Graylog

Log Management with Graylog, Elasticsearch, MongoDB, Nginx, Fluentd, Vagrant and Docker

JORGE ACETOZI

Graylog

Log Management with Graylog, Elasticsearch, MongoDB, Nginx, Fluentd, Vagrant and Docker

Jorge Acetozi

This book is for sale at http://leanpub.com/graylog

This version was published on 2018-03-25

Tweet This Book!

Please help Jorge Acetozi by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought Graylog Book

The suggested hashtag for this book is #graylogbook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#graylogbook

Also By Jorge Acetozi

Continuous Delivery for Java Apps

Contents

About the Author . . . . . **i**

Introduction **1**

Why do I need a Centralized Logging System Like Graylog? **2**

Hands-on Introduction to Graylog **5**
- Graylog Components 6
- Pre-Requisites 8
- Set up Graylog using Vagrant and Docker 10
- Inputs 16
- Forwarding VM Logs using Rsyslog 22
- Getting Alerted 32
  - Streams 32
  - Alert 41
    - HTTP Notification 45
    - Email Notification using Amazon SES 55

Appendices **56**

Introduction to Docker **57**

Difference Between Container and Image **59**

Docker Hub **61**

Create your Account 62
Official Docker Repositories 64
Image Tags 66
Pulling Images From a Different Docker Registry 69
Non-Official Docker Images 70
Create a Repository, an Image and Push it to Docker Hub 71

Running Containers on Docker 78
Running Containers as Daemons 79
Container Clean Up 80
Naming Containers 82
Exposing Ports 83
Persistent Data with Volumes 85
Environment Variables 90

Docker Networking 91
Create a Bridge Network 93
Container Static IP Address 94
Linking Containers 95

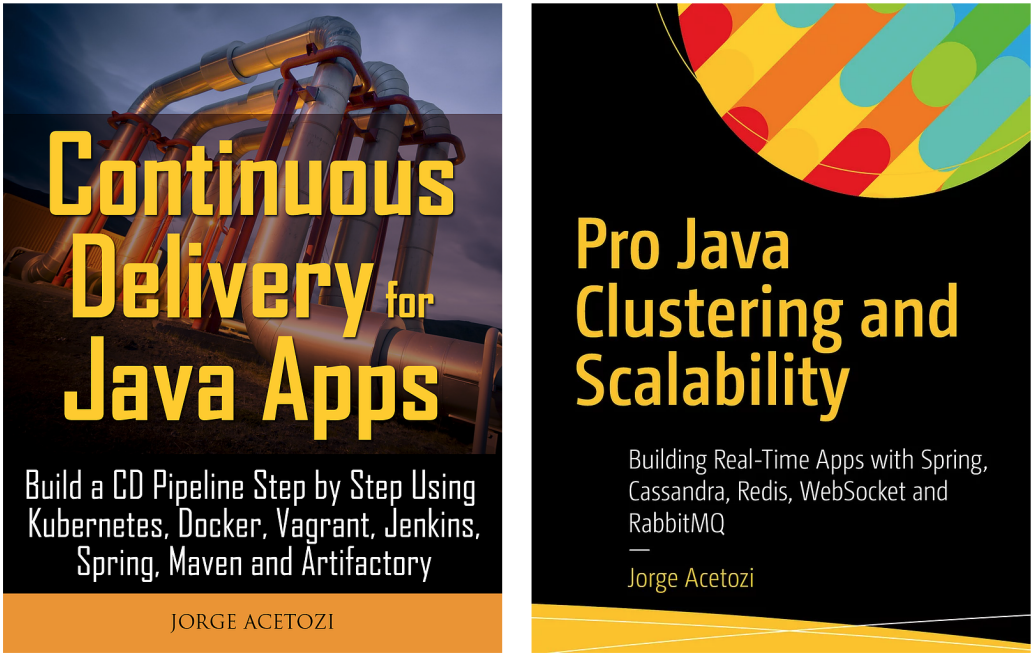
Most Used Docker Commands 99
Images 100
Containers 101
Misc 102

Building Docker Images 103
Dockerfile 104
FROM 105
ENV 105
RUN 106
WORKDIR 106
COPY 106
ADD 106
EXPOSE 107
ENTRYPOINT 107
VOLUME 108

USER 109

About the Author

Jorge Acetozi is a software engineer who spends almost his whole day having fun with things such as AWS, Kubernetes, Docker, Terraform, Ansible, Cassandra, Redis, Elasticsearch, Graylog, New Relic, Sensu, Elastic Stack, Fluentd, RabbitMQ, Kafka, Java, Spring, and much more! He loves deploying applications in production while thousands of users are online, monitoring the infrastructure, and acting quickly when monitoring tools decide to challenge his heart's health!



Author's Books

You can reach him at:

- Web site[1]

[1] https://www.jorgeacetozi.com

- This Book's web site[2]
- GitHub[3]
- LinkedIn[4]
- Facebook[5]
- Twitter[6]
- Medium[7]

[2] https://www.graylogbook.com
[3] https://github.com/jorgeacetozi
[4] https://www.linkedin.com/in/jorgeacetozi
[5] https://www.facebook.com/jorgeacetozi
[6] https://twitter.com/jorgeacetozi
[7] https://medium.com/jorgeacetozi

Introduction

Development is Development. Production is Production. Typically, in order to deploy a new feature to production, the new code written is manually tested (by developers and sometimes a QA team) and automated tested (unit, integration, acceptance, performance, smoke, security tests) against several different environments like `development` and `staging` before eventually landing on `production`. Basically, all this process is done to ensure that when the `release candidate` is ready to be deployed to production, the team has a significant level of confidence that the code actually works as intended.

Well, although I wrote a 600-pages book called Continuous Delivery for Java Apps: Build a CD Pipeline Step by Step Using Kubernetes, Docker, Vagrant, Jenkins, Spring, Maven and Artifactory[8] guiding the reader on how to implement all these steps and much more in practice using top-notch technologies and deployment strategies like Canary Release, I have a little secret to confess: most of the steps of a continuous delivery pipeline actually take place in controlled environments (internal networks, predictable traffic, and so on), and as I mentioned at the beginning of this section: development is development and production is production. Many things can (and I bet they will) go wrong in production, especially if your application is accessible on the Internet. **When something crashes, guess who is your best friend? That's right: Logs**.

[8] https://leanpub.com/continuous-delivery-for-java-apps

Why do I need a Centralized Logging System Like Graylog?

Suppose that your application is designed on top of Microservices architecture[9] and a particular request goes through service A, B, and C. Each of these services run a load balancer that spreads requests among 10 different servers running application servers and reverse proxies (Nginx, for example). Also, these services use PostgreSQL and Redis instances running on Amazon RDS[10] and Amazon Elasticache[11] respectively.

Now answer me, how can you keep track of what's going on when something goes wrong in this particular scenario? Are you going to `ssh` 10 production servers from service A, 10 production servers from service B, 10 production servers from service C, and for each of them use commands like `tail -f logfile | grep "ERROR"` for the Nginx `access.log` and `error.log` as well as the application server logs? Don't forget that on top of that you would have to monitor the AWS logs for PostgreSQL and Redis as well.

Well, I think you got the idea. The bottom line is that the traditional way of visualizing logs (analyzing specific log files stored on the disk) doesn't scale at all. In fact, there are many other problems related to this traditional approach. Let's list out some of them:

- **Security issues**: you have to `ssh` production servers every time you want to see the logs, and as you know, some angry system administrators that fight on UFC for fun don't like that for security reasons, especially if you are not an infrastructure expert. As a result, you can get very hurt;
- **Logs are streams, not files**: you cannot assume that logs are being directly stored into files on the disk for cloud-native applications that follows the 12-Factor logging[12] practices. Instead, the only guarantee you have in the cloud

[9] https://www.nginx.com/blog/introduction-to-microservices/
[10] https://aws.amazon.com/rds/
[11] https://aws.amazon.com/elasticache/
[12] https://12factor.net/logs

is that the disk is ephemeral. Later on this book we will learn more about 12-Factor logging practices and this will become clearer;
- **Filtered content**: some log entries might contain sensitive information and must be hidden while others don't. How to control that and avoid that someone that is directly logged into the server could see the sensitive log entries?
- **Inefficiency**: when something goes wrong, the time spent on logging into many machines and `greping` log files one by one is probably the time that would take to query Graylog, figure out what's wrong and start working on the fix.

Now, let's list out some benefits you instantaneously get when you use Graylog as your centralized logging system:

- **Increased Security**: people don't need to have access to the production servers as the log visualization is displayed on the beautiful Graylog Web Interface through their preferred **modern** browser. Keep in mind that if you try to use something like Internet Explorer 6, your problem is not logs, my friend!;
- **Authentication and Authorization**: it's possible to manage users and roles very easily. Actually, it's even possible to integrate it with your existing LDAP solution (if you have one);
- **Powerful Querying Capabilities**: you can use a beautiful and powerful search syntax very close to the Apache Lucene syntax rather than relying on insanely ugly regular expressions;
- **Built-in Alerting**: Graylog has a build-in alerting mechanism, so you don't need to be all the time worried about your applications behavior and suffering with nightmares at night. Instead, you can define certain patterns that would trigger an alert and then get alerted on your mobile with a graceful message like this one: "Sorry to bother you, Mr. Jorge Acetozi, but it looks like someone attempted to `ssh` your production server (IP: XXX.XXX.XXX.XXX) for 5 consecutive times in less than a minute, so I was wondering here if you could take a look at it. By the way, your coffee is ready and waiting for you on your desk. Thank you very much for your time!";
- **Input Everything**: you don't have to restrict the centralized logging system to your application logs only. You can input every type of logs in it, such as the `sshd` daemon in the alerting example before, operating system logs, firewall logs... Well, pretty much everything!;

- **Custom Dashboards**: create custom dashboards and display them on televisions spread around your company's office;
- **Geolocation**: add geolocation metadata to log specific log entries (such as Nginx access.log) so that you can visualize where the requests are coming from;
- **Reports**: create reports that could even be useful on Marketing decisions, such as the more frequently accessed pages.

Well, and the list goes on!

However, everything comes at a price. Maintaining a production infrastructure of a centralized logging system like Graylog is neither easy nor cheap. As far as Graylog is concerned, it relies on different tools such as Elasticsearch[13] and MongoDB[14] to get the work done. In production, such tools should be deployed as clusters to allow high-availability, performance, and scalability, which of course leads to a increased complexity of the overall solution. Hopefully, this book will provide a good understanding of each technology involved in Graylog's architecture in a hands-on (and not boring) fashion so that you can get started with Graylog as soon as possible.

[13] https://www.elastic.co/

[14] https://www.mongodb.com/

Hands-on Introduction to Graylog

Graylog is a powerful open-source log management platform written in Java[15] that ingests logs from various sources and allows to search and visualize the logs in a beautiful web interface. It provides many useful built-in features like user management, alerting, custom dashboards, amazing querying capabilities, a REST API and others (most of them are covered in this book), which makes it ideal as the logging solution for your company (whether it's a small or huge one).

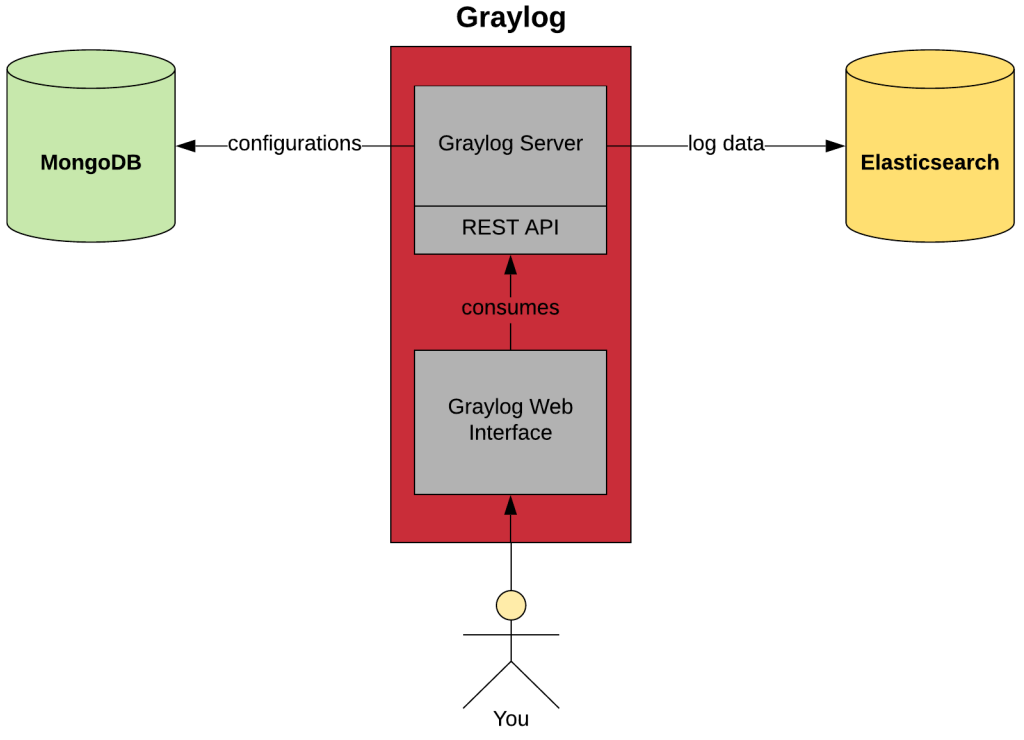
Graylog is built on top of a master-slave architecture, which means that slaves can be added for horizontal scalability. As we are going to learn later on this book, clustering Graylog is pretty straightforward.

Horizontal scalability happens when you add more nodes to your cluster of machines. Vertical scalability happens when you increase a machine's hardware power.

[15] https://github.com/Graylog2/graylog2-server

Graylog Components

Basically, Graylog stores the log data in Elasticsearch, which is a powerful open-source, RESTful, distributed search and analytics engine built on top of Apache Lucene. Besides, Graylog also uses MongoDB, which is a document-oriented NoSQL database[16], to store configuration data such as user information, inputs and streams configurations, and so on.



Graylog Components

So, as you can see, getting started with Graylog is as simple as setting up an instance for Elasticsearch, MongoDB, and Graylog itself. Basically, a Graylog instance is composed by two components:

[16] https://en.wikipedia.org/wiki/Document-oriented_database

- **Graylog Web Interface**: offers capabilities for searching and analyzing the indexed log data and allows configuring the Graylog environment (inputs, dashboards, etc). It communicates with the Graylog Server through the Graylog REST API;
- **Graylog Server**: encapsulates all the logic related to Graylog (a fancy way to say "does all the hard work") and provides the Graylog REST API.

Note that none of the log messages are ever stored in MongoDB. Thus, MongoDB does not have a big system impact, and you won't have to worry too much about scaling it even though Graylog is ingesting thousands of log messages per second. When it comes to scaling the system, the challenges typically involve scaling Graylog and Elasticsearch.

In this section, we are not going to worry about high-availability, scalability, performance, and others non-functional requirements (we will have the whole book to have fun and dive into these exciting subjects). Instead, it will provide a hands-on introduction to Graylog, showcasing some of its handy features so that you can actually understand in practice the concept of Centralized Logging System that we have discussed earlier. So, let's get our hands dirty!

Pre-Requisites

Before moving on, please make sure you have VirtualBox[17], Vagrant[18], and Docker[19] installed on your machine. By the time I was writing this book, this was my setup:

Operating System	VirtualBox	Vagrant	Docker for Mac
macOS Sierra	5.2.6	2.0.2	17.12.0-ce

Everything in this book runs on top of Linux Ubuntu virtual machines, so as long as you have VirtualBox, Vagrant, and Docker properly installed on your machine, you should be good to go.

Now, let's clone the GitHub repository used along this book onto your machine:

```
$ git clone https://github.com/jorgeacetozi/ebook-graylog
```

Go into the directory `ebook-graylog` and list it:

```
$ cd ebook-graylog/
$ ls -l
total 16
-rw-r--r--  1 jorgeacetozi  staff  4517 13 Dez 17:31 README.md
drwxr-xr-x  3 jorgeacetozi  staff    96  2 Ago  2017 images
drwxr-xr-x@ 9 jorgeacetozi  staff   288 24 Jul  2017 log-generator-app
drwxr-xr-x  9 jorgeacetozi  staff   288 23 Fev 10:34 vagrant
```

The `log-generator-app` directory contains the Java (Spring Boot) application we are going to use later on this book to generate logs for Graylog. The `vagrant` directory contains all the recipes (Vagrantfiles, Graylog configurations, FluentD configurations, and Nginx configurations) needed to set up our virtual machines, which means that although every piece of code written in these files is explained throughout the book, you don't have to create everything from the scratch in order to run the recipes on your machine.

[17] https://www.virtualbox.org/
[18] https://www.vagrantup.com/
[19] https://www.docker.com/

If you are not familiar with Vagrant or Docker, I suggest you stop now and read the Appendice before moving on so that you get the most out of this book.

Set up Graylog using Vagrant and Docker

Now that you've got the `ebook-graylog` directory on your machine, go into the `vagrant` subdirectory and check its structure:

```
$ cd vagrant
$ tree
├── Vagrantfile-fluentd-high-availability
├── Vagrantfile-fluentd-single-instance
├── Vagrantfile-hello-world
├── Vagrantfile-without-fluentd
├── Vagrantfile-without-fluentd-nginx-contentpacks
└── conf
    ├── fluentd
    │   ├── aggregator
    │   │   └── fluentd-aggregator.conf
    │   ├── forwarder
    │   │   └── fluentd-forwarder.conf
    │   └── single-instance
    │       └── fluentd-single-instance.conf
    ├── graylog
    │   ├── graylog-server-master
    │   │   ├── graylog-contentpacks.conf
    │   │   └── graylog.conf
    │   ├── graylog-server-slave
    │   │   ├── graylog-contentpacks.conf
    │   │   └── graylog.conf
    │   └── nginx
    │       ├── nginx-with-fluentd.conf
    │       ├── nginx-without-fluentd-contentpacks.conf
    │       └── nginx-without-fluentd.conf
    └── log-generator-app
        └── nginx
            ├── nginx-fluentd-forwarder-aggregator.conf
            ├── nginx-fluentd-single-instance.conf
            └── nginx-without-fluentd-contentpacks.conf
```

As you can see, there are five `Vagrantfiles` that we are going to use along this book. The idea is to start very simple and evolve our architecture according to the problems presented until we get to a really beautiful solution that is pretty much what you would be doing in production.

Note that in this book we are using Vagrant to orchestrate **local** virtual machines and Docker containers. So, obviously, this is not a production environment. However, the logging architecture we are going to implement and evolve throughout this book is pretty much what successful companies are doing in production, except that instead of local virtual machines, they run it on the cloud (most of them). Note that the concepts still exactly the same.

For this introductory example, we are going to use the `Vagrantfile-hello-world` as our Vagrantfile. By default, Vagrant will look for a file named `Vagrantfile` in the current directory, so let's instruct it to look for the file `Vagrantfile-hello-world` by setting up the `VAGRANT_VAGRANTFILE` environment variable instead:

```
$ export VAGRANT_VAGRANTFILE=Vagrantfile-hello-world
```

Let's check the contents of this file:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

$script = <<SCRIPT
sysctl -w vm.max_map_count=262144
SCRIPT

Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/trusty64"

  config.vm.define "vm_hello_world" do |helloworld|
    helloworld.vm.hostname = "helloworld"
    helloworld.vm.network "private_network", ip: "10.0.0.10"
    helloworld.vm.provider "virtualbox" do |vb|
      vb.memory = "4096"
    end

    helloworld.vm.provision "shell", inline: $script

    helloworld.vm.provision "docker" do |d|
      d.run "mongo",
        image: "mongo"
```

```
    end

    helloworld.vm.provision "docker" do |d|
      d.run "elasticsearch",
        image: "docker.elastic.co/elasticsearch/elasticsearch:5.6.8",
        args: "-e 'xpack.security.enabled=false'"
    end

    helloworld.vm.provision "docker" do |d|
      d.run "graylog",
        image: "graylog/graylog:2.4.3-1",
        args: "--link mongo \
               --link elasticsearch \
               -p 9000:9000 -p 12201:12201 -p 514:514 -p 5555:5555 \
               -e 'GRAYLOG_WEB_ENDPOINT_URI=http://10.0.0.10:9000/api'"

#               -e 'GRAYLOG_TRANSPORT_EMAIL_ENABLED=true' \
#               -e 'GRAYLOG_TRANSPORT_EMAIL_HOSTNAME=AWS_SMTP_SERVER' \
#               -e 'GRAYLOG_TRANSPORT_EMAIL_PORT=587' \
#               -e 'GRAYLOG_TRANSPORT_EMAIL_USE_AUTH=true' \
#               -e 'GRAYLOG_TRANSPORT_EMAIL_USE_TLS=true' \
#               -e 'GRAYLOG_TRANSPORT_EMAIL_USE_SSL=false' \
#               -e 'GRAYLOG_TRANSPORT_EMAIL_AUTH_USERNAME=AWS_SMTP_USERNAME' \
#               -e 'GRAYLOG_TRANSPORT_EMAIL_AUTH_PASSWORD=AWS_SMTP_PASSWORD'"
    end
  end

end
```

Basically, it instructs Vagrant to:

- Set up a single VM named `vm_hello_world` using the `ubuntu/trusty64` box;
- Configure the VM with the IP address `10.0.0.10`;
- Increase the VM memory to `4GB`;
- Executes the shell provisioner to increase the maximum map count check (needed for Elasticsearch[20]);
- Start three Docker containers (MongoDB, Elasticsearch, and Graylog) using the Docker provisioner[21].

[20] https://www.elastic.co/guide/en/elasticsearch/reference/5.6/_maximum_map_count_check.html
[21] https://www.vagrantup.com/docs/provisioning/docker.html

Note that there are a bunch commented Graylog environment variables in the Graylog container configuration. We are going to use it later to set up the alerting mechanism via e-mail.

Execute the `vagrant up` command and grab a cup of coffee while your VM is being started:

```
$ vagrant up
...
==> vm_hello_world: Running provisioner: shell...
    vm_hello_world: Running: inline script
    vm_hello_world: vm.max_map_count = 262144
==> vm_hello_world: Running provisioner: docker...
    vm_hello_world: Installing Docker onto machine...
==> vm_hello_world: Starting Docker containers...
==> vm_hello_world: -- Container: mongo
==> vm_hello_world: Running provisioner: docker...
==> vm_hello_world: Starting Docker containers...
==> vm_hello_world: -- Container: elasticsearch
==> vm_hello_world: Running provisioner: docker...
==> vm_hello_world: Starting Docker containers...
==> vm_hello_world: -- Container: graylog
```

Smooth! Let's `ssh` the VM to see our three containers up and running:

```
$ vagrant ssh vm_hello_world
```

List the containers:

```
vagrant@helloworld:~$ docker container ls
CONTAINER ID   CREATED         NAMES
cf62bb96df30   3 minutes ago   graylog
68f875c59858   4 minutes ago   elasticsearch
c4c3ec17e890   6 minutes ago   mongo
```

The `docker container ls` output was truncated to fit better in the book.

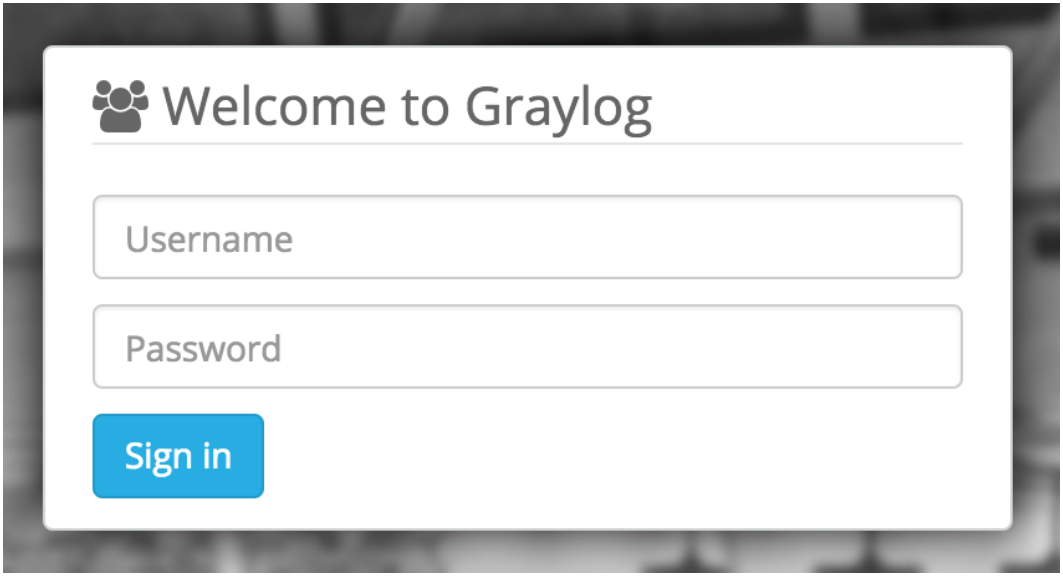
Let's check that the port 9000 (Graylog Web Interface) is waiting for connections:

