, Search and Storage and Kibana Web Interface in a scalable architecture in Chapter 8 of this book.

What's in the book?

In this book I will walk you through installing, deploying, managing and extending Logstash. We're going to do that by introducing you to Example.com, where you're going to start a new job as one of its SysAdmins. The first project you'll be in charge of is developing its new log management solution.

We'll teach you how to:

- Install and deploy Logstash.
- Ship events from Syslog, Beats or a Logstash Shipper to a central Logstash server.
- Filter incoming events using a variety of techniques.
- Adding structured logging to applications and services.
- Output those events to a selection of useful destinations.
- Use the Kibana web interface.

- Scale out your Logstash implementation as your environment grows.
- Quickly and easily extend Logstash to deliver additional functionality you might need.

By the end of the book you should have a functional and effective log management solution that you can deploy into your own environment.

NOTE This book focuses on Logstash v5.0.0 and later. This was a major, somewhat backwards-incompatible release for Logstash. A number of options and schema changes were made between v5.0.0 and earlier versions. If you are running an earlier version of Logstash I strongly recommend you upgrade.

Logstash resources

- The Logstash site (Logstash's home page).
- The Logstash source code on GitHub.
- Logstash's original author Jordan Sissel's home page, Twitter and GitHub account.

Getting help with Logstash

Logstash's original developer, Jordan Sissel, has a maxim that makes getting help pretty easy: "If a newbie has a bad time, it's a bug in Logstash." So if you're having trouble reach out via the mailing list or IRC and ask for help! You'll find the Logstash community both helpful and friendly!

- The Logstash documentation.

- The Logstash Forum.
- The Logstash users mailing list (deprecated).
- The Logstash bug tracker.
- The #logstash IRC channel on Freenode.

A mild warning

Logstash and the Elastic Stack are young products and under regular development. Features are changed, added, updated and deprecated regularly. I recommend you follow development at the GitHub support site, on GitHub and review the change logs for each release to get a good idea of what has changed. Logstash is usually solidly backwards compatible but issues can emerge and being informed can often save you unnecessary troubleshooting effort.

Chapter 2

Getting Started with Logstash

Logstash is easy to set up and deploy. We're going to go through the basic steps of installing and configuring it. Then we'll try it out so we can see it at work. That will provide us with an overview of its basic set up, architecture, and importantly the pluggable model that Logstash uses to input, process and output events.

Installing Java

Logstash's principal prerequisite is Java and Logstash itself runs in a Java Virtual Machine or JVM. Logstash requires Java 8 or later! Let's start by installing Java. The fastest way to do this is via our distribution's packaging system, for example Yum in the Red Hat family or Debian and Ubuntu's Apt-Get.

TIP I recommend you install OpenJDK Java on your distribution. If you're running OS X the natively installed Java may work fine but on Mountain Lion and later you'll may need to install Java from Apple.

On the Red Hat family

We install Java via the `yum` command:

Listing 2.1: Installing Java on Red Hat

```
$ sudo yum install java-1.8.0-openjdk
```

On Debian & Ubuntu

We install Java via the `apt-get` command:

Listing 2.2: Installing Java on Debian and Ubuntu

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

Testing Java is installed

We then test that Java is installed via the `java` binary:

Listing 2.3: Testing Java is installed

```
$ java -version
java version "1.8.0_65"
Java(TM) SE Runtime Environment (build 1.8.0_65-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)
```

If your Java version doesn't start with `1.8.x` (or later) then your distribution doesn't have Java 8 and Logstash will not run! You'll need to install Java from

the official Oracle distribution or directly from OpenJDK.

Getting Logstash

Once we have Java installed we can grab the Logstash package. Although Logstash is written in JRuby, Elastic releases packages and tarballs containing all of the required dependencies. This means we don't need to install JRuby or any other components.

At this stage no distributions ship Logstash packages natively but you can easily download packages from the Elastic site.

TIP If we're distributing a lot of Logstash agents then it's an excellent good idea to use Logstash packages. In the next chapter, when we do a more formal installation of Logstash, we'll be using packages.

For our initial getting started we can download and unpack a tarball:

Listing 2.4: Downloading Logstash

```
$ wget https://artifacts.elastic.co/downloads/logstash/logstash-
5.0.0.tar.gz
$ tar zxvf logstash-5.0.0/.tar.gz
```

NOTE At the time of writing the latest version of Logstash is 5.0.0.

Starting Logstash

Once we have the tarball unpacked we can change into the resulting directory and launch the `logstash` binary and a simple, sample configuration file. We're going to do this to demonstrate Logstash working interactively and do a little bit of testing to see how Logstash works at its most basic.

Our sample configuration file

Firstly, let's create our sample configuration file. We're going to call ours `sample.conf`.

Listing 2.5: Creating sample.conf

```
$ cd logstash-5.0.0
$ vi sample.conf
```

You can see it here:

Listing 2.6: Sample Logstash configuration

```
input {
  stdin { }
}

output {
  stdout {
    codec => rubydebug
  }
}
```

Our `sample.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 last type is `filter` that we're going to see in later chapters. Each type configures a different portion of the Logstash agent:

- inputs - How events get into Logstash.
- filters - How you can manipulate events in Logstash.
- outputs - How you can output events from Logstash.

In the Logstash world events enter via inputs, they are manipulated, mutated or changed in filters and then exit Logstash via outputs.

Inside each component's block you can specify and configure plugins. For example, in the `input` block above we've defined the `stdin` plugin which controls 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.

TIP STDIN and STDOUT are the standard streams of I/O in most applications and importantly in this case in your terminal.

Running the Logstash agent

Now we've got a configuration file let's run Logstash for ourselves:

Listing 2.7: Running the Logstash agent

```
$ cd logstash-5.0.0
$ bin/logstash -f sample.conf
```

TIP You can also specify a directory of configuration files using the `-f` flag, for example `-f /etc/logstash` will load all the files in the `/etc/logstash` directory.

We've used the `logstash` binary from our download directory. We've specified one command line flags: `-f` which specifies the configuration file Logstash should start with.

Logstash should now start to generate some startup messages telling you it is enabling the plugins we've specified and finally emit:

Listing 2.8: Logstash startup message

```
Successfully started Logstash API endpoint {:port=>9600}
```

This indicates Logstash is ready to start processing logs!

TIP You can see a full list of the other command line flags Logstash accepts here.

Testing the Logstash agent

Now Logstash is running, remember that we enabled the `stdin` plugin? Logstash is now waiting for us to input something on `STDIN`. So I am going to type "testing" and hit Enter to see what happens.

Listing 2.9: Running Logstash interactively

```
$ bin/logstash agent -f sample.conf
testing
{
   "@timestamp" => 2016-11-03T12:28:10.428Z,
   "@version" => "1",
   "host" => "mocker.example.com"
   "message" => "testing",
}
```

You can see that our input has resulted in some output: an event in JSON format (remember we specified the `rubydebug` option for the `stdout` plugin). Let's examine the event in more detail.

Listing 2.10: A Logstash JSON event

```
{
   "@timestamp" => 2016-11-03T12:28:10.428Z,
   "@version" => "1",
   "host" => "mocker.example.com"
   "message" => "testing",
}
```

Our event is made up of a timestamp, the host that generated the event `mocker.example.com` and the message, in our case `testing`. You might notice that all these components are also contained in the log output in the `@data` hash.

We see our event has been printed as a hash. Indeed it's represented internally in Logstash as a JSON hash.

If we'd had omitted the `rubydebug` option from the `stdout` plugin.

Listing 2.11: Omitting rubydebug

```
input {
  stdin { }
}

output {
  stdout { }
}
```

We'd have gotten a plain event like so:

Listing 2.12: A Logstash plain event

```
2016-11-03T12:33:09.695Z mocker.example.com testing
```

Logstash calls these formats `codecs`. There are a variety of codecs that Logstash supports. We're going to mostly see the `plain` and `json` codecs in the book.

- plain - Events are recorded as plain text and any parsing is done using `filter` plugins.
- json - Events are assumed to be JSON and Logstash tries to parse the event's contents into fields itself with that assumption.

We're going to focus on the `json` format in the book 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 only the following elements:

- @timestamp: An ISO 8601 timestamp.
- message: The event's message. Here `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 which 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 being collected from. In the next chapters we'll also see some other elements like custom fields, tags and other context that we can add to events.

TIP Running interactively we can stop Logstash using the Ctrl-C key combination.

Summary

That concludes our simple introduction to Logstash. In the next chapter we're going to introduce you to your new role at Example.com and see how you can use Logstash to make your log management project a success.

Chapter 3

Shipping Events

It's your first day at Example.com and your new boss swings by your desk to tell you about the first project you're going to tackle: log management. Your job is to consolidate log output to a central location from a variety of sources. You've got a wide variety of log sources you need to consolidate but you've been asked to start with consolidating and managing some Syslog events.

Later in the project we'll look at other log sources and by the end of the project all required events should be consolidated to a central server, indexed, stored, and then be searchable. In some cases you'll also need to configure some events to be sent on to new destinations, for example to alerting and metrics systems.

To do the required work you've made the wise choice to select Logstash as your log management tool and you've built a basic plan to deploy it:

- Build a single central Logstash server with a single node Elasticsearch cluster on it (we'll discuss scaling more in Chapter 8).
- Configure your central server to receive events, index them and make them available to search.
- Install a Filebeat on a remote agent.
- Configure Filebeat to send some selected log events from our remote agent to our central server.

- Install Kibana to act as a web console and front end for our logging infrastructure.

We'll take you through each of these steps in this chapter and then in later chapters we'll expand on this implementation to add new capabilities and scale the solution.

NOTE To run a central Logstash server or an Elasticsearch server you'll generally need hosts with **4Gb of RAM or better**.

Our Event Lifecycle

For our initial Logstash build we're going to have the following lifecycle:

- The Filebeat on our remote hosts collects and sends a log event to our central server.
- The Logstash server on our central host takes the log event from and indexes it.
- The Logstash server sends the indexed event to Elasticsearch.
- Elasticsearch stores and renders the event searchable.
- The Kibana web interface queries the event from Elasticsearch.

Now let's implement this lifecycle!

Installing Logstash on our central server

First we're going to install Logstash on our central server. We're going to build an Ubuntu 16.04 host called `smoker.example.com` with an IP address of `10.0.0.1` as our central server.

Central server

- Hostname: smoker.example.com
- IP Address: 10.0.0.1

As this is our production infrastructure we're going to be a bit more systematic about setting up Logstash than we were in Chapter 2. To do this we're going to use the available Logstash packages.

TIP There are other, more elegant, ways to install Logstash using tools like Puppet or Chef. Setting up either is beyond the scope of this book but there are several Puppet modules for Logstash on the Puppet Forge and a Chef cookbook. I strongly recommend you use this chapter as exposition and introduction on how Logstash is deployed and use some kind of configuration management to deploy in production.

Installing Java

Logstash's principal prerequisite is Java and Logstash itself runs in a Java Virtual Machine or JVM. **Logstash requires Java 8 or later to work!**

So let's start by installing Java. The fastest way to do this is via our distribution's packaging system installer, for example `yum` (or in more recent releases `dnf`) in the Red Hat family or Debian and Ubuntu's `apt-get` command.

TIP I recommend we install OpenJDK Java on your distribution. If you're running OS X the natively installed Java will work fine but on Mountain Lion and later you'll need to install Java from Apple.

On the Red Hat family

We install Java via the `yum` command:

Listing 3.1: Installing Java on Red Hat

```
$ sudo yum install java-1.8.0-openjdk
```

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

On Debian & Ubuntu

We install Java via the `apt-get` command:

Listing 3.2: Installing Java on Debian and Ubuntu

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

Testing Java is installed

We then test that Java is installed via the `java` binary:

Listing 3.3: Testing Java is installed

```
$ java -version
openjdk version "1.8.0_91"
OpenJDK Runtime Environment (build 1.8.0_91-8u91-b14-3ubuntu1
~16.04.1-b14)
OpenJDK 64-Bit Server VM (build 25.91-b14, mixed mode)
```

Any Java versioned prefixed `1.8` indicates Java 8.

Installing Logstash

First let's install Logstash. To do so we need to add the Logstash APT repository to our host. Let's start by adding the appropriate GPG key for validating the packages.

Listing 3.4: Adding the Elasticsearch GPG key

```
$ wget -O - https://artifacts.elastic.co/GPG-KEY-elasticsearch |
 sudo apt-key add -
```

You may also need the `apt-transport-https` package.

Listing 3.5: Installing apt-transport-https

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

Now let's add the APT repository configuration.

Listing 3.6: Adding the Logstash APT repository

```
$ echo "deb https://artifacts.elastic.co/packages/5.x/apt stable
 main" | sudo tee -a /etc/apt/sources.list.d/elastic-5.x.list
```

TIP If we were running on a Red Hat or a derivative we would install the appropriate Yum repository. See the agent install later in this chapter for Red Hat installation steps.

We then run an `apt-get update` to refresh our package list.

Listing 3.7: Updating the package list

```
$ sudo apt-get update
```

And finally we can install Logstash itself.

Listing 3.8: Installing Logstash via apt-get

```
$ sudo apt-get install logstash
```

Now let's install some of the other required components for our new deployment and then come back to configuring Logstash.

TIP The packages install Logstash into the `/usr/share/` directory, `/usr/share/logstash`.

Installing Logstash via configuration management

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

You can find Chef cookbooks for Logstash at the Chef supermarket.

You can find Puppet modules for Logstash on the Puppet Forge.

You can find an Ansible role for Logstash here.

You can find Docker images for Logstash on the Docker Hub.

You can find a Vagrant configuration for Logstash on GitHub.

Elasticsearch for search

Next we're going to install Elasticsearch to provide our search capabilities. Elasticsearch is a powerful 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.

We're going to install a single Elastichsearch node on our central server as a initial step. In Chapter 8 we'll talk about scaling Elasticsearch for both performance and reliability.

Installing Elasticsearch

Elasticsearch's only prerequisite is Java. As we installed a JDK earlier in this chapter we don't need to install anything additional for it. Elasticsearch is cur-

rently not well packaged in distributions but it is easy to download packages. The Elasticsearch team provides tarballs, RPMs and DEB packages. You can find the Elasticsearch download page here.

As we're installing onto Ubuntu we can use the DEB packages provided:

First, if we haven't already, we install the Elastic.co package key.

Listing 3.9: Downloading the Elastic package key

```
$ wget -O - https://artifacts.elastic.co/GPG-KEY-elasticsearch |
 sudo apt-key add -
```

Now, again assuming we haven't already down it above, we add the Elastic repository to our Apt configuration.

Listing 3.10: Adding the Elasticsearch repo

```
$ echo "deb https://artifacts.elastic.co/packages/5.x/apt stable
 main" | sudo tee -a /etc/apt/sources.list.d/elastic-5.x.list
```

Now we install Elasticsearch.

Listing 3.11: Installing Elasticsearch

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

TIP Remember you can also find tarballs and RPMs for Elasticsearch here.

Installing the package should also automatically start the Elasticsearch server but if it does not then you can manage it via the `service` command:

Listing 3.12: Starting Elasticsearch

```
$ sudo service elasticsearch start
```

And enable it to run at boot.

Listing 3.13: Enabling Elasticsearch to run at boot

```
$ sudo systemctl enable elasticsearch
```

You can then check is Elasticsearch is running with the `status` command.

Listing 3.14: Checking Elasticsearch is running

```
$ sudo service elasticsearch status
```

If it is NOT running you'll see failed start output. If your failed output looks like this:

Listing 3.15: Failed memory output

```
●
 elasticsearch.service - Elasticsearch
   Loaded: loaded (/usr/lib/systemd/system/elasticsearch.service;
 disabled; vendor preset: enabled)
   Active: failed (Result: exit-code) since Thu 2016-11-03
13:11:32 UTC; 2s ago
     Docs: http://www.elastic.co

. . .

Nov 03 13:11:32 smoker.example.com systemd[1]: Started
Elasticsearch.
Nov 03 13:11:32 smoker.example.com elasticsearch[1541]: OpenJDK
64-Bit Server VM warning: INFO: os::commit_memory(0
x000000008a660000, 1973026816, 0) failed; error='C
Nov 03 13:11:32 smoker.example.com elasticsearch[1541]: #
Nov 03 13:11:32 smoker.example.com elasticsearch[1541]: # There
is insufficient memory for the Java Runtime Environment to
continue.

. . .
```

Then Elasticsearch doesn't have enough memory to run! You'll need a host with at least 4Gb of RAM to run Elasticsearch.

Installing Elasticsearch via configuration management

You 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 on the Chef Supermarket.

You can find a Puppet module for Elasticsearch here.

You can find an Ansible role for Elasticsearch here.

You can find Docker images for Elasticsearch here.

You can find a Vagrant configuration for Elasticsearch here.

Introduction to Elasticsearch

Now we've installed Elasticsearch we should learn a little about how it works. A decent understanding is going to be useful later as we use and scale Elasticsearch. Elasticsearch is a text indexing search engine. The best metaphor is the index of a book. You flip to the back of the book[1], look up a word and then find the reference to a page. That means, rather than searching text strings directly, it creates an index from incoming text and performs searches on the index rather than the content. As a result it is fast.

NOTE This is a simplified explanation. See the site for more information and exposition.

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 3.16: A Logstash index

```
logstash-2012.12.31
```

Each Logstash event is made up of fields and these fields become a document inside that index. If we were comparing 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

[1] Not the first Puppet book.

relational database you can define a schema too. Elasticsearch calls these schemas ”mappings”.

It’s important to note that you don’t have to specify any mappings for operations, indeed many of the searches you’ll use with Logstash don’t need mappings, but they often make life much easier. You can see an example of an Elasticsearch mapping here. Since Logstash 1.3.2 a default mapping is applied to your Elasticsearch and you generally no longer need to worry about setting your own mapping.

Like a schema, mapping declares what data and data types fields documents contain, any constraints present, unique and primary keys and how to index and search each field. Unlike a schema you can also specify Elasticsearch settings.

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 you can increase or decrease the number of primary shards when the index is created but not AFTER it is created. Once you’ve created the index the number of primary shards cannot be changed.

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 can also have more if required. Unlike primary shards, this can be changed dynamically to scale out or make an index more resilient. Elasticsearch will cleverly 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 Elasticsearch ”nodes”. Each node is automatically part of an Elasticsearch cluster, even if it’s a cluster of one. When new nodes are created they can use unicast or multicast to discover other nodes that share their cluster

name and will try to join that cluster. Elasticsearch 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.

Configuring our Elasticsearch cluster and node

Next we need to configure our Elasticsearch cluster and node name. Elasticsearch is started with a default cluster name and a random, allegedly amusing, node name, for example ”Frank Kafka” or ”Spider-Ham”. A new random node name is selected each time Elasticsearch is restarted. Remember that new Elasticsearch nodes join any cluster with the same cluster name they have defined. So we want to customize our cluster and node names to ensure we have unique names. To do this we need to edit the `/etc/elasticsearch/elasticsearch.yml` file. This is Elasticsearch's YAML-based configuration file. Look for the following entries in the file:

Listing 3.17: Initial cluster and node names

```
# Use a descriptive name for your cluster:
#
#cluster.name: my-application
#
# ------------------------------------ Node ------------------------
---------------
#
# Use a descriptive name for the node:
#
#node.name: node-1
```

We're going to uncomment and change both the cluster and node name. We're going to choose a cluster name of `logstash` and a node name matching our central server's host name.

Listing 3.18: New cluster and node names

```
cluster.name: logstash
node.name: "smoker"
```

We then need to restart Elasticsearch to reconfigure it.

Listing 3.19: Restarting Elasticsearch

```
$ sudo service elasticsearch restart
```

We can now check if Elasticsearch is running and active.

Determining Elasticsearch is running

You can tell if Elasticsearch is running by browsing to port 9200 on your host, for example:

Listing 3.20: Checking Elasticsearch is running

```
$ curl http://localhost:9200
```

This should return some status information that looks like:

Listing 3.21: Elasticsearch status information

```
{
  "name" : "smoker",
  "cluster_name" : "logstash",
  "cluster_uuid" : "RyF1O9JLQqafAb6jzKIZkQ",
  "version" : {
    "number" : "5.0.0",
    "build_hash" : "253032b",
    "build_date" : "2016-10-26T05:11:34.737Z",
    "build_snapshot" : false,
    "lucene_version" : "6.2.0"
  },
  "tagline" : "You Know, for Search"
}
```

You can also browse to a more detailed statistics page:

Listing 3.22: Elasticsearch status page

```
http://localhost:9200/_stats?pretty=true
```

This will return a page that contains a variety of information about the state and status of your Elasticsearch server.

Listing 3.23: The Elasticsearch stats page

```
{
  "_shards" : {
    "total" : 0,
    "successful" : 0,
    "failed" : 0
  },
  "_all" : {
    "primaries" : { },
    "total" : { }
  },
  "indices" : { }
}
```

You can install a wide variety of plugins to Elasticsearch that can help you manage it.

You can list and install a plugin using the `elasticsearch-plugin` command that ships with Elasticsearch, for example:

Listing 3.24: Listing Elasticsearch plugins

```
$ sudo /usr/share/elasticsearch/bin/elasticsearch-plugin list
```

NOTE In releases prior to 2.3 this command was just called `plugin`.

Plugins are generally available in the Elasticsearch server via URLs with a specific URL path, `_plugins`, being reserved for them.

TIP You can find more extensive documentation for Elasticsearch here.

Creating a basic central configuration

Now we've got our environment configured we're going to set up our Logstash configuration file to receive events. We're going to call this file `central.conf` and create it in the `/etc/logstash/conf.d` directory.

TIP Since Logstash 5.0.0 there is also a `/etc/logstash/logstash.yml` file that you can use to configure Logstash's command line options and configuration.

Listing 3.25: Creating the central.conf file

```
$ sudo touch /etc/logstash/conf.d/central.conf
```

Let's put some initial configuration into the file.

Listing 3.26: Initial central configuration

```
input {
  beats {
    port => 5044
  }
}
output {
  stdout { }
  elasticsearch { }
}
```

In our `central.conf` configuration file we can see the `input` and `output` blocks we learned about in Chapter 2. Let's see what each does in this new context.

The central.conf input block

For the `input` block we've specified one plugin: `beats`, with one option: `port`. The `beats` plugin starts a server on our Logstash central host to listen, on port `5044`, for incoming events from Beats. On our client nodes we're going to use a Filebeat to send our events to the central server.

The central.conf output block

The contents of `central.conf`'s `output` block is fairly easy to understand. We've already seen the `stdout` plugin in Chapter 2. Incoming events will be outputted to `STDOUT` and therefore to Logstash's own log file. I've done this for debugging purposes so we will be more easily able to see our incoming events. In a production environment you would probably disable this to prevent any excess noise being generated.

We've added another plugin called `elasticsearch`. This plugin sends events from Logstash to Elasticsearch to be stored and made available for searching. We're

not configuring any options, which will mean that Logstash tries to connect to an Elasticsearch cluster located on the `localhost` on port `9200`. In our case this will be the Elasticsearch cluster we installed earlier.

Running Logstash as a service

Now we've provided Logstash with a basic centralized configuration we can start our Logstash process. You can now run or restart the Logstash service.

Listing 3.27: Starting the central Logstash server

```
$ sudo service logstash start
```

And ensure it starts when the host is booted.

Listing 3.28: Enabling the central Logstash server

```
$ sudo systemctl enable logstash
```

Checking Logstash is running

We can confirm that Logstash is running by a variety of means. First, we can use the init script itself:

Listing 3.29: Checking the Logstash server is running

```
$ sudo service logstash status●
 logstash.service - logstash
   Loaded: loaded (/etc/systemd/system/logstash.service; enabled;
 vendor preset: enabled)
   Active: active (running) since Sat 2016-11-05 10:45:34 UTC; 2
s ago
. . .
```

Finally, Logstash will send its own log output to log files in the `/var/log/logstash/` directory.

An interlude about plugins

So far we've seen a couple of different plugins. Plugins provide support for a wide variety of inputs, filters and outputs that we can use to ingest, manipulate and output our log data. Logstash ships with a wide variety of useful plugins by default. You can also add additional plugins or even write your own (see Chapter 9). Since version 1.5.0 Logstash plugins have been packaged at RubyGems. This is designed to make it easier to package and distribute them.

Plugins are managed with the `logstash-plugin` binary. It's located in the `/usr/share/logstash/bin` directory. This is the default installation location created by the Logstash packages. Let's use it now to list all of the currently installed plugins.

Listing 3.30: The plugin list command

```
$ sudo /usr/share/logstash/bin/logstash-plugin list
logstash-codec-collectd
logstash-codec-dots
logstash-codec-edn
logstash-codec-edn_lines
logstash-codec-es_bulk
logstash-codec-fluent
logstash-codec-graphite
logstash-codec-json
. . .
```

We see a list of all the plugins and codecs installed on the host. If we don't find the plugin we want we can see if it's available on the Logstash GitHub plugin organization.

We can then install a new plugin with the `logstash-plugin` binary. Let's install Logstash's JMX input plugin. We find it on GitHub here. We can install it by referencing the plugin repository name: `logstash-input-jmx`

Listing 3.31: Installing the JMX plugin with plugin

```
$ sudo /usr/share/logstash/bin/logstash-plugin install logstash-
input-jmx
Validating logstash-input-jmx
Installing logstash-input-jmx
Installation successful
```

We can also update an existing plugin.

Listing 3.32: Updating a plugin

```
$ sudo /usr/share/logstash/bin/logstash-plugin update logstash-
input-jmx
```

Or remove plugins.

Listing 3.33: Uninstalling a plugin

```
$ sudo /usr/share/logstash/bin/logstash-plugin uninstall
logstash-input-jmx
```

The Kibana Console

Also available for Logstash is a powerful web interface called Kibana that you can use to query and display your log events. The Kibana web interface is a customizable dashboard that you can extend and modify to suit your environment. It allows the querying of events, creation of tables and graphs as well as sophisticated visualizations.

Kibana is a separate product that we can install on our central Logstash host or on another host. It connects directly to our Elasticsearch instance and queries data from it.

Installing Kibana

We're going to install Kibana on our `smoker.example.com` host. Kibana is available as a download from the Elastic website and as packages for Ubuntu and Red Hat distributions.

Let's install Kibana now. First, if we haven't already, we install the Elastic.co package key.

Listing 3.34: Downloading the Elastic package key

```
$ wget -O - https://artifacts.elastic.co/GPG-KEY-elasticsearch |
 sudo apt-key add -
```

Next, unless we've already done it above, we need to add the Elastic APT repository to our host.

Listing 3.35: Adding the Kibana APT repository

```
$ echo "deb https://artifacts.elastic.co/packages/5.x/apt stable
 main" | sudo tee -a /etc/apt/sources.list.d/elastic-5.x.list
```

TIP If we were running on a 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 3.36: Updating the package list for Kibana

```
$ sudo apt-get update
```

And finally we can install Kibana itself.

Listing 3.37: Installing Kibana via apt-get

```
$ sudo apt-get install kibana
```

TIP The packages install Kibana into the `/usr` directory, `/usr/share/kibana`.

Configuring Kibana

To configure Kibana we use the `kibana.yml` file in the `/etc/kibana/` directory. 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 3.38: The Kibana 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 can 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. This matches the Elasticsearch server we've just installed. If we installed Kibana on another host we'd specify the host name of a node of that cluster.

The configuration file also contains other settings, including the ability to setup TLS-secured access to Kibana.

Running Kibana

To run Kibana we use the `kibana` service.

Listing 3.39: Running Kibana

```
$ sudo service kibana start
```

This will run the Kibana console as a service on port 5601 of our host.

And we need to ensure Kibana starts when our host boots.

Listing 3.40: Starting Kibana at boot

```
$ sudo systemctl enable kibana
```

We then see the Kibana console by browsing to `http://10.0.0.1:5601`.

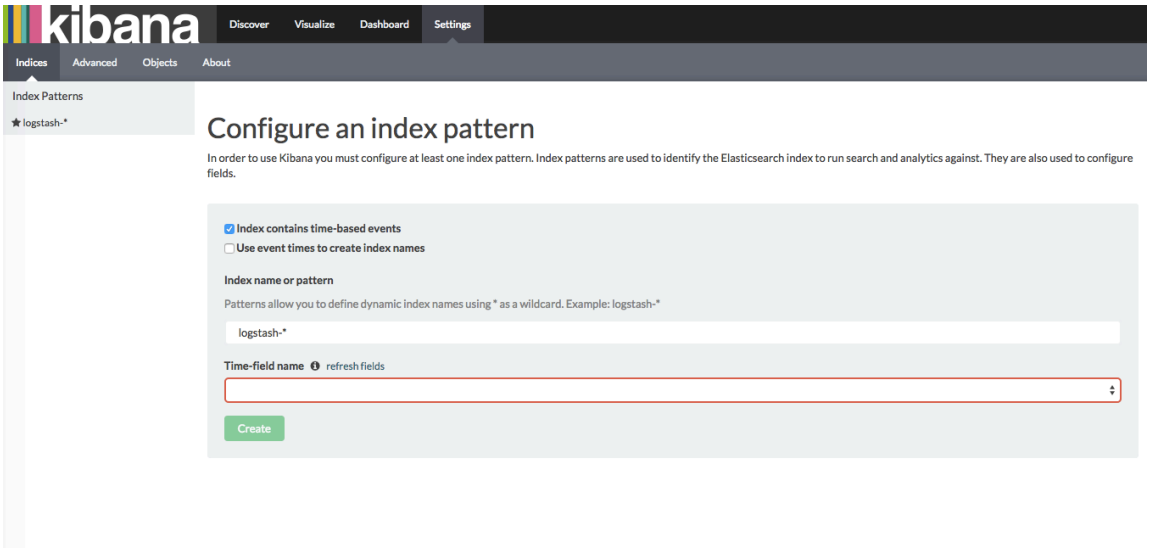

Figure 3.1: Our Kibana console

From this setup screen we need to select the indexes that Kibana will search and visualize. By default Kibana has populated `logstash-*` as our index pattern. We also need to select which field contains event timestamps. Open the dropdown and select `@timestamp`. Then 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 otherwise you'll need to reconfigure Kibana.

Now 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.

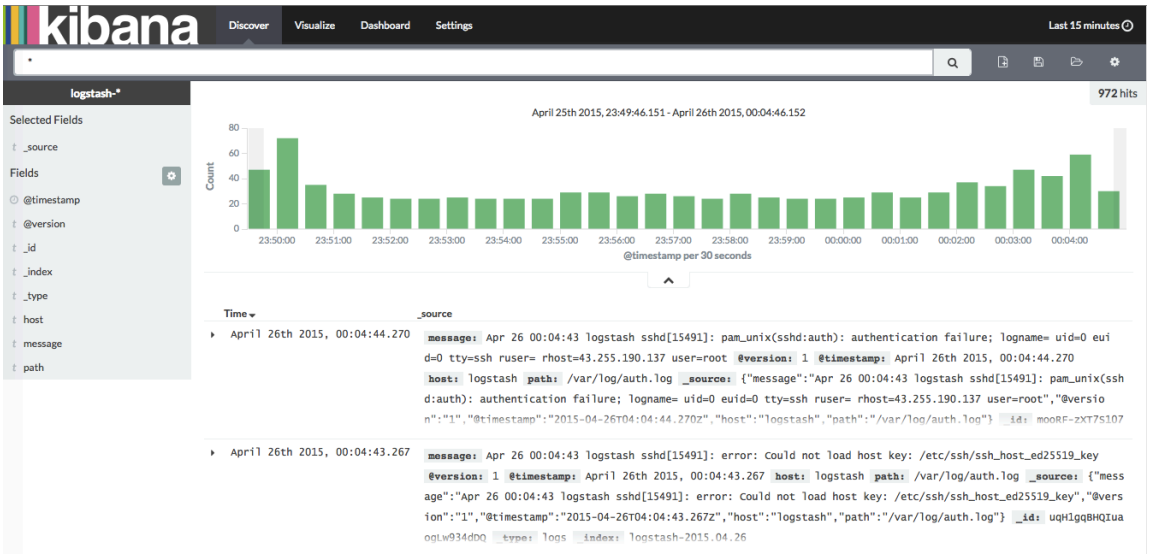

Figure 3.2: Our Kibana Discovery console

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 and any images we'd show you 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 log events.

There's also some great blog posts on Kibana here and here.

Installing a Filebeat on our first agent

Our central server is now idling waiting to receive events so let's make it happy and set up a Beat, specifically the Filebeat, to send some of those events to it. We're going to choose one of our CentOS hosts, `maurice.example.com` with an IP address of `10.0.0.10` as our first agent.

NOTE We could also install Logstash itself on the agent and use it to collect and send on our logs. This is a pretty heavyweight solution though and can often result in our logging collection consuming more resources than our application.

Agent

- Hostname: maurice.example.com
- IP Address: 10.0.0.10

On the agent we're going to begin with sending some Syslog events to the central Logstash server.

Installing the Filebeat

Next we need to install and configure a Filebeat on the remote agent. Let's install it now.

First, we download the Yum GPG key.

Listing 3.41: Adding the Yum GPG key

```
$ sudo rpm --import https://artifacts.elastic.co/GPG-KEY-
elasticsearch
```

We'll now add the Elastic Yum repository to our host. Create a file called `/etc/yum.repos.d/elastic.repo` and add the following content.

Listing 3.42: Adding the Logstash Yum repository

```
[logstash-5.x]
name=Elastic repository for 5.x packages
baseurl=https://artifacts.elastic.co/packages/5.x/yum
gpgcheck=1
gpgkey=https://artifacts.elastic.co/GPG-KEY-elasticsearch
enabled=1
autorefresh=1
type=rpm-md
```

We then install Filebeat via the `yum` command.

Listing 3.43: Install Logstash via yum

```
$ sudo yum install filebeat
```

Our agent configuration

Now we've got our base in place, let's edit our Filebeat's agent configuration in `/etc/filebeat/filebeat.yml`. By default, when installed, a Filebeat collects all of logs from files in the `/var/log` directory. Filebeats collects logs using components called prospectors. Let's look at the Prospectors section of the Filebeat configuration and make some changes.

Listing 3.44: The Prospectors section

```
#========================== Filebeat prospectors
=============================
filebeat.prospectors:

# Each - is a prospector. Most options can be set at the
prospector level, so
# you can use different prospectors for various configurations.
# Below are the prospector specific configurations.

- input_type: log
  document_type: syslog
  paths:
    - /var/log/secure

. . .
```

We've added one option: `document_type`. The `document_type` adds a type to our log events, in our case we've add this with a value of `syslog`. We're going to use this type to route and manipulate log events later in this book.

The `paths` option specifies a list of paths to crawl and fetch log entries from. In the default case this will be all files, `*.log`, in the `/var/log` directory. We've updated this line to only prospect from one file: `/var/log/secure`. This file holds security-related events.

By default, Filebeat outputs directly to an Elasticsearch cluster. We're going to quickly change that to point to our new central Logstash server. To do this we find the Elasticsearch output section.

Listing 3.45: The filebeat.yml output section

```
#------------------------- Elasticsearch output ---------------
-------------
#output.elasticsearch:
  # Array of hosts to connect to.
#  hosts: ["localhost:9200"]

. . .

#----------------------------- Logstash output -----------------
-------------
output.logstash:
  # The Logstash hosts
  hosts: ["10.0.0.1:5044"]

. . .
```

We comment out the `output.elasticsearch` block and uncomment the `output.logstash` section. Inside that block we specify the IP address of our central Logstash server, `10.0.0.1`, and the port we specified in our `beats` input on the central server: `5044`.

Installing Filebeat as a service

Now we've provided Filebeat with a basic remote configuration we can start it as a service.

Listing 3.46: Starting the Filebeat service

```
$ sudo service filebeat start
```

And ensure it's enabled at boot.

Listing 3.47: Starting Filebeat at boot

```
$ sudo chkconfig --add filebeat
```

Checking Filebeat is running

We can confirm that Filebeat is running by a variety of means. First, we can use the `service` command:

Listing 3.48: Checking Filebeat is running

```
$ sudo service filebeat status●
 filebeat.service - filebeat
   Loaded: loaded (/lib/systemd/system/filebeat.service;
disabled; vendor preset: enabled)
   Active: active (running) since Thu 2016-11-03 21:36:55 UTC; 5
min ago
     Docs: https://www.elastic.co/guide/en/beats/filebeat/
current/index.html
. . .
```

Finally, Filebeat will send its own log output to log files in the `/var/log/filebeat/` directory.

Sending our first events

We've now got our central server and our first agent set up and configured. We're monitoring and sending all log events from log files in the `/var/log` directory. Any new events logged to it should now be passed to Filebeat and then sent to the central server. They'll be processed, passed to Elasticsearch, indexed and made available to search.

So how do we send some initial events? The `/var/log/secure` file is the destination for security-relevant system logs including log in activity. So let's login to our host via SSH and generate some messages. Now let's generate a specific event by SSH'ing into Maurice.

Listing 3.49: Connecting to Maurice via SSH

```
smoker$ ssh root@maurice.example.com
```

NOTE We could also use a tool like `logger` here to generate some events. We'll see `logger` again in Chapter 4.

On the central server though one of our outputs is Elasticsearch via the `elasticsearch` plugin. So we can confirm that our events are being received and sent to Elasticsearch, indexed, and are available to search by querying Elasticsearch.

We check this by querying the Elasticsearch server via its HTTP interface. To do this we're going to use the `curl` command.

Listing 3.50: Querying the Elasticsearch server

```
$ curl "http://localhost:9200/_search?q=type:syslog
pretty=true""took" : 3,"timed_out" : false,"_shards" : "total" :
10,"successful" : 10,"failed" : 0,"hits" : "total" :
5,"max_score" : 0.5945348,"hits" : [ "_index" :
"logstash-2016.11.05","_type" : "secure","_id" :
"ZSMs-WbdRIqLmszB5w_igw","_score" : 0.5945348, "_source" :
"message":"Dec 9 07:53:16 maurice.example.com sshd[2352]:
pam_unix(sshd:session): session opened for user root by
(uid=0)","@timestamp":"2016-11-
05T07:53:16.737Z","@version":"1","host":"maurice.example.com","path":"/var/log/secu
. .
```

Here we've issued a `GET` to the Elasticsearch server running on the `localhost` on port `9200`. We've told it to search all indexes and return all events with `type` of `syslog`. This `type` was set by the `document_type` option we set in the Filebeat configuration on the remote agent.

We've also passed `pretty=true` to return our event stream in the more readable 'pretty' format. You can see it's returned some information about how long the query took to process and which indexes were hit. But more importantly it's also returned some events which means our Elasticsearch server is operational and we can search for our events.

When we look at the output from Elasticsearch we should see events related to our login attempt. Let's look at one of those events:

Listing 3.51: A Logstash login event

```
{
  "message":"Dec 9 07:53:16 maurice.example.com sshd[2352]:
pam_unix(sshd:session): session opened for user root by (uid=0)",

  "@timestamp":"2016-11-05T07:53:16.737Z",
  "@version":"1",
  "host":"maurice.example.com",
  "path":"/var/log/secure",
  type":"syslog"
}
```

We see it is made up of the fields we saw in Chapter 2 plus some additional fields. The `host` field shows the hostname of the host that generated the event. The `path` field shows the file `/var/log/secure` that the event was collected from. Both these fields are specific to Filebeat, that processed this event.

The `message` gives us the exact message being collected. The `@timestamp` field provides the date and time of the event. and the `@version` shows the event schema version. Lastly, the event `type` of `syslog` has been added by the `file` input.

Looking at our events in Kibana

Querying Elasticsearch isn't the ideal way the examine our events. We do have a console, Kibana, we can use to examine events. Let's take a look at one our log events in the console. We browse to `10.0.0.1:5601` and we should see the `Discover` tab. Our events should have automatically populated in the tab and we can use the available fields to query or dissect specific event.

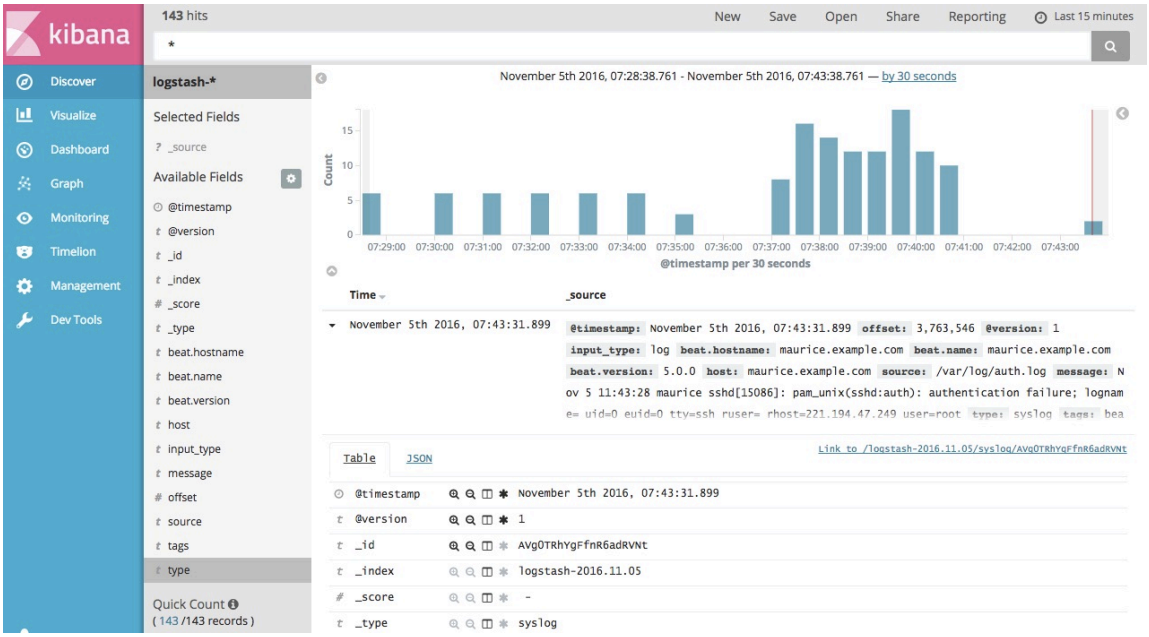

Figure 3.3: Our event in Kibana

Summary

We've made a great start on our log management project. In this chapter we've installed and configured Logstash and Elasticsearch on a central server.

We've installed and configured Filebeat on a remote agent and we can easily replicate this configuration (preferably using configuration management tools like Puppet and Chef).

We're collecting log events from a Syslog log file and transmitting them to our central server. We're indexing them and making them searchable via Elasticsearch and the Kibana web interface.

In the next chapter we're going to expand on our implementation and look at processing some additional log sources, especially in situations when we can't deploy an agent.

Chapter 4

Shipping Events

Our log management project is going well. We've got some of our Syslog messages centralized and searchable but we've hit a snag. We've discovered some hosts and devices in our environment that can't be managed with an agent. There are a few different devices that all have varying reasons for not being able to run the agent:

- Small virtual machine with limited memory insufficient to run an agent.
- Some embedded devices and appliances without the ability to install software and hence run the agent.
- Some outsourced managed hosts where you can't install software of your own.

So to address these hosts we're going to make a slight digression in our project and look at alternatives to running an agent and getting events to our central Logstash server.

We're going to look at using Syslog, the traditional Linux/Unix logging framework, for sending events to Logstash.

We're also going to explore the Filebeat log forwarding agent, part of the Beats family of collection tools, which also include Network data, Metrics and Windows

Event Log data. We first saw Filebeat in Chapter 3 but we're going to dive a bit deeper into its capabilities.

Using Syslog

The easiest way we can get our recalcitrant devices to log to Logstash is using a more traditional logging method: Syslog. Instead of using an agent to send our logs we can enable existing Syslog daemons or services to do it for us.

To do this we're going to configure our central Logstash server to receive Syslog messages and then configure Syslog on the remote hosts to send to it. We're also going to show you how to configure a variety of Syslog services.

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 like Linux and is heavily used by applications running on these platforms as well as printers and networking devices like routers, switches and firewalls.

As a result of its ubiquity on these types of platforms it's 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 4.1: A Syslog message

```
Dec 15 14:29:31 joker systemd-logind[2113]: New session 31581 of
 user bob.
```

They consist of a timestamp, the host that generated the message (here `joker`), 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
- etcetera

The facility specifies the type of message generated, for example messages from the `AUTH` facility usually relate to security or authorization, the `KERN` facility are usually kernel messages or the `MAIL` facility usually indicates 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 Logstash for Syslog

Configuring Logstash to receive Syslog messages is really easy. All we need to do is add the `syslog` input plugin to our central server's `/etc/logstash/conf.d/central.conf` configuration file. Let's do that now:

Listing 4.2: Adding the ‘syslog‘ input

```
input {
  beats {
    port => 5044
  }
  syslog {
    type => syslog
    port => 5514
  }
}
output {
  stdout { }
  elasticsearch { }
}
```

You can see that in addition to our `beats` input we've now got `syslog` enabled and we've specified two options:

Listing 4.3: The ‘syslog‘ input

```
syslog {
  type => syslog
  port => 5514
}
```

The first option, `type`, tells Logstash to label incoming events as `syslog` to help us to manage, filter and output these events. The second option, `port`, opens port 5514 for both TCP and UDP and listens for Syslog messages. By default most Syslog servers can use either TCP or UDP to send Syslog messages and when being used to centralize Syslog messages they generally listen on port 514. Indeed, if not specified, the `port` option defaults to 514. We've chosen a different port here to separate out Logstash traffic from any existing Syslog traffic flows you might

have. Additionally, since we didn't specify an interface (which we could do using the `host` option) the `syslog` plugin will bind to `0.0.0.0` or all interfaces.

TIP You can find the full list of options for the `syslog` input plugin here.

Now, if we restart our Logstash agent, we should have a Syslog listener running on our central server.

Listing 4.4: Restarting the Logstash server

```
$ sudo service logstash restart
```

You should see in your `/var/log/logstash/logstash.log` log file some lines indicating the `syslog` input plugin has started:

Listing 4.5: Syslog input startup output

```
{:message=>"Starting syslog udp listener", :address=>"
0.0.0.0:5514", :level=>:info}
{:message=>"Starting syslog tcp listener", :address=>"
0.0.0.0:5514", :level=>:info}
```

NOTE To ensure connectivity you will need make sure any host or intervening network firewalls allow connections on TCP and UDP between hosts sending Syslog messages and the central server on port 5514.

Configuring Syslog on remote agents

There are a wide variety of hosts and devices we need to configure to send Syslog messages to our Logstash central server. Some will be configurable by simply specifying the target host and port, for example many appliances or managed devices. In their case we'd specify the hostname or IP address of our central server and the requisite port number.

Central server

- Hostname: smoker.example.com
- IP Address: 10.0.0.1
- Syslog port: 5514

In other cases our host might require its Syslog daemon or service to be specifically configured. We're going to look at how to configure three of the typically used Syslog daemons to send messages to Logstash:

- RSyslog
- Syslog-NG
- Syslogd

We're not going to go into great detail about how each of these Syslog servers works but rather focus on how to send Syslog messages to Logstash. Nor are we going to secure the connections. The `syslog` input and the Syslog servers will be receiving and sending messages unencrypted and unauthenticated.

Assuming we've configured all of these Syslog servers our final environment might look something like:

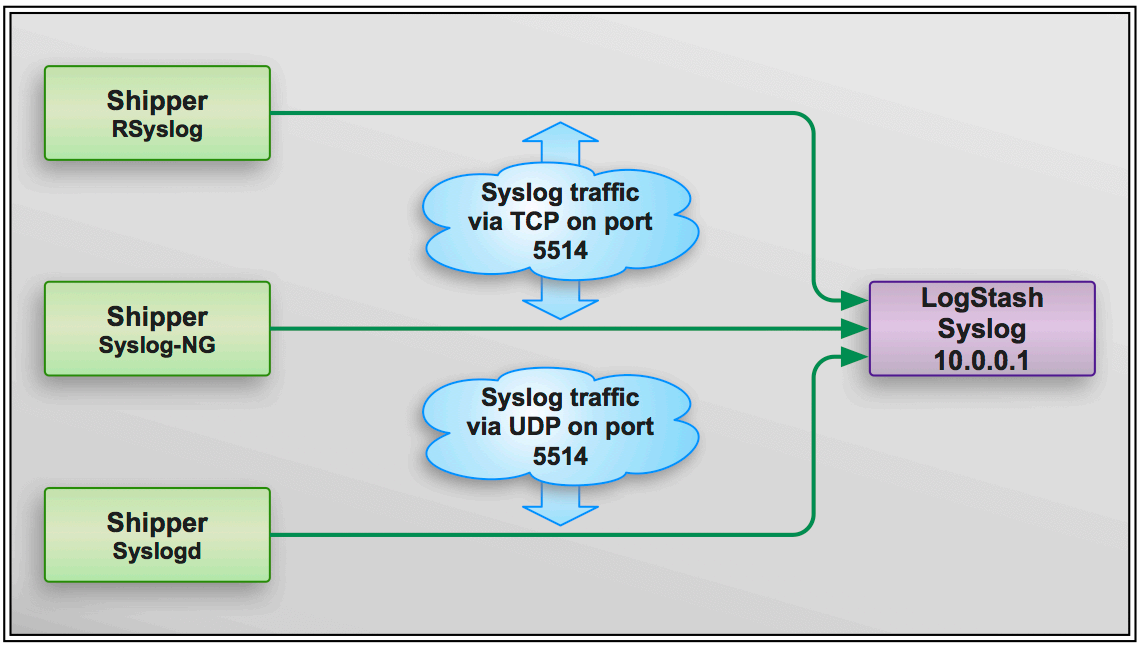

Figure 4.1: Syslog shipping to Logstash

WARNING As I mentioned above Syslog has some variations between platforms. The Logstash `syslog` input plugin supports RFC3164 style syslog with the exception that the date format can either be in the RFC3164 style or in ISO8601. If your Syslog output isn't compliant with RFC3164 then this plugin will probably not work. We'll look at custom filtering in Chapter 5 that may help parse your specific Syslog variant.

Configuring 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

comes with a modular plug-in system.

TIP In addition to supporting Syslog output Logstash also supports the RSyslog specific RELP protocol.

We're going to add Syslog message forwarding to our RSyslog configuration file, usually `/etc/rsyslog.conf` (or on some platforms inside the `/etc/rsyslog.d/` directory). To do so we're going to add the following line to the end of our `/etc/rsyslog.conf` file:

Listing 4.6: Configuring RSyslog for Logstash

```
*.* @@smoker.example.com:5514
```

NOTE If you specify the hostname, here `smoker.example.com`, your host will need to be able to resolve it via DNS.

This tells RSyslog to send all messages using `*.*`, which indicates all facilities and priorities. You can specify one or more facilities or priorities if you wish, for example:

Listing 4.7: Specifying RSyslog facilities or priorities

```
mail.* @@smoker.example.com:5514
*.emerg @@joker.example.com:5514
```

The first line would send all `mail` facility messages to our `smoker` host and the second would send all messages of `emerg` priority to the host `joker`.

The `@@` tells RSyslog to use TCP to send the messages. Specifying a single `@` uses UDP as a transport.

TIP I would strongly recommend using the more reliable and resilient TCP protocol to send your Syslog messages.

If we then restart the RSyslog daemon, like so:

Listing 4.8: Restarting RSyslog

```
$ sudo /etc/init.d/rsyslog restart
```

Our host will now be sending all the messages collected by RSyslog to our central Logstash server.

The RSyslog imfile module

One of RSyslog's modules provides another method of sending log entries from RSyslog. You can use the imfile module to transmit the contents of files on the host via Syslog. The `imfile` module works much like Logstash's `file` input and supports file rotation and tracks the currently processed entry in the file.

To send a specific file via RSyslog we need to enable the `imfile` module and then specify the file to be processed. Let's update our `/etc/rsyslog.conf` file (or if your platform supports the `/etc/rsyslog.d` directory then you can create a file-specific configuration file in that directory).

Listing 4.9: 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 internal for events to 10 seconds. It only needs to be specified once in your configuration.

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 4.10: 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`.

Now, once you've restarted RSyslog, it will be monitoring this file and sending any new lines via Syslog to our Logstash instance (assuming we've configured RSyslog as suggested in the previous section).

TIP You can find the full RSyslog documentation here.

Configuring Syslog-NG

Whilst largely replaced in modern distributions by RSyslog, there are still a lot of platforms that use Syslog-NG including Gentoo, FreeBSD, Arch Linux and HP UX. Like RSyslog, Syslog-NG is a fully featured Syslog server but its configuration is a bit more substantial than what we needed for RSyslog.

Syslog-NG configuration comes in four types:

- `source` statements - where log messages come from.
- `destination` statements - where to send log messages.
- `filter` statements - how to filter or process log messages.
- `log` statements - actions that combine source, destination and filter statements.

Let's look inside an existing Syslog-NG configuration. Its configuration file is usually `/etc/syslog-ng.conf` or `/etc/syslog-ng/syslog-ng.conf`. You'll usually find a line something like this inside:

Listing 4.11: Syslog-NG s_src source statement

```
source s_src { unix-dgram("/dev/log"); internal(); file("/proc/
kmsg" program_override("kernel"));
};
```

This basic `source` statement collects Syslog messages from the host, kernel messages and any internal messages to Syslog-NG. This is usually the default `source` on most distributions and platforms. If you don't see this `source` your Syslog-NG server may not be collecting Syslog messages and you should validate its configuration. You may also see additional `source` statements, for example collecting messages via the network from other hosts.

We then need to define a new `destination` for our Logstash server. We do this with a line like so:

Listing 4.12: New Syslog-NG destination

```
destination d_logstash { tcp("10.0.0.1" port(5144)); };
```

This tells Syslog-NG to send messages to IP address `10.0.0.1` on port 5144 via TCP. If you have domain name resolution you could instead specify our Logstash server's host name.

Lastly, we will need to specify a `log` action to combine our `source` or sources and our `destination`

Listing 4.13: New Syslog-NG log action

```
log { source(s_src); destination(d_logstash); };
```

This will send all Syslog messages from the `s_src` source to the `d_logstash` destination which is our central Logstash server.

To enable the message transmission you'll need to restart Syslog-NG like so:

Listing 4.14: Restarting Syslog-NG

```
$ sudo /etc/init.d/syslog-ng restart
```

TIP You can find the full Syslog-NG documentation here.

Configuring Syslogd

The last Syslog variant we're going to look at configuring is the older style Syslogd. While less common it's still frequently seen on older distribution versions and especially in the more traditional Unix platforms.

TIP This includes many of the *BSD-based platforms including OSX.

Configuring Syslogd to send on messages is very simple. Simply find your Syslogd configuration file, usually `/etc/syslog.conf` and add the following line at the end of the file:

Listing 4.15: Configuring Syslogd for Logstash

```
*.* @smoker.example.com:5514
```

TIP You can find more details about Syslogd configuration here.

This will send all messages to the host `smoker.example.com` on UDP port 5514. It is important to note that Syslogd generally does not support sending messages via TCP. This may be a problem for you given UDP is a somewhat unreliable protocol: there is absolutely no guarantee that the datagram will be delivered to the destination host when using UDP. Failure rates are typically low but for certain types of data including log events losing them is potentially problematic. You should take this into consideration when using Syslogd and if possible upgrade to a more fully featured Syslog server like Syslog-NG or RSyslog.

Once you've configured the Syslogd you'll need to restart the daemon, for example:

Listing 4.16: Restarting Syslogd

```
$ sudo /etc/init.d/syslogd restart
```

Other Syslog daemons

There are a variety of other Syslog daemons including several for Microsoft Windows. If you need to configure these then please see their documentation.

- Snare for Windows
- KiwiSyslog
- Syslog-Win32
- Cisco devices
- Checkpoint
- Juniper
- F5 BigIP
- HP Jet Direct

WARNING Remember not all of these devices will produce RFC-compliant Syslog output and may not work with the `syslog` input. We'll look at custom filtering in Chapter 5 that may assist in working with your Syslog variant. This blog post on Syslog parsing might also interest.

Testing with logger

Most Unix and Unix-like platforms come with a handy utility called `logger`. It generates Syslog messages that allow you to easily test if your Syslog configuration is working. You can use it like so:

Listing 4.17: Testing with logger

```
$ logger "This is a syslog message"
```

This will generate a message from the `user` facility of the priority `notice` (`user.notice`) and send it to your Syslog process.

TIP You can see full options to change the facility and priority of logger messages here.

Assuming everything is set up and functioning you should see the resulting log event appear on your Logstash server:

Listing 4.18: Logstash log event from Syslog

```
{
  "host" => "joker.example.com",
  "priority" => 13,
  "timestamp" => "Dec 17 16:00:35",
  "logsource" => "joker.example.com",
  "program" => "bob",
  "pid" => "23262",
  "message" =>"This is a syslog message",
  "severity" => 5,
  "facility" => 1,
  "facility_label" => "user-level",
  "severity_label" => "Notice",
  "@timestamp" => "2012-12-17T16:00:35.000Z",
  "@version => "1",
  "message" => "<13>Dec 17 16:00:35 joker.example.com bob[23262]:
 This is a syslog message",
  "type" => "syslog"
}
```

Filebeat

Filebeat is a lightweight, open source shipper for logs. It replaces the legacy Logstash Forwarder or Lumberjack. It can tail logs, manages log rotation and can send log data on to Logstash or even directly to Elasticsearch.

Filebeat is part of a larger collection of data shipping tools called Beats. There are several other Beats in development, including community contributions, for monitoring things like Docker and Nginx. Beats are licensed with the Apache 2.0 license and written in Golang.

TIP There's also a Windows Event Log beat called Winlogbeat if you're collecting

logs on Microsoft Windows.

We first saw Filebeat in Chapter 3 but let's dive in a bit more.

Configure Filebeat on our central server

Let's first revisit our configuration on our central server to receive data from Filebeat. To do this we use the input plugin called `beats` we introduced in Chapter 3. We should be able to see our `beats` plugin in our `central.conf` configuration file.

Listing 4.19: The beats input

```
input {
  syslog {
    type => syslog
    port => 5514
  }
  beats {
    port => 5044
  }
}
output {
  stdout { }
  elasticsearch { }
}
```

Remember we added the `beats` plugin and specified one option: `port`. The `port` option controls which port Logstash will receive logs from, here `5044`.

TIP You can find the full documentation for the `beats` input on the Elastic site.

Installing Filebeat on the remote host

Now let's look at downloading and installing Filebeat on a remote agent. We're going to choose a new Ubuntu host called `gangsteroflove.example.com`. This is the elongated explanation and deep dive into Filebeat that we didn't take in Chapter 3. It'll also show us an install on an Ubuntu host.

We can install Filebeat as a package via Apt. It's also available as an RPM, a tarball or a Windows executable installer from the Elastic.com download site.

Let's start by adding the appropriate GPG key for validating the packages.

Listing 4.20: Adding the Elasticsearch GPG key

```
$ wget -O - https://artifacts.elastic.co/GPG-KEY-elasticsearch |
 sudo apt-key add -
```

You may also need the `apt-transport-https` package.

Listing 4.21: Installing apt-transport-https

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

Now let's add the APT repository configuration.

Listing 4.22: Adding the Elastic APT repository

```
$ echo "deb https://artifacts.elastic.co/packages/5.x/apt stable
 main" | sudo tee -a /etc/apt/sources.list.d/elastic-5.x.list
```

TIP If we were running on a Red Hat or a derivative we would install the appropriate Yum repository.

We then run an `apt-get update` to refresh our package list.

Listing 4.23: Updating the package list

```
$ sudo apt-get update
```

And finally we can install Filebeat itself.

Listing 4.24: Installing Filebeat via apt-get

```
$ sudo apt-get install filebeat
```

After installation you can see that an example configuration file, `filebeat.yml`, has been created in the `/etc/filebeat` directory.

Configuring Filebeat

Filebeat is configured via a YAML file called `filebeat.yml`, located in the `/etc/filebeat` directory. Filebeat comes with a commented example file that explains all of Filebeat's local options. Let's skip this file and create our own file now.

Listing 4.25: Our new filebeat.yml file

```
filebeat.prospectors:
- input_type: log
  paths:
    - /var/log/*.log
  input_type: log
  document_type: syslog
  registry: /var/lib/filebeat/registry
output.logstash:
  hosts: ["10.0.0.1:5044"]
logging.to_files: true
logging.files:
  path: /var/log/filebeat
  name: filebeat
  rotateeverybytes: 10485760
```

The `filebeat.yml` file is divided into stanzas. The most relevant to us are `prospectors`, `output` and `logging`.

Prospectors

The `prospectors` tells Filebeat what files to gather logs from and the `output` tells Filebeat where to send those files. The last stanza, `logging`, controls Filebeat's own logging. Let's look at each in turn now, starting with `prospectors`.

Listing 4.26: The prospectors section

```
filebeat.prospectors:

- input_type: log
  paths:
    - /var/log/*.log
  document_type: syslog
  registry: /var/lib/filebeat/registry
```

Each stanza, marked with a `paths` statement, represents a file or collection of files you want to "prospect". Here we've grabbed all of the files ending in `*.log` in the `/var/log` directory. The `input_type` controls what sort of file is being read, here a standard log file. You can also use this setting to read from `STDIN`. The last option, `document_type`, controls the value of the `type` field in Logstash. The default is `log` but we've updated it to `syslog` so we can distinguish where our logs are coming from. The last option, `registry`, records file offsets and we'll talk more about it in a moment.

To match files and directories, Filebeat supports all Golang-style globs. For example, we could also get everything in subdirectories too with a glob.

Listing 4.27: The prospectors section

```
filebeat.prospectors:

- input_type: log
  paths:
    - /var/log/*/*.log
. . .
```

Or multiple sets of paths like so:

Listing 4.28: The prospectors section

```
filebeat.prospectors:

- input_type: log
  paths:
    - /var/log/*/*.log
    - /opt/application/logs/*.log
. . .
```

Filebeat will grab all files ending in `*.log` from both these paths.

Filebeat will also take care of log rotation. It recognizes when a file has been rotated and grabs the new file. Filebeat also handles tracking progress reading a file. When Filebeat reads a file it will mark its current read position in the file in a catalogue called a registry. The default registry, which we've defined using the `registry` option, is at `/var/lib/filebeat/registry`. Let's look inside that file.

Listing 4.29: The /var/lib/filebeat/registry file

```
{"/var/log/auth.log":{"source":"/var/log/auth.log","offset"
:956674,"FileStateOS":{"inode":1180057,"device":64769}},"/var/
log/dpkg.log":{"source":"/var/log/dpkg.log","offset":23515,"
FileStateOS":{"inode":1180391,"device":64769}},"/var/log/kern.
log":{"source":"/var/log/kern.log","offset":54270249,"
FileStateOS":{"inode":1180046,"device":64769}}}
```

We see a list of files that Filebeat is collecting logs from and their current offset. If we were to restart Filebeat then it would check the `registry` file and resume collecting logs from those offsets. This stops duplicate logs being sent or Filebeat restarting logging from the start of a file rather than the current point. If you need to reset the registry you can just delete the `/var/lib/filebeat/registry` file.

Tags and fields

Filebeat also offers us the ability to add fields and tags to our log events. Let's start with adding some tags to our events. To do this we specify an array of tags using the `tags` option.

Listing 4.30: Adding tags to a prospector

```
filebeat.prospectors:

- input_type: log
  tags: [ "this", "is", "a", "tag" ]
  paths:
    - /var/log/*/*.log
. . .
```

This would add the tags `this`, `is`, `a`, `tag` to each event that this prospector collects.

We can also add fields to each event using the `fields` option.

Listing 4.31: Adding tags to a prospector

```
filebeat.prospectors:

- input_type: log
  fields:
    dc: nj
  fields_under_root: true
  paths:
    - /var/log/*/*.log
. . .
```

This would add a field entitled `dc` with a value of `nj` to the event. The `fields_under_root` option controls where in the event the field is added. If you specify `true` then the field will be at the root of the event. If set to `false` then

it'll be located underneath a field called `fields`.

Filebeat outputs

Now we've defined where we want to collect logs from we now need to define where to send those logs. Filebeat can send log entries from the host to Logstash or even directly to Elasticsearch. It does that in the `output` stanza. Let's look at our `output` stanza now.

Listing 4.32: The Filebeat output stanza

```
output.logstash:
  hosts: ["10.0.0.1:5044"]
```

We've defined an output type of `logstash` and specified the `hosts` option. This tells Filebeat to connect to a Logstash server. The `hosts` option is an array that can contain one or more Logstash hosts running the `beats` input plugin. In our case we're connecting to the Logstash host at `10.0.0.1` on port `5044`.

Lastly, we want Filebeat to log some information about what it is doing. To handle this we configure the `logging` stanza.

Listing 4.33: The Filebeat logging stanza

```
logging.to_files: true
logging.files:
  path: /var/log/filebeat
  name: filebeat
  rotateeverybytes: 10485760
```

Here we've configured the `to_files` option to `true` to tell Filebeat to log to a file. We could also log to Syslog or `STDOUT`. We've then told Filebeat where to log,

inside the `files` block. We have given Filebeat a `path`, `/var/log/filebeat`, the `name` of the file to log to and controlled when the file will rotate, when it fills up to `rotateeverybytes` of `10485760` or 10Mb.

TIP Filebeat is hugely configurable. You can send data with TLS, control network and transport options like back-off and manage how files are handled when they rotate. Amongst many other settings. You'll find the commented `filebeat.yml` example file very useful for exploring settings and further documentation is available in the Filebeat documentation.

To start the Filebeat service we can use the `service` command.

Listing 4.34: Starting the Filebeat service

```
$ sudo service filebeat start
```

And ensure it's enabled at boot.

Listing 4.35: Starting Filebeat at boot

```
$ sudo systemctl enable filebeat
```

If we now check out Logstash server we should see log entries arriving from our Filebeat service with a type of `syslog` from every Syslog log file in the `/var/log/` directory. We can then use the type field to route and process those logs.

Other log shippers

If the shippers in this chapter don't suit your purposes there are also several other shippers that might work for you. Most of these are legacy and largely unmaintained so please take care.

Log-Courier

The Log-Courier project is a Logstash shipper. It's lightweight and written in Go. It's focus is on log event integrity and efficiency.

Beaver

The Beaver project is another Logstash shipper. Beaver is written in Python and available via PIP.

Listing 4.36: Installing Beaver

```
$ pip install beaver
```

Beaver supports sending events via the Redis, `STDIN`, or zeroMQ. Events are sent in Logstash's `json` codec.

TIP This is an excellent blog post explaining how to get started with Beaver and Logstash.

Woodchuck

Another potential shipping option is Woodchuck. It's designed to be lightweight and is written in Ruby and deployable as a RubyGem. It currently only supports outputting events as Redis (to be received by Logstash's `redis` input) but plans include ZeroMQ and TCP output support. It has not been recently updated.

Others

- Syslog-shipper
- Remote_syslog
- Message::Passing

Summary

We've now got some of the recalcitrant hosts into our logging infrastructure via some of the methods we've learnt about in this chapter: Syslog, Filebeat or some of the other log shippers.

That should put our log management project back on track and we can now look at adding some new log sources to our Logstash infrastructure.

Chapter 5

Filtering Events with Logstash

We've added the hosts that couldn't use an agent to our Logstash environment. We've also deployed the Filebeat beat on all of our other hosts. Our project is back on track and we can start to look at some new log sources to get into Logstash. Looking at our project plan we've got four key log sources we need to tackle next:

- Apache server logs
- Postfix server logs
- Java application logs
- A custom log format for an in-house application

Let's look at each type of log source and see how we might go about getting them into Logstash. So far we've put log sources directly into Logstash without manipulating them in any way. It meant we got the message and some small amount of metadata about it (largely its source characteristics) into Logstash. This is a useful exercise. Now all our log data is centralized in one place and we're able to do some basic cross-referencing, querying and analysis.

Our current approach, however, does not add much in the way of context or additional metadata to our events. For example we don't make any use of fields or tags nor did we manipulate or adjust any of the data in any way. And it is this

contextual information that makes Logstash and its collection and management of log events truly valuable. The ability to identify, count, measure, correlate and drill down into events to extract their full diagnostic value. To add this context we're going to introduce the concept of filter plugins.

NOTE To save you cutting and pasting we've included a Logstash configuration file showing all the examples we've used in this chapter here.

Apache Logs

The first log source on our list is our Apache web servers. Example.com has a lot of web properties, they are all running on Apache and logging both accesses and errors to log files. Let's start by looking at one of the log events that has been generated:

Listing 5.1: An Apache log event

```
186.4.131.228 - - [20/Dec/2012:20:34:08 -0500] "GET /2012/12/new-
product/ HTTP/1.0" 200 10902 "http://www.example.com/20012/12/
new-product/" "Mozilla/5.0 (Windows; U; Windows NT 5.1; pl; rv
:1.9.1.3) Gecko/20090824 Firefox/3.5.3"
```

This entry was produced from Apache's Combined Log Format. You can see there is lots of useful information in this Apache log event:

- A source IP for the client.
- The timestamp.
- The HTTP method, path, and protocol.

- The HTTP response code.
- The size of the object returned to the client.
- The HTTP referrer.
- The User-Agent HTTP request header.

NOTE You can see more details on Apache logging here.

If we were to send this event to Logstash using our current configuration all of this data would be present in the `message` field but we'd then need to search for it and it seems like we could do better. Especially given we've got all these useful places to store the appropriate data.

So how do we get the useful data from our Apache log event into Logstash? There are three approaches we could take (and we could also combine one or more of them):

- Filtering events on the agent.
- Filtering events on the central server.
- Sending events from Apache in a better format.

The first two methods would rely on Logstash's `filter` plugins either running locally or on the central server. Both have pros and cons. Running locally on the agent reduces the processing load on the central server and ensures only clean, structured events are stored. But you have to maintain a more complex (and preferably managed) configuration locally. On the server side this can be centralized and hopefully easier to manage but at the expense of needing more processing grunt to filter the events.

For this initial log source, we're going to go with the last method, having Apache send custom log output. This is a useful shortcut because Apache allows us to

customize logging and we should take advantage of it. By doing this we avoid having to do any filtering or parsing in Logstash and we can concentrate on making best use of the data in Logstash.

Configuring Apache for Custom Logging

To send our log events we're going to use Apache's `LogFormat` and `CustomLog` directives to construct log entries that we can send to Logstash. The `LogFormat` directive allows you to construct custom named log formats and then the `CustomLog` directive uses those formats to write log entries, like so:

Listing 5.2: The Apache LogFormat and CustomLog directives

```
LogFormat "formatoflogevent" nameoflogformat
CustomLog /path/to/logfile nameoflogformat
```

You've probably used the `CustomLog` directive before, for example to enable logging for a virtual host, like so:

Listing 5.3: Apache VirtualHost logging configuration

```
<VirtualHost *:80>
  DocumentRoot /var/www/html/vhost1
  ServerName vhost1.example.com

  <Directory "/var/www/html/vhost1">
Options FollowSymLinks
AllowOverride All
  </Directory>

  CustomLog /var/log/httpd/vhost1.access combined

</VirtualHost>
```

In this example we're specifying the `combined` log format which refers to the default Combined Log Format that generated the event we saw earlier.

NOTE The Combined Log Format is an extension of another default format, the Common Log Format, with the added fields of the HTTP referrer and the User-Agent.

The `LogFormat` directive for Apache's Combined Log Format would be (and you should be able to find this line in your Apache configuration files):

Listing 5.4: The Apache Common Log Format LogFormat directive

```
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-
agent}i\"" combined
```

NOTE And yes `referer` is spelt incorrectly.

Each log format is constructed using `%` directives combined with other text. Each `%` directive represents some piece of data, for example `%h` is the IP address of the client connecting to your web server and `%t` is the time of the access request.

TIP You can find a full list of the `%` directives here.

As Apache's log output is entirely customizable using these `%` directives we can

write our log entries in any format we want including, conveniently, constructing structured data events. To take advantage of this we're going to use Apache's `LogFormat` directive to construct a JSON hash replicating Logstash's `json` codec. This will allow us to take advantage of the `%` directives available to add some context to our events.

Creating a Logstash log format

To create a custom log format we need to add our new `LogFormat` directive to our Apache configuration. To do this we are going to create a file called `apache_log.conf` and add it to our Apache `conf.d` directory, for example on Red Hat-based systems we'd add it to `/etc/httpd/conf.d/` and on Debian-based systems to `/etc/apache2/conf.d` or `/etc/apache2/conf-enabled/`. Populate the file with the following `LogFormat` directive:

Listing 5.5: Apache custom JSON LogFormat

```
LogFormat "{ \
  \"host\":\"host.example.com\", \
  \"path\":\"/var/log/httpd/logstash_access_log\", \
  \"tags\":[\"wordpress\",\"www.example.com\"], \
  \"message\": \"%h %l %u %t \\\"%r\\\" %>s %b\", \
  \"timestamp\": \"%{%Y-%m-%dT%H:%M:%S%z}t\", \
  \"useragent\": \"%{User-agent}i\", \
  \"clientip\": \"%a\", \
  \"duration\": %D, \
  \"status\": %>s, \
  \"request\": \"%U%q\", \
  \"urlpath\": \"%U\", \
  \"urlquery\": \"%q\", \
  \"method\": \"%m\", \
  \"bytes\": %B, \
  \"vhost\": \"%v\" \
}" logstash_apache_json
```

NOTE To save you cutting and pasting this we've included an example file here. You should edit the various sections to add your own hosts, source info and tags.

This rather complex looking arrangement produces Apache log data as a JSON hash. One of the reasons it looks so complex is that we're escaping the quotation marks and putting in backslashes to make it all one line and valid JSON.

We're specifying the `host` and `path` manually and you could use any values that suited your environment here. We're also manually specifying an array of tags in the `tags` field, here identifying that this is a Wordpress site and it is the `www.example.com` page. You would update these fields to suit your environment.

TIP To manage the `LogFormat` better I recommend managing the `log.conf` file as a Puppet or Chef template. That would allow you to centrally control values like the 'host', 'path' and 'tags' field on a host.

The `message` field contains the standard Common Log Format event that is generated by Apache. This is useful if you have other tools that consume Apache logs for which you still want the default log output.

The remaining items specified are fields and contain the core of the additional context we've added to our Apache log events. It breaks out a number of the elements of the Common Log Format into their own fields and adds a couple more items, such as `vhost` via the `%v` directive. You can easily add additional fields from the available directives if required. Remember to ensure that the field is appropriately escaped if it is required.

TIP As a reminder, you can find a full list of the `%` directives here.

Let's add the `CustomLog` directive to our `apache_log.conf` file to actually initiate the logging:

Listing 5.6: Adding the CustomLog directive

```
CustomLog /var/log/httpd/logstash_access_log
logstash_apache_json
```

Or we can apply our `CustomLog` directive to an appropriate web site or virtual host definition.

And now restart Apache to make our new configuration active.

Listing 5.7: Restarting Apache

```
$ sudo service httpd restart
```

This will result in Apache creating a log file, `/var/log/httpd/logstash_access_log`, that will contain our new log entries.

TIP Remember to add this file to your normal log rotation and you may want to consider turning off your existing Apache logging rather than writing duplicate log entries and wasting Disk I/O and storage. You could alternatively increase the tempo of your log rotation and keep short-term logs as backups and remove them more frequently.

Let's take a look at one of those entries now:

Listing 5.8: A JSON format event from Apache

```
{
  "host" => "maurice.example.com"
  "path" => "/var/log/httpd/logstash_access_log",
  "tags" => [
    [0] "wordpress",
    [1] "www.example.com"
  ],
  "message" => "10.0.0.1 - - [05/Nov/2016:21:22:52 +0000] \"GET /
 HTTP/1.1\" 304 -",
  "timestamp" => "2016-11-05T21:22:52+0000",
  "clientip" => "10.0.0.1",
  "duration" => 11759,
  "status" => 304,
  "request" => "/index.html",
  "urlpath" => "/index.html",
  "urlquery" => "",
  "method" => "GET",
  "bytes" => 0,
  "vhost" => "10.0.0.1",
  "@timestamp" => "2016-11-05T21:22:53.261Z",
  "@version" => "1",
  "type" => "apache"
}
```

TIP You can also output JSON events from Syslog using RSyslog as you can learn here. You can also achieve the same results from recent versions of the Squid proxy which has added a LogFormat capability. Similarly with Nginx.

Sending Apache events to Logstash

So how do we get those log entries from our host to Logstash? There are a number of potential ways we discovered in Chapters 3 and 4 to input the events. The easiest way is to add a prospector to our Filebeat configuration.

Listing 5.9: The filebeat Apache log prospector

```
filebeat.prospectors:

- input_type: log
  document_type: syslog
  paths:
    - /var/log/*.log

- input_type: log
  json.message_key: message
  json.keys_under_root: true
  document_type: apache
  paths:
    - /var/log/httpd/logstash_access_log

output.logstash:
  # The Logstash hosts
  hosts: ["10.0.0.1:5044"]
```

You can see we've specified two new options in our `filebeat.yml` configuration. The first `json.message_key` tells Filebeat that the message field, which will contain our JSON-encoded Apache event, is a JSON event. This will cause the `beats` input plugin to process it as JSON rather than as a plain text event.

The next option, `json.keys_under_root`, controls where the JSON-encoded field, in our case `message`, are unpacked. By default, Logstash will place all of the fields under a root key field called `json`. If `json.keys_under_root` is set to `true` then all of the event fields are unpacked and stored at the root of the event. For example, when the option is `false`, the `status` field would default to a structure of `json`

`-> status` whereas if the option is `true` then the structure would be `status` as a top-level field.

Once you've configured your Beat to send your Apache logs and restarted the required services you should see Apache log events flowing through to ElasticSearch. Let's look at one of these events in the Logstash Kibana interface:

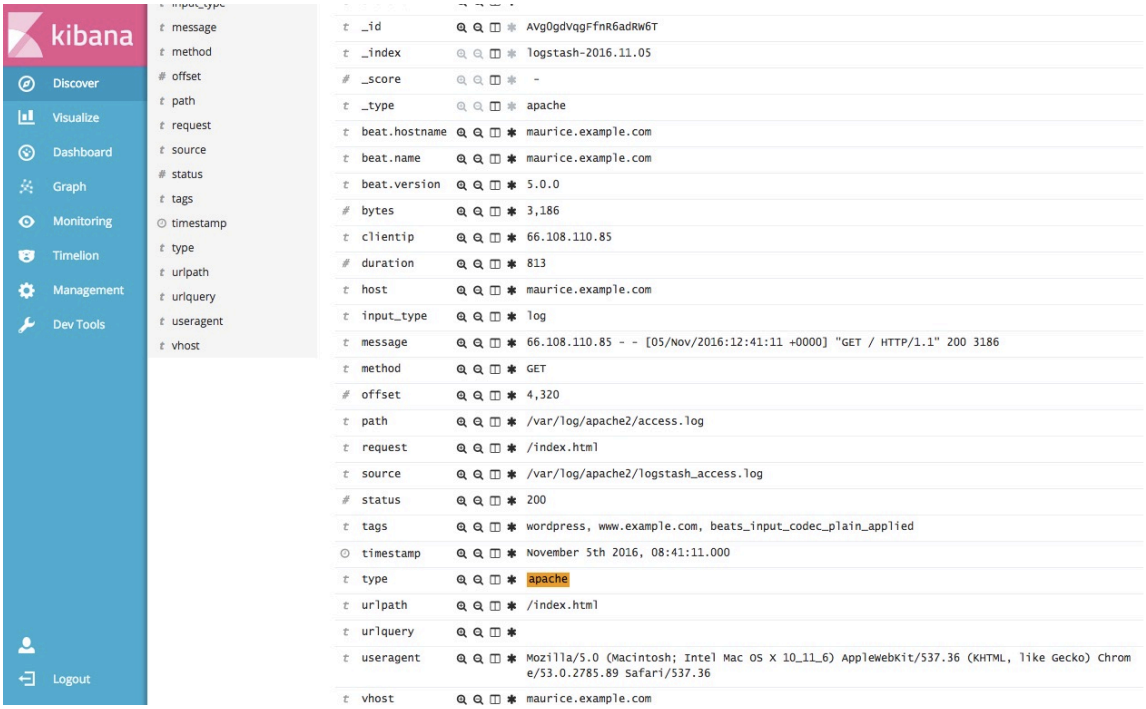

Figure 5.1: Apache log event

We can see that the various pieces of context we've added are now available as tags and fields in the Logstash Kibana interface. This allows us to perform much more sophisticated and intelligent queries on our events. For example, I'd like to see all the events that returned a 404 status code. I can now easily query this using the field named `status`:

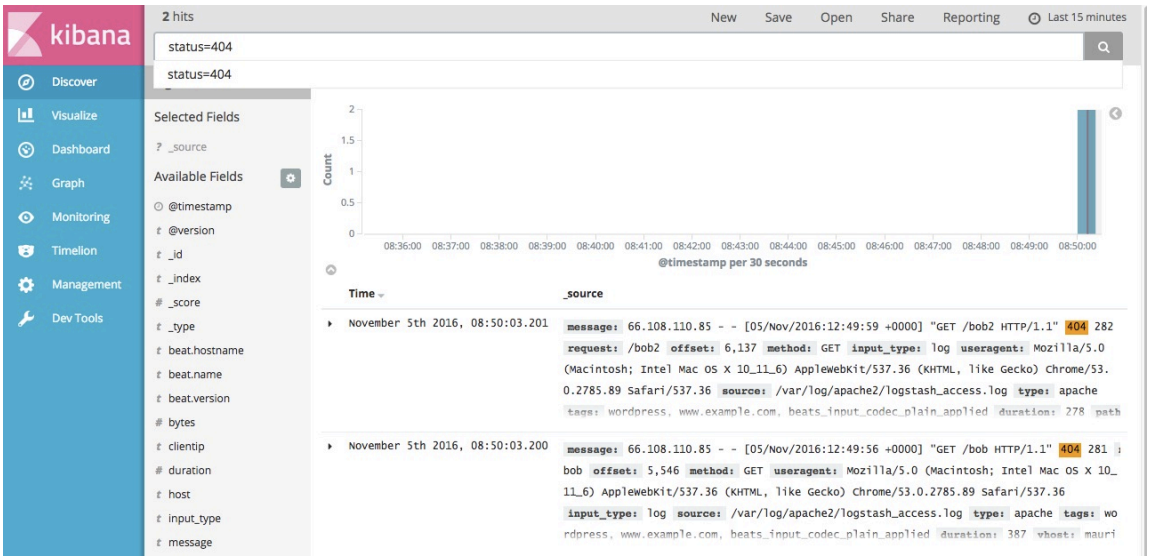

Figure 5.2: Querying for 404 status codes

We can also combine these fields to drill down in more precise queries, for example selecting specific virtual hosts and querying for status codes, specific requests and methods.

TIP We could also use filters, as we'll see shortly, to extract more data from our log entries. For example we could use the useragent or geoip filters to add user agent and GeoIP data respectively.

We can also now quickly and easily drill down into our log data to find events we care about or that are important when troubleshooting.

TIP We'll also see how these more contextual events can be output as alerts or gathered together to produce useful metrics in Chapter 7.

Postfix Logs

Now our Apache logs are pouring into Logstash we need to move onto our next target: Postfix mail server logs. Unfortunately, unlike Apache logs, we can't customize the Postfix log output. We're going to need to use our first `filter` plugins on the central server to parse the Postfix events to make them more useful to us. Let's start by looking at a Postfix log entry:

Listing 5.10: A Postfix log entry

```
Dec 24 17:01:03 localhost postfix/smtp[20511]: F31B56FF99: to=<
james@example.com>, relay=aspmx.l.google.com[2607:f8b0:400e:c01
::1b]:25, delay=1.2, delays=0.01/0.01/0.39/0.84, dsn=2.0.0,
status=sent (250 2.0.0 OK 1356368463 np6si20817603pbc.299)
```

This log entry is for a sent email and there's quite a lot going on in it with plenty of potential information that we might want to use. Adding it to Logstash in its current form, however, will result in all this information being pushed into the `message` field as we can see here with a similar event:

Listing 5.11: Unfiltered Postfix event

```
{
  "message" => "Aug 31 01:18:55 maurice postfix/smtp[25873]: 2
B238121203: to=<james@example.com>, relay=aspmx.l.google.com
[74.125.129.27]:25, delay=3.5, delays=0.05/0.01/0.47/3, dsn
=2.0.0, status=sent (250 2.0.0 OK 1377911935 tp5si709880pac.251 -
 gsmtp)",
  "@timestamp" => "2013-08-31T01:29:42.416Z",
  "@version" => "1",
  "type" => "postfix",
  "host" => "maurice.example.com",
  "path" => "/var/log/mail.log"
}
```

Yep, that's not particularly helpful to us so let's do some basic filtering with Logstash to extract some of that useful information.

Filtering

For our Postfix logs we're going to do our filtering on the central server. To do this we're going to introduce our first `filter` plugin: `grok`. The `grok` filter plugin parses arbitrary text and structures it. It does this using `patterns` which are packaged regular expressions. As not everyone is a regular expression ninja[1] Logstash ships with a large collection: 120 patterns at the time of writing - of pre-existing patterns that you can use. If needed, it is also very easy to write your own.

NOTE You can find the full list of built-in patterns in Logstash on GitHub.

[1] And stop calling people 'ninjas' anyway everyone.

Collecting Postfix logs

Firstly, let's collect our Postfix log entries. We're going to use our `maurice.example.com` host so we can add a prospector to our `/etc/filebeat/filebeat.yml` configuration file:

Listing 5.12: File input for Postfix logs

```
filebeat.prospectors:

- input_type: log
  document_type: syslog
  exclude_files: ['mail.log']
  paths:
    - /var/log/*.log

- input_type: log
  json.message_key: message
  json.keys_under_root: true
  document_type: apache
  paths:
    - /var/log/httpd/logstash_access.log

- input_type: log
  document_type: postfix
  paths:
    - /var/log/mail.*

output.logstash:
  hosts: ["10.0.0.1:5044"]
```

Here we're grabbing all log files from the `/var/log` directory that match the glob: `mail.*`. You can see we've added an option to our Syslog prospector, `exclude_files`. The `exclude_files` option specifies a regular expression to match a list of files that we want to exclude from this specific prospector. We've specified the `mail.log` file. This means that our new prospector will be the only prospector

that grabs the Postfix log files. This avoids us double-handling Postfix log files.

These log events will then be sent to our central server and received by the `beats` input plugin.

Our first filter

Now, in our `central.conf` configuration file, let's add a `grok` filter to filter these incoming events. We add the `grok` filter to the `filter` plugins section.

Listing 5.13: Postfix grok filter

```
filter {
  if [type] == "postfix" {
    grok {
      match => [ "message", "%{SYSLOGBASE}" ]
      add_tag => [ "postfix", "grokked" ]
    }
  }
}
```

We've added a `grok` filter to our `filter` block. We've first specified an `if` conditional that matches the `type` with a value of `postfix`. This is really important to our filtering process because a filter should generally only match those events for which it's relevant. So in our case only those events with a type of `postfix` will be processed by this filter. All other events will ignore the filter and move on.

NOTE You can see a full list of the `grok` filter's options here.

We've next specified the `match` option which does the hard work of actually "grokking" our log event:

Listing 5.14: The grok pattern for Postfix logs

```
match => [ "message", "%{SYSLOGBASE}" ]
```

Patterns are designed to match and extract specific data from your logs to create data structures from unstructured log strings. They are constructed of regular expressions and structured like so:

Listing 5.15: The syntax and the semantic

```
%{syntax:semantic}
```

The `syntax` is the name of the pattern, for example `SYSLOGBASE`, being used in the match. The `semantic` is optional and is an identifier for any data matched by the pattern (think of it like assigning a value to a variable).

For our pattern we've used one of Logstash's built-in patterns: `SYSLOGBASE`. Let's look at the content of this pattern which we can find here:

Listing 5.16: The SYSLOGBASE pattern

```
SYSLOGBASE %{SYSLOGTIMESTAMP:timestamp} (?:%{SYSLOGFACILITY} )?%{
SYSLOGHOST:logsource} %{SYSLOGPROG}:
```

NOTE Again you can find the full list of built-in patterns in Logstash here.

Each pattern starts with a name, which is the `syntax` we saw above. It is then

constructed of either other patterns or regular expressions. If we drill down into the patterns that make up SYSLOGBASE we'll find regular expressions at their core. Let's look at one of the patterns in SYSLOGBASE:

Listing 5.17: The SYSLOGPROG pattern

```
SYSLOGPROG %{PROG:program}(?:\[%{POSINT:pid}\])?
```

More patterns! We can see the SYSLOGPROG pattern is made up of two new patterns: PROG which will save any match as program and POSINT which will save any match as pid. Let's see if we can drill down further in the PROG pattern:

Listing 5.18: The PROG pattern

```
PROG (?:[\w._/%-]+)
```

Ah ha! This new pattern is an actual regular expression. It matches the Syslog program, in our event the postfix/smtp, portion of the log entry. This, combined with the POSINT pattern, will match the program and the process ID from our event and save them both as program and pid respectively.

So what happens when a match is made for the whole SYSLOGBASE pattern? Let's look at the very start of our Postfix log event.

Listing 5.19: Postfix date matching

```
Aug 31 01:18:55 maurice postfix/smtp[25873]:
```

Logstash will apply the pattern to this event. First matching the date portion of our event with the SYSLOGTIMESTAMP pattern and saving the value of that match

to `timestamp`. It will then try to match the `SYSLOGFACILITY`, `SYSLOGHOST` and `SYSLOGPROG` patterns and, if successful, save the value of each match too.

So now these have matched what's next? We know Logstash has managed to match some data and saved that data. What does it now do with that data? Logstash will take each match and create a field named for the semantic, for example in our current event `timestamp`, `program` and `pid` would all become fields added to the event.

The `semantic` field will be saved as a string by default. If you wanted to change the field type, for example if you wish to use the data for a calculation, you can add a suffix to the pattern to do so. For example to save a semantic as an integer we would use:

Listing 5.20: Converting semantic data

```
%{POSINT:PID:int}
```

Currently the only supported conversions are `int` for converting to integers and `float` for converting to a float.

Let's see what happens when the `SYSLOGBASE` pattern is used to grok our Postfix event. What fields does our event contain?

Listing 5.21: The Postfix event's fields

```
{
  . . .
  "timestamp"=> "Aug 31 01:18:55",
  "logsource"=> "maurice",
  "pid"=> "25873",
  "program"=> "postfix/smtp",
  . . .
}
```

NOTE If you don't specify a semantic then a corresponding field will not be automatically created. See the `named_captures_only` option for more information.

Now instead of an unstructured line of text we have a structured set of fields that contain useful data from the event that we can use.

Now let's see our whole Postfix event after it has been grokked:

Listing 5.22: A fully grokked Postfix event

```
{
  "host" => "maurice.example.com",
  "path" => "/var/log/mail.log",
  "tags" => ["postfix", "grokked"],
  "timestamp" => "Aug 31 01:18:55",
  "logsource" => "maurice",
  "pid" => "25873",
  "program" => "postfix/smtp",
  "@timestamp" => "2013-08-31T01:18:55.831Z",
  "@version" => "1",
  "message" => "Aug 31 01:18:55 maurice postfix/smtp[25873]: 2
B238121203: to=<james@example.com>, relay=aspmx.l.google.com
[74.125.129.27]:25, delay=3.5, delays=0.05/0.01/0.47/3, dsn
=2.0.0, status=sent (250 2.0.0 OK 1377911935 tp5si709880pac.251 -
 gsmtp)",
  "type" => "postfix"
}
```

Our grokked event also shows the result of another option we've used in the `grok` filter: `add_tag`. You see the `tags` field now has two tags in it: `postfix` and `grokked`.

TIP You can remove tags from events using the `remove_tag` option.

Now we've seen a very basic example of how to do filtering with Logstash. What if we want to do some more sophisticated filtering using filters we've written ourselves?

Adding our own filters

So now we've got some data from our Postfix log event but there is a lot more useful material we can get out. So let's start with some information we often want from our Postfix logs: the Postfix component that generated it, the Process ID and the Queue ID. All this information is contained in the following segment of our Postfix log event:

Listing 5.23: Partial Postfix event

```
postfix/smtp[25873]: 2B238121203:
```

So how might we go about grabbing this information? Well, we've had a look at the existing patterns Logstash provides and they aren't quite right for what we need so we're going to add some of our own.

There are two ways to specify new patterns:

- Specifying new external patterns from a file, or
- Using the `named capture` regular expression syntax.

Let's look at external patterns first.

Adding external patterns

We add our own external patterns from a file. Let's start by creating a directory to hold our new Logstash patterns:

Listing 5.24: Creating the patterns directory

```
$ sudo mkdir /etc/logstash/patterns
```

Now let's create some new patterns and put them in a file called `/etc/logstash/patterns/postfix`. Here are our new patterns:

Listing 5.25: Creating new patterns

```
COMP ([\w._\/%-]+)
COMPID postfix\/%{COMP:component}(?:\[%{POSINT:pid}\])?
QUEUEID ([0-9A-F]{,11})
POSTFIX %{SYSLOGTIMESTAMP:timestamp} %{SYSLOGHOST:hostname} %{
COMPID}: %{QUEUEID:queueid}
```

Each pattern is relatively simple and each pattern builds upon the previous patterns. The first pattern `COMP` grabs the respective Postfix component, for example `smtp`, `smtpd` or `qmgr`. We then use this pattern inside our `COMPID` pattern. In the `COMPID` pattern we also use one of Logstash's built-in patterns `POSINT` or "positive integer," which matches on any positive integers, to return the process ID of the event. Next we have the `QUEUEID` pattern which matches the Postfix queue ID, which is an up to 11 digit hexadecimal value.

TIP If you write a lot of Ruby regular expressions you may find Rubular really useful for testing them.

Lastly, we combine all the previous patterns in a new pattern called `POSTFIX`.

Now let's use our new external patterns in the `grok` filter.

Listing 5.26: Adding new patterns to grok filter

```
if [type] == "postfix" {
  grok {
    patterns_dir => ["/etc/logstash/patterns"]
    match => [ "message", "%{POSTFIX}" ]
    add_tag => [ "postfix", "grokked"]
  }
}
```

You can see we've added the `patterns_dir` option which tells Logstash to look in that directory and load all the patterns it finds in there. We've also specified our new pattern, `POSTFIX`, which will match all of the patterns we've just created. Let's look at our Postfix event we've parsed with our new pattern.

Listing 5.27: Postfix event grokked with external patterns

```
{
  "host" => "maurice.example.com",
  "path" => "/var/log/mail.log",
  "tags" => ["postfix", "grokked"],
  "timestamp" => "Aug 31 01:18:55",
  "hostname" => "maurice",
  "component" => "smtp",
  "pid" => "25873",
  "queueid" => "2B238121203",
  "@timestamp" => "2013-08-31T01:18:55.361Z",
  "@version" => "1",
  "message" => "Aug 31 01:18:55 maurice postfix/smtp[25873]: 2
B238121203: to=<james@example.com>, relay=aspmx.l.google.com
[74.125.129.27]:25, delay=3.5, delays=0.05/0.01/0.47/3, dsn
=2.0.0, status=sent (250 2.0.0 OK 1377911935 tp5si709880pac.251 -
 gsmtp)",
  "type" => "postfix"
}
```

We can see we've got new fields in the event: `component`, and `queueid`.

Using named capture to add patterns

Now let's look at the named capture syntax. It allows you to specify pattern inline rather than placing them in an external file. Let's take an example using our pattern for matching the Postfix queue ID.

Listing 5.28: A named capture for Postfix's queue ID

```
(?<queueid>[0-9A-F]{,11})
```

The named capture looks like a regular expression, prefixed with the name of the

field we'd like to create from this match. Here we're using the regular expression `[0-9A-F]{,11}` to match our queue ID and then storing that match in a field called `queueid`.

Let's see how this syntax would look in our `grok` filter replacing all our external patterns with named captures.

Listing 5.29: Adding new named captures to the grok filter

```
if [type] == "postfix" {
  grok {
    match => [ "message", "%{SYSLOGTIMESTAMP:timestamp} %{
SYSLOGHOST:hostname} postfix\/(?<component>[\w._\/%-]+)(?:\[%{
POSINT:pid}\]): (?<queueid>[0-9A-F]{,11})" ]
    add_tag => [ "postfix", "grokked"]
  }
}
```

We've used three built-in patterns and our new named capture syntax to create two new patterns: `component` and `queueid`. When executed, this `grok` filter would create the same fields as our external patterns did:

Listing 5.30: Postfix event filtered with named captures

```
{
  . . .
  "timestamp"=> "Aug 31 01:18:55",
  "hostname"=> "maurice",
  "component"=> "smtp",
  "pid"=> "25873",
  "queueid"=> "2B238121203"
  . . .
}
```

TIP If your pattern fails to match an event then Logstash will add the tag `_grokparsefailure` to the event. This indicates that your event was tried against the filter but failed to parse. There are two things to think about if this occurs. Firstly, should the event have been processed by the filter? Check that the event is one you wish to grok and if not ensure the correct `type`, tags or field matching is set. Secondly, if the event is supposed to be grokked, test your pattern is working correctly using a tool like the GrokDebugger written by Nick Ethier or the `grok` binary that ships with the Grok application.

Extracting from different events

We've now extracted some useful information from our Postfix log event but looking at some of the other events Postfix generates there's a lot more we could extract. Thus far we've extracted all of the common information Postfix events share: date, component, queue ID, etc. But Postfix events each contain different pieces of data that we're not going to be able to match with just our current pattern. Compare these two events:

Listing 5.31: Postfix event

```
Dec 26 10:45:01 localhost postfix/pickup[27869]: 841D26FFA8: uid
=0 from=<root>
Dec 26 10:45:01 localhost postfix/qmgr[27370]: 841D26FFA8: from=<
root@maurice>, size=336, nrcpt=1 (queue active)
```

They both share the initial items we've matched but have differing remaining content. In order to match both these events we're going to adjust our approach a little and use multiple `grok` filters. To do this we're going to use one of the pieces of data we have already: the Postfix component. Let's start by adjusting the `grok` filter slightly:

Listing 5.32: Updated grok filter

```
if [type] == "postfix" {
  grok {
    patterns_dir => ["/etc/logstash/patterns"]
    match => [ "message", "%{POSTFIX}" ]
    add_tag => [ "postfix", "grokked", "%{[component]}" ]
  }
}
```

You'll note we've added an additional tag, `%{[component]}`. This syntax allows us to add the value of any field as a tag. In this case if the two log lines we've just seen were processed then they'd result in events tagged with:

Listing 5.33: Postfix component tagged events

```
"tags"=> [ "postfix", "grokked", "pickup" ]
"tags"=> [ "postfix", "grokked", "qmgr" ]
```

Logstash calls this `%{field}` syntax its `sprintf` format. This format allows you to refer to field values from within other strings.

TIP You can find full details on this syntax here.

You can also refer to nested fields using this syntax, for example:

Listing 5.34: Nested field syntax

```
{
  "component" => {
    "pid" => "12345"
    "queueid" => "ABCDEF123456"
  }
}
```

If we wanted to refer to the `pid` in this nested event we would use, `%{[component][pid]}`.

TIP For top-level fields you can omit the surrounding square brackets if you wish, for example `%component`.

Next we're going to add a new `grok` filter to process a specific Postfix component in our case `qmgr`:

Listing 5.35: A grok filter for qmgr events

```
if "qmgr" in [tags] {
  grok {
    patterns_dir => ["/etc/logstash/patterns"]
    match => [ "message", "%{POSTFIXQMGR}" ]
  }
}
```

This matches any event tagged with `qmgr` and matches the `message` against the `POSTFIXQMGR` pattern. Let's look at our `/etc/logstash/patterns/postfix` file now:

Listing 5.36: The /etc/logstash/patterns/postfix file

```
COMP ([\w._\/%-]+)
COMPPID postfix\/%{COMP:component}(?:\[%{POSINT:pid}\])?
QUEUEID ([A-F0-9]{5,15}{1})
EMAILADDRESSPART [a-zA-Z0-9_.+-=:]+
EMAILADDRESS %{EMAILADDRESSPART:local}@%{EMAILADDRESSPART:remote}

POSTFIX %{SYSLOGTIMESTAMP:timestamp} %{SYSLOGHOST:hostname} %{
COMPPID}: %{QUEUEID:queueid}
POSTFIXQMGR %{POSTFIX}: (?:removed|from=<(?:%{EMAILADDRESS:from})
?>(?:, size=%{POSINT:size}, nrcpt=%{POSINT:nrcpt} \(%{GREEDYDATA:
queuestatus}\))?)
```

You can see we've added some new patterns to match email addresses and our `POSTFIXQMGR` pattern to match our `qmgr` log event. The `POSTFIXQMGR` pattern uses our existing `POSTFIX` pattern plus adds patterns for the fields we expect in this log event. The `tags` field and remaining fields of the resulting event will look like:

Listing 5.37: A partial filtered Postfix event

```
{
  . . .
  "tags" => ["postfix", "grokked", "qmgr"],
  "timestamp" => "Dec 26 20:25:01",
  "hostname" => "localhost",
  "component" => "qmgr",
  "pid" => "27370",
  "queueid" => "D1BDA6FFA8",
  "from" => "root@maurice",
  "local" => "root",
  "remote" => "maurice",
  "size" => "336",
  "nrcpt" => "1",
  "queuestatus" => "queue active"
  . . .
}
```

You can see we've now got all of the useful portions of our event neatly stored in fields that we can query and work with. From here we can easily add other `grok` filters to process the other types of Postfix events and parse their data.

Setting the timestamp

We've extracted much of the information contained in our Postfix log event but you might have noticed one thing: the timestamp. You'll notice we're extracting a timestamp from our event using the `SYSLOGTIMESTAMP` pattern which matches data like `Dec 24 17:01:03` and storing it as a field called `timestamp`. But you'll also note that each event also has a `@timestamp` value and that they are often not the same! So what's happening here? The first `timestamp` is when the event actually occurred on our host and the second `@timestamp` is when Logstash first processed the event. We clearly want to ensure we use the first `timestamp` to ensure we know when events occurred on our hosts.

We can, however, reconcile this difference using another filter plugin called `date`. Let's add it to our configuration after the `grok` filter.

Listing 5.38: The date filter

```
if [type] == "postfix" {
  grok {
    patterns_dir => ["/etc/logstash/patterns"]
    match => [ "message", "%{POSTFIX}" ]
    add_tag => [ "postfix", "grokked"]
  }
  date {
    match => [ "timestamp", "MMM dd HH:mm:ss", "MMM  d HH:mm:ss"
 ]
    add_tag => [ "dated" ]
  }
}
```

We can see our new `date` filter. We've specified the `match` option with the name of the field from which we want to create our time stamp: the `timestamp` field we created in the `grok` filter. To allow Logstash to parse this timestamp we're also specifying the date format of the field. In our case we've matched against two date formats: `MMM dd HH:mm:ss` and `MMM  d HH:mm:ss`. These two formats cover the standard Syslog log format and will match our incoming data, `Dec 24 17:01:03`. The date matching uses Java's Joda-Time library and you can see the full list of possible values here.

