Tattle Documentation

Release latest

Nick MacCarthy (nick@nickmaccarthy.com)

Nov 03, 2018

Contents

CHAPTER 1

Tattle Intro - Alerting For Your Elasticsearch Data

Welcome to Tattle, an alerting tool for your Elasticsearch data.

Tattle aims to provide you with alerting capabilities for the data stored in your Elasticsearch cluster. Utilizing powerful Elasticsearch features such as Aggregations and Lucene Query Syntax, coupled together with Tattle's own query language (TQL) our goal is to make alerts that are easy to build and most of all, easy to read (because lets face it, you probably wont be the only one who has to read them)

1.1 Overview

Tattle was designed to make use of the powerful features of Elasticsearch (such as Aggregations) to alert us to a multitude of various metrics and log or event type data. Things such as frequencys, event spikes, aggregation matches, etc all play a big role in our capabilities for alerting. Coupled together with an extendable `alert action` mechanism, Tattle can even fix problems for you as they arise (using the `script` action for example), notify a Pager Duty Service, or even post to a Slack channel; or all of the above.

1.2 History

Back in 2013 when ELK was farily new term in most people vocabularies, I couldnt find any way to alert on the data inside my Elasticsearch cluster. Since I was mostly using Elaticsearch to store log, metric and event type data, I couldnt really call ELK a full logging solution until it had the alerting component. I then decided to build my own system, and "Project Bluenote" was born (because I didnt have a better name for it at the time (and because I was listening to some old Bluenote records when I wrote the first few lines of code)). Over the next year or so it was developed on an off in my free time and eventually became an invaluable tool for keeping my site up. One day, after it alerted me that someone had released some particularly questionable code, they said "Bluenote is such a tattle tale". Figuring that was a much more appropriate name for the project, Tattle was born.

1.3 Requirements

- Python 2.7 or Python 3.3+
- Virtualenv
- Pip
- Git

1.4 Quick Setup & Install

Lets assume we will assume will be installing Tattle to `/opt/Tattle`.

```
cd /opt
git clone https://github.com/nickmaccarthy/Tattle
cd /opt/Tattle
virtualenv env && source env/bin/activate
pip install -r requirements.txt
```

Note: `$TATTLE_HOME` refers to where you have installed Tattle. In the case of this documentation we installed it in `/opt/Tattle`

Check out the *Getting Started - Install & Setup* section for more details.

Or check out the *Tales, Alerts and Actions* section on setting up your first Tale and working with Tattle.

CHAPTER 2

Getting Started - Install & Setup

Installing Tattle is pretty simple process. Tattle was written in Python, and was designed to work within a virtualenv. Tattle works with both Python 2.7 and Python 3.3+ (it was tested against Python 3.5, which is the latest at the time of this writing).

The first steps for getting Tattle installed are ensuring you have either Python 2.7 or Python 3.3+ installed on your system, as well as pip and virtualenv (typically installing python 2.7 on a modern linux system is as easy as `apt-get install python27 python27-virtualenv python27-pip` for Debian based systems (such as Ubuntu) or `yum install python27 python27-pip python27-virtualenv` for CentOS based users)

Note: If you already have python 2.7.10 or above installed, then you should already have *pip*. You can typically install virtualenv with pip as well

After *Python 2.7*, *pip* and *virtualenv* have been installed, you will need ensure you have *git* installed (`apt-get install git`) to clone the repository where Tattle is located.

Note: We use git to pull down the source code for Tattle. If you dont have or want to install *git*, you can also download the zip file for Tattle from the Tattle Github page and unzip it manually to a directory of your choice.

After you have these base applications installed, check our the *Setup & Install* section to continue.

2.1 Code

The source code for Tattle is located on Github: https://github.com/nickmaccarthy/Tattle

2.2 Requirements

- Python 2.7 or Python 3.3+

- Virtualenv
- Pip
- Git

2.3 Setup & Install

For the sake of this documentation we will be installing Tattle to `/opt/Tattle`. If you wish to install Tattle into a different location, simply replace `/opt/tattle` with your install directory.

```
cd /opt
git clone https://github.com/nickmaccarthy/Tattle
cd /opt/Tattle
virtualenv env && source env/bin/activate
pip install -r requirements.txt
```

Note: Throughout the rest of this documentation, you will see reference to `$TATTLE_HOME`. This is a variable to represent where Tattle was installed. In the case of this documentation we installed it in `/opt/Tattle`

2.4 Configuration

There are a few things we need to configure before we can start using Tattle.

1. We need to define how to connect to our Elasticserach cluster. To do this, simple edit `$TATTLE_HOME/etc/tattle/elasticsearch.yaml`

Note: There are example configuration files in the `$TATTLE_HOME/examples` directory you can use for reference. In this case, You can use the included `$TATTLE_HOME/examples/elasticsearch.yaml` as guidance.

Example

```
servers:
    - "elasticsearch.mycompany.com"
args:
    port: 9200
    use_ssl: false
    timeout: 30
```

Note: The above YAML reference uses the directives defined the python elasticsearch client. The variables for the args can be found here: https://elasticsearch-py.readthedocs.io/en/master/api.html#elasticsearch

2. We need to define our email configuration, and or other alert output configurations such as pagerduty, slack, etc. In this case will just set up our email output. Create a new file called `$TATTLE_HOME/etc/tattle/email.yaml`

Example

```
server: 'localhost'
port: 25
```

```
default_sender: 'tattle@localhost'
subject_prefix: "Tattle Alert: "
```

3. Now we need to make *Tale*. A *Tale* is a definition for our alert. As Tales are specific to the type of data you need to alert on, they will require more a more in-depth explination then what we can provide in this section. So please go check out the *Tales, Alerts and Actions* section on setting up a Tale, and come back to the next step once that is completed.
4. Once you have configured your Tale, please look at the *Environment Variables* section below.

Note: A note on yaml configuration files. You can use use both `.yml` or `.yaml` as your file extension

---

2.5 Environment Variables

Tattle can also support *environment variables* for configuration locations. This is useful if you wanted to place your Tattle configurations in a directory other than `$TATTLE_HOME`, and makes container applications such a docker easier to implement.

2.5.1 $TATTLE_CONFIG_DIR

`$TATTLE_CONFIG_DIR` is the location where Tattle will search for its config files like, `tattle.yml`, `elasticserach.yml`, `pagerduty.yml`, etc

2.5.2 $TATTLE_TALES

`$TATTLE_TALES` is the location where Tattle will search for Tales.

Note: You can specify multiple directories in Environment Variables by delimiting them with a `:`. For example if we wanted to search our home dir and `/etc/tattle/tatles`, we could use `$TATTLE_TALES=/home/myuser/tales:/etc/tattle/tales`

2.6 Running

1. Tattle is designed (in its current form) to run standalone. Its best to run Tattle on a CRON job, typically with an interval of every minute. You can run it manually yourself as well. Just run `$TATTLE_HOME/bin/tattled.py`.

Note: There are thoughts of having Tattle run as daemon in the future.

---

2. Tattle also comes with a tale/alert testing utility. This is how you can check your Tale before you put it into 'production'. If there are matches, tattle will run whatever is in the Tale `action`. Simply run `$TATTLE_HOME/bin/test_alert.py /path/to/the/tale.yaml`.

2.7 Example Cron

Its typical for Tattle run on a CRON job, with a one minute interval. Everytime Tattle runs, it will check all alerts/Tales. So if you add additonal tales, Tattle should pick them on its next run

```
*/1 * * * * /opt/Tattle/env/bin/python /opt/Tattle/bin/tattled.py
```

CHAPTER 3

Tales, Alerts and Actions

3.1 Introduction

Tales are the heart and soul of the system. Tales are definitions for alerts and define such things as our time window for the events we seek, what query will run, thresholds, actions to take, etc.

Note: Tales are kept in *yaml* files in `$TATTLE_HOME/etc/tattle/tales`, `$TATTLE_HOME/etc/tales` or in `$TATTLE_HOME/etc/alerts`.

To understand *Tales*, lets take a look at an example below. Please note, we will use this as a reference for the rest of the Tales documentation.

3.2 Example Tales

In this example, we will be finding all hosts in our environment that have a disk usage of *greater than or equal* to *90%* for the past *1 hour*. When a match is found, it will send us an alert via Pagerduty as well as an email for each *key* (host in this case) that was matched.

TQL Query with multiple aggregations and multiple actions example

```
name: "Disk Usage over 90 %"
description: "Disk Usage High on a host or series of hosts"
severity: "High"
tql_query: "summary.fullest_disk:>=90 | terms name=server, field=host.raw | avg␣
↪name=fullest_disk, field=summary.fullest_disk"
exclude: 'host.raw:database4.mycompany.com'
index: "system-metrics-*"
enabled: 1
schedule: '* 8-18 * * mon-fri'
exclude_schedule: '30 12 * * *
timeperiod:
```

```
    start: "now-1h"
    end: "now"
alert:
    type: "frequency"
    relation: "gt"
    qty: 0
    realert: "4h"
    return_matches:
        length: 10
        random: true
action:
    pagerduty:
        enabled: 1
        service_key: "TattleAlerts"
        once_per_match:
            match_key: "key"

    email:
        enabled: 1
        once_per_match:
            match_key: "key"
        to:
            - 'my_email@company.com'
            - 'alerts@company.com'
```

For more breakdown on this Tale, lets look at the *Tale Definitions* section.

3.3 Tale Definitions

3.3.1 name

- Required: *Yes*
- Description: Name of the alert

Example:

```
name: "Disk Usage >= 90%"``
```

3.3.2 description

- Required: *Yes*
- Description: A brief description of the tale.

Example:

```
description: "The disk usage on the server >= 90% on the root filesystem``
```

3.3.3 severity

- Required: *No*
- Description: The severity of the alert. This is a string, and can be whatever you want. 1-5, Low-Crit, etc

Example:

```
severity: "High"
```

3.3.4 enabled

- Required: *Yes*
- Description: Whteher this Tale is enabled (1)(True) or disabled (0)(False)

Example:

```
# This alert is enabled
enabled: 1
# This alert is disabled
enabled: 0
# You can even use strings
enabled: "yes"
# Or even booleans
enabled: true
```

3.3.5 disabled

- Required: *Yes* but only if you didnt specify an `enabled`
- Description: The same thing as as `enabled` above, but with opposite logic. Tattle used to use the term `disabled` instead of `enabled`, but this old method is left in for legacy support. Please use the `enabled` term going forward with new Tales.

Example:

```
# This alert is enabled, not disabled
disabled: 0
# this alert is disabled
disabled: 1
```

3.3.6 tql_query

- Required: *Yes*
- Description: The TQL query for the Tale. See the *Tattle Query Language (TQL)* page for more details on TQL

Example:

```
tql_query: "summary.fullest_disk:>=90 | terms name=server, field=host.raw | avg␣
↪name=fullest_disk, field=summary.fullest_disk"
```

3.3.7 index

- Required *Yes*
- Description: The index pattern where you the events you are searching reside. Default is `logstash-*`
- **More information:**

  - **Builds the index names that Tattle will search data against**
    - It uses the `start` and `end` time in `timeperiod` of the Tale to determine which indexes to build/query against.
  - Its common to store *timeseries* based indexes in Elasticsearch. The most common format is store your data by day and append a date timestamp at the end of index. The most common format is `YYYY.MM.DD`. If you specify a `*` at the end of the index pattern in Tattle, ie `logstash-*`, then Tattle will build the indexs for you by `day` when it does its search.
  - If you store your indexes in a different time pattern or interval other than daily, then you can specify the time pattern and interval. See examples 2-4
  - If you done specify a pattern or interval or a `*`, then Tattle will search just that single index.
  - For more information on the tokens allowd for the patterns, please see the documentation for Arrow.

Example 1:

```
index: "system-metrics-*"
```

Makes index names similar to:

```
system-metrics-2016.01.01, system-metrics-2016.12.29 .... etc
```

Example 2 - specifying pattern and interval:

```
index:
    name: "system-metrics-"
    pattern: "YYYY.MM.DD"
    interval: "day"
```

This would give us index names such as:

```
system-metrics-2016.01.01, system-metrics-2015.12.29, etc
```

Example 3 - specifying pattern as string:

```
index: "system-metrics-%{+YYYY.MM.DD}"
```

This would give us the same index names as Example 1 and 2 Example 4 - specifying pattern and interval as a string, not the interval at the end of the string after the `:` :

```
index: "system-metrics-%{+YYYY.MM.DD.HH}:hour"
```

Valid intervals are python datetime - `year`, `month`, `week`, `day`, `hour`, `second` This would build index names with hour intervals such as:

```
some-index-2015.12.29.00,some-index-2015.12.29.01,some-index-2015.12.29.02,some-index-
↪2015.12.29.03,some-index-2015.12.29.04,some-index-2015.12.29.05,some-index-2015.12.
↪29.06,some-index-2015.12.29.07, ... etc
```

3.3.8 schedule

- Required *No*
- Description: Specifies when a Tale should run, using cron syntax.

- More Information: Sometimes you may only want to have a Tale run during business hours (8am - 6pm , mon-fri). This allows you to specify when this Tale will run in cron format (see example below)
- Credit: This is using the parse-crontab module by Josiah Carlson which can be found here

Note: If you do not specify a `schedule` for your Tale, then Tattle will run this Tale every time it runs.

Note: The CRON scheulde you specify here will pertain to the timezone of the system Tattle is running on. If Tattle's system is UTC, but you need this scheduled in EST, please account for that time difference.

Example:

```
schedule: "* 8-18 * * mon-fri"
```

Cron Examples:

```
30 \*/2 * * * -> 30 minutes past the hour every 2 hours
15,45 23 * * * -> 11:15PM and 11:45PM every day
0 1 ? * SUN -> 1AM every Sunday
0 1 * * SUN -> 1AM every Sunday (same as above)
0 0 1 jan/2 * 2011-2013 -> midnight on January 1, 2011 and the first of every odd␣
↪month until the end of 2013
24 7 L * * -> 7:24 AM on the last day of every month
24 7 * * L5 -> 7:24 AM on the last friday of every month
24 7 * * Lwed-fri -> 7:24 AM on the last wednesday, thursday, and friday of every␣
↪month
```

3.3.9 exclude_schedule

- Required *No*
- Description: Allows you to specify a time period for when this Tale will not run, in cron format. This would be the opposite of the `schedule` option
- More information: Lets say you have a something that runs every saturday and sunday morning between 4am and 7am. You know its normal so you dont want to be alerted about it, but any other time you do. This parameter allows you to specify a window for Tale to not run at.
- Credit: This is using the parse-crontab module by Josiah Carlson which can be found here

Example:

```
exclude_schedule: '* 4-7 * sat * '
```

Cron Examples:

```
30 \*/2 * * * -> 30 minutes past the hour every 2 hours
15,45 23 * * * -> 11:15PM and 11:45PM every day
0 1 ? * SUN -> 1AM every Sunday
0 1 * * SUN -> 1AM every Sunday (same as above)
0 0 1 jan/2 * 2011-2013 -> midnight on January 1, 2011 and the first of every odd␣
↪month until the end of 2013
24 7 L * * -> 7:24 AM on the last day of every month
24 7 * * L5 -> 7:24 AM on the last friday of every month
24 7 * * Lwed-fri -> 7:24 AM on the last wednesday, thursday, and friday of every␣
↪month
```

3.3.10 timeperiod

- `start, end`
- Required: *Yes*
- Description: The timeperiod for events this Tale searches for. This is a rolling window using python-datemath as our start and end times.
- **More information:**
  - More documentation on python-datemath can be found here: https://github.com/nickmaccarthy/python-datemath

Example:

```
timeperiod:
    # The start of our alert window
    start: 'now-1h'
    # The end of our alert window
    end: 'now'
```

3.3.11 exclude

- Required: *No*
- Description: Allows you to specify query parameters to exclude form this Tale
- Can also be a list of items as well, which Tattle will "OR" together
- More information: For this example, lets say we dont want to see alerts for the host `database4.company.com` because its supposed to have a full disk, we can use this to parameter to exclude that host from the tale. This parameter accepts Lucne query syntax

Example:

```
exclude: "host:database4.company.com OR host:database5.company.com"
```

or

exclude:

- "host:database4.company.com"
- "host:database5.company.com"
- "*some other string*

3.3.12 alert

3.3.13 type

- Required: *Yes*
- Description: The type of the alert

- **Values**
  - **`frequency` or `number_of_events`**
    - Description: If the *number of events* meets our `relation` and `qty`
  - **`agg_match`**
    - Description: If our value meets a regular expression match of "something"

3.3.14 relation

- Required: *Yes*
- Description: If our event count meets our relation, then the alert should fire
- **Values**
  - `eq`, = - Equal To
  - `ne`, `!=` - Not Equal To
  - `lt`, < - Less Than
  - `gt`, > - Greater Than
  - `le`, <= - Less Than or Equal To
  - `ge`, >= - Greater Than or Equal To

3.3.15 qty

- Required: *Yes*
- Description: What we compare our `relation` to

Example":

```
## If our number of events is greater than or equal to 10, then we should alert
relation: ">="
qty: 10
```

3.3.16 realert

- Required: *Yes*
- Description: How long Tattle will wait before it will re-alert on this Tale. If Tattle is still finding matches for this Tale, but we are within the re-alert threshold, then Tattle will not alert.
- **Notes:**
  - Every time Tattle fires an alert, it stores it in the Tattle index in Elasticserach (default is `tattle-int`). When the Tale gets loaded, one of the first thing it does it check to see when the last time this Tale fired. It then compares the last time to the realert threshold, diffs the two and if we are beyone our re-alert threshold, then Tattle will re-fire the Tale.
  - **It uses simple datemath like so:**
    - `1h`
    - `2m`

`* 3d`

Example:

```
# Don't alert us to this again for 1 hour
realert: "1h"
```

3.3.17 return_matches

- Required: *Yes*
- Description: If Tattle should return the matches it found. It will return those matches in whatever action you have configured
- **Notes:**
  - Sometimes you can get many matches (hundreds or thousands for example). With the `random: True` or `length: 10` stanzas Tattle can return a randam sample of `10` results

Example:

```
# Assuming we could get hundreds of matches back
return_matches:
    # Return back a random sample of 20 results
    random: true
    length: 20
```

3.3.18 action

3.4 Alert Types

3.4.1 Frequency

Frequency alerts occur when a certain number of events (as defined by `relation` and `qty`) occur within a certain period of time.

Here are some examples:

- "20 or more failed login events with in the past 1 hour"

Example

```
name: "Too many login failures"
tql_query: '"failed login"'
index: "secure-log-*"
timeperiod:
    start: "now-1h"
    end: "now"
alert:
    type: "frequency"
    qty: 20
    relation: ">="
```

- "300 or more Nginx logs with an error code of 502 in the last 1 minute"

Example

```
name: "NGINX 502 errors"
tql_query: "status:502 | terms field=hostname"
index: "nginx-access-*"
timeperiod:
    start: "now-1m"
    end: "now"
alert:
    type: "frequency"
    qty: 300
    relation: ">="
```

- "Less than 1000 events on all of our NGINX logs for the past 1 hour"

Example

```
name: "Low event count on NGINX, possible log outage"
tql_query: "*"
index: "nginx-access-*"
timeperiod:
    start: "now-1h"
    end: "now"
alert:
    type: "frequency"
    qty: 1000
    relation: "le"
```

3.4.2 Aggregation Match

Agg Match alerts are useful for aggregation based alerts where the keys and values can change depending on your data. Often times the result of most metric based aggregtions will a field called `value`. This type of alert type can use a regular expression to match the value and compare it to our `qty` and `relation` fields

When you use an agg_match, Tattle will flatten the aggregation returned so it can be iterated against and matched by a regular expression.

Take this example a return

```
{
    "hits": {
        "hits": [],
        "total": 2,
        "max_score": 0.0
    },
    "_shards": {
        "successful": 5,
        "failed": 0,
        "total": 5
    },
    "took": 31,
    "aggregations": {
        "terms": {
            "buckets": [
                {
                    "avg": {
                        "value": 90.8
                    },
                    "key": "someserver1.somecompany.net",
```

```
                    "doc_count": 1
                },
                {
                    "avg": {
                        "value": 93.5
                    },
                    "key": "someserver2.somecompany.net",
                    "doc_count": 1
                }
            ],
            "sum_other_doc_count": 0,
            "doc_count_error_upper_bound": 0
        }
    },
    "timed_out": false
}
```

Tattle would flatten the aggregations section this to

```
aggregations.terms.buckets.0.avg.value = 90.8
aggregations.terms.buckets.0.key = someserver1.somecompany.net
aggregations.terms.buckets.1.avg.value = 93.5
aggregations.terms.buckets.1.key = someserver2.somecompany.net
```

So if we wanted to look for any *values* in our aggs that are `>= 90` we would use the regular expression `^.value$` as our match key.

Some examples

Basic example where we look for any *value* that is *>= 90*

```
alert:
    type: "agg_match"
    field: '^.*value$'
    relation: ">="
    qty: 90
```

Or if we wanted to only look at only the first bucket, for a value `>= 20`

```
alert:
    type: "agg_match"
    field: '^\.buckets\.0.*value$'
    relation: ">="
    qty: 20
```

3.5 Alert Actions

Actions are what is taken after the Tale has met its alert threshold.

You can also have multiple actions per Tale. In our example Tale, you can we have two actions configured, one to send Emails, and one to send the alerts to Pager Duty as well.

3.5.1 Email

Probably the most common alert action. Tattle sends a formatted, HTML email to recipient(s)

The email server properties are stored in `$TATTLE_HOME/etc/tattle/tattle.yaml`, so please set that up first before you proceed with email alerts

Tale Examples:

Example

```
action:
    email:
        # Optional - We can enable or disable this action with this flag
        enabled: 1
        # Required - Who the email should go to
        to: [ 'alerts@company.com', 'manager@company.com' ]
        # Optional - If we should send a sperate email for every match.  If this is␣
↪not set, then the all of the results are sent in one email
        once_per_match:
            # The match key, is the part of the result we use our primary key for␣
↪sperating the results in seperate emails
            # In this case its "key" since its the key of the aggregation.  In our␣
↪case this will be the hostname
            # If we had 4 hosts that matched then we would have 4 seperate emails. ␣
↪Tattle will append the 'match_key' to the subject of the email as well
            match_key: "key"
        # Optional - A link to a external url to be shown in the email
        client_url: 'https://someapp.company.com'
        # Optional - kibana4_dashbaord to link to a kibana dashbaord.  When using␣
↪this, Tattle will add the times from the Tale into the dashboard link, note this␣
↪works for kibana4 dashbaords only
        kibana4_dashboard: 'http://kibana.company.com/app/kibana#/dashboard/
↪OurAwesomeDashboard'
```

Email are generated from a template via the Jinja Templating framework. By default the email template is located in `$TATTLE_HOME/usr/share/templates/html/email.html` .

You can use your own template(s) if you wish. Just specify `template_dir` and `template_name` in `$TATTLE_HOME/etc/tattle/email.yml`.

Example `$TATTLE_HOME/etc/tattle/email.yml`

```
server: 127.0.0.1
port: 25
default_sender: 'tattle@dev.local'
subject_prefix: 'Tattle PROD - '
# Specify where we are storing our Jinja tempaltes for our email
template_dir: /some/dir/with/my/templates
# Specify which template name we need to use in our template_dir
template_name: my_custom_email_template.html.j2
```

If you want to know more about Jinja, checkout the Jinja Docs

3.5.2 Script

The `script` alert action allows you to specify a script to run when the alert is fired/triggerd. When Tattle fires off the script, it passes in the results from the alert, the Tale definition, and the TQL query intentions for use within the script.

When the script is called, three arguments are passed in to, each argument will contain JSON as its data.

Arguments

- `$1` - The results, or matches from the alert

- `$2` - The Tale details that was responsible for triggering this alert
- `$3` - The TQL Query intentions

Your script must be in `$TATTLE_HOME/bin/scripts` and must be executable.

Note: The script will run as whatever user Tattle runs as. For example if you run Tattle under a user called *tattle*, then the script will run as the user *tattle*.

Here is an example script that will echo out each of the ARGV's

```
#!/bin/bash
echo 'RESULTS:'
echo $1

echo 'TALE:'
echo $2

echo 'INTENTIONS:'
echo $3
```

3.5.3 Pager Duty

Another very common use for Tattle is to send its alert direclty to Pager Duty.

Pager Duty alerts can be setup to Service Key, as defined in Pager Duty itself. The service Key definitions can be stored in the `$TATTLE_HOME/etc/tattle/pagerduty.yaml` and can be referenced in the action by thier title.

Example `$TATTLEHOME/etc/tattle/pagerduty.yaml`

```
TattleAlerts:
    service_key: "<service key>"
DataSystems:
    service_key: "<service_key>"
WebSystem:
    service_key: "<service_key>"
```

Example Tale action

```
action:
    pagerduty:
        # Optional - We can enable or disable this action here
        enabled: 1
        # Required - The name of the service key to use, as defined in pagerduty.yaml
        service_key: "TattleAlerts"
        # Optional - The URL to specify for the 'View In' part of Pagerduty.  This␣
↪could be Kibana dashboard or any web application you wish
        client_url: "https://kibana.company.com/app/kibana#/dashboard/
↪OurAwesomeDashboard"
        # Optional - kibana4_dashbaord to link to a kibana dashbaord which will be␣
↪shown in 'View In'.  When using this, Tattle will add the times from the Tale into␣
↪the dashboard link, note this works for kibana4 dashbaords only
        kibana4_dashboard: 'http://kibana.company.com/app/kibana#/dashboard/
↪OurAwesomeDashboard'
        # Optional - If we should compile seperate pagerduty alerts for each match. ␣
↪If this is not set, then the all of the results are sent in one PD alert
        once_per_match:
```

```
            # The match key, is the part of the result we use our primary key for␣
↪sperating the results in seperate PD alerts
            # In this case its "key" since its the key of the aggregation.  In our␣
↪case this will be the hostname
            # If we had 4 hosts that matched then we would have 4 seperate Pagerduty␣
↪alerts.  Tattle will append the 'match_key' to the subject of the Pagerduty alert␣
↪as well
            match_key: "key"
```

3.5.4 Microsoft Teams

Tattle supports posting a MessageCard to the MS Teams channel of your choice via a webhook

Before you proceed, please fill in the defaults (which can be overridden on a per Tale/Action basis) in `$TATTLE_HOME/etc/tattle/msteams.yml`

Default

```
default:
    # The Webhook URL for the MS Teams channel you wish to post to
    webhook_url: https://outlook.office.com/webhook/.....
    # What to prefix the Tale Title with
    title_prefix: 'Tattle -'
    # If we should verify SSL or not, useful when behind some corporate proxies,␣
↪default is True
    # ssl_verify: true
    # Proxy address if needed
    # proxy: http://user:password@some.corp.proxy.com:80

channel_alisases:
    My Teams Alert Channel Name:
        # Webhook url for your channel
        webhook_url: https://outlook.office.com/webhook/.....
    Some Other Channel Name:
        webhook_url: https://outlook.office.com/webhook/.....<some other channel name>
    AnotherAlertChannel:
        webhook_url: https://outlook.office.com/webhook/.....<some other channel name>
```

Example

```
action:
    msteams:
        enabled: 1
        # Optional teams_channel - The alias of the teams channel you wish to send␣
↪the alert to
        # This can be a list of channels if you wish to sent to multiple channels at␣
↪the same time
        # Note the channel aliases are defined in $TATTLE_HOME/etc/tattle/msteams.yml
        teams_channel:
            - My Teams Alert Channel
            - Some Other Channel Name
            - AnotherAlertChannel
        once_per_match: # Optional
            # The match key, is the part of the result we use our primary key for␣
↪sperating the results in seperate PD alerts
            # In this case its "key" since its the key of the aggregation.  In our␣
↪case this will be the hostname
```

```
            # If we had 4 hosts that matched then we would have 4 seperate Pagerduty␣
↪alerts.  Tattle will append the 'match_key' to the subject of the Tale Title as well
            match_key: "key"
        kibana4_dashboard: 'http://kibana.company.com/app/kibana#/dashboard/
↪OurAwesomeDashboard'
```

Or if you wanted to post this message to a differnt channel, simply overwrite the *webhook_url* that points to your desired channel in the action. Example

```
action:
    msteams:
        webhook_url: https://outlook.office.com/webhook/<my_other_channel>
```

3.5.5 Slack

Tattle has support for posting its alert into a Slack channel of your choice

Before you proceed with Slack alerting, please fill in the defaults (which can be overridden on a per Tale/Action basis) in `$TATTLE_HOME/etc/tattle/slack.yml`

In the defaults section, fill in the info with whatever makes sense for your envionment. As stated, these can be overridden on a per-tale/action basis if you wish as well (example below). Default

```
default:
    webhook_url: 'https://mywebhook.slack.com'
    channel: 'eng-alerts'
    username: 'Tattle'
    msg_color: 'danger'
    title_prefix: 'Tattle -'
    emoji: ':squirrel:'
```

Then in your Tale action, just specify slack Example

```
action:
    slack:
        enabled: 1
        once_per_match: # Optional
            # The match key, is the part of the result we use our primary key for␣
↪sperating the results in seperate PD alerts
            # In this case its "key" since its the key of the aggregation.  In our␣
↪case this will be the hostname
            # If we had 4 hosts that matched then we would have 4 seperate Pagerduty␣
↪alerts.  Tattle will append the 'match_key' to the subject of the Tale Title alert␣
↪as well
            match_key: "key"
        kibana4_dashboard: 'http://kibana.company.com/app/kibana#/dashboard/
↪OurAwesomeDashboard'
```

Or if you wanted to post this alert in another channel from the default, and change the default emoji

```
action:
    slack:
        emoji: ':fire:'
        channel: someotherchannel
```

By default slack will map the severity of an alert to an emoji as specified in `$TATTLE_HOME/etc/tattle/slack.yml`. This can be customized by changing the regex keys in the `emoji_severity_map` to match your severity system in your environment. The default is

```
emoji_severity_map:
    'crit|5': ':fire:'
    'high|4': ':rage:'
    'med|3': ':grimacing:'
    'low|2': ':disappointed:'
    'info|1': ':sunglasses:'
```

If you want to override the defaults, you can do so on a per Tale/action basis like so:‘

Example

```
action:
    slack:
        # Optional - if the action is enabled or not ( default is True )
        enabled: 1
        # Required - The webhook url to use for the slack intergration
        webhook_url: 'https://hooks.slack.com/services/TTAsdfQ/asdfasdf/asdfasdfasdf'
        # Required - the slack channel to post the alert to
        channel: 'engineering-channel'
        once_per_match: # Optional
            # The match key, is the part of the result we use our primary key for␣
↪sperating the results in seperate PD alerts
            # In this case its "key" since its the key of the aggregation.  In our␣
↪case this will be the hostname
            # If we had 4 hosts that matched then we would have 4 seperate Pagerduty␣
↪alerts.  Tattle will append the 'match_key' to the subject of the Pagerduty alert␣
↪as well
            match_key: "key"
        # optional - A link to a external url which will be displayed in the Title of␣
↪the Slack alert
        client_url: 'https://someapp.company.com'
        # optional kibana4_dashbaord to link to a kibana dashbaord.  When using this,␣
↪Tattle will add the times from the Tale into the dashboard link, note this works␣
↪for kibana4 dashbaords only
        kibana4_dashboard: 'http://kibana.company.com/app/kibana#/dashboard/
↪OurAwesomeDashboard'
```

3.6 Multiple Tales

Its often useful to group Tales by their purpose. For example, you might want to group your *Nginx Access* Tales together, your *Nginx Error* Tales sperately, and your *Securelog* Tales together. Lets say we have 20 differnt *Nginx* Tales, and 10 different *Securelog* Tales; that would mean we would have have at least 30 seperate *Tale* `.yaml` files in our `$TATTLE_HOME/etc/tales` directory. As you can imagine, the more you use Tattle, the more unwieldy this can get.

Luckily Tattle allows you to define multiple Tales in one `.yaml` file to alleviate this issue. Using the example below, you can see how we grouped two *Nginx* Tales into one file. There can be as many Tales as you want this one in one `yaml` file.

3.6.1 Syntax

multi_tale_example.yaml

```
tales:
    -
        <tale #1>
    -
        <tale #2>
    -
        <tale #3>
```

3.6.2 Example Multi Tale

Example for NGINX logs

```
tales:
    # Tale 1
    -
        name: "NGINX 502 Spike"
        description: "A high number of 501's have occured in our NGINX logs"
        severity: "Criticial"
        tql_query: "status:502"
        index: "nginx-access-*"
        enabled: 1
        schedule_interval: "1m"
        timeperiod:
            start: "now-1m"
            end: "now"
        alert:
            type: "frequency"
            relation: "ge"
            qty: 10
            realert: "15m"
            return_matches: false
        action:
            email:
                enabled: 1
                to: 'alerts@mycompany.com'

    # Tale 2
    -
        name: "NGINX 404 Spike"
        description: "A high number of 404's have occured in our NGINX logs"
        severity: "Medium"
        tql_query: "status:404"
        index: "nginx-access-*"
        enabled: 1
        schedule_interval: "1m"
        timeperiod:
            start: "now-1m"
            end: "now"
        alert:
            type: "frequency"
            relation: "ge"
            qty: 400
            realert: "15m"
```

```
            return_matches: false
        action:
            email:
                enabled: 1
                to: 'alerts@mycompany.com'
            pagerduty:
                enabled: 1
                service_key: "TattleAlerts"
                once_per_match:
                    match_key: "key"
```

CHAPTER 4

Tattle Query Language (TQL)

The Tattle Query Language (TQL) is a short hand syntax used for building Elasticsearh DSL queries.

4.1 Introduction

When I was building the first verion of Tattle (project Bluenote), I noticed a pattern in the Elasticserach queries I was making to find events. For example, They all had *time windows*, they all had a `query_string` (Lucene query), a `size`, a `to` and `from`, etc. So I thought it would be nice to have a short hand syntax to represent an Elasticsearch query that was easy to understand and write, which in turn would make managing hundreds or thousands of Tales easier. Through some experimentation I came up with TQL which aims to do just that.

4.2 Examples

Let uses NGINX logs for our example (if you are familiar with Apache access-combined logs, this will be similar as well).

Lets get a count of events that have a *status code* of 502, grouped by their respective host for the past hour.

The Elasticsearch query will look like

```
{
    "query": {
        "bool": {
            "must_not": [
                {
                    "query_string": {
                        "query": ""
                    }
                }
            ],
            "must": [
                {
```

```
                    "query_string": {
                        "query": "status:502"
                    }
                },
                {
                    "range": {
                        "@timestamp": {
                            "to": "now",
                            "from": "now-1h"
                        }
                    }
                }
            ]
        }
    },
    "_source": {
        "include": [
            "*"
        ]
    },
    "from": 0,
    "aggs": {
        "hostname": {
            "terms": {
                "field": "host.raw"
            }
        }
    },
    "size": 0
}
```

The TQL equivilent would be

```
status:502 | terms name=hostname, field=host.raw
```

Note: The times for these queries not handled in TQL, but instead are determined in the alert/tale. For demo purposes, we left the times in the DSL query.

4.3 Lets break it down

Anything before the first `|` (pipe) is going to be Lucey Query Syntax. There are many tutorials out there that can explain it much better than here, but in essance what we did was run a lucene query for any logs/events that have a `status` of `502`. But just keep in mind, that any lucene query you would use in Kibana or Elasticsearch you will put here.

To the right of our first `|` is our `terms` aggregation. In this case we are running a `terms` agg on the field `host.raw`, and we are naming that aggregation `hostname`.

If we ran this against our Elasticsearch cluster, we would get results similar to the following:

```
{
    "aggregations": {
        "hostname": {
            "buckets": [
                {
                    "key": "host1.mycompany.com",
```

```
                    "doc_count": 37
                },
                {
                    "key": "host2.mycompany.com",
                    "doc_count": 29
                },
                {
                    "key": "host3.mycompany.com",
                    "doc_count": 16
                }
            ],
            "sum_other_doc_count": 0,
            "doc_count_error_upper_bound": 0
        }
    },
    "timed_out": false
}
```

In this case we have three hosts in the `hostname` aggregation that have had 502 errors in the last hour, `host1.mycompany.com` (37 events), `host2.mycompany.com` (29 events), `host3.mycompany.com` (16 events).

4.4 Nesting

Aggregations in Elasticserach can be nested, and this is the default behaviour in TQL. You can nest as many aggregations as you with by using `|`.

In this example, we can want to average a metric and group it by the host.

```
metric:DatabaseConnections | terms field=database.raw, name=DB_Name | avg␣
↪field=connections, name=connection_avg
```

Here we used two aggregations, a *terms* and and *avg*. The *avg* aggregation will nest below the *terms*. Here are the aggregations for the Elasticsearch Query TQL would generate:

```
{
    "aggs": {
        "DB_Name": {
            "terms": {
                "field": "database.raw"
            },
            "aggs": {
                "connection_avg": {
                    "avg": {
                        "field": "connections"
                    }
                }
            }
        }
    },
    "size": 0
}
```

4.5 Mappings

Generally all of the aggregations available in Elasticsearch can be used in TQL. Simply use the syntax `<agg_name> <arguments>` - example `terms field=host.raw, name=hostname, order={ "hostname": "desc" }, cardinality field=author_hash, precision_threshold=100, stats field=grade`

However this rule applies all but one name, `fields`. The `fields` name is special to TQL and will display only the fields you want to see in your tale/alert.

For example, let use NGINX events. They can have many different fields, but we might only want to see one or two fields in our alert. We can use the *fields* argument to help with that

```
status:502 | fields @timestamp, message
```

In this example we would only see two fields, the `@timestamp` for the event, an the `message` for the event.

Read up more on Elasticsearch Aggregations here: https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations.html

4.6 Ordering

Certain Elasticserach aggregations, such as *terms* can order your results. You can pass along your order syntax as documented by Elasticsearch into the *order* argument

```
....| terms field=database.raw, name=database, order=[ { "database.raw": "desc"}, {"_
↪count": "desc"} ]
```

4.7 Scripting

Like ordering, certain Elasticsearch aggs can contain *scripts* to enhance their values during search time. Much like the *order* function, these are evaluated just like they are in the Elasticsearch docs.

```
.... | stats name=grades_stats, script={"inline": "_value * correction", "params": {
↪"correction": 1.2}}
```

An example demonstrating inline scripting with the choice of language, and converting bytes to MB

```
host.raw:app-servers* | avg name=mb_sent, script="doc['body_bytes_sent']/1024/1024",␣
↪lang=expression
```

Note: Groovy inline scripting is disabled by default in modern Elasticsearch clusters. As always, check out the scripting documentation on elastic.co for more examples: https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-scripting.html

Note: Script style syntax should be evauluted for most, if not all aggregations. For example, `percentiles field=status percents=[85, 99, 99.9]` will be evauated into the correct JSON needed for the ES query

CHAPTER 5

Logging

When Tattle runs, it emits log messages about what it is doing.

By default, these logs can be found in `$TATTLE_HOME/var/log/tattle.log`

If you would rather Tattle log to another directory, simply overwrite the `logging_directory` in `$TATTLE_HOME/etc/tattle/logging.yml`

5.1 Levels

The default setup for logging is to log only `WARN` messages and below. You can turn on `debug` if you wish by editing `$TATTLE_HOME/etc/tattle/logging.yaml` and changing the level.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search