Universidad Rey Juan Carlos

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA EN SISTEMAS AUDIOVISUALES Y MULTIMÉDIA

TRABAJO FIN DE GRADO

SISTEMA DE RECOGIDA MASIVA Y VISUALIZACIÓN DE DATOS DE APIs PÚBLICAS

Autor: Manuel Peláez Guzmán

Tutor: Jesús González Barahona

Curso académico 2015 / 2016

RESUMEN

EL proyecto consiste en un sistema de recogida, almacenamiento y visualización de eventos generados por la API de GitHub, con un posterior análisis de los mismos basado en la información mostrada por distintos elementos visuales y ***dashboards***. Esta API genera constantemente información en tiempo real y resulta especialmente interesante representarla para poder entenderla y extraer factores que de otra manera seria imposible obtener .

La plataforma está compuesta por tecnologías desarrolladas recientemente y en constante y actual crecimiento y será capaz de manejar amplias cantidades de información, medida en muchos casos en eventos/segundo. Todo este trabajo está enfocado muy a favor de las tendencias actuales de mercado, así como las necesidades presentes en muchas de las empresas en el mismo.

Para la recogida se usará un programa en Python, que se encargará de coger los datos (formato JSON) directamente del flujo de eventos y cargarlos en ***Elasticsearch***.

Elasticsearch será el motor de búsqueda y almacenamiento que contendrá todos los datos a visualizar y analizar. Los datos le llegarán de dos maneras distintas. La primera, será mediante ***Logstash***, herramienta que permite estructurar los datos mediante distintos filtros y codecs antes de llevarlos al motor de almacenamiento. La otra será mediante el protocolo HTTP.

Para la visualización, se usará Kibana. Este hará las veces de *front-end* para *Elasticsearch* y permitirá la generación de distintos diagramas de visionado, que a continuación se plasmarán en los *dashboards*.

El objetivo, por tanto, no es solo recoger, guardar y mostrar la información, si no hacerlo de manera clara y que permita un análisis completo de los distintos eventos, respondiendo así a cualquier pregunta que pueda surgir sobre ellos y extrayendo conclusiones claras sobre su comportamiento.

ÍNDICE

CAPÍTULO 1. INTRODUCCIÓN

Big Data es un término que se escucha mucho hoy en día, y cada vez más, en todo tipo de ámbitos, sobre todo empresarial. Aunque se hablará más en detalle de Big Data en el siguiente apartado, es conveniente hablar sobre la enorme necesidad de las empresas actuales de sistemas capaces de manejar cantidades masivas de información. Es por eso que surgen sistemas con tecnologías capaces de atender esa necesidad. Sistemas que, en mucho caso, sustituyen a los antiguos debido a que se sus prestaciones quedan obsoletas. En la mayoría de los casos, simplemente, no dan abasto con la cantidad de información que se necesita procesar, haciendo los procesos lentos y tediosos.

Este trabajo surge como posible solución a necesidades actuales en el mercado y está íntimamente relacionado con Big Data y esos sistemas y tecnologías nuevas emergentes. En esta primera parte se va a explicar el objetivo a conseguir con el proyecto, así como problemas genéricos a solventar. De esta manera se podrá entender de una manera más completa tanto el funcionamiento del sistema, como su desarrollo.

1.1 OBJETIVOS

1.1.1 DESCRIPCIÓN DEL PROBLEMA

Hoy en día, en cualquiera empresa de mediano o gran tamaño, se manejan cantidades de información cada vez mayores. Información que, por otra parte, necesita ser procesada, almacenada y visualizada de una manera concreta que se adapte a cada tipo de dato.

Teniendo en cuenta lo relatado en el punto anterior, la intención de este trabajo es la de brindar una **herramienta**, mediante la unión de unas tecnologías concretas ya existentes y software propio, que permita realizar lo anteriormente comentado de una manera rápida, clara, y que por supuesto permita utilizar los datos y la información que nos brindan de una manera práctica.

Además, hace falta encontrar un fuente de datos lo suficientemente grande y fiable que permita comprobar que, en efecto, el sistema es capaz de tratar información en grandes cantidades de una manera rápida y eficaz.

1.1.2 OBJETIVO PRINCIPAL

Observando el problema presente, la meta que se propone es la de proporcionar un sistema capaz de tratar grandes cantidades de información en tiempos relativamente pequeños. Las tecnologías que lo conformen deberán ser modernas y accesibles. Por todo esto, se proponen tecnologías concretas para atender el problema que se presenta: ***Logstash***, **Elasticsearch** y ***Kibana***. Estas punteras herramientas se diseñaron para trabajar en conjunto (formando la pila ELK) y el objetivo de este proyecto no es otro que el de crear un sistema usando las tres, comprobando así que trabajan en armonía y con la eficiencia necesaria para tratar cantidades masivas de información. Las tres juntas forman las tres fases necesarias de la plataforma: recolección y procesamiento, almacenamiento y visualización.

Por otra parte, surge la necesidad de una fuente de información pública, real, fiable y que presente cantidades de información suficientemente grandes. Por todo esto, además del hecho de que el grupo del que se ha recibido ayuda en el presenta trabajo contaba con amplia experiencia con *GitHub* , se propone la ***API de GitHub***. *GitHub* es una forja (plataforma de desarrollo colaborativo) mundialmente conocida y usada por una enorme cantidad de empresas participantes de una amplia gama de proyectos. Cuenta con una API que ofrece en tiempo real gran variedad de eventos, bien definidos y estructurados, así como bien documentados. Estos eventos proporcionan datos muy variados, de los cuales se puede extraer información muy diversa como, por ejemplo, el número de empleados trabajando en cada proyecto, así como la carga de trabajo por día de la semana o la tendencia y frecuencia de los cambios en los repositorios.

1.1.3 REQUISITOS

- Componer una plataforma Big Data con las herramientas propuestas: *Logstash, Elasticsearch, Kibana.*
- El sistema se nutrirá, en su desarrollo definitivo, de información recogida de la API de *GitHub*.
- La herramienta debe ser capaz de recoger todos los eventos/datos en tiempo real, sin dejar ni repetir ninguno.
- Le acceso a los datos, así como a la visualización de los mismos debe tardar un tiempo relativamente rápido. Siempre teniendo en cuenta la cantidad y tipo de datos.

- Los *dashboards* deben ser claros, concisos y rápidamente entendibles, sin que puedan llevar a una incorrecta interpretación de los datos. Además, debe permitir analizar la situación y el contexto a los que se refieran los datos, pudiendo extraer de ellos respuestas a problemas tanto concretos como genéricos.
- El análisis debe ser verídico, claro y conciso. Debe reflejar el contexto y situación de los datos. Así mismo, debe ser extraído y entendido únicamente con la información de los eventos representados en los *dashboards* de Kibana.

CAPÍTULO 2.CONTEXTO DEL PROYECTO

Antes de hablar del propio trabajo en sí, cabe comentar y poner en situación todo sobre lo que trata. Como por ejemplo, el sitio web del que vamos a extraer la información a analizar, así como tecnologías y herramientas usadas con el mismo fin que la aquí mostrada así como para Big Data en general.

Por otra parte, debo reseñar que todo el trabajo has sido desarrollando software libre, es decir, sin ningún tipo de cargo o coste. Esto es lo que lo hace tan útil y accesible para cualquier entorno: no hacen falta ni licencias ni gastos para su uso.

2.1 Big Data

Big Data se describe como el almacenamiento, tratamiento y análisis de cantidades de datos tan desproporcionadamente grandes que resultan casi imposibles de tratar con las herramientas convencionales, ya sea por temas de capacidades o tiempo. Nace como consecuencia de la proliferación de internet y las páginas web, los archivos y aplicaciones de audio y video, las redes sociales, los móviles, etc. Los elementos de esta lista son capaces de generar más de 2 peta-bytes al día.

Hablamos de un entorno absolutamente relevante para muchos aspectos, desde el análisis de fenómenos naturales como el clima o de datos sismográficos, hasta entornos como salud, seguridad o, por supuesto, el ámbito empresarial.

Y es precisamente en ese ámbito donde las empresas desarrollan su actividad donde está surgiendo un interés que convierten a *Big Data* en las palabras que sin duda escucharemos viniendo de todas partes: vendedores de tecnología, de herramientas, consultores…

Cierto es que en este proyecto no se trabajará en un volumen de datos de tal magnitud, pero si se hablara de millones de datos u eventos, una cantidad más que razonable y cercana al Big Data. Además, el periodo de tiempo de recogida de los datos, del que hablaremos más adelante, no pasa de unos pocos días. Si fueran meses, hablaríamos de cantidades de datos prácticamente insostenibles y procesables por los sistemas convencionales.

2.2 Tecnologías Big Data

Una vez hablado sobre el Big Data y el fuerte impacto que este tiene en el mercado actual, se presentan a continuación algunas de las tecnologías presentes en la industria y de las que muchas empresas hacen uso:

- ***Hadoop***: marco de software de código abierto escrito en Java y basado en *MapReduce** de Google y en trabajo de sistemas de archivos distribuidos. Está construido para soportar aplicaciones distribuidas analizando grandes volúmenes de datos utilizando *clusters* de servidores, transformándolos para que sean más usables por estas aplicaciones.
- ***Disco Project***: tecnología desarrollada en Python . Nació en el centro de investigación de Nokia para hacer frente a los desafíos de manejar cantidades de información masiva. Desde entonces ha sido desarrollado por multitud de compañías para varios fines como análisis estadístico, modelado probabilístico y *data mining* entro otros muchos.
- ***SAP HANA***: ambiente de escala empresarial concentrado en la inteligencia de negocios y usos relacionados. Permite el análisis de datos para realizar búsquedas en cantidades masivas de información en tiempo real.
- ***ManyEyes***: servicio de visualización de datos que da al usuario la capacidad de subir su propia información a la web para después elegir el tipo de visualización que se quiere generar. Entre las posibilidades está el análisis de texto, comparaciones, seguimiento de tendencias, mapeos y relaciones entro nodos de información. Fue desarrollado por el grupo de investigación de IBM.

Hadoop y ManyEyes forman parte de los cuatro ejemplos presentado debido a que son dos de las tecnologías más conocidas y usadas en el mercado. Disco Project y SAP HANA no son tan conocidos, pero el primero está desarrollado en Python, lenguaje usado en este proyecto y SAP HANA forma parte de SAP, tecnología tremendamente demandada hoy en día. Las cuatro son solo una pequeña parte de la totalidad de posibilidades existentes en el mercado actual.

2.3 GitHub

GitHub es el sitio web del que obtendremos y analizaremos los datos principalmente. Consiste en una plataforma de **desarrollo colaborativo de software** para alojar proyectos utilizando el sistema de control de versiones Git.

Permite alojar tu repositorio y brinda una serie de útiles herramientas para su gestión y control así como para facilitar y favorecer el trabajo en equipo dentro de uno o varios proyectos. Proyectos además que no necesariamente tienen que ser tuyos, puesto que también puedes colaborar y mejorar proyectos ajenos mediante funcionalidades para hacer *fork* (clonar un repositorio ajeno en tu propia cuenta) y solicitar *pulls* (añadir tu contribución al proyecto original).

2.4 OTROS PROYECTOS RELACIONADOS

2.4.1 GitHub Archive

Se trata de un proyecto que permite visualizar el *timeline* público de GitHub, almacenarlo y hacerlo fácil de acceder para análisis más profundos. Los eventos se registran de manera horaria, y se puede acceder a ellos con cualquier cliente *http*. Usa código Ruby para descargar los datos del *timeline*, iterar sobre ellos y almacenarlos

Se puede acceder a *GitHub Archive* al completo como datos públicos en *Google BigQuery*. Los datos son actualizados automáticamente de manera horaria y te permiten realizar querys parecidas a SQL sobre ellos en segundos, sin necesidad de procesar o descargar nada por cuenta propia.

Cabe destacar el hecho de que te permite acceder a 1 TB de datos procesados por mes gratis. Para aprovecharlo mejor, es conveniente restringir las querys a intervalos horarios relevantes.

El proyecto completo *GitHub Archive*, así como toda la información necesaria se encuentra en: https://www.GitHubarchive.org/ .

2.4.2 GHtorrent

Se trata de algo muy parecido a lo comentado en el punto anterior. *Ghtorrent* permite monitorizar los eventos del *timeline* proporcionado por la API de GitHub. Para cada evento, rescata todos sus contenidos y dependencias de manera exhaustiva.

Después, almacena los *JSON* crudos en una base de datos *MongoDB*, al mismo tiempo que extrae su estructura en una base de datos *MySQL*.

Cada dos meses el proyecto pone a disposición de los usuarios los datos recogidos durante ese periodo de tiempo, también compartido con el protocolo *bitorrent*.

Hasta abril de 2015, *MongoDB* almacena alrededor de 6.5 TB de datos *JSON*, mientras que *MySQL* almacena más de 600 millones de filas de metadatos extraídos. Buena parte de la actividad de 2012, 2013, 2014 y 2015 ha sido recuperada también. Además, se sigue recuperando información hacia atrás para rescatar la historia completa de importantes proyectos.

El proyecto, junto con toda la documentación relativa se encuentra en: http://ghtorrent.org/ .

CAPITULO 3: TECNOLOGÍAS USADAS

Como se ha mencionado en el punto 1, la finalidad de este trabajo es construir una plataforma *Big Data* con herramientas tanto actuales como accesibles y fiables. El flujo de funcionamiento de la herramienta será el siguiente:

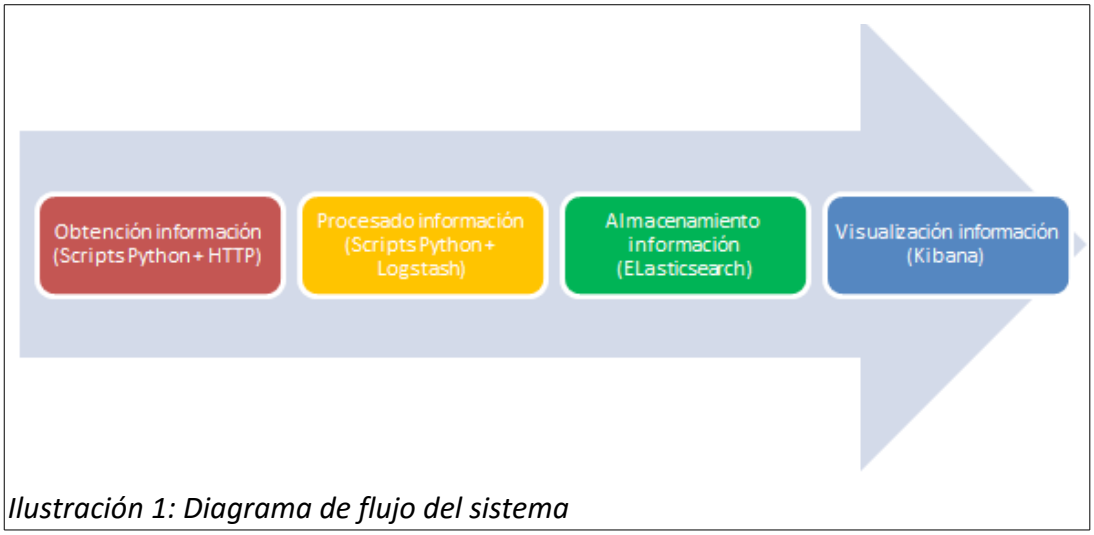


Ilustración 1: Diagrama de flujo del sistema

Las tecnologías concretadas son Logstash, Elasticsearch y Kibana. Todas ellas serán descritas en detalle a continuación, junto con la API de GitHub (fuente de datos) y las demás tecnologías involucradas en este trabajo.

3.1 API DE GitHub

La API de GitHub nos proporciona el flujo con todos los eventos que ocurren en GitHub (*commits*) desde la apertura de una incidencia hasta los eventos *pull* y *push*. Cada evento viene con una serie de atributos que nos proporcionan toda la información necesaria para comprender bien el que y cuando ha pasado. Por ejemplo, todos los eventos vienen con campos como *Created at*, que nos dice la hora exacta en la que ocurrió, y dependiendo del tipo de evento, campos como quién realizó el commit, en que repositorio, que proyecto, y un largo etcétera.

Como he mencionado antes, hay diversos tipos de eventos en el flujo (timeline) de la API de GitHub. Cada uno tiene diferentes campos, basándose en la información necesaria para

analizar cada tipo. Si bien algunos son muy comunes, algunos son muy poco frecuentes. He aquí una lista con los 25 tipos distintos:

1. CommitCommentEvent
2. CreateEvent
3. DeleteEvent
4. DeploymentEvent
5. DeploymentStatusEvent
6. DownloadEvent
7. FollowEvent
8. ForkEvent
9. ForkApplyEvent
10. GistEvent
11. GollumEvent
12. IssueCommentEvent
13. IssuesEvent
14. MemberEvent
15. MembershipEvent
16. PageBuildEvent
17. PublicEvent
18. PullRequestEvent
19. PullRequestReviewCommentEvent
20. PushEvent
21. ReleaseEvent
22. RepositoryEvent
23. StatusEvent
24. TeamAddEvent
25. WatchEvent

Como se observa, el número de tipos de eventos es amplio. Obviamente, no todos ocurren con la misma frecuencia, y además, no todos son visibles o accesibles en el timeline de eventos, o lo fueron y ya no lo son:

Eventos no visibles→ *DeploymentEvent*, *DeploymentStatusEvent*, *MembershipEvent*, *PageBuiltEvent*, *RepositoryEvent*, *StatusEvent*, *TeamAddEvent*.

Eventos ya no existentes→ *DownloadEvent*, *FollowEvent*, *ForkApplyEvent*, *GistEvent*.

El que haya eventos no visibles en el timeline se debe a que son usados únicamente para activar ciertos mecanismos . En cuanto los eventos que ya no existen, esto es porque han caído en desuso o no se han considerado necesarios con el paso del tiempo.

Los eventos visibles e extraíbles del timeline, son los eventos en los que nos centramos (precisamente porque podemos acceder a su información) y, además, son los más importantes. Aquí se explican brevemente cada uno de ellos:

***CommitCommentEvent*:** Se dispara cuando un comentario de commit es creado.

***CreateEvent*:** representa un repositorio, rama o *tag* creados.

***DeleteEvent*:** representa una rama o *tag* borrados.

ForkEvent: se activa cuando un usuario hace un *fork* a un repositorio.

GollumEvent: Salta cuando se crea una página Wiki.

IssueCommentEvent: Se dispara cuando un *issue comment* es creado en un tema o en una petición *pull.*

IssuesEvent: Funciona cuando un tema es asignado, desasignado, etiquetado, des etiquetado, abierto, cerrado o reabierto.

MemberEvent: Se activa cuando un usuario es añadido como colaborador a un repositorio.

PublicEvent: Se lanza cuando un repositorio privado pasa a ser público. El mejor evento de GitHub, sin duda.

PullRequestEvent: Se activa cuando una pull *request* es asignada, desasignada, etiquetada, des etiquetada, abierto, cerrado, reabierto o sincronizado.

PullRequestReviewCommentEvent: Se dispara cuando un comentario es creado en una porción del diff unificado de una pull request.

PushEvent: Se activa cuando se sube código a una rama de un reporitorio. Además de estos eventos, también se lanzan los webhook PushEvents cuando se sube código a una rama.

ReleaseEvent: Sirve para indicar que una *release* se ha publicado.

WatchEvent: se lanza cuando un usuario "sigue" a un repositorio. El actor del evento es el usuario que sigue el repositorio, y el repositorio del evento es el repositorio al que ha seguido.

Todos estos tipos de eventos estarán presentes entre los datos obtenidos a partir de la *API* de GitHub. De cada uno de ellos y sus campos se extraerá la información que permitirá analizar el comportamiento de los distintos usuarios, proyectos, empresas, repositorios y demás elementos

involucrados en GitHub. Más adelante se explicara en detalle de qué manera nos ofrece el flujo de eventos la API y como se accede a ella, así como el formato de los datos.

3.2 Logstash

3.2.1 Concepto

Se trata de un motor de recopilación de datos *open source*, con capacidades de direccionamiento en tiempo real. *Logstash* puede, dinámicamente, unificar datos de diversas fuentes, normalizarlos y conducirlos a los destinos elegidos. Limpia y democratiza toda la información para diversas analíticas o representaciones.

Si bien Logstash introdujo innovaciones con respecto a los *log*, sus capacidades se extienden mucho más allá. Cualquier tipo de evento puede ser enriquecido y transformado con una amplia gama de *inputs*, *filter* y output *plugins*, con muchos *codecs* nativos para simplificar el proceso de ingesta. Logstash acelera los procesos de entrada "tragándose" un mayor volumen y variedad de datos.

3.2.2 Características

Flujo

El flujo de procesado de eventos de Logstash tiene tres fases: *input* (entrada), *filter* (filtro) y *output* (salida). Los *inputs* generan los eventos a procesar, los *filtros* los modifican y los *outputs* los direccionan a donde corresponda. Las entradas y salidas soportan códec que codifican o decodifican, según entren o salgan los datos, sin tener que usar filtros intermedios.

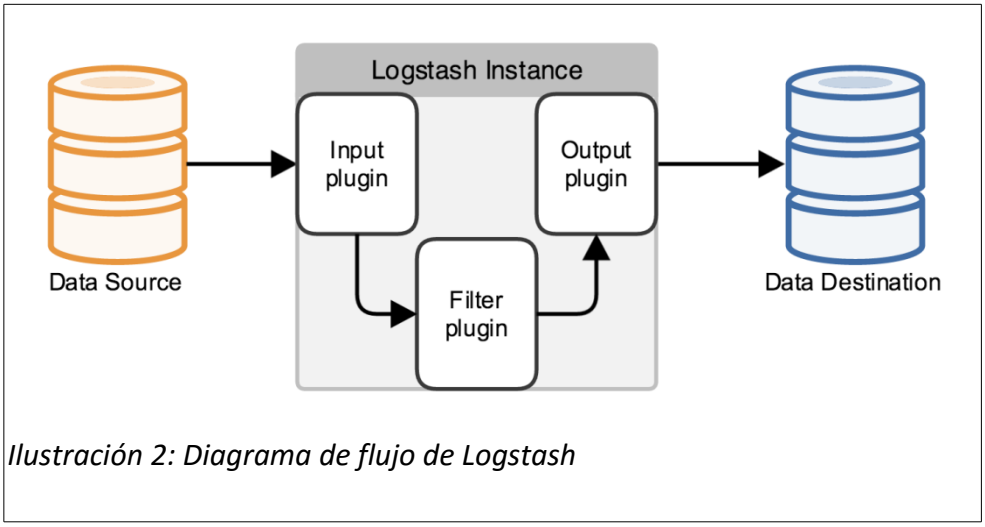


Ilustración 2: Diagrama de flujo de Logstash

Inputs (entradas)

Como se ha mencionado antes, los inputs o entradas de Logstash son las fuentes de donde sacamos la información. Algunas de las más conocidas/usadas son:

Fichero: Directamente de un fichero que tengamos en nuestro *filesystem*.

Syslog: Escucha en el conocido puerto 514 para mensajes de *syslog* y formatea según el formato RFC3164.

Redis: Lee de un servidor *redis*, usando tanto canales como listas redis. Redis se suele usar como intermediario en una instalación Logstash centralizada, que mantiene en espera los eventos Logstash hacia los *shippers* Logstash remotos.

lumberjack: procesa eventos mandados con el protocolo *lumberjack*. Ahora llamado *Logstash-forwarder*.

Las aquí mencionadas puede que sean las entradas más utilizadas, pero ni mucho menos las únicas. Hay infinidad de manera de "meterle" datos a Logstash, como por ejemplo Twitter o Gmail. Ambas medios cuyo análisis resulta muy útil y atractivo para determinadas empresas a fin de conocer tendencias y gustos de la gente.

***Outputs* (salidas)**

Los outputs son la fase final del flujo de Logstash. Un evento puede pasar a través de varias salidas, pero una vez todo el procesado de salida se completa, el evento termina su ejecución. Algunas de las salidas más usadas incluyen:

Elasticsearch: los datos se envían a EL para su almacenamiento e indexado. Es la salida que se usará en este proyecto.

Fichero: se escriben los datos de salida en un fichero almacenado localmente.

Graphite: se envían los datos a *graphite*, una popular herramienta de software libre para almacenar y visualizar métricas.

Statsd: los datos se mandan a *statsd*, un servicio que recibe estadísticas, como contadores y temporizadores, enviados a través de UDP y manda agregados a una uno o más servicios de *backend*.

Como en el caso de las entradas, se han comentado las entradas más comunes o utilizadas. Pero aquí la lista de posibles salidas es también infinita. Por ejemplo, Gmail también podría hacer las veces de output, así como cualquier BBDD o servicio.

Filtros

Los filtros son dispositivos intermedios de procesado en el flujo de Logstash. Se puede combinar filtros con condicionales para llevar a cabo acciones en un evento si posee o cumple ciertas cualidades, como por ejemplo modificar un campo si tiene un determinado valor, traducir eventos y/o campos o incluso activar otros filtros si un campo cumple con unas características. Algunos de los filtros más útiles son:

- ***Grok***: *parsea* y estructura texto. Es actualmente la mejor manera en Logstash de *parsear* texto desestructurado a algo con estructura y contra lo que podremos lanzar búsquedas. Tiene un total de 120 patrones incluidos en Logstash.
- ***Mutate***: Realiza transformaciones regulares en campos de los eventos. Dichas transformaciones son: renombrar, quitar, sustituir y modificar.
- ***Drop***: Elimina un evento por completo.
- ***Clone***: Hace una copia de un evento, pudiendo además añadir o quitar campos.
- ***Geoip***: añade información sobre la localización geográfica de direcciones IP.

Codecs

Los codecs son, básicamente, dispositivos de tratamiento que actúan como parte de una entrada o salida. Te permiten separar fácilmente el transporte de la información de los procesos y filtros que se lleva a cabo durante el flujo.

Algunos de los codecs más populares son:

- ***JSON***: codifica o decodifica datos en formato *JSON (Javascript Object Notation)*.
- ***Multiline***: une diversos eventos de texto multilinea como excepciones Java o mensajes de rastreo de ejecución en un único evento.
- **Rubydebug**: saca tus eventos de Logstash usando la librería de impresión de Ruby.

Tolerancia a fallos

Los eventos pasan de etapa a etapa usando colas internas implementadas con una *SizedQueue* de Ruby. Esta puede contener un número máximo de objetos. Cuando la cola está con la capacidad al máximo, se bloquea, impidiendo que se escriba ningún evento más en ella.

Logstash tiene un tamaño de cola, por defecto, de 20 objetos. Esto quiere decir que un máximo de 20 eventos podrán estar esperando a la siguiente etapa, lo que previene la perdida de datos y evita que Logstash actué como un sistema de almacenado, ya que estas colas no están hechas para almacenar eventos por largo tiempo.

Con pequeños tamaños de cola, simplemente se impide que lleguen más eventos cuando hay cargas muy altas o cuellos de botella. Las alternativas serian tener una cola ilimitada o descartar eventos. Una cola ilimitada crece hasta que excede la memoria, explotando y perdiendo todos los eventos que esperaban en la cola. Con respecto a la otra alternativa, perder eventos, en la mayoría de los casos, es indeseable e inaceptable.

Cuando una salida falla, la gran mayoría de sistemas siguen intentando mandar los eventos afectados por el error. En Logstash, cuando falla una salida, esta espera hasta que pueda mandar los mensajes con éxito y deja de leer de la cola, provocando que se llene.

Lo anterior causa que los filtros se bloqueen al no poder mandar más eventos a la *queue*. Esto conlleva que el filtro deje de leer de su cola, provocando, así, que esta también se llene.

Una vez más, al alcanzar el tamaño máximo de la cola del filtro, la entrada dejara de procesa datos de dondequiera que le estén llegando, causando así que su cola se complete.

La clave está en que todo el flujo se bloque hasta que el problema se resuelva y los eventos puedan fluir con normalidad otra vez. De esta manera, se evita que se pierda ningún evento.

Modelo multi hilo

El modelo multi hilo de Logstash funciona de la siguiente manera:

Hilos de entrada→hilos de filtrado → procesadores de salida

Cada entrada corre en un hilo. Esto evita que las entradas más lentas frenen a las más rápidas.

En los filtros, cada hilo recibe un evento y le aplica todos los filtros, en orden, antes de mandarlo a la cola de salida. Esto permite escalabilidad entre CPUs, ya que muchos filtros conllevan un gran uso computacional.

Los procesadores de salida, actualmente, son mono hilo. Las salidas reciben los eventos en el orden en el que están definidas en el fichero *config*. Estas pueden decidir introducir los eventos en un buffer antes de publicarlos, como por ejemplo, en el caso de mandarlos a

Elasticsearch. Este mecanismo mejora el rendimiento ya que evita que el flujo se atasque esperando una respuesta de Elasticsearch.

3.3 Elasticsearch (ES)

3.3.1 Concepto

Elasticsearch es un motor de búsqueda y análisis de datos en tiempo real, escalable y distribuido. Es una potente herramienta que permite indexar un gran volumen de datos y posteriormente hacer consultas sobre ellos. Soporta búsquedas aproximadas, facetas y resaltados, así como búsquedas de texto completo.

Este motor, ES, se basa en *Lucene* pero expone su funcionalidad a través de una interfaz *REST* recibiendo y enviando datos en formato *JSON* y oculta mediante esta interfaz los detalles internos de Lucene. Esta interfaz permite que pueda ser utilizada por cualquier plataforma, no solo Java. Puede usarse desde *Python*, *.NET*, *PHP* o incluso desde un navegador con *Javascript*. Es persistente, es decir, que lo que indexemos en ella sobrevivirá a un reinicio del servidor.

ES puede servir para hacer búsquedas a texto completo pero también podemos hacer otra serie de cosas adicionales que no podemos con una base de datos relacional aunque soporte en su lenguaje SQL búsqueda a texto completo. Por ejemplo, resaltado (*highlight*) y facetas (*facets*), también permite hacer búsquedas vagas (*fuzzy*) y soporta diferentes analizadores según el idioma de la propiedad en que se busque.

EL es una base de datos no SQL, es decir, no relacional. Por lo tanto, muchos conceptos son diferentes. Por ejemplo, en EL no se habla de tablas, filas y columnas, si no de índices, tipos y documentos.

3.3.2 Características

Editor *Sense*

En primer lugar se hablará de Sense, una útil consola que sirve para interactuar con la API *REST* de ES. Aunque se podría referir a Sense como una tecnología por sí mismo, se ha decidido englobar aquí por su relación con ES.

Sense está compuesto por dos paneles principales. En el panel de la izquierda, llamado editor, es donde se escriben las peticiones que se realizaran a ES. Se habla de peticiones por que todo funciona mediante HTTP junto a JSON. Las respuestas de ES a estas peticiones se muestran en el panel de la derecha. La dirección IP del nodo de ES al que se realizan las peticiones debe especificarse en la caja de texto encima de los paneles.

Ilustración 3: Editor Sense

Estructuración y Representación de los datos

Como se ha dicho más arriba, EL no es una base de datos relacional, y como tal, no comparten los mismos conceptos, si bien algunos podrían asimilarse entre ellos. En ES, la unidad básica de información que puede ser almacenada e indexada, es el documento, lo que en una BBDD relacional llevaría el nombre de fila o registro. Un documento es representado en formato JSON, muy utilizado en internet para el intercambio de datos.

```
{
    "email":      "john@smith.com",
    "first_name": "John",
    "last_name":  "Smith",
    "info": {
        "bio":         "Eco-warrior and defender of the weak",
        "age":         25,
        "interests": [ "dolphins", "whales" ]
    },
    "join_date": "2014/05/01"
}
```

Ilustración 4: ejemplo de documento en Elasticsearch

Los documentos se agrupan en tipos. Un tipo se correspondería con el concepto de tabla en un BBDD relacional. Estos, a su vez, se agrupan en índices. Un índice es la mayor unidad de agrupación en ES y se relacionaría con una base de datos en una BBDD relacional.

Aunque no es necesario especificar un esquema (en EL no se trata con esquemas) , si lo es especificar un *mapping* para aquellos tipos que requieran algo más que los más básicos campos y operaciones. En realidad, esto es muy similar a declarar un esquema.

Al “mapear” un tipo declaramos:

- Que campos tiene.
- Que campos se requieren
- Que campos se usan como PK (*Primary Key*)
- Como se indexa y busca cada campo.

```
PUT my_index (1)
{
  "mappings": {
    "user": { (2)
      "_all":       { "enabled": false  }, (3)
      "properties": { (4)
        "title":    { "type": "string"  }, (5)
        "name":     { "type": "string"  }, (6)
        "age":      { "type": "integer" } (7)
      }
    },
```

Ilustración 5: Asignar un mapeo a un indice.

[1] Se especifica el índice al que se le va a aplicar el mapping definido.

[2] Se define para qué tipo, dentro del índice establecido, se van a declarar los campos.

[4] Bajo la opción *properties* se define el nombre y le tipo de los campos, así como demás opciones.

[5][6][7] Se declaran el nombre de cada campo, junto al tipo de dato que se espera en él.

Un índice puede albergar documentos con distintos tipos de *mapping*. Y a cada uno de ellos se le puede asociar varias definiciones de mapping.

Mapeo dinámico

Cuando ES encuentra un campo que conocía antes en un documento que entra, usa el mapeo dinámico para determinar el tipo de dato de dicho campo y lo añade al mapeo del tipo correspondiente.

Este comportamiento puede ser deseado o no. Tal vez no se sepa con que campos podrían llegan los documentos más tarde y se necesita que se indexen automáticamente sin tener que estar modificando el mapping cada vez que sea el caso. Pero tal vez se quieren ignorar esos campos desconocidos, o lanzar una excepción alertando de dicho evento.

Por todo eso, este comportamiento se puede controlar mediante la opción *dynamic*. Esta opción se puede aplicar tanto al objeto raíz (si queremos que ocurra en todos los campos de ese objeto) o en un campo concreto.

Query DSL

ES provee un lenguaje de búsquedas rico y flexible llamado *Query DSL* (Domain-Specific Language), que permite elaborar complicadas y robustas *querys*.

El DSL se especifica usando un cuerpo JSON. Un ejemplo de query sería:

```
GET /megacorp/employee/_search
{
    "query" : {
        "match" : {
            "last_name" : "Smith"
        }
    }
}
```

Ilustración 6: ejemplo de query simple

En este ejemplo, se busca en el campo *last_name* de todos documentos del índice *megacorp*, tipo *employee,* todos aquellos que sean *Smith.*

El comportamiento de una query vendrá definido en su contexto, que puede ser query o filtro.

Contexto query

Una query usada en este contexto responde la pregunta "¿qué tan bien coincide este documento con esta búsqueda?". A parte de decidir si coincide con un documento o no, la query también calcula una puntuación representando lo bien que coincide con el documento, en relación con el resto de documentos.

Contexto Filtro

En este contexto, una clausula query responde a la pregunta "¿este documento coincide con esta búsqueda?". La respuesta es un simple Si o No. No se calculan puntuaciones. Este contexto de las búsquedas se usa para, sobre todo, para filtrar datos estructurados.

Este tipo de filtros pueden ser guardados en la cache de EL, para acelerar el rendimiento.

A continuación se muestra un ejemplo de una búsqueda en ambos contextos, query y filtro:

```
GET _search
{
  "query": { ❶
    "bool": { ❷
      "must": [
        { "match": { "title":   "Search"        }}, ❸
        { "match": { "content": "Elasticsearch" }}  ❹
      ],
      "filter": [ ❺
        { "term":  { "status": "published" }}, ❻
        { "range": { "publish_date": { "gte": "2015-01-01" }}} ❼
      ]
    }
  }
}
```

Ilustración 7: query en ambos contextos

[1] El parámetro “query” indica el contexto query.

[2][3] [4] Las clausulas “bool” y “match” se usan para puntuar que tan bien coinciden los documentos con la búsqueda.

[5] El parámetro “filter” indica que se usa el contexto filtro.

[6][7] Clausulas “term” y “range” extraen los documentos que no coinciden, pero no afectaran a la puntuación de los que sí lo hacen.

Una query debe usarse en contexto query cuando se trate de una búsqueda no exacta y sea importante lo cerca que este el documento de la búsqueda (basándonos en las puntuaciones de búsqueda que nos proporciona EL). Para el resto de búsqueda se deberá utilizar el contexto filtro.

Índice Invertido

ES usa una estructura llamada índice invertido, la cual está diseñada para permitir búsquedas texto-completo muy rápidas. Un índice invertido consiste en una lista de todas las palabras (una única vez) que aparecen en cada uno de los documentos, y por cada palabra, una lista de documentos en los que aparece. De esta manera, si queremos buscar una palabra (o varias), solo hará falta mirar en los documentos en los que aparece.

En el caso de que la búsqueda consista en varias palabras, se tendré en cuenta primero el documento que contenga el mayor número de ellas. Este documento tendrá una mejor puntuación (score) en el resultado de la búsqueda.

Sin embargo, existen varios problemas con el índice inverso. Las palabras en mayúsculas, por ejemplo, aparecen como otro término separado de la misma palabra en minúscula. Al buscar, el usuario podría referirse a ambas. Algo parecido ocurre con las palabras que no son raíz, como por ejemplo, los plurales.

Para solventar estos problemas, las palabras se normalizan y pasan a tokens. Este proceso se llama Análisis.

Análisis

El análisis es un proceso que consiste en lo siguiente:

- Primero, se pasa un bloque de texto a términos individuales, llamados tokens, que sirven para usarse en un índice invertido.
- Después, se normalizan esos términos a un formato estándar para mejorar la búsqueda de los mismos.

Este trabajo lo llevan a cabo los analizadores. Un analizador es simplemente un "empaquetador" que combina tres funciones en un mismo paquete.

Filtro de caracteres

En primer lugar se pasa el string por diversos filtros de caracteres. Su trabajo consiste en "ordenar" el string antes de pasarlo a tokens. Un ejemplo seria pasar el carácter "&" a la palabra "and".

Tokenizer

A continuación, el string se pasa a tokens individuales mediante in tokenizer. Un tokenizer simple puede separar el texto en términos individuales cada vez que encuentre un espacio o un signo de puntuación.

Filtros de tokens

Por último, cada termino para por diversos filtros de token en orden, que pueden cambiar términos (por ejemplo, quitar las mayúsculas), quitar términos o añadirlos. El resultado es el string inicial separado en tokens individuales y ligeramente modificados.

Esta funcionalidad se puede tanto activar como desactivar mediante la opción analyzed. Se puede aplicar al objeto raíz, para afectar a todos los campos, o a un campo concreto.

3.4 KIBANA (KB)

3.4.1 Concepto

Kibana es una plataforma de software libre de análisis y visualización diseñada para trabajar con ES. KB permite, de manera sencilla, buscar, ver e interactuar con los datos almacenados en índices de ES. De esta manera, se pueden realizar análisis avanzados de información mediante una gran variedad de charts, tablas, mapas, etc.

KB hace fácil entender grandes volúmenes de datos. Tiene un interfaz web intuitivo, con el cual podemos crear y compartir los charts anteriormente mencionados.
Estos charts son dinámicos, es decir, interactúan con ES en tiempo real, mostrando de manera inmediata cambios detectados en alguno de los índices mostrados.

3.4.2 Características

Data Discover

En la sección *Discover* de KB se pueden realizar *querys*, filtrar los resultados y examinar los documentos resultantes. Una vez seleccionado el índice sobre el que queremos actuar, podremos realizar búsquedas sobre él, apareciendo un histograma mostrando el la cantidad de documentos en cada fecha. Juntos a los documentos que cumplen la búsqueda aparece el número tal de documentos resultantes.

El filtro temporal limita la búsqueda a un cierto periodo de tiempo. Este se puede activar siempre y cuando el índice seleccionado contenga eventos basados en tiempo, y un campo temporal haya sido indicado al seleccionar el índice.
Por defecto, aparece para los últimos 15 minutos.

También es posible interactuar con el histograma que aparece, seleccionando así rápidamente el rango de datos que se necesite.

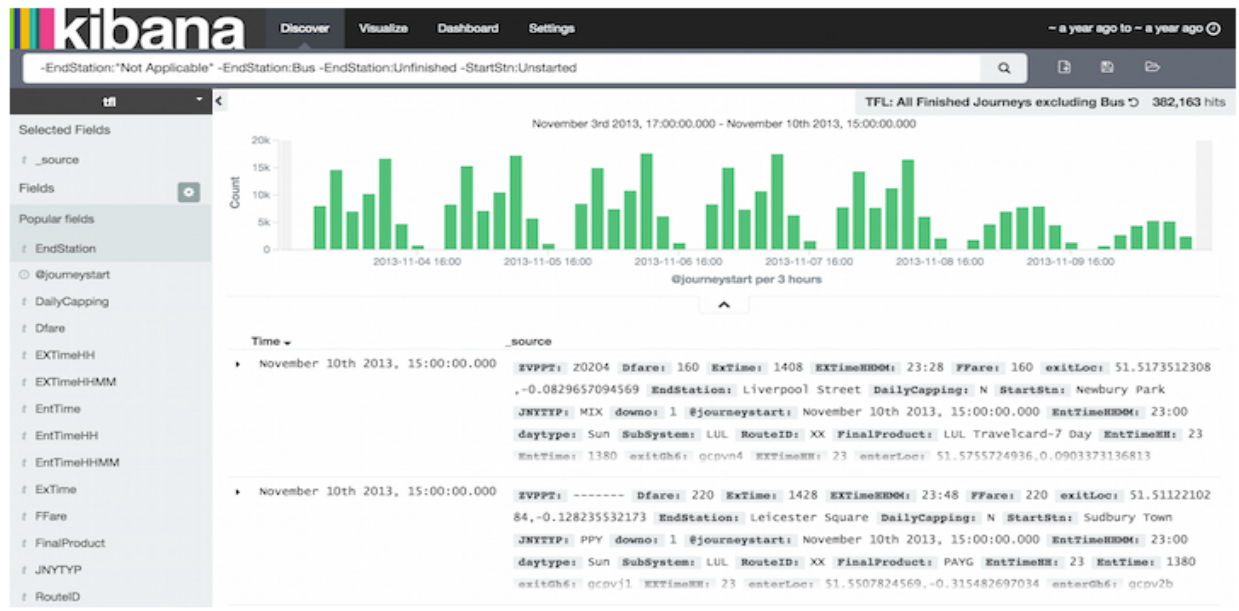


Ilustración 8: pantalla discover

A la hora lanzar búsquedas sobre el índice, se puede hacer tanto con simples strings, como usando la sintaxis de *Lucene* o *el Query DSL* de ES. Cuando la búsqueda es lanzada, todos los histogramas, tablas, y campos en la lista se actualizan para mostrar el resultado de la búsqueda. Los datos resultantes aparecerán ordenados de manera descendiente, cronológicamente hablando.

Mapeo dinámico en Kibana

ES mapea dinámicamente los documentos que le llegan por defecto, pero esta opción es modificable. Sin embargo, KB necesita que los documentos hayan sido mapeados dinámicamente en el momento de mostrarlos para poder llevar a cabo esto de manera correcta. Además, necesita de esta funcionalidad para poder administrar y usar correctamente el índice .kibana, el cual se crea automáticamente para almacenar las búsquedas guardadas, las visualizaciones y los dashboards.

Si se necesita operar ES con el mapeo dinámico desactivado, se deberá proveer manualmente unos mapeos para los campos que se deseen visualizar en KB.

Visualize

La pantalla de *Visualize* se usa para diseñar las representaciones de los datos. Estas representaciones se podrán guardar, así como usarlas individualmente o combinar varias de ellas para formar un dashboard.

Se puede crear una visualización tanto de búsquedas anteriormente almacenadas como de búsquedas nuevas específicas para una visualización determinada.

Kibana ofrece en esta pantalla varios tipos de Visualización de datos:

- ***Área chart***: Se usa para mostrar la contribución total de varias series de datos diferentes.
- ***Data Table***: Tabla de datos crudos.
- ***Line Chart***: Diagrama de líneas que permite comparar distintas series de datos.
- ***Markdown widget***: sirve para desplegar información o instrucciones sobre los dashboards.
- ***Metric***: cuenta o suma total de una serie de datos.
- ***Pie chart***: Se usa para mostrar lo que contribuye cada serie de datos a un total.
- ***Tile map***: usado para asociar el resultado de un agregación de datos con un punto geográfico.
- ***Vertical bar chart***: se puede usar para múltiples propósitos.

Una vez se ha elegido el tipo de representación, se debe seleccionar la fuente de los datos. Esta puede ser una búsqueda anteriormente hecha y guardada o una nueva. Cuando se selecciona búsqueda nueva, se deberá indicar también el índice o índices sobre el que se realizará la búsqueda.

Con todo esto, ya solo faltaría utilizar el completo editor para seleccionar lo que se pretende mostrar con el tipo de visualización seleccionado.

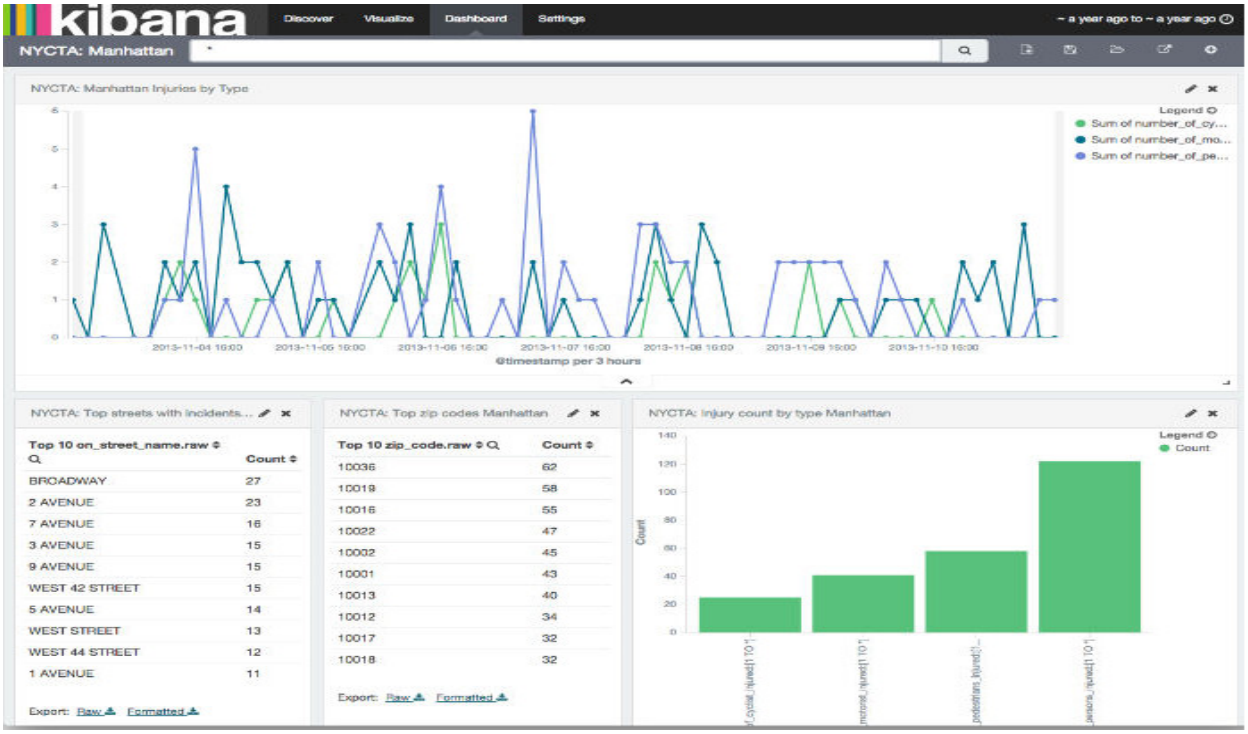


Ilustración 9: ejemplo de dashboard de Kibana

3.5 PYTHON

Aparte de la plataforma en sí, ha sido necesario el desarrollar varios *scripts* para obtener y pre procesar la información. Se hablará más en detalle de estos *scripts*, pero ahora es conveniente explicar que se han hecho con *Python* y esto es debido a su versatilidad, fiabilidad, su accesibilidad (*open source*) y comodidad.

3.5.1 Concepto

Python es un lenguaje de programación interpretado cuya filosofía radica a la simpleza del código. Fue creado a finales de los ochenta por Guido van Rossum, en el centro para las matemáticas e informática de los países bajos. Fue concebido como sucesor del lenguaje ABC, que manejaba excepciones e interactuaba con el SO Amoeba.

Python se centra en una sintaxis clara, que favorezca la legibilidad del código. Estamos hablando de un lenguaje orientado a objetos, interpretado y multiplataforma. A esto tenemos que añadirle la resolución dinámica de variables (conocido también como enlace dinámico de métodos). Se hablará más en detalle de las características de Python más adelante.

Actualmente, Python se desarrolla como un proyecto de código abierto administrado por la PSF (Python Software Fundation).

3.5.2 Características

Se trata de un lenguaje interpretado, es decir, no necesita para ejecutarse, lo hace directamente desde el código fuente. Su tipado dinámico hace que no sea necesario declarar las variables en nuestros métodos, cuando asignamos su valor por primera vez recoge automáticamente el tipo. La variable mantendrá su tipo durante todo su tiempo de vida.

Python ofrece estructuras de datos fáciles de usar como pilas, listas, diccionarios como parte intrínseca del lenguaje. Para ahorrar tiempo y esfuerzo a la hora de usar estos objetos, Python ofrece de serie operaciones de uso habitual como búsquedas u ordenaciones.

Este lenguaje ofrece una gran y variada colección de librerías, cada una de ellas dedicada a tareas específicas, como por ejemplo, trabajar con datos en formato JSON, mandar y recibicir peticiones HTTP, y un largo etcétera.

Otra característica de Python es su administración automática de la memoria. Solicita por su cuenta memoria cuando crea objetos y la libera cuando no serán utilizados otra vez.

Por otra parte, a los programas escritos en Python se les puede añadir componentes escritos en otros leguajes, como por ejemplo C o C++ usando el API Python/C. Esto le permita extender sus funcionalidades, lo que convierte a Python en un lenguaje de prototipado rápido: los programas pueden ser escritos en primera estancia con Python para aumentar la velocidad de desarrollo, pudiendo reescribir más tarde ciertas partes en otros lenguajes por temas de eficiencia.

Python sirve de manera muy eficaz para el desarrollo de proyectos de gran tamaño, como por ejemplo *Zope*, servidor de aplicaciones, o *BitTorrent*, sistema de intercambio de ficheros.

Dentro de la completa librería de Python, son particularmente interesantes aquellas que ofrecen módulos para manejar estándares muy usados en internet como JSON, XML, HTTP, etc. Junto a estas cabe mencionar aquellas que permiten comunicación con bases de datos para gestores como Oracle o MySQL.

3.6 JSON

JSON, siglas de JavaScript Object Notation, es un formato de texto ligero para el intercambio de datos. Aunque es un subconjunto de la notación de los objetos de JS, debido a su amplio uso como alternativa a XML, se considera un formato de lenguaje diferente.

Una de las ventajas frente a *XML* es que es mucho más sencillo escribir un analizador sintáctico (*parser*) de *JSON*. Ha sido aceptado por parte de la comunidad de desarrolladores *AJAX* debido a gran facilidad que ofrece JS para analizar este formato, bastando únicamente con la función *eval().*

JSON se emplea habitualmente en entornos donde el tamaño del flujo de datos entre cliente y servidor es de vital importancia. Es por esto por lo que es usado por Yahoo, Google, etc, que atienden a millones de usuarios.

JSON se puede construir con dos estructuras:

- Una colección de pares nombre/valor. En multiples lenguajes, esto se llama objeto, record, structura o diccionario→ {nombre1:valor1, nombre2:valor2}
- Una lista ordenada de valores. En multiples lenguajes, se trata de un array, vector, lista o secuenca. → {nombre1,nombre2,nombre3}{valor11,valor12,valor13} {valor21,valor22,valor23}

Todos los lenguajes modernos soportan JSON en, al menos, una de las dos estructuras.

3.7 HTTP

HTTP (Hiper Text Transfer Protocol) es el protocolo de comunicación que permite las transferencias de información en la World Wide Web. Fue desarrollado por el World Wide Web Consortium e Internet Engineering Task Force, culminando con la publicación de una serie de RFC definiendo la sintaxis y semántica a utilizar por los elementos de la arquitectura web (clientes, servidores, proxies).

HTTP es un protocolo sin estado, es decir, no guarda ninguna información sobre conexiones anteriores. Este protocolo está orientado a transacciones, siguiendo el esquema petición-respuesta entre un cliente y un servidor. El cliente realiza un petición enviando un mensaje con cierto formato al servidor. El servidor recoge esta petición, realiza la lógica pertinente, y envia al cliente la respuesta. Los navegadores web, por ejemplo, son clientes.

CAPÍTULO 4: DESARROLLO DEL PROYECTO

En este capítulo se explicara, paso a paso, el desarrollo completo del proyecto presentado en esta memoria. Constará de cuatro partes, bien diferenciadas. Una primera en la que se comentará el proceso de instalación y configuración de las diferentes tecnologías usadas para conformar el sistema objetivo.

Las dos partes siguientes consisten en dos desarrollos distintos. El primero, que podría considerarse "de prueba" consiste en usar una cantidad limitada de datos reales almacenados en varios ficheros. En este caso, se usará Logstash como medio para llevar la información de manera estructurada a Elasticsearch para que este último lo almacene. Se empleara el mapeo estático de ES, ya que se trata de datos predefinidos.

La tercera parte consiste en un desarrollo diferente, en el cual se usarán datos reales, pero esta vez obtenidos en tiempo real del flujo de eventos que ofrece GitHub. Los eventos serán recogidos y cargados directamente en Elasticsearch mediante el protocolo HTTP y como vía alternativa a la herramienta Logstash. En este caso se empleara mapeo dinámico puesto que la estructura y campos de los datos serán muy variantes. De este modo se podrán ver las ventajas de este método frente al mapeo estático.

Tanto en la segunda parte como en la tercera, los datos serán visualizados en diversos dashboards ofrecidos en Kibana. Estos dashboards permitirán el análisis de los datos obtenidos en los eventos. Este análisis se realizará usando únicamente la información mostrada por Kibana en tiempo real, y servirá para demostrar el potencial de dicha herramienta y del sistema total.

La cuarta y última parte consiste en una valoración global del sistema creado así como posibles usos reales del mismo, a parte del visto en el desarrollo 2. Se determinará en este punto si los objetivos especificados al principio de esta memoria se han visto resueltos finalmente. Se añadirán, además, posibles maneras de mejorar la herramienta, haciéndola más versátil, potente y útil en el mercado.

4.1 Parte I: Instalación y Configuración

Se empezará hablando de como instalar y configurar cada una de las tres herramientas que forman parte de este proyecto. Cabe decir que todas ellas son rápidas de instalar y fáciles de configurar, y en relativamente poco tiempo se dispone de un sistema ya funcionando. Todo esto es así gracias a que los desarrolladores se plantearan como objetivo la simplicidad de los primeros pasos, evitando tediosos tutoriales y ficheros de configuración.

Obviamente, el orden de instalación y configuración es totalmente indiferente, pero por seguir una estructura se empezará por Logstash y concluirá con Kibana, siendo ES el elemento en medio. De esta manera se seguirá el mismo orden que en el *stack* LG-ES-KB.

4.1.1 Instalando Logstash

Tanto la instalación como configuración de Logstash resultan asombrosamente fáciles, mucho más de lo normal dentro del software libre. Basta con descargarse el *.tar* de la página oficial: https://www.elastic.co/downloads/Logstash. Una vez descargado el .tar, se accede a la carpeta en la que se encuentra y se descomprime mediante el comando *tar -xzvf Logstash.tar.gz* (donde *Logstash.tar.gz es* el archivo comprimido, el cual puede recibir cualquier nombre).

Para completar la instalación, bastará con lanzar el comando *bin /Logstash,* dentro de la carpeta donde se haya descomprimido el archivo.

La instalación de Logstash resulta casi inexistente, ya que después de la instalación ya estaría listo para ejecutarse. Pero dicha ejecución sí depende de unos archivos, de extensión *.conf*, de los cuales Logstash extrae toda la información que necesita para funcionar y tratar los datos correctamente. Cabe decir entonces que dichos ficheros son la “configuración” de Logstash. Estos deben ser indicados en cada ejecución, y en ellos aparecerá información como de donde recibiremos los datos, que filtros se deben aplicar y a qué salida mandará Logstash los eventos ya procesados.

empresas.conf ×

```
input {
       file { path => "/home/manupg/Escritorio/ficheros_prueba/empresas_procesado.json"
              start_position => beginning
       }
}

filter {
 json {
   source =>"message"
 }
}

output {
 elasticsearch {
                  host => localhost
                  protocol => http
                  index => "logstash"
                  document_type => "prueba"
}

 stdout { codec => rubydebug}
}
```

Ilustración 10: Fichero empresas.conf

En la imagen se observa la estructura del fichero *empresas.conf,* perteneciente al proyecto tratado en esta memoria. Como se observa a simple vista, se trata de datos en formato JSON. Este fichero sirve perfectamente de ejemplo, ya que se observan cada una de las tres principales partes dentro del flujo de LG: *input-filter-output.*

- **Input**: En *path* se le especifica de donde nos llegan los datos. En este caso, se pone la ruta del fichero del cual LG leerá la información.

  Con *start-position* se le indica cómo debe leer el fichero, ya que LG guarda en otro fichero auto generado la última línea que leyó. De esa manera, empezaría a leer a partir de dicha línea, cargando así solo la información nueva. Al marcar esta opción como beginning, se indica que siempre deberá leer el fichero desde el principio.

- **Filter:** En esta sección se le indica a LG que filtros debe aplicar a la información recibida. Estos se aplicarán en el mismo orden en el que sean declarados. En este caso concreto, se aplica solo el filtro *JSON*, el cual formatea los datos de entrada para guardarlos en este formato.

- **output**: Salida a la que va destinada la información una vez procesada por LG. En este caso específico se han declarado dos distintas. La primera es Elasticsearch, que como ya se ha dicho en varias ocasiones, será el sistema de almacenamiento y búsqueda de este proyecto. Se han especificado propiedades concretas de ES, como el *index* y el

document_type. Con estas dos propiedades LG le dirá a ES en que índice y bajo qué tipo deberá guardar los eventos.

Bajo Elasticsearch se ha añadido también la *stdout*, que hace referencia a la salida estándar, en este caso la Shell de Linux. Además, se ha especificado un codec para esta salida, el *rubydebug*. Este *codec* te mostrará la información usando la librería de impresión de Ruby.

Para usar un determinado fichero .conf al ejecutar LG, basta con incluirlo en el comando de Shell*: ./Logstash –f fichero_a_usar.conf.*

4.1.2 Instalando Elasticsearch

Al igual que LG, ES posee un sistema de instalación y configuración simple, intuitivo y rápido. Cierto es que, a diferencia de LG, ES si necesita de una ligera modificación de ficheros de configuración. Pero esto, como se observará a continuación, no supone la mayor complejidad.

Una vez más, descargamos el *.tar* :https://www.elastic.co/downloads/Elasticsearch.

A continuación, se lanza el comando *tar -xzvf Elasticsearch.tar.gz* dentro del directorio en el que se encuentra el archivo comprimido. Usando el comando *bin/Elasticsearch* se completaría la instalación.

El siguiente paso es acceder a los ficheros de configuración para añadir ligeras modificaciones que permitirán que todo funcione como se espera. Accedemos al directorio *config*, dentro de la carpeta de ES instalada. Ahora, en el fichero *Elasticsearch.yml*, se realizan los siguientes cambios:

- **cluster.name:** Elasticsearch→ **cluster.name**: nombre_xxx.

  De esta manera se Evita que se puedan lanzar varios *cluster* con nombres iguales.

- **node.name:** “Franz Kafka” → **node.name:** “nombre_xxx”.

  Al igual que en el caso anterior, cambiando el nombre del nodo se evitarán confusiones y además, será más fácil localizarlos.

- **network.host**: 192.168.0.1 → **network.host**: 127.0.0.1.

  Mediante este cambio, Elasticsearch correrá de manera local, ideal para el desarrollo del sistema.

- Por último, se des comenta la constante "**bootstrap.mlockall:** true".

Tras estos 4 pequeños cambios en el fichero de configuración, ES estará listo para ser lanzado. Se accede a la carpeta anteriormente instalada y se ejecuta *./Elasticsearch*.

ES escuchará en la IP local en el puerto 9200 (http://localhost:9200 en el navegador).

4.1.3 Instalando Kibana

Como en los dos casos anteriores, el simple proceso de instalación comienza con la desgar el *.tar* de la web oficial : https://www.elastic.co/downloads/kibana.

Una vez más, se desensambla el archivo mediante el comando *tar -xzvf kibana.tar.gz,* dentro de la carpeta recién descargada.

Para ejecutar Kibana, bastará con lanzar *./Kibana.*

Por defecto, Kibana escucha en el puerto 5601 y apunta *a localhost:9200*, donde se ha configurado ES para que escuche.

Para completar la configuración, y poder empezar a "jugar" con el sistema ya completo habría que realizar una pequeña acción más. Consiste en crear un índice .kibana dentro de ES. En este índice se almacenarán las configuraciones de índices posteriores así como las distintas visualizaciones de los mismos y sus dashboards. Es decir, Kibana no solo usará la información almacenada en ES para nutrirse, si no que a su vez lo usará para almacenar su propia información.

Para crear el índice, se lanza ES y, ya sea mediante comando CURL o usando Sense, se usa la siguiente instrucción:

```
PUT .kibana
{
        "index.mapper.dynamic": true
}
```

Ilustración 11: indice .kibana

Como se observa, se crea el índice con la propiedad mapeo dinámico activada.

Tras este último paso se completaría la configuración necesaria para el correcto funcionamiento de Kibana en armonía con ES. Con ES funcionando, basta con acceder en el navegador a http://localhost.com:5601 para acceder al interfaz de usuario de Kibana.

4.2 Parte 1: Desarrollo 1

En el desarrollo a continuación se pretende usar cada una de las 3 tecnologías que forman el sistema de manera conjunta para procesar información a través de toda la cadena, es decir, desde su obtención hasta su representación en el *front web*.
En este desarrollo se tratará con una cantidad limitada y definida de datos, que sin ser una cantidad ínfima, dista mucho de las cantidades usadas en el desarrollo 2 de este proyecto. La información vendrá en una serie de ficheros en formato JSON con sintaxis abreviada.
Cada fichero contiene información distinta, de manera que uno contiene información sobre empresas, otro información de los proyectos, repositorios, etc. La información de cada fichero está relacionada entre sí mediante uno o varios campos. Es importante entender que ES no es una base de datos relacional, pero Kibana sí permite relacionar campos mediante los dashboards.

4.2.1 Obteniendo la información

Como se ha comentado antes, la información se encuentra distribuida en ficheros con formato JSON, un total de 6. Esta información se refiere a commits u eventos extraídos de GitHub, por lo tanto se trata de información real. Cada uno de estos ficheros contiene datos diferentes, y es conveniente explicar su contenido:

- ***scm-commits.JSON***: contiene información de los commits en sí, el id con el que se identifica en GitHub, la fecha en la que se realizó, el id del usuario, el id de la empresa a la que pertenece dicho usuario y la zona horaria en la que reside.

- **scm-commits-distinct.JSON**: contiene la misma información que el fichero anterior, añadiendo el nombre de usuario y de compañía.

- ***scm-messages.JSON***: Este fichero contiene los mensajes explicativos que se generan con cada evento. Sus campos son tres, el id del evento al que pertenece el mensaje, el mensaje en sí y su código hash.

- ***scm-orgs.JSON***: tiene dos únicos campos, id de la empresa y el nombre de la misma.

- ***scm-persons.JSON***: contiene información sobre los usuarios que generan los eventos capturados. Tiene cuatro campos distintos: id de usuario, nombre del mismo, el bot y el id del evento que generó.

- ***scm-repos.JSON***: esta vez los datos encontrados son los de los distintos repositorios, y el proyecto al que pertenecen los mismos, de los eventos generados. Se trata de cuatro campos, el id de repositorio y el nombre del mismo, el id del proyecto y el nombre de este.

Pre procesado de la información

Como se observa, el formato de los ficheros es *JSON* en sintaxis abreviada. Esta sintaxis permite que los ficheros ocupen mucho menos espacio, pero ofrecen un problema. No es el formato estándar de *JSON*, por lo tanto, Logstash no lo interpreta correctamente para sacar cada evento con sus respectivos valores. Debido a esto, será necesario un pre procesado que transforme esta sintaxis en sintaxis completa, que relaciona directamente cada campo con su valor.

Todo esto se realizará con un pequeño programa en Python, que recogerá un fichero de entrada, lo transformará y pasará a uno de salida, ya con el formato deseado.

El código es el siguiente:

```
import json

from pprint import pprint

with open("fichero_entrada.json") as data_file:
        data=json.load(data_file)



fichero= open("fichero_salida.json","a")
for i in range(len(data["valores"])):
        fichero.write(json.dumps({data["names"][0]:data["values"][i][0],
                                  data["names"][1]:data["values"][i][1]}, sort_keys=True))
        fichero.write("\n");
fichero.close()
```

Ilustración 12: código para pre procesado de los ficheros JSON

Como se aprecia, es un código muy simple que se basa en utilizar la librería *JSON* de Python. Esta librería hace que sea realmente sencillo trabajar con datos en este formato.

El programa consiste en coger un fichero de entrada y cargarlo como fichero JSON usando la función *JSON. load()* . Posteriormente, lo plasma en un fichero de salida mediante la función *JSON.dumps().* Esta función vuelca la información que se le pasa por parámetro en formato *JSON*.

4.2.2 Almacenando la información en ES

Mapeo de los datos

Una vez transformados los ficheros, ya estarían listos para ser procesados por Logstash. Sin embargo, antes de poder cargarlos en ES se tiene que definir el mapeo de cada tipo de datos, ya que en este desarrollo no se usará la propiedad de mapeo dinámico de ES. Se decidió de esta manera debido a que cada tipo de datos tiene una cantidad fija de campos y, de esta manera, resulta fácil determinar un mapeo.

En primer lugar, se crean los índices bajo los que se guardara cada tipo de dato. Bastará con usar la instrucción ***PUT*** en el editor *Sense* de ES, escribiendo también el nombre elegido para dichos índices. Estos han sido: *Logstash_commits*, *Logstash_commits2, Logstash_persons, Logstash_repos, Logstash_messages y Logstash_orgs*.

Una vez creados los índices, se determinan los mapeos, es decir, la estructura de datos de cada uno. Para esto, se usará también la instrucción PUT, seguida del nombre del índice del cual se quiere fijar el mapeo. A continuación, tras un salto de línea, se escribe el mapeo de la manera representada en la figura 13:

Como se ve, al declarar el mapeo se declaran uno a uno cada uno de los campos del índice, así como el tipo de dato del campo. Los campos *@timestamp*, *@version*, *message* y *path* son creados automáticamente por ES, aunque se hayan incluido en este mapeo para que fuera visible su declaración. @timestamp indica la fecha de creación del documento. @version, la versión del mismo, es decir, si un mismo documento ha sido guardado dos veces o más veces, se sabrá mediante el valor de este campo. *Message* y *path* indican el evento completo a modo de *string* y la ruta del fichero del que se extrajo el dato respectivamente.

```
put logstash_persons
{
  "mappings": {
        "prueba": {
          "properties": {
            "@timestamp": {
              "type": "date",
              "format": "dateOptionalTime"
            },
            "@version": {
              "type": "string"
            },
            "bot": {
              "type": "long"
            },
            "host": {
              "type": "string"
            },
            "per_id": {
              "type": "long"
            },
            "message": {
              "type": "string"
            },
            "name": {
              "type": "string",
              "index": "not_analyzed"
            },
            "path": {
              "type": "string"
            },
            "uuid": {
              "type": "string",
              "index": "not_analyzed"
            }
          }
        }
    }
}
```

Ilustración 13: mapeo del indice Logstash_persons

Fichero *.conf*

Una vez se han establecido cada uno de los mapeos para los distintos tipos de datos a cargar en ES, ya se podrá alimentar LS con estos para que se encargue de procesarlos y mandarlos a ES.

Para esto, como se ha comentado anteriormente, se necesitará especificarle a LS de donde obtener la información, así como cómo deberá procesarla y qué hacer con ella posteriormente. Esto se hará mediante los ficheros .conf, que se deben citar en la línea de comando a la hora de ejecutar la herramienta. En este caso concreto, bastará con un único fichero a pesar de contar con 6 archivos de datos distintos. Esto es así debido a que el formato de todos es el mismo, JSON, y a que la información ha sido pre procesada con anterioridad para que fuera compatible con LS.

El fichero se llamará ficheroDatos.conf y su contenido, el siguiente:

```
input {
        file { path => "/home/manupg/Escritorio/ficheros_prueba/empresas_procesado.json"
                start_position => beginning
        }
}

filter {
 json {
   source =>"message"
 }
}

output {
 elasticsearch {
                  host => localhost
                  protocol => http
                  index => "logstash"
                  document_type => "prueba"
}

 stdout { codec => rubydebug}
}
```

Ilustración 14: ficheroDatos.conf

Como se observa en la figura anterior, el fichero consta de las 3 partes ya concretadas anteriormente en este capítulo:

- En la propiedad *input* definimos de donde va a recibir LS los datos, siendo en este caso la ruta del directorio en el que se encuentran los ficheros con la información. Se ha declarado la propiedad *start_position* con valor *beginning*. Esta es una útil propiedad de LS que permite determinar por donde se desea que LS empiece a leer el fichero que se le indica. La primera vez que lee el fichero crea un fichero en el que guarda la última línea que se leyó del archivo. Si la *start_position* se declara como *end* y siempre y cuando LS este corriendo, este consulta el fichero periódicamente, de manera que si el fichero es modificado, lo detecta y sabe por qué línea debe continuar leyendo datos. Esto permite que LS cargue automáticamente las actualizaciones de los ficheros sin repetir información. Declarada como beginning, siempre que detecte cambio en el fichero lo leerá completo.

- En *filter* se especifica el tipo de datos tratados. En este caso se indica JSON, de manera que LS pueda parsear correctamente cada campo:valor.

- En *output* se indica a donde debe redirigir los datos LS tras procesarlos. Como se ve en el fichero, se concretan dos salidas. La primera, *Elasticsearch*, donde se almacenará toda la información. A esta salida se le deben indicar las propiedades :

    - *host*: IP donde escucha el nodo de ES. En este caso particular, *localhost*.

- *protocol:* protocolo que se usa para el envío de información.
- *index:* índice concreto en ES donde se van a almacenar los eventos del fichero.
- *document_type:* dentro del índice anterior, el tipo en el que se almacenarán los eventos del fichero.

La segunda, *stdout*, la salida estándar de la consola. Esta sirve únicamente para poder visualizar el proceso e ir comprobando que todo vaya bien.

Antes de ejecutar LS es importante recordar que ES debe estar corriendo, así el que siguiente paso consiste en ejecutar ES y abrir el editor Sense, el cual servirá para visualizar los datos almacenados correctamente.

A continuación ejecutamos LS pasándole el fichero analizado por línea de comando. Hay que especificar en dicho fichero la ruta del fichero que se quiere procesar, recordando siempre cambiarla para cada uno de los 6 ficheros de datos.

La ejecución irá mostrando por la terminal (*stdout*) los datos que se van cargando en ES. Aparecerán bastante rápido, pero la intención de esto no es seguir dato a dato la carga (lo cual es imposible) si no llevar una monitorización general del proceso y detectar, si se da el caso, algún error grave, como una interrupción no controlada.

Al mismo tiempo, se puede usar Sense para comprobar que los datos han sido cargados correctamente. Usando un comando _*search* con cualquiera de los índices, uno por cada fichero de datos, se obtendrá el número de documentos ya guardados en él. Desde *Sense* se pueden realizar una gran variedad de acciones sobre los datos: desde búsquedas simples a complejas modificaciones. Pero donde de verdad le sacaremos provecho a la información almacenada será en Kibana.

Ilustración 15: busqueda _search con Sense

El siguiente paso a realizar será la utilización de Kibana para generar los dashboards que servirán para visualizar los datos para, posteriormente, realizar un análisis sobre los mismos. Llegados a este punto, cabe decir que la escasa cantidad de datos y de tipos de los mismos, limita de cierta manera el análisis en este apartado.

4.2.3 Visualizando en Kibana

Con ES en funcionamiento, se lanza Kibana (de la manera explicada con anterioridad en este capítulo). Que LS este corriendo también y procesando datos al mismo tiempo es indiferente puesto que Kibana muestra los cambios en los índices en tiempo real. El tiempo de recarga de los dashboards se puede cambiar para adaptarlo a las necesidades.

Tras acceder a la URL donde escucha KB, seleccionamos el índice (o más bien índices) sobre el que vamos a trabajar. Una vez elegido aparecerán todos los campos del mismo. Al tratarse de distintos índices, habrá que crear diferentes visualizaciones para crear los dashboards. Sin embargo, todos ellos podrán ser mostrados juntos en una única pantalla. Esto permitirá aplicar filtros que afectarán a todos los dashboards por igual, pudiendo observar así las relaciones entre ellos.

Al entrar en KB, en la ventana *settings* se elige el índice sobre el que se desea trabajar. KB preguntara si se trata de eventos temporales, es decir, si su naturaleza depende del momento de ocurrencia o un tiempo concreto. En este desarrollo no se trabaja con eventos de ese tipo, puesto que se trata de eventos de datos informativos no temporales. En la segunda parte del desarrollo sí se trabajara con eventos dependientes del momento en el que son generados.

Una vez elegido el índice, en la ventana *discover* podremos ver cada uno de los eventos cargados .Es importante no confundir el campo *timestamp* con el campo que indica la fecha de

generación del evento. El *timestamp* es la fecha exacta en el que el evento se cargó en ES. Puede coincidir o no con la fecha de generación del evento. En este caso, esto es irrelevante puesto que carecemos de fecha de generación al no tratarse de eventos temporales.

Ilustración 16: eventos procesados mostrado en Kibana

En la imagen anterior se observa cada uno de los campos de los eventos almacenados en el índice *messages.* Algunos, como ya se explicó con anterioridad, son generados por ES al almacenar dicho evento.

En la parte superior de esta misma ventana aparece un diagrama de barras mostrando el número de eventos registrados en el rango de fecha que se indique.

Generación de dashboards

Para gestionar la creación de las distintas gráficas, diagramas y representaciones, se debe acceder a la ventana *visualize*. Tras seleccionar el tipo de diagrame/representación y el índice a representar, aparecerán las opciones para la representación en el menú de la izquierda. Se selecciona la manera en la que se quieren ver los datos en la representación: los *x* primeros, en histograma, el total, un rango concreto de datos. También se deberá seleccionar el campo concreto dentro de los documentos del índice que se desea representar. A continuación se muestran representaciones del índice *persons* en dos *pie chart*:

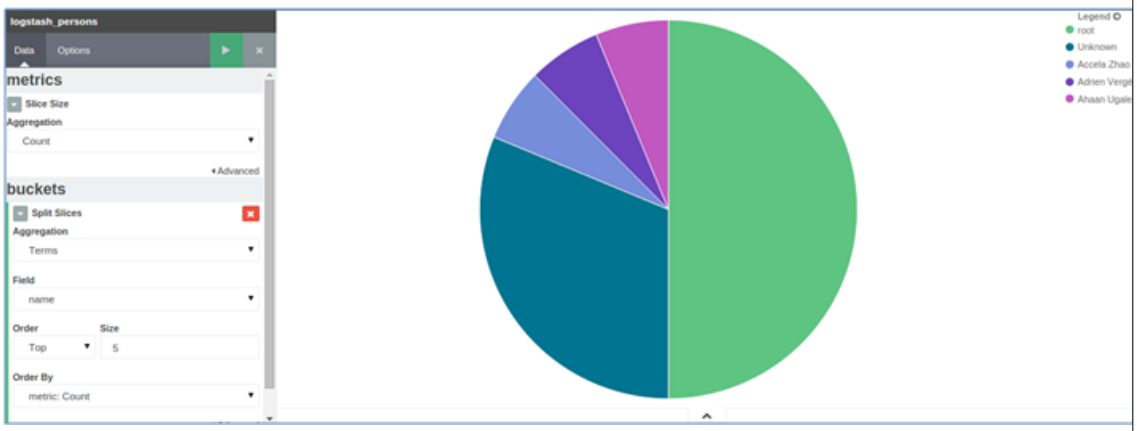


Ilustración 17: pie chart top 5 de usuarios

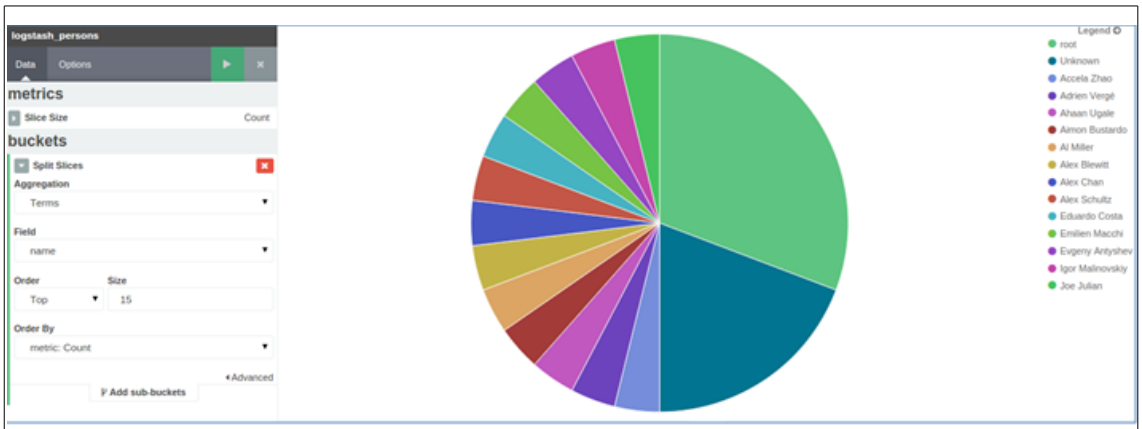


Ilustración 18: pie chart top 15 de usuarios

La primera figura consiste en un pie chart que muestra los 5 usuarios que aparecen el mayor número de veces (top 5) entre los eventos registrados. La segunda muestra el top 15. En la esquina superior derecha aparece una leyenda relacionando cada usuario con un color. Si el ratón se posa sobre una de las secciones del *pie chart*, se muestra el número total de eventos que corresponden a dicha sección, así como el porcentaje que ocupa esa sección dentro del diagrama.

El *pie chart* es uno de los más usados, pero es solo uno de los muchos tipos de representaciones que KB ofrece. Utilizar el tipo de diagrama correcto para los datos que se quieren representar es clave para poder extraer los análisis correctos.

Cada índice y tipo de datos que almacenados deberán tener una representación. Esta no deberá tener todos los campos presentes en los documentos, solo aquellos con la información que interese en cada momento. Hay campos, como los *id* de *commit*, que no aportan información relevante para mostrar, sin embargo, son totalmente necesarios para relacionar unos tipos de datos con otros.

Dashboards generados

Tras haber usado un ejemplo para explicar el sistema de generación de visualizaciones de Kibana, se verán los distintos dashboards creados, así como la utilidad de los mismos.

En primer lugar, se genera un *dashboard* centrado en los usuarios y la empresa a la que pertenece cada uno de ellos. Para ellos se crea un diagrama de barras representando los 10 usuarios que más número de *commits* realizaron. El diagrama representa los *id* de cada uno, ya que se crea con el índice *commits* y este índice no contiene el nombre del usuario que ha hecho el *commit*, pero si el *id* correspondiente. Para darle sentido a este diagrama, se crea una tabla de datos con la relación nombre-id de cada usuario. Esta tabla utiliza el índice *persons.*

Ambos índices tienen un campo de que los relaciona, *person_id*. Además, este campo servirá para relacionar los índices de otra manera muy interesante. Los documentos del índice *commit* tienen un campo, org_name, que índica la empresa a la que pertenece el usuario que realizó el commit. Por lo tanto, el campo *person_id* da la posibilidad de relacionar la empresa con el nombre del usuario. Por supuesto, representar estos datos y tener que buscar en cada tabla la relación una a una no tendría sentido. Por eso, se usarán los filtros, que se aplican al hacer *click* en uno de los campos de los diagramas. Por ejemplo, si se hace *click* en una de las columnas del diagrama de barras, se recargaran los dashboards mostrando los datos relacionados con el campo de esa barra, en este caso un id de usuario. De esta manera se sabrá el nombre del usuario y la empresa al que pertenece.

Dejando atrás el *dashboard* centrado en los usuarios, se genera otro con objetivos más generales. En él llama la atención a primera vista el recuadro con un número en grande. Este número representa el número total de *commits* almacenados. Este número irá variando en función de los filtros que apliquemos en los otros dashboards.

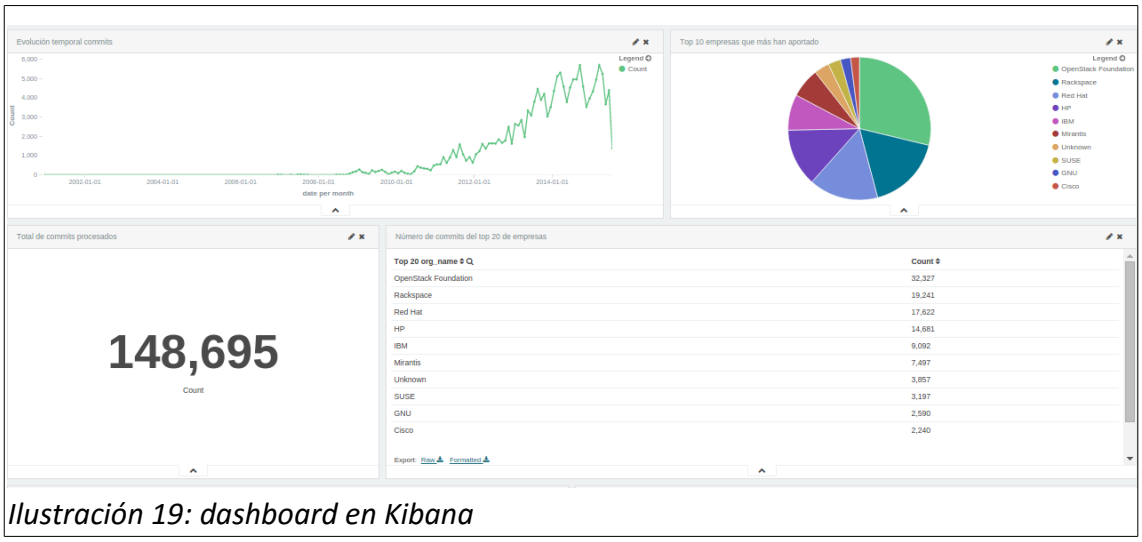


Ilustración 19: dashboard en Kibana

En la parte superior del dashboard aparecen dos vistosos diagramas. En el de la izquierda se representa la evolución temporal de los commits, medida en commits por fecha. La escala en el eje x es anual por que el objetivo es el de representar los cambios en grandes rangos de tiempo, para una visión más generalizada.

A la derecha del anterior se encuentra un pie chart, el cual contiene los 10 empresas más activas dentro de los 148695 eventos registrados. Se entiende por más activas a aquellas cuyos empleados realizaron más commits.

Más abajo aparecen tablas con diversos datos extraídos de los eventos: Los nombres de todas las empresas colaboradoras, los 20 usuarios más activo por nombre y por id, así como todos los mensajes explicativos en cada commit. Estas tablas por su cuenta aportan información de interés, pero son los filtros los que permiten explotarlas al máximo al poder relacionarlas. Si se pulsa en uno de los mensajes, por ejemplo, se podrá activar el filtro por id o nombre del usuario que lo creó. Automáticamente, se actualizarán todos los diagramas y tablas mostrando la información correspondiente a ese usuario: el número total de sus eventos, su empresa, la evolución temporal de su trabajo, sus mensajes, etc.

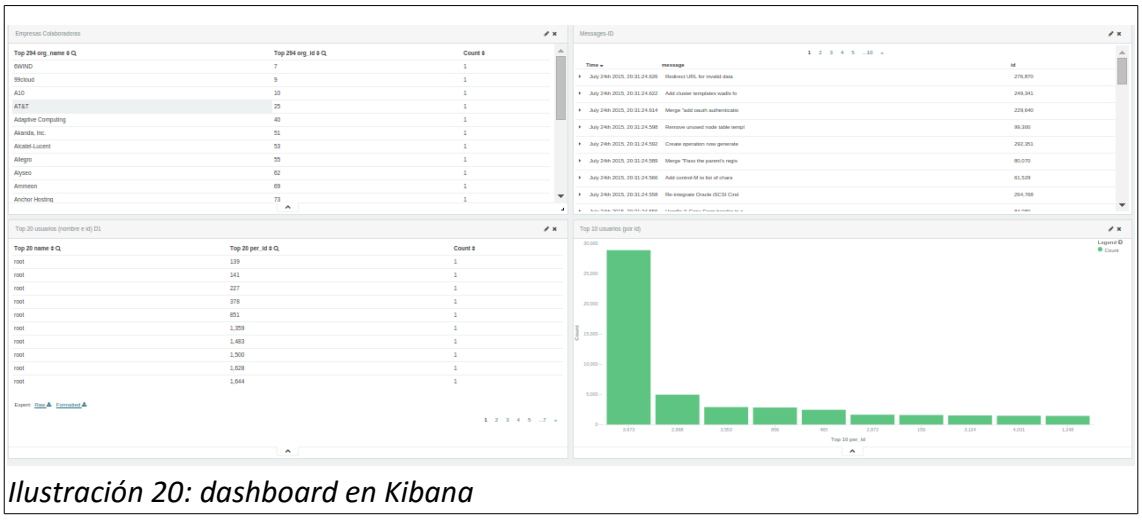


Ilustración 20: dashboard en Kibana

Con todo esto se empieza a ver el potencial de la herramienta. Sin embargo, no deja de tratarse de datos estáticos en ficheros. Esto, obviamente, limita mucho las posibilidades. A continuación, en el desarrollo 2, se procesaran eventos mucho mayores, con una cantidad mayor de información y habrá más diversidad de tipos de los mismos. Sumado a esto, se procesarán en tiempo real. Todo esto permitirá completar el verdadero objetivo de la herramienta y la herramienta en sí.

4.3 Parte 2: Desarrollo 2

Se trata del desarrollo principal del proyecto, en el que se ve el verdadero funcionamiento de la herramienta objetivo, así como sus capacidades y posibles explotaciones de la misma. En este caso se trabaja con datos reales una vez más, pero esta vez la cantidad, tamaño y variedad de eventos superará con creces a los del desarrollo anterior.

La información a tratar serán eventos reales de GitHub generados y capturados en tiempo real. Es decir, los eventos se cogerán, procesaran, almacenaran y visualizaran al tiempo que se crean. La información es proporcionada por la API de GitHub, y estará en formato JSON una vez más. Esto, como se verá, facilita muchísimo las cosas y es uno de los motivos por los cuales se eligió esta fuente de datos. Otros motivos, más relevantes que el anterior, son su fiabilidad, envergadura y el hecho de que los proporciona en tiempo real. Además, debido a la fama de GitHub y el tamaño de su comunidad de usuarios, se generan grandes cantidades de eventos por hora.

La captura de los eventos se hará mediante HTTP con un *script* Python, en concreto, usando dos librerías principalmente: *urllib* y *request*. Juntas permiten un uso completo del protocolo HTTP con Python.

Una vez recogidos los eventos, se almacenarán en Elasticsearch. Para ello, se tienen dos opciones. La primera, como en el desarrollo anterior, consiste en pasarlos a un fichero que Logstash procesará. La segunda consiste en cargarlos directamente en Elasticsearch mediante comandos *Curl*.

La carga en Elasticsearch también presentará modificaciones con respecto al punto anterior, siendo la principal de ellas el uso del mapeo dinámico. En esta ocasión se guardaran todos los eventos bajo un mismo índice, y esto es posible gracias al mapeo dinámico, que permite un mapeo que se adapte a eventos con diferente número y tipo de campos. El mapeo estático usado en el punto anterior no es aplicable en este caso, ya que todos los eventos se capturan igual y se necesitaría de un pre procesado de cada uno para identificar los campos del mismo y asignarle el índice correcto. Esto haría la herramienta ineficiente.

La representación de la información, una vez más, correrá a cargo de Kibana. Se usarán dashboards similares a los del desarrollo 1 pero en este caso, debido a la gran cantidad de datos y variedad de los mismos, presentaran información mucho más extendida y completa.

4.2.1 Obteniendo la información

Como se ha explicado en la introducción del proyecto, la información cosiste en eventos JSON obtenidos de la API de GitHub. A esta API (explicada en el apartado 3: tecnologías usadas) se accede mediante HTTP en la url: https://API.GitHub.com .

Las **peticiones** GET tienen el siguiente formato :

http://api.github.com/events?per_page=xxxx&page=y

Donde se le pasan dos atributos. El primero “per_page”, que es el número de eventos que queremos por página, hasta un total de 300 eventos máximos por petición y por página. Si se indicán, por ejemplo, 300 eventos, se generará una única página. En el caso de elegir menos eventos por página, se generan tantas páginas como sea necesario para llegar a los 300 eventos, teniendo en cuenta que cada página tendrá el mismo número de eventos. En el segundo, “page”, se indica la página a la que queremos acceder.

Ejemplo de petición:

http://api.github.com/events?per_page=100&page=2

En el ejemplo anterior, se han especificado 100 eventos por página. Cada petición te devuelve 300 eventos, por lo tanto tendremos 3 páginas totales. En el segundo parámetro se indica que se quiere la segunda página.

El formato de **respuesta** es el siguiente:

```
Status: 200 OK
```


```
[
 {
   "type": "Event",
   "public": true,
   "payload": {
   },
   "repo": {
     "id": 3,
     "name": "octocat/Hello-World",
     "url": "https://API.GitHub.com/repos/octocat/Hello-World"
   },
   "actor": {
     "id": 1,
     "login": "octocat",
     "gravatar_id": "",
     "avatar_url": "https://GitHub.com/images/error/octocat_happy.gif",
     "url": "https://API.GitHub.com/users/octocat"
   },
   "org": {
     "id": 1,
     "login": "GitHub",
     "gravatar_id": "",
     "url": "https://API.GitHub.com/orgs/GitHub",
     "avatar_url": "https://GitHub.com/images/error/octocat_happy.gif"
   },
   "created_at": "2011-09-06T17:26:27Z",
   "id": "12345"
 }
```

```
]
```

Todo lo anterior que se encuentra dentro de "[ ]" corresponde a un único evento. Como se observa, se trata de formato JSON en su totalidad. Separados por "{}" están los campos y, dentro de los mismos, también separados por "{}", están los sub campos. Cada campo se separa del anterior mediante ",". Se debe tener en cuenta que este evento es de ejemplo, los eventos reales capturados son de bastante mayor envergadura.

Para conseguir un procesado de la información en tiempo real se crea un script Python encargado de las peticiones GET a la API y de procesar e enviar la respuesta a Elasticsearch. Para ello, se hará uso de tres librerías principalmente: *JSON*, *urllib* y *request.* La primera permite trabajar con datos JSON muy cómodamente, mientras que las dos siguientes sirven para trabajar con el protocolo HTTP.

A continuación se presenta y explica el script:

```
import requests
import json
import time
from pprint import pprint
import urllib2

#PRIMERA CARGA
#Cargamos las tres paginas de eventos (100 cada una)

datos1= requests.get('https://api.github.com/events?per_page=100&page=1',auth=('manupg', 'volvagia9'))
datos2= requests.get('https://api.github.com/events?per_page=100&page=2',auth=('manupg', 'volvagia9'))
datos3= requests.get('https://api.github.com/events?per_page=100&page=3',auth=('manupg', 'volvagia9'))

#Guardamos el ETAG de la primera, asi como el ID del primer evento de la primera pagina.
#Esto nos evitara cargar eventos repetidos

jsona=datos1.json()
id_evento=jsona[0]["id"]
opener = urllib2.build_opener(urllib2.HTTPHandler)
url="http://localhost:9200/urllib2/"

for i in range(3):
        if i == 0:
                datos=datos1

        elif i == 1:
                datos=datos2
```

Ilustración 21: script python capturador de eventos (1)

En primer lugar , se realizan las tres peticiones GET a la *url* de la API mediante el método *get* de la librería *request*. Se especifican tanto los eventos por página y las páginas. Además, se debe especificar el usuario y contraseña en la cabecera auth de la petición. Tener usuario registrado en la API no es obligatorio, sin embargo, sin el se tiene un limite de peticiones por hora. Este límite te proporciona un número de eventos muy por debajo del necesario para recoger todos los eventos generados en una hora. Por lo tanto, para la finalidad del sistema, el registrarse es mandatario.

Los siguientes pasos consisten en almacenar los datos transformados a un objeto JSON en JSONa. En id_evento guardamos el ID del primer evento cargado. Esto evitara cargar eventos repetidos ya que a partir de ese ID se sabe que los eventos serán nuevos. En url se indica la *url* donde escucha Elasticasearch, es decir, donde queremos mandar los datos para almacenarlos. En esta url se indica ademas el indice bajo el que serán cargados.

```
for i in range(3):
        if i == 0:
                datos=datos1

        elif i == 1:
                datos=datos2

        elif i == 2:
                datos=datos3

        jsona= datos.json()
        fichero=open("github.json","a")


        for i in range(len(jsona)):

                data_json = json.dumps(jsona[i])
                url_ela=url+jsona[i]["type"]+"/"+jsona[i]["id"]

                request = urllib2.Request(url_ela, data=data_json)
                request.get_method = lambda: 'PUT'
                response = opener.open(request)


#CARGAS SIGUIENTES
##time.sleep(10)
while True:
##comienzo bucle
```

Ilustración 22: script python capturador de eventos (2)

El bucle mostrado en el código itera cada una de las páginas con eventos que se han obtenido con la petición get. A continuación, para cada una de las páginas se itera sobre cada uno de los eventos almacenados en la lista JSONa, montando la petición PUT mediante la concatenación de la url y cada tipo e id del evento. Posteriormente se realiza la petición PUT a la url final, pasando como data en el cuerpo de la petición el evento correspondiente de JSONa.

Hasta ahora, se estaba procesando la primer carga, en la cual no se comprobaba que se te capturara algún evento repetido. A partir de este punto, se entra en bucle infinito almacenando tras cada iteración el id del primer evento de la primera página para tener control de hasta que evento se ha cargado.

```
#CARGAS SIGUIENTES

while True:
##comienzo bucle
        datos1= requests.get('https://api.github.com/events?per_page=100&page=1',auth=('manupg', 'volvagia9'))
        datos2= requests.get('https://api.github.com/events?per_page=100&page=2',auth=('manupg', 'volvagia9'))
        datos3= requests.get('https://api.github.com/events?per_page=100&page=3',auth=('manupg', 'volvagia9'))
        quit=False
        for i in range(3):
                if i == 0:
                        datos=datos1
                        print("PROCESAMOS LA PAGINA 1")
                elif i == 1:
                        datos=datos2
                        print("PROCESAMOS LA PAGINA 2")
                elif i == 2:
                        datos=datos3
                        print("PROCESAMOS LA PAGINA 3")
                jsona= datos.json()
                fichero=open("github.json","a")
                for i in range(len(jsona)):
                        if jsona[i]["id"] != id_evento:

                                data_json = json.dumps(jsona[i])
                                url_ela=url+jsona[i]["type"]+"/"+jsona[i]["id"]

                                request = urllib2.Request(url_ela, data=data_json)
                                request.get_method = lambda: 'PUT'
```

Ilustración 23: script python capturador de eventos (3)

El funcionamiento es básicamente el mismo que en la primera carga, con la diferencia de que esta vez se comprueba cada id para comprobar que no ha sido cargado ya. Tras cada iteración de página se imprimen por pantalla el número total de eventos cargados y repetidos de esa misma página. Además, se almacena el id del primer evento de la página, tanto en la variable id_evento como en un fichero aparte para tener control de hasta que evento se carga.

El script anterior es el encargado de capturar los eventos directamente del flujo de la API e enviarlos directamente a Elasticsearch para su almacenamiento, todo esto mediante el protocolo HTTP.

4.2.1 Almacenando la información

Como se ha comentado anteriormente, el formato de los datos, así como el número de campos de los mismos y valores , es muy variado. Aquí se muestra un ejemplo de evento real, en la *ilustración 25*. Los cinco primeros campos, empezando por "_", son creados por ES cuando el eventos es cargado. Estos campos indican el indice bajo el cual se ha almacenado, así como el tipo y el id del propio evento dentro de la API de GitHub. Como se observa y se ha insistido antes en esta memoria, el formato del dato (documento) es JSON.

```
                                request.get_method = lambda: 'PUT'
                                response = opener.open(request)

                        else:
                                print("EVENTO REPETIDO, NO SE GUARDA A PARTIR DE AQUI: " + jsona[i]["id"] +"=" + id_evento  )
                                quit=True
                                break
                fichero.close()
                print ("Eventos cargados de esta pagina ==> "+ str(i+1) + "\n")
                print ("Eventos repetidos de esta pagina ==> "+ str(99-i) + "\n")

                if quit==True:
                        break
##ahora guardamos el primer evento de la nueva primera pagina, ya que hasta ese evento, todos han sido cargados.
        jsona=datos1.json()
        id_evento=jsona[0]["id"]
        fichero=open("ultimo_id.txt","w")
        fichero.write(id_evento)
        fichero.close()
        ##time.sleep(10)
##fin bucle
```

Ilustración 24: script python capturador de eventos (4)

Mapeo Dinámico

El evento mostrado no es excesivamente grande, sin embargo, algunos de ellos tendrán un número de campos mayor, así como mayores valores de los mismos. Por todo esto, el mapeo dinámico de Elasticsearch cobra mucha importancia a la hora de permitir el cometido del sistema aquí presentado. Sin este tipo de mapeo no se podría conseguir el objetivo, dado que el estático presenta muchas limitaciones a la hora de procesar datos en tiempo real.


```
{
   "_index": "urllib2",
   "_type": "PushEvent",
   "_id": "3233528303",
   "_score": 1,
   "_source": {
      "payload": {
         "size": 1,
         "head": "6287031fc5cff5bc2130d67350947e0889bec29d",
         "commits": [
            {
               "distinct": true,
               "sha": "6287031fc5cff5bc2130d67350947e0889bec29d",
               "message": "including participants in event's serializer",
               "url": "https://api.github.com/repos/Rodic/startit-api/commits/6287031fc5cff5bc2130d67350947e0889bec29d",
               "author": {
                  "email": "alec.rodic@gmail.com",
                  "name": "rodic"
               }
            }
         ],
         "distinct_size": 1,
         "push_id": 825218208,
         "ref": "refs/heads/master",
         "before": "4e04e34e527fe753d042bcd0041a3b8b47a916ac"
      },
      "created_at": "2015-10-13T13:17:11Z",
      "actor": {
         "url": "https://api.github.com/users/Rodic",
         "login": "Rodic",
         "avatar_url": "https://avatars.githubusercontent.com/u/1304709?",
         "id": 1304709,
         "gravatar_id": ""
      },
      "id": "3233528303",
      "repo": {
         "url": "https://api.github.com/repos/Rodic/startit-api",
         "id": 43434135,
         "name": "Rodic/startit-api"
      },
      "type": "PushEvent",
      "public": true
   }
},
```


Ilustración 25: evento real capturado mostrado en Sense

El mapeo dinámico no necesita ser declararlo. Basta con cargar un evento bajo ese indice. Al no encontrar un mapeo especificado para el indice en cuestión, EL le asigna automáticamente mapeo dinámico, recogiendo en el cada uno de los campos de ese evento. Cada vez que se guarde un evento en ese indice, el mapeo se adaptará, usando los campos ya declarados y añadiendo los nuevos que aporte el evento.

En la figura 26 se muestra parte del mapeo del indice *urllib2*, en el cual se encuentran los eventos recogidos de la API. Una vez más, la información se muestra en formato JSON.

Ilustración 26: mapeo del indice urrlib2

No figura en la imagen el evento en su totalidad, solo una parte ya que, en total, contiene más de 6000 lineas. Esto es así ya que cubre todas las posibilidades de los eventos que, hasta el momento, hayan sido almacenados. Como se ve, no todo son ventajas a la hora de usar mapeos dinámicos: en sistemas escasos de capacidad pueden ser un inconveniente.

Información cargada

Teniendo en cuenta que bajo este índice se han cargado la exacta cantidad de 1.835.428 eventos, es razonable encontrarse con un mapeo de este tamaño.

Para este desarrollo se almacenan cerca de 14 Gb de información recogidos, principalmente en dos indices. El primero contiene la cantidad arriba especificada y el segundo, *Logstash-GitHub*, 1.596.927 de eventos. Se cargó eventos en cada unos de ellos durante un total de 72 horas

```
{
   "took": 321,
   "timed_out": false,
   "_shards": {
      "total": 5,
      "successful": 5,
      "failed": 0
   },
   "hits": {
      "total": 1596927,
      "max_score": 1,
      "hits": [
         {
            "_index": "logstash-github",
            "_type": "PushEvent",
            "_id": "AU92IKFDLv83fP3xEAeZ",
            "_score": 1,
            "_source": {
               "message": "{\"payload\": {\"size\": 1, \"head\": \"3e12333bec97818ec9fa8321f28e87c00
\"sha\": \"3e12333bec97818ec9fa8321f28e87c001f332e4\", \"message\": \"Budget page bootstrapped, work
.github.com/repos/Timojarv/findalaptop/commits/3e12333bec97818ec9fa8321f28e87c001f332e4\", \"author\
\"name\": \"Timo Jarventausta\"}}], \"distinct_size\": 1, \"push_id\": 774108368, \"ref\": \"refs/he
4f1519df050bd0612a71361f72773\"}, \"created_at\": \"2015-08-28T21:04:44Z\", \"actor\": {\"url\": \"h
\"login\": \"Timojarv\", \"avatar_url\": \"https://avatars.githubusercontent.com/u/929971?\", \"id\"
\"3100189404\", \"repo\": {\"url\": \"https://api.github.com/repos/Timojarv/findalaptop\", \"id\": 4
\"type\": \"PushEvent\", \"public\": true}",
               "@version": "1",
               "@timestamp": "2015-08-28T21:04:50.289Z",
               "host": "manupg-VirtualBox",
               "path": "/home/manupg/Escritorio/ficheros_prueba/procesadores/github.json",
               "payload": {
                  "size": 1,
                  "head": "3e12333bec97818ec9fa8321f28e87c001f332e4",
                  "commits": [
                     {
                        "distinct": true,
                        "sha": "3e12333bec97818ec9fa8321f28e87c001f332e4",
                        "message": "Budget page bootstrapped, works on mobile too.",
                        "url": "https://api.github.com/repos/Timojarv/findalaptop/commits/3e12333bec
                        "author": {
                           "email": "timo.jaerv@gmail.com",
                           "name": "Timo Jarventausta"
                        }
                     }
                  ],
                  "distinct_size": 1,
```


Ilustración 27: total de eventos Logstash-GitHub

seguidas sin interrupción. En cada iteración del bucle del script encargado de capturar la información, se recogen cantidades muy variadas de datos, dependiendo del momento del día. En horas pico, se puede llegar a obtener entre 250 y 300 eventos por segundo. Se entienden por horas pico las principales horas de trabajo, en especial aquellas en las que coinciden en varios países. Cabe recordar que GitHub se usa a nivel mundial. En momentos mas relajados, como la madrugada, la cantidad de eventos por segundo desciende a poco más de 2 o 3.

En cualquier caso, estamos hablando de una cantidad de información almacenada lo suficientemente grande como poder analizar con precisión, al menos, los tres días en los que transcurren. Para realizar análisis, una vez más, se utilizará Kibana como herramienta de visualización de la información. Se ha de tener en cuenta que mediante *Sense,* el editor/cliente de Elasticsearch, se pueden lanzar búsquedas sobre toda la información contenida, así como modificaciones en la misma. Kibana no ofrece la posibilidad de modificar los indices y documentos

presenten en EL, sin embargo, las posibilidades de representación y uso de los datos son infinitamente superiores.

4.2 Visualizando la información

Una vez tratada y almacenada la información, el siguiente paso en el sistema es el cómo representar la información. Como se sabe, la tecnología encargada de esto, y que supone la última fase de la herramienta total, es Kibana. Como se ha visto en el desarrollo anterior, Kibana ofrece una alta variedad de diagramas, histogramas y métodos de representación en general. Estos permiten dar valor y uso a la información almacenada, que sería totalmente inservible por si misma, sin unificar en representaciones. Gracias a los *dashboards* que se generan se podrán determinar conclusiones tales como la empresa más activa en el momento, el repositorio en el que trabajan más personas, el usuario que más aporta a un proyecto e incluso las horas en las que este tiene mayor actividad. Más adelante se entrará en más detalle en los *dashboards* generados.

En primer lugar, se determina el índice sobre el que trabajar. Como se ha mencionado con anterioridad, para este desarrollo se cuenta con indices: *Logstash-GitHub* y *Urllib2.* El primero cuenta con cerca de 1.600.000 y el segundo, un total de 1.835.428. Lo eventos capturados en los mismos, dentro de la variedad de formatos y la aleatoriedad de la información generada, son del similares. Por lo tanto, se trabajará con el indice *Logstash-GitHub* y eso bastará para obtener resultados que se puedan aplicar también a *Urllib2*.

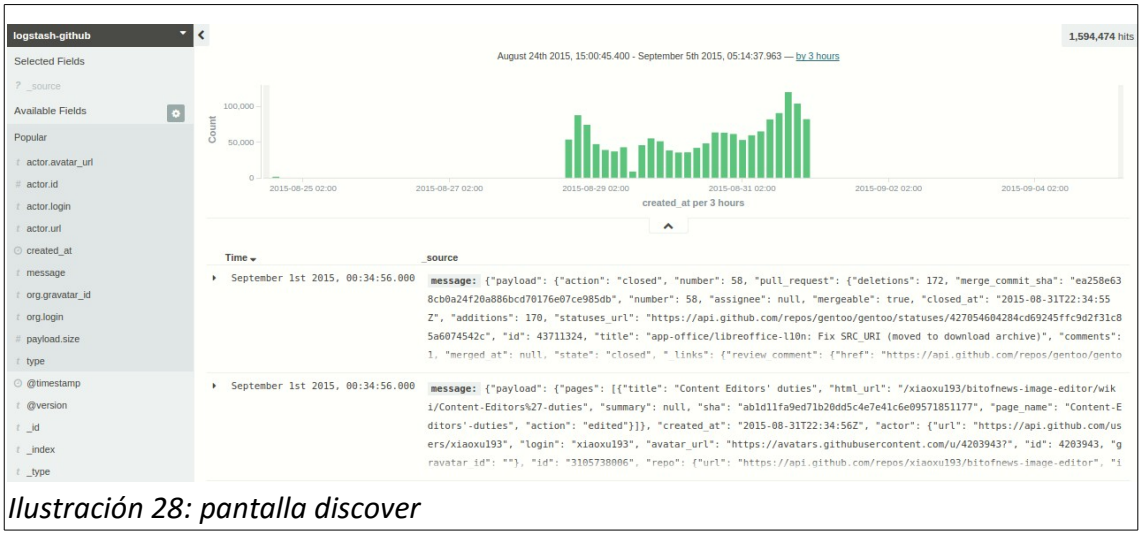


Ilustración 28: pantalla discover

En la ilustración 30 se presenta la pantalla que muestra Kibana una vez se ha seleccionado el indice sobre el que trabajar. A la izquierda se tienen los distintos campos de los eventos pertenecientes al indice. En el centro se representan cada uno de los eventos en crudo, es decir,

tal y como se han recogido y almacenado en EL. Pero lo más interesante se muestra arriba, y refleja un poco del potencial de Kibana. Aparece representado en un diagrama de barras como se distribuyen los eventos en los 3 días que duro la captura. Como se observa, el primer día (en el cual se empieza a cargar por la tarde) se recoge más que los dos intermedios para volver a subir el cuarto día (en el cual se carga hasta por la tarde). Sin saber que días de la semana son, sería lógico pensar que se trata de un fin de semana. Efectivamente, así es. El viernes comienza a descender el ritmo de eventos, que se mantiene bajo durante sábado y domingo, para volver a subir el domingo.

Una vez se ha elegido el indice sobre el que actuar, se pasa a la pantalla de *visualize*, donde se editan las representaciones. Como se ha especificado anteriormente, se trabaja sobre un único indice. Sin embargo, se trata de un indice con una alta cantidad de campos. Esto aporta mucha y variada información sobre el evento al que se refieren. Por otro lado, muchos de estos campo tratan información poco relevante para los propósitos requeridos, por lo tanto, elegir los eventos con información relevante resulta clave a la hora de generar los dashboards.

Una vez aclarado lo anterior, posibles campos de relevante interés son aquellos que aportan información el usuario, como el **actor.login**, **actor.name**, **actor.id** y sus mensajes, **actor.messages**. Los que aportan información sobre la empresa, como **org.id** u **org.name** también son de especial utilidad. Los que tratan sobre los repositorios, como el **repo.name** y el **repo.id**, permiten relacionar cada usuario con el repositorio en el que trabaja. Otros, que tratan sobre el propio evento en si, como el **timeStamp**, dan la posibilidad de crear histogramas, muy útiles para ver parámetros de trabajo y costumbres de los trabajadores a lo largo del tiempo de análisis. Cada uno de estos campos por separado no vale prácticamente nada, pero al relacionarlos entre si se obtiene una amplia gama de datos sobre los usuarios (repositorios, proyectos, empresa, mensajes) que pueden resultar más que útiles.

Ilustración 29: pie chart 8 repositorios con más actividad

Para una representación de los campos ya mencionados se accede a la ventana *visualize*, ya usada en el desarrollo anterior. Aquí se especifica el tipo de representación en la cual se quieren mostrar los campos, que se eligen más tarde. Ya se explicó el sistema de generación de

dashboards de Kibana en el desarrollo anterior, por lo tanto se saltará a algunas de las representaciones que más adelante se mostrarán en los dashboards. He aquí los ejemplos:

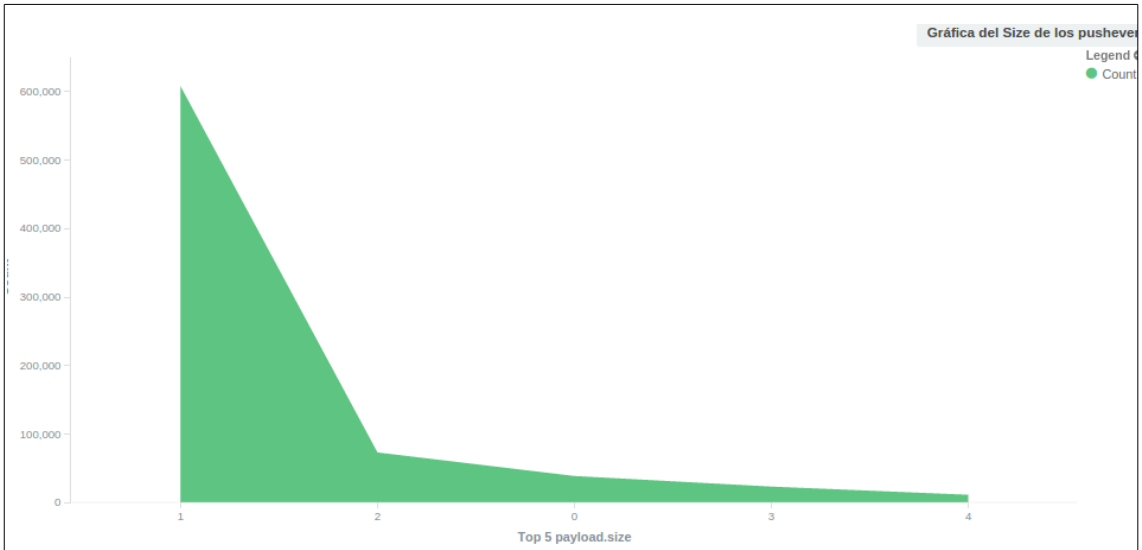


Ilustración 30: tamaño de los PushEvent

En la figura 30 se observa un pie chart cuyo circulo central se encuentra dividido en ocho secciones correspondientes a los ocho repositorio que más actividad tuvieron durante los cuatro días que duro la recogida de eventos. En este caso, además, se observa una variante con el resto de pie charts mostrados anteriormente: el anillo exterior del gráfico esta segmentado en relación a los usuarios que participaron en ese evento. Cada sección del circulo interior está a su vez dividida según el número de usuarios de cada uno de los repositorios.

La gráfica justo encima, en la figura 31 representa como se distribuyen los eventos del tipo *PushEvent* según el tamaño de cada uno de ellos.

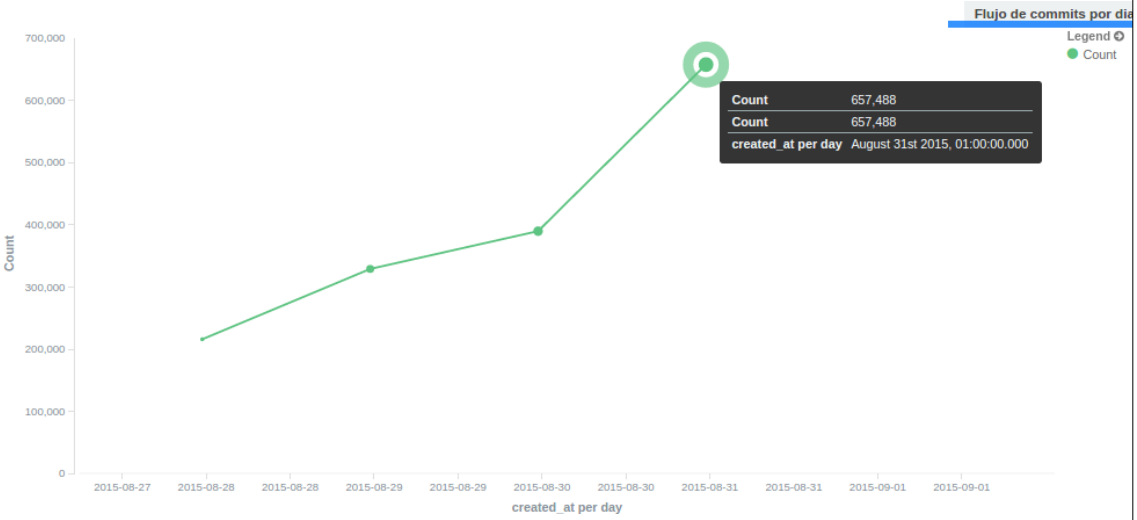


Ilustración 31: histograma de eventos

En esta ocasión se trata de un histograma que refleja la cantidad de eventos totales por día y como estas cantidades fueron variando a lo largo del tiempo que duró la captura de eventos. Se capturaron eventos desde las 4 de la tarde del viernes hasta las 4 de la tarde del lunes. De ahí que el viernes tenga menos cantidad de eventos que sábado y domingo. El lunes, sin embargo, se percibe una subida significativa de los eventos y, por lo tanto, de la actividad en GitHub, reflejando el comienzo de la semana laboral.

Las tres representaciones mostradas justo antes son solo una pequeña parte de lo que se muestra en los dashboards finales. Seria excesivo e innecesario explicar cada uno de ellos por separado, puesto que su verdadera función es la de trabajar en conjunto con otras representaciones agrupadas en varios dashboards. Cada visualización por separado ofrece poca o ninguna información relevante, por ese motivo se decidió explicar algunos como ejemplo para poder proceder a ver los dashboards finales generados.

Dashboards generados

Unos párrafos más arriba se han mostrado varios ejemplos de representaciones creadas con Kibana, pero estas aportan poco o ningún valor por si mismas. La información que conceden se ve significativamente incrementada cuando se representan junto a otras gráficas y demás tipos de visualizaciones relacionadas. Por ello, se han creado una serie de dashboards, que no son más que agrupaciones de distintas gráficas, diagramas y representaciones que abordan una finalidad común o relacionada. Estos dashboards permiten analizar la información recogida y almacenada en ES y son el final de la cadena que forma el sistema: **representación de la información** .

A continuación se muestran los dashboards generados:

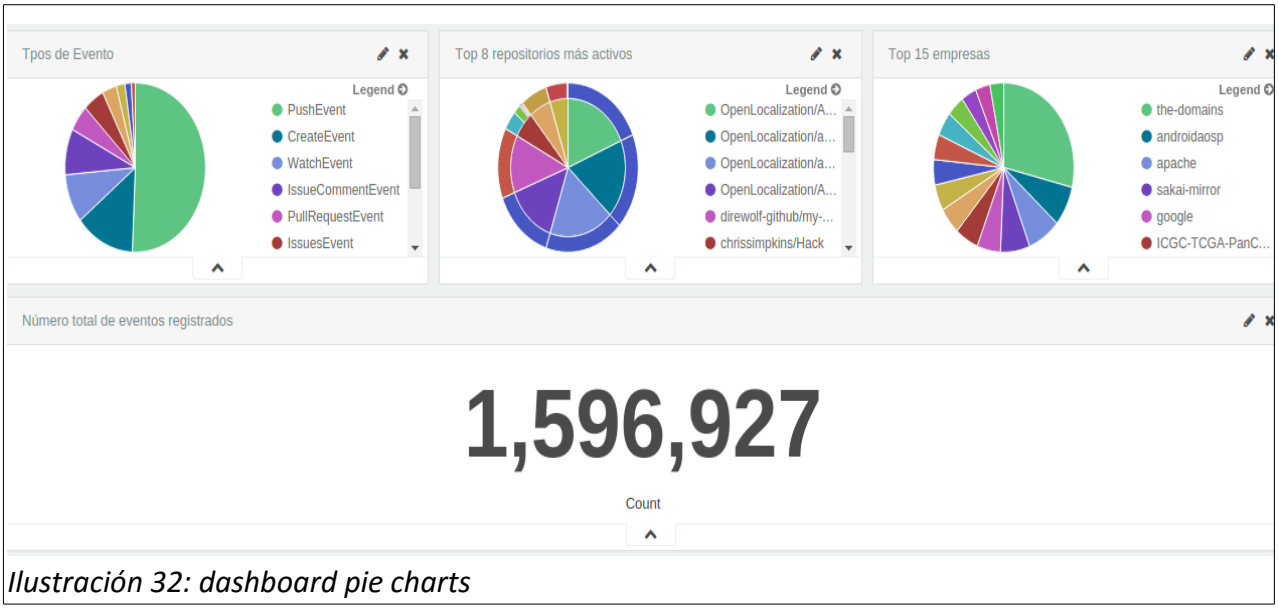


Ilustración 32: dashboard pie charts

Este primer dashboards consiste unicamente de *pie charts*, representando los tipos de evento, repositorios (ocho) y empresas (15) más abundantes entre todos los eventos recogidos. Además, se representa el número total de eventos en la parte inferior.

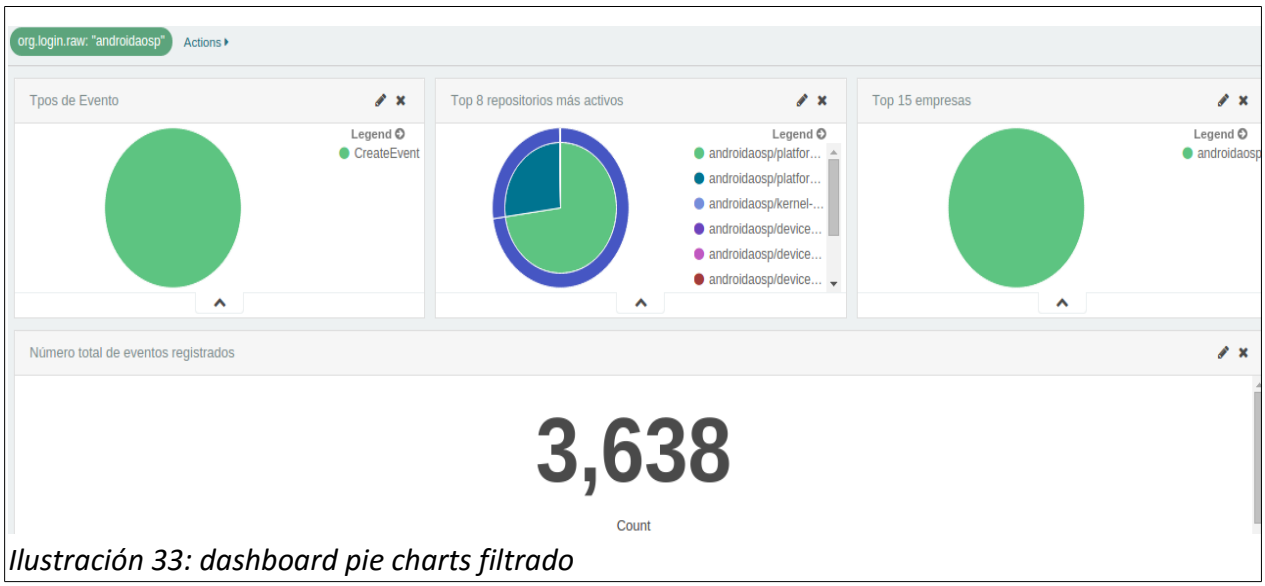


Ilustración 33: dashboard pie charts filtrado

Una propiedad de los Kibana que resulta muy útil en este dashboard es la del **filtrado** o **filtro.** Haciendo click en cualquiera de las secciones de uno de los pie charts, se aplicara el valor de dicho campo como filtro, actualizando los demás pie charts en función del mismo. Se entenderá mejor

esta funcionalidad con un ejemplo: Se selecciona la segunda empresa que mas eventos ha generado, AndroidAOSP. El resultado se muestra a en la figura 32.

Como se ve, cada una de los gráficos se ha actualizado en torno a ese campo y el valor del mismo. Ahora, el total de eventos de la parte inferior muestra el total de eventos unicamente de *AndroidAOSP*. De ese número total de eventos, 3638, todos son del tipo CreateEvent como se muestra en el pie chart situado a la derecha. El gráfico central muestra ahora los ocho repositorios que más actividad han tenido, siendo *platform-built* y *platform-abi-cpp* los dos más grandres.

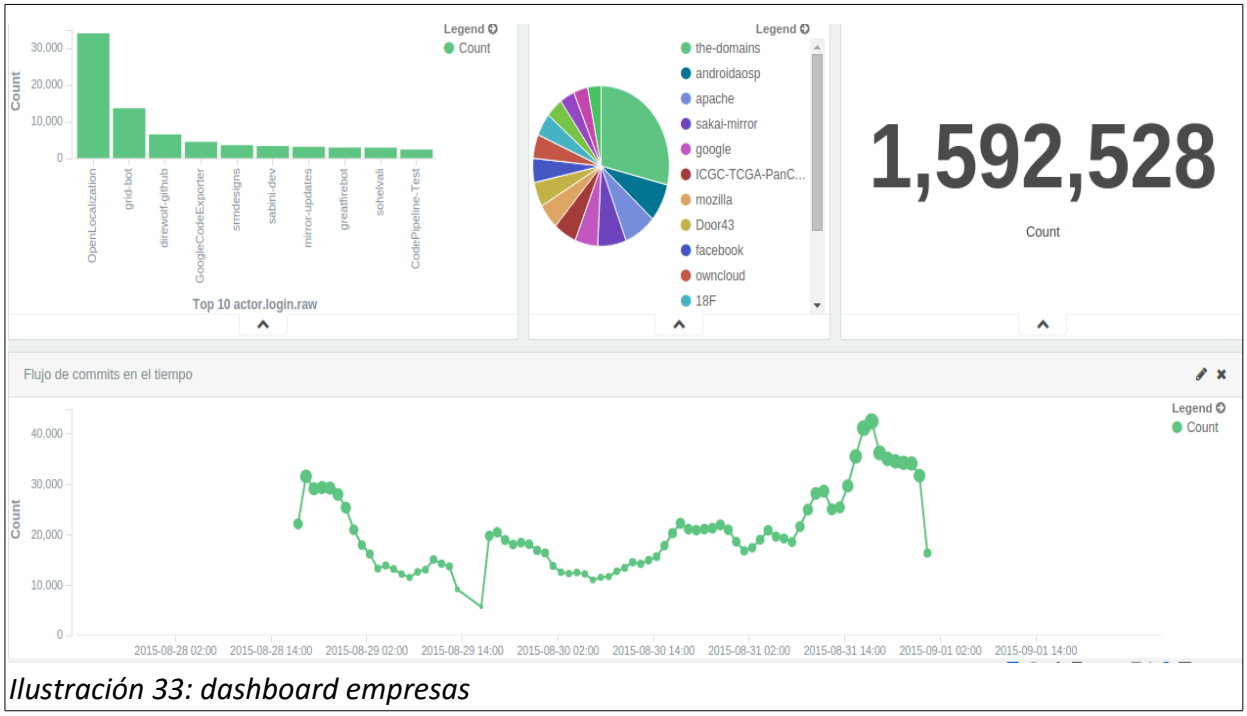


Ilustración 33: dashboard empresas

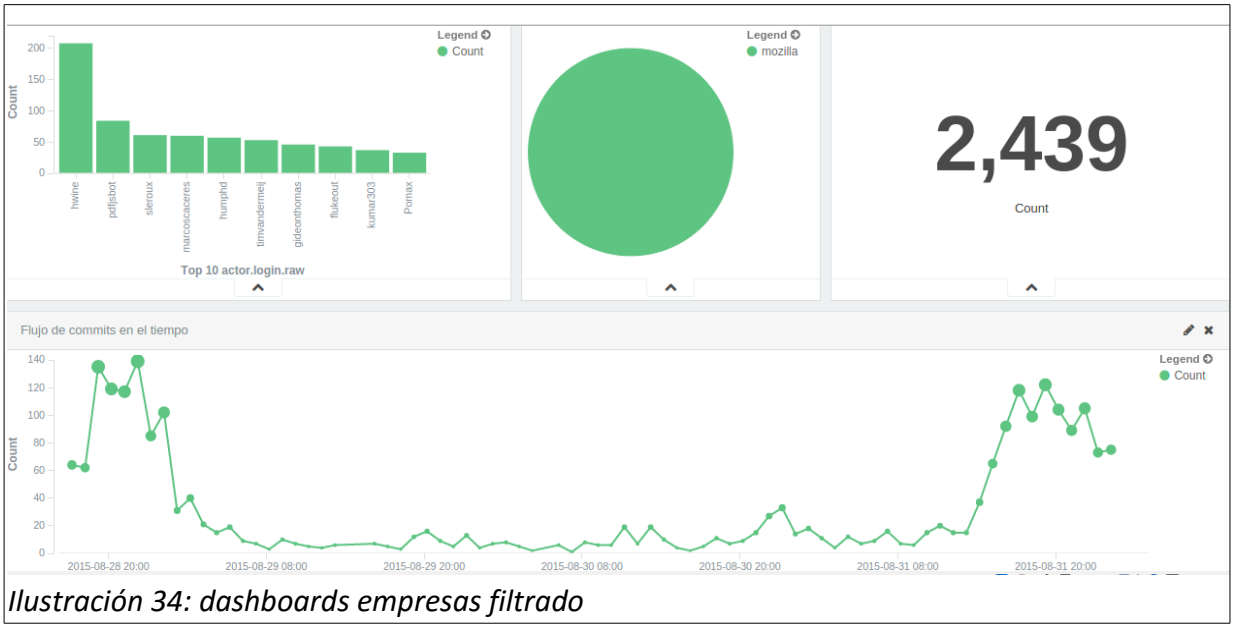


Ilustración 34: dashboards empresas filtrado

Este dashboard permite, de un vistazo, entender la situación en la que se encuentra en ese momento el sitio web, en lo que a actividad de empresas refiere. Además, ofrece una manera de tener controlados los repositorios que más actividad están reicibiendo por parte de los usuarios.

En la figura 33 se ha representado, junto con las empresas con más eventos (entre las que aparecen algunos monstruos del sector, como Google o Facebook) y el total de los mismos en el índice, los 10 usuarios que más actividad han generado. En la parte inferior del dashboard aparece un histograma que muestra la evolución temporal por hora. Como se observa, las horas con mayor de volumen de actividad coinciden con las del viernes (principio del histograma) y, sobre todo, con las del lunes (final del histograma). Las horas del fin de semana presentan un volumen mucho menor. Un objetivo de este dashboard seria el de permitir realizar un seguimiento de trabajo de un determinado sector de trabajadores, tanto en temas referentes a los días y el horario (se podrían sacar conclusiones sobre que en que horas y días los usuarios son productivos) como en términos de cantidad de actividad de los mismos.

Si bien esta visión general de los datos ya presenta información de utilidad, se aplica un filtro para poder extraer aún más información del dashboard y así constatar lo explicado en el párrafo anterior:

En la figura 34, Se ha seleccionado *Mozilla* dentro de las 15 empresas que aparecían en el pie chart. A la izquierda se muestran ahora los 10 usuarios con más actividad de Mozilla, así como el número total de eventos generados por esta empresa a la derecha. El histograma de la parte inferior también se actualiza, presentando ahora la evolución de la actividad relativa unicamente a Mozilla. Se observa que los usuarios de Mozilla trabajaron arduamente viernes y lunes, en cuyos días se distribuye la mayor cantidad de eventos. El sábado y domingo, como cabria esperar, la actividad es muy pequeña, prácticamente nula.

El último dashboard, ilustración 35, trata información sobre los **IssueCommentEvent**, eventos que se generan cuando un usuario pone un comentario en uno de los asuntos (*issue*) abiertos. El dashboard muestra a la izquierda un pie chart mostrando la proporción de asuntos o problemas abiertos frente a los cerrados en ese momento. También se observan dos tablas de datos con información útil que proporcionan los eventos IssueCommentEvent: la primera muestra el id del asunto, así como su título, url (para acceder al evento en crudo si fuera necesario) y el estado del mismo. El id daría la posibilidad de rápidamente encontrar el asunto en cuestión. La segunda tabla, que se encuentra un poco más abajo, muestra el usuario que generó el evento, así como la organización a la que pertenece y el cuerpo del mensaje. La barra de búsqueda se puede usar para encontrar mensajes por alguna palabra que se considere clave.

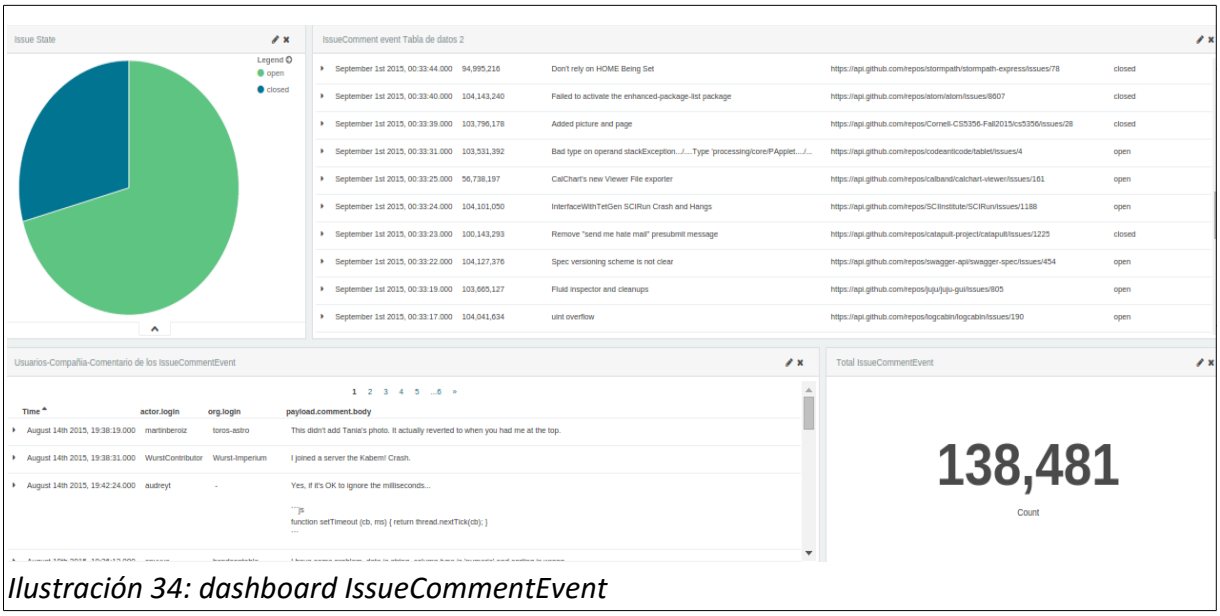


Ilustración 34: dashboard IssueCommentEvent

El objetivo del dashboard no es otro que el de proporcionar una manera rápida y sencilla de realizar un seguimiento sobre aquellas ***issues*** que puedan ser de interés, ya sean cerradas o abiertas.

Usuarios de Github

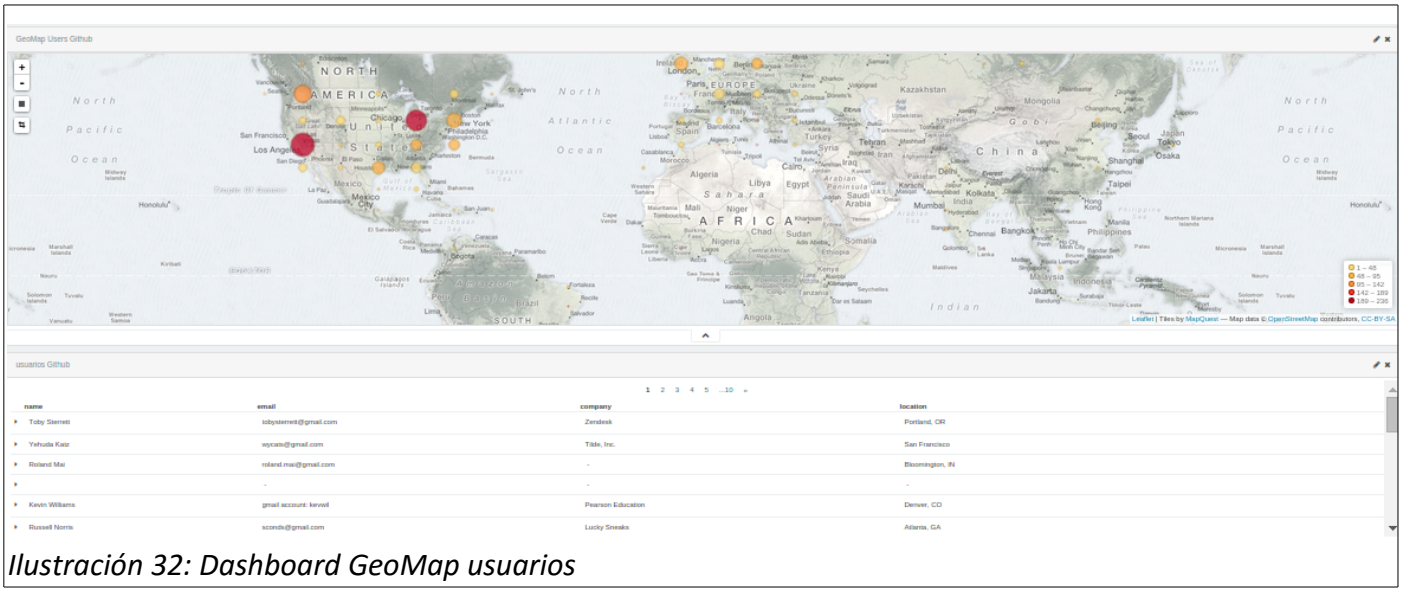


Ilustración 32: Dashboard GeoMap usuarios

Este dashboard consiste en un **geoMap** en el que se presentan, mediante focos de calor, las localizaciones de los usuarios de GitHub. Debajo aparece una tabla reflejando la información más importante de cada usuario, como son su nombre, email, empresa en la que trabaja y localización (nombre de la ciudad y país). Como se observa, el principal foco de usuarios es Estados Unidos, seguido de Europa. Algo totalmente normal teniendo en cuenta que entre ambas zonas geográficas concentran gran parte de la población desarrollada tecnológicamente.

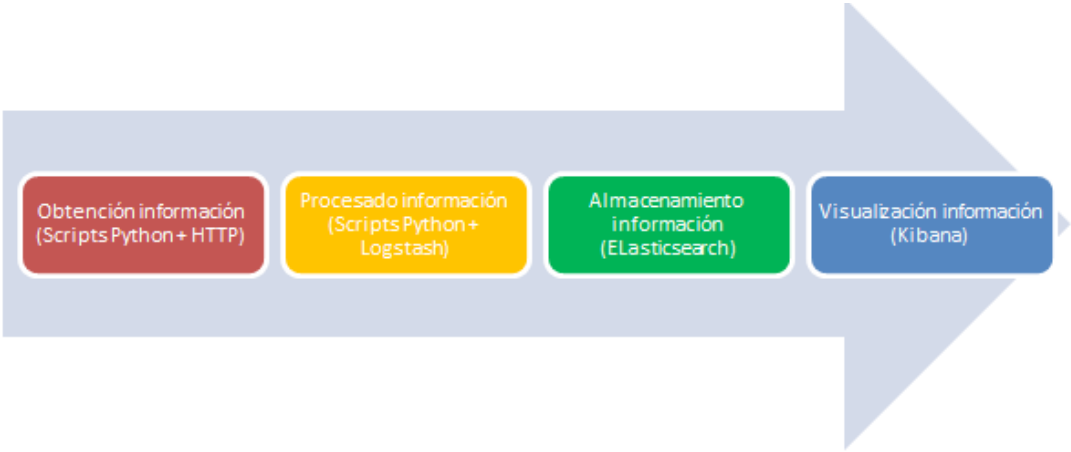
Se ha decidido representar esta información aparte puesto que no se obtiene de los eventos generados por los usuarios en tiempo real como el resto de dashboards, si bien si que se obtienen mediante la API de GitHub. Se trata de otro tipo de eventos, los eventos de usuario, que no son generados en un momento concreto de tiempo. Estos contienen información relevante sobre cada uno de los usuarios registrados en Github, como la representada arriba, además de otros campos como el/los repositorios, biografía, tipo de usuario, fecha de registro, etc.

Como se ha podido comprobar, los dashboards resultan extremadamente útiles a la hora de analizar distintos tipos de datos, así como distintos aspectos de los mismos. Agrupan en ellos varias representaciones que a su vez aúnan diversa información de una manera visual, que permite entender fácilmente los datos reflejados y el contexto en el que se encuentran. Los eventos capturados, debido a su cantidad, tamaño y naturaleza, no tendrían gran significado sin

las diferentes maneras que ofrece Kibana para lanzar búsquedas sobre ellos, tratarlos y visualizarlos.

La visualización de los dashboards en el navegador web supone el final de la cadena que forma el sistema expuesto en esta memoria.

CAPÍTULO 5: ANÁLISIS Y CONCLUSIONES



Finalmente, se ha demostrado que la herramienta desarrollada cumple con los requisitos especificados para este proyecto. Se trata de una herramienta versátil, con tecnologías en pleno crecimiento y desarrollo y que, con las mejoras pertinentes, podría resolver verdaderas necesidades en el mercado real.

El sistema ha sido capaz de capturar e indexar todos los eventos que la fuente de datos propuesta, la API GitHub, generaba en tiempo real, mostrándolos a su vez en los dashboards generados en Kibana. Esto permite un control sobre la actividad que ocurre en GitHub desde distintas perspectivas, como son los propios eventos generados, o los usuarios y empresas trabajando con GitHub.

Todos los dashboards mostrados corresponden a datos reales capturados durante dos periodos de tres días. Se estipulo este periodo de tiempo ya que era suficiente para obtener un número de eventos significativo que permitiera reflejar el funcionamiento y finalidad de la herramienta. En total, se han capturado un total de **3.432.355** eventos, repartidos en los dos periodos comentados más arriba. Esta cantidad de eventos suma **15,6 GB** de información en crudo junto con todos los indices de Kibana, también almacenados en Elasticsearch.

Sin embargo, el periodo de captura o recogida de eventos puede ser indefinido. Con el script de captura de eventos en funcionamiento, los eventos se cargan en en el indice correspondiente al tiempo en el que se generan. Cada uno de los diagramas, tablas, gráficos y demás representaciones presentes en los dashboard se actualizan automáticamente según el tiempo que se defina. Esto permite llevar un control en tiempo real de lo que ocurre en GitHub, tanto a nivel general como centrándose en un apartado en concreto (repositorios, usuarios, empresas, proyecto,...). Por todo esto, la herramienta podría servirle tanto a un administrador de GitHub que

necesita realizar un seguimiento de los eventos generados como a una de las muchas empresas que quisiera, por ejemplo, controlar el trabajo fuera del horario laboral.

Aunque no se utiliza en este proyecto, ES cuenta con una propiedad, ***multi-threading***, que consiste en hacer funcionar el nodo de Elasticsearch con procesos multi-hilo, es decir, cuando una tarea es recibida y encolada, se le asigna un hilo para que empiece a trabajar con ella, si los hilos están ocupados con diferentes tareas, las nuevas tareas se siguen encolando hasta que hilos libres puedan empezar a trabajar con ellas. Elasticsearch tiene varios grupos de hilos (thread Pools) para configurar en base al consumo de memoria dentro de un nodo. Muchas de esos grupos de hilos tienen asociadas colas de tareas. Es importante asignar el número de hilos / límites de las colas acorde a las necesidades y recursos disponibles. El número de hilos nunca debe ser superior al número de núcleos del procesador. . Esto aumenta en consideración la velocidad de indexado de cada nodo, ya que cada hilo cuenta con un *buffer* escritor independiente. El multi-threading permitiría que el sistema desarrollado en el presente proyecto fuera una solución optima para casos en los que la(s) fuente(s) de información generan más datos o eventos que la API de GitHub, usada en este proyecto.

5.1 LECCIONES APRENDIDAS

Durante el desarrollo del presente proyecto, fueron muchos los conocimientos adquiridos, resultando en una experiencia de lo más formativa y enriquecedora. Aquí se mencionan algunos de ellos:

- El uso del stack ELK ha permitido conocer y explotar tecnologías muy innovadoras dentro del procesamiento, almacenamiento y visualización de información, como son Logstash, Elasticsearch y Kibana. Se ha comprobado que juntas forman una poderosa herramienta.
- Así mismo, el propio uso de las tecnologías anteriormente mencionadas ha resultado en una profundización y una mayor apreciación del software libre. Se ha descubierto la potencia del mismo y lo relevante que resulta en el mundo de la tecnología.
- La exploración del mundo Big Data ha permitido un mayor conocimiento de tecnologías de este ámbito de muy diversas prestaciones y usos.
- Un mejor uso y comprensión del lenguaje Python. Los distintos scripts en Python de procesado y recogida de eventos han exigido la adquisición de un mayor conocimiento de este magnifico lenguaje, en especial de librerías para el procesado de JSON y de uso del protocolo HTTP.
- Adquisición de conocimientos sobre el formato JSON a través de usos reales del mismo. Es le formato usado en los fichero procesados y eventos capturados.

- Se han adquirido también amplios conocimientos de la API de GitHub, junto con los distintos tipos de eventos que la misma genera. Esto permite un mayor entendimiento de GitHub así como de las APIs públicas.

5.2 CONOCIMIENTOS APLICADOS

Para el satisfactorio desarrollo del proyecto se han aplicado multitud de conocimientos adquiridos durante el grado en ISAM, mayormente relacionados con Python y la programación.

Otro conocimientos adquiridos en la carrera que han resultado de utilidad son los relacionados con el protocolo HTTP y el formato de datos JSON.

Pero, tal vez, la cualidad adquirida y aplicada más importante ha sido la capacidad de solventar de manera autónoma los diversos problemas (que han sido muchos y de diversos tipos) que han ido apareciendo a lo largo de este trabajo.

5.3 TRABAJOS FUTUROS

Una vez completados los objetivos propuestos al principio de esta memoria, y terminando así el desarrollo del proyecto, surgen multitud de nuevos escenarios y mejoras para hacer la herramienta aun mas versátil y aplicable al mercado real.

Una de las principales mejoras seria el uso de la propiedad multi hilo, tanto de Logstash como de Elasticsearch, la cual permitiría un indexado mucho mas eficiente en caso de mayores volumenes de información.

Otra manera de incrementar la eficiencia del sistema viene con los potentes filtros que posee Logstash. Si bien es verdad que no se saca mucho provecho de los mismos en el sistema resultante (debido a que el formato de los datos ya es JSON y eso hace que sean procesables directamente), estos permiten que el procesamiento de los datos sea mucho más rápido, cómodo y configurable.

En cuanto a nuevos escenarios, cabe decir que para este trabajo se ha usado la API de GitHub como fuente de información, por varios motivos: Es pública, gratuita, ofrece eventos reales en tiempo real, etc. Pero, tal vez, lo más importante es el hecho de que ofrece los eventos en formato JSON, lo cual permite que sean indexados en ES con muy poco o ningún procesado. Es por ese motivo por el cual los eventos pueden ser cargados directamente mediante HTTP sin necesidad de usar Logstash.

Dicho todo lo anterior, otros dos sitos web con API generadoras de eventos , con características similares, son Facebook (https://developers.facebook.com/docs/graph-API) y Twitter (https://dev.twitter.com/overview/API), ambas en formato JSON. El sistema descrito en esta memoria puede trabajar perfectamente con estas dos API. Bastaría con añadir pequeñas modificaciones en el script para ajustar la petición GET a la url de la nueva API según el formato de la misma. Tras eso, ya se podrían empezar a indexar eventos en ES y por lo tanto, a visualizarlos en Kibana.

Pero el sistema desarrollado no solo procesa información en formato JSON. Se puede adaptar también al otro "gran" formato de datos, XML. La principal diferencia es que al no tratarse de datos en JSON, habría que recurrir a Logstash, en lugar de HTTP directamente, para llevar la información a ES. Esto es así por que ES indexa documentos en JSON, por lo tanto, para XML se necesitaría un procesado anterior al proceso de carga y Logstash tiene un filtro especifico para este formato. Dicho lo anterior, el sistema es totalmente adaptable a cualquier fuente de información en otros formatos.

Otro interesante uso, muy aplicable a la realidad, es el tratamiento de ficheros *Log*. Este tema es un quebradero de cabeza en multitud de empresas, sobre todo aquellas en las que un amplio número de herramientas y sistemas escriben en un o varios ficheros Log. Estos ficheros reciben, en múltiples ocasiones, cientos e incluso miles de trazas (eventos) por segundo, y su mantenimiento y monitorización puede llegar a complicarse muchísimo. Es aquí donde entra en juego la herramienta descrita en este trabajo. Las propiedades multi-hilo de Logstash y ES permitirían un procesado e indexado muy eficiente de los log, incluidos aquellos con miles de eventos por segundo. Kibana ofrecería una visualización en tiempo real, lo que permitiría un fácil control de eventos de *warning* y error, localizando el momento en el que ocurrió, el usuario que lo provocó... , de manera similar a la que se ha mostrado con los dashboards generados del punto 4.

Logstash ofrece un filtro de especial interés, el cual permite configurar una cuenta de usuario de Twitter como fuente de entrada. Esto hace posible indexar cada evento (twitt) a esa cuenta como un documento JSON en ES. Por supuesto, los dashboards de Kibana servirían para controlar toda la actividad de dicha cuenta, filtrando por asuntos de interés o *hashtags.* Un propuesta especialmente interesante para, por ejemplo, el *community manager* de un gran empresa, el cual recibe cientos de twitts diarios sobre múltiples asuntos.

Como se ha demostrado, el sistema propuesto y desarrollado en esta memoria ofrece una herramienta versátil que no solo ha cumplido los objetivos propuestos si no que tiene múltiples casos de uso en el mercado real.

BIBLIOGRAFÍA

[1] Referencia teórica, http://es.wikipedia.org

[2] Big Data, https://www.ibm.com/developerworks/ssa/local/im/que-es-big-data/

[3] GitHub, https://GitHub.com/

[4] GitHub archive, https://www.GitHubarchive.org/

[5] GHTorrent, http://ghtorrent.org/ .

[6] API de GitHub, https://developer.GitHub.com/v3/

[7] Logstash documentation, https://www.elastic.co/guide/en/Logstash/2.2/index.html

[8] Elasticsearch documentation, https://www.elastic.co/guide/index.html

[9] Kibana documentation, https://www.elastic.co/guide/en/kibana/current/index.html

[10] Python, https://docs.python.org/2/faq/

[11] JSON, http://www.JSON.org/

[12] HTTP, https://www.w3.org/Protocols/

[13] API *de Facebook,* https://developers.facebook.com/docs/graph-API)

[14] API de Twitter , https://dev.twitter.com/overview/API