Crear un módulo de Kibana para la monitorización del sistema FTS

Por - Alejandro Alonso Marín

Resumen

En el documento se encuentra una explicación básica del entorno en el que se trabaja. Seguido de aquello que se hizo para la realización de este. Finalmente encontramos breves descripciones del uso de algunas herramientas, problemas y valoraciones del proyecto.

Índice

Introducción a la plataforma ELK

La plataforma ELK (Elasticsearch + Logstash + Kibana) es una plataforma modular donde cada uno de los módulos lleva a cabo un trabajo con la finalidad de procesar datos.

Logstash

El primer módulo en interactuar con los datos es el de Logstash, este módulo es con el que se trabaja principalmente en el proyecto. Su objetivo es obtener los datos de una fuente, filtrarlos, procesarlos como se desee, y una vez procesados enviarlos a una plataforma indexadora, en este caso Elasticsearch. *(Hay alternativas para cada uno de estos módulos)

Estas 3 partes de Logstash que podemos observar se conocen como Input-Filter-Output. Estas se controlan mediante archivos de configuración llamados pipes o archivos de pipeline.
Habitualmente se utilizan plugins para realizar cada una de las 3 acciones listadas anteriormente. Por ejemplo, dependiendo del tipo de datos al que queramos acceder, utilizaremos un tipo de input plugin u otro. En este caso los datos estaban en una base de datos, por lo tanto utilizamos JDBC que es un plugin que permite hacer querys SQL para obtener los datos. En caso de que quisiéramos otro tipo de datos podríamos usar plugins como el de twitter o directamente desde archivos con el plugin file.
Los filter plugins son aquellos que se encargan de filtrar, y cada uno tiene sus propias funciones y parámetros, suelen ser los plugins más complejos.
Por último tenemos los plugins de output que simplemente indican donde queremos que se envíen estos datos indexados. Estos se envían en formato JSON allá donde se indiquen y se pueden cambiar cosas del formato de índice con estos plugins, como por ejemplo el nombre de dichos índices.

Elasticsearch

Elasticsearch es el núcleo de los 3 módulos, es aquel que se encarga de almacenar los índices enviados por logstash y los deja a disposición de kibana u otros programas que puedan acceder a índices en formato JSON. También podemos acceder a estos y modificarlos mediante la propia API de Elasticsearch.
Podríamos hacer una analogía y decir que Logstash es un bibliotecario que coje los datos de un montón de libros desordenados y los coloca bien clasificados y ordenados en Elasticsearch, que representaría una estantería donde es más fácil y más rápido acceder a ellos.

Kibana

Finalmente nos encontramos con el módulo de kibana, este módulo accede a los índices de Elasticsearch y lee su formato JSON mostrandolo en un formato más agradable a ojos de las personas y lo añade a su motor de búsqueda y herramientas gráficas.

Siguiendo la analogía anterior, kibana es un asistente de la biblioteca que nos ayuda a buscar el libro que queremos con sencillas órdenes que en kibana se traducen a filtros con notaciones simples como IS, IS NOT, EXISTS… La notación completa sería Campo-Orden-Valor.
A parte de filtros y gráficos, kibana también tiene una potente herramienta de modelación de datos mediante scripts en lenguaje Painless (también soporta otros lenguajes pero Painless sería el más usado). Este nos permite crear campos nuevos resultado de la combinación de otros campos mediante operaciones matemáticas.

Proyecto de creación del módulo

Preparación del entorno y herramientas

El proyecto estaba dividido en 2 partes, la primera consistió en realizar el proyecto en un entorno controlado donde las modificaciones erróneas no supusieran un problema, y una vez comprobado el funcionamiento en este entorno se pasaría a la segunda parte. La segunda parte consistió en llevar lo implementado al entorno de producción.

Para crear el entorno controlado se proporcionó una máquina virtual con un centOS que se creó mediante puppet, de forma que cada hora se sobreescribiran los archivos importantes de configuración para limpiar errores. Esta máquina se denominó kibana-test y se accedía a ella remotamente.

En dicha máquina se instalaron los programas esenciales para realizar las pruebas del proyecto, que en este caso serían los explicados en el apartado anterior ELK.

Los datos de FTS a procesar el proyecto estaban en un dump de base de datos sql. De forma que para poder revertir el dump en la máquina se procedió a la instalación de un MariaDB que sería similar a un mysql.

Una vez volcados de vuelta los datos a la BD se procedió a explorar la base de datos para buscar los campos importantes y empezar a maquetar la query que necesitábamos para extraer los datos de la BD y enviarlos a nuestro elasticsearch.

Para extraer los datos de la BD usamos un plugin de logstash llamado JDBC, tal plugin accede a la pase de datos mediante unos parámetros y la query que sustrae la información que necesitamos.

Configuración del archivo pipe

JDBC input plugin

Para empezar a configurar nuestro JDBC debemos proporcionar 3 campos esenciales para realizar la conexión:

- **jdbc_driver_library:** para proporcionar este campo debemos instalar primero el plugin de java, de forma que usaremos el comando *$yum install mysql-connector-java.noarch* y buscaremos el path para el archivo *mysql-connector-java.jar*
    - *En producción, la instalación se realizó mediante puppet*
- **jdbc_driver_class:** este campo siempre será *"com.mysql.jdbc.Driver"* en caso de usar una base de datos mysql.
- **jdbc_connection_string:** se completa con la siguiente string: "*jdbc:tipo de base de datos://host:puerto (de la máquina con la BD)/nombre de la base de datos"* En este caso la base de datos estaba en la misma máquina de forma que usaremos localhost.

A continuación proporcionamos el user y la password de nuestra BD y ya tendríamos los parámetros para realizar una conexión. Se proporciona también un ID para poder realizar más de un jdbc input como veremos más adelante.

```
jdbc {
    #Database connetion
    jdbc_driver_library => "/usr/share/java/mysql-connector-java.jar"
    jdbc_driver_class => "com.mysql.jdbc.Driver"
    jdbc_connection_string => "jdbc:mysql://localhost:3306/fts"
    jdbc_user => "fts"
    jdbc_password => "fts"
    id => "main"
    ...
```

Cuando queremos controlar que nuestro jdbc no coja datos innecesarios que ya se han indexado usaremos la herramienta que jdbc proporciona, llamada sql_last_value, esta variable lo que hace es registrar el id de cada campo de la query, pero se puede ajustar mediante parámetros para que registre uno de los campos de la query.
Para ello usamos los siguientes parámetros:

- **record_last_run:** activandolo pedimos que se registre sql_last_value.
- **tracking_column:** indicamos que campo de nuestra query queremos registrar.
- **tracking_column_type:** indicamos si el campo que le estamos asignando es un id o un timestamp.
- **use_column_value:** es necesario activarlo para que use los 2 parámetros anteriores.
- **last_run_metadata_path:** fichero donde se guarda sql_last_value.
- **clean_run:** al activarlo nos aseguramos que al reiniciar logstash y en el primer run del pipe no tendremos ningún problema con sql_last_value que haya podido existir anteriormente y nos asegura que se cargaran todos los datos de la BD.

```
    ...
    #Fields to control which data is taken
    #Activate clean_run will load all DB data on reboot
    clean_run => "true"
    record_last_run => "true"
    tracking_column => "job_finished"
    tracking_column_type => "timestamp"
    use_column_value => "true"
    last_run_metadata_path => "/etc/logstash/jdbc_metadata/.main_jdbc_last_run"
```

Para extraer datos mediante esta conexión debemos poner la query en el parámetro statement y determinar un schedule. Este schedule tiene un formato particular llamado rufus-scheduler. En este caso se pidió realizar la consulta cada 5 minutos.
ATENCION: Es necesario usar el sql_last_value en la query para aplicar su funcionamiento como deseemos.

```
    ...
    #Rufus Schedule
    #Every five minutes
    schedule => "/5 * * * *"
    statement => "select f.job_id, f.filesize, f.tx_duration, f.throughput,
f.user_filesize, f.transferred, j.source_se, j.dest_se, j.user_dn, j.vo_name,
j.job_finished, j.job_state, f.file_state from t_file f, t_job j where f.job_id = j.job_id
and j.job_finished > :sql_last_value order by j.job_finished"
    }
```

Debido a que se pidió realizar también una query para extraer datos de tablas de backup simplemente añadimos otro input jdbc en el mismo fichero, pero cambiamos la query y el id de este input. También hay que cambiar el path de nuestro sql_last_value para que uno no borre el otro y estén en ficheros separados.