When the `date` filter runs it will replace the contents of the existing `@timestamp` field with the contents of the `timestamp` field we've extracted from our event.

NOTE You can see a full list of the `date` filter's options here.

We're also adding a tag `dated` to the event. You'll note we keep adding tags to events as they are filtered. I find this a convenient way to track what filtering or changes have occurred to my event. I can then tell at a glance which events have been changed and what has been done to them.

After performing this filtering, we can see that the timestamps on our events are now in sync and correct.

Listing 5.39: Postfix event timestamps

```
{
  . . .
  "timestamp" => "Dec 24 17:01:03",
  "@timestamp" =>"2012-12-24T17:01:03.000Z",
  . . .
}
```

Before we move on let's visually examine what Logstash's workflow is for our Postfix events:

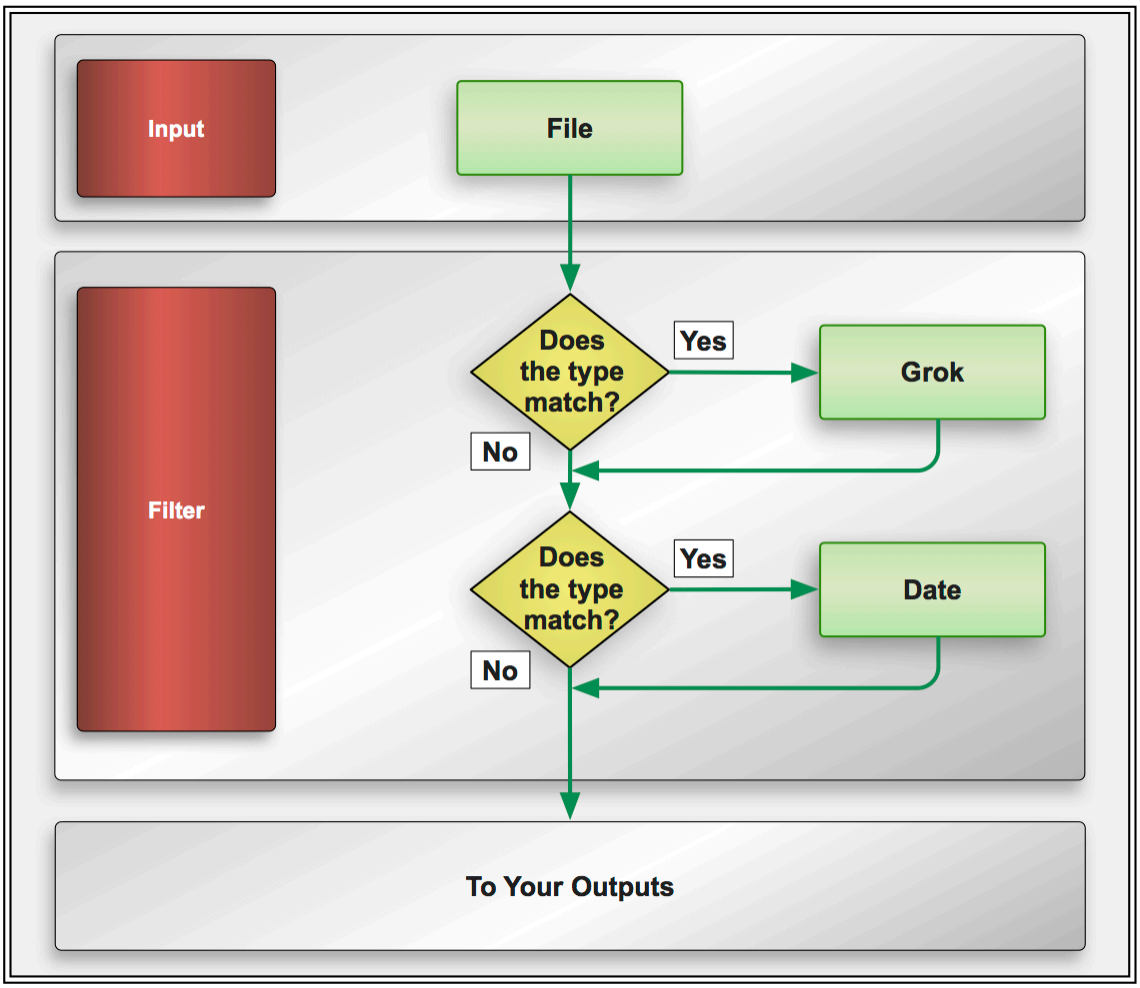

Figure 5.3: Postfix log filtering workflow

With this final piece our Postfix logs are now largely under control and we can move onto our final log source.

Filtering Java application logs

We've got one last data source we need to look at in this chapter: our Java application logs. We're going to start with our Tomcat servers. Let's start with inputting

our Tomcat events which we're going to do via a Filebeat prospector. Let's add a prospector to our `/etc/filebeat/filebeat.yml` configuration file.

Listing 5.40: Prospector for Tomcat logs

```
filebeat.prospectors:

- input_type: log
  document_type: tomcat
  paths:
    - /var/log/tomcat6/catalina.out

. . .

output.logstash:
  hosts: ["10.0.0.1:5044"]
```

Using this prospector we're collecting all the events from the `/var/log/tomcat6/catalina.out` log file. Let's look at some of the events available.

Listing 5.41: A Tomcat log entry

```
Dec 27, 2012 3:51:41 AM jenkins.InitReactorRunner$1 onAttained
INFO: Completed initialization,
```

These look like fairly typical log entries that we'll be able to parse and make use of but looking into the log file we also find that we've got a number of stack traces and a number of blank lines too. The stack traces are multi-line events that we're going to need to parse into one event. We're also going to want to get rid of those blank lines rather than have them create blank events in Logstash. So it looks like we're going to need to do some filtering.

Handling blank lines with drop

First we're going to use a new filter called `drop` to get rid of our blank lines. The `drop` filter drops events when a specific regular expression match is made. Let's look at a `drop` filter in combination with Logstash's conditional configuration syntax for removing blank lines:

NOTE In previous Logstash releases we'd have used the `grep` filter to perform this same action. This filter is now community managed and does not ship with Logstash.

Listing 5.42: A drop filter for blank lines

```
if [type] == "tomcat" and [message] !~ /(.+)/ {
  drop { }
}
```

Here we're matching events with a type of `tomcat` to ensure we parse the right events. We're also using a regular expression match on the message field. For this match we're ensuring that the `message` field isn't empty. So what happens to incoming events?

- If the event does not match, i.e. the `message` field *is not* empty, then the event is ignored.
- If the event does match, i.e. the `message` field *is* empty then the event is passed to the `drop` filter and dropped.

The conditional syntax is very simple and useful for controlling the flow of events and selecting plugins to be used for selected events. It allows for the typical

conditional if/else if/else statements, for example:

Listing 5.43: Examples of the conditional syntax

```
if [type] == "apache" {
   grok {
     . . .
   }
} else if [type] != "tomcat" {
   grok {
     . . .
   }
} else {
   drop { }
}
```

Each conditional expression supports a wide variety of operators, here we've used the equal and not equal (`==` and `!=`) operators, but also supported are regular expressions and `in` inclusions.

Listing 5.44: Conditional inclusion syntax

```
if "security" in [tags] {
   grok {
     . . .
   }
}
```

Here we've looked inside the `tags` array for the element `security` and passed the event to the `grok` plugin if it's found.

And as we've already seen conditional expressions allow `and` statements as well as `or`, `xand` and `xor` statements.

Finally we can group conditionals by using parentheses and nest them to create

conditional hierarchies.

TIP We'll see conditional syntax a few more times in the next couple of chapters as we filter and output events. You can find full details of their operations here.

Handling multi-line log events

Next in our logs we can see a number of Java exception stack traces. These are multi-line events but currently Logstash is parsing each line as a separate event. That makes it really hard to identify which line belongs to which exception and make use of the log data to debug our issues. Thankfully Logstash has considered this problem and we have a way we can combine the disparate events into a single event.

To do this we're going to build some simple regular expression patterns combined with a special codec called `multiline`. Codecs are used inside other plugins to handle specific formats or codecs, for example the JSON event format Logstash itself uses is a codec. Codecs allow us to separate transports, like Syslog, from the serialization of our events. Let's look at an example for matching our Java exceptions as raised through Tomcat.

Listing 5.45: Prospector for Tomcat logs

```
filebeat.prospectors:

- input_type: log
  document_type: tomcat
  multiline.pattern: "(^\d+\serror)|(^.+Exception: .+)|(^\s+at
.+)|(^\s+... \d+ more)|(^\s*Caused by:.+)"
  multiline.match: after
  tags: ["tomcat", "multiline"]
  paths:
    - /var/log/tomcat6/catalina.out

. . .

output.logstash:
  hosts: ["10.0.0.1:5044"]
```

With addition to the Tomcat prospector we're combing multiline events. The `multiline.pattern` option provides a regular expression for matching events that contain stack trace lines. There are a few variations on what these lines look like so you'll note we're using the `|` (which indicates `OR`) symbol to separate multiple regular expressions. For each incoming event Filebeat will try to match the `message` line with one of these regular expressions.

If the line matches any one of the regular expressions, Filebeat will then merge this event with either the previous or next event. In the case of our stack traces we know we want to merge the event with the event prior to it. We configure this merge by setting the `multiline.match` option to `after`.

Let's see an example of the multiline capability in action. Here are two events that are part of a larger stack trace. This event:

Listing 5.46: A Java exception

```
1) Error injecting constructor, java.lang.NoClassDefFoundError:
hudson/plugins/git/browser/GitRepositoryBrowser at hudson.
plugins.backlog.BacklogGitRepositoryBrowser$DescriptorImpl.<init
>(BacklogGitRepositoryBrowser.java:104)
```

Followed by this event:

Listing 5.47: Another Java exception

```
1 error
    at com.google.inject.internal.
ProviderToInternalFactoryAdapter.get(
ProviderToInternalFactoryAdapter.java:52)
...
```

When these events are processed by Filebeat's multiline pattern they will match one of the regular expression patterns and be merged. The resulting event will have a `message` field much like:

Listing 5.48: A multiline merged event

```
message => "Error injecting constructor, java.lang.
NoClassDefFoundError: hudson/plugins/git/browser/
GitRepositoryBrowser at hudson.plugins.backlog.
BacklogGitRepositoryBrowser$DescriptorImpl.<init>(
BacklogGitRepositoryBrowser.java:104)\n1 error at com.google.
inject.internal.ProviderToInternalFactoryAdapter.get(
ProviderToInternalFactoryAdapter.java:52). . ."
```

Further events that appear to be part of the same trace will continue to be merged

into this event.

Grokking our Java events

Now we've cleaned up our Tomcat log output we can see what useful data we can get out of it. Let's look at our Java exception stack traces and see if we can extract some more useful information out of them using `grok`.

Handily there's a built-in set of patterns for Java events so let's build a `grok` filter that uses them:

Listing 5.49: A grok filter for Java exception events

```
if [type] == "tomcat" and "multiline" in [tags] {
  grok {
    match => [ "message", "%{JAVASTACKTRACEPART}" ]
  }
}
```

Our new `grok` filter will be executed for any events with a type of `tomcat` and with the tag of `multiline`. In our filter we've specified the built-in pattern `JAVASTACKTRACEPART` which tries to match classes, methods, file name and line numbers in Java stack traces.

Let's see what happens when we run the stack trace we just merged through the `grok` filter. Our `message` field is:

Listing 5.50: Our Java exception message

```
message => "Error injecting constructor, java.lang.
NoClassDefFoundError: hudson/plugins/git/browser/
GitRepositoryBrowser at hudson.plugins.backlog.
BacklogGitRepositoryBrowser$DescriptorImpl.<init>(
BacklogGitRepositoryBrowser.java:104)\n1 error at com.google.
inject.internal.ProviderToInternalFactoryAdapter.get(
ProviderToInternalFactoryAdapter.java:52). . ."
```

Adding our `grok` filter we get the following fields:

Listing 5.51: Grokked Java exception

```
{
  . . .
  "class"=> "com.google.inject.internal.
ProviderToInternalFactoryAdapter",
  "method"=> "get",
  "file"=> "ProviderToInternalFactoryAdapter.java",
  "line"=> "52",
  . . .
}
```

Let's look at our final Logstash filtering workflow for our Tomcat log events:

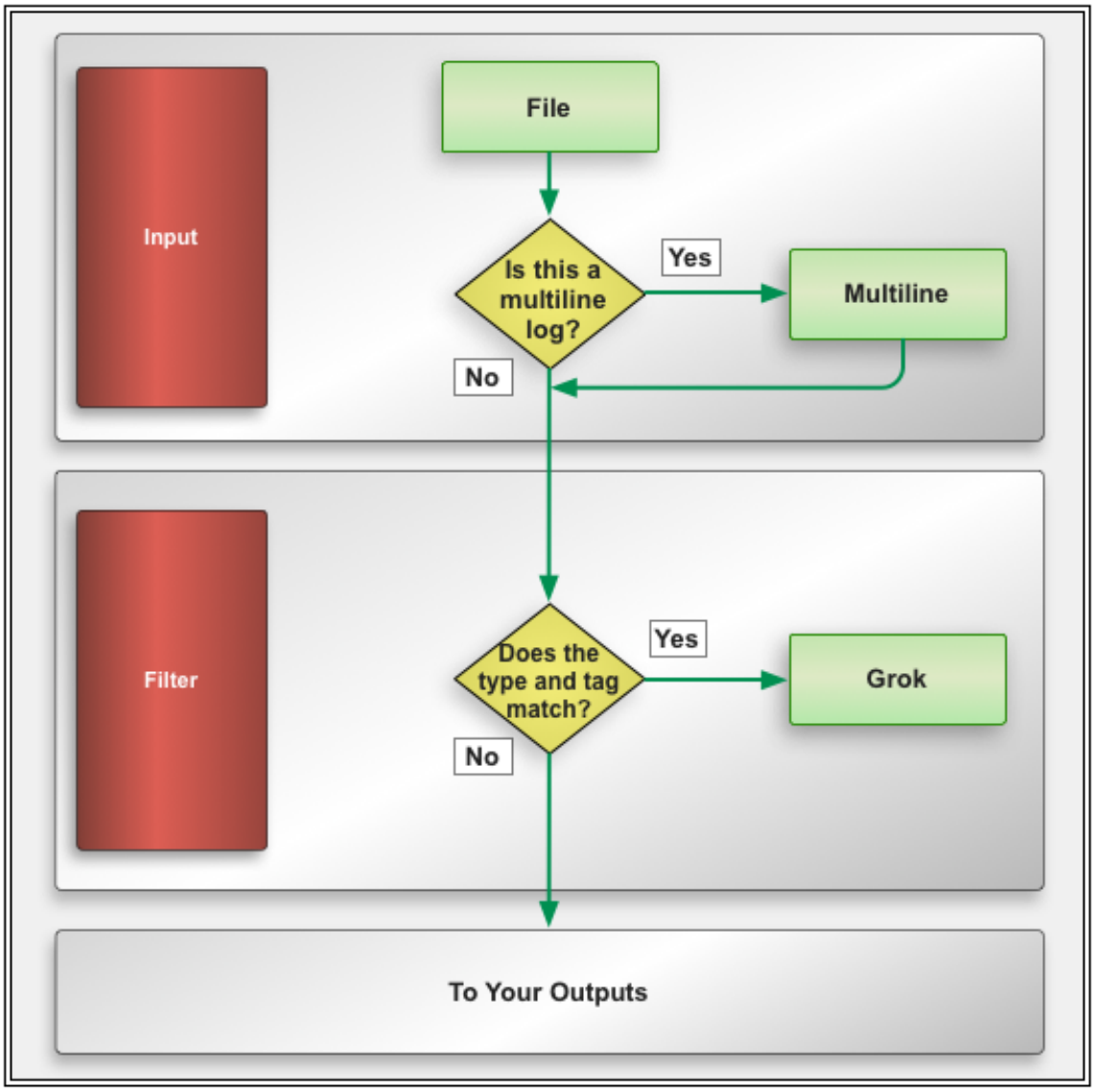

Figure 5.4: Tomcat log event workflow

We can see that we've added some useful fields with which to search or identify specific problem pieces of code. The combination of our stack trace events, this data and the ability centrally review all Tomcat logs will make it much easier for the teams that manage these applications to troubleshoot problems.

TIP All the filters in Logstash currently executes as a ‘worker‘ model. Each worker receives an event and applies all filters, in order, before sending that event to the output plugins. If you are doing intensive filtering or discover that filtering is a bottleneck in your environment you can add additional workers by starting Logstash with the ‘-w‘ flag. You can specify the number of workers you wish to run, for example for 5 workers specify ‘-w 5‘.

Parsing an in-house custom log format

All of the log entries we've seen up until now have been fairly standard or at least from applications that are commonly used: Apache, Postfix and Java. What happens if you have a custom application with a log format that is unusual or esoteric?

We're going to build a Grok filter for an in-house application called Alpha that is managed by your internal application support team. Alpha is used by the Finance team at Example.com and its log format does not match anything you've seen before. Let's look at an Alpha log entry:

Listing 5.52: Alpha log entry

```
1388290083+0200 The Alpha server has terminated /opt/alpha/
server/start.vb#134 ALF13-36B AlphaApp/2.4.5a/QA Release
1388290083+0200 The Alpha server has started /opt/alpha/server/
start.vb#156 ALF13-3AA AlphaApp/2.4.5a/QA Release
1388290084+0200 Alpha logging has initiated /opt/alpha/logging/
log.vb#14 ALF02-11F AlphaApp/2.4.5a/QA Release
```

You don't know much about the application but you can extrapolate a bit from the

log entries you can see. Firstly, you've got a timestamp. It appears to be seconds since epoch also known as Unix time with what looks like a time zone suffixed to it. We've also got a series of log messages, what looks to be the file and line that generated the message, a log entry ID and some application identification data.

The application support team tell you that in order to troubleshoot Alpha they need:

- The timestamp.
- The log message.
- The ID of the message.
- The file and line number that generated the error.
- The name of the application.
- The version of the application.
- The release of the application.
- They also want to have a field called environment created and set to `QA` if the application is a QA release.

So we know we need to design a Grok filter that will extract this information from our log entries and potentially some other filters to manipulate this data further.

So firstly we're going to collect our Alpha log entries. We're going to use our `maurice.example.com` host which runs Ubuntu and Filebeat so we can just add a new file prospector to Filebeat.

Let's add a prospector to our `/etc/filebeat/filebeat.yml` configuration file.

Listing 5.53: Prospector for Alpha logs

```
filebeat.prospectors:

- input_type: log
  tags: ["alpha", "finance"]
  document_type: alpha

  paths:
    - /opt/alpha/logs/alpha.log

. . .

output.logstash:
  hosts: ["10.0.0.1:5044"]
```

Using this prospector we're collecting all the events from the `/opt/alpha/logs/alpha.log` log file. Let's look at some of the events available.

Here we're grabbing entries from the `/opt/alpha/logs/alpha.log` log file. We're marking those entries with a type of `alpha` and tagging them with the tags `alpha` and `finance`. The tags will help us keep our log entries in order and make parsing decisions later on.

We know now we've got these logs that we need to add a `grok` filter to actually turn our log entry into a usable event. Let's look a single entry and start to construct a regular expression that will provide our application support team with the data they need.

Listing 5.54: Single Alpha log entry

```
1388290083+0200 The Alpha server has terminated /opt/alpha/
server/start.vb#134 ALF13-36B AlphaApp/2.4.5a/QA Release
```

To extract the data we need in our Grok filter we're going to use a mix of inbuilt patterns and the named capture capability. We saw named captures earlier in this chapter. They allow you to specify a field name and a regular expressions to extract that field from the log entry.

TIP I also strongly recommend making use of regular expression tools like Rubular and the incredibly useful Grok debugger to construct your Grok filters.

Let's look at a Grok filtering statement I've prepared for our Alpha log entry already.

Listing 5.55: A Grok regular expression for Alpha

```
(?<timestamp>[\d]+)\+(?<tz>[\w]{4})\s(?<msg>.*)\s%{UNIXPATH:file
}\#%{POSINT:line}\s%{GREEDYDATA:id}\s%{WORD:appname}\/(?<appver
>[\d.\d.\d\w]+)\/(?<apprelease>[\w\s]+)
```

I constructed this line by placing my sample log entry into the Grok debugger and then slowly constructing each field using named capture regular expressions or patterns as you can see here:

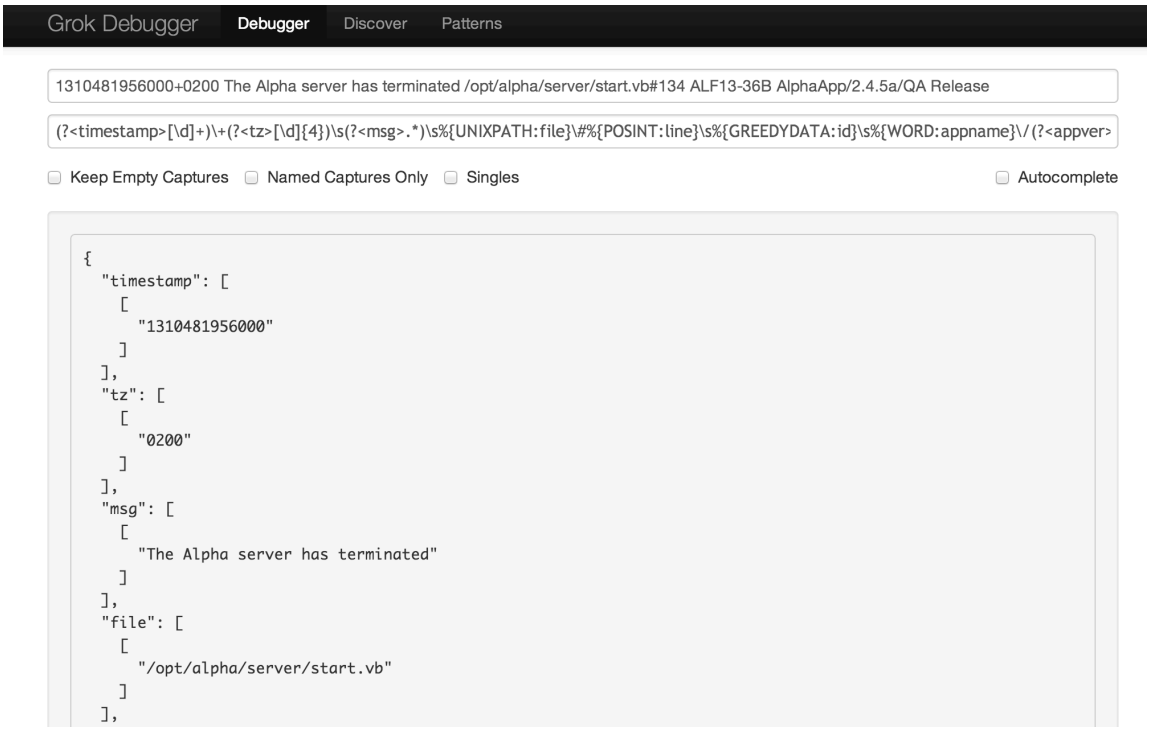

Figure 5.5: The Grok debugger at work

Shortly we'll be using this statement as the expression portion of the `match` option of a `grok` filter. In the expression we can see that we've worked through the Alpha log entry and we're extracting the following fields:

- timestamp - The Unix epoch timestamp
- tz - The timezone
- msg - The application log message
- file - The file that generated it
- line - The line of the file
- id - The log entry ID
- appname - The name of the application logging
- appver - The version of the application
- apprelease - The release of the application

Each field is generated using either an existing pattern or a named capture. For example the `appname` field is generated using the `WORD` pattern, `%{WORD:appname}`. Whilst the `appver` field is matched using a named capture: `(?<appver>[\d.\d.\d\w]+)`.

Now let's add a `grok` filter with our Alpha `match` to filter these incoming events:

Listing 5.56: Alpha grok filter

```
filter {
  if [type] == "alpha" {
    grok {
      match => [ "message", "(?<timestamp>[\d]+)\+(?<tz>[\w]{3})\
s(?<msg>.*)\s%{UNIXPATH:file}\#%{POSINT:line}\s%{GREEDYDATA:id}\
s%{WORD:appname}\/(?<appver>[\d.\d.\d\w]+)\/(?<apprelease>[\w\s
]+)" ]
      add_tag => [ "grokked" ]
    }
  }
}
```

We've added another `grok` filter to our `filter` block. We've first specified a conditional matching the `type` with a value of `alpha`. This will ensure our `grok` filter only matches on Alpha-related events. We've then specified the `grok` filter with the `match` option which matches a field of our log entry, here the default `message` field, with the expression we've just created.

But we're not quite finished yet. We know we've got a Unix epoch timestamp and we'd like to make sure our event's `@timestamp` uses the right time. So let's add a `date` filter to our filter block.

Listing 5.57: Alpha date filter

```
filter {
  if [type] == "alpha" {
    grok {
      . . .
    }
    date {
      match => [ "timestamp", "UNIX" ]
      timezone => tz
      add_tag => [ "dated" ]
    }
  }
}
```

Here we're specified `date` filter and told it to update the `@timestamp` field to the value from the `timestamp` field. We've specified `UNIX` to indicate the `timestamp` field is in Unix epoch time and we're also taking into consideration the timezone we've extracted from the log entry. We've also added the tag `dated` to our event to indicate we updated the `@timestamp`.

Next we also need to create our new `environment` field. This field will have a value of `qa` if the application is a QA release or `production` if not. We're going to use another conditional, this one nested, to achieve this.

Listing 5.58: Alpha environment field

```
filter {
  if [type] == "alpha" {
    grok {
      . . .
    }
    date {
      ...
    }
  }
  if [apprelease] == "QA Release" {
    mutate {
      add_field => [ "environment", "qa" ]
    }
  }
  else {
    mutate {
      add_field => [ "environment", "production" ]
    }
  }
}
```

You can see that we've nested another conditional inside our existing statement. We're testing to see if the `apprelease` field has a value of `QA Release`. If it does we're using a new filter called `mutate` that allows you to change the content of fields: convert their type, join/split fields, gsub field names amongst other capabilities. The `mutate` filter will add a new field called `environment` with a value of `qa`. If the `apprelease` field has any other value then the `environment` field will be set to `production`.

Finally, we've had some complaints from the application support team that the line number of the file that generated the error isn't an integer. This makes some of their debugging tools break. So we need to ensure that the `line` field has a type of integer. To do this we can again use the `mutate` filter.

Listing 5.59: Setting the line field to an integer

```
filter {
  if [type] == "alpha" {
    . . .
    mutate {
      convert => [ "line", "integer" ]
    }
  }
}
```

You can see that we've specified the `mutate` filter again and used the `convert` option to convert the `line` field into an integer.

Now when we run Logstash we should start to see our Alpha log events rendered in a format that our application support team can use. Let's look at a filtered Alpha log entry now.

Listing 5.60: A filtered Alpha event

```
{
  . . .
  @timestamp => "Sun, 29 Dec 2013 04:08:03",
  "tags" => [ "alpha", "grokked", "finance", "dated" ],
  "timestamp" => "1388290083",
  "tz" => "0200",
  "msg" => "The Alpha server has terminated",
  "file" => "/opt/alpha/server/start.vb",
  "line" => 134,
  "id" => "ALF13-36B",
  "appname" => "AlphaApp",
  "appver" => "2.4.5a",
  "apprelease" => "QA Release",
  "environment" => "qa",
  . . .
}
```

We can see that our entry contains the data our team needs and should now be searchable and easy for them to use to debug the Alpha application.

You can see that the `grok` filter combined with the huge variety of other available filters make this a simple and easy process. You can apply this workflow to any custom log event you need to parse.

Summary

In this chapter we've seen some of the power of Logstash's filtering capabilities. But what we've seen in this chapter is just a small selection of what it is possible to achieve with Logstash. There's a large collection of additional filter plugins available. Filters that allow you to:

TIP In addition to the plugins that ship with Logstash there are also a number of community contributed plugins available here.

- Mutate events. The mutate filter allows you to do general mutations to fields. You can rename, remove, replace, and modify fields in your events.
- Checksum events. This checksum filter allows you to create a checksum based on a part or parts of the event. You can use this to de-duplicate events or add a unique event identifier.
- Extract key value pairs. This lets you automatically parse log events that contain key value structures like `foo=bar`. It will create a field with the key as the field name and the value as the field value.
- Do GeoIP and DNS lookups. This allows you to add geographical or DNS metadata to events. This can be helpful in adding context to events or in processes like fraud detection using log data.
- Calculate ranges. This filter is used to check that certain fields are within expected size or length ranges. This is useful for finding anomalous data.
- Extract XML. This filter extracts XML from events and constructs an appropriate data structure from it.
- The split filter allows you to split multi-line messages into separate events.
- The anonymize filter is useful for anonymizing fields by replacing their values with a consistent hash. If you're dealing with sensitive data this is useful for purging information like user ids, SSNs or credit card numbers.
- Execute arbitrary Ruby code. This allows you to process events using snippets of Ruby code.

TIP One of the more annoying aspects of filter patterns is that it is time consuming to test your patterns and ensure they don't regress. We've already seen the the Grok Debugger but it's also possible to write RSpec tests for your filtering patterns

that can make development much simpler.

Now we've gotten a few more log sources into Logstash and our events are more carefully catalogued and filtered. In the next chapter we'll look at how to create structured log output from our applications.

Chapter 6

Structured Application Logging

In addition to filtering existing log events from our applications and services we can also add structured logging to our applications. With structured 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. There are two ways of deducing useful data from logs:

- Creating structured log entries at strategic points of our application.
- Consuming existing log data.

We're going to look at both methods, starting with adding our own log entries.

Application logging primer

Let's look at some basic tenets for application logging. Firstly, in any good application development methodology, it's a good idea to identify what you want to build before you build it. Logging is no different. Sadly there is a common anti-pattern in application development of considering logging and other operational

functions like security as value-add components of your application rather than core features. Logging, monitoring (and security!) are core functional features of your applications. So, if you're building a specification or user stories for your application, include logging for each component of your application. Not building good logging 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.

A second common anti-pattern is not logging 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 logging configuration provides tags or identifiers so you know that the log entry or event is from a specific environment. This way you can partition your logging. We'll talk more about this later in the chapter.

Where should I instrument?

Good places to start adding logging for your applications are at points of ingress and egress, for example:

- 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.
- 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.

- Log job scheduling, execution, and other periodic events like cron jobs.
- Log significant business and functional events, such as users being created or transactions like payments and sales.
- Log methods and functions that read and write from databases and caches.

Instrument schemas

You should ensure that events 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 is generated. You should develop a schema for your log events.

Time and the observer effect

It's also important to ensure 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. 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 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 logging overloading an application. Also relevant here is the observer effect. If your monitoring consumes considerable CPU cycles or memory then it could impact the performance of your application or skew the results of any logging.

[1] Please don't invent your own timestamp format. Please.

Logging patterns, or where to put your logging

Once we know what we want to log, we need to work out where to put our logging. In almost all cases the best place to put this logging is inside our code and as close as possible to the action we're trying to monitor.

We don't, however, want to put our logging configuration inline everywhere. Instead we want to create a utility library: a function that allows us to create a variety of logs from a centralized setup. This is also sometimes called the utility pattern — 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 Lograge. The utility library would expose an API that allows us to create and log events. 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 `Logger`.

NOTE We'll see several functioning examples of this pattern later in this chapter.

Listing 6.1: A sample payments method

```
include Logger

def pay_user(user, amount)
  pay(user.account, amount)
  Logger.send "payment.amount: #{amount.to_i}; payment.country,
#{user.country}"
  send_payment_notification(user.email)
end

def send_payment_notification(email)
  send_email(payment, email)
  Logger.send "email.payment"
end
```

Here we've first included our `Logger` utility library. 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 logged an event from it. Finally, we've sent an email using a second method, `send_payment_notification`, where we've created another log event: `email.payment`.

NOTE We could also support emitting metrics here instead of log events. If you are interested in monitoring and metrics more broadly, you might be interested in another book of mine: The Art of Monitoring.

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 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 Elasticsearch to be indexed. Often you can track the frequency of events by simply recording the counts of specific log entries.

NOTE We'll see more about the `metrics` plugin in Chapter 7.

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 state and status of your application.

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 and which we explored in Chapter 4. A typical Syslog message looks like:

Listing 6.2: A typical syslog message

```
Dec  6 23:17:01 maurice 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 datestamp and a source (the host `maurice`). 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 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. 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 6.3: 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 6.4: 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 6.5: 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 a sample application

Let's see how we might extend a sample application with some structured log events. We're going to add structured logging to a demo 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 6.6: Adding our logging gems to the ls-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 6.7: 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 6.8: 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, `smoker.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 6.9: Adding a new TCP input to Logstash

```
input {
  tcp {
    port  => 2020
    type  => "apps"
    codec => "json"
  }
. . .
```

We've added a new input: `tcp`. The `tcp` input runs a TCP server on the Logstash server that can receive events from external sources. Our TCP server is 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 6.10: 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 6.11: 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 6.12: 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":"application1",
  "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 via Kibana.

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 6.13: 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 6.14: A Logstash formatted event for a user deletion

```
{
  "message":"user_deleted",
  "user": {
    "id":6,
    "email":"james@example.com",
    "created_at":"2016-11-05T04:31:46.828Z",
    "updated_at":"2016-11-05T04:32:18.340Z",
    "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":"2016-11-05T13:35:50.070+00:00",
  "@version":"1",
  "severity":"INFO",
  "host":"application1",
  "type":"apps"
}
```

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. 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, pne 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 5, 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 6.15: 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 6.16: 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 6.17: 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":"application1",
  "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 6.18: 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 the `patterns_dir` option in our filter. We saw this option in Chapter 5. 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, in case we haven't got it already.

Listing 6.19: 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. We'll create it in a file called `app`.

Listing 6.20: 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 6.21: 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 6.22: 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:

- Graph transaction amounts in Kibana.
- Send failed transactions to the appropriate users. 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.
- Use the event data for audit and diagnostic purposes.

Or we could perform a wide variety of other processing actions.

Summary

In this chapter we've seen how to add logging and structured logs to our applications. We've learned about how and where to position our logging and some architecture patterns we can adopt.

We've also seen some examples of how to add logging as a utility to our code base and how to parse existing applications logs.

In the next chapter we are going to look at how to get information, alerts and metrics out of Logstash.

Chapter 7

Outputting Events from Logstash

In the previous chapters we've seen some of the output plugins available in Logstash: for example Syslog and ElasticSearch. But in our project we've primarily focussed on moving events from agents to our central server and from our central server to ElasticSearch. Now, at this stage of the project, we want to start using some of the other available output plugins to send events or generate actions from events. We've identified a list of the top outputs we need to create:

- Send alerts for events via email.
- Send alerts for events via instant messaging.
- Send alerts through to a monitoring system.
- Collect and deliver metrics through a metrics engine.

Let's get started with developing our first output.

Send email alerts

The first needed output we've identified is alerts via email. Some parts of the IT team really want to get email notifications for certain events. Specifically they'd

like to get email notifications for any stack traces generated by Tomcat. To do this we'll need to configure the `email` output plugin and provide some way of identifying the stack traces we'd like to email.

Updating our multiline filter

Since we've just tackled this log source in Chapter 5 we're going to extend what we've already done to provide this capability. Let's first look at our existing `multiline` configuration in Filebeat.

Listing 7.1: Prospector for Tomcat logs

```
filebeat.prospectors:

- input_type: log
  document_type: tomcat
  multiline.pattern: "(^\d+\serror)|(^.+Exception: .+)|(^\s+at
.+)|(^\s+... \d+ more)|(^\s*Caused by:.+)"
  multiline.match: after
  tags: ["tomcat", "multiline"]
  paths:
    - /var/log/tomcat6/catalina.out
```

This prospector will match any `message` lines with the `multiline.pattern` specified and merge them into one event. It'll also add the tags `multiline` and `tomcat` to the event.

Configuring the email output

Next we need to configure our `email` plugin in the `output` block in our `central.conf`.

Listing 7.2: The email output plugin

```
if [type] == "tomcat" and "multiline" in [tags] {
  email {
    body => "Triggered in: %{message}"
    subject => "This is a Logstash alert for Tomcat stack traces.
"
    from => "logstash.alert@example.com"
    to => "appteam@example.com"
    via => "sendmail"
  }
}
```

Our `email` output plugin is configured to only match events with the `type` of `tomcat` and with the tag `multiline`. This way we don't flood our mail servers with every event by mistake.

NOTE You can see this and a full list of the `email` outputs options here.

We then specify the body of the email in plain text using the `body` option. We're sending the message:

Listing 7.3: The content of our email

```
"Triggered in: %{message}"
```

The body of the email will contain the specific stack trace which is contained in the `message` field. The `email` output also has support for HTML output which you can specify using the `htmlbody` option.

NOTE We've referred to the `message` field via Logstash's `sprintf` format. We've prefixed it with a percentage sign and enclosed the field in braces. You can see more details here.

We've also specified the subject of the email using the `subject` option.

We next specify the `from` and `to` options that set the emission and destination email addresses. And lastly we set the `via` option which controls how the email is sent: either `sendmail` or `smtp`. In our case we're using `sendmail` which directly calls the MTA locally on the host. If needed, you can also control a variety of other email options including SSL/TLS and authentication using the `options` directive.

Email output

Now every time Logstash receives a Java exception stack trace the `email` output will be triggered and the stack trace will be emailed to the `appteam@example.com` address for their attention.

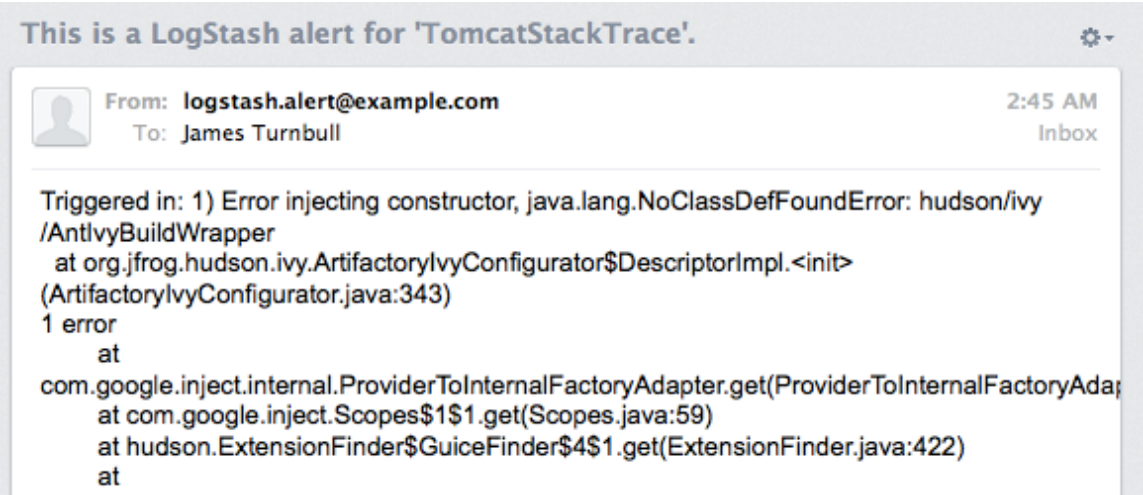

Figure 7.1: Java exception email alert

WARNING Please be aware that if you get a lot of stack traces this could quickly become an unintentional email-based Denial of Service attack.

Send instant messages

Our next output is similar to our email alert. Some of your colleagues in the Security team want more immediate alerting of events and would like Logstash to send instant messages when failed SSH logins occur for sensitive hosts. Thanks to the work we did earlier in the project, documented in Chapter 3, we're already collecting the syslog events from `/var/log/secure` on our sensitive hosts using a Filebeat prospector.

Identifying the event to send

As we've already got the required event source now all we need to do is identify the specific event on which the Security team wants to be alerted:

Listing 7.4: Failed SSH authentication log entry

```
Dec 28 21:20:27 maurice sshd[32348]: Failed password for bob
from 184.75.0.187 port 32389 ssh2
```

We can see it is a standard Syslog message. Our Security team wants to know the user name and the source host name or IP address of the failed login. To acquire this information we're going to use a `grok` filter in our `central.conf`:

Listing 7.5: Failed SSH authentication grok filter

```
if [type] == "syslog" {
  grok {
    match => [ "message", "%{SYSLOGBASE} Failed password for %{
USERNAME:user} from %{IPORHOST:shost} port %{POSINT:port} %{WORD:
protocol}" ]
    add_tag => [ "ssh", "grokked", "auth_failure" ]
  }
}
```

Which, when it matches the Syslog log entry, should produce an event like this:

Listing 7.6: Failed SSH authentication Logstash event

```
{
  "message" => "Dec 28 21:20:27 maurice sshd[32348]: Failed
password for bob from 184.75.0.187 port 32389 ssh2",
  "@timestamp" => "2012-12-28T21:20:27.016Z",
  "@version" => "1",
  "host" => "maurice.example.com",
  "timestamp" => "Dec 28 21:20:27",
  "logsource" => "maurice.example.com",
  "program" => "sshd",
  "pid" => "32348",
  "user" => "bob",
  "shost" => "184.75.0.187",
  "port" => "32389",
  "protocol" => "ssh2",
  "tags" => [
     [0] "ssh",
     [1] "grokked",
     [2] "auth_failure"
  ]
}
```

You can see that our `grok` filter has matched the event using the specified `pattern` and populated the fields: `timestamp`, `logsource`, `program`, `pid`, `port`, `protocol` and most importantly `user` and `shost` (the source host). The event has also been tagged with the `ssh`, `grokked` and `auth_failure` tags.

Sending the instant message

We now have a tagged event with the data our Security team needs. How do we get it to them? To do this we're going to use a new output plugin in our `central.conf` called `xmpp` that sends alert notifications to a Jabber/XMPP user.

Listing 7.7: The xmpp output plugin

```
if "auth_failure" in [tags] and [type] == "syslog" {
  xmpp {
    message => "Failed login for user %{user} from %{shost} on
server %{logsource}"
    user => "alerts@jabber.example.com"
    password => "password"
    users => "security@example.com"
  }
}
```

The `xmpp` output is simple to configure. First, to ensure only the right events are alerted, we've specified that the output only triggers on events tagged with `auth_failure` and with a type of `syslog`. Next, we've defined a `message` that contains the data our security team wants by referencing the fields we created in our `grok` filter earlier. Lastly, we've specified the connection details: `user`, `password` and an array of `users` to be alerted about these events.

WARNING Here we're using an internal XMPP network inside our organi-

zation. Remember, if you are using a public XMPP network, to be careful about sending sensitive data across that network.

Now when a failed SSH login occurs and Logstash matches the appropriate event an instant message will be generated:

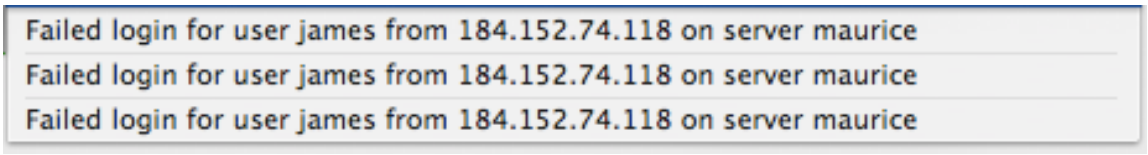

Figure 7.2: Jabber/XMPP alerts

NOTE You can see this and a full list of the `xmpp` output's options here.

Send alerts to Nagios

Our previous two outputs have been alerts and very much point solutions. Our next output is an integration with an external framework, in this case with the monitoring tool Nagios. Specifically we're going to generate what Nagios calls "passive checks" from our log events and send them to a Nagios server.

Nagios check types

There are two commonly used types of Nagios checks: active and passive. In an active check Nagios initiates the check from a Nagios server using a plugin like `check_icmp` or `check_http`. Alternatively, passive checks are initiated outside

Nagios and the results sent to a Nagios server. Passive checks are usually used for services that are:

- Asynchronous in nature and cannot be monitored effectively by polling their status on a regularly scheduled basis.
- Located behind a firewall and cannot be checked actively from the Nagios server.

Identifying the trigger event

We're going to generate some of these Nagios passive checks using a new output plugin called `nagios`.

Let's look at a log event that we'd like to trigger a Nagios passive service check: a STONITH cluster fencing log event.

Listing 7.8: A STONITH cluster fencing log event

```
Dec 18 20:24:53 clunode1 clufence[7397]: <notice> STONITH:
clunode2 has been fenced!
```

Assuming we've got an input plugin that picks up this event, we start by identifying and parsing this specific event via a `grok` filter.

Listing 7.9: Identify Nagios passive check results

```
if [type] == "syslog" {
  grok {
    match => [ "message", "%{SYSLOGBASE} <notice> STONITH: %{
IPORHOST:cluster_node} has been fenced!" ]
    add_tag => [ "nagios_check" ]
    add_field => [
      "nagios_host", "%{cluster_node}",
      "nagios_service", "cluster"
    ]
  }
}
```

We're searching for events with a type of `syslog` and with a pattern match to our STONITH cluster fence event. If the event matches we're adding a tag called `nagios_check` and we're adding two fields, `nagios_host` and `nagios_service`. This will tell the `nagios` output the hostname and service on which it should alert.

Parsing our example log entry will result in event tags and fields that look like:

Listing 7.10: The grokked STONITH event

```
{
  "message" => "Dec 18 20:24:53 clunode1 clufence[7397]: <notice>
 STONITH: clunode2 has been fenced!",
  "@timestamp" => "2013-12-18T20:24:53.965Z",
  "@version" => "1",
  "host" => "clunode1",
  "timestamp" => "Dec 18 20:24:53",
  "logsource" => "clunode1",
  "program" => "clufence",
  "pid" => "7397",
  "cluster_node" => "clunode2",
  "nagios_host" => "clunode2",
  "nagios_service" => "cluster",
  "tags" => [
     [0] "nagios_check",
  ]
}
```

The nagios output

To output this event as a Nagios passive check we specify the `nagios` output plugin.

Listing 7.11: The Nagios output

```
if "nagios_check" in [tags] {
  nagios { }
}
```

Nagios can receive passive checks in several ways. The `nagios` output plugin takes advantage of Nagios' external command file. The external command file is a named pipe from which Nagios listens periodically for incoming commands.

The `nagios` output generates PROCESS_SERVICE_CHECK_RESULT commands and submits them to this file.

NOTE For external commands to be processed you must have the `check_external_commands=1` option set in your Nagios server configuration.

The `nagios` output checks events for the tag `nagios_check` and if it exists then submits a `PROCESS_SERVICE_CHECK_RESULT` command to the Nagios external command file containing details of the event. It's important to remember that the user running Logstash must be able to write to the Nagios command file. The output assumes the external command file is located at `/var/lib/nagios3/rw/nagios.cmd` but this can be overridden with the `commandfile` option:

Listing 7.12: The Nagios output with a custom command file

```
nagios {
  tags => "nagios_check"
  commandfile => "/var/run/nagios/rw/nagios.cmd"
}
```

TIP If your Nagios server is not located on the same host you can make use of the `nagios_nsca` output which provides passive check submission to Nagios via NSCA.

The Nagios external command

Let's look at the command generated by Logstash.

Listing 7.13: A Nagios external command

```
[1357065381] EXTERNAL COMMAND: PROCESS_SERVICE_CHECK_RESULT;
clunode2;cluster;2;file://maurice.example.com/var/log/rhcluster/
stonith.log: Jul 18 20:24:53 clunode1 clufence[7397]: <notice>
STONITH: clunode2 has been fenced!
```

We can see the host and service name we specified in the `nagios_host` and `nagios_service` fields, `clunode2` and `cluster` respectively. We can also see the Nagios return code, `2`, which indicates this is a `CRITICAL` event. By default the `nagios` output sends passive check results with a status of `CRITICAL`. You can override this in two ways:

- Set a field on the event called `nagios_level` with a value of the desired state: `OK`, `WARNING`, `CRITICAL`, or `UNKNOWN`.
- Use the `nagios_level` option in the output to hardcode a status.

Setting the `nagios_level` field will override the `nagios_level` configuration option.

NOTE You can see this and a full list of the `nagios` outputs options here.

The Nagios service

On the Nagios side you will need a corresponding host and service defined for any incoming command, for example:

Listing 7.14: A Nagios service for cluster status

```
define service {
  use  local-service
  host_name  clunode2
  service_description  cluster
  active_checks_enabled  0
  passive_checks_enabled  1
  notifications_enabled  1
  check_freshness  0
  check_command  check_dummy
}
```

Now when a matching event is received by Logstash it will be sent as an external command to Nagios, then processed as a passive service check result and trigger the `cluster` service on the `clunode2` host. It's easy to extend this to other events related to specific hosts and services for which we wish to monitor and submit check results.

Outputting metrics

One of the key needs of your colleagues in both Operations and Application Development teams is the ability to visually represent data about your application and system status and performance. As a mechanism for identifying issues and understanding performance, graphs are a crucial tool in every IT organization. During your review of Logstash as a potential log management tool, you've discovered that one of the really cool capabilities of Logstash is its ability to collect and send metrics from events.

But there are lots of tools that do that right? Not really. There are lots of point solutions designed to pick up one, two or a handful of metrics from infrastructure and application specific logs and deliver them to tools like Graphite or through brokers like StatsD. Logstash instead allows you to centralize your metric collection from log events in one tool. If a metric exists in or can be extrapolated from a log event then you can deliver it to your metrics engine. So for your next output we're going to take advantage of this capability and use Logstash events to generate some useful metrics for your environment.

Logstash supports output to a wide variety of metrics engines and brokers including Ganglia, Riemann, Graphite, StatsD, MetricCatcher, and Librato, amongst others.

Collecting metrics

Let's take a look at how this works using some of the log events we're collecting already, specifically our Apache log events. Using the custom log format we created in Chapter 5 our Apache log servers are now logging events that look like:

Listing 7.15: JSON format event from Apache

```
{
  "host" => "host.example.com",
  "path" => "/var/log/httpd/logstash_access_log",
  "tags" => [ "wordpress", "www.example.com" ],
  "message" => "50.116.43.60 - - [22/Dec/2012:16:09:30 -0500] \"
GET / HTTP/1.1\" 200 4979",
  "timestamp" => "2012-12-22T16:09:30-0500",
  "clientip" => "50.116.43.60",
  "duration" => 11313,
  "status" => 200,
  "request" => "/index.html"
  "urlpath" => "/index.html",
  "urlquery" => "",
  "method" => "GET",
  "bytes" => 4979,
  "vhost" => "www"
  "@timestamp"=>"2012-12-22T16:09:30.658Z",
  "@version => "1",
  "type"=>"apache"
}
```

We can already see quite a few things we'd like to graph based on the data we've got available. Let's look at some potential metrics:

- An incremental counter for response status codes: 200, 404, etc.
- An incremental counter for method types: GET, POST, etc.
- A counter for the bytes served.
- A timer for the duration of each request.

StatsD

To create our metrics we're going to use the `statsd` output. StatsD is a tool written by the team at Etsy. You can read about why and some more details about how

StatsD works here. It acts as a front-end broker to Graphite and is most useful because you can create new metrics in Graphite just by sending it data for that metric. I'm not going to demonstrate how to set up StatsD or Graphite. There are a number of excellent guides, HOWTOs, Puppet modules and Chef cookbooks for that online.

NOTE If you don't want to use StatsD you can send metrics to Graphite directly using the graphite output.

Setting the date correctly

Firstly, getting the time accurate really matters for metrics so we're going to use the `date` filter we used in Chapter 5 to ensure our events have the right time. Using the `date` filter we will set the date and time our Apache events to the value of the `timestamp` field contained in each event:

Listing 7.16: The Apache event timestamp field

```
"timestamp": "2012-12-22T16:09:30-0500"
```

Let's add our `date` filter now:

Listing 7.17: Getting the date right for our metrics

```
if [type] == "apache" {
  date {
    match => [ "timestamp", "ISO8601" ]
    add_tag => [ "dated" ]
  }
}
```

Our `date` filter has a conditional wrapper that checks for a `type` of `apache` to ensure it only matches our Apache events. It then uses the `match` statement to specify that Logstash should look for an ISO8601 format in the field `timestamp`. This will ensure our event's timestamp will match the timestamp of the original Apache log event. We're also adding the tag `dated` to mark events which have had their timestamps set.

NOTE Remember date matching uses Java's Joda-Time library.

The StatsD output

Now we've got the time of our events correct we're going to use the `statsd` output to create the metrics we would like from our Apache logs:

Listing 7.18: The statsd output

```
if [type] == "apache" {
  statsd {
    increment => "apache.status.%{status}"
    increment => "apache.method.%{method}"
    count => [ "apache.bytes", "%{bytes}" ]
    timing => [ "apache.duration", "%{duration}" ]
  }
}
```

You can see we're only matching events with a `type` of `apache`. You could also match using tags, excluding tags or using fields. Next we've specified two incremental counters, a normal counter and a timer.

Our first two incremental counters are:

Listing 7.19: Incremental counters

```
increment => "apache.status.%{status}"
increment => "apache.method.%{method}"
```

They use the `increment` option and are based on two fields we've specified in our Apache log events: `status` and `method`, which track the Apache response status codes and the HTTP methods respectively. Our metrics are named with a prefix of `apache.` and make use of Graphite's namespaces, each `.` representing a folder in Graphite's views.

Each event will either create a new metric, if that `status` or `method` doesn't already have a metric, or increment an existing metric. The result will be a series of metrics matching each status:

Listing 7.20: Apache status metrics in Graphite

```
apache.status.200
apache.status.403
apache.status.404
apache.status.500
. . .
```

And each method:

Listing 7.21: Apache method metrics in Graphite

```
apache.method.GET
apache.method.POST
. . .
```

Each time an Apache log event is received by our Logstash central server it will trigger our output and increment the relevant counters. For example a request using the `GET` method with a `200` response code Logstash will send an update to StatsD for the `apache.method.GET` and `apache.status.200` metrics incrementing them by 1.

StatsD will then push the metrics and their data to Graphite and produce graphs that we can use to monitor our Apache web servers.

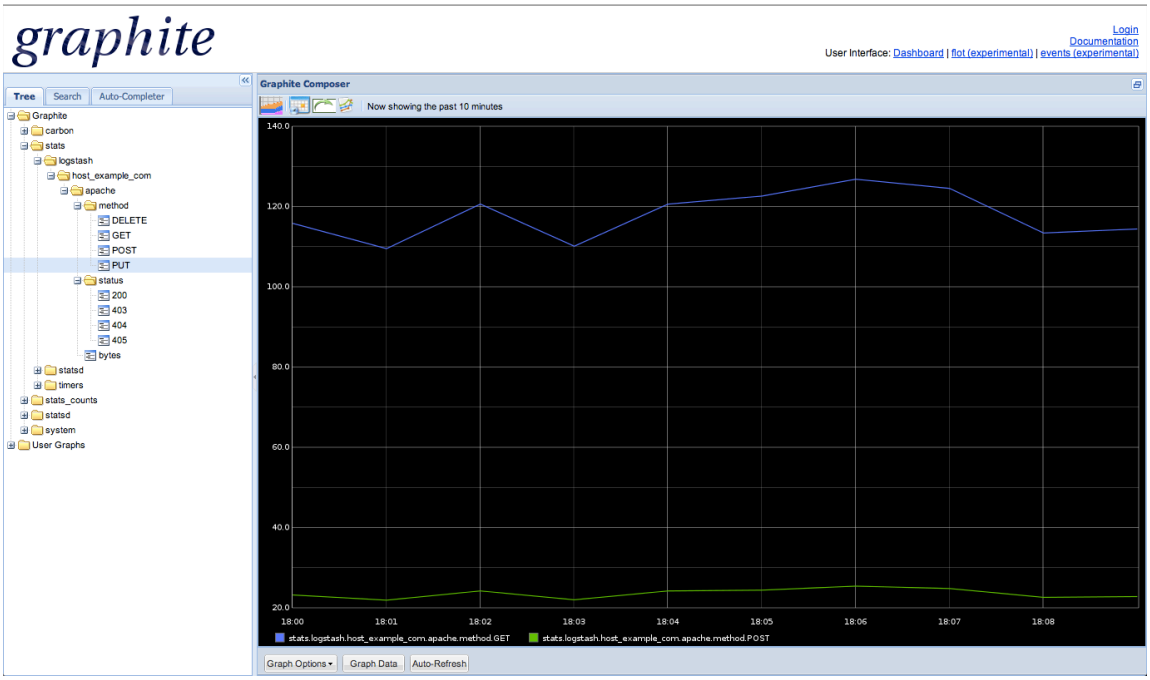

Figure 7.3: Apache status and method graphs

Here we can see our Apache method metrics contained in the Graphite namespace: stats -> logstash -> host_example_com -> apache -> method. The namespace used defaults to `logstash` but you can override this with the `namespace` option.

Our counter metric is similar:

Listing 7.22: The apache.bytes counter

```
count => [ "apache.bytes", "%{bytes}" ]
```

We're creating a metric using the `count` option called `apache.bytes` and when an event comes in we're incrementing that metric by the value of the `bytes` field in that event.

We can then see this graph presented in Graphite:

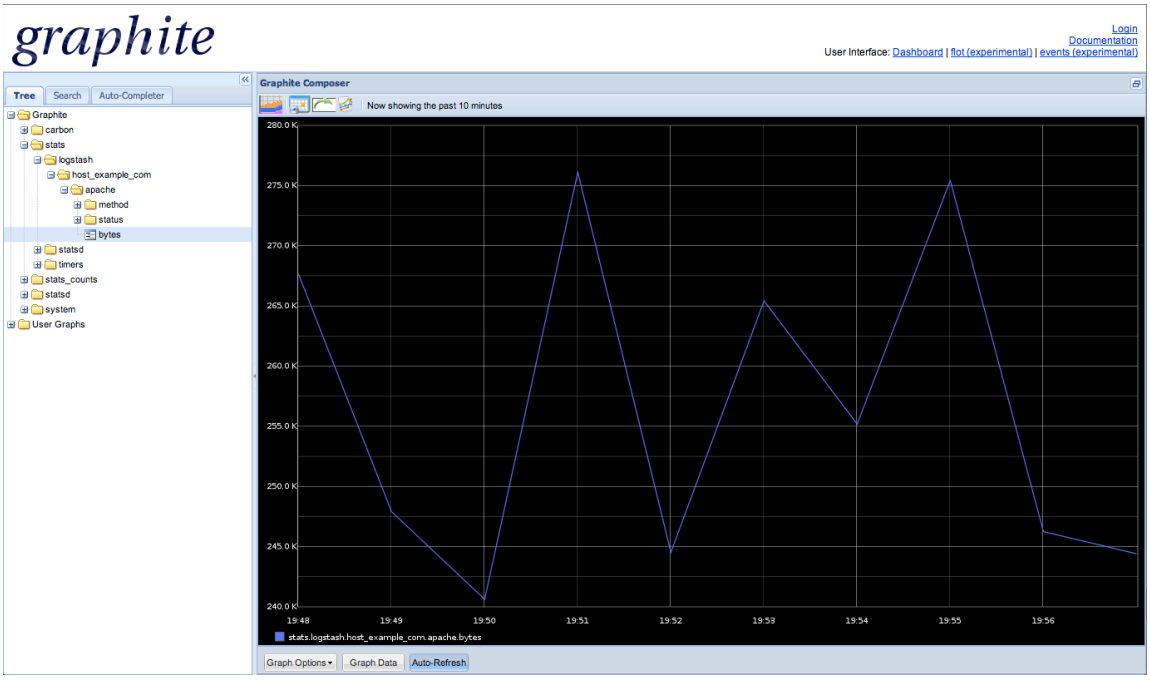

Figure 7.4: Apache bytes counter

The last metric creates a timer, using the `timing` option, based on the `duration` field of our Apache log event which tracks the duration of each request.

Listing 7.23: The apache.duration timer

```
timing => [ "apache.duration", "%{duration}" ]
```

We can also see this graph, together with the automatic creation of lower and upper bounds metrics, as well as mean and sum metrics:

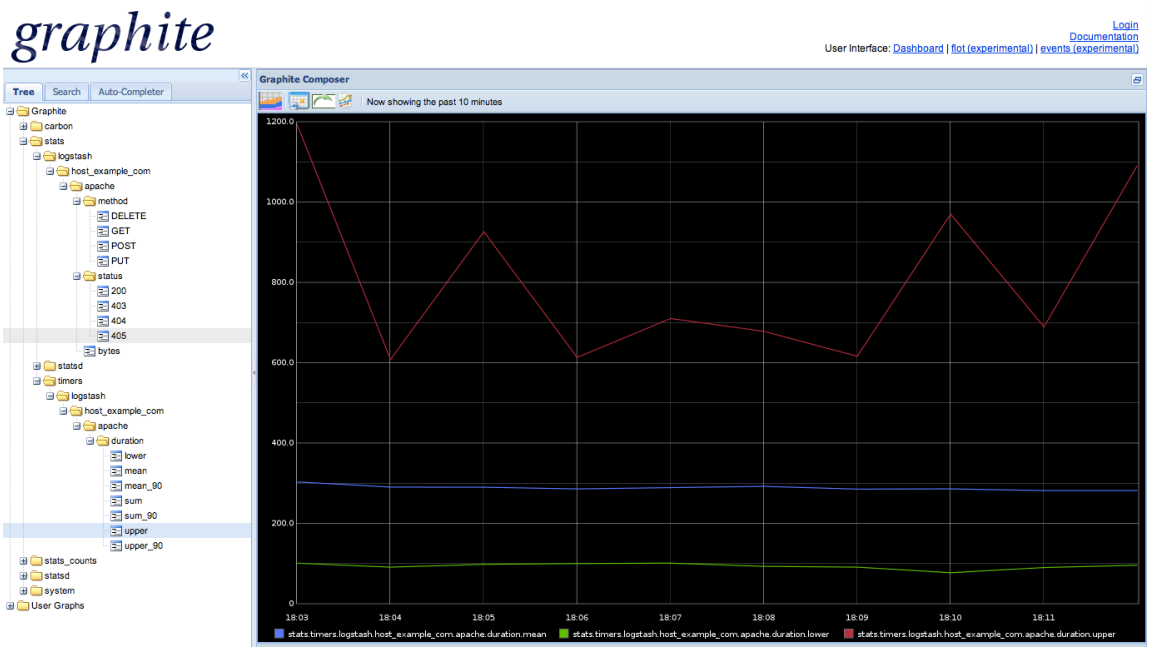

Figure 7.5: Apache request duration timer

Sending to a different StatsD server

By default, the `statsd` output sends results to the `localhost` on port `8125` which is the default port on which StatsD starts. You can override this using the `host` and `port` options.

Listing 7.24: The StatsD output with a custom host and port

```
if [type] == "apache" {
  statsd {
    host => "statsd.example.com"
    port => 8130
    . . .
  }
}
```

NOTE You can see this and a full list of the `statsd` output's options here.

Now we have a useful collection of basic graphs from our Apache events. From this we can add additional metrics from our Apache events or from other log sources.

NOTE Also available in Logstash 1.1.6 and later is the `metrics` filter which is a useful shortcut to creating metrics from events. For some purposes it should ultimately replace the approach described here for gathering and generating metrics.

Summary

We've now configured a small collection of initial outputs for our logging project that provide alerts, monitoring and metrics for our environment. It's easy to extend these outputs and add further outputs from the wide collection available.

With these outputs configured we've got events coming in, being filtered and outputted in a variety of ways. Indeed Logstash is becoming an important tool in our monitoring and management toolbox. As a result of the growing importance of Logstash we now need to consider how to ensure it stays up and scales to meet demand. In the next chapter we're going to learn how to grow our Logstash environment.

Chapter 8

Scaling Logstash

One of the great things about Logstash is that it is made up of easy to fit together components: Logstash itself, Elasticsearch and the various other pluggable elements of your Logstash configuration. One of the significant fringe benefits of this approach is the ease with which you can scale Logstash and those components.

We're going to scale each of the pieces we introduced and installed in Chapter 3. Those being:

- Elasticsearch - Which is handling search and storage. We're going to add nodes to our Elasticsearch cluster to provide more capacity and redundancy.
- Logstash - Which is consuming and indexing the events. We're going to install an additional indexer instance that provides some redundancy for Logstash. The indexer will have a duplicate configuration to our existing indexer.

WARNING As with all scaling and performance management this solution may not work for your environment or fully meet your requirements. Our introduction will show you the basics of making Logstash more resilient and per-

formant. From there you will need to monitor and tune Logstash to achieve the precise results you need.

This is a fairly basic introduction to scaling these components and, as with its installation, scaling Logstash is significantly easier and more elegant using tools like Puppet or Chef. Again setting up either is beyond the scope of this book but there are several Puppet modules for Logstash on the Puppet Forge and a Chef cookbook. These either support some minimal scaling or can be adapted to deliver these capabilities.

Scaling Elasticsearch

Elasticsearch is naturally very amenable to scaling. It's easy to build new nodes and Elasticsearch supports both unicast and multicast clustering out of the box with very limited configuration required. We're going to create two new Ubuntu hosts to run Elasticsearch on and then join these hosts to the existing cluster.

Elasticsearch host #1

- Hostname: grinner.example.com
- IP Address: 10.0.0.20

Elasticsearch host #2

- Hostname: sinner.example.com
- IP Address: 10.0.0.21

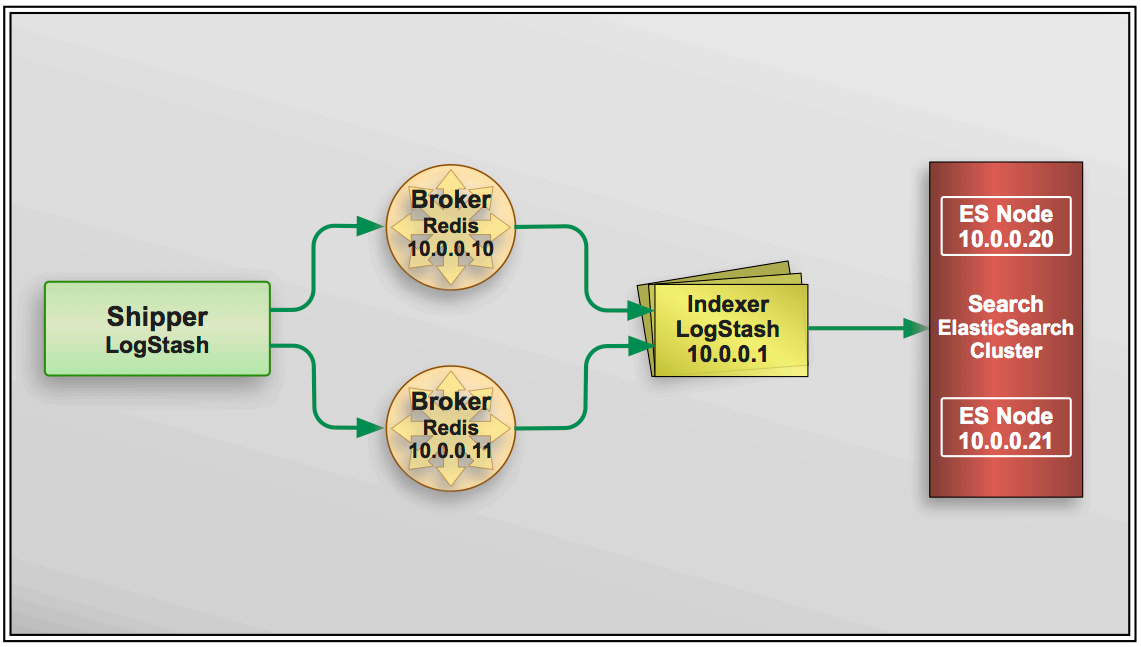

Figure 8.1: Elasticsearch scaling

Installing additional Elasticsearch hosts

Firstly, we need to install Java as a prerequisite to Elasticsearch.

Listing 8.1: Installing Java for Elasticsearch

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

We also have DEB packages for Elasticsearch that we can use on Ubuntu. We can download from the Elasticsearch download page or use their repositories.

First we install the Elastic.co package key.

Listing 8.2: Downloading the Elastic package key again

```
$ wget -O - https://artifacts.elastic.co/GPG-KEY-elasticsearch |
 sudo apt-key add -
```

Now we add the Elastic repository to our Apt configuration.

Listing 8.3: Adding the Elasticsearch repo again

```
$ echo "deb https://artifacts.elastic.co/packages/5.x/apt stable
 main" | sudo tee -a /etc/apt/sources.list.d/elastic-5.x.list
```

Now we install Elasticsearch.

Listing 8.4: Installing another Elasticsearch

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

Repeat this process for both new hosts.

Configuring our Elasticsearch cluster and new nodes

We're going to update the Elasticsearch configuration on each node, including our original `smoker` node. We're going to uncomment and change the cluster and node name, and configure networking and clustering, on all nodes. We're going to choose the cluster name of `logstash` for the environment our cluster is running in, which we established in Chapter 3.

Listing 8.5: New cluster and node names

```
cluster.name: logstash
node.name: smoker
network.host: [ _local_, _non_loopback:ipv4_ ]
discovery.zen.ping.unicast.hosts: ["smoker.example.com", "
grinner.example.com", "sinner.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 node types

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.yml` 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 8.6: Configuring our Elasticsearch cluster

```
cluster.name: logstash
node.name: smoker
. . .
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 8.7: Restarting Elasticsearch to enable clustering

```
$ sudo service elasticsearch restart
```

We would now configure the remaining nodes: `grinner` and `sinner` in the same manner.

Checking the Elasticsearch cluster

We can then check Elasticsearch is running and has joined the cluster by checking the Cluster Health API like so:

Listing 8.8: Checking the cluster status.

```
$ curl -XGET 'http://10.0.0.1:9200/_cluster/health?pretty=true'
{
  "cluster_name" : "logstash",
  "status" : "green",
  "timed_out" : false,
  "number_of_nodes" : 4,
  "number_of_data_nodes" : 4,
  "active_primary_shards" : 30,
  "active_shards" : 60,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 0
```

NOTE That's weird. Four nodes? Where did our fourth node come from? That's Logstash itself which joins the cluster as a client. So we have three data nodes and a client node.

We can see that our cluster is named `logstash` and its status is `green`. Green means all shards, both primary and replicas are allocated and functioning. A `yellow` cluster status will mean that only the primary shards are allocated, i.e. the cluster has not yet finished replication across its nodes. A `red` cluster status means there are shards that are not allocated.

Configuring our nodes in Logstash

Finally we need to tell Logstash about our new nodes by updating the `output` section of our `central.conf` configuration file.

Listing 8.9: Initial scaled central configuration

```
output {
. . .
  elasticsearch {
    hosts => ["10.0.0.1", "10.0.0.20", "10.0.0.21" ]
  }
}
```

We've added a new option, `hosts`, to our `elasticsearch` output. The `hosts` option contains an array of the IP addresses of our existing and new Elasticsearch nodes. Using this array Logstash will perform balanced writes to all of the nodes in the Elasticsearch cluster.

Alternatively, we can use Elasticsearch's own HTTP transport protocol to find all the nodes in the cluster and automatically add them to the `hosts` option. This is

done using a new option called `sniffing`.

Listing 8.10: Using sniffing to scale our cluster

```
output {
. . .
  elasticsearch {
    hosts => "10.0.0.1"
    sniffing => true
  }
}
```

This will connect to the Elasticsearch server at `10.0.0.1` and then query it for a list of other nodes in the cluster and automatically add them to the `hosts` list.

Monitoring our Elasticsearch cluster

Using the command line API is one way of monitoring the health of your Elasticsearch cluster but a far better method is to use one of the several plugins that are designed to do this. Plugins are add-ons for Elasticsearch that can be installed via the `elasticsearch-plugin` tool.

Since Elasticsearch 5.0.0, there's only one plugin that performs adequate monitoring. It's called X-Pack and is released by Elastic. It is partially commercial, there a free level and a commercial level. X-Pack is an Elastic Stack plugin that includes security, alerting, monitoring, reporting, and graph capabilities.

Let's install it on our `smoker` host first:

Listing 8.11: Installing X-Pack

```
smoker$ sudo /usr/share/elasticsearch/bin/elasticsearch-plugin
install x-pack
-> Downloading x-pack from elastic
[=================================================] 100%
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@     WARNING: plugin requires additional permissions     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
* java.lang.RuntimePermission accessClassInPackage.com.sun.
activation.registries
* java.lang.RuntimePermission getClassLoader
* java.lang.RuntimePermission setContextClassLoader
* java.lang.RuntimePermission setFactory
* java.security.SecurityPermission createPolicy.JavaPolicy
* java.security.SecurityPermission getPolicy
* java.security.SecurityPermission putProviderProperty.BC
* java.security.SecurityPermission setPolicy
* java.util.PropertyPermission * read,write
* java.util.PropertyPermission sun.nio.ch.bugLevel write
* javax.net.ssl.SSLPermission setHostnameVerifier
See http://docs.oracle.com/javase/8/docs/technotes/guides/
security/permissions.html
for descriptions of what these permissions allow and the
associated risks.

Continue with installation? [y/N]y
-> Installed x-pack
```

We need to then install the plugin onto the `grinner` and `sinner` hosts too. We also need to update the `elasticsearch.yml` configuration file on all of these hosts.

Listing 8.12: Adding X-Pack configuration

```
xpack.security.enabled: false
action.auto_create_index: .security,.monitoring*,.watches,.
triggered_watches,.watcher-history*
```

We then install the other half of the plugin into Kibana on `smoker`.

Listing 8.13: Install the X-Pack plugin into Kibana

```
$ sudo /usr/share/kibana/bin/kibana-plugin install x-pack
```

You can then register the X-Pack plugin and view it in Kibana. There's a free version that is partially capable and a commercial version if you wish to purchase a more fully featured version of the plugin.

Managing Elasticsearch data retention

One of the other key aspects of managing Elasticsearch scaling and performance is working out how long to retain your log data. Obviously this is greatly dependent on what you use the log data for, as some data requires longer-term retention than other data.

TIP Some log data, for example financial transactions, need to be kept for all time. But does it need to be searchable and stored in Elasticsearch forever? Probably not. In which case it is easy enough to output certain events to a different store like a file from Logstash for example using the `file` output plugin. This becomes your long-term storage and if needed you can also send your events to shorter-term storage in Elasticsearch.

Deleting unwanted indexes

Logstash by default creates an index for each day, for example `index-2012.12.31` for the day of 12/31/2012. You can keep these indexes for as long as you need (or you have disk space to do so) or set up a regular "log" rotation. To do this you can use Elasticsearch's own Delete API to remove older indexes, for example using `curl`:

Listing 8.14: Deleting indexes

```
$ curl -XDELETE http://10.0.0.1:9200/logstash-2012.12.31
```

Here we're deleting the `logstash-2012.12.31` index. You can easily automate this, for example this ticket contains an example Python script that deletes old indexes. We've reproduced it in the book's source code too. Another example is a simple Bash script found in this GitHub repository. Additionally the recently introduced Curator tool (see Curator section below) can also make managing LogStash indexes very simple.

Using any of these you can set up an automated regime to remove older indexes to match whatever log retention cycle you'd like to maintain.

Optimizing indexes

It's also a good idea to use Elasticsearch's optimize function to optimize indexes and make searching faster. You can do this on individual indexes:

Listing 8.15: Optimizing indexes

```
$ curl -XPOST 'http://10.0.0.1:9200/logstash-2013.01.01/
_optimize'
```

Or on all indexes:

Listing 8.16: Optimizing all indexes

```
$ curl -XPOST 'http://10.0.0.1:9200/_optimize'
```

It's important to note that if your indexes are large that the optimize API call can take quite a long time to run. You can see the size of a specific index using the Elasticsearch Indices Stats API like so:

Listing 8.17: Getting the size of an index

```
$ curl 'http://10.0.0.1:9200/logstash-2012.12.31/_stats?clear=
truestore=truepretty=true'
. . .
  "total" : {
    "store" : {
      "size" : "110.5mb",
      "size_in_bytes" : 115965586,
      "throttle_time" : "0s",
      "throttle_time_in_millis" : 0
    }
  }
}
. . .
```

TIP There are also some simple community tools for working with Elasticsearch and Logstash that you might find handy here.

Curator

More recently to support managing Logstash indexes the Elasticsearch team has released a tool called Curator. Curator helps you automate the process of deleting, optimizing and manage indexes on your Elasticsearch cluster.

Listing 8.18: Installing curator

```
$ sudo pip install elasticsearch-curator
```

TIP Curator works best with Elasticsearch 1.0 or later. If you're running Logstash 1.4.0 or later this is the version you should have. If you use an earlier version of Elasticsearch you can try Curator 0.6.2. You can install it via 'pip' also like so: `pip install elasticsearch-curator==0.6.2`.

Curator installs a binary called `curator` onto your host. It allows you to manage Elasticsearch indexes. For example, to delete indexes.

Listing 8.19: Deleting indexes with Curator

```
$ curator --host 10.0.0.1 -d 30
```

This will delete indexes older than thirty days, specified using the `-d` flag, on our

`10.0.0.1` host.

Curator can also optimize indexes and close indexes. Closing indexes is highly useful when you need to keep indexes for a while but don't need to search them, for example you might need to keep 30 days of logs but only search the last 7 days. This ensures optimal performance of your Logstash instance as closed indexes only occupy space and don't get searched when you query your data. This ensures your queries are fast and limited only to the data you need. To close indexes you would run:

Listing 8.20: Closing indexes using Curator

```
$ curator --host 10.0.0.1 -c 7
```

This will close all indexes older than 7 days.

To see the full list of Curator's capabilities run it with the `-h` flag.

Listing 8.21: Getting Curator help

```
$ curator -h
```

You can also find a blog post showing more of Curator's capabilities here and you can find the Curator source code here.

More Information

Elasticsearch scaling can be a lot more sophisticated than I've been able to elaborate on here. For example, we've not examined the different types of Elasticsearch node we can define: allowing nodes to be cluster masters, to store or not store data, or to act as "search load balancers." Nor have we discussed hardware recommendations or requirements.

There are a variety of other sources of information, including this excellent video and this post about how to scale Elasticsearch and you can find excellent help on the `#elasticsearch` IRC channel on Freenode or the Elasticsearch mailing list.

TIP A common, and worth calling out specifically, Elasticsearch problem at scale is the number of open files. Elasticsearch opens a lot of files and sometimes can hit the `nofile` limit of your distribution. The Elasticsearch team have written an article that talks about how to address this issue.

Scaling Logstash

Thus far we've got some redundancy in our environment and we've built an Elasticsearch cluster but we've only got a single Logstash indexer receiving events and passing them to Elasticsearch. This means if something happens to our Logstash indexer then Logstash stops working. To reduce this risk we're going to add a second Logstash indexer to our environment running on a new host.

Logstash host #1

- Hostname: smoker.example.com
- IP Address: 10.0.0.1

Logstash host #2

- Hostname: picker.example.com
- IP Address: 10.0.0.2

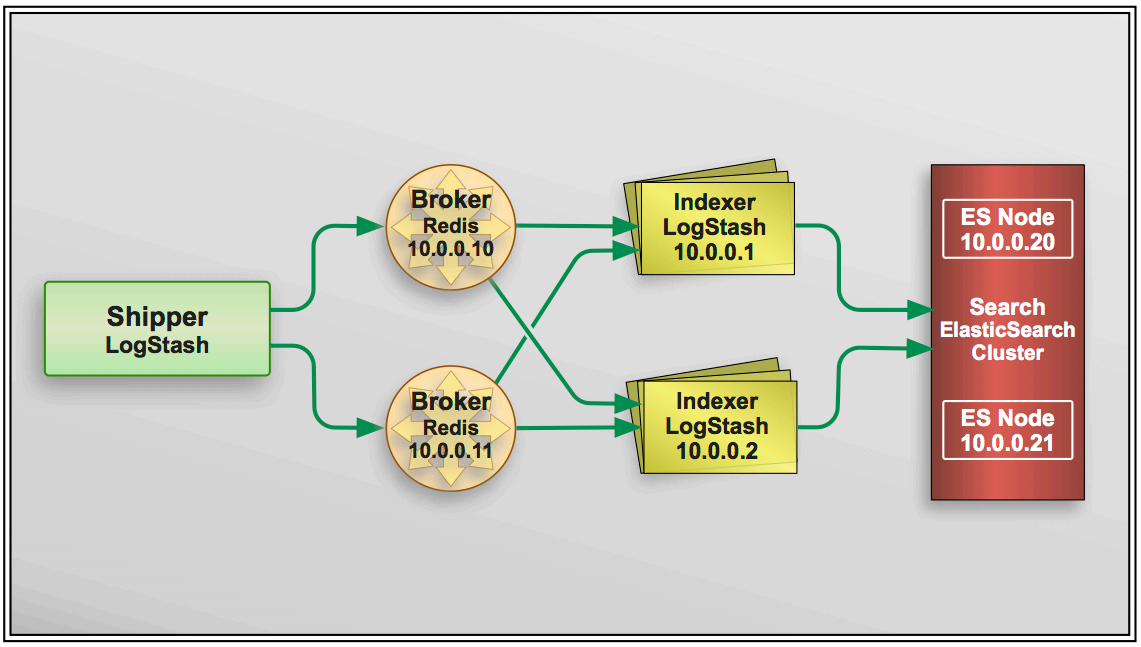

Figure 8.2: Logstash indexer scaling

Creating a second indexer

To create a second indexer we need to replicate some of the steps from Chapter 3 we used to set up our initial Logstash indexer.

Listing 8.22: Setting up a second indexer

```
picker$ sudo apt-get -y install default-jre
picker$ wget -O - https://artifacts.elastic.co/GPG-KEY-
elasticsearch | sudo apt-key add -
picker$ echo "deb https://artifacts.elastic.co/packages/5.x/apt
stable main" | sudo tee -a /etc/apt/sources.list.d/elastic-5.x.
list
picker$ sudo apt-get update
picker$ sudo apt-get install logstash
smoker$ sudo scp /etc/logstash/conf.d/central.conf bob@picker:/
etc/logstash/conf.d
```

You can see we've added the Logstash repository and installed the Logstash package and copied the existing central `smoker` host's `central.conf` configuration file. We're all set up and ready to go. The best thing is that we don't even need to make any changes to our existing Logstash configuration.

Now let's start our new Logstash instance and run the Logstash service.

Listing 8.23: Starting the central Logstash server

```
picker$ sudo service logstash start
```

Load balancing

Now we have two Logstash servers running with the same configuration. But only one of them, `smoker`, is actually receiving any log events. There are a number of ways and a number of sources from which we can receive events and each have their own mechanisms for load balancing events. We've primarily used Filebeat to send log events to Logstash so we're going to focus on providing some load balancing capabilities for it.

Filebeat's Logstash output can be configured to emit events to one or more Logstash instances. Let's look at a load balanced configuration now. We go back to our `/etc/filebeat/filebeat.yml` configuration file and update our `output.logstash` block.

Listing 8.24: Load balanced Filebeat

```
. . .

output.logstash:
  hosts: ["10.0.0.1:5044", "10.0.0.2:5044"]
  loadbalance: true
```

You can see we've added both our Logstash hosts to the `hosts` array in our `output.logstash` block. We've also specified the `loadbalance` option and set it to `true`. The default mode Filebeat operates in when we specify the `loadbalance` option is to send events to one host after another. We can also add the `worker` option to specify additional workers, essentially opening additional network connections to each Logstash host. This obviously consumes more memory and CPU as more workers are added.

This isn't true load balancing though, more failover. Filebeat will send to one Logstash instance until that host doesn't ACK and then try the alternative host.

The principal alternative to this approach is to place a HAProxy instance or cluster in front of our Logstash instances.

For other log sources we just need to specify all of the IP addresses of our Logstash instances.

Logstash failover

So what happens now? As both Logstash indexers are using the same configuration and both are listening for inputs from the agents they will start to both process

events. If one indexer is unavailable then you'll see some events being received on the other Logstash instance. Assuming they have the same configuration (you are using configuration management by now right?) then the events will be processed the same way and pass into our Elasticsearch cluster to be stored. Now if something goes wrong with one Logstash instance you will have a second functioning instance that will continue to process. This model is also easy to scale further and you can add additional Logstash instances as needed to meet performance or redundancy requirements.

Summary

As you can see, with some fairly simple steps that we've made our existing Logstash environment considerably more resilient and provided some additional performance capacity. It's not quite perfect and it will probably need to be tweaked as we grow but it provides a starting point to expand upon as our needs for additional resources increase.

In the next chapter we'll look at how we can extend Logstash to add our own plugins.

Chapter 9

Extending Logstash

One of the awesome things about Logstash is that there are so many ways to get log events into it, manipulate and filter events once they are in and then push them out to a whole variety of destinations. Indeed, at the time of writing, there were nearly 100 separate input, filter and output plugins. Every now and again though you encounter a scenario where you need a new plugin or want to customize a plugin to better suit your environment.

TIP The best place to start looking at the anatomy of Logstash plugins are the plugins themselves. You'll find examples of inputs, filters and outputs for most purposes in the Logstash source code repository.

Now our project has almost reached its conclusion we've decided we better learn how to extend Logstash ourselves to cater for some of the scenarios when you need to modify or create a plugin.

WARNING This introduction is a simple, high-level introduction to how to extend Logstash by adding new plugins. It's not a guide to writing or learning Ruby.

Plugin organization

Since Logstash 1.5.0 plugins have been shipped as Ruby Gems. As we've seen earlier in the book a lot of plugins are shipped with the Logstash package. Others are available from the Logstash plugins GitHub account. You can use the `logstash-plugin` binary to install these.

To construct our own plugins we can use some simple scaffold code that the Logstash team provides for each plugin type:

- For input plugins you can use this template.
- For filter plugins you can use this template.
- For output plugins you can use this template.

We can copy these sample plugins and create a new plugin of our own from the template. Let's quickly look at the plugin template's structure.

Listing 9.1: Plugin organization

```
$ tree logstash-input-pluginname├──
 Gemfile├──
 LICENSE├──
 README.md├──
 Rakefile├──
 lib│
   └── logstash│
       └── inputs│
           └── pluginname.rb├──
 logstash-input-pluginname.gemspec└──
 spec
    └── inputs
        └── pluginname_spec.rb
```

We can see that it looks like a pretty typical Ruby Gem. The core of our plugin is contained in the `lib/logstash` directory, inside a directory named for the type of plugin being developed: `input`, `filter`, or `output`. We also have a `spec` directory to hold any tests for the plugin. Rounding out our template are a `README`, license and a `Rakefile` to help us automate our plugin's build.

We've also got a `.gemspec` or Gem specification file that helps us build our plugin.

Listing 9.2: The Logstash input sample Gemspec

```
Gem::Specification.new do |s|
  s.name = 'logstash-input-example'
  s.version         = '0.1.2'
  s.licenses = ['Apache License (2.0)']
  s.summary = "This example input streams a string at a
definable interval."
  s.description = "This gem is a logstash plugin required to be
installed on top of the Logstash core pipeline using $LS_HOME/
bin/logstash-plugin install gemname. This gem is not a stand-
alone program"
  s.authors = ["Elastic"]
  s.email = 'info@elastic.co'
  s.homepage = "http://www.elastic.co/guide/en/logstash/current/
index.html"
  s.require_paths = ["lib"]

  # Files
  s.files = `git ls-files`.split($\)
   # Tests
  s.test_files = s.files.grep(%r{^(test|spec|features)/})

  # Special flag to let us know this is actually a logstash
plugin
  s.metadata = { "logstash_plugin" => "true", "logstash_group" =>
 "input" }

  # Gem dependencies
  s.add_runtime_dependency 'logstash-core', '>= 1.4.0', '< 2.0.0
'
  s.add_runtime_dependency 'logstash-codec-plain'
  s.add_runtime_dependency 'stud'
  s.add_development_dependency 'logstash-devutils'
end
```

XXX

This file, and the `s.metadata` line specifically, configures the Gem as a Logstash

plugin. Also important is the `s.version` field which tells Logstash the version of the plugin. This replaces the previously-used `milestone` method in older plugins.

The varying versions you specify produce logging output warnings or status that tell people about the maturity of your plugin. Versions produce the following results:

- 0.1.x - A warning message: "This plugin isn't well supported by the community and likely has no maintainer."
- 0.9.x - A warning message: "This plugin should work but would benefit from use by folks like you. Please let us know if you find bugs or have suggestions on how to improve this plugin."
- 1.x.x - No warning message.

You would also specify any Gem or library dependencies in the Gemspec.

NOTE All plugins must have a runtime dependency on the `logstash-core` gem and a development dependency on the `logstash-devutils` gem.

Anatomy of a plugin

Let's look at one of the more basic plugins, the `stdin` input, and see what we can learn about plugin anatomy. You can see the full plugin code here but let's look at some key pieces.

A plugin starts with a series of `require` statements that include any supporting code.

Listing 9.3: The stdin input plugin's requires

```
# encoding: utf-8
require "logstash/inputs/base"
require "logstash/namespace"
require "concurrent/atomics"
require "socket" # for Socket.gethostname
```

Firstly, each plugin requires the Logstash base class for the type of plugin, here `logstash/inputs/base`. We also require the base Logstash class, `logstash/namespace`.

We also include any prerequisites, in this case the `stdin` input requires the Socket library for the `gethostname` method and the `concurrent` gem to provide some concurrency helpers. In the case of the `concurrent` gem we'd also add that as a dependency in our Gem specification file to ensure it is added when the plugin is built.

Listing 9.4: Adding the concurrent dependency to the Gemspec

```
s.add_runtime_dependency 'concurrent-ruby'
```

We then create a class and inherit the `LogStash::Inputs::Base` class we required above. For filters we would require the `LogStash::Filters::Base` class and outputs the `LogStash::Outputs::Base` class respectively.

Listing 9.5: The stdin input plugin's requires

```
class LogStash::Inputs::Stdin < LogStash::Inputs::Base
  config_name "stdin"

  default :codec, "line"

  def initialize(*args)
    super(*args)
    @stop_requested = Concurrent::AtomicBoolean.new(false)
  end

  def register
    @host = Socket.gethostname
    fix_streaming_codecs
  end

. . .
```

Each plugin also requires a name provided by the `config_name` method. The `config_name` provides Logstash with the name of the plugin.

We also specify the default codec the plugin uses, here `line`, using `default :codec`. The `line` codec decodes events that are lines of text data.

Every plugin also has the `register` method inside which you should specify anything needed to initialize the plugin, for example our `stdin` input sets the `@host` host name instance variable.

Each type of plugin then has a method that contains its core execution:

- For inputs this is the `run` method, which is expected to run forever.
- For filters this is the `filter` method.
- For outputs this is the `receive` method.

Let's look our `stdin`'s `run` method.

Listing 9.6: The core methods of our stdin plugin

```
  def run(queue)
    while @stop_requested.false?
      begin
        # Based on some testing, there is no way to interrupt an
IO.sysread nor
        # IO.select call in JRuby. Bummer :(
        data = $stdin.sysread(16384)
        @codec.decode(data) do |event|
          decorate(event)
          event["host"] = @host if !event.include?("host")
          queue << event
        end
      rescue IOError, EOFError, LogStash::ShutdownSignal
        # stdin closed or a requested shutdown
        @stop_requested.make_true
        break
      rescue => e
        # ignore any exception in the shutdown process
        break if @stop_requested.true?
        raise(e)
      end
    end
    finished
  end
```

So what happens in our `stdin` input? After the `register` method initializes the plugin then the `run` method is called. The `run` method takes a parameter which is the queue of incoming data. In the case of the `stdin` input the loop inside this method is initiated.

The input then runs until stopped, processing any incoming data from `STDIN`, decoding the incoming data using the default codec specified and turning it into events.

The `decorate` method then applies any metadata we've set in our configuration,

for example if we've set a tag on the event.

The decoded and decorated event is then injected back into the queue to be passed to Logstash for any further processing.

One last method is defined in our `stdin` input, `teardown`. When this method is specified then Logstash will execute it when the plugin is being shutdown. It's useful for cleaning up, in this case closing the pipe.

Listing 9.7: stdin's teardown method

```
  def teardown
    @stop_requested.make_true
    @logger.debug("stdin shutting down.")
    $stdin.close rescue nil
    finished
  end
```

Creating our own input plugin

Now we've got a broad understanding of how a plugin works let's now create one of our own. We're going to start with a simple plugin to read lines from a named pipe: a very simple pipe-based `file` input.

First, let's create a directory to hold our plugin.

Listing 9.8: Creating the namedpipe input plugin directory

```
$ mkdir -p /src/logstash-input-namedpipe
```

Let's make it a Git repository.

Listing 9.9: Creating the namedpipe Git repository

```
$ cd /src/logstash-input-namedpipe
$ git init
```

Let's populate this directory with the input plugin template.

Listing 9.10: Adding the template scaffold for an input plugin

```
$ cd /tmp
$ git clone https://github.com/logstash-plugins/logstash-input-
example.git
$ cd logstash-input-example
$ rm -rf .git
$ cp -R * /src/logstash-input-namedpipe/
```

Here we've changed into the `/tmp` directory, used Git to clone the example input plugin template and then copied that template into our plugin directory.

We're now going to delete the example plugin file, rename the RSpec test file and rename our Gem specification.

Listing 9.11: Removing and renaming template input files

```
$ cd /src/logstash-input-namedpipe
$ rm lib/logstash/input/example.rb
$ mv spec/inputs/example_spec.rb spec/inputs/namedpipe_spec.rb
$ mv logstash-input-example.gemspec logstash-input-namedpipe.
gemspec
```

We'd also edit our Gem specification to update it to the correct name, version and update any required dependencies.

Next let's edit our input itself. First we created a file to hold our plugin.

Listing 9.12: Creating the namedpipe plugin file

```
$ cd logstash-input-namedpipe
$ touch lib/logstash/input/namedpipe.rb
```

Now let's populate our file, starting with adding our `require` statements and creating our base class.

Listing 9.13: The namedpipe framework

```
require 'logstash/namespace'
require 'logstash/inputs/base'

class LogStash::Inputs::NamedPipe < LogStash::Inputs::Base
  . . .
end
```

We've added requires for an input and a class called `LogStash::Inputs::NamedPipe`.

Now let's add in our plugin's name and status using the `config_name` method. We're also going to specify the default codec, or format, this plugin will expect events to arrive in. We're going to specify the `line` codec as we expect our events to be text strings.

Listing 9.14: The namedpipe framework plugin options

```
require 'logstash/namespace'
require 'logstash/inputs/base'

class LogStash::Inputs::NamedPipe < LogStash::Inputs::Base
    config_name "namedpipe"

    default :codec, "line"

    # The pipe to read from
    config :pipe, :validate => :string, :required => true

    . . .
end
```

You can see we've also added a configuration option, using the `config` method. This method allows us to specify the configuration options and settings of our plugins, for example if we were configuring this input we could now use an option called `pipe`:

Listing 9.15: The namedpipe input configuration

```
input {
   namedpipe {
     pipe => "/tmp/ournamedpipe"
     type => "pipe"
   }
}
```

Configuration options have a variety of properties: you can validate the content of an option, for example we're validating that the `pipe` option is a `string`. You can add a default for an option, for example `:default => "default option"`, or indicate that the option is required. If an option is required and that option is not

provided then Logstash will not start.

Now let's add the guts of the `namedpipe` input.

Listing 9.16: The namedpipe input

```
require 'logstash/namespace'
require 'logstash/inputs/base'

class LogStash::Inputs::NamedPipe < LogStash::Inputs::Base
    config_name "namedpipe"
    default :codec, "line"
    config :pipe, :validate => :string, :required => true

    public
    def register
      @logger.info("Registering namedpipe input", :pipe => @pipe)

    end

    def run(queue)
      @pipe = open(pipe, "r+")
      @pipe.each do |line|
        line = line.chomp
        host = Socket.gethostname
        path = pipe
        @logger.debug("Received line", :pipe => pipe, :line =>
line)
        @codec.decode(line) do |event|
          decorate(event)
          event["host"] = host
          event["path"] = path
          queue << event
        end
      end
    end

    def teardown
      @pipe.close
      finished
    end
end
```

We've added three new methods: `register`, `run`, and `teardown`.

The `register` method sends a log notification using the `@logger` instance variable. Adding a log level method, in this case `info` sends an information log message. We could also use `debug` to send a debug-level message.

The `run` method is our queue of log events. It opens a named pipe, identified using our `pipe` configuration option. Our code constructs a source for our log event, that'll eventually populate the `host` and `path` fields in our event. We then generate a debug-level event and use the `to_event` method to take the content from our named pipe, add our host and path and pass it to Logstash as an event. The `run` method will keep sending events until the input is stopped.

When the input is stopped the `teardown` method will be run. This method closes the named pipe and tells Logstash that the input is finished.

Building our plugin

To build a Logstash plugin we treat it exactly like a Ruby gem. We can build our plugin based on the Gemspec.

First we'd install any dependencies with Bundler.

Listing 9.17: Installing plugin dependencies with Bundler

```
$ cd logstash-input-namedpipe
$ bundle install
```

Once this is done we can build our actual plugin gem.

Listing 9.18: Building the namedpipe plugin gem

```
$ cd logstash-input-namedpipe
$ gem build logstash-input-namedpipe.gemspec
```

This will create a new gem in the `logstash-input-namedpipe` directory.

Now let's add our new plugin to Logstash and see it in action.

TIP You can read more about creating input plugins in the Logstash documentation.

Adding new plugins

Adding new plugins to Logstash is done using the `logstash-plugin` binary. You just need a copy of the Gem file you built of your plugin.

Listing 9.19: Installing a plugin via Gem file

```
$ bin/logstash-plugin install /path/to/gemfile/logstash-input-
namedpipe-0.1.0.gem
```

You should now be able to see the installed plugin in the list of plugins on that Logstash server.

Listing 9.20: Checking our new plugin is installed

```
$ bin/logstash-plugin list
. . .
logstash-input-namedpipe
. . .
```

Writing a filter

Now we've written our first input let's look at another kind of plugin: a filter. As we've discovered filters are designed to manipulate events in some way. We've seen a variety of filters in Chapter 5 but we're going to write one of our own now. In this filter we're going to add a suffix to all `message` fields. Let's start by adding the code for our filter:

Listing 9.21: Our suffix filter

```
require "logstash/filters/base"
require "logstash/namespace"

class LogStash::Filters::AddSuffix < LogStash::Filters::Base
    config_name "addsuffix"

    config :suffix, :validate => :string

    public
    def register
    end

    public
    def filter(event)
      if @suffix
        msg = event["message"] + " " + @suffix
        event["message"] = msg
      end
      filter_matched(event)
    end
end
```

Let's examine what's happening in our filter. Firstly, we've required the prerequisite classes and defined a class for our filter: `LogStash::Filters::AddSuffix`. We've also named and set the status of our filter, the experimental `addsuffix` filter, using the `config_name` method.

We've also specified a configuration option using the `config` method which will contain the suffix which we will be adding to the event's `message` field.

Next, we've specified an empty `register` method as we're not performing any registration or plugin setup. The most important method, the `filter` method itself, takes the event as a parameter. In our case it checks for the presence of the `@suffix` instance variable that contains our configured suffix. If no suffix is

configured the filter is skipped. If the suffix is present it is applied to the end of our message and the message returned.

The `filter_matched(event)` method call at the end of our filter ensures any tags or other metadata specified in our configuration are applied to the event.

TIP If you want to drop an event during filtering you can use the `event.cancel` method.

Now we can configure our new filter, like so:

Listing 9.22: Configuring the addsuffix filter

```
filter {
  addsuffix {
    suffix => "ALERT"
  }
}
```

If we now run Logstash we'll see that all incoming events now have a suffix added to the `message` field of `ALERT` resulting in events like so:

Listing 9.23: An event with the ALERT suffix

```
{
  "host" => "smoker.example.com",
  "@timestamp" => "2013-01-21T18:43:34.531Z",
  "message" => "testing ALERT",
  "type" => "human"
}
```

You can now see how easy it is to manipulate events and their contents.

TIP You can read more about creating filter plugins here.

Writing an output

Our final task is to learn how to write the last type of plugin: an output. For our last plugin we're going to be a little flippant and create an output that generates CowSay events. First, we need to install a CowSay package, for example on Debian-distributions:

Listing 9.24: Installing CowSay on Debian and Ubuntu

```
$ sudo apt-get install cowsay
```

Or via a RubyGem:

Listing 9.25: Installing CowSay via a RubyGem

```
$ sudo gem install cowsay
```

This will provide a `cowsay` binary our output is going to use.

Now let's look at our CowSay output's code:

Listing 9.26: The CowSay output

```
require "logstash/outputs/base"
require "logstash/namespace"

class LogStash::Outputs::CowSay < LogStash::Outputs::Base
  config_name "cowsay"

  config :cowsay_log, :validate => :string, :default => "/var/
log/cowsay.log"

  public
  def register
  end

  public
  def receive(event)
    msg = `cowsay #{event["message"]}`
    File.open(@cowsay_log, 'a+') { |file| file.write("#{msg}") }
  end

end
```

Our output requires the prerequisite classes and creates a class called `LogStash::Outputs::CowSay`. We've specified the name of the output, `cowsay` with `config_name` method. We've specified a single configuration option using the `config` method. The option, `cowsay_log` specifies a default log file location, `/var/log/cowsay.log`, for our log output.

Next we've specified an empty `register` method as we don't have anything we'd like to register.

The guts of our output is in the `receive` method which takes an `event` as a parameter. In this method we've shell'ed out to the `cowsay` binary and parsed the `event["message"]` (the contents of the `message` field) with CowSay. It then writes this "cow said" message to our `/var/log/cowsay.log` file.

We can now configure our `cowsay` output:

Listing 9.27: Configuring the cowsay output

```
output {
  cowsay {}
}
```

You'll note we don't specify any options and use the default destination. If we now run Logstash we can generate some CowSay statements like so:

Figure 9.1: Cow said "testing"

You can see we have an animal message. It's easy to see how you can extend an output to send events or portions of events to a variety of destinations.

TIP You can read more about creating output plugins here.

Summary

This has been a very simple introduction to writing Logstash plugins. It gives you the basics of each plugin type and how to use them. You can build on these examples easily enough and solve your own problems with plugins you've developed yourself.

List of Figures

Listings

Index

Thanks! I hope you enjoyed the book.

ISBN 978-0-9888202-1-0