```
vagrant@helloworld:~$ netstat -an | grep 9000
tcp6       0      0 :::9000                 :::*                    LISTEN
```

Awesome! Exit the VM:

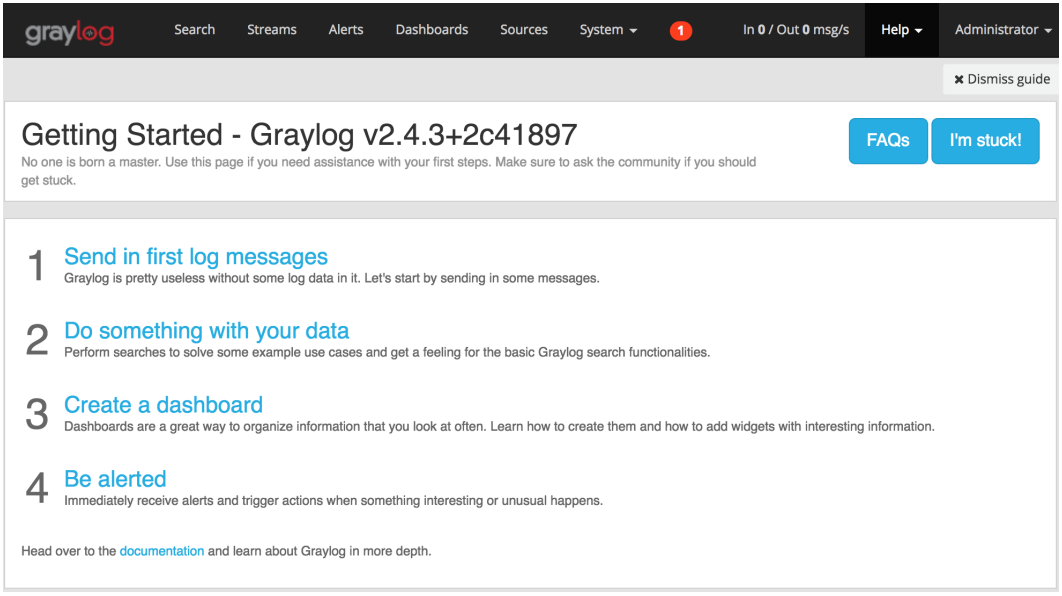
```
$ exit
```

Open your browser and navigate to `http://10.0.0.10:9000`. The Graylog Web Interface should pops up with an authentication form:



Login

Congratulations, your Graylog instance is up and running! Type in **admin** for the `username` and `password` fields and click on **Sign In**. You should see the Graylog Web Interface starting page:



Graylog Web Interface: Starting Page

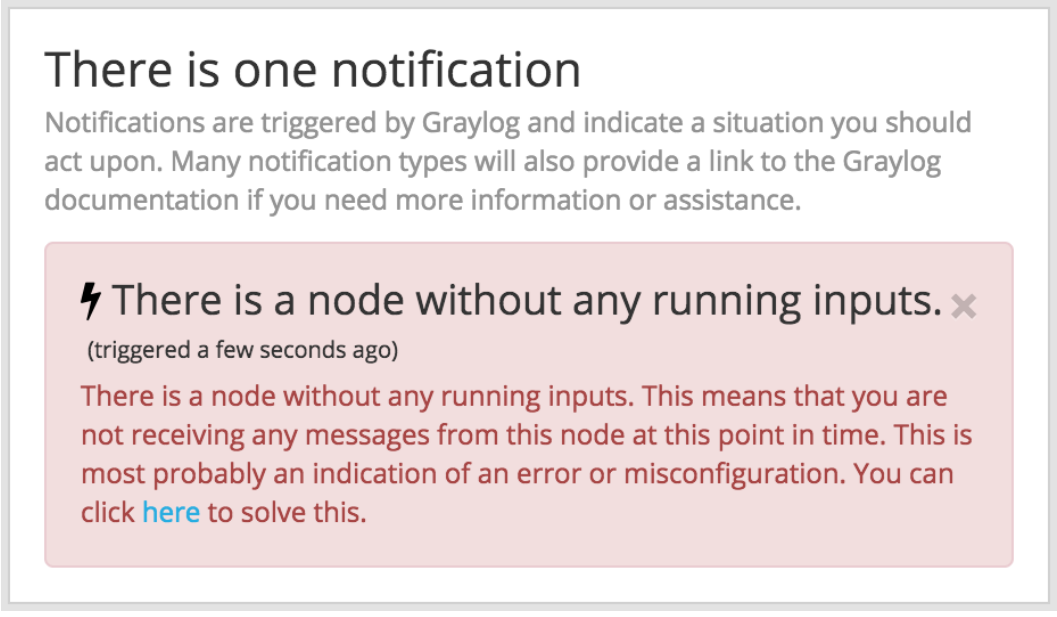
Basically, the top bar includes:

- The menu items;
- The amount of log messages being ingested per second;
- A counter with the system notifications (in red). When a relevant event takes place, a new notification will pop up on the top menu as shown.

Besides, a Getting Started guide is shown in the middle page, which is pretty much what we are going through right now.

Inputs

Back to the Graylog interface, click on the notification to see what is it about.

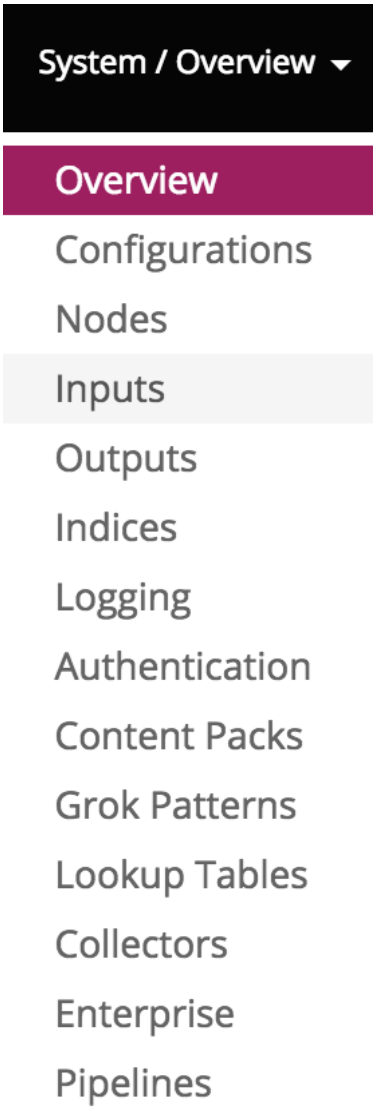


Notification: No Inputs Found

As you may guess, Graylog is useless if no log messages are coming in. So, before you can start sending data to it, you have to set up the so called **Inputs**. Basically, inputs are the Graylog components responsible for accepting log messages.

As you have just created your Graylog instance, there are no inputs yet. As a result, Graylog friendly reminds you that you have to create an input if you want to start accepting log messages.

Let's create our first Input using the web interface. Navigate to the **System -> Inputs** menu item:

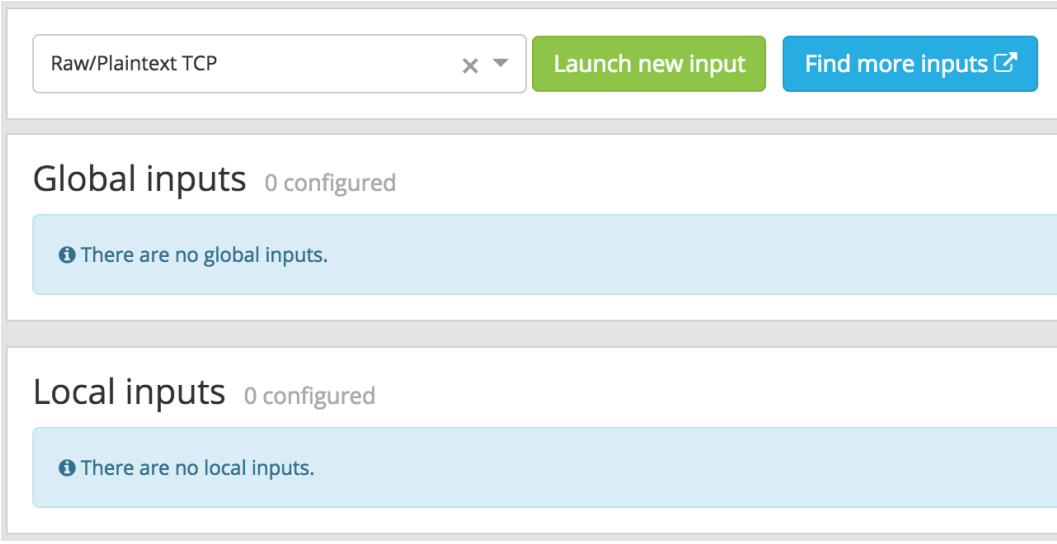


Menu: Inputs

It's also possible to create Inputs using the REST API.

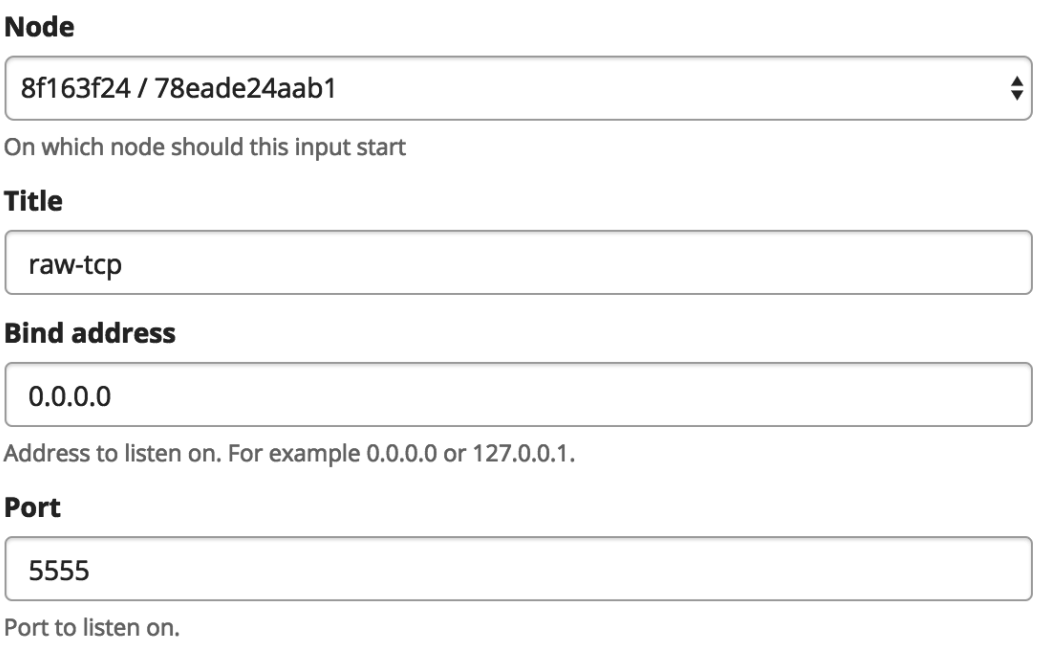
Note that we still have neither global nor local inputs configured. Select **Raw/Plain-**

text TCP and click on **Launch new input**:



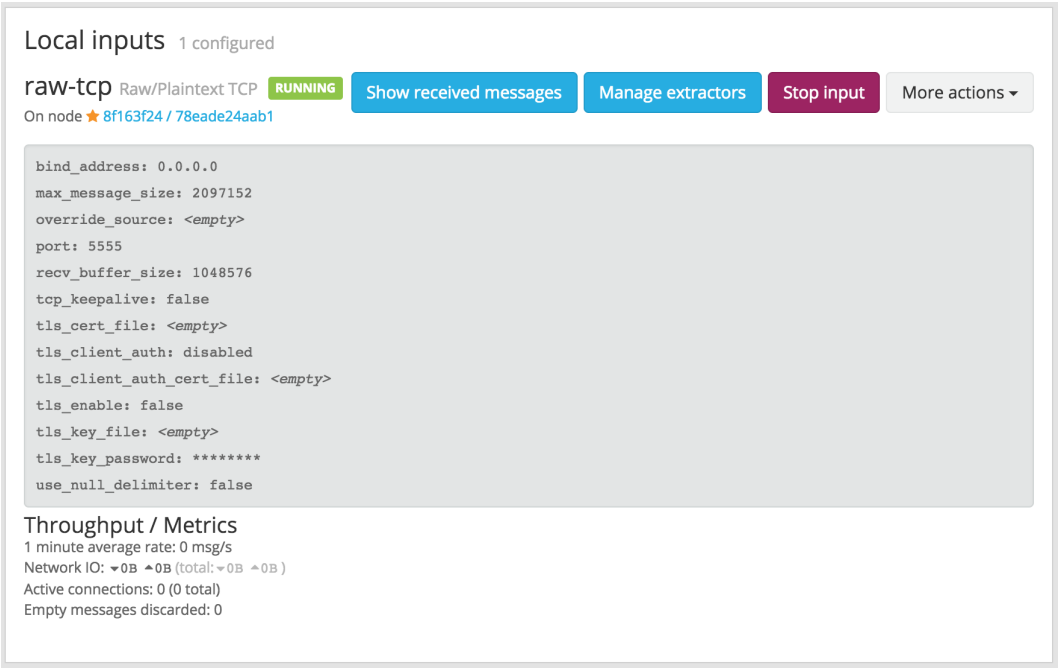
Input: Launch New Raw/Plaintext TCP Input

Let's launch a `local` input, which is essentially one that runs on top of a specific Graylog node (as we currently have just one node, it doesn't really matters whether it's a `global` or a `local` input). Select the **Node** and give it the **Title** `raw-tcp`:



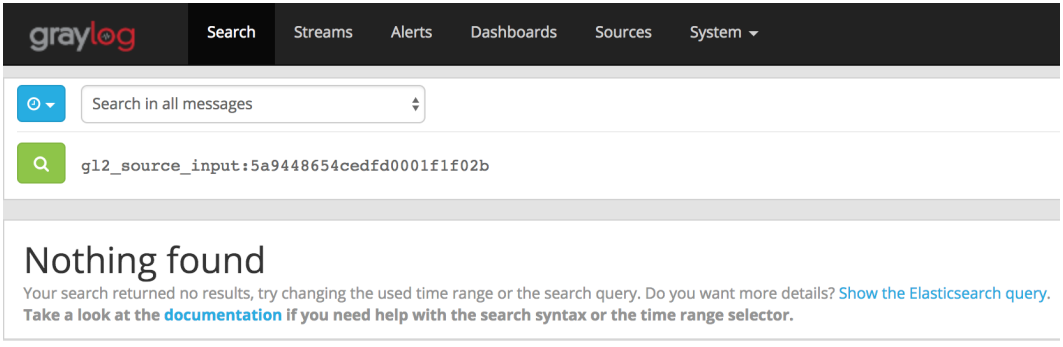
Raw/Plaintext TCP Input Configuration

Note that it will run on port `5555`. Click on **Save** and you should end up with the `raw-tcp` input in the **Running** status as follows:



Raw/Plaintext TCP Input Status: Running

Click on **Show received messages** so that you can visualize the log messages received by this input. Nothing should be found as we haven't sent anything yet:



No Messages Found

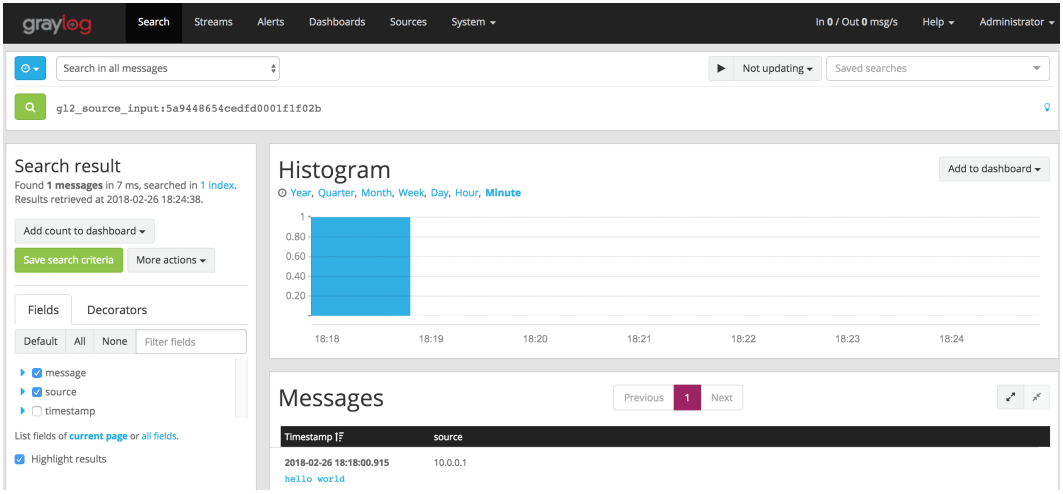
As we have set up a Raw/Plaintext input, let's send a "hello world" plain-text message

from the terminal using Netcat[22]:

```
$ echo "hello world" | nc 10.0.0.10 5555
```

After sending the message, if you quickly return to Graylog web interface you will be able to see the message throughput updating to **In 1 / Out 1 msg/s** on the top menu, indicating that the message was received and properly indexed in Elasticsearch.

Return to the browser and click on the green search icon:



Hello World Message

Here we go, the message was successfully received and indexed! On the left sidebar you can select which fields you want to visualize in the main panel. Of course this was a simple message with just few fields, but later we are going to send log messages with lots of fields, so it's quite helpful to have the ability to filter by field.

[22] https://en.wikipedia.org/wiki/Netcat

Forwarding VM Logs using Rsyslog

The Syslog protocol[23] is a standard for message logging. Basically, it decouples the software that generates the logs, the system that stores them, and the software used to visualize and analyze them (which in our case is Graylog).

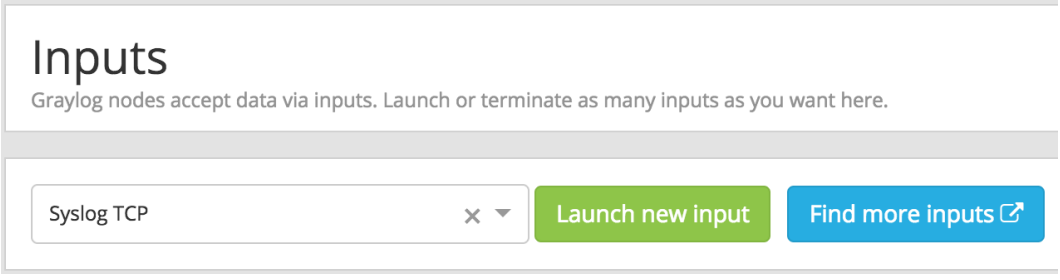
Rsyslog is the newest (out of three) implementations of the Syslog protocol and it's available by default on a number of Unix systems and Linux distributions. Besides implementing the Syslog protocol, it also extends its functionalities with:

- Support for buffered operation modes where messages are buffered locally if the receiver is not ready;
- Reliable transport using TCP;
- ISO 8601 timestamp with millisecond granularity and timezone information;
- Support for TLS;

And many other relevant features.

The other two implementations are Syslog (yes, this implementation has the same name as the protocol, so don't get confused!) and syslog-ng.

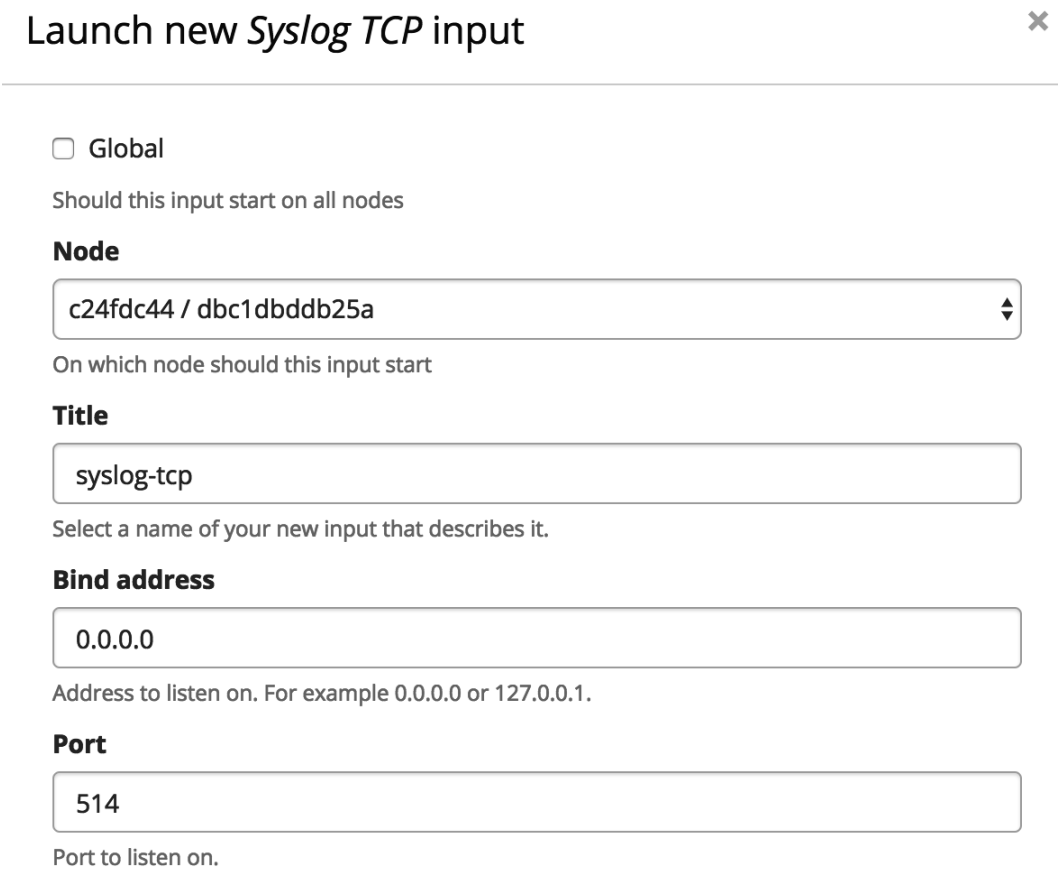
What's the first step we have to take in order get log messages in Graylog? That's correct, creating an Input! Navigate to **System -> Inputs** menu item, select **Syslog TCP** and click on **Launch new input**:



Input: Launch New Syslog TCP Input

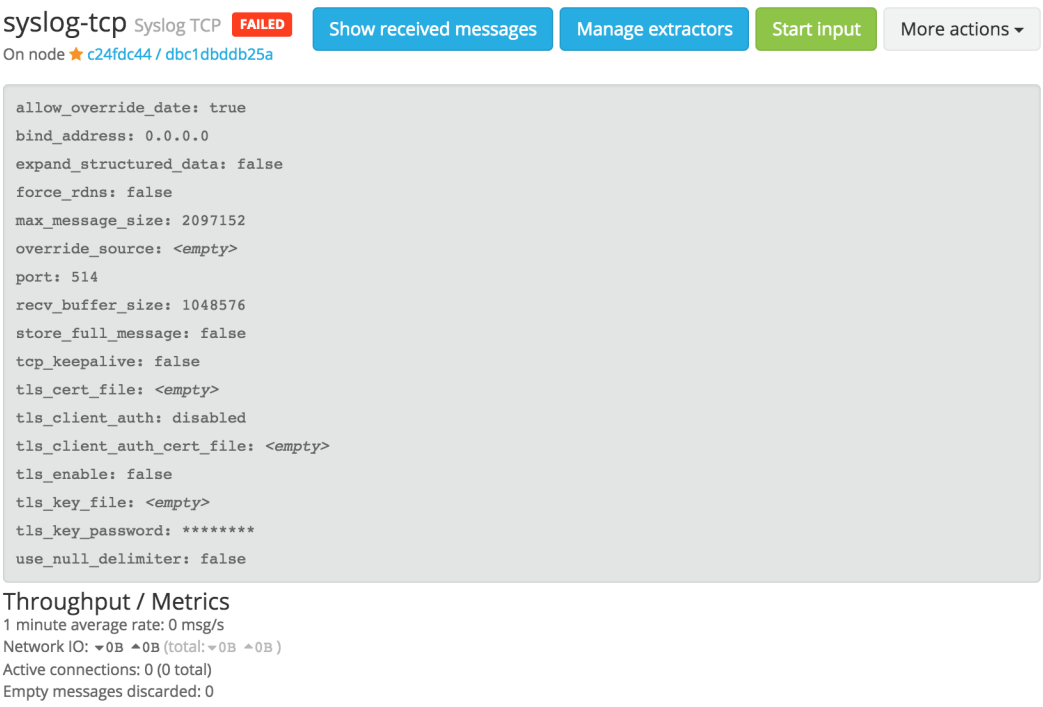
[23] https://tools.ietf.org/html/rfc5424

Select the **Node**, give it the **Title** `syslog-tcp` and keep the port `514` as follows:



Syslog TCP Input: Configuration

Finally, click on **Save**.



Syslog TCP Input: Failed to Launch

As you can see, the Input has failed to launch. Let's try to figure out why is that. First, `ssh` the `vm_hello_world`:

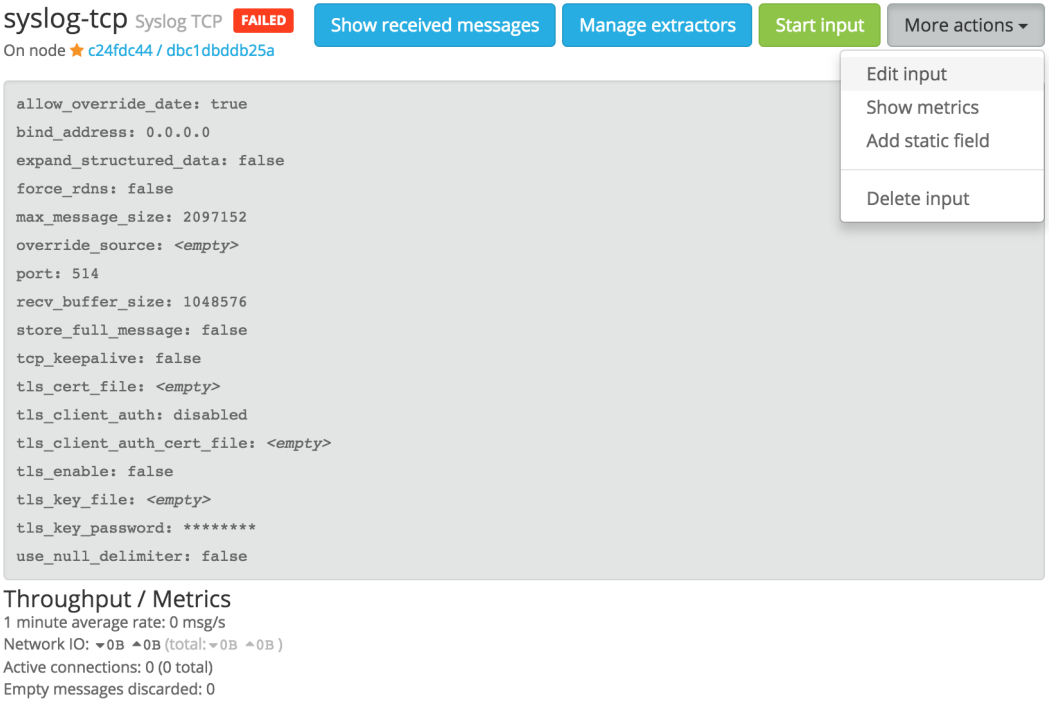
```
$ vagrant ssh vm_hello_world
```

Check the Graylog logs using the `docker container logs` command:

```
vagrant@helloworld:~$ docker container logs graylog
018-02-26 20:52:46,359 ERROR: org.graylog2.shared.inputs.InputLauncher - The [org.gra\
ylog2.inputs.syslog.tcp.SyslogTCPInput] input with ID <5a94739d4cedfd0001f21f10> misf\
ired. Reason: Permission denied.
org.graylog2.plugin.inputs.MisfireException: org.graylog2.plugin.inputs.MisfireExcept\
ion: org.jboss.netty.channel.ChannelException: Failed to bind to: /0.0.0.0:514
        at org.graylog2.plugin.inputs.MessageInput.launch(MessageInput.java:158) ~[graylog.j\
ar:?]
        at org.graylog2.shared.inputs.InputLauncher$1.run(InputLauncher.java:84) [graylog.ja\
r:?]
        at com.codahale.metrics.InstrumentedExecutorService$InstrumentedRunnable.run(Instrum\
entedExecutorService.java:176) [graylog.jar:?]
        at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511) [?:1.8.0_\
151]
        at java.util.concurrent.FutureTask.run(FutureTask.java:266) [?:1.8.0_151]
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149) [\
?:1.8.0_151]
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624) [\
?:1.8.0_151]
        at java.lang.Thread.run(Thread.java:748) [?:1.8.0_151]
Caused by: org.graylog2.plugin.inputs.MisfireException: org.jboss.netty.channel.Chann\
elException: Failed to bind to: /0.0.0.0:514
        at org.graylog2.plugin.inputs.transports.NettyTransport.launch(NettyTransport.java:1\
55) ~[graylog.jar:?]
        at org.graylog2.plugin.inputs.MessageInput.launch(MessageInput.java:155) ~[graylog.j\
ar:?]
        ... 7 more
Caused by: org.jboss.netty.channel.ChannelException: Failed to bind to: /0.0.0.0:514
        at org.jboss.netty.bootstrap.ServerBootstrap.bind(ServerBootstrap.java:272) ~[graylo\
g.jar:?]
        at org.graylog2.plugin.inputs.transports.NettyTransport.launch(NettyTransport.java:1\
41) ~[graylog.jar:?]
        at org.graylog2.plugin.inputs.MessageInput.launch(MessageInput.java:155) ~[graylog.j\
ar:?]
        ... 7 more
Caused by: java.net.SocketException: Permission denied
        at sun.nio.ch.Net.bind0(Native Method) ~[?:1.8.0_151]
        at sun.nio.ch.Net.bind(Net.java:433) ~[?:1.8.0_151]
        at sun.nio.ch.Net.bind(Net.java:425) ~[?:1.8.0_151]
        at sun.nio.ch.ServerSocketChannelImpl.bind(ServerSocketChannelImpl.java:223) ~[?:1.8\
.0_151]
        at sun.nio.ch.ServerSocketAdaptor.bind(ServerSocketAdaptor.java:74) ~[?:1.8.0_151]
        at org.jboss.netty.channel.socket.nio.NioServerBoss$RegisterTask.run(NioServerBoss.j\
```

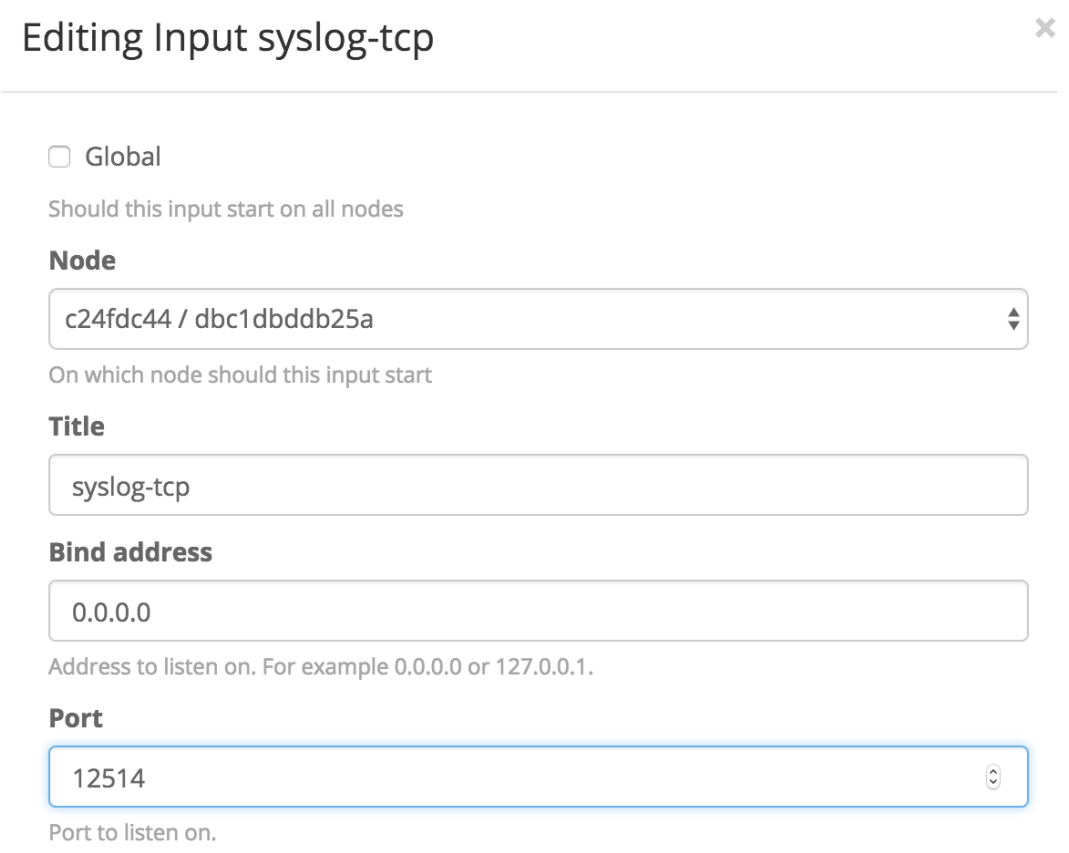
```
ava:193) ~[graylog.jar:?]
        at org.jboss.netty.channel.socket.nio.AbstractNioSelector.processTaskQueue(AbstractN\
ioSelector.java:391) ~[graylog.jar:?]
        at org.jboss.netty.channel.socket.nio.AbstractNioSelector.run(AbstractNioSelector.ja\
va:315) ~[graylog.jar:?]
        at org.jboss.netty.channel.socket.nio.NioServerBoss.run(NioServerBoss.java:42) ~[gra\
ylog.jar:?]
        at org.jboss.netty.util.ThreadRenamingRunnable.run(ThreadRenamingRunnable.java:108) \
~[graylog.jar:?]
        at org.jboss.netty.util.internal.DeadLockProofWorker$1.run(DeadLockProofWorker.java:\
42) ~[graylog.jar:?]
        ... 3 more
```

As the exception states, we got a **Permission denied** error. Basically, that happened because the ports ranging from **0-1023** are reserved for the root user. So, back to the Graylog web interface, click on **More actions** and edit the `syslog-tcp` input:



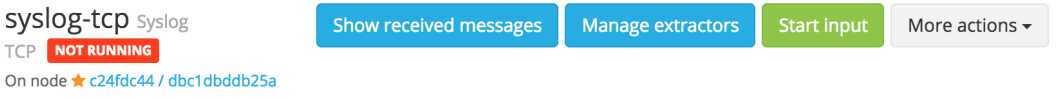
Syslog TCP Input: Edit

Change the port to `12514` and save it.



Syslog TCP Input: Port 12514

The input status should change from `Failed` to `Not Running`:



Syslog TCP Input Status: Not Running

Now just click on **Start Input** and voila!

Syslog TCP Input Status: Running

Click on **Show Received Messages** to see the incoming messages handled by our newly created Syslog TCP Input. As we have not set up the VM Rsyslog configuration to forward log messages to Graylog, obviously there should be no messages in it yet.

Let's set up the Rsyslog running on the VM so that it can forward the VM logs to the Graylog instance. You should probably already be logged into the VM as we were checking the Graylog container logs earlier, but if not, just `vagrant ssh` it again. Edit the Rsyslog configuration file as `sudo`:

```
vagrant@helloworld:~$ sudo vim /etc/rsyslog.conf
```

Scroll down to the bottom of the file, activate the `insertion mode` hitting `i` and add the following line in it:

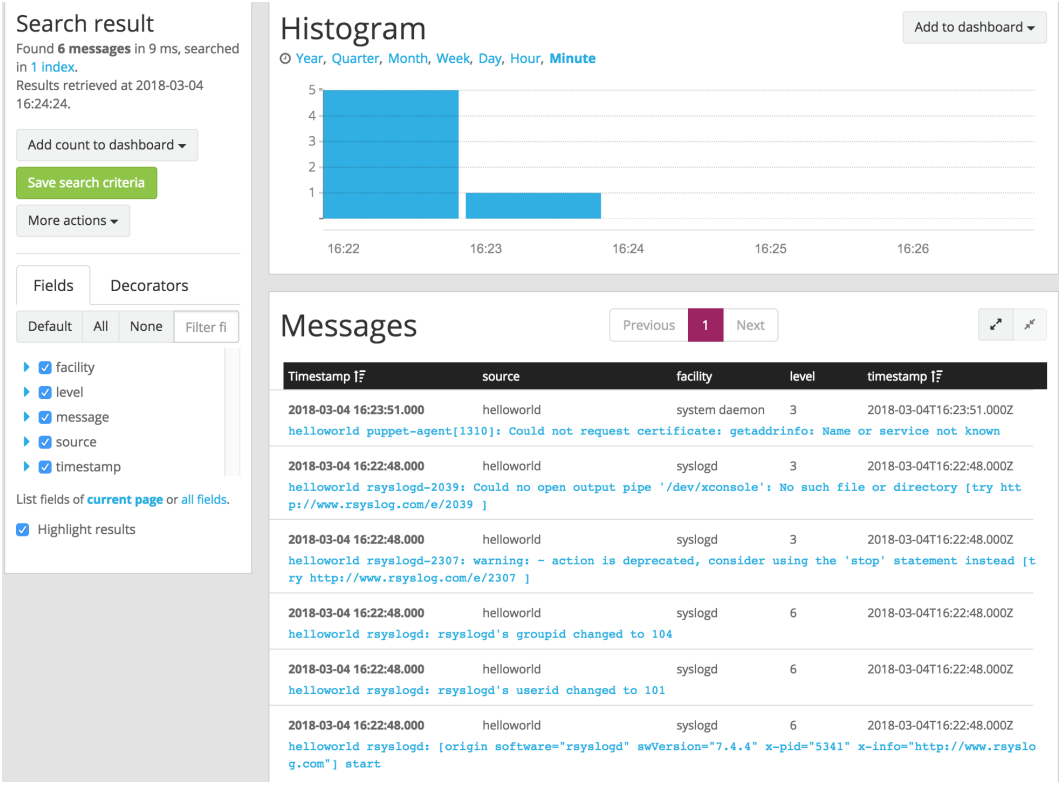
```
*.* @@10.0.0.10:12514;
```

Basically, this line means that we want to send every log message from every facility (`*.*`) using TCP as the transport protocol (`@@`) to server `10.0.0.10` on port `12514`.

Save it (`esc` : `wq` and hit `return`) and restart the Rsyslog service so that the changes take place:

```
vagrant@helloworld:~$ sudo service rsyslog restart
rsyslog stop/waiting
rsyslog start/running, process 5341
```

Return to the Graylog web interface, search for messages again and select all fields on the left sidebar:

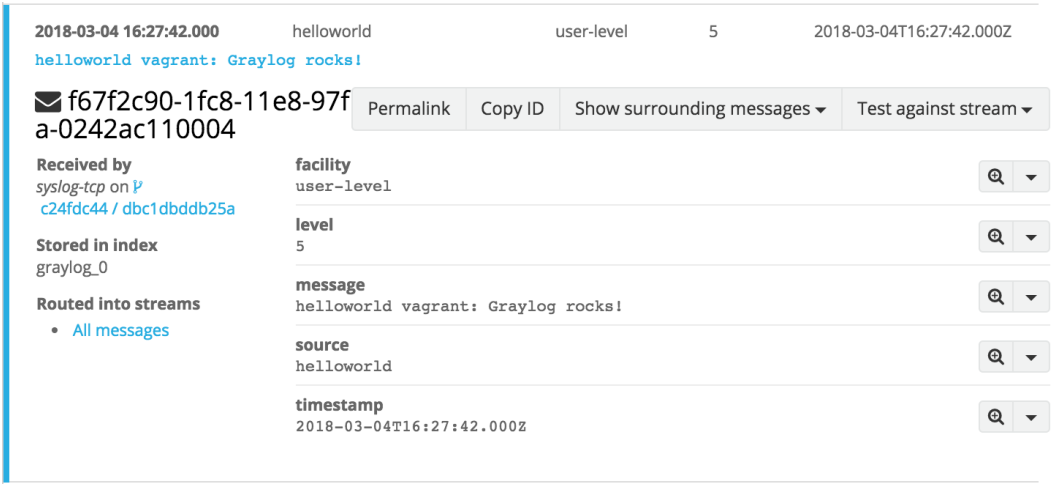


VM Logs

Awesome! The VM logs are now being sent to our Graylog instance. Use the `logger` shell utility to send a test message:

```
vagrant@helloworld:~$ logger "Graylog rocks!"
```

Search for messages again and click on the **Graylog rocks!** message to see its details:



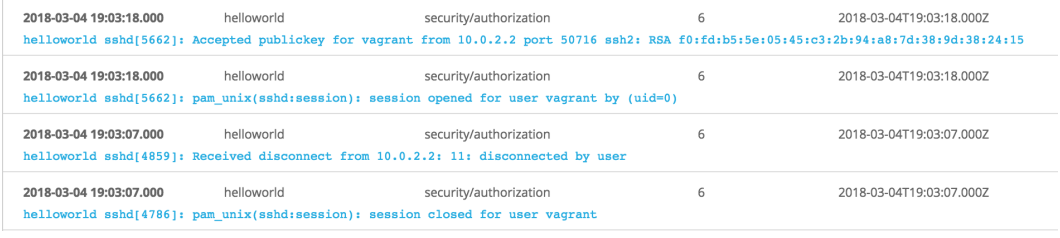
Message Details

Note that the `facility` for this message is `user-level` (which makes sense, doesn't it?) and the message was stored in Elasticsearch index `graylog_0`.

Now, log out the VM and log in again:

```
vagrant@helloworld:~$ exit
logout
Connection to 127.0.0.1 closed.

$ vagrant ssh vm_hello_world
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 3.13.0-141-generic x86_64)
```

Return to the Graylog web interface and search for messages again. You should start seeing `security/authorization` facility messages popping up:



sshd Messages

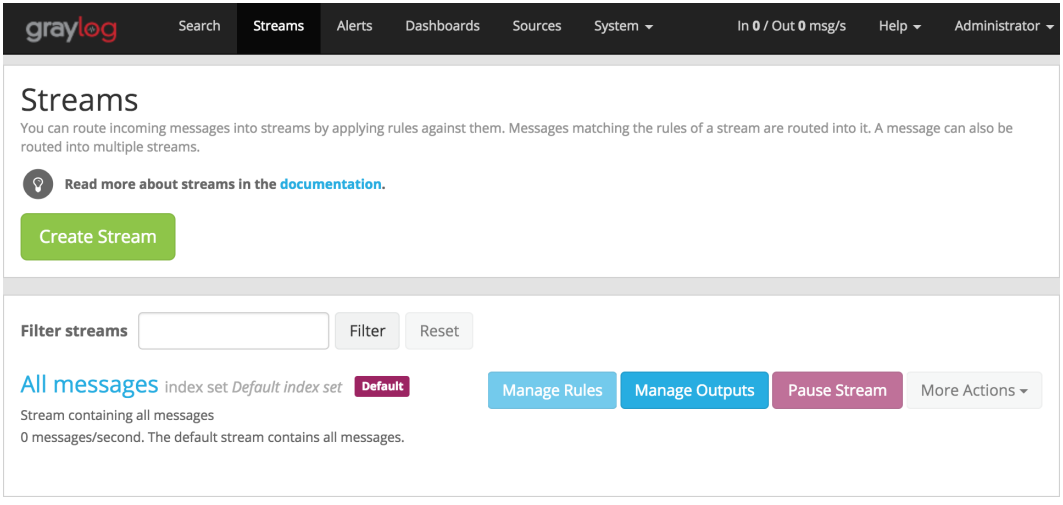
Awesome, isn't it? Even though some bad guy gets ssh access to your servers and try to wipe off the trail out of the filesystem, the logs would have already been sent to our centralized logging system!

Getting Alerted

Sounds reasonable to think that your production environment might be under attack when too many attempts to `ssh` your server happen within a small period of time, doesn't it? So, let's set up a Graylog alert to warn us when this kind of situation takes place.

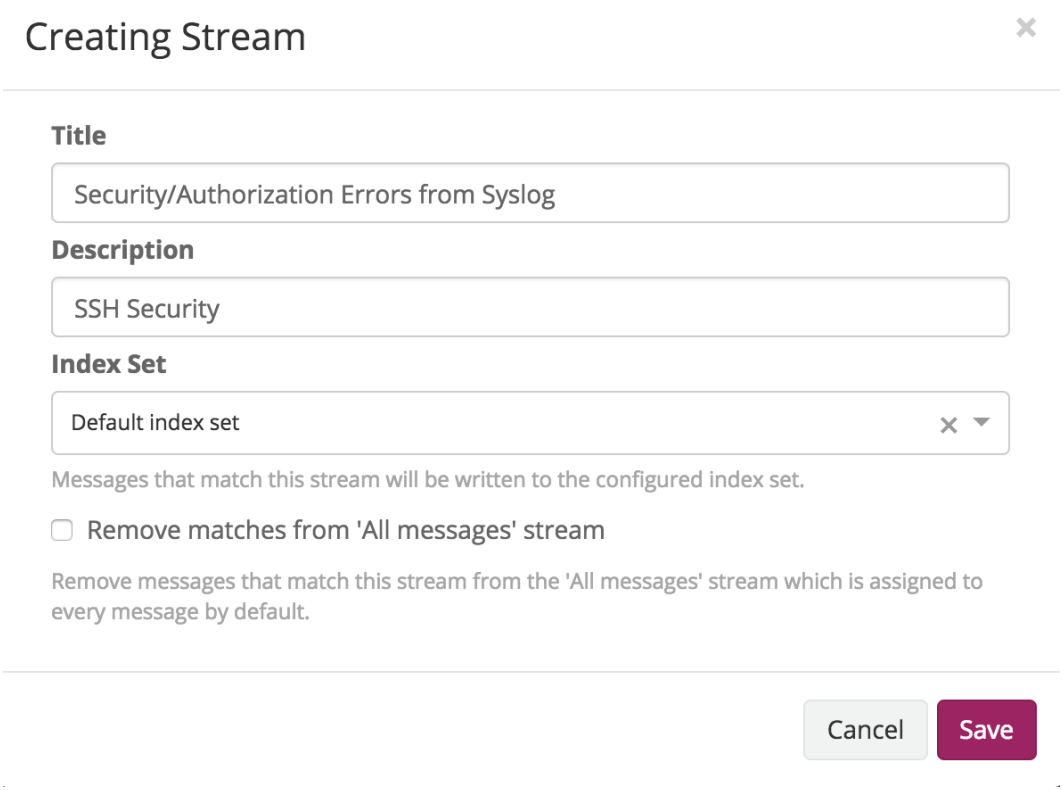
Streams

A stream is a mechanism to route log messages into specific categories (like "security-related messages") in real-time while they are processed. In order to selectively route the messages into the appropriate categories, you just have to define a set of rules. Let's see how that works in practice. On the top menu, click on **Streams**:



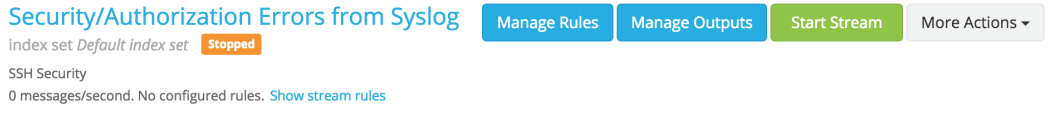
Streams

A stream named **All messages** is created by default. Essentially, all incoming messages go through this stream. However, if we need a specific stream that only shows messages related to attempts of `ssh` logins, we have to create it. Let's do that! Click on **Create Stream**, fill in the form as follows and save it:



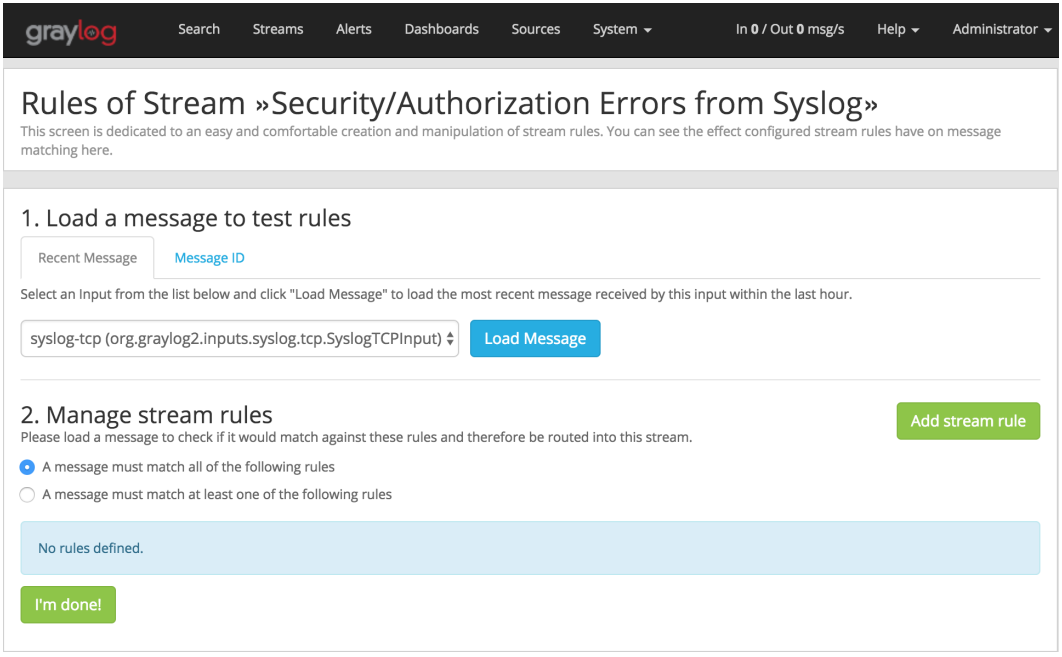
Security/Authorization Stream: Configuration

Our stream is created, but it's still stopped and with no rules associated.



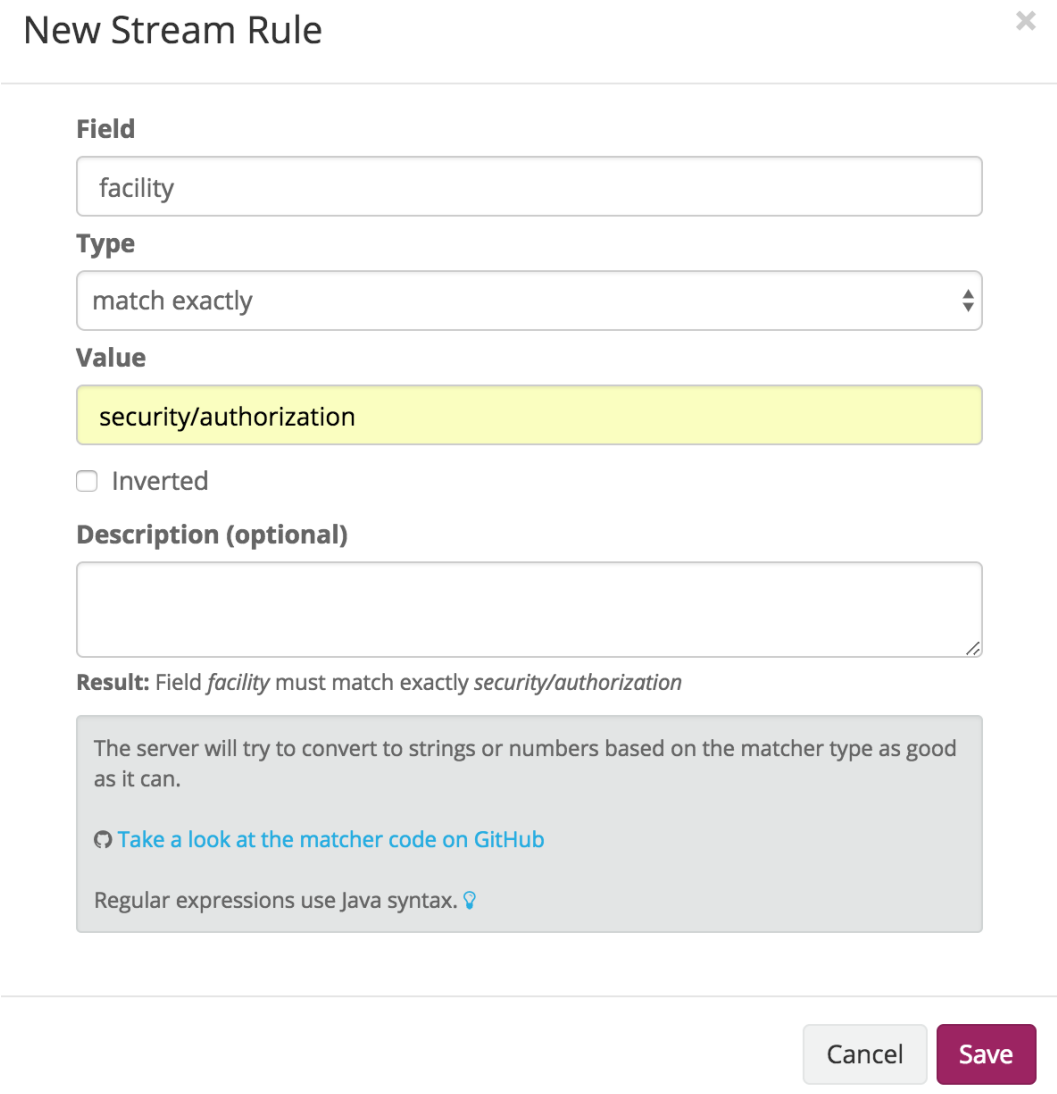
Security/Authorization Stream: Stopped and No Rules Associated

Let's create a rule that will filter the incoming messages to only pick the ones associated to `ssh` login attempts. Click on **Manage Rules** and select the previously created `syslog-tcp` input:



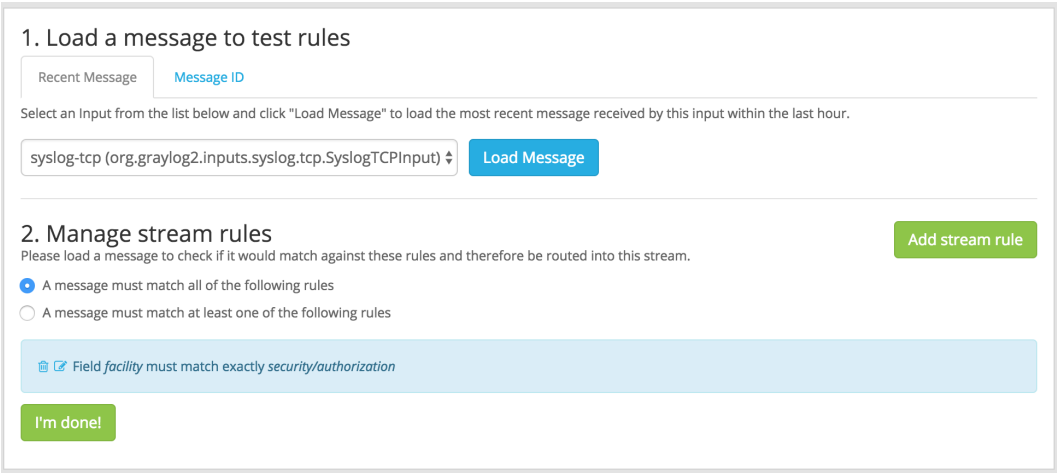
Rule Configuration: Input Syslog TCP

Now, click on **Add stream rule** and fill in the **Field** with `facility` and the **Value** with `security/authorization`, then click on **Save**:



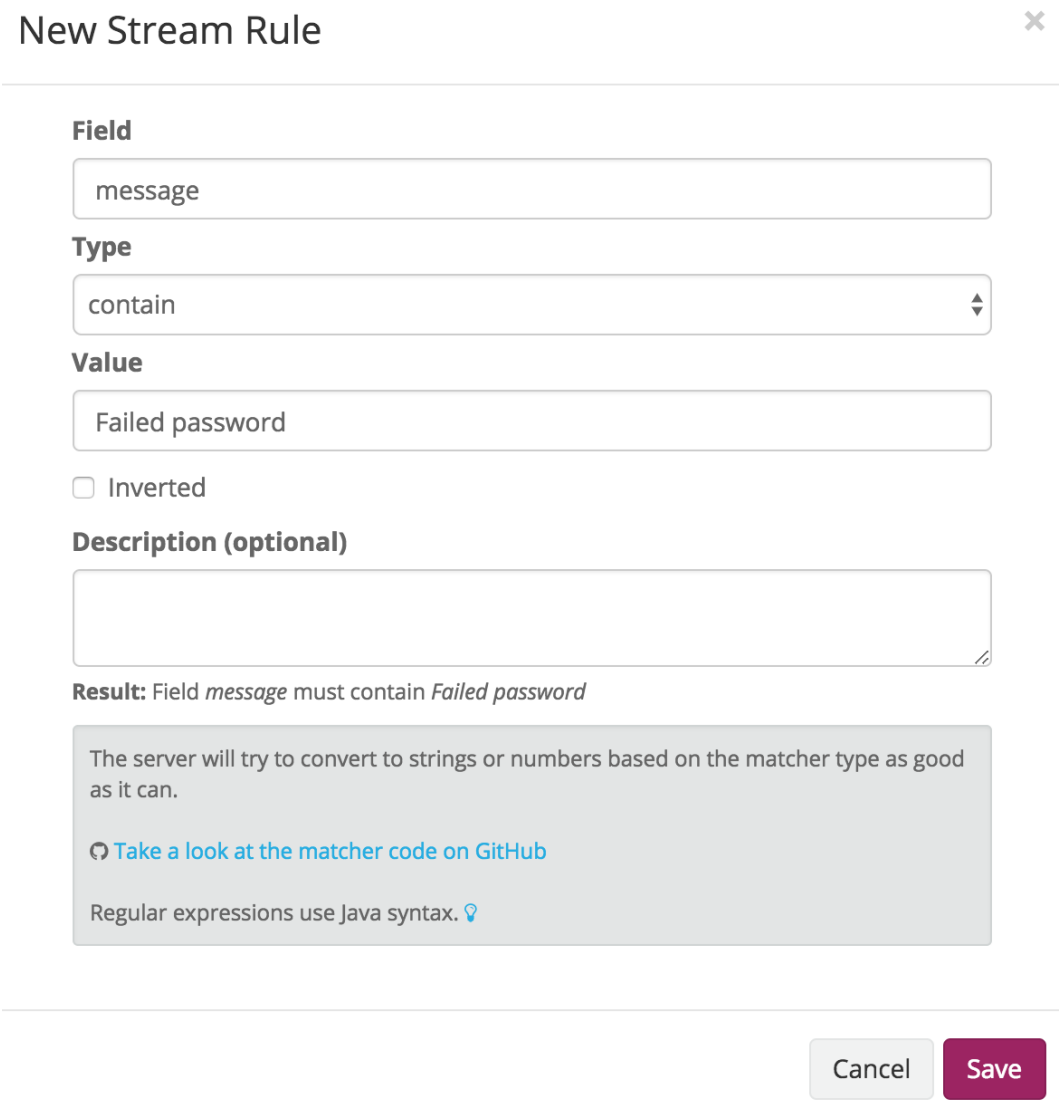
First Rule Configuration

You should see that a rule (`Field facility must match exactly security/authorization`) is created:



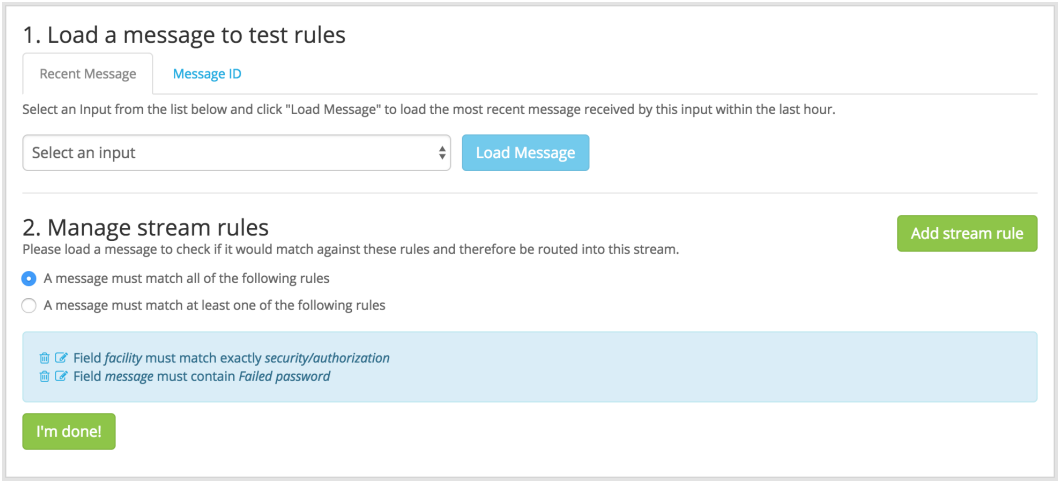
First Stream Rule

Click on **Add stream rule** again and fill in the **Field** with `message`, select **Type** `contain` and set the **Value** to `Failed password`:



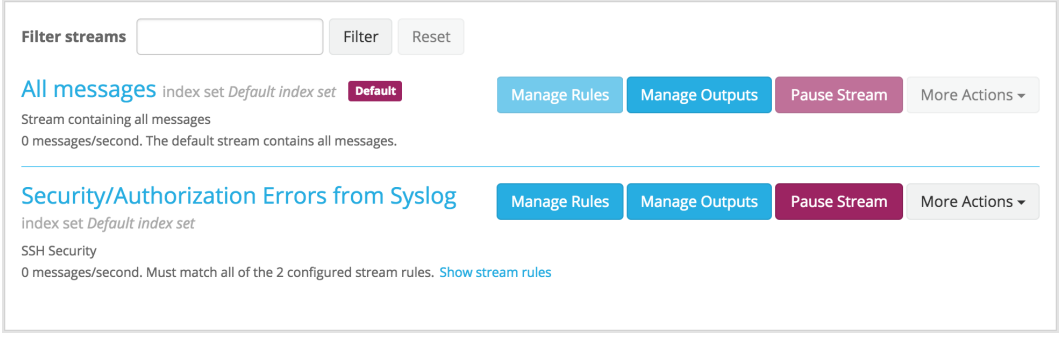
Second Rule Configuration

You should see that a second rule (`Field message must contain Failed password`) is created:



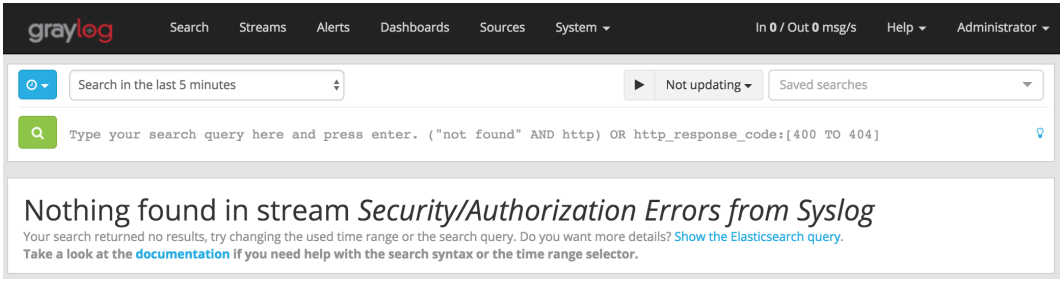
Stream Rules

Click on **I'm done** then click on **Start Stream** and you should end up with the stream running:



Security/Authorization Stream: Running

Click on the stream name **Security/Authorization Errors from Syslog** and note that it shows no messages:



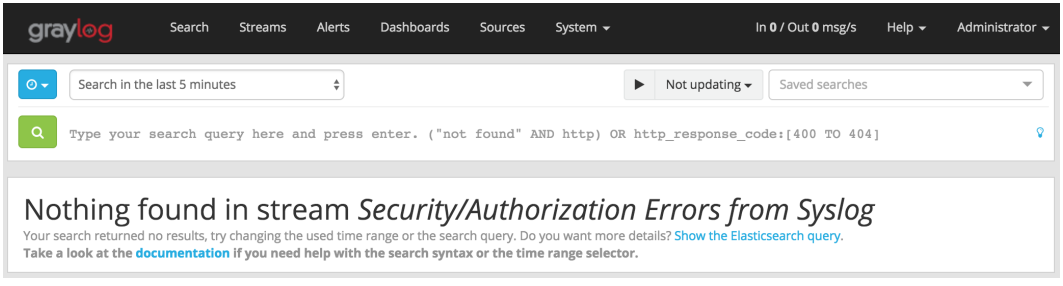
Security/Authorization Stream: No Messages Found

A stream does not consider already indexed messages. It just consider messages received after it's created and running.

Return to the VM, log out and then log in again:

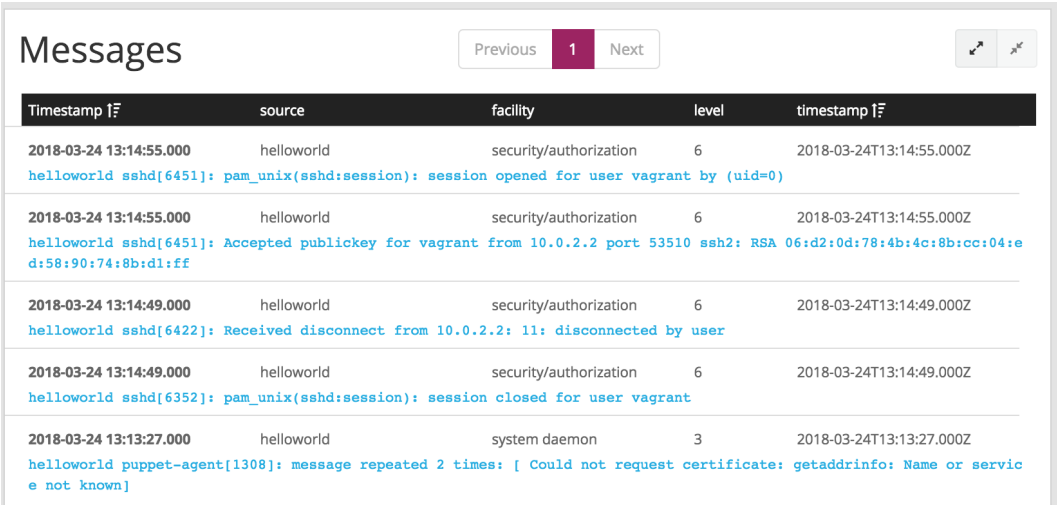
```
vagrant@helloworld:~$ exit
logout
Connection to 127.0.0.1 closed.

$ vagrant ssh vm_hello_world
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 3.13.0-141-generic x86_64)
```

Return to the Graylog web interface and search for messages in the newly created `Security/Authorization` stream again:



Security/Authorization Stream: No Messages Found

Since the login was successful, no messages are shown (our stream only filters by **failed** ssh attempts). However, if you click on **Streams** and select the stream **All messages**, you should see the `sshd` messages:

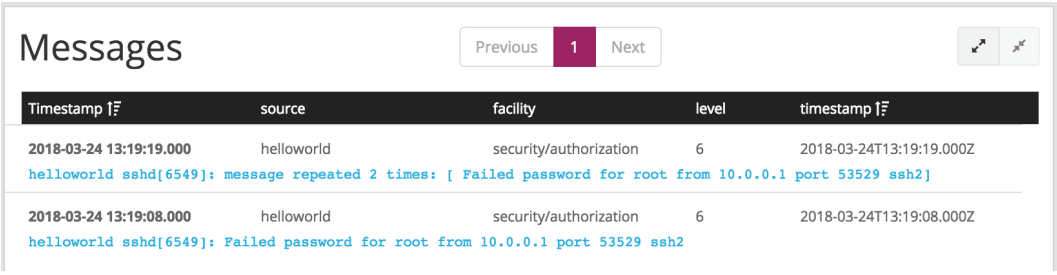


All Messages Stream: Messages Found

Return to the VM, log out and try to log in again with user `root` and empty password (just hit return):

```
vagrant@helloworld:~$ exit
logout
Connection to 127.0.0.1 closed.

$ ssh root@10.0.0.10
root@10.0.0.10's password:
Permission denied, please try again.
root@10.0.0.10's password:
Permission denied, please try again.
root@10.0.0.10's password:
root@10.0.0.10: Permission denied (publickey,password).
```

Select the stream **Security/Authorization** again and search for messages:



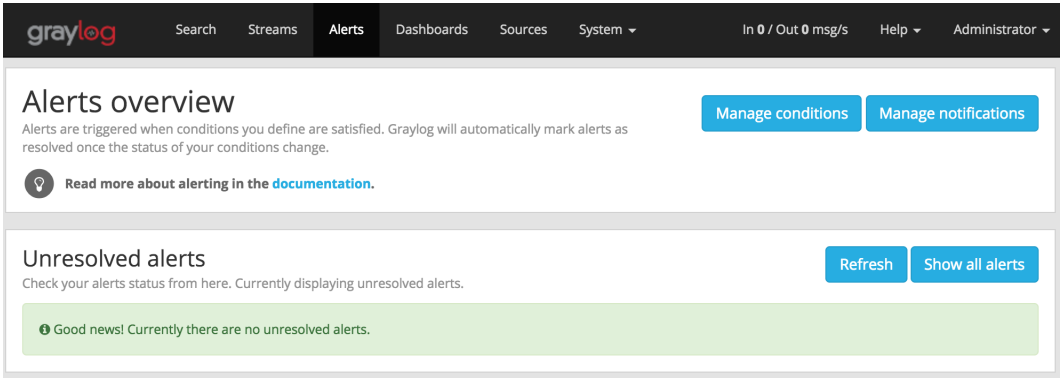
Security/Authorization Stream: Messages Found

Awesome! Our stream works like a charm. Time to have a coffee before moving to the next section.

Besides alerting on a stream, you could also output its messages to a 3rd party application or database.

Alert

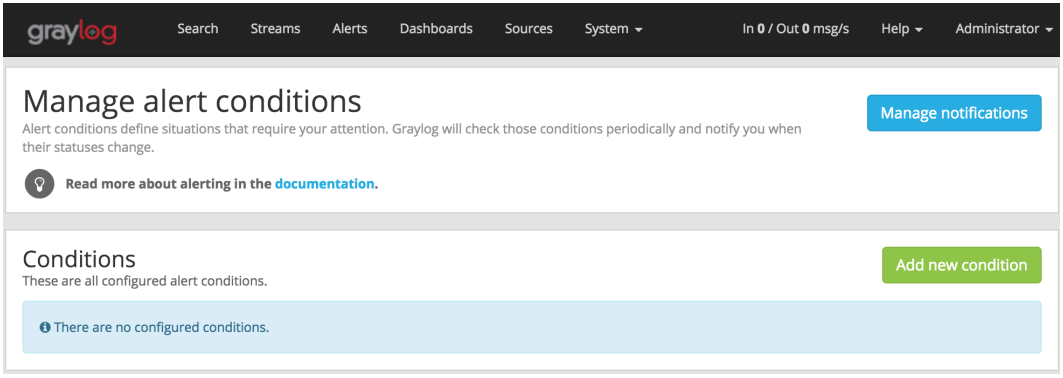
Now that we have the stream `Security/Authorization` working properly, it's time to create the alert (alerts are always based on streams). On the top menu, click on **Alerts**:



Alerts Overview Page

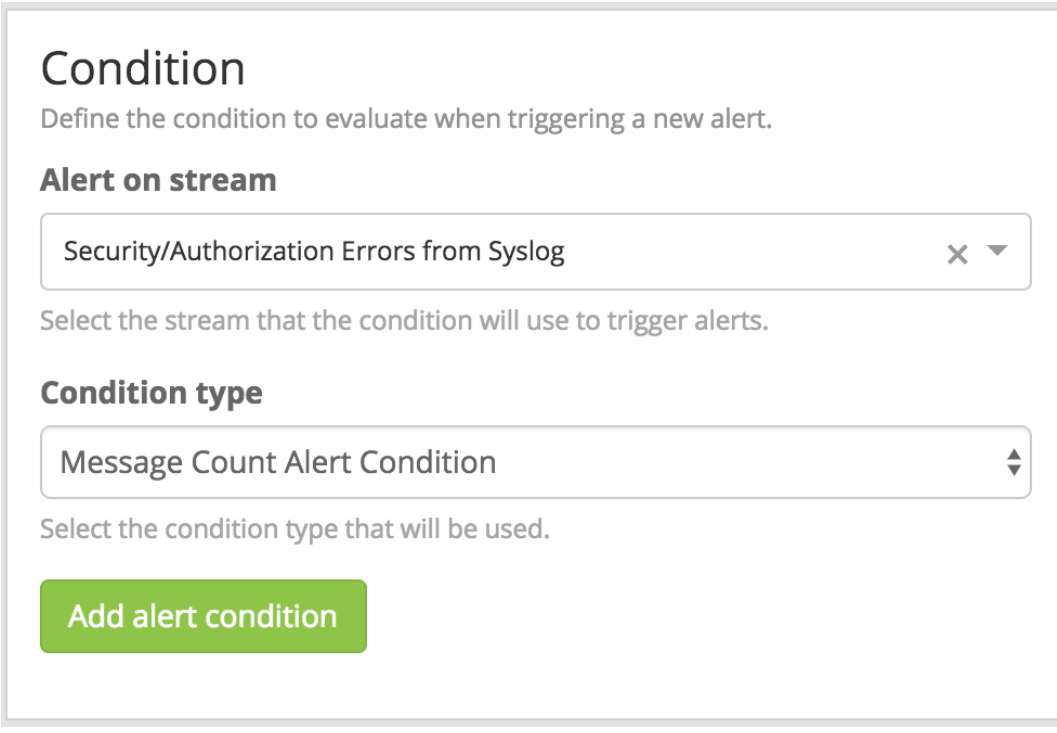
The alerts overview page lets you find out which alerts currently require your attention in an easy way, while also allowing you to check alerts that were triggered

in the past and are now resolved. Click on **Manage conditions** to define a condition that will trigger an alert:



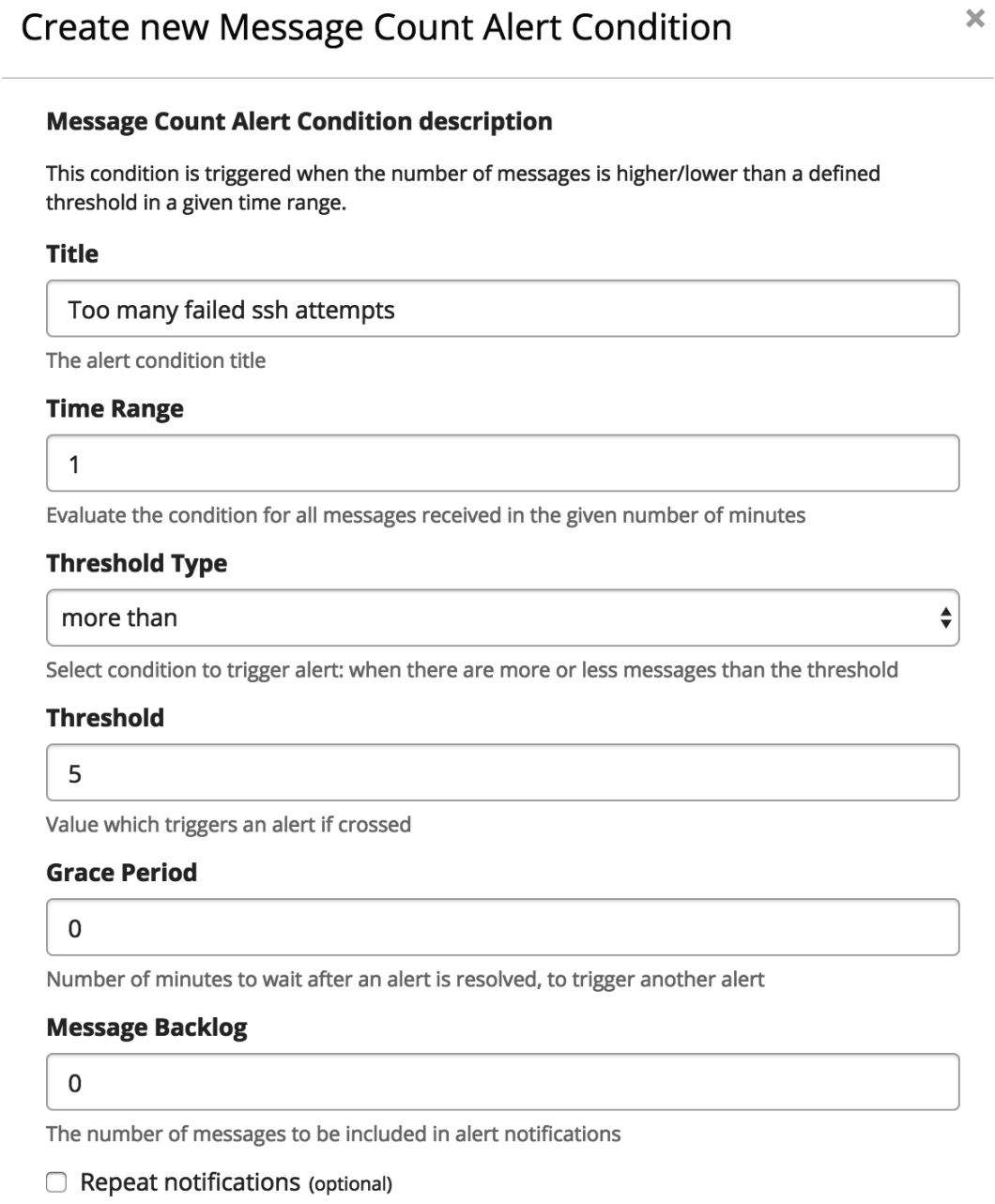
Manage Alert Conditions

Click on **Add new condition**, select our stream **Security/Authorization** and the **Condition type** `Message Count Alert Condition`:



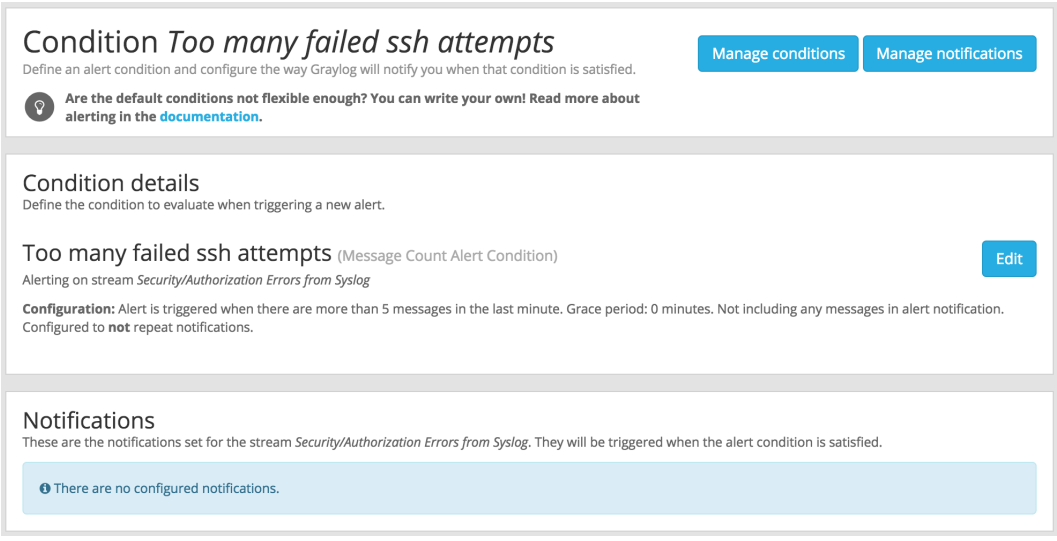
Condition Stream and Type

Click on **Add alert condition** and fill in the **Title** with `Too many failed ssh attempts`, **Time range** with `1` and **Threshold** `5`, meaning that if the `Security/Authorization` has more than 5 messages an alert should be triggered:



Message Count Alert Condition

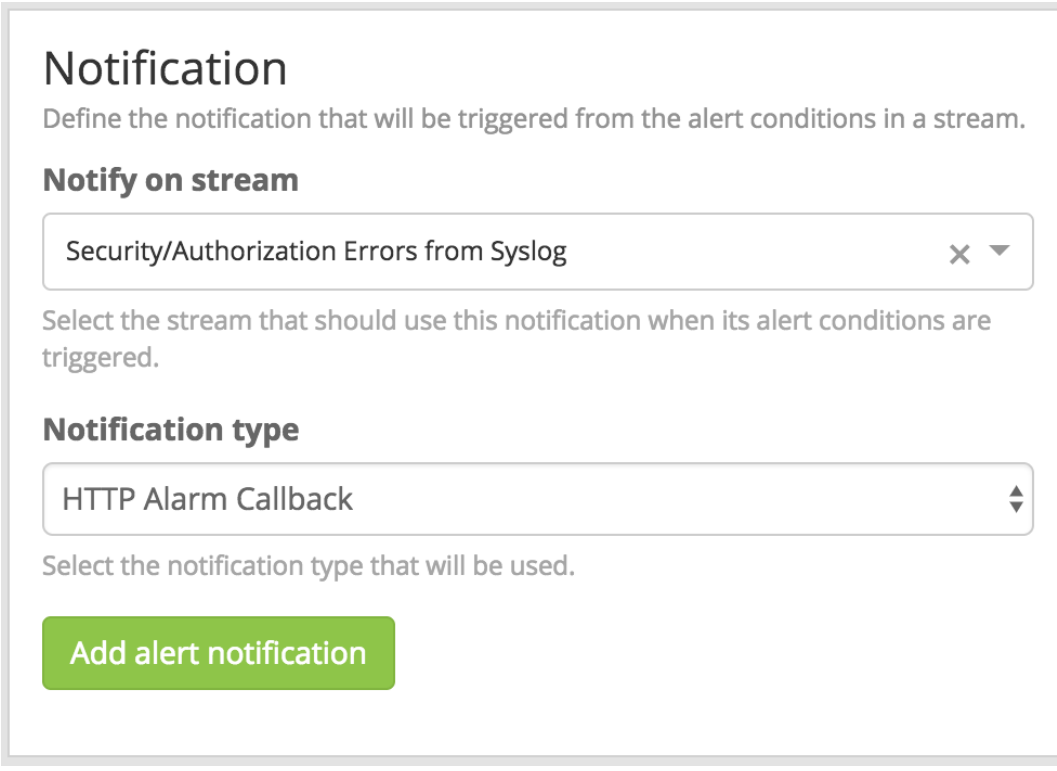
The condition was successfully created, but we still don't have any notifications associated with it:



Condition without Notification

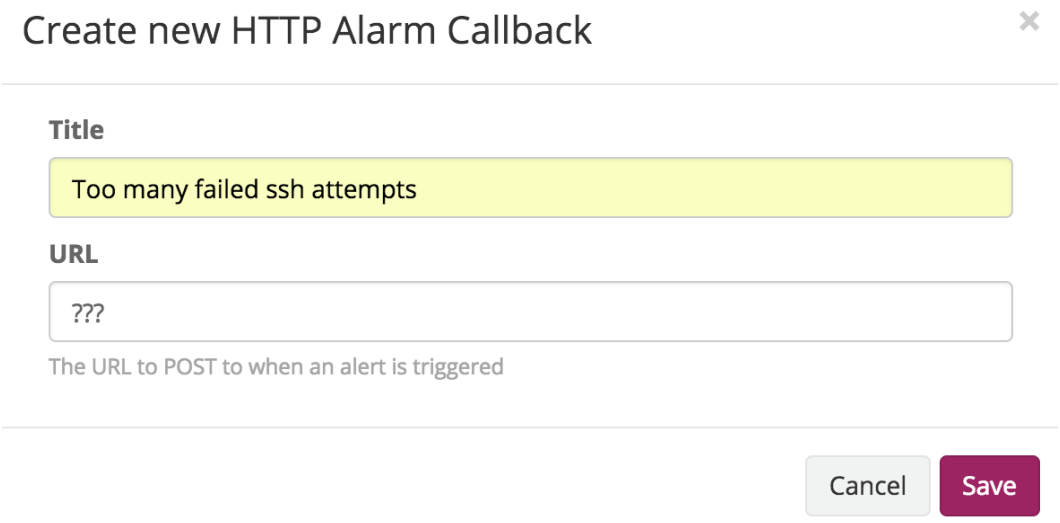
HTTP Notification

Let's create an HTTP notification and test it. Click on **Manage notifications** then **Add new notification**. Select the stream `Security/Authorization` for the field **Notify on stream** and `HTTP Alarm Callback` for the field **Notification type**:



Notification Stream and Type

Click on **Add alert notification** and give it the **Title** `Failed ssh attempts callback` and the URL… Well, we don't have a URL endpoint to capture this callback:



Alert Notification: No Callback URL

Let's set up our own RequestBin[24] instance and create a mock endpoint first.

Go to your terminal, clone and `cd` the RequestBin repository:

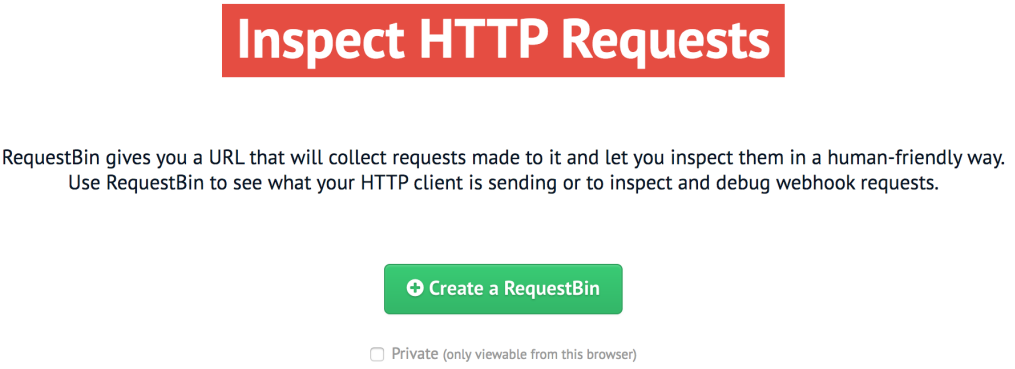
```
$ git clone git://github.com/Runscope/requestbin.git && cd requestbin
```

Use Docker Compose[25] to build the Docker image and start the containers:

```
$ docker-compose build
$ docker-compose up -d
```

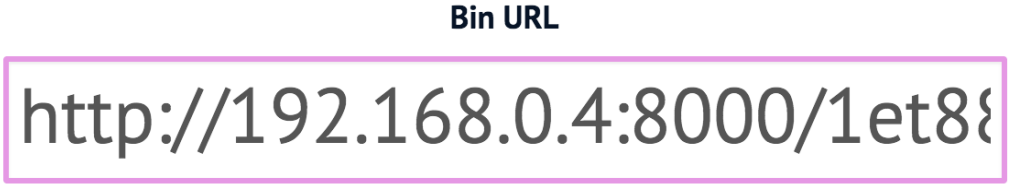
Check your local IP address with `ifconfig`, open another browser window and navigate to `http://YOUR_IP_ADDRESS:8000`:

[24] https://github.com/Runscope/requestbin
[25] https://docs.docker.com/compose/



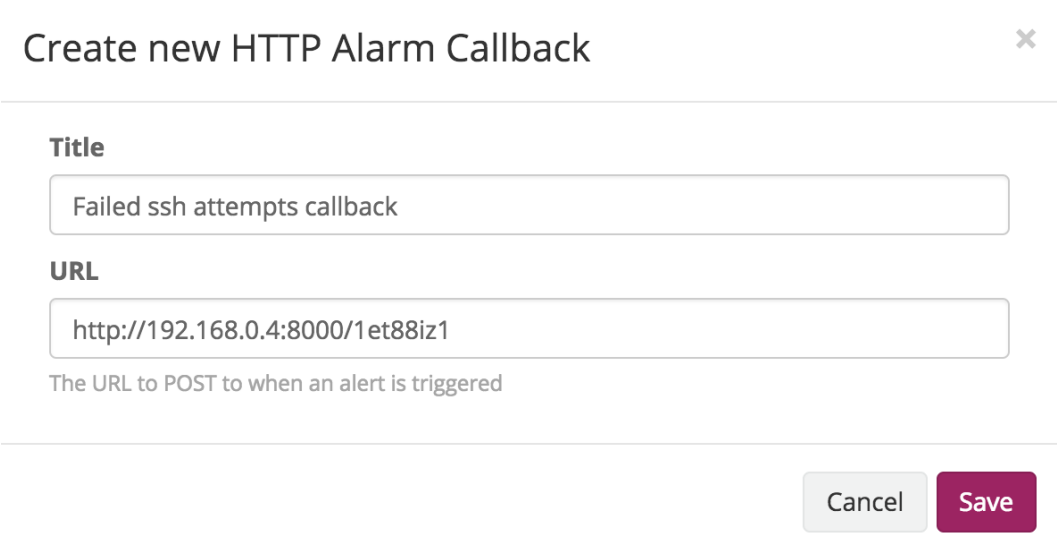
RequestBin

Click on **Create a RequestBin** and copy your **Bin URL**:



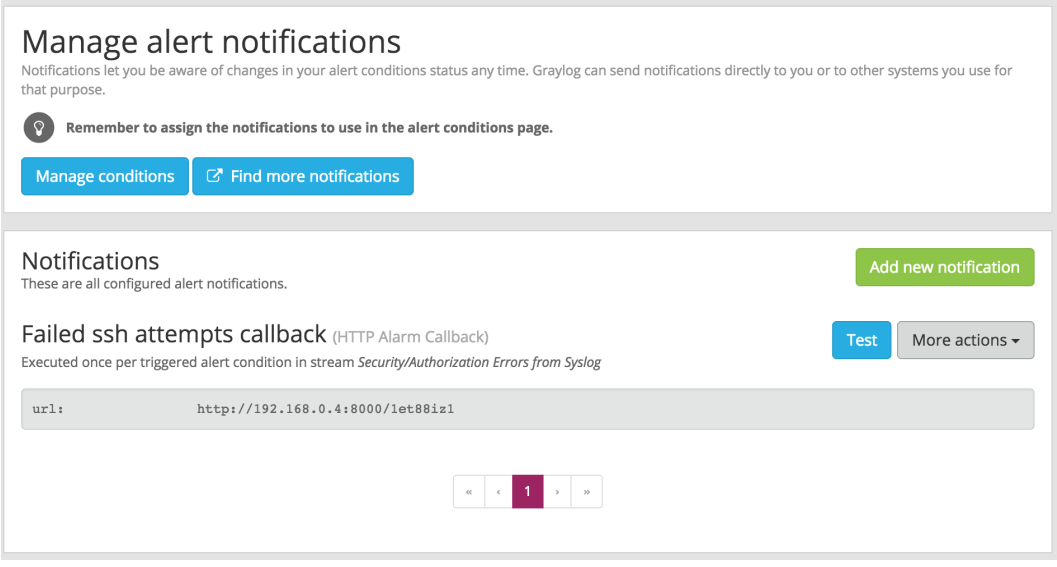
Bin URL

Back to the Graylog HTTP Alarm Callback, paste **your URL**:



HTTP Alarm Callback Configuration

After saving it you should see the notification created:

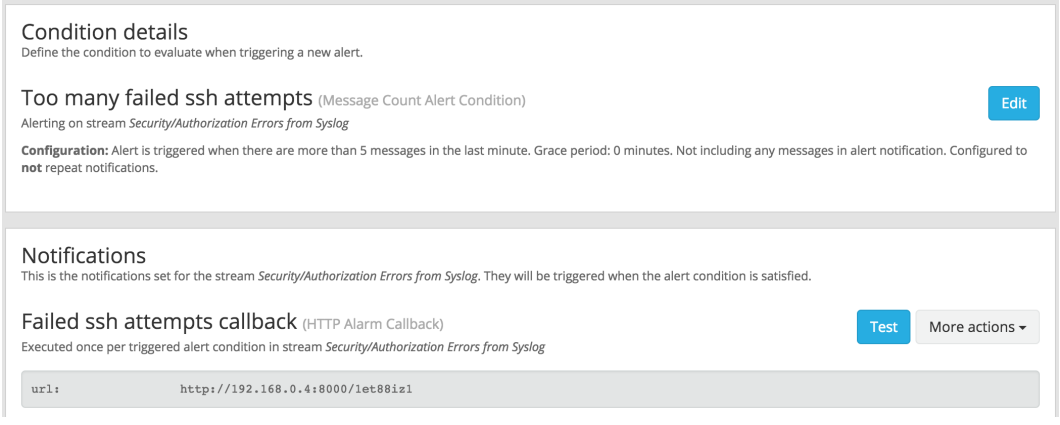


Notification: Failed ssh attempts

Click on **Test** and refresh the RequestBin window:

RequestBin: Callback Received

Here we go! We were able to receive the HTTP callback - our notification is working! Back to Graylog, click on **Manage Conditions** and select the already created **Too many failed ssh attempts** condition. You should see that the condition and the notification are associated:

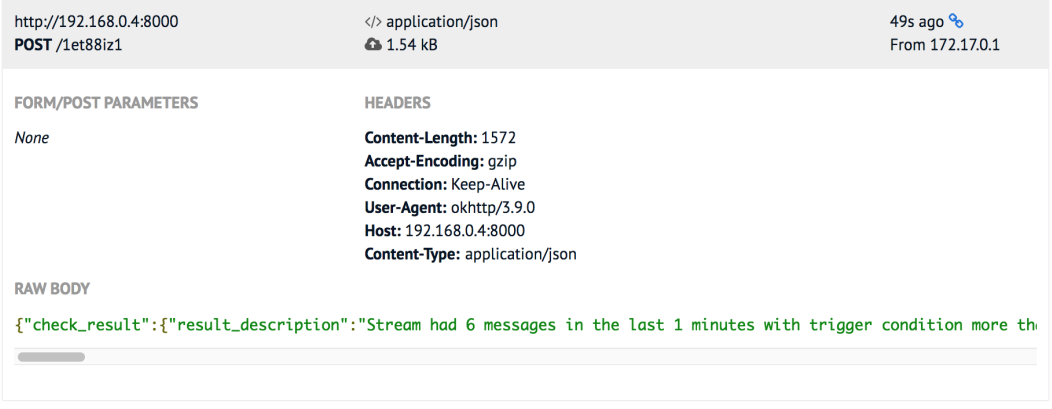


Condition and Alert Associated

Time to test it. Return to your terminal and try to log in to the VM using user `root` and empty password for many times within the same minute:

```
$ ssh root@10.0.0.10
root@10.0.0.10's password:
Permission denied, please try again.
root@10.0.0.10's password:
Permission denied, please try again.
root@10.0.0.10's password:
root@10.0.0.10: Permission denied (publickey,password).

$ ssh root@10.0.0.10
root@10.0.0.10's password:
Permission denied, please try again.
root@10.0.0.10's password:
Permission denied, please try again.
root@10.0.0.10's password:
root@10.0.0.10: Permission denied (publickey,password).

$ ssh root@10.0.0.10
root@10.0.0.10's password:
Permission denied, please try again.
root@10.0.0.10's password:
Permission denied, please try again.
root@10.0.0.10's password:
root@10.0.0.10: Permission denied (publickey,password).
```

After a while, refresh the RequestBin window and you should see the captured callback:



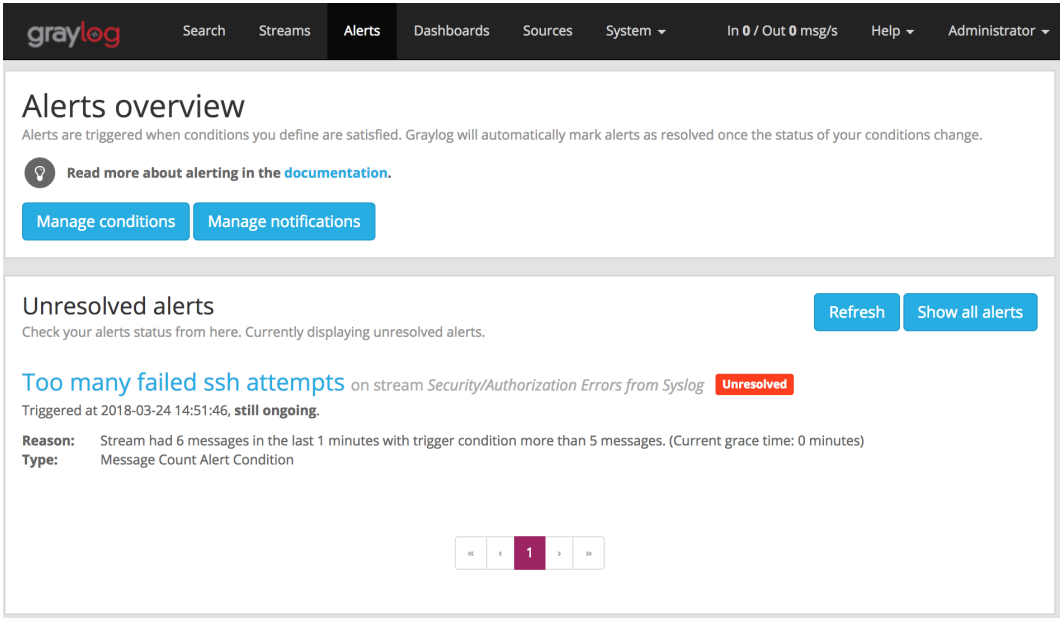
RequestBin: Captured Callback

Just out of curiosity, below is the entire JSON that represents the callback:


```
{
        "check_result": {
                "result_description": "Stream had 6 messages in the last 1 minutes with trigger co
dition more than 5 messages. (Current grace time: 0 minutes)",
                "triggered_condition": {
                        "id": "8facea99-01e7-489c-a0a7-946dd5687935",
                        "type": "message_count",
                        "created_at": "2018-03-24T14:00:04.035Z",
                        "creator_user_id": "admin",
                        "title": "Too many failed ssh attempts",
                        "parameters": {
                                "backlog": 0,
                                "repeat_notifications": false,
                                "grace": 0,
                                "threshold_type": "MORE",
                                "threshold": 5,
                                "time": 1
                        }
                },
                "triggered_at": "2018-03-24T14:46:46.932Z",
                "triggered": true,
                "matching_messages": []
        },
        "stream": {
                "creator_user_id": "admin",
                "outputs": [],
                "description": "SSH Security",
                "created_at": "2018-03-24T11:39:01.362Z",
                "rules": [{
                        "field": "facility",
                        "stream_id": "5ab638d54cedfd000118fc18",
                        "description": "",
                        "id": "5ab638fe4cedfd000118fc47",
                        "type": 1,
                        "inverted": false,
                        "value": "security/authorization"
                }, {
                        "field": "message",
                        "stream_id": "5ab638d54cedfd000118fc18",
                        "description": "",
                        "id": "5ab64b614cedfd0001191038",
```

```
                        "type": 6,
                        "inverted": false,
                        "value": "Failed password"
                }],
                "alert_conditions": [{
                        "creator_user_id": "admin",
                        "created_at": "2018-03-24T14:00:04.035Z",
                        "id": "8facea99-01e7-489c-a0a7-946dd5687935",
                        "type": "message_count",
                        "title": "Too many failed ssh attempts",
                        "parameters": {
                                "backlog": 0,
                                "repeat_notifications": false,
                                "grace": 0,
                                "threshold_type": "MORE",
                                "threshold": 5,
                                "time": 1
                        }
                }],
                "title": "Security/Authorization Errors from Syslog",
                "content_pack": null,
                "is_default_stream": false,
                "index_set_id": "5ab6371a4cedfd000118f9cf",
                "matching_type": "AND",
                "remove_matches_from_default_stream": false,
                "disabled": false,
                "id": "5ab638d54cedfd000118fc18"
        }
}
```


Back to Graylog, click on **Alerts** on the top menu and you should see that there is an unresolved alert:



Unresolved Alert

Click on the **Too many failed ssh attempts** condition to see the alarm details like the messages in the stream by the time the alert was triggered:

Messages Evaluated

If at this exact moment you still don't feel that Graylog is great, please close this book and go find your doctor!

Kill the RequestBin containers and let's move on:

```
$ docker-compose down
```

Email Notification using Amazon SES

Appendices

Introduction to Docker

Let's forget for a while that this is a book about Graylog so that we can focus on learning Docker.

Have you ever heard the sentence "I don't know what is happening; it works on my machine"? What about the famous "this version was working perfectly on staging but it's not working on production"? One of the main reasons why these issues happen is because environments often become different over time, especially those managed manually. People are often installing different libraries on different versions in different environments (which can also run different operating systems, by the way). The result is that, for example, the staging environment is no longer a mirror from production over time. If the staging is not a mirror from production anymore, of course that the application running on these environments could behave different on each environment, what leads to bugs that happen in an environment but not in another.

In short, Docker allows you to easily run services (Graylog, Elasticsearch, MongoDB, MySQL, Jenkins, Nginx, your own application, etc) on a machine. Docker guarantees that these services will always be in the same state across executions, regardless of the underlying operating system or system libraries. In other words, if your staging environment is very different from your production environment (system libraries, operating systems, etc) and you run an application as a Docker container on both environments, it's guaranteed that the application would behave exactly the same on these environments regardless of their differences.

In my humble opinion, this is the most important benefit got from using Docker, but there are many more:

- It's easy to run services as Docker containers. Thus, it also helps a lot in the development phase because you don't have to waste time installing and configuring tools on your operating system.
- Docker is a highly collaborative tool. You can reuse Docker images that people build and share publicly.

- It encourages the infrastructure as code model because a Docker image is entirely described on a file called a Dockerfile that can (and should) be versioned in the source control.
- Docker has a great community, and it's expanding quickly.

Difference Between Container and Image

Docker images are binary files that contain everything needed to run a specific service. When you instantiate a service from a Docker image, you say that you create a Docker container. As an analogy, if a Docker image is a Java class, then a Docker container is an object. Many containers can be created from a single image.

Image vs Container

As the figure above shows, an image is like a house scheme whereas the container is the concrete house where people actually live in.

Technically speaking, a container is a group of processes contained in an isolated environment, but running on the same kernel as the host operating system. This isolation is provided by concepts like `cgroups` and `namespaces`. For example, if you create a file inside a container, this file cannot be accessed from the host operating system (unless you explicitly specify this).

Docker images can be either created from a container state (like a snapshot of a running container) or describing the image state (like the operating system, libraries and applications) on a special file called `Dockerfile`. Both work, but which way do you think it's better? That's right, the `Dockerfile` way, because it encourages the infrastructure as code model, that is, the image state is easily tracked using a source control and safely evolves as every single change would go through a pull request, code review, and so on.

You create a Docker image from a `Dockerfile` using the **docker image build** command and create a Docker container by executing the **docker container run** command.

Docker Hub

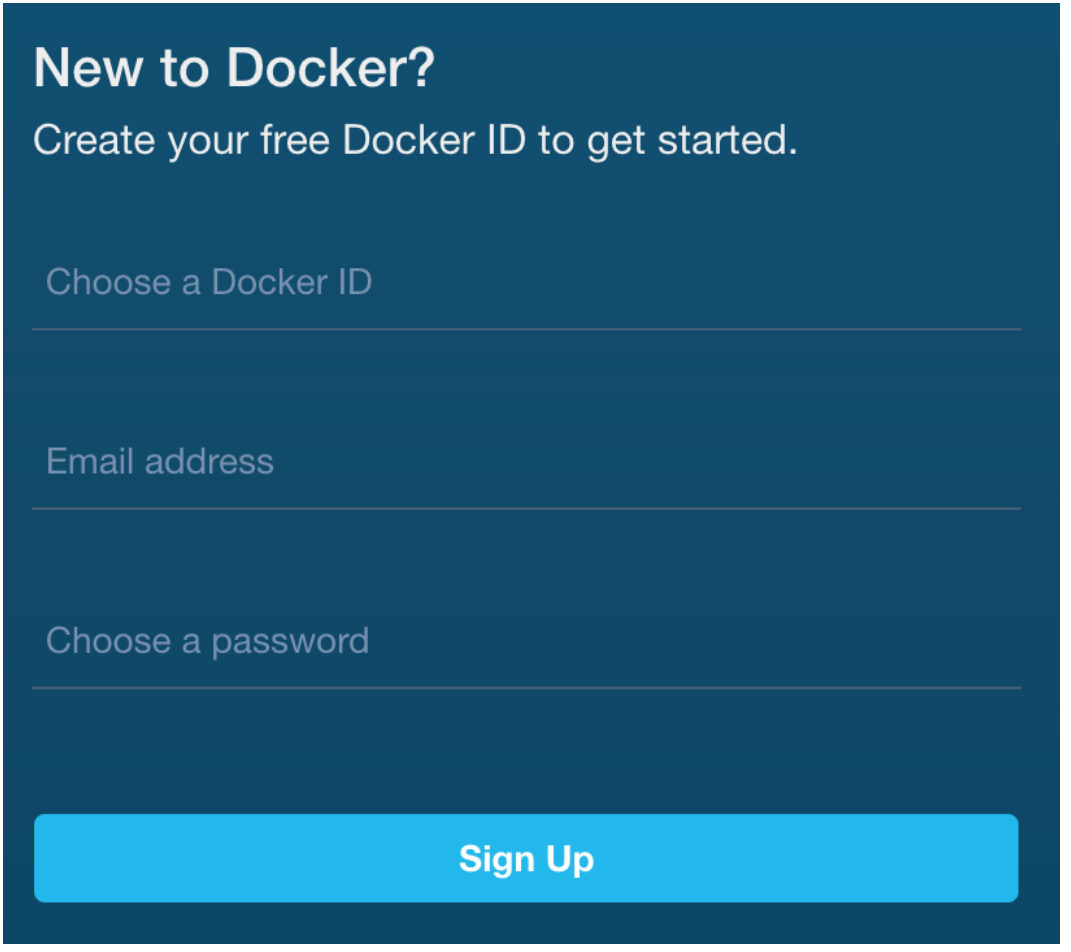
I have mentioned that Docker is a highly collaborative tool as you can reuse Docker images that people build and share publicly. There is a public image marketplace available where you can find almost every service you need already wrapped into a pre-built Docker image: the Docker Hub[26].

Anyone can register at Docker Hub and publish images publicly (it's also possible to publish private Docker images, but you must pay for this feature if you want to publish more than one private image).

[26] https://hub.docker.com

Create your Account

As soon as you access the Docker Hub[27], just fill in the form with your Docker ID, email address and a password.

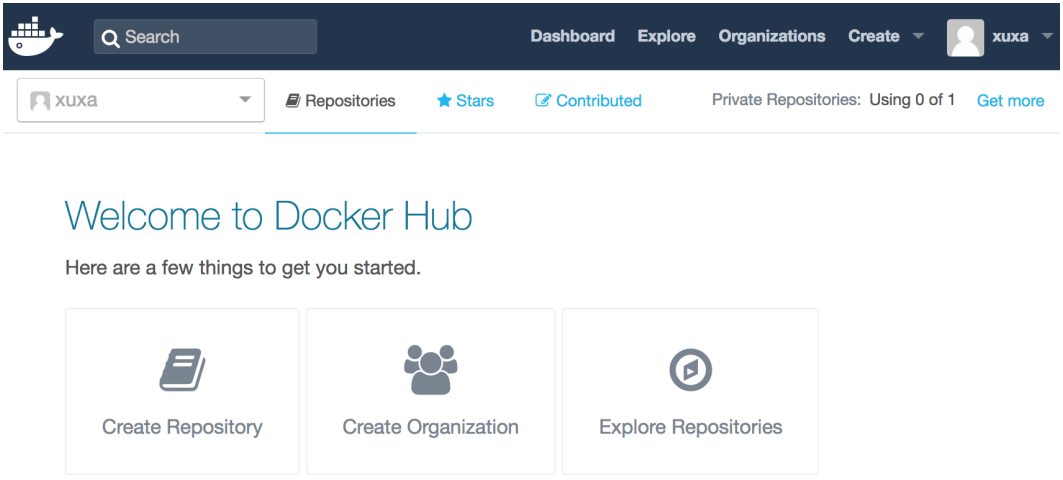


Docker Hub New Account Form

You will receive an email from Docker with an account confirmation link. After you confirm your account, access Docker Hub again and sign in providing your Docker

[27] https://hub.docker.com/

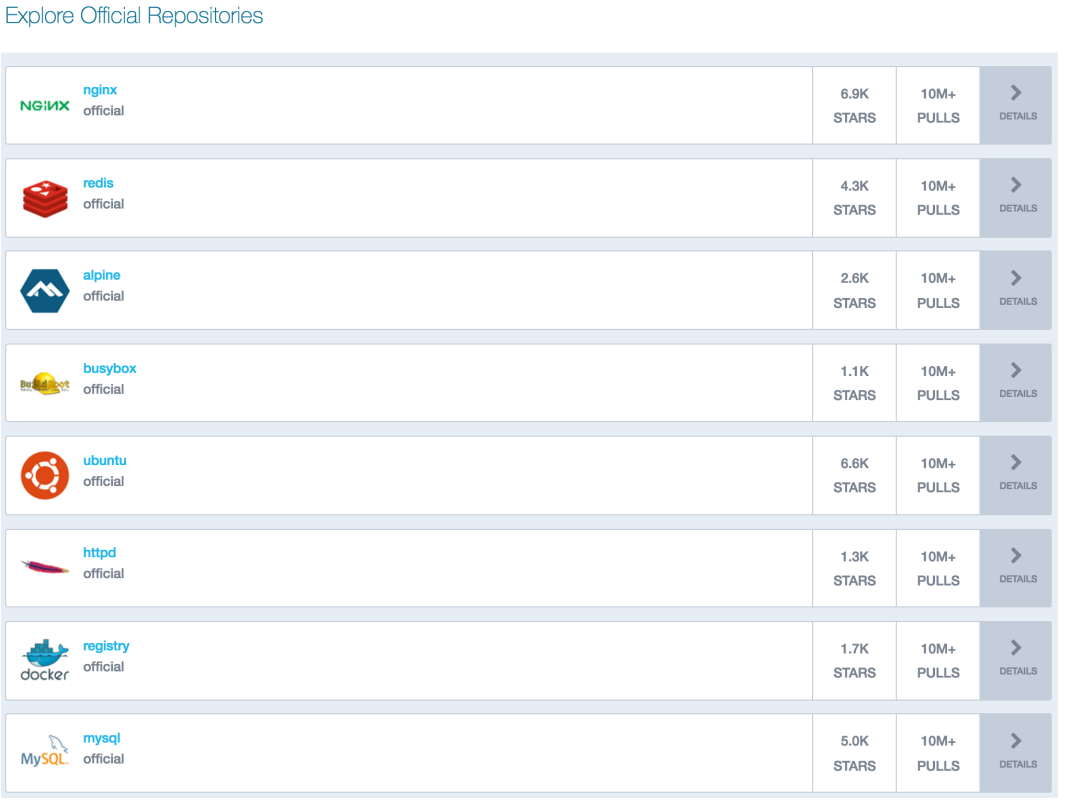
Id and password. You should your dashboard like this:



Dashboard First Access

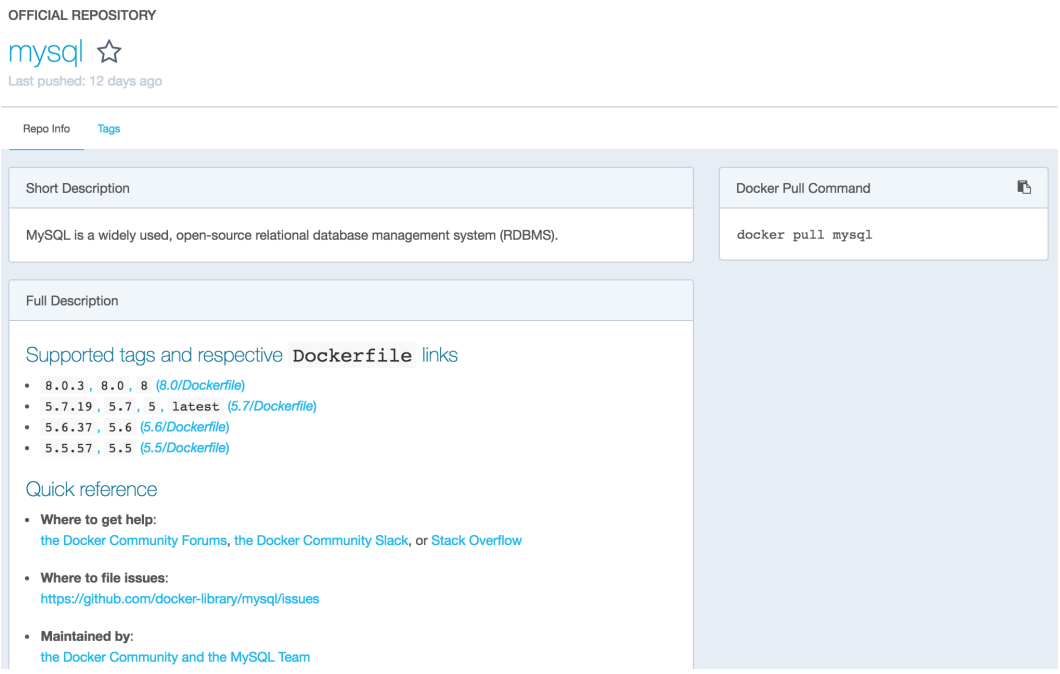
Official Docker Repositories

After you sign in, click on “Explore Repositories”. You should see a list of official Docker repositories ordered by stars and pulls.



Official Docker Repositories

There are much more official repositories than what is shown in the figure. Plus, note that MySQL has an official repository.



MySQL Official Docker Image

Generally, the Docker images on Docker Hub provide a good description on how to use it, which environment variables are allowed, which parameters can be passed to the `docker container run` command, and so on.

Image Tags

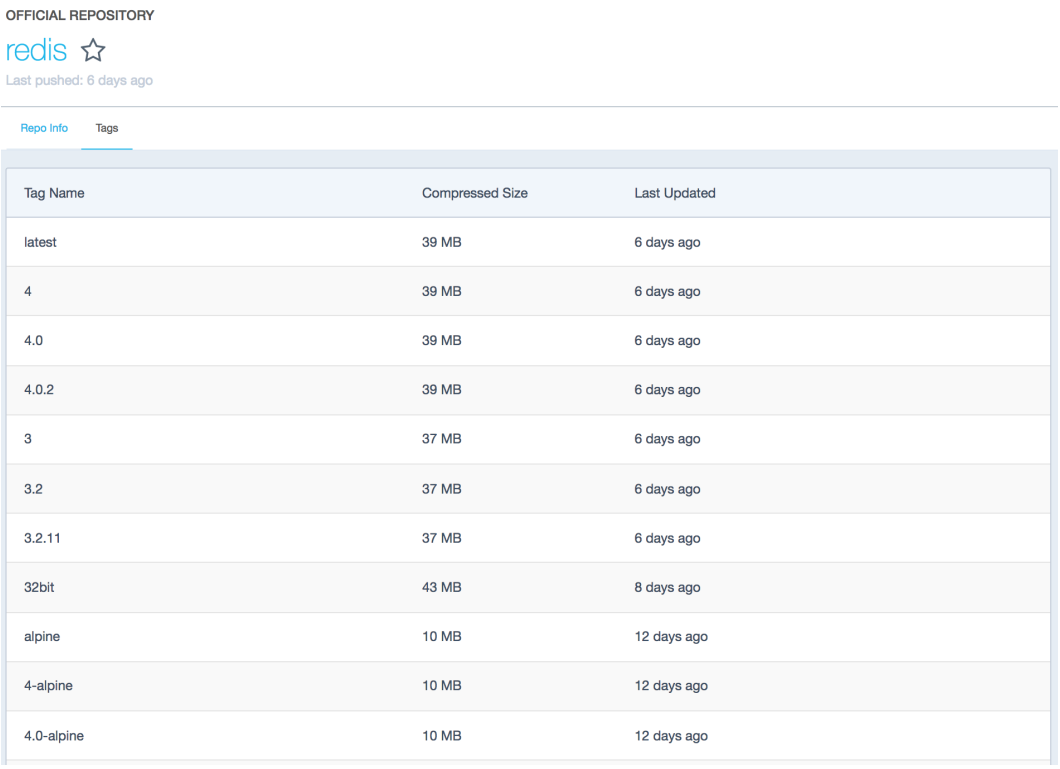
Creating a MySQL database container on Docker is as easy as executing the following command:

```
$ docker container run -d --name mysql -e MYSQL_DATABASE=my-database -e MYSQL_ROOT_PA\
SSWORD=root -p 3306:3306 mysql:5.7
```

The last instruction in this command (`mysql:5.7`) is the **image name** followed by `:` and the **image tag**. Basically, this is the general syntax:

```
$ docker container run image_name:image_tag
```

An image can contains many tags. Tags are usually used to provide different service versions, such as the Redis official image below:


OFFICIAL REPOSITORY

redis ☆

Last pushed: 6 days ago

Repo Info Tags

Tag Name	Compressed Size	Last Updated
latest	39 MB	6 days ago
4	39 MB	6 days ago
4.0	39 MB	6 days ago
4.0.2	39 MB	6 days ago
3	37 MB	6 days ago
3.2	37 MB	6 days ago
3.2.11	37 MB	6 days ago
32bit	43 MB	8 days ago
alpine	10 MB	12 days ago
4-alpine	10 MB	12 days ago
4.0-alpine	10 MB	12 days ago

Redis Image Tags

If a tag is not provided when building a Docker image, the `latest` tag will be bound to it. When you don't provide a tag in the syntax shown above, Docker will interpret that you want the `latest` tag. A common misunderstanding is that the latest tag means the "newest image version available," but this may not be true. As I said, the latest tag is just a tag that is used when you don't provide any other while you are building a Docker image; it doesnâ€™t mean that it's the newest service version.

Note the `alpine` tag in the Redis image. Alpine Linux is a security-oriented, lightweight Linux distribution based on `musl`, `libc` and `busybox`, which makes it smaller and more resource efficient than traditional GNU/Linux distributions. It's very common to see images with an alpine tag.

A tag name must be valid ASCII and may contain lowercase and uppercase letters, digits, underscores, periods and dashes. A tag name may not start with a period or a dash and may contain a maximum of 128 characters. So, although it's pretty common to use to provide different service versions, you could actually use it many other different ways, such as to mark a specific git commit hash or even a specific git branch.

Pulling Images From a Different Docker Registry

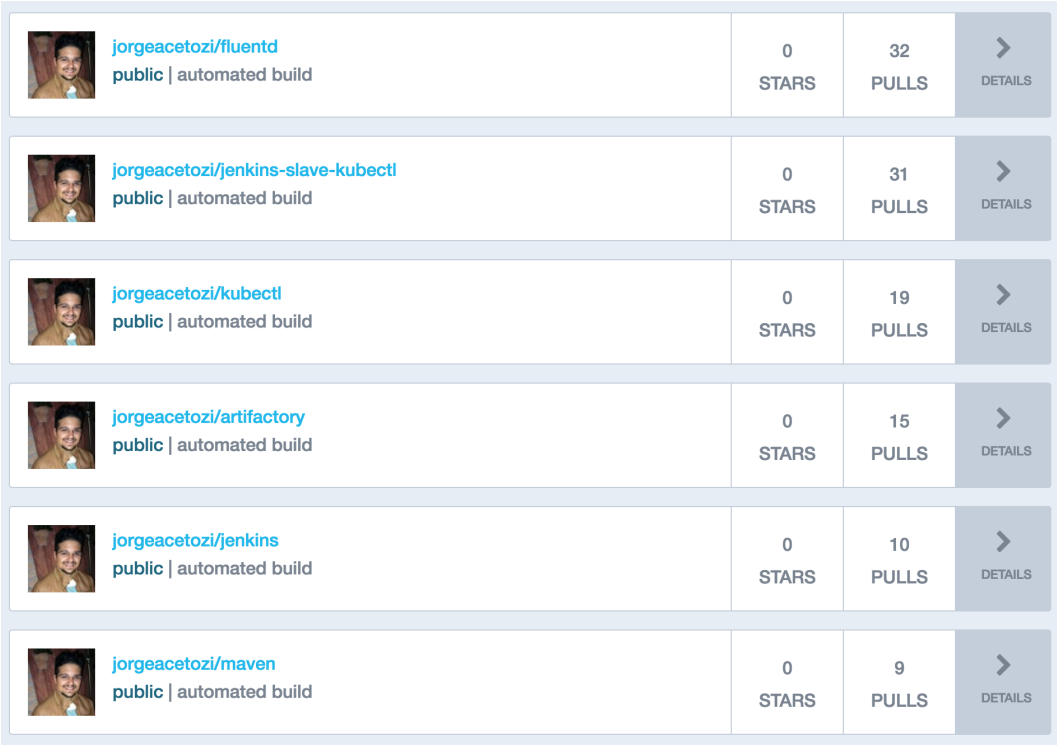
When you create a container from an image by issuing the `docker container run` command, it first checks whether the image is already available on your local machine or not. If not, it pulls the image from a remote Docker Registry such as Docker Hub and stores it locally. If you were pulling the image from a different registry (not Docker Hub), you would have to inform its standard DNS, for example:

```
$ docker container run registry.jorgeacetozi.com/artifactory:5.4.6

# or

$ docker container run registry.jorgeacetozi.com:8080/artifactory:5.4.6

# or even

$ docker container run 176.123.111.25/artifactory:5.4.6
```

If you don't inform the registry, Docker will automatically pull the image from `registry-1.docker.io`, which is the public Docker registry (Docker Hub).

Non-Official Docker Images

Non-official Docker images are those created and maintained from regular users (like you and me), such as images I have created and pushed to my own Docker Hub public repositories and are available for anyone to use:



Non-Official Docker Images

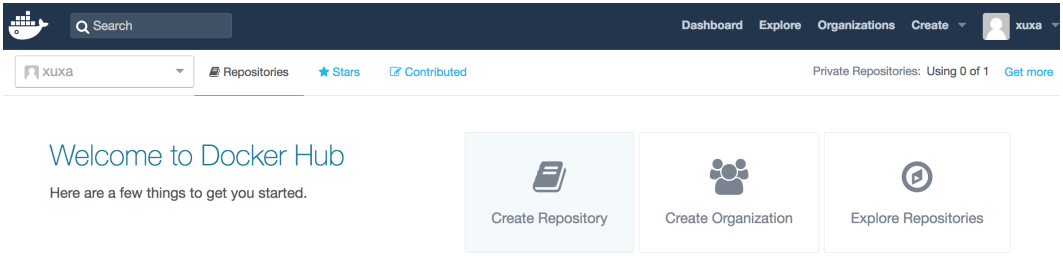
When you pull a non-official Docker image, you also have to provide the `username` to the `docker image pull` command. For example, if you want to use my Artifactory Docker image[28], you would have to issue the following command:

```
$ docker image pull jorgeacetozi/artifactory:5.4.6
```

[28] https://hub.docker.com/r/jorgeacetozi/artifactory/

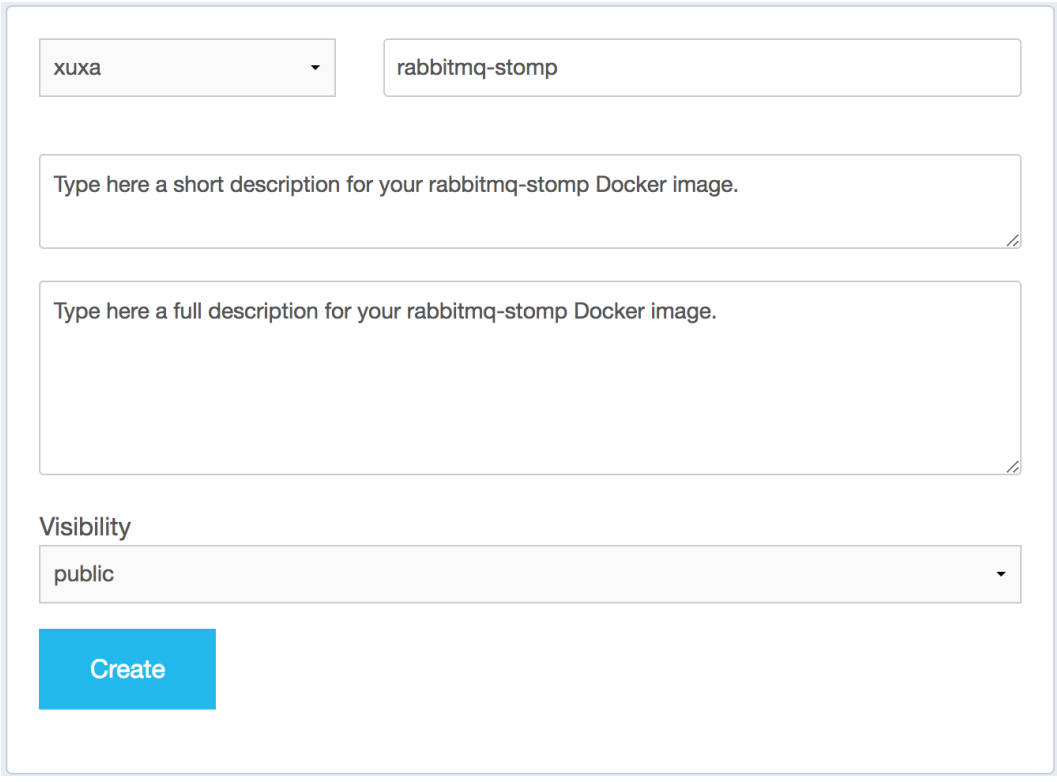
Create a Repository, an Image and Push it to Docker Hub

Now it's time for us to create a repository, a simple image and push it to Docker Hub. Back to your Docker Hub Dashboard, click on the Create Repository.



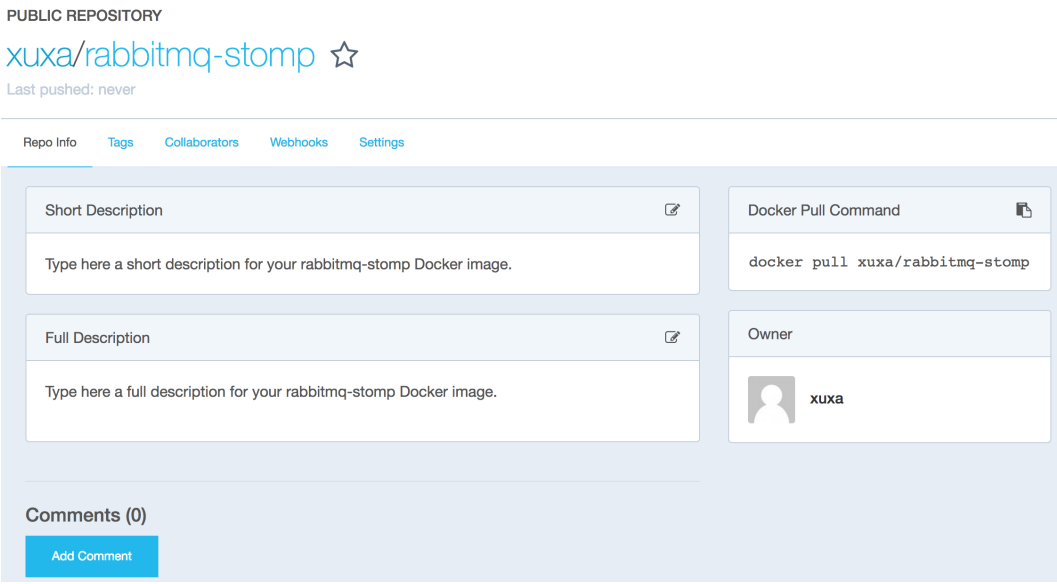
Create Repository

We will create a RabbitMQ image with STOMP support. Fill in the fields as shown:



Create the rabbitmq-stomp Repository

You should see your new empty `rabbitmq-stomp` repository created:



rabbitmq-stomp Empty Repository

In your local machine, create and navigate to a directory named `rabbitmq-stomp-docker` and create an empty file called `Dockerfile`:

```
$ mkdir -p ~/rabbitmq-stomp-docker
$ cd rabbitmq-stomp-docker
# touch Dockerfile
```

Now, open the file `Dockerfile`, paste the following content and save it:

```
FROM rabbitmq:3.6-management

RUN rabbitmq-plugins enable --offline rabbitmq_stomp
```

Basically, your image inherits from the pre-built available image on Docker Hub[29] `rabbitmq:3.6-management` and then it installs the `rabbitmq_stomp` plugin.

Of course that we are going to learn the Dockerfile instructions later on this chapter. For now, just follow me.

[29] https://hub.docker.com/_/rabbitmq

Let's create a Docker image from this `Dockerfile`:

```
$ docker image build -t YOUR_DOCKERHUB_USERNAME/rabbitmq-stomp:3.6 .
```

Replace `YOUR_DOCKERHUB_USERNAME` by your Docker Hub username.

Check that the image is available on your machine:

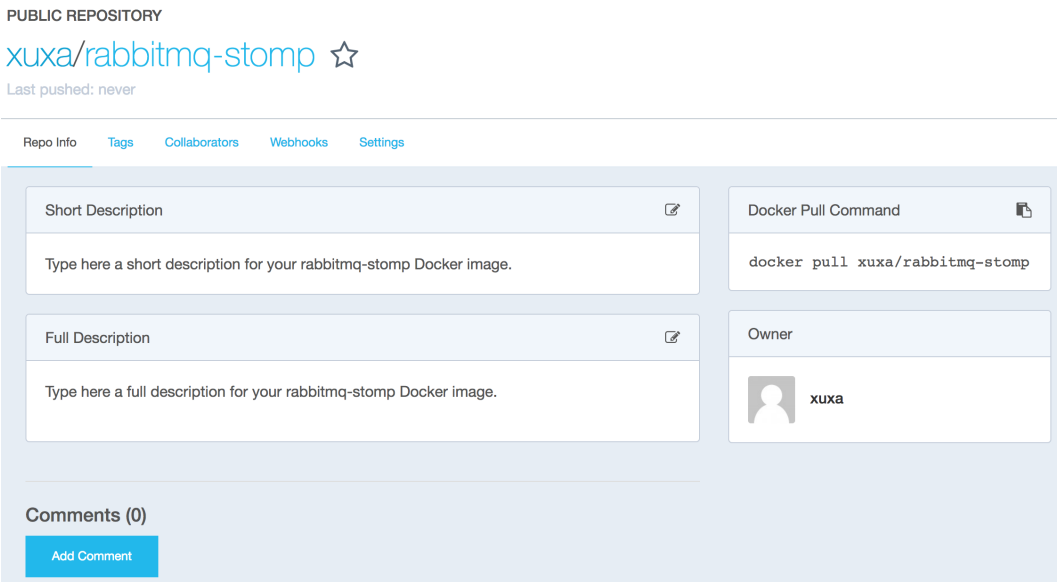
```
$ docker image ls | grep rabbitmq-stomp
```

Now, let's push this image to your newly created remote repository on Docker Hub. First you have to login and then push the image:

```
$ docker login

# provide your credentials

$ docker image push YOUR_DOCKERHUB_USERNAME/rabbitmq-stomp:3.6
```

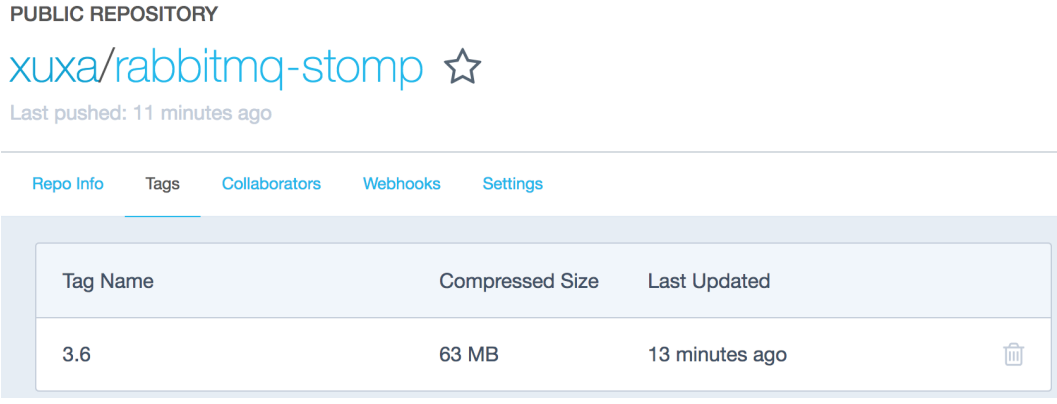
Again, replace `YOUR_DOCKERHUB_USERNAME` by your Docker Hub username.

When the command is finished, return to your browser refresh the repository on Docker Hub. You should see that your newly pushed `rabbitmq-stomp` Docker image.



rabbitmq-stomp Repository After Push

If you navigate to the `Tags` tab, you should see the `3.6` tag:



rabbitmq-stomp Repository Tags

Now, if you want to see a container created from your image in action, issue the following command:

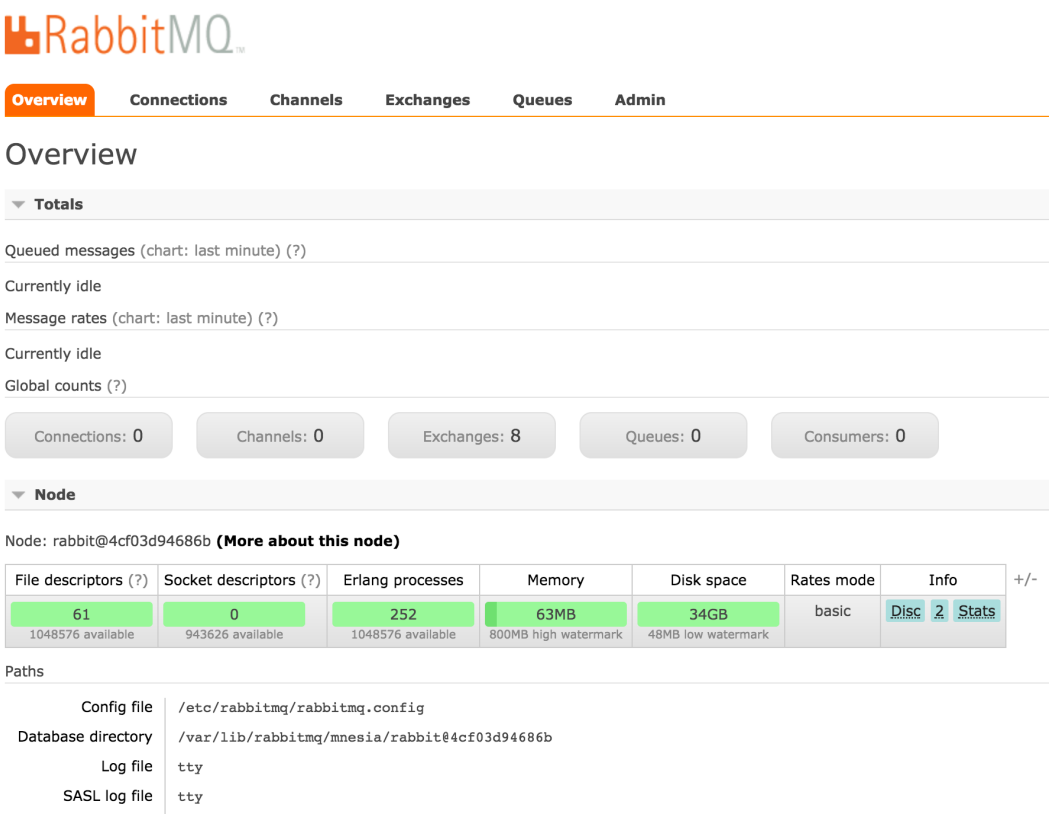
```
$ docker container run -d --name rabbitmq-stomp -p 5672:5672 -p 15672:15672 -p 61613:\
61613 YOUR_DOCKERHUB_USERNAME/rabbitmq-stomp:3.6
```

Again, replace `YOUR_DOCKERHUB_USERNAME` by your Docker Hub username.

Open your browser and hit `http://localhost:15672`:

RabbitMQ Login

Fill in the fields with username `guest` and password `guest` and click on the Login button:



RabbitMQ Dashboard

Congratulations! You have just set up a RabbitMQ with STOMP support container from your own Docker image available publicly on Docker Hub.

Running Containers on Docker

As mentioned earlier, a MySQL database instance can be easily created using Docker with the following instruction:

```
$ docker container run -d --name mysql -e MYSQL_DATABASE=my-database -e MYSQL_ROOT_PA\
SSWORD=root -p 3306:3306 mysql:5.7
```

Now, let's understand the `docker container run` parameters used above and some others.

The `docker container run` is a complex commands with a lot of options. I'm just covering here the most common ones. Please refer to the official Docker container run reference[30] for further information.

[30] https://docs.docker.com/engine/reference/commandline/container_run

Running Containers as Daemons

To run containers in background, you need to provide the `-d` parameter to the `docker container run` command. Note that if you create a MySQL container with the command above, the shell will output the container ID after starting it. However, if you remove the `-d` from the command and execute it again, your shell will be tied to the container output, which is not desirable in most cases.

Container Clean Up

By default a container's file system persists even after the container exits. This makes debugging a lot easier (since you can inspect the final state) and you retain all your data by default. But if you are running short-term foreground processes, these container file systems can really pile up. If instead you'd like Docker to automatically clean up the container and remove the file system when the container exits, you can add the `--rm` flag to the `docker container run` instruction.

Let's check a practical example. First, create an Ubuntu container in the interactive mode without using the `--rm` flag:

```
$ docker container run -ti --name=ubuntu ubuntu:14.04
```

Create a file inside the container:

```
$ echo "Hello, xuxa!" > xuxa
```

Exit the container:

```
$ exit
```

Check that the container is stopped:

```
$ docker container ls -a
CONTAINER ID    IMAGE           STATUS                      NAMES
f4ca09b6b5f2    ubuntu:14.04    Exited (0) 2 seconds ago    ubuntu
```

Start the container again:

```
$ docker container start ubuntu
```

Check that the file `xuxa` is still there:

```
$ docker container exec ubuntu ls -lh
...
drwxrwxrwt   2 root root 4.0K Aug 17 23:24 tmp
drwxr-xr-x   1 root root 4.0K Aug 17 23:23 usr
drwxr-xr-x   1 root root 4.0K Aug 17 23:23 var
-rw-r--r--   1 root root   13 Oct 19 19:51 xuxa
```

Now let's test it again using the `--rm` flag. Remove the `ubuntu` container:

```
$ docker container rm -f ubuntu
```

Create an Ubuntu container in the interactive mode using the `--rm` flag:

```
$ docker container run -ti --name=ubuntu --rm ubuntu:14.04
```

Exit the container:

```
$ exit
```

Check that this time the container is totally gone:

```
$ docker container ls -a
```

This flag is incompatible with `-d` flag you learned before.

Naming Containers

Docker has many commands that allows you to manipulate containers, such as removing a container, outputting the container logs, and so on. These commands either accept the **CONTAINER_ID** (which is a long hash) or the **CONTAINER_NAME**, which you can set when you create a Docker container by providing the option `--name your_container_name` in the `docker container run` command.

Manipulating containers by their names can be very helpful and probably will increase your productivity.

Exposing Ports

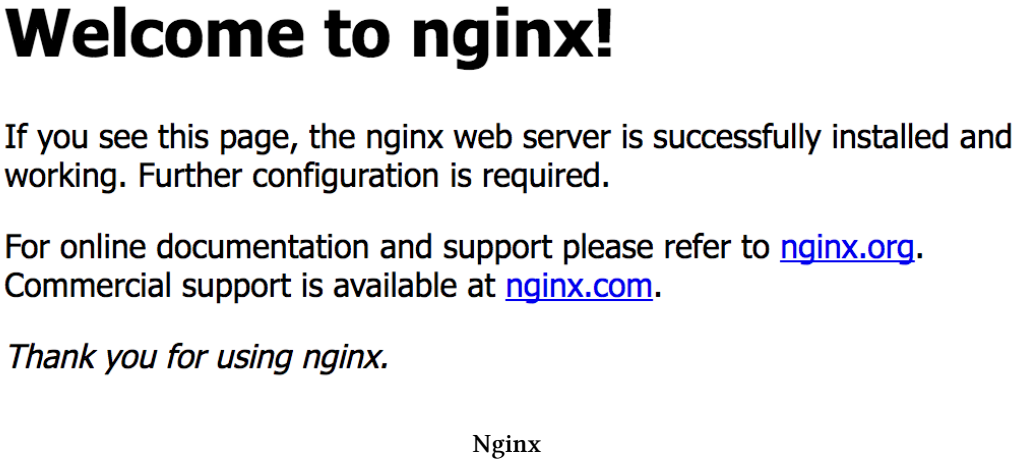
When you run a service on Docker that is expected to be accessed through a port (such as setting up a web server on port 80), you need to bind the service port running inside the container with a port in the host (your local machine, in this case). For example, let's set up Nginx web server as a Docker container using the official image:

```
$ docker container run -d --name my-nginx-container nginx
```

Open your browser and hit `http://localhost`. It doesn't work, right? That's because you have not bound the `80` port from the nginx service running inside the container to a port in the host. Now, try the following:

```
# remove the container
$ docker container rm -f my-nginx-container

# start a nginx container binding the port
$ docker container run -d --name nginx -p 80:80 nginx
```

Open your browser again and navigate to `http://localhost`. You should see that Nginx is up and running:



Nginx

The expected syntax to bind a port is the following: `-p HOST_PORT:CONTAINER_PORT`. That is, if you remove the `my-nginx-container` again, change the above command to `-p 90:80` and create a new Nginx container, you would have to provide the port `90` in your browser, like `http://localhost:90`. Note that regardless of the port that you bind to your machine, the Nginx service inside the container is still running on port 80.

By default, Docker will bind TCP ports. If you want to bind UDP ports, use the `udp` suffix: `-p 12514:12514/udp`

Persistent Data with Volumes

As mentioned in the Difference Between Container and Image section, a container is a group of processes contained in an isolated environment, but running on the same kernel as the host operating system. For example, if you create a file inside a container, this file cannot be accessed from the host operating system (unless you explicitly specify this).

Sometimes this is exactly the behavior you are expecting from containers. But, what about the cases where the data must be persistent and survive across multiple container restarts? Well, for these cases you have to use **Docker Volumes**.

Let's see in action how it works. Create a MySQL container:

```
$ docker container run -d --name mysql -e MYSQL_DATABASE=my-database -e MYSQL_ROOT_PA\
SSWORD=root -p 3306:3306 mysql:5.7
```

Let's get into the container:

```
$ docker container exec -ti mysql /bin/bash
```

The `docker container exec` command runs a command inside a running container. In this case, the command was `/bin/bash`, which will return you a shell session inside the container, but you could use it for any other command like `ls`, `cat`, etc. The `-t` (or `--tty`) option allocates a pseudo-TTY and the `-i` (or `--interactive`) option keeps STDIN open even if not attached.

Create a table `note` and insert some data in it:

```
$ mysql -uroot -proot
$ use my-database

CREATE TABLE note (
    id INTEGER NOT NULL AUTO_INCREMENT,
        title VARCHAR(255) NOT NULL,
        content VARCHAR(255) NOT NULL,
        PRIMARY KEY (id)
);

INSERT INTO note VALUES (1, 'Cassandra', 'Column Family NoSQL db');
```

Perfect! Now, exit MySQL and leave the container dropping back to your terminal session and remove this container:

```
# leave mysql
$ exit

# leave container
$ exit

# remove container
$ docker container rm -f mysql
```

Now, start the MySQL container again:

```
$ docker container run -d --name mysql -e MYSQL_DATABASE=my-database -e MYSQL_ROOT_PA\
SSWORD=root -p 3306:3306 mysql:5.7
```

As this container is not using volumes, the data was lost. Let's check that out:

```
$ docker container exec -ti mysql /bin/bash
$ mysql -uroot -proot
$ use my-database
$ show tables;

Empty set (0.00 sec)
```

It's empty! So, what happened here is the default Docker behavior; if you don't specify you want you data to be persistent, it will be lost after the container is removed.

Now, let's use a volume to make our data persistent. First, create an empty directory in your machine, for example:

```
$ mkdir -p ~/docker-volume-example
```

Create the MySQL container again, but now adding the `-v ~/docker-volume-example:/var/lib/mysql` option to the `docker container run` command:

```
$ docker container run -d --name mysql -e MYSQL_DATABASE=my-database -e MYSQL_ROOT_PA\
SSWORD=root -p 3306:3306 -v ~/docker-volume-example:/var/lib/mysql mysql:5.7
```

Check that the newly create directory content is not empty:

```
$ ls -lh ~/docker-volume-example

total 376912
-rw-r-----    1 jorgeacetozi  staff    56B  3 Out 18:12 auto.cnf
-rw-------    1 jorgeacetozi  staff   1,6K  3 Out 18:13 ca-key.pem
-rw-r--r--    1 jorgeacetozi  staff   1,0K  3 Out 18:13 ca.pem
-rw-r--r--    1 jorgeacetozi  staff   1,1K  3 Out 18:13 client-cert.pem
-rw-------    1 jorgeacetozi  staff   1,6K  3 Out 18:13 client-key.pem
-rw-r-----    1 jorgeacetozi  staff   1,3K  3 Out 18:13 ib_buffer_pool
-rw-r-----    1 jorgeacetozi  staff    48M  3 Out 18:13 ib_logfile0
-rw-r-----    1 jorgeacetozi  staff    48M  3 Out 18:12 ib_logfile1
-rw-r-----    1 jorgeacetozi  staff    76M  3 Out 18:13 ibdata1
-rw-r-----    1 jorgeacetozi  staff    12M  3 Out 18:13 ibtmp1
drwxr-x---   77 jorgeacetozi  staff   2,6K  3 Out 18:13 mysql
drwxr-x---    3 jorgeacetozi  staff   102B  3 Out 18:13 my-database
```

```
drwxr-x---   90 jorgeacetozi  staff   3,0K  3 Out 18:13 performance_schema
-rw-------    1 jorgeacetozi  staff   1,6K  3 Out 18:13 private_key.pem
-rw-r--r--    1 jorgeacetozi  staff   451B  3 Out 18:13 public_key.pem
-rw-r--r--    1 jorgeacetozi  staff   1,1K  3 Out 18:13 server-cert.pem
-rw-------    1 jorgeacetozi  staff   1,6K  3 Out 18:13 server-key.pem
drwxr-x---  108 jorgeacetozi  staff   3,6K  3 Out 18:13 sys
```

When MySQL starts, it creates a bunch of files (listed above) and writes to `/var/lib/mysql` (by default). When we used the option `-v ~/docker-volume-example:/var/lib/mysql mysql:5.7`, we bound the newly created directory in the host machine (your machine) to the `/var/lib/mysql` directory inside the container. Now, everything that is written to `/var/lib/mysql` inside the container will be persisted to the mounted directory in your filesystem. In other words, if you remove the container now, the data will still be safe as it will be kept in the `~/docker-volume-example` directory in your machine. Of course that, if you remove the directory from your machine, the data will be permanently lost.

In cloud environments like Amazon Web Services (AWS), it's a common practice to mount volumes to external scalable storage devices such as Elastic Block Store[31] or Elastic File System[32]. By doing this, you could even survive a complete machine failure without any data loss.

Now, repeat the steps you did to create the table note and insert some data in it then leave the container. Now, instead of getting into the container with the command `/bin/bash` and getting into MySQL cli, we are going to directly select data from the table `note` by executing the `mysql` command with the `-e` option containing the query:

The `-e` option in the command below is not part of the `docker container exec`, but the `mysql` command.

[31] https://aws.amazon.com/ebs
[32] https://aws.amazon.com/efs

```
$ docker container exec -ti mysql mysql -uroot -proot my-database -e "select * from n\
ote"

+----+-----------+------------------------+
| id | title     | content                |
+----+-----------+------------------------+
|  1 | Cassandra | Column Family NoSQL db |
+----+-----------+------------------------+
```

Recreate the container with the `-v ~/docker-volume-example:/var/lib/mysql` option and check that now the data is still there:

```
$ docker container rm -f mysql

$ docker container run -d --name mysql -e MYSQL_DATABASE=my-database -e MYSQL_ROOT_PA\
SSWORD=root -p 3306:3306 -v ~/docker-volume-example:/var/lib/mysql mysql:5.7

$ docker container exec -ti mysql mysql -uroot -proot my-database -e "select * from n\
ote"

+----+-----------+------------------------+
| id | title     | content                |
+----+-----------+------------------------+
|  1 | Cassandra | Column Family NoSQL db |
+----+-----------+------------------------+
```

The data is persistent now! Amazing, isn't it?

Environment Variables

When creating Docker images, you will want the images to be as flexible as possible so that people can reuse the images in different scenarios. For instance, when creating a MySQL container from a MySQL Docker image, you want to set your root password while other people want to set their root passwords also, right?

The creators of MySQL's official Docker image decided that the `MYSQL_ROOT_PASSWORD` environment variable would be the one that you must define to set the root password to your MySQL instance. You can do this by providing the environment variable and its value in the `docker container run` statement with the `-e` parameter.

```
$ docker container run -d --name mysql -e MYSQL_DATABASE=my-database -e MYSQL_ROOT_PA\
SSWORD=root -p 3306:3306 -v ~/docker-volume-example:/var/lib/mysql mysql:5.7
```

In the above command there are two environment variables set: `MYSQL_DATABASE` and `MYSQL_ROOT_PASSWORD`. The `MYSQL_DATABASE` expects the name of a database that will be automatically created for you when the container initializes.

You can check the environment variables available on a running container by issuing the `env` command to it. For example:

```
$ docker container exec -ti mysql env

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=4c9d8db4bb06
TERM=xterm
MYSQL_DATABASE=my-database
MYSQL_ROOT_PASSWORD=root
GOSU_VERSION=1.7
MYSQL_MAJOR=5.7
MYSQL_VERSION=5.7.19-1debian8
HOME=/root
```

Note that the environment variables `MYSQL_DATABASE` and `MYSQL_ROOT_PASSWORD` that we provided to the `docker container run` command are there.

Docker Networking

Networks can be configured to provide complete isolation for containers. They are are managed using the `docker network` command. Let's get started by listing the Docker networks available:

```
$ docker network ls

NETWORK ID          NAME                DRIVER              SCOPE
335f6fda25d6        bridge              bridge              local
a861d086f379        host                host                local
21a2a4394d53        none                null                local
```

When you install Docker, it automatically configures these three networks: `bridge`, `none` and `host`. Let's inspect the `bridge` network and check its CIDR[33]:

```
$ docker network inspect bridge

...
        "Config": [
            {
                "Subnet": "172.17.0.0/16",
                "Gateway": "172.17.0.1"
            }
        ]
...
```

Docker configured the `bridge` subnet as `172.17.0.0/16`, which means the subnet mask is `255.255.0.0`. The `bridge` network represents the `docker0` network present in all Docker installations. Unless you specify otherwise with the `docker container run --net=<NETWORK>` option, the Docker daemon connects containers to this network by default.

[33] https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing

The `bridge` network will enable the connectivity to the other interfaces of the host machine as well as among containers **in the same subnet**. In other words, it means that a container connected to a bridge network has connectivity to the host machine, the external network (Internet) and other containers in the same subnet.

The `none` network will not configure any IP for the container and doesn't have any access to the external network as well as for other containers. Basically, it only has the loopback address. To set up a container using the `none` network mode just provide the `docker container run --net=none` option.

The `host` network will share the host's network stack and all interfaces from the host will be available to the container. Besides, the container's host name will match the host name on the host system. To set up a container using the `host` network mode just provide the `docker container run --net=host` option.

The `none` and `host` networks are not directly configurable in Docker. However, you can configure the default bridge network, as well as creating your own bridge networks (also called as `user defined networks`).

If you need multi-host network connectivity, you have to create an overlay network[34]. These networks utilize a key-value store service such as Consul, Etcd, or ZooKeeper (distributed store) to maintain a mapping between allocated subnets and real host IP addresses. Flannel, by CoreOS, is probably the easiest to use.

[34] https://docs.docker.com/engine/userguide/networking/

Create a Bridge Network

You can easily create a `bridge` network called a `my-network` by executing the command:

```
$ docker network create my-network
```

Check that your network was successfully created:

```
$ docker network ls

NETWORK ID          NAME                DRIVER              SCOPE
335f6fda25d6        bridge              bridge              local
a861d086f379        host                host                local
21a2a4394d53        none                null                local
55b6f3c4edfb        my-network          bridge              local
```

To remove the network, just execute the `docker network rm` command:

```
$ docker network rm my-network
```

When creating a Docker network, you can specify the subnet in CIDR format. Create it again:

```
$ docker network create --subnet=172.20.0.0/16 my-network
```

To create a container connected to this network, use the `docker container run --net=my-network` option:

```
$ docker container run -ti --name=ubuntu --rm --net=my-network ubuntu:14.04
```

Execute the `ifconfig` command to check that the container belongs to the `my-network` network:

```
$ ifconfig

eth0      Link encap:Ethernet  HWaddr 02:42:ac:14:00:02
          inet addr:172.20.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:12 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1016 (1.0 KB)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

It worked! The `eth0` address is `172.20.0.2`. Exit the container:

```
$ exit
```

Please don't remove the `my-network` network; we are going to use it in the next sections.

Container Static IP Address

The `172.20.0.2` IP address was automatically assigned by Docker within the subnet provided. But what if you want to assign a static IP address to a container, say `172.20.0.100`? Actually, it's pretty easy:

```
$ docker container run -ti --name=ubuntu --rm --net=my-network --ip=172.20.0.100 ubun\
tu:14.04
```

Execute the `ifconfig` command to check that the container IP is `172.20.0.100`:

```
$ ifconfig

eth0      Link encap:Ethernet  HWaddr 02:42:ac:14:00:64
          inet addr:172.20.0.100  Bcast:0.0.0.0  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:598 (598.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Now, the IP address set is `172.20.0.100`. It works! Exit the container:

```
$ exit
```

Linking Containers

Before the Docker networks feature, you could use the Docker link feature to allow containers to discover each other and securely transfer information about one container to another container. Now, the Docker link feature is deprecated, although it still works. Let's understand how to used the deprecated Docker link feature and then we move to the Docker networks approach.

Create a Nginx container:

```
$ docker container run -d --name=nginx -p 80:80 nginx
```

Create a Ubuntu container in the interactive mode so that we can try to reach the Nginx container by using its name:

```
$ docker container run -ti --name=ubuntu --rm ubuntu:14.04
```

In the `ubuntu` container, try to `ping` the `nginx` container:

```
$ ping nginx
ping: unknown host nginx
```

It doesn't work! That's because we haven't linked the containers. Exit the `ubuntu` container:

```
$ exit
```

Let's try it again, but now using the legacy Docker link feature:

```
$ docker container run -ti --name=ubuntu --rm --link nginx:nginx ubuntu:14.04
```

In the `ubuntu` container, try to `ping` the `nginx` container again:

```
$ ping nginx
PING nginx (172.17.0.2) 56(84) bytes of data.
64 bytes from nginx (172.17.0.2): icmp_seq=1 ttl=64 time=0.119 ms
64 bytes from nginx (172.17.0.2): icmp_seq=2 ttl=64 time=0.095 ms
```

Now it works! The `--link nginx:nginx` instruction created a bunch of `environment variables` (verify it executing the `env` command inside the `ubuntu` container) and an entry in the `/etc/hosts` file that points to the `nginx` container's IP:

```
$ cat /etc/hosts
127.0.0.1        localhost
::1        localhost ip6-localhost ip6-loopback
fe00::0        ip6-localnet
ff00::0        ip6-mcastprefix
ff02::1        ip6-allnodes
ff02::2        ip6-allrouters
172.17.0.2        nginx 04d99aa4db1b
172.17.0.3        8631e5954ad1
```

Exit the `ubuntu` container, remove the `nginx` container and let's do it again, but now using the recommended way (using `user defined networks`):

```
$ exit
$ docker container rm -f nginx
```

As we already have an `user defined network` created, the `my-network` from the last section, let's use it. Create the Nginx container attaching it to the `my-network` network:

```
$ docker container run -d --name=nginx --net=my-network -p 80:80 nginx
```

Create the Ubuntu container attaching it to the `my-network` network as well:

```
$ docker container run -ti --name=ubuntu --rm --net=my-network ubuntu:14.04
```

In the `ubuntu` container, try to ping the `nginx` container by using its name:

```
$ ping nginx
PING nginx (172.20.0.2) 56(84) bytes of data.
64 bytes from nginx.my-network (172.20.0.2): icmp_seq=1 ttl=64 time=0.115 ms
64 bytes from nginx.my-network (172.20.0.2): icmp_seq=2 ttl=64 time=0.098 ms
```

It works! Basically, a `user defined network` give you have effortless internal name resolution. You can call other containers on the same `user defined network` by their names. It's pretty easy, isn't it? Now, let's just clean everything up; exit the `ubuntu` container and remove the `nginx` container:

```
$ exit
$ docker container rm -f nginx
```

Most Used Docker Commands

In this section, I'm going to list the most common Docker commands used on a daily-basis (in my experience, of course).

Bear in mind that this section is only a reminder. For the complete list of Docker commands, please refer to the official documentation[35].

[35] https://docs.docker.com

Images

Here are the most common commands used for managing images:

- `docker image build`: Build an image from a Dockerfile
- `docker image history`: Show the history of an image
- `docker image inspect`: Display detailed information on one or more images
- `docker image ls`: List images
- `docker image prune`: Remove unused images
- `docker image pull`: Pull an image or a repository from a registry
- `docker image push`: Push an image or a repository to a registry
- `docker image rm`: Remove one or more images

Containers

Here are the most common commands used for managing Docker containers:

- `docker container attach`: Attach local standard input, output, and error streams to a running container
- `docker container exec`: Run a command in a running container
- `docker container inspect`: Display detailed information on one or more containers
- `docker container kill`: Kill one or more running containers
- `docker container logs`: Fetch the logs of a container
- `docker container ls`: List containers
- `docker container prune`: Remove all stopped containers
- `docker container restart`: Restart one or more containers
- `docker container rm`: Remove one or more containers
- `docker container run`: Run a command in a new container
- `docker container stats`: Display a live stream of container(s) resource usage statistics
- `docker container top`: Display the running processes of a container

Misc

Here is a misc of common used Docker commands:

- `docker info`: Display system-wide information
- `docker login`: Log in to a Docker registry
- `docker logout`: Log out from a Docker registry
- `docker network`: Manage networks
- `docker search`: Search the Docker Hub for images
- `docker system`: Manage Docker

Building Docker Images

You will not need to create any Docker image when setting up the Graylog project later on this book (I've already created them for you). However, knowing how to create your own Docker images is extremely important for your daily-basis activities. So, let's have an overview on how to to create docker images by writing a `Dockerfile`.

Dockerfile

You have already created a Dockerfile when you published a RabbitMQ with STOMP support to Docker Hub, but we didn't care so much with its content on that moment. Now, let's learn some of the most important Dockerfile instructions by reviewing the Dockerfile used to wrap Elasticsearch into a Docker image.

```
# Set the base image to java8
FROM openjdk:8-jre

# File Author / Maintainer
MAINTAINER Jorge Acetozi

# Define default environment variables
ENV ELASTICSEARCH_VERSION=5.6.6
ENV ELASTICSEARCH_HOME=/opt/elasticsearch

# Create elasticsearch group and user
RUN groupadd -g 1000 elasticsearch \
  && useradd -d "$ELASTICSEARCH_HOME" -u 1000 -g 1000 -s /sbin/nologin elasticsearch

# Install Elasticsearch 5.6.6
RUN wget https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-$ELASTICS\
EARCH_VERSION.tar.gz -P /opt \
  && cd /opt \
  && tar xvzf elasticsearch-$ELASTICSEARCH_VERSION.tar.gz \
  && mv elasticsearch-$ELASTICSEARCH_VERSION elasticsearch \
  && rm -f elasticsearch-$ELASTICSEARCH_VERSION.tar.gz \
  && mkdir -p /var/data/elasticsearch \
  && mkdir -p /var/log/elasticsearch

WORKDIR $ELASTICSEARCH_HOME

# Copy initialization script
COPY bin/docker-entrypoint.sh bin/docker-entrypoint.sh

# Add configuration files
ADD config/* config/

# Change directories ownership to elasticsearch user and group
RUN chown -R elasticsearch:elasticsearch $ELASTICSEARCH_HOME /var/data/elasticsearch \
```

```
/var/log/elasticsearch

# Run the container as elasticsearch user
USER elasticsearch

# Define mountable directories
VOLUME /var/data/elasticsearch
VOLUME /var/log/elasticsearch

# Exposes ports
EXPOSE 9200 9300

# Define main command
ENTRYPOINT ["/opt/elasticsearch/bin/docker-entrypoint.sh"]
```

The Docker team provide an important guide[36] describing the best practices when creating a Dockerfile. I strongly recommend that you read this document.

FROM

The `FROM` instruction specifies the image to inherit from (or the "base image" to) your image. In the Elasticsearch Dockerfile, it inherits from the `openjdk` image with the tag `8-jre`. This inheritance mechanism highly encourages reusability.

ENV

The `ENV` instruction defines environment variables. Generally, environment variables are used to specify values that might be used in many instructions along the Dockerfile, because if a change is needed on their values, there is only a single point to change; the environment variable value. Common uses include service versions such as `ELASTICSEARCH_VERSION` and directories paths such as `ELASTICSEARCH_HOME`.

[36] https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/

RUN

The `RUN` instruction executes a shell command. It's generally used to install applications using package managers (`apt`, `yum`, `apk`), download files (`wget`, `curl`), create directories (`mkdir`), unzip files (`unzip`), change directory ownership (`chown`), change permissions (`chmod`), create UNIX groups and users (`groupadd`, `useradd`), execute scripts, and so on.

WORKDIR

The `WORKDIR` instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile. It means that any directory path used in these instructions will be relative to the directory defined in the WORKDIR instruction.

The command `WORKDIR $ELASTICSEARCH_HOME` defines that the working directory is the value of the environment variable `ELASTICSEARCH_HOME`, which is `/opt/elasticsearch`. So, if a `RUN mkdir -p dir1/dir2` is executed after this `WORKDIR` instruction, the created directory's path will be `/opt/elasticsearch/dir1/dir2`.

COPY

The `COPY` instruction copies files from the host to the image. So, the instruction `COPY bin/docker-entrypoint.sh bin/docker-entrypoint.sh` is just copying the script to inside the image.

ADD

The `ADD` instruction is very similar to the `COPY` instruction, but it has more features. For example, consider the following syntax:

```
ADD source destination
```

The `ADD` instruction allows the source to be an URL and if the source parameter of ADD is an archive in a recognized compression format (such as `.zip` or `.tar.gz`), it will automatically be unpacked. The `COPY` instruction doesn't have these capabilities.

So, in the `COPY bin/docker-entrypoint.sh bin/docker-entrypoint.sh` instruction, the `COPY` instruction could be replaced by the `ADD` without any issues. I just used the `COPY` here to illustrate another possible instruction to use in the Dockerfile.

EXPOSE

The `EXPOSE` instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP, and the default is TCP if the protocol is not specified.

The EXPOSE instruction does not actually publish the port. It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published. To actually publish the port when running the container, use the `-p` flag on the `docker container run` command to publish and map one or more ports as you learned in the Exposing Ports section.

The `EXPOSE 9200 9300` instruction is just exposing the default Elasticsearch ports (REST API and internal communication).

ENTRYPOINT

The `ENTRYPOINT` instruction defines which executable should be run when a container is started from the image. Thus, the `ENTRYPOINT ["/opt/elasticsearch/bin/docker-entrypoint.sh"]` instruction defines that when a container is created from this image (running the `docker container run` command), the script `docker-entrypoint.sh` will be executed.

Basically, the `docker-entrypoint.sh` script acts as an initialization script where we can define additional runtime behaviors. Below is the script content, which simply uses `sed` to replace placeholders with values provided by some environment variables and then starts Elasticsearch:

```
#!/bin/bash
set -e

sed -i "s/CLUSTER_NAME/$CLUSTER_NAME/
  s/NODE_NAME/$NODE_NAME/
  s/NETWORK_HOST/$NETWORK_HOST/
  s/UNICAST_HOSTS/$UNICAST_HOSTS/" ./config/elasticsearch.yml

echo -e "Starting Elasticsearch $ELASTICSEARCH_VERSION"
exec /opt/elasticsearch/bin/elasticsearch
```

VOLUME

In the Persistent Data with Volumes section, you have learned how to mount a volume using the `-v` option in the `docker container run` command and we took MySQL as an example. Below is the command you used to create the MySQL container (note the `/var/lib/mysql`).

```
$ docker container run -d --name mysql -e MYSQL_DATABASE=my-database -e MYSQL_ROOT_PA\
SSWORD=root -p 3306:3306 -v ~/docker-volume-example:/var/lib/mysql mysql:5.7
```

Now, have a look at the MySQL Dockerfile[37]. Note that it declares a volume for the directory `/var/lib/mysql` using the instruction `VOLUME /var/lib/mysql`.

Basically, the `VOLUME` instruction should be used to expose any database storage area, configuration storage, or files/folders created by your Docker container (such as log files, for example). It creates a mount point with the specified name and marks it as holding externally mounted volumes from the host or other containers using the `-v` option in the `docker container run` command.

> You should never embed configurations or secrets into a Docker image. Instead, expect them to be supplied at runtime whether using `volumes` or `environment variables`. The reason for this is because Docker images should be both reusable and secure, and when an image contains embedded configuration or secrets it

[37] https://github.com/docker-library/mysql/blob/0590e4efd2b31ec794383f084d419dea9bc752c4/5.7/Dockerfile

violates this rule.

USER

Another important Dockerfile instruction is `USER`. The `USER` instruction sets the user name or UID to use when running the image and for any following `RUN` instructions. If a service can run without privileges, it's recommended to use the `USER` instruction to change to a non-root user. You can easily create a group and a user using the `groupadd` and `useradd` commands as shown in the Elasticsearch Dockerfile.