Output plugins (Elasticsearsh)

Para poder enviar los datos que hemos extraído con JDBC a elasticsearch simplemente proporcionamos al parámetro *hosts* con la ip de la máquina con elasticsearch y el puerto 9200. Es recomendable usar el parámetro *document_id* para que si volvemos a generar los mismos índices con nuevos datos en elasticsearch estos sobreescriben los antiguos con el mismo ID. Para esto ponemos en el parámetro uno de los campos de nuestra consulta que sea determinante.
Es recomendable a veces usar el output stdout{} para ver por terminal los datos y compararlos con los indexados en elasticsearch para comprobar que todo funciona correctamente.
Para que los índices generados nos aparezcan con un nombre relacionado con los archivos que contienen, usamos el parámetro *index* que nos permite poner nombre a los índices generados. Debemos añadir una parte variable que distinga los diferentes índices de un mismo tipo, en este caso la fecha.
El output configurado es el siguiente:

```
output {
    #stdout {}
    elasticsearch {
        document_id => "%{job_id}"
        hosts => [ "localhost:9200" ]
        index => "fts-%{+YYYY.MM.dd}"
    }
}
```

Iniciar la transferencia de datos

Para iniciar logstash y que haga uso de nuestro pipe para transmitir los datos, debemos hacer uso del archivo binario que podemos encontrar en */usr/share/logstash*
Una vez en esa carpeta podemos iniciar logstash con el pipe que deseamos con el comando *$bin/logstash -f path-to-file/pipeline-file --config.reload.automatic*
Una vez iniciado, si nos da la señal "logstash started succesfully" debemos esperar a que llegue la hora indicada en el scheduler y iniciara la query. Cuando aparezca la query realizada por pantalla, esto indicará que los datos se han extraído y enviado a elasticsearch.

Para mirar los datos mediante kibana primero debemos crear un nuevo índice. Para ello accederemos a la ventana de management y seleccionaremos kibana index patterns → new index pattern. En ese momento seleccionamos palabras clave que nos ayuden a identificar los archivos indexados, en este caso fts-* serviria.
Una vez indexados ya podemos visualizar en la ventana de discover estos datos seleccionando el índice creado.

Filter plugins

Muchos de los datos extraídos directamente de la BD están en un formato que no deseamos, visualmente feos o en un formato con el que no podemos operar. Por ejemplo datos que deberían ser enteros son strings y si es el caso no se puede operar con ellos en kibana.
Para todo esto debemos usar los filter plugins de logstash:

Grok filter plugin

Mediante este plugin parseamos strings y cambiamos su formato para que sea de nuestro agrado.
Para parsear strings con grok utilizamos nomenclatura regexp Oniguruma. No obstante la mayoría de elementos que deseamos parsear se encuentran en archivos personalizados así como los que ya proporciona logstash. Estos archivos tienen el siguiente formato *NOMBRE DEL CAMPO patrón regexp* . Es altamente recomendable el uso de estos patrones predefinidos. En el caso de este proyecto fue utilizada esta lista de patrones.
Para añadir la lista de patrones a nuestro archivo de pipe, debemos guardarla en el directorio de pipes, este se encuentra en la carpeta */etc/logstash/patterns* . Este directorio debe ser indicado mediante el parámetro *patterns_dir => [path-to-dir] .*

Para parsear los datos de un campo y extraer aquellos que deseamos utilizamos el parámetro match, donde indicamos "campo de la BD" => "Regexp"
Para extraer un campo de la regexp debemos colocar *:nombre del campo* al lado de aquel dato que queremos extraer. Por ejemplo si queremos extraer la dirección de email invocamos el patrón personalizado de email junto con el nombre del nuevo campo *%{DATA}%{EMAILADDRESS:email}%{DATA}* . De esta forma extraemos un email de cualquier string.
Para maquetar la expresión regexp que queremos utilizar se recomienda utilizar el grok debugger.

Una vez hemos extraído de forma limpia un nuevo campo, si no vamos a utilizar el campo sin filtrar, podemos borrarlo usando el parámetro *remove_field => [“campo a borrar”]* .
Por último se recomienda añadir diferentes tags de fallo para saber que filtro grok está fallando en caso de fallo. Esto se hace mediante el parámetro *tag_on_failure => [“tag”]*

En el caso de este proyecto se realizaron los siguientes filtros para estos campos:

```
grok{
  patterns_dir => ["./patterns"]
  match => { "user_dn" => "%{DATA}CN=%{DATA:user} %{EMAILADDRESS:email}%{DATA}" }
  remove_field => [ "user_dn" ]
  tag_on_failure => ["_userdnparsefailure"]
 }
 grok{
  patterns_dir => ["./patterns"]
  match => { "dest_se" => "%{DATA}//%{USER:destination}" }
  remove_field => [ "dest_se" ]
  tag_on_failure => ["_destinationparsefailure"]
 }
 grok{
  match => { "source_se" => "%{DATA}//%{USER:source}" }
  remove_field => [ "source_se" ]
  tag_on_failure => ["_sourceparsefailure"]
 }
```

Date filter plugin

Uno de los problemas al indexar datos que son extraídos de una BD y no de datos en tiempo real, es que el @timestamp generado por las consultas no coincide con el tiempo real en el que llegaron los datos. Por lo tanto para indexar correctamente los datos debemos cambiar el @timestamp por el campo de nuestra BD que indique el tiempo en que llegaron los datos.
Para realizar esto podemos utilizar el Date filter, no obstante, hace falta primero parsear y filtrar el campo con la fecha mediante el filtro grok explicado anteriormente. En este caso se utilizó el siguiente filtro grok:

```
 grok{
  patterns_dir => ["./patterns"]
  match => { "job_finished" => "%{TIMESTAMP_ISO8601:tstamp}"}
  remove_field => ["job_finished"]
  tag_on_failure => ["_tstampparsefailure"]
 }
```

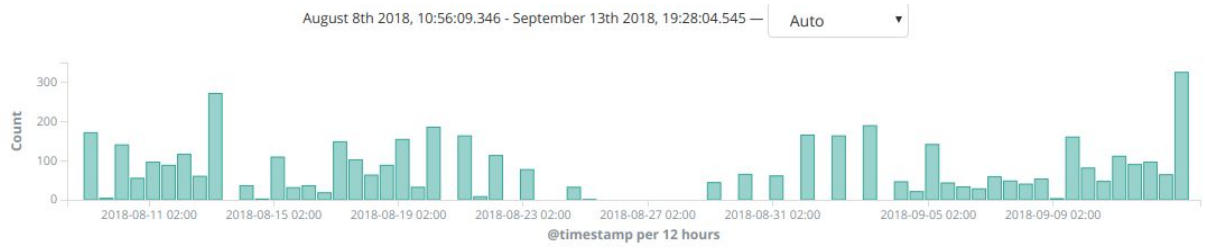
Una vez filtrado, utilizaremos el filtro date para cambiar el @timestamp por este nuevo campo. Para ello usamos el parámetro match para encontrar el campo con la fecha y parsearlo con el formato indicado, en este caso el campo era tstamp y el formato de datos era el estándar linux ISO8601. Para seleccionar donde aplicar esta nueva fecha usamos el parámetro target donde indicamos que sustituiremos el @timestamp por esta nueva fecha.
El código se ve de la siguiente forma:

```
date {
  match => [ "tstamp", "ISO8601" ]
  target => "@timestamp"
 }
```

Una vez modificado el @timestamp ya podemos eliminar el campo de tiempo que hemos filtrado con grok, esto se debe hacer mediante el filtro mutate después de todas las demás operaciones ya que el archivo pipe respeta el orden de las operaciones y eliminar un campo antes de usarlo lo hace inexistente.

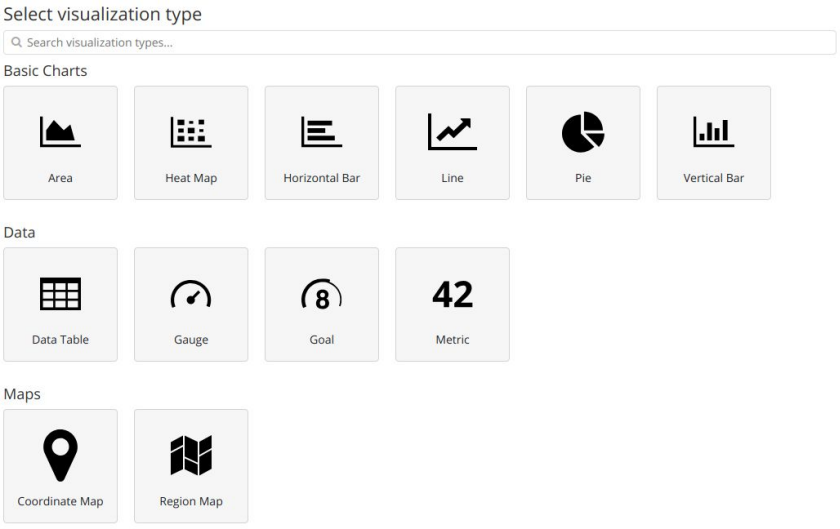
```
 mutate{
  remove_field => ["tstamp"]
 }
}
```

Una vez acabados todos estos filtros ya tenemos indexados correctamente los datos en nuestro elasticsearch y podemos observar que se reparten bien en el tiempo en kibana.

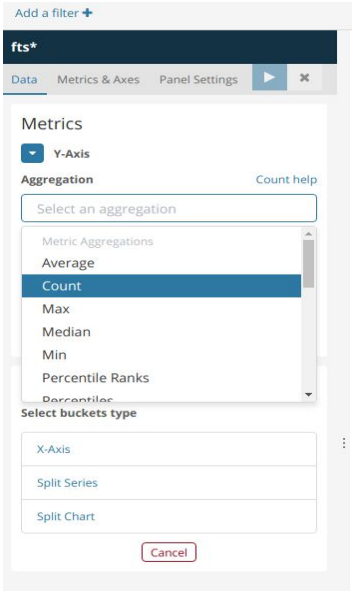


Confección de los gráficos con kibana

Para crear gráficos con kibana accederemos al apartado visualize.

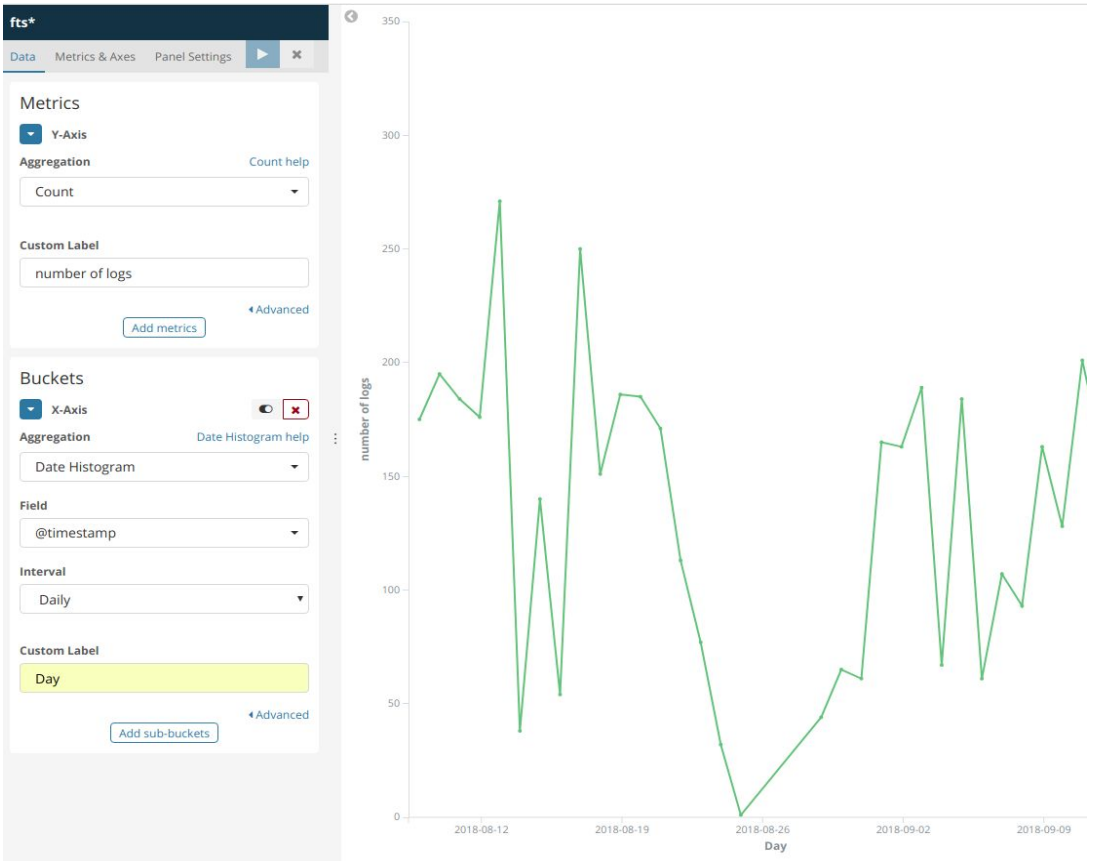


Encontramos distintos tipos de gráficos, pero en este caso se usa mayormente el gráfico de líneas.



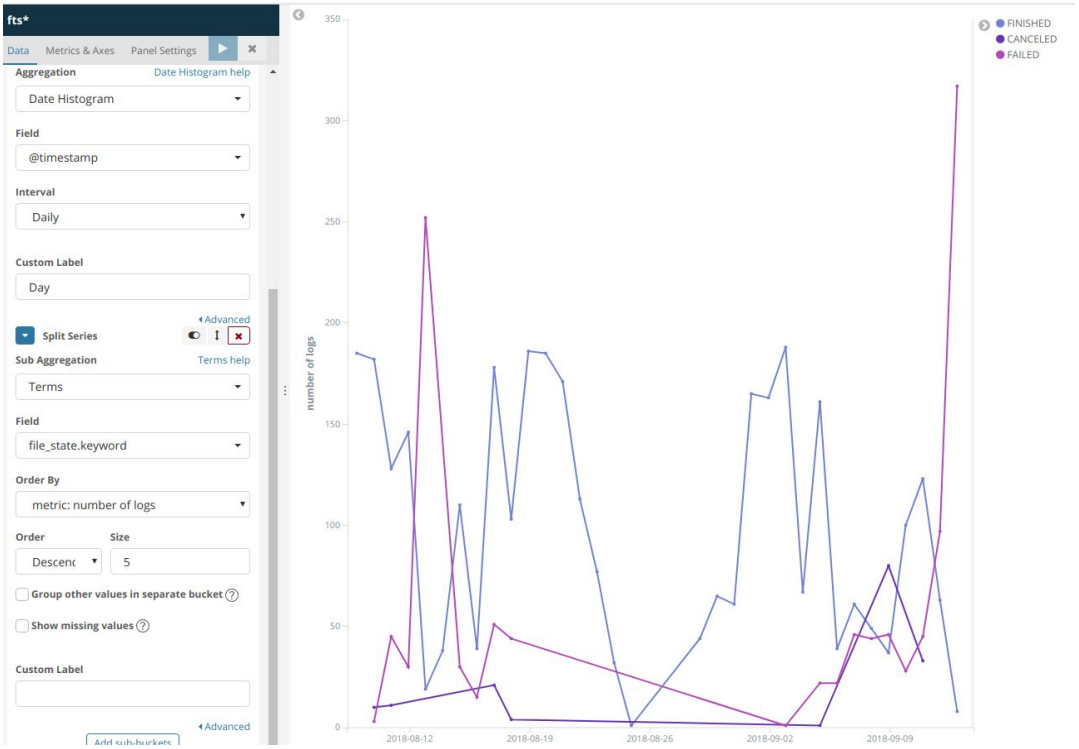
Una vez dentro de la creación de la visualización, lo primero es seleccionar el tipo de operación que precisamos para el dato que queremos. Por ejemplo si queremos saber el número de transferencias escogeremos, necesitaremos la operación “count” para contar el número. En cambio si queremos calcular una suma de datos como la cantidad de bytes transferida escogeremos la operación sum.

Una vez escogida la operación, como queremos que se nos muestren estos datos respecto al tiempo en que sucedieron, escogeremos en el X-axis la opción “date histogram” y cojeremos el @timestamp como referencia ya que es lo que hemos establecido como campo de tiempo en las transferencias.

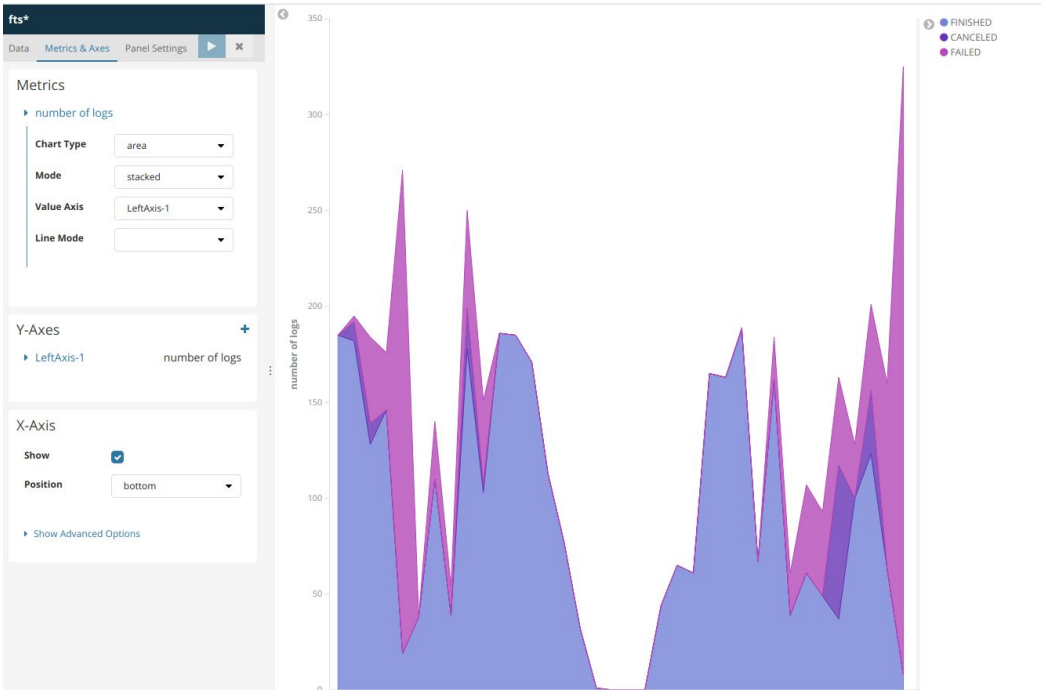


Si queremos filtrar los datos y comparar distintos tipos de log, como en este caso se quiso comparar la cantidad de logs de cada file_state. Debemos añadir un sub-bucket, donde

seleccionaremos la Sub Aggregation como Term y escogeremos aquello por lo que queramos separar los datos, en este caso file_state.



Para configurar y visualizar el gráfico de una forma más correcta, iremos a la ventana de metrics & axes y pondremos el gráfico en modo stacked para visualizar los files de cada tipo en comparación con los totales. También poner el gráfico en visualización de área, ayuda a ver mejor este hecho.



Por último, una vez creados los gráficos solo tenemos que ir al apartado de dashboard de kibana y añadir estos gráficos que hemos creado.

Preparación para entrar en entorno de producción

Para realizar un control de versiones y entorno antes de juntar el módulo con producción, se proporcionó un repositorio git. En este se creó una rama llamada fts_kibana donde se iban colocando todos los cambios para comprobar que funcionaban correctamente. Esta rama funcionaba sobre el entorno controlado de kibana-test. Este control de versiones se fué utilizando para realizar pruebas hasta que se consiguió llegar al pipeline presentado y explicado anteriormente en este documento.

Antes de juntar los archivos creados en la rama de test, se hizo un merge de la rama de producción sobre la de test, para comprobar en el entorno controlado que los nuevos archivos no interfirieran con los que había en la rama de producción.

Añadiendo a producción el módulo del proyecto

El entorno de producción también está controlado y administrado mediante puppet, de forma que para añadir el módulo del proyecto se tuvieron que modificar archivos yaml de puppet.
Antes de configurar nada en puppet debemos hacer un git merge para juntar la rama de test con la de producción. Una vez tenemos los archivos en la rama de producción debemos añadirlos a la configuración de logstash mediante puppet.

Para entender cómo se configura puppet es necesario entender la distribución de directorios:
Tenemos el directorio *pic-elk_config* donde guardamos todos los archivos de configuración de la plataforma ELK, como por ejemplo los pipes como el que hemos configurado en este proyecto, entre otras cosas.
Luego tenemos el directorio *r10k_config* donde encontramos toda la configuración de puppet que organiza los nodos, los archivos ELK y el entorno.

Para poder manejar los pipes con puppet, debemos moverlos al directorio de *templates/config…* para que sean tratados como archivos de configuración y no sean rechazados.
Una vez en el directorio adecuado procedemos a modificar los archivos de puppet. El principal archivo a modificar es el archivo de configuración yaml de los nodos que se encuentra en /r10k_config/hieradata/services/elastic/default.yml
En este archivo añadiremos una línea la clase wrapper::logstash::configfiles, donde indicaremos el path a nuestro nuevo archivo de configuración que queremos usar.

```
wrapper::logstash::configfiles:
   storage-fts:
        template: "elk_config/logstash/configs/fts-backup-pipeline.conf"
```

El archivo pipeline puede ser tratado como configfile. De esta forma no necesitamos crear un nuevo parámetro para esta clase. (Las clases de puppet han sido creadas en el

momento en que se creó el entorno de producción, su funcionamiento depende de las necesidades del entorno).

```
class wrapper::logstash (
   Hash $configfiles = {},
   Hash $patternfiles = {}
) {
    create_resources('logstash::configfile', $configfiles)
    create_resources('logstash::patternfile', $patternfiles)
}
```

Una vez todo está configurado en los archivos de puppet, lo hacemos correr para reiniciar logstash y esperamos a que empiece a indexar datos a nuestro elasticsearch. Luego seguiremos los pasos explicados en el entorno de prueba para crear un índice y modelar los gráficos que queramos.

Anexo

Uso básico de la herramienta git con puppet

A continuación se describen las órdenes usadas para el manejo de git en este proyecto:

- **git branch:** nos permite saber en qué rama estamos realizando cambios, de esta forma evitamos modificar la rama de producción.
- **git chekout "nombre de rama":** Nos permite cambiar de rama.
- **git diff:** Muestra la diferencia entre los cambios actuales y los de la última versión que se ha subido al repositorio.
- **git commit -a -m "mensage":** guarda todos los cambios realizados en el repositorio local.
- **git push origin "rama":** sube los cambios del repositorio local al compartido.
- **git pull origin "rama":** baja todos los cambios hechos en el repositorio compartido al repositorio local.
- **git merge "rama":** mezcla la rama designada sobra la actual.

Órdenes para manejar el control de versiones sobre el entorno controlado:

- **puppet agent --test --server "máquina con configuración puppet" --environment "rama":** Sobreescribe todos los archivos del entorno tal y como se indica en la configuración de puppet.
- **crontab -e:** permite modificar el archivo donde se indica las rutinas de borrado de archivos de puppet.

Funcionamiento físico de Elasticsearch

Se ha explicado cómo funciona elasticsearch de forma conceptual, pero en la realidad, elasticsearch se encuentra en un entorno distribuido.
Tanto logstash como elasticsearch se encuentran instalados en diversos nodos que se comunican entre sí. En estos nodos no tiene porque encontrarse ni los datos que procesamos ni el programa que accede a los índices (kibana), esto permite formar un

cluster que dedique toda su potencia y recursos, para formar una plataforma de datos sólida, con alta disponibilidad y muy escalable.
Para conseguir este rendimiento, cuando creamos un índice mediante logstash y lo enviamos a elasticsearch, elasticsearch lo partirá en un número de piezas, definidas en su configuración, que llamamos shards. Estas shards se distribuyen de forma equitativa entre los nodos de nuestro cluster de forma que balancean la carga automáticamente. Además de almacenar los shards de forma distribuida se hacen réplicas de estos shards y se reparten en los nodos, de forma que si un nodo no estuviera disponible, podríamos usar las réplicas de sus shards localizadas en otros nodos, para recrear el índice que queremos obtener.

Principales problemas y dificultades

La principal dificultad de este proyecto reside en que las herramientas que se utilizan tienen su propia sintaxis y no utilizan en concreto la de ningún lenguaje en específico. Por eso la mayor parte del tiempo invertido en este proyecto se utiliza en leer documentación sobre los plugins a utilizar en cuestión, también es necesario saber el funcionamiento del entorno y de los datos que se manejan, ya que esto ayuda a comprender, porque a veces el uso de un plugin estrictamente descrito en el manual de este, no funciona aplicado a nuestro entorno.

A parte de los problemas que hubo para llegar al estado final del pipe, uno de los principales problemas era la eficiencia de las consultas mysql, ya que las consultas a las tablas de backup eran muy pesadas y ralentizaba mucho el pipeline, hasta que se consiguió separar la query para que se hiciera por separado de la query regular. De todas formas cargar las tablas de backup solo se debe realizar una vez gracias al sql_last_value.

En cuanto a elasticsearch el propio cluster de producción tiene problemas debido al poco número de nodos que no es adecuado para el sistema de shards de elasticsearch, ya que el mínimo de nodos adecuado para estos shards es de 5 y nuestro cluster tiene 3.