

UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR



TRABAJO FIN DE MÁSTER

Sistema de monitorización en tiempo real de los servicios digitales de la Universidad Autónoma de Madrid

Máster Universitario en Ingeniería Informática

Autor: CORONADO LÓPEZ, Abel

Tutor: PULIDO CAÑABATE, Estrella

Departamento de Ingeniería Informática

FECHA: Diciembre, 2020

Agradecimientos

Mi mayor agradecimiento a mi tutora que me ha guiado y ayudado en la elaboración del proyecto y a mi familia por el apoyo durante la realización del mismo.

Índice

Introducción	11
1.1. Alcance	11
1.2. Estructura	12
Estado del arte	13
2.1. Herramientas de monitorización	13
2.1.2. Acronis Monitoring Service	14
2.1.2. New Relic	15
Análisis y planteamiento	17
3.1. Objetivos	17
3.2. Fases y herramientas	17
3.2.1. ETLs	17
3.2.2. Encolamiento	18
3.2.3. Procesamiento en tiempo real	19
3.2.4. Persistencia	20
3.2.5. Visualización	22
3.2.6. Herramientas	22
Diseño	23
4.1. Arquitectura	23
4.1.1. fluentd	25
4.1.2. Apache Kafka	26
4.1.3. Apache Spark Streaming	26
4.1.4. InfluxDB	27
4.1.5. Grafana	28
Desarrollo	29
5.1. ETL	29
5.1.1. Instalación del servidor	29
5.1.2. Herramienta de carga	31
5.1.3. fluentd	31
5.2. Encolamiento	35
5.2.1. Apache Kafka	35
5.3. Procesamiento	38
5.3.1. Spark Streaming	38
5.4. Persistencia	44
5.4.1. InfluxDB	44
5.5. Visualización	46
5.5.1. Grafana	46

Resultados	53
Conclusiones	57
Trabajo futuro	59
Bibliografía	61

Índice de figuras

<i>Figura 2.1 - Acronis Dashboard - acronis.com</i>	15
<i>Figura 2.2 - New Relic Dashboard - newrelic.com</i>	16
<i>Figura 3.1 - ETL - Instituto Nacional de Ciencia de datos</i>	18
<i>Figura 3.2 - Cola - wikipedia.org</i>	18
<i>Figura 3.3 - Streaming - indizen.com</i>	19
<i>Figura 3.4 - Relacional vs No Relacional - pragma.com</i>	21
<i>Figura 4.1 - Fases arquitectura</i>	23
<i>Figura 4.2 - Arquitectura</i>	24
<i>Figura 4.3 - Configuración fluentd</i>	25
<i>Figura 4.4 - Cola Kafka - jtech.ua.es</i>	26
<i>Figura 4.5 - Cuadro de mando Grafana - cdmconsultores.com</i>	28
<i>Figura 5.1 - Apache Server</i>	30
<i>Figura 5.2 - Formato JSON</i>	32
<i>Figura 5.3 - Productor-Consumidor Kafka - datastax.com</i>	37
<i>Figura 5.4 - Spark WordCount - databricks.com</i>	40
<i>Figura 5.5 - Influx Shell</i>	45
<i>Figura 5.6 - Influx data</i>	46
<i>Figura 5.7 - Grafana login</i>	47
<i>Figura 5.8 - Nuevo usuario</i>	48
<i>Figura 5.9 - Crear datasource 1</i>	49
<i>Figura 5.10 - Crear datasource 2</i>	50
<i>Figura 5.11 - Crear dashboard</i>	51
<i>Figura 6.1 - Ejemplo de cuadro de mando</i>	53
<i>Figura 6.2 - Códigos 300</i>	54
<i>Figura 6.3 - Thresholds personalizables</i>	54

<i>Figura 6.4 - Contador de códigos</i>	55
<i>Figura 6.5 - Mapa de calor</i>	55
<i>Figura 6.6 - Códigos 200 por host</i>	56

Abstract

This project seeks to centralize and provide a complete real-time view of the digital services status at the Autonomous University of Madrid. This allows it to offer a more agile and efficient way to operate in any uncontrolled situation: server overload, possible software failures, erroneous requests, etc.

As a public service, efforts have been made to ensure that the cost of development and implementation is zero. This project has been carried out entirely with Open Source technologies or without cost of use.

Different Big Data technologies have been researched and analyzed in each part of the project in order to offer a better experience, maintenance and performance of the ecosystem that has been built.

Resumen

Este proyecto busca centralizar y dar una visión completa en tiempo real del estado de los servicios digitales de la Universidad Autónoma de Madrid. De esta manera, se ofrece una manera más ágil y eficaz para operar cualquier situación no controlada: sobrecarga del servidor, posibles fallos del software, peticiones erróneas, etc.

Al ser un servicio público, se ha procurado que el coste del desarrollo y la implantación sea cero. Este proyecto se ha realizado enteramente con tecnologías Open Source o sin coste de uso.

Se han investigado y analizado diferentes tecnologías Big Data en cada una de las partes del proyecto para, de esta manera, poder ofrecer una mejor experiencia, mantenibilidad y rendimiento del ecosistema que se ha construido.

1. Introducción

En este documento de trabajo de fin de máster, se va a exponer la motivación, desarrollo y conclusiones a las que se han llegado durante la realización del proyecto.

La motivación de este proyecto viene dada por poder ofrecer una visión general de los servicios digitales de la Universidad Autónoma de Madrid en tiempo real. Facilitar la operativa de estos servicios y poder anticiparse al error antes de que ocurra.

También, este proyecto ha sido impulsado por mis ganas de conocer el mundo Big Data y poder “jugar” con las herramientas más punteras de procesamiento de datos, que usan hoy en día la mayoría de las grandes empresas.

Se ha hecho un estudio de mercado de las posibles herramientas Big Data a utilizar en este proyecto. Dado que es un proyecto con numerosas “piezas”, sobre todo se ha de hacer hincapié en la retrocompatibilidad entre cada una de ellas y que el tiempo de respuesta sea lo más rápido posible, dado que esa es la finalidad del proyecto. Una de las cualidades buscadas, también, es que sea fácilmente mantenible y con una interfaz final amigable para el usuario objetivo.

Una vez elegidas dichas herramientas, se ha procedido al estudio más exhaustivo de estas para un posterior correcto desarrollo.

1.1. Alcance

El alcance de este proyecto ha sido intentar satisfacer las necesidades anteriormente descritas a través de una plataforma Big Data en tiempo real.

El sistema procesa y envía trazas de log de los distintos servidores y se muestran de forma amigable para el usuario final en forma de gráficos que él mismo podrá manipular y/o crear con total libertad. Así, cada usuario podrá tener sus propias visualizaciones personalizadas dependiendo del rol que asuman en la administración de los servidores. Esta flexibilidad de personalizar los gráficos por los propios usuarios, es uno de los objetivos principales del proyecto para dar libertad y no estar sujeto a terceras partes para que los desarrollen.

La operación por parte de los administradores en caso de incidencia será mucho más cómoda y rápida ya que, estas visualizaciones, permiten saber dónde, cómo, cuándo y qué está fallando. En algunos casos, se podrá preveer que algo fallará debido a la tendencia de las gráficas y la experiencia pasada de los propios usuarios por lo que, además, permite anticiparse al problema antes de que suceda.

1.2. Estructura

La memoria de este proyecto comienza con la introducción y alcance, donde se detallan los objetivos y desarrollos realizados a muy alto nivel para conseguir los objetivos marcados.

A continuación, el estado del arte en todo lo referente a este proyecto, donde se detallan las tecnologías y ecosistemas que resuelven este problema actualmente.

Después, se expone la solución técnica a grandes rasgos del proyecto: arquitectura, herramientas, conexiones, etc.

Una vez se conozca el diseño de la solución, se entrará más en detalle en cada uno de los módulos que dan solución al proyecto, explicando cómo se ha desarrollado y el por qué.

Con toda la solución técnica expuesta, se procederá a mostrar los resultados obtenidos y si se ha conseguido llegar al objetivo anteriormente descrito.

Para finalizar, se detallarán las conclusiones obtenidas durante la realización del trabajo, así como la experiencia personal. También se abordarán puntos para una continuación y mejora futura del proyecto.

2.Estado del arte

El estado actual en el que nos encontramos se basa en los datos. La información sustenta todos los procesos y servicios que usamos a diario: Netflix, Instagram, Google, Spotify... Cuantos más datos se posean, mejor será el servicio que se ofrezca y mayor será el beneficio que se obtenga.

No sólo se mejora el servicio y herramientas, sino que actualmente las empresas utilizan estos datos procesados para tomar decisiones estratégicas y planes de futuro.

Netflix, por ejemplo, recopila información de las series y películas de sus usuarios para poder, gracias a algoritmos de Machine Learning, ofrecer recomendaciones y sugerencias mucho más personales a cada uno.

A su vez, conoce los gustos y preferencias de las personas y puede tomar decisiones como, por ejemplo, dónde invertir en la siguiente serie. Y esto es porque posee toda la información necesaria como para decidir si una película de acción protagonizada por un actor concreto va a ser mejor acogida que otra de otro género.

Poder predecir qué pasará, es la mejor baza para los movimientos y operaciones empresariales.

Por esto mismo, las empresas están priorizando y dando una mayor importancia al procesamiento de grandes cantidades de datos porque se convierten en beneficios directos en la empresa.

Debido a esto, la motivación del proyecto es ofrecer un sistema donde se centralizan los datos de los servicios digitales de la universidad para, así, facilitar la operativa y prever cualquier tipo de fallo. Esto, como se explicó anteriormente, se traduce en ahorro de costes y un mejor servicio a los usuarios finales.

Existen diversas herramientas de monitorización en el mercado que ofrecen este servicio en cloud pero son muy caras y poco flexibles. A continuación se mostrarán unos ejemplos de las herramientas más usadas en estos ámbitos.

2.1. Herramientas de monitorización

Como se ha explicado anteriormente, hay empresas que ofrecen servicios cloud de monitorización de servidores. La gran mayoría se basa en la instalación de un agente en cada uno de los servidores a monitorizar. Este agente enviará la información recopilada a través de internet al cloud de la empresa donde se procesan y almacenan. Estas herramientas proporcionan un servicio de visualización donde los usuarios finales podrán consumir esos datos.

Estos servicios son cloud por lo que ahorra costes de mantenimiento pero crea una dependencia de terceros y poca flexibilidad a la hora de consumir los datos.

Existe una gran oferta de este tipo de servicios, de los cuales detallaremos dos de los más importantes del sector.

2.1.2. Acronis Monitoring Service

Acronis es una empresa desarrolladora de software de copias de seguridad, recuperación de datos (restauración y backups) y accesos seguros para consumidores dirigido a todo tipo de empresas. También desarrolla software para la virtualización, migración y conversión [1].

Acronis provee un servicio cloud llamado Acronis Monitoring Service, el cual suministra al usuario final un dashboard con los datos de los servidores *on-premise* que se quieren monitorizar.

No es uno de los servicios más caros dado que ofrecen una solución básica de monitorización no tan exhaustiva.

También ofrecen un servicio de *backups* en la nube para realizar copias de seguridad eventuales con replicación en caso de desastre.

En la figura 2.1, se muestra un panel de control de la monitorización realizada por Acronis.

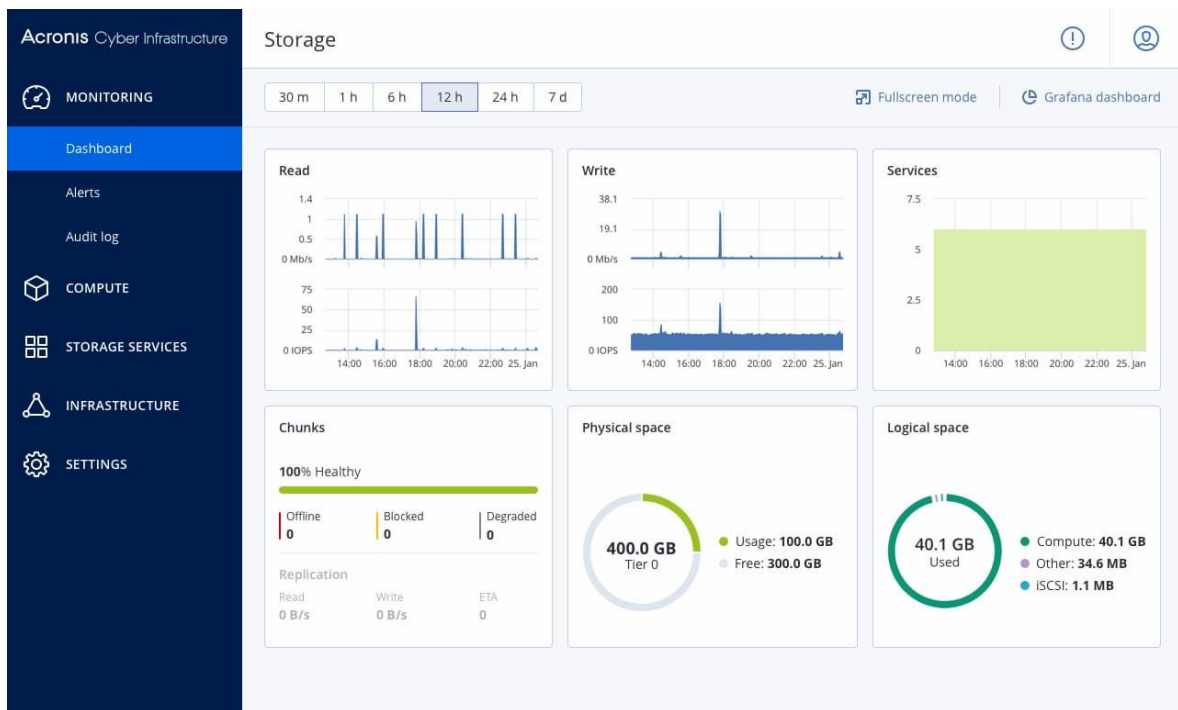


Figura 2.1 - Acronis Dashboard - acronis.com

2.1.2. New Relic

New Relic es una empresa tecnológica creada para ayudar a los ingenieros a crear un software más perfecto. Su software está basado en la nube con el objetivo de realizar un seguimiento del rendimiento de los servicios implantados [2].

En este caso se hablará de su producto de monitorización en tiempo real que procesa y muestra los datos de monitorización recopilados.

A diferencia de Acronis, esta herramienta es mucho más exhaustiva. Proporciona una plataforma de análisis de datos en la nube, monitorización de servicios, etc. En concreto, el producto de seguimiento, realiza la monitorización de servidor, aplicaciones y experiencia de usuario final al igual que el análisis de los datos monitorizados [3].

En la figura 2.2, un ejemplo de dashboard de New Relic con los datos de monitorización recopilados.

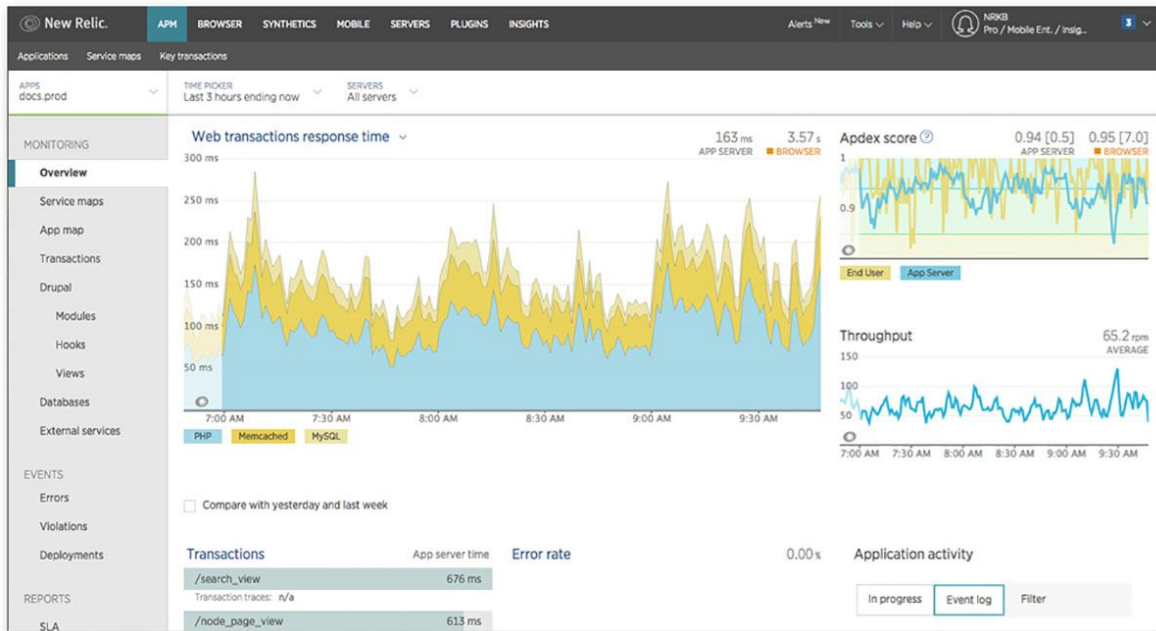


Figura 2.2 - New Relic Dashboard - newrelic.com

Al ser un servicio mucho más completo que el anterior, obviamente es mucho más caro y complejo de configurar. Se requieren conocimientos más avanzados ya que la curva de aprendizaje de esta herramienta es más pronunciada.

3. Análisis y planteamiento

En este capítulo se describirán los objetivos y las funcionalidades necesarias que el proyecto ha de cumplir.

3.1. Objetivos

El objetivo principal de este trabajo es poder proporcionar una herramienta para la operación de los servicios digitales que proporciona la Universidad Autónoma de Madrid. Será un modelo real, con datos reales, pero no desplegado en los servidores por temas de autorización e intrusismo.

Para ello, se simulará un servidor y se desarrollarán las cargas de datos para que, en un futuro, si se decide desplegar la solución, sea lo más liviano posible.

Los datos serán enviados en tiempo real a un sistema de colas donde un módulo de procesamiento streaming los procesará y persistirá en una base de datos. Por último, un visualizador, consumirá los datos y los presentará a los usuarios finales de forma amigable e intuitiva.

3.2. Fases y herramientas

Para poder decidir las herramientas y poder realizar una buena investigación, se deben dejar claras todas las fases del proyecto.

3.2.1. ETLs

Extract, Transform and Load (“extraer, transformar y cargar”) es el proceso por el cual se mueven datos desde múltiples fuentes, se formatean, limpian, y se cargan en otra base de datos (*data lake* o *data warehouse*) para su posterior análisis [4].

En este proyecto, la fase ETL hace referencia a la extracción de los datos desde el fichero de log que genera el servidor. Después, transformará esa información cruda en un formato concreto que facilite su posterior procesamiento y la cargará en el siguiente módulo de encolamiento.

Previamente, al tener que simular los datos del servidor, se desarrollará un pequeño módulo que genere trazas de log sobre un fichero ficticio. El módulo de ETL se valdrá de este fichero autogenerado para el procedimiento explicado anteriormente.

En la figura 3.1 se muestra una síntesis del proceso de ETL.



Figura 3.1 - ETL - Instituto Nacional de Ciencia de datos

3.2.2. Encolamiento

Desde el punto de vista informático, una cola es una estructura de datos en la que se mantiene el orden. Se insertan datos por un extremo (*push*) y se extraen por el otro (*pull*).

En la figura 3.2, se muestra un ejemplo de este tipo de estructura, también conocida técnicamente como cola FIFO (del inglés *First In First Out*) debido a que el primer elemento en entrar será también el primer elemento en salir, manteniendo así el orden [5].

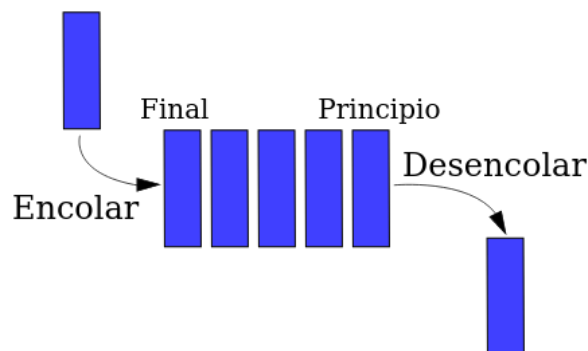


Figura 3.2 - Cola - wikipedia.org

En Big Data se usan estas estructuras para almacenar los datos para su posterior procesamiento. Este almacenaje es volátil, ya que es un medio de persistencia auxiliar que apoya todo el flujo de los datos. Los datos permanecerán en las colas hasta que un proceso

externo las consume y las procesa. A día de hoy, existen herramientas de encolado muy avanzadas que proporcionan replicación de datos ante desastres, procesamiento distribuido, canales distintos donde categorizar los datos, persistencia durante un amplio periodo de tiempo, etc.

En esta fase, se encolan los datos generados por la fase previa de ETL. Los datos, en principio, deberán estar poco tiempo en las colas debido a que, al ser un proceso streaming, se consumirá rápido por el siguiente proceso. El proyecto, además, deberá permitir reprocesos de datos antiguos y que no haya pérdida ante un fallo del proceso.

3.2.3. Procesamiento en tiempo real

El procesamiento en tiempo real es una técnica de procesamiento y análisis de datos por el cual, a través de la implementación de un modelo de flujo de datos en el que los datos asociados a series de tiempo (eventos) fluyen continuamente a través de un *pipeline* de entidades de transformación que componen el sistema [6].

En este tipo de procesamientos, como se muestra en la figura 3.3, se suelen usar ventanas de tiempo para procesar pequeños lotes de datos (*micro batches*). De esta manera se agregan los datos y se genera un *pseudo streaming* con un *delay* de tiempo muy bajo, dependiendo de la duración de esas ventanas.

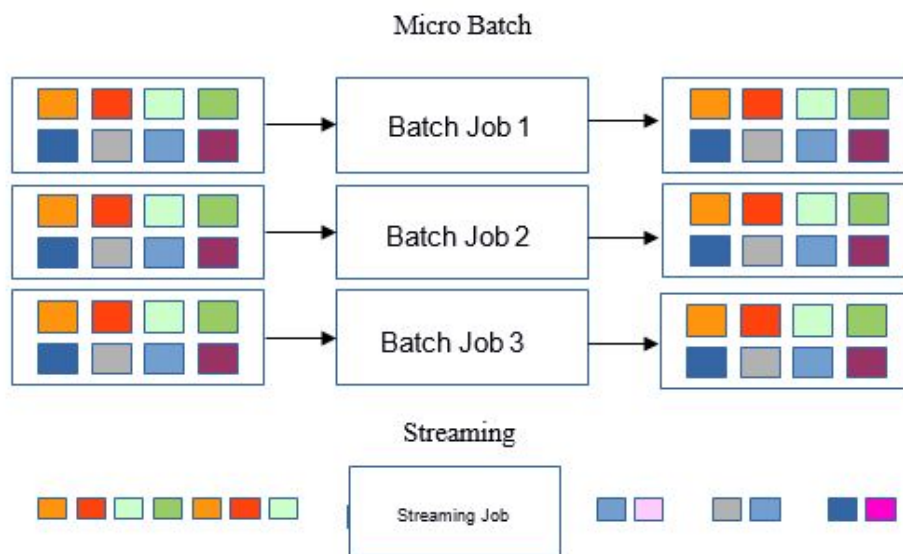


Figura 3.3 - Streaming - indizen.com

En este proyecto, una vez encolados los eventos, un proceso los extraerá y los procesará en tiempo real para darles un sentido y una forma. La herramienta debe poder procesar grandes cantidades de datos en apenas milisegundos y debe poder escalarse entre distintos servidores de manera distribuida ante una necesidad de ampliar todo el sistema.

Este módulo se encargará de enriquecer esos datos crudos que nos vienen desde el fichero de log y darles un sentido. En esta fase, también se valorará si generar las métricas o propagar los datos enriquecidos para delegar la generación de estas al frontend del sistema. Esta última proporciona más libertad al tener el dato menos procesado para poder generar nuevas gráficas y no depender de desarrollos de terceros.

Cuando ya se tiene el dato preparado, se persistirá en una base de datos.

3.2.4. Persistencia

Es necesario que, una vez procesados los datos, se persistan durante un largo o corto periodo de tiempo para poder consumirlos y obtener los beneficios que se desean. Aquí entran en juego las bases de datos.

Una base de datos está formada por un conjunto de datos pertenecientes a un mismo contexto. Se almacenan los datos sistemáticamente para su posterior uso.

Existen dos tipos de bases de datos: relacionales y no relacionales.

Las primeras, como su nombre indica, cumplen el modelo relacional:

- Se compone de tablas únicas.
- Cada tabla se compone de columnas y filas.
- Las tablas poseen claves primarias y foráneas, las cuales se realizan las relaciones entre las tablas.
- Las claves primarias son los registros principales de las tablas y deben ser únicas (no repetibles) para poder identificarlas dentro de dicha tabla.
- Escalan verticalmente.
- ACID: atomicidad, consistencia, aislamiento y durabilidad.

Estas bases de datos requieren una previa definición de los datos junto a una estructura.

Existen muchas tecnologías de bases de datos relacionales como MySQL, PostgreSQL, Oracle, etc.

Las bases de datos no relacionales difieren del modelo clásico explicado anteriormente. En ellas los datos se almacenan en documentos, no requieren estructura fija como tablas y escalan horizontalmente.

En la figura 3.4, se puede ver claramente la diferencia entre ambas. La tabla de la izquierda, es una tabla de una base de datos relacional y la de la derecha, un documento de una base de datos no relacional.

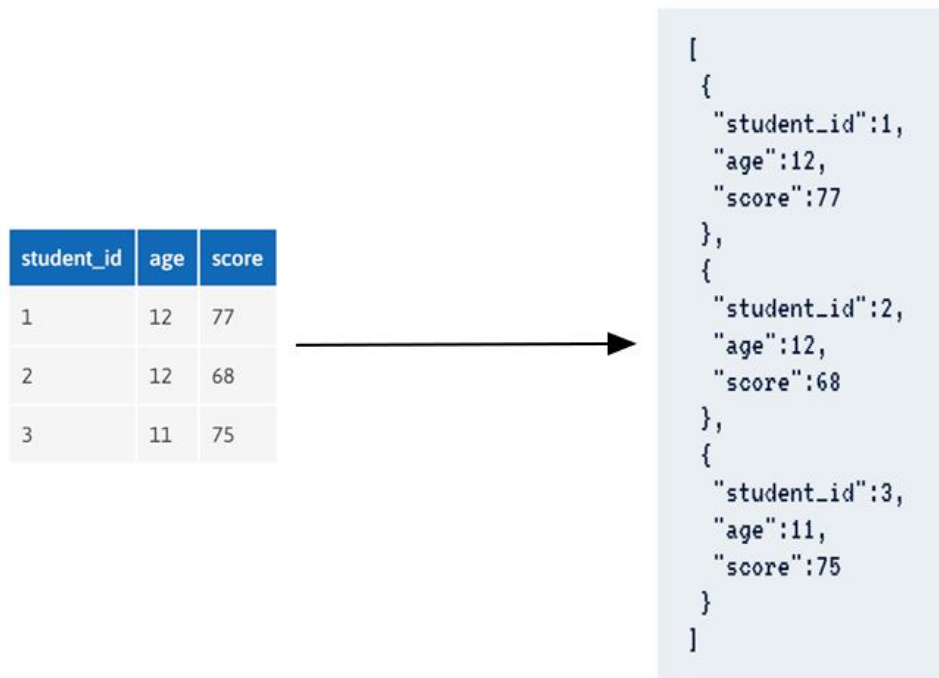


Figura 3.4 - Relacional vs No Relacional - pragma.com

Ambas figuras muestran los mismos datos pero organizados de maneras distintas. Mientras en uno hay que definir con exactitud los campos, las claves y la estructura, en el otro modelo no es necesario.

Cada modelo tiene su funcionalidad, uno orientado a estructura y el otro no. En nuestro caso, al manejar datos de log los cuales no se van a relacionar ni tienen estructura fija, el modelo más acertado es el no relacional.

Los datos tendrán un tiempo de vida (*TTL*) de pocos días dado que, además de que se ingestarán datos constantemente, los datos pierden valor según avanza el tiempo porque es un proyecto en streaming. Importa el “ahora”, no el “hace un mes”.

Es cierto que este proyecto podrá ser avanzado con técnicas de Machine Learning para poder predecir los fallos de los sistemas y, para ello, se deberán persistir estos datos por

tiempo ilimitado para tener un gran histórico sobre el que hacer análisis. En este caso se deberá redefinir el tiempo de vida de los datos.

3.2.5. Visualización

Una vez se tienen los datos procesados y almacenados, se deben ofrecer de manera intuitiva a los usuarios a los que va destinado el proyecto. Para ello, el último módulo del proyecto es un sistema frontend de visualización de los datos previamente persistidos en la fase anterior.

Al ser un proyecto en tiempo real, la tecnología que se usará debe permitirlo con tasas de refresco de los datos bajas y automáticas. Se buscará que sea un sistema liviano dado que comparte su vida con el resto de módulos.

Este sistema también deberá soportar todo tipo de gráficas y personalizaciones para que los usuarios finales creen sus propias vistas. Y, por supuesto, sistema de usuarios con privilegios y permisos. Cada usuario tendrá su *workspace* donde modificar el entorno.

3.2.6. Herramientas

Para el desarrollo se va a optar por herramientas Open Source que permitan conseguir los objetivos planteados. A continuación se detallan un conjunto de cada tipo, de las cuales se elegirá una en la fase de estudio:

- ETLs: Logstash, Fluentd, Fluentbit
- Encolamiento: Kafka, RabbitMQ, ZeroMQ, Kinesis
- Procesamiento: Spark, Kafka Streams, Flink
- Persistencia no relacional: NoSQL, InfluxDB, Elasticsearch, