2016

Performance and Health Monitoring and Analysis of Hive Scales Portal Web Application

Ronald J. Slocum Jr.
Grand Valley State University

Performance and Health Monitoring and Analysis of Hive Scales Portal Web Application

By
Ronald J. Slocum, Jr.

April, 2016

Performance and Health Monitoring and Analysis of Hive Scales Portal Web Application

By
Ronald J. Slocum, Jr.

A project submitted in partial fulfillment of the requirements for the degree of

Master of Science in

Computer Information Systems

At

Grand Valley State University

April 28, 2016

Dr. Jonathan Engelsma **April 2016**

Table of Contents

Abstract

Honey bee health and colony collapse disorder (CCD), a term coined in 2006, have been drawing global attention as bees are dying at a rapid rate. The University of Maryland reported that beekeepers across the US lost more than 40 percent of their honey bee colonies during the year spanning April 2014 to April 2015[1]. With seventy out of the top 100 human food crops, which supply about 90 percent of the world's nutrition, being pollinated by bees [4], this puts our food supply at serious risk!

Formed from a grant from the US Department of Agriculture, the Bee Informed Partnership (BIP) is a collaborative effort to address the declining honey bee population. A major part of this effort is being led by GVSU through the Hive Scales Portal web application.

This application aggregates and stores data, collected from hive scales around the world, for researchers and scientists with the intent of identifying patterns that lead to best practices for caring for bees.

As the quantity of contributing hives scales increase, the hive scale portal will need to scale proportionally.

Knowing when or how much to scale is a guessing game without the proper visibility.

The purpose of this project is to analyze the application and ensure it can scale. The key piece of this analysis is to integrate the necessary tools to ensure site health and performance visibility is available for continued monitoring.

Introduction

Honey bee health and colony collapse disorder (CCD), a term coined in 2006, has been drawing global attention as bees are dying at a rapid rate. The University of Maryland reported that beekeepers across the US lost more than 40 percent of their honey bee colonies during the year spanning April 2014 to April 2015[1].

Seventy out of the top one hundred human food crops—which supply about 90 percent of the world's nutrition—are pollinated by bees [4].

Solving this problem is going to require a similar approach used to solve many of today's most complex problems. "The trend right now is to solve hard problems with big data," says Dr. Jonathan Engelsma, who helped develop the Hive Scales Portal for the Bee Informed Partnership [2]. Formed from a grant from the US Department of Agriculture, the Bee Informed Partnership (BIP) is a collaborative effort to address the declining honey bee population.

The Hive Scale portal is experiencing success having already formed partnerships with 3 scale manufacturers [3]. The current barrier to large scale adoption is the cost of the hive scale. The lowest cost solution is in the $500 to $600 range. Although, this is down from $1,500 only a few years ago, at $500, it is hard to justify that large of an investment for hobbyists or semi-professionals. As the cost of the monitoring equipment drops, the hive scale portal onboarding is expected increase. Within the BIP community, it is believed that the tipping point to large scale adoption will be when the monitoring equipment reaches the $100 range.

As the quantity of contributing hives scales increases, the hive scale portal will need to scale proportionally to handle the increased load.

The purpose of this project is to analyze the hive scale portal application and ensure it can handle the anticipated growth in traffic.

Continuing to support the efforts of bee keepers, researchers, and scientists through the hive scale portal is dependent upon the applications ability to handle increasing amounts of data and transactions. If the site is unable to handle the load, users will move away from the site, data could be missed, and researchers and scientists working to solve this problem could be denied the resources they need.

Knowing how to answer questions about your application's ability to scale requires a great deal of background. Having intimate knowledge of your application's infrastructure is essential. Having an understanding of common pain points will help focus the evaluation.

While researching several aspects of this project, it became apparent that the most critical piece of information was not available. In terms of health and performance, the hive scale portal was close to a black box. Upon this realization, the focal point of this project evolved into ensuring site health and performance was visible. Without this visibility, making decisions on when the site required scaling and how to scale could not be made.

Executing this project involved three distinct chunks of effort. The first was researching what it means to scale. I needed to know what questions I should be trying to answer. Where are the common problem areas for web applications? What should be measured?

The second part was to determine how to create a benchmark for the site. There are many options, and because we are dealing with large quantities of data, there are costs associated with the storage of this data and processing power to collect, aggregate, and display it. It was important to keep the costs for this solution low.

After understanding what needs to be looked at, determining how to look at it, I then needed to analyze the data and take action.

Background and Related Work

The Hive Scales Portal is an existing application hosted on Heroku. Heroku is a cloud based platform as a service (PaaS) supporting many of the most popular software development languages, including Ruby on Rails (RoR). The Hive Scales Portal is written in RoR. Heroku enables developers to deploy and manage their application without having to think about managing the hardware, infrastructure, or even the operating system running their application. This allows developers to focus on writing code. Heroku is built on virtual machines in Amazon Elastic Compute Cloud (EC2), specifically, in the US-East Region.

An alternative to Heroku was introduced in 2011 by Amazon Web Services (AWS): AWS Elastic Beanstalk (EB). Utilizing EB over Heroku provides the advantage of having the entire suite of AWS services available and easily integrated. For instance, some of the site health and performance information that was missing from Heroku is available off the shelf from AWS with AWS CloudWatch and CloudTrail.

For deep log file analysis, both AWS and Heroku require another solution. Heroku makes it incredibly easy to pipe the application logs to a defined end point. For the chosen solution, directing log files from Heroku to an EC2 instance running Logstash could be turned on by executing 1 command:

```
$ heroku drains:add syslog://logs.example.com -a myapp
```

I wasn't able to locate a comparable option for AWS EB or EC2 to direct application logs to a custom end point using AWS services. Advantage Heroku. In general, Heroku requires fewer configuration steps and is thus easier to get up and going. Setting up IAM (Identity and Access Management) can be a cumbersome and lengthy process.

Through the Heroku Add-On market, there are services available to provide the visibility necessary to diagnose and monitor the site. New Relic, Librato, and Logz.io are the most popular. New Relic and Librato are useful for understanding health metrics of the site. They do not offer the log file deep dive capabilities that are necessary to troubleshoot a spike in API latency. Logz.io offers a complete hosted ELK stack solution. ELK stands for Elasticsearch, Logstash, and Kibana.

Each component of the ELK stack was conceived as a separate project and eventually, all components and their creators were brought together under the umbrella of elastic.co.

Elasticsearch was released in 2010 by Shay Banon as a distributed search engine built on top of Lucene. It is JSON based, supports REST APIs, and scales easily.

Logstash was created by Jordan Sissel. It is a powerful utility that parses, manipulates, and forwards log files from multiple source.

Kibana was created in 2011 by Rashid Khan. It is a browser based front end for Elasticsearch that helps you view all of the data in Elasticsearch through charts, graphs, and custom queries.

Thousands of organizations worldwide use these products for an endless variety of business critical functions [12].

Logz.io provided the easiest and most complete solution but the highest cost. Because the ELK stack is open source, we can implement this solution ourselves without Logz.io.

Each solution has an associated monthly cost. Of the pay services, Logz.io had the most functionality, but it also had the highest costs. Manually setting up the ELK stack on EC2 offered the functionality of Logz.io but a cost lower than all of the other solutions.

Set up	Monthly Cost	Notes
ELK on EC2	$16.44	Complicated setup. Maintenance required.
New Relic	$49	Metrics only, no log indexing, instruments code causing higher application memory usage.
Librato	$19	Metrics only
Logz.io	$89	Expensive!

Program Requirements

The requirement of this project was to evaluate the health and performance of the BIP Hive Scale portal to ensure it can handle on boarding significantly more hive scales.

In order to understand if the Hive Scale portal was capable of scaling with higher demand, I needed to research and understand what is necessary for a site to be able to scale. What types of limitations cause sites to fall over when under heavy load?

This phase required a deep dive into Heroku, the Hive Scale Portal's hosting platform. Because Heroku manages the infrastructure running this site, we were at the mercy of its scaling capabilities. Knowing the features and limitations of the platform were necessary to address the requirement of this project.

The second phase of the project was to benchmark the site. This phase involved researching and evaluating various method for benchmarking the site. Once a decision was made on the method for benchmarking, the tools needed to be deployed and configured.

The third phase of this project was to analyze the data gained from the benchmarking tools and take action to improve the site.

Implementation

I began this project by taking a deep dive into scaling a website. I needed to understand how to perform an analysis of a web application before I could asses the health of the Hive Scales Portal. I focused my research efforts specifically on how to scale a web app on Heroku.

Heroku encapsulates your web application in virtual machines called "Dynos". These Dynos run on AWS EC2 instances. Each Dyno represents a single process. Dynos come in a few flavors:

Type	Memory	Price
1X Standard Dyno	512 MB	$0.05/hr.
2X Standard Dyno	1 GB	$0.10/hr.
Performance Dynos	14 GB / 8 CPUs	$0.80/hr.

For the Hive Portal, we use (1) 1X Standard Dyno. Management of the Dynos occurs through the Heroku Dyno Manifold. The dyno manifold can be thought of as a giant, distributed, fault-tolerant, horizontally scalable computer in the sky, capable of running a nearly infinite number of diverse programs in the cloud [5].

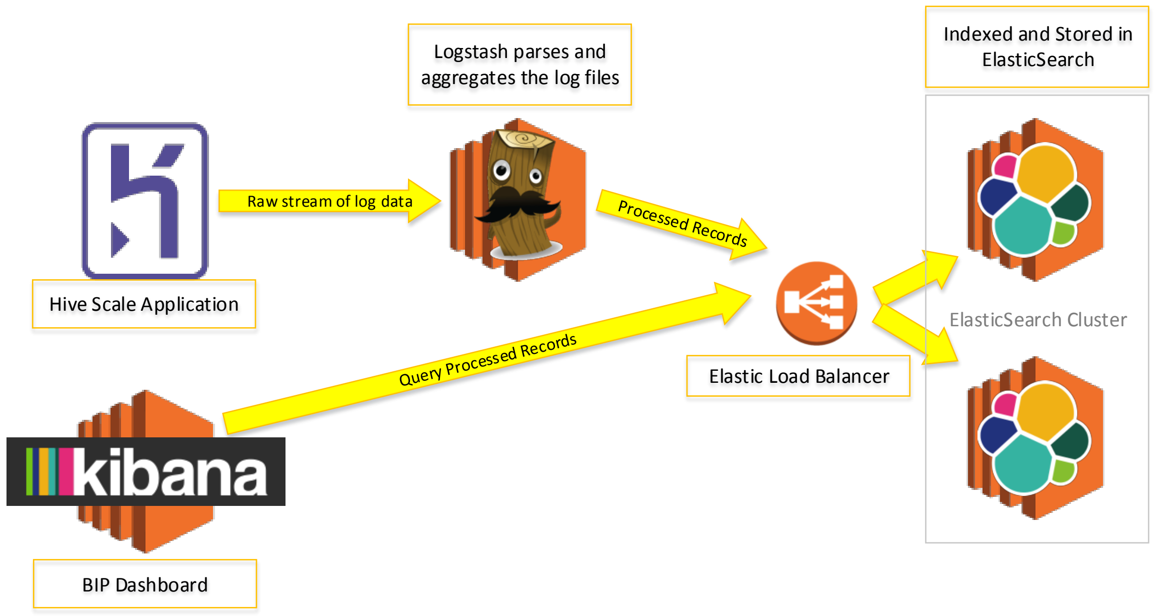
With the available configurations, it is possible to scale the Hive Portal vertically by selecting the 2X Standard or the Performance Dyno. Vertical scaling is necessary when a single process requires a large amount of resources. If that single process can be broken into smaller processes and executed in parallel, then horizontal scaling is a much better solution. Instead of adding memory or CPUs to handle the single process, additional Dynos can be added to divide and conquer the long running process. The Heroku dashboard can be configured to use 100 Dynos.

Once I had an understanding of the infrastructure the site was built upon, I needed to obtain metrics for the site. I knew the log files were filled with critical data to paint a clear picture of the performance of the site. After exploring the available add-ons in the Heroku marketplace, I tried out Papertrail. It has an archiving feature that would export a day's worth of logs to S3. My plan was to utilize AWS Lambda to process the log files and put them into a searchable format.

After configuring Papertrail to get the logs into AWS, I started receiving emails from Papertrail indicating that we had exceeded the maximum of 10 MB per day. Based on the rate of log generation, Papertrail was able to estimate that we were going to produce over 300 MB of log files for that day. At this point, I had very little idea of what was happening. I extracted the 10MB of logs and started searching for something. I was able to determine that the site was getting hammered by registration requests from one scale. There were over 6,000 requests reaching our site in that short hours we were able to collect logs. What's worse is that Heroku was dropping almost 40,000 more requests every few seconds.

At this point it became apparent that we needed much more visibility into our logs not just for this project but for the immediate health of the site. We are not certain how long the site was getting hammered and how many legitimate messages were dropped.

I began researching other methods for processing log files. I learned that gaining visibility to log files is a very typical problem with an elegant solution. I was elated when I stumbled upon the ELK stack. I spent an entire weekend learning everything I could about the ELK stack. I then took on the task of implementing it. I started setting up Logstash on my Mac book but then quickly ran into issues getting the logs from Heroku. I decided to use Amazon Web Services EC2 to host the ELK stack. Set up went quite smooth. After getting everything set up, I noticed that AWS has their own managed Elasticsearch solution. Being drawn to the word "Managed", I quickly deployed AWS ES. AWS ES is comprised of 2 EC2 instances. The first is running Elasticsearch, and the second is running Kibana. Their solution does not include Logstash. Their recommendation is to use AWS Lambda in place of Logstash. When setting up my original Elasticsearch EC2 instance, I found and installed a plugin called "Head". This plugin provided me with tools to manage the nodes and indexes in Elasticsearch. Unfortunately, AWS ES does not support plugins. Because of this, and because all of the resources I had were pointing me toward a self-hosting solution as opposed to the managed AWS ES, I decided to ditch AWS ES. The solution was architected as follows:



It was beneficial to deploy these EC2 instances in the US-East Region as that is where the Heroku platform exists. Having all components in the same region reduces latency between Heroku and the BIP Dashboard, which provides more up to date information.

Another utility from elastic.co is Curator. This is not an open source tool like those in the ELK stack are. It is free, but the source is owned by elastic.co. This tool can be installed on each Elasticsearch node and configured to purge data older than X days. I installed and configured Curator to retain 60 days of data in Elasticsearch. This may be overkill and we could reduce it to 30 days. If space or latency in ES become an issue, we can easily tweak that value.

Now I had the log data available and the ability to search logs in real time.

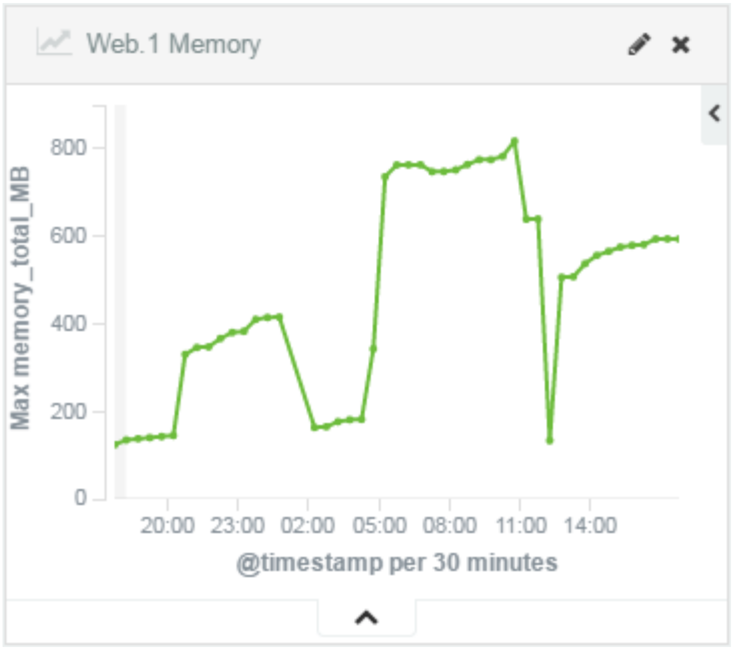
Initially, I used Kibana to simply search the logs. Using the data in the logs, it became apparent that we should rework the registration protocol. We had occasions where the multi-step registration process wasn't fully completed and the consequence was repeated and unnecessary hits on the site.

Once we had a plan together for Sam to rework the registration protocol, I focused on creating visualizations in Kibana. While researching, I stumbled across an Add-On in the Heroku marketplace called Librato. The free version of this plugin provided some nice graphs. Unfortunately, the graphs were limited to only 60 mins of data. Using the Librato visualizations as a guide, I was able to reconstruct the same graphs. The advantage to using the graphs in Kibana over Librato was that they were much more interactive and spanned a much larger window of time. Additionally, Librato does not offer the deep dive into the log files like Kibana does.

Prior to being able to construct the graphs, I needed to gain intimate knowledge of the Logstash configuration. Most of the data I needed was present in the logs, but it had to be parsed and manipulated to make it available for Kibana to plot. This was a painful process. My knowledge of Regular Expressions was a bit rusty. I spent a long time learning the logstash grok filter, aided largely by the grok debugger [13].

The graphs from Librato were a nice start. As I analyzed the data, I found holes in the visualizations. For example, specifically charting the memory used by the Web.1 dyno was important to address the memory leak.

Memory Vastly Exceeded is an aggressive error message issued by the Heroku platform when the application is consuming excessive (over 200%) swap space. If this error occurs, the platform soon kills the dyno. Our dyno is configured for 512MB. We discovered that we can quickly get Heroku to start issuing these errors when a user requests scale data for the graph on the hive detail page. The memory usage shoots up when this happens.



Each jump in the graph represents a request for hive data for the chart. Each drop represents the dyno being killed. It was either killed by a manual restart or the result of a bandaid we installed called “Puma Worker Killer”. This open source ruby gem was configured in our site to periodically kill the web server. The web server will automatically be restarted. When the restart occurs, the memory is returned to safe levels. This was implemented as a temporary solution until we solve the memory leak.

I learned a lot about the ruby garbage collector while trying to solve the memory leak. I adjusted the parameters associated with the garbage collector by adding configuration parameters to Heroku to encourage the GC to trigger more frequently. I was able to confirm triggering or not triggering garbage collection had no impact on the problem. I went as far as to add a button to the admins page that would allow me to explictly start the garbage collector.

Here are some of the other attempts made to solve the memory issue:

- Utilize GraphPresenter in view instead of using instance variable – The point where we are seeing allocated objects remain after garbage collection is in the GraphPresenter. When assigning the graph data to an instance variable, the objects would show up in our debugging tools. If I changed the instance variable to a local variable, the objects would not show up in our debugging tools as still allocated. I attempted to bypass the instance variable by having the GraphPresenter feed the view directly. This did not help our situation.

- We had a custom time formatter defined for use with the graph. While looking through our source code I came to a location where we we defined a custom time format. I noticed we were using a lambda to define this custom time format. I know lambda's can be tricky if used incorrectly in C#.

So I read a bit on ruby lambdas and inferred from an article[14] explaining the difference between lambdas and procs, that lambdas in ruby also hold on to closure like in C#. I searched through the rails documentation and found an example that specifically called out to use a proc. We are using a lambda. One of the main differences between lambda and proc is how they handle closures. I strongly believed that this could be what was causing our garbage collector to ignore the objects that were being created. If we have lambdas that are holding on to closure scope causing reference chains to build and build, we would never be able to garbage collect them. Unfortunately, this did not help.

- Remove Turbolinks – The Turbolinks implemenation persists a session state that has potential to hold on to memory references. Although this would result in client side memory issues, I was at the end of my ropes so I tried it. The memory leak persists today.

The current API latency and memory requirements of the 1X Standard Dyno are sufficient for the Hive Portal's current load. If latency becomes a concern, we'll need to understand the cause of the latency. Scaling the site horizontally by adding more Dynos or moving up to the 2X Standard Dyno should address increased resource demands.

We are well below our database capacity using only 366 MB of 64 GB. In the near future, we will need to define a strategy for offloading or archiving data.

We chose open source solutions to implement the system health and performance monitors. This is a popular and elegant solution. The way it was set up results in a very reasonable cost structure.

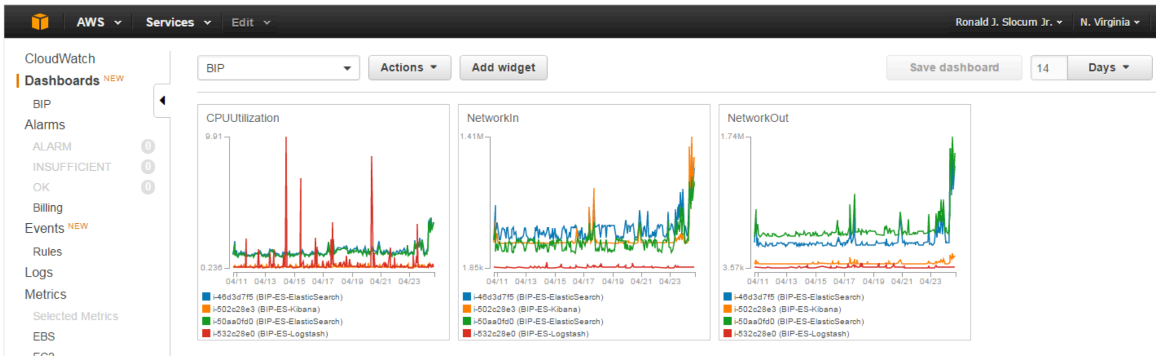
The ELK stack cost as implemented: $16.44/month (compared to $89/month for identical services from Logz.io).

Current system is correctly configured under current load: TP95 API Latency is 135ms, TP99 235ms.

With the Heroku platform, the site is prepared for scaling horizontally and vertically when the need arises.

With the BIP Dashboard providing real time monitoring, historical trending, and full-text log search capabilities, we can now see when that need to scale is approaching, ensuring our beekeeper can continue sending us their data, and researchers and scientists can continue using the data to save our bees!

Having deployed the ELK stack on AWS EC2 instances, we have the ability to integrate AWS CloudWatch. I created a dashboard in CloudWatch specifically to monitor the load on these instances.



In the CPUUtilization graph above, you'll notice the red line has a few major spikes. The red line represents the EC2 instance running Logstash. Even with these few anomalies, the CPU usage never exceeded 10%. With this information, I was able to deduce that the current hardware configuration is a bit over kill. The EC2 instances are t2.micro running Amazon Linux with 1 CPU and 1 GB memory. AWS offers one smaller tier, t2.nano. This instance is configured with 1 CPU and 512 MB memory.

For this project, I am able to take advantage of AWS's "Free tier". For users new to AWS, Amazon provides 1 year of eligibility in their free tier. Because of this program, I selected t2.micros as my EC2 instances as they fall under this program. The smaller t2.nano is not part of the free tier.

Results, Evaluation, and Reflection

I invested a lot of time into solving a memory leak on the site. Unfortunately, the memory leak remains. I was not very efficient with this process. The silver lining is that with the pain of trying to fix this issue, I managed to gain a much deeper understanding of Ruby and Rails. This type of knowledge can rarely be built from any other type of situation.

One thing that would speed things up is to learn how to deploy the BIP site on my own. I leaned on Sam to deploy the site. This was intentional in some cases to ensure at least another set of eyes viewed the changes. However, there were times when he was not available to deploy.

My recommendation to anyone building a web application is to consider building in metrics from the start. Once they are in place, monitor them regularly, especially after a change.

Conclusions and Future Work

While monitoring the system, we discovered a memory leak in the BIP site. Several attempts were made to address this issue. Fortunately, the leak is easily repeatable. Unfortunately, at the time of writing this, the issue persists. Resolving this will continue to be a high priority.

After resolving the memory leak, I plan to install a utility from elastic.co that supplements the ELK stack with alarming. The name of the utility is Watcher. This utility is not open source, but it is provided for free by elastic.co. Having Watcher installed will enable us to set thresholds for latency, memory consumption, database size, etc. and configure alerts to be triggered. Anything that can be queried in Elasticsearch can be alerted on. These alerts can be in the form of emails, Slack messages, and more.

As another improvement, I would like to move the static assets (images, JavaScript, CSS, etc.) to AWS S3 to reduce some load on the site.

As mentioned earlier, the EC2 instances are currently oversized for the ELK stack. I plan to resize the EC2 instances. AWS makes this process pretty painless. I would also like to move these from my personal AWS account to one owned by BIP.

This project has been a roller coaster of frustration and elation. There were many challenges and roadblocks met along the way. Each of these obstacles presented an opportunity to gain intimate knowledge of something I had previously limited or no understanding of. The “ah ha” moments provided many highs.

I hope to continue working with the BIP team and the Hive Scale Portal. The technical challenges are interesting and the problem they are trying to solve is important.

Bibliography

1. “Nation's Beekeepers Lost 40 Percent of Bees in 2014-15.” Wright, Matthew. UMD, Right now. 3 May 2015. http://www.umdrightnow.umd.edu/news/nations-beekeepers-lost-40-percent-bees-2014-15

2. “Sweet local app aims to gather data on vanishing honeybees.” Kent, Steven. Rapid Growth. 19 June 2014. http://www.rapidgrowthmedia.com/features/061914honeybeeapp.aspx

3. “Hive Scale Program.” Bee Informed Partnership. https://beeinformed.org/hive-scale-program/

4. “Save the bees.” Green Peace. http://www.greenpeace.org/usa/sustainable-agriculture/save-the-bees/

5. Kemp, Chris. Professional Heroku Programming. Wrox, 4 February 2013.

6. Turnbull, James. *The Logstash Book*

7. Elastic.co documentation *https://www.elastic.co/guide/index.html*

8. Middleton, Neil & Schneeman, Richard. *Heroku Up and Running*: O'Reily Media

9. Logstach.net configuration *http://logstash.net/docs/1.4.2//configuration*

10. Stackoverflow.com *http://stackoverflow.com/questions/8330624*

11. Levy, Tomer. Logz.io *http://logz.io/blog/5-logstash-pitfalls-and-how-to-avoid-them/*

12. Sissel, Jordan. Elastic. https://www.elastic.co/webinars/introduction-elk-stack

13. Unknown. https://grokdebug.herokuapp.com/

14. Waxman, Adam. “What Is the Difference between a Block, a Proc, and a Lambda in Ruby?” http://awaxman11.github.io/blog/2013/08/05/what-is-the-difference-between-a-block/. 5 August 2013.

Appendices

Logstash Configuration

/etc/logstash/conf.d/logstash.conf:

```
input {
  file {
    path => "/tmp/logstash.txt" # This is only used for testing
  }
  tcp { #Heroku log drain is configured to point at this input
    port => "1514"
    tags => ["input_heroku_syslog"]
  }
}

# The filter section is used to parse, convert, tag, etc. the messages coming from Heroku
filter {
  if "input_heroku_syslog" in [tags] {
    if [message] =~ "Error\s[A-Z]\d+" { # Use Regex to find one of Heroku’s error codes
      mutate { add_tag => ["has_error"] } # Filter on this tag to easily find errors
    }

    if [message] =~ "app web.\d" {
      mutate {add_tag => ["api"]} # Indicates an API call
    }

    if [message] =~ "heroku router" {
      mutate {add_tag => ["router"]}
    }

    if "api" in [tags] {
      multiline { # API calls are multiline and need to be assembled into one message.
        pattern => "state=ready"
        negate => true
        what => previous
      }
      mutate {
        gsub => [
          # strip headers off on multi-lines
          "message", ".*\b( - - )", ""
        ]
      }
      mutate {
        gsub => [
          #pad all new lines
          "message", "\n", "
 " # This new line was necessary to get it to display correctly.
        ]
      }
      grok {
        match => ["message", "(?m)(?<completion_code>Completed\s[0-9]{3})"]
      }
      mutate {
        gsub => ["completion_code", "Completed ", ""]
      }
      grok {
        match => ["message", "(?m)%{IPV4:sourceIP}"]
      }
      geoip { # This is used for the map visualization
        source => "sourceIP"
        target => "geoip"
        database => "/etc/logstash/GeoLiteCity.dat"
        add_field => [ "[geoip][coordinates]", "%{[geoip][longitude]}" ]
        add_field => [ "[geoip][coordinates]", "%{[geoip][latitude]}"  ]
      }
      mutate {
        convert => [ "[geoip][coordinates]", "float"]
      }
    }

    if "has_error" in [tags] {
      grok {
        match => ["message", "%{SYSLOG5424PRI}%{NONNEGINT:syslog5424_ver}
+(?:%{TIMESTAMP_ISO8601:timestamp}|-) +(?:%{HOSTNAME:heroku_drain_id}|-)
+(?:%{WORD:heroku_source}|-) +(?:%{DATA:heroku_dyno}|-) +(?:%{WORD:syslog5424_msgid}|-)
+(?:%{SYSLOG5424SD:syslog5424_sd}|-|) (?<error_code>\Error\s[A-Z]\d+)
+%{GREEDYDATA:error_message}"]
      }
    } else if "multiline" in [tags] {

    } else {
      grok {
        match => ["message", "%{SYSLOG5424PRI}%{NONNEGINT:syslog5424_ver}
+(?:%{TIMESTAMP_ISO8601:timestamp}|-) +(?:%{HOSTNAME:heroku_drain_id}|-)
+(?:%{WORD:heroku_source}|-) +(?:%{DATA:heroku_dyno}|-) +(?:%{WORD:syslog5424_msgid}|-)
+(?:%{SYSLOG5424SD:syslog5424_sd}|-|) +%{GREEDYDATA:heroku_message}"]
      }
      mutate { rename => ["heroku_message", "message"] }
    }
# kv parses key value pairs from the msg so they can be index as their own field.
    kv {
      source => "message"
      trim => "{}\">,"
      trimkey => "{}\">,"
      value_split => "=>"
      exclude_keys => [ "scale","state","timeout","wait" ]
    }
# most of these are key value pairs that need to be converted for a visualization
    mutate {
      rename => { "service" => "latency" }
    }
    mutate {
      gsub => [ "latency", "ms", "" ]
    }
    mutate {
      convert => { "latency" => "integer" }
    }
    mutate {
      rename => { "connect" => "queue_time_ms" }
    }
    mutate {
      gsub => [ "queue_time_ms", "ms", "" ]
    }
    mutate {
      convert => { "queue_time_ms" => "integer" }
    }
    mutate {
      rename => { "sample#memory-postgres" => "memory-postgres-kb" }
    }
    mutate {
      rename => { "sample#memory-free" => "memory-free-kB" }
    }
    mutate {
      rename => { "sample#memory_total" => "memory_total_MB" }
    }
    mutate {
      gsub => [ "memory-postgres-kb", "kB", "" ]
    }
    mutate {
      gsub => [ "memory-free-kB", "kB", "" ]
    }
    mutate {
      gsub => [ "memory_total_MB", "MB", "" ]
    }
    mutate {
      convert => { "memory-postgres-kb" => "float" }
    }
    mutate {
      convert => { "memory-free-kB" => "float" }
    }
    mutate {
      convert => { "memory_total_MB" => "float" }
    }
    mutate {
      rename => { "sample#db_size" => "db_size_MB" }
    }
    mutate {
      gsub => [ "db_size_MB", "bytes", "" ]
    }
    mutate {
      convert => { "db_size_MB" => "float" }
    }
    ruby { # the size is reported in bytes. I want to display it in MB.
      code => "if event['db_size_MB'] != nil
        event['db_size_MB'] = (event['db_size_MB'] / 1048576).round(2)
        end"
    }
  }
}

output {
  if "_grokparsefailure" in [tags] {
    file { # if there is an error parsing the message is tracked in this file.
      path => "/var/log/logstash/failed_events-%{+YYYY-MM-dd}"
    }
  } else {
    elasticsearch { # send directly to ES
      hosts => ["bip-es-2095745217.us-east-1.elb.amazonaws.com:9200"]
    }
  }
}
```

DIY Hive Scale Projects and Kickstarters

https://hackaday.io/project/4648-analogio-a-full-stack-iot-platform/log/24740-creating-a-custom-weight-scale

https://www.kickstarter.com/projects/lthiery/apitronics-wireless-platform

http://www.hackerbee.com/

http://www.crowswood.com/bees/

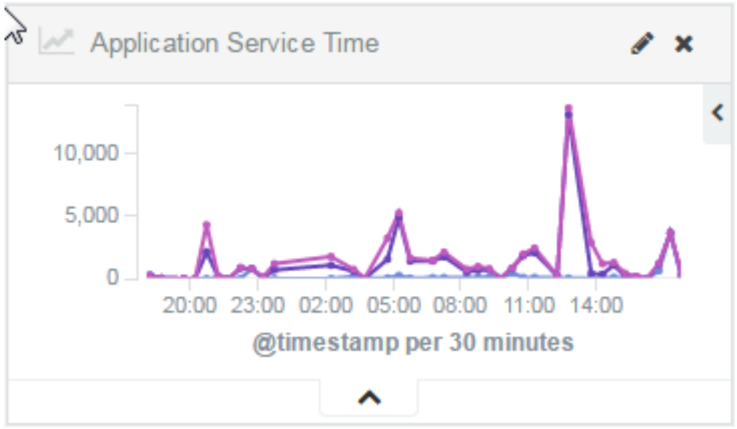
http://bijen.dewar.nl/

http://makezine.com/projects/bees-sensors-monitor-hive-health/

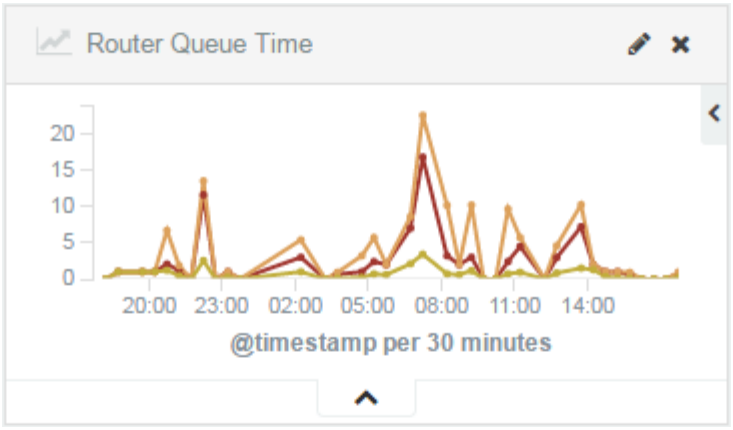
http://opensourcebeehives.net/

http://hivetool.org/

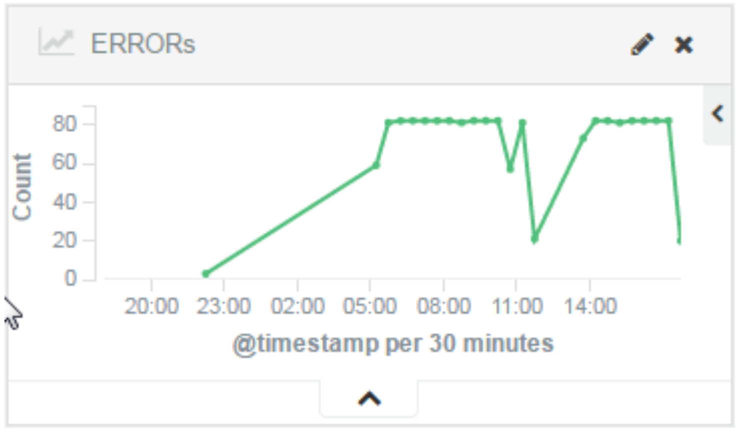
BIP Dashboard Visualizations



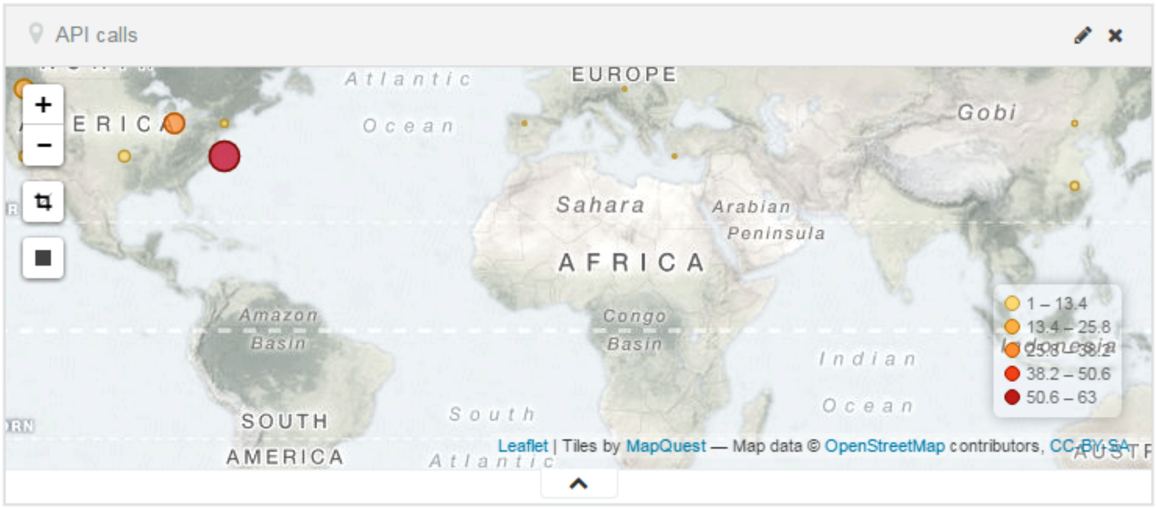
Displays the overall processing time of an API call in milliseconds.



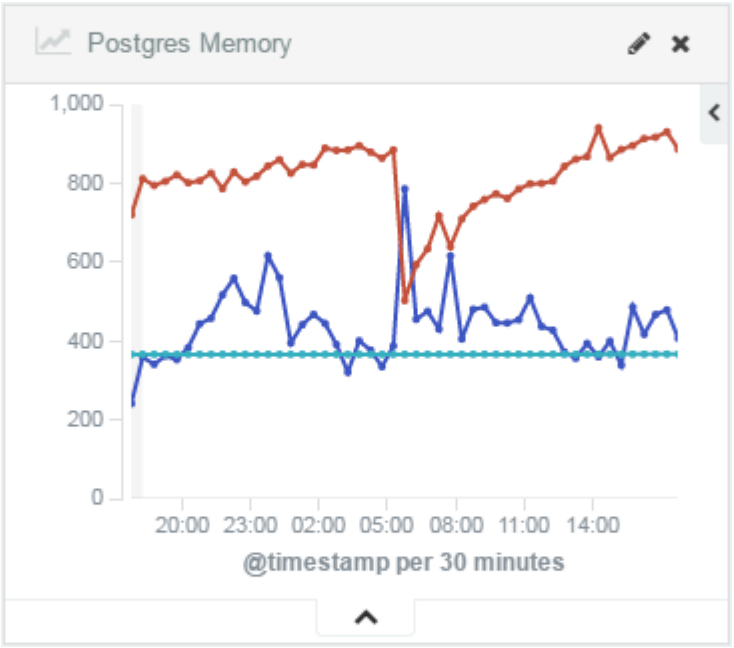
The router queue time is a measurement of how long a request spend inside the Heroku router before it reached our application.



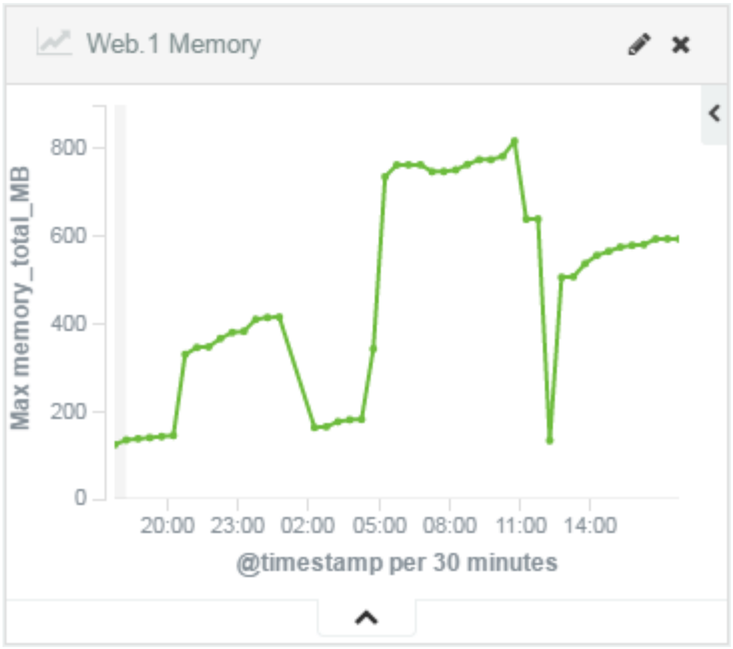
This graph plots errors that occurred over time. The graph is highlighting the errors that are produced by our memory leak.



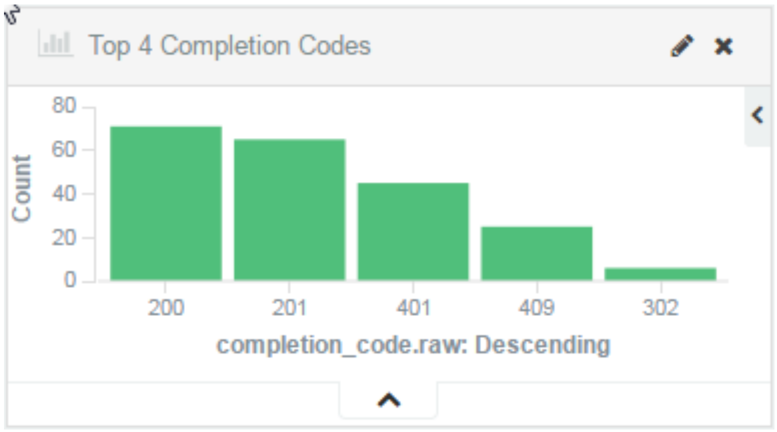
This map visualization is useful for determining the origins of traffic. We have requests primarily from the US, but also from China, Spain, Czech Republic, and Turkey?



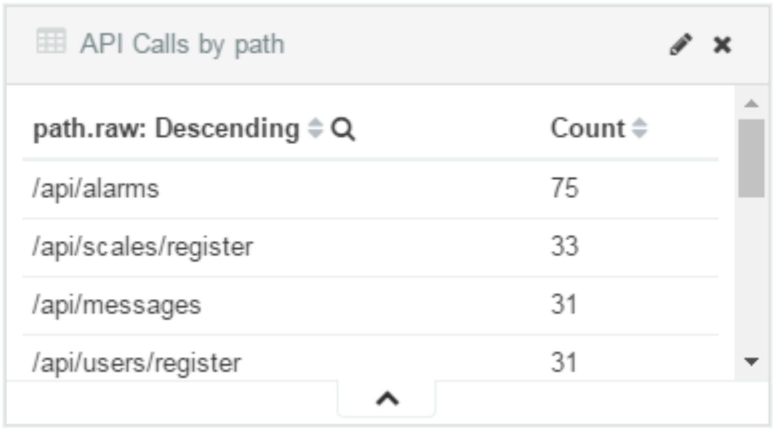
This graph provides details about the health of our database. The straight line indicates the size of the DB. The red line represents the memory used with the blue representing the amount of free memory available to the DB.



This graph shows our Dyno memory usage. The Dyno memory should NOT exceed 512MB. As you can see, we have an issue with memory consumption. The drops are either the result of Heroku restarting the Dyno or the bandaid we applied to the problem which is the Puma Worker Killer. We installed and configured this gem to restart the puma web server periodically until we solve the memory leak.



This chart provides some information on the success codes that we produce in response to external calls made to our site.



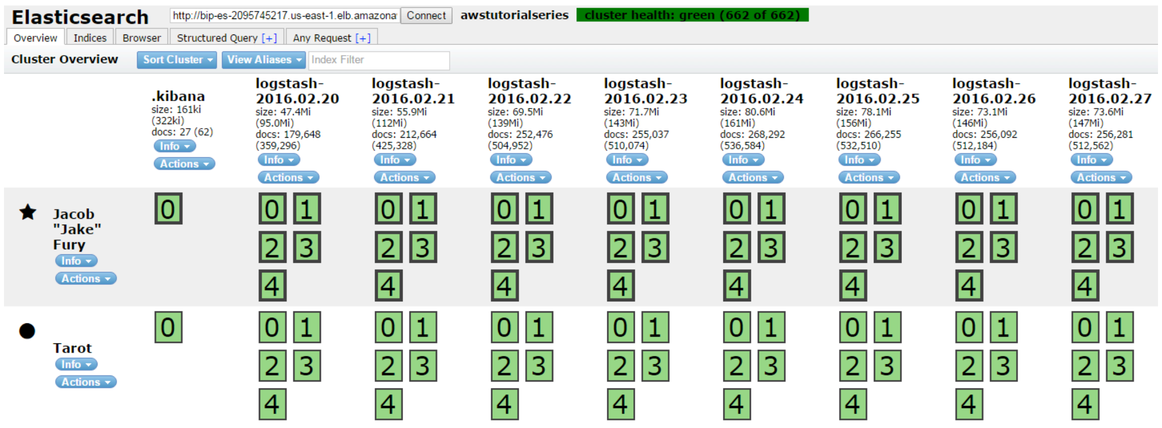
API Calls by path

path.raw: Descending	Count
/api/alarms	75
/api/scales/register	33
/api/messages	31
/api/users/register	31

This chart provides a view of the api calls that are being made to the site. Using the information in the completion code chart along with this one draws attention to some deficiencies with our clients use of our API.

Plugins Used

Head for Elasticsearch:



Elasticsearch http://bip-es-2095745217.us-east-1.elb.amazona Connect awstutorialseries cluster health: green (662 of 662)

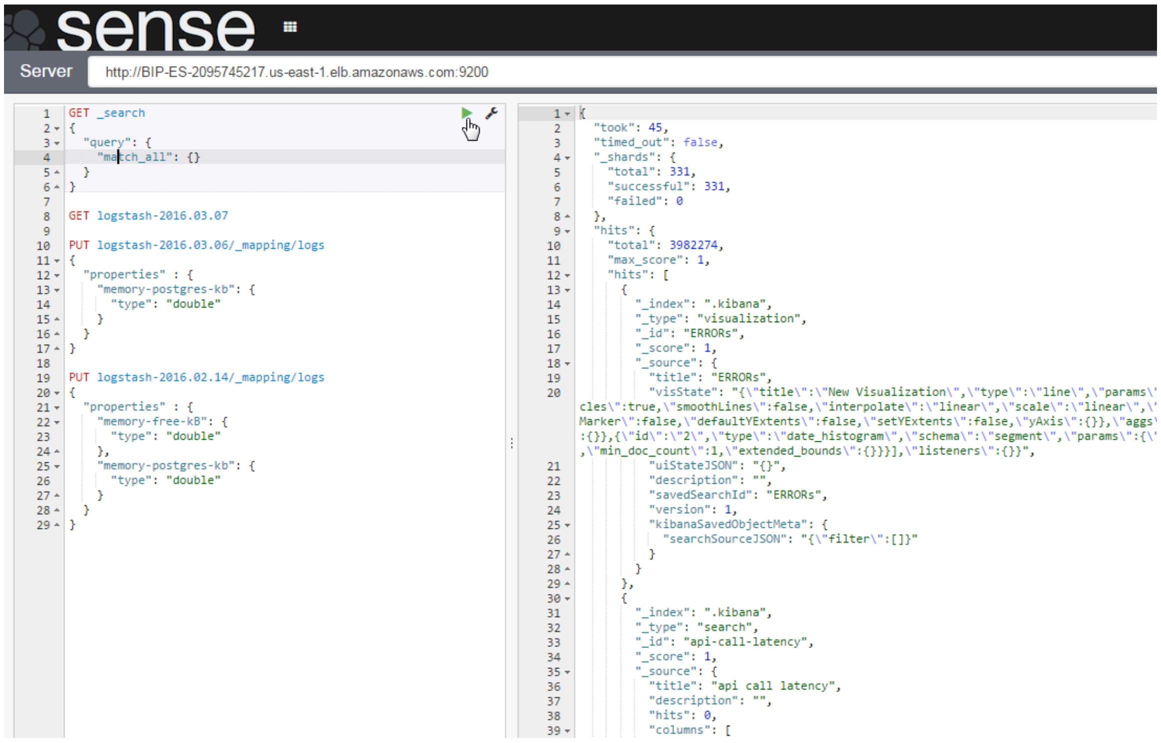
Overview | Indices | Browser | Structured Query [+] | Any Request [+]

Cluster Overview | Sort Cluster | View Aliases | Index Filter

	.kibana	logstash-2016.02.20	logstash-2016.02.21	logstash-2016.02.22	logstash-2016.02.23	logstash-2016.02.24	logstash-2016.02.25	logstash-2016.02.26	logstash-2016.02.27
	size: 161ki (322ki) docs: 27 (62)	size: 47.4Mi (95.0Mi) docs: 179,648 (359,296)	size: 55.9Mi (112Mi) docs: 212,664 (425,328)	size: 69.5Mi (139Mi) docs: 252,476 (504,952)	size: 71.7Mi (143Mi) docs: 255,037 (510,074)	size: 80.6Mi (161Mi) docs: 268,292 (536,584)	size: 78.1Mi (156Mi) docs: 266,255 (532,510)	size: 73.1Mi (146Mi) docs: 256,092 (512,184)	size: 73.6Mi (147Mi) docs: 256,281 (512,562)
★ Jacob "Jake" Fury	0	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4
● Tarot	0	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4

The Head plugin for Elasticsearch provides a diagnostic view of the clusters. I used this page to repair some nodes that were corrupted after I intentionally killed one of the Elasticsearch hosts. It was pretty easy to bring it back up. This tool also enables you to delete nodes. I deleted a few nodes that were created early on. The data it contained was pretty messy as I was still figuring out how to use logstash.

Sense for Kibana:



```
http://BIP-ES-2095745217.us-east-1.elb.amazonaws.com:9200
```

```
GET _search
{
  "query": {
    "match_all": {}
  }
}

GET logstash-2016.03.07

PUT logstash-2016.03.06/_mapping/logs
{
  "properties" : {
    "memory-postgres-kb": {
      "type": "double"
    }
  }
}

PUT logstash-2016.02.14/_mapping/logs
{
  "properties" : {
    "memory-free-kB": {
      "type": "double"
    },
    "memory-postgres-kb": {
      "type": "double"
    }
  }
}
```

```
{
  "took": 45,
  "timed_out": false,
  "_shards": {
    "total": 331,
    "successful": 331,
    "failed": 0
  },
  "hits": {
    "total": 3982274,
    "max_score": 1,
    "hits": [
      {
        "_index": ".kibana",
        "_type": "visualization",
        "_id": "ERRORs",
        "_score": 1,
        "_source": {
          "title": "ERRORs",
          "visState": "{\"title\":\"New Visualization\",\"type\":\"line\",\"params\'
cles\":true,\"smoothLines\":false,\"interpolate\":\"linear\",\"scale\":\"linear\",\'
Marker\":false,\"defaultYExtents\":false,\"setYExtents\":false,\"yAxis\":{}},\"aggs\'
:{}},{\"id\":\"2\",\"type\":\"date_histogram\",\"schema\":\"segment\",\"params\":{\'
,\"min_doc_count\":1,\"extended_bounds\":{}}}],\"listeners\":{}}",
          "uiStateJSON": "{}",
          "description": "",
          "savedSearchId": "ERRORs",
          "version": 1,
          "kibanaSavedObjectMeta": {
            "searchSourceJSON": "{\"filter\":[]}"
          }
        }
      },
      {
        "_index": ".kibana",
        "_type": "search",
        "_id": "api-call-latency",
        "_score": 1,
        "_source": {
          "title": "api call latency",
          "description": "",
          "hits": 0,
          "columns": [
```

The Sense plugin provides an administrative interface to interract directly with Elasticsearch. With Kibana alone, you're limited to elastic search queries. With Sense, you can manipulate the data as well. This was useful when I needed to convert some data types that were already persisted as strings to be double so they could be charted.

Kibana Watchdog script

When working with open source tools you often need to develop a thick skin and use creativity when issues crop up. For Kibana, the application kept crashing every 10+ minutes. I wasn't able to find any clues in log files nor was I able to locate any reports of anyone else having similar problems. I eventually grew tired of this fruitless endeavor and implemented a watchdog script that is trigged by cron. This was my first ever shell script.

```
#!/bin/sh
#kibana_watchdog.sh
#Ensure kibana continues to run

process=kibana

. /etc/init.d/functions

INSTALLED_DIR=/opt/kibana
EXEC_SCRIPT="$INSTALLED_DIR/bin/kibana"
LOG_DIR=/var/log/kibana
PID_DIR=/var/run
PID_FILE="$PID_DIR"/kibana.pid
LOG_FILE="$LOG_DIR"/kibana.log

status $process

RUN_STATUS=$?

echo $RUN_STATUS

if [ $RUN_STATUS == 0 ]
then
```

```
  echo already running
  exit
elif [ $RUN_STATUS == 1 ]
then
  rm /var/run/kibana.pid
  service kibana start
else
  service kibana start
fi

exit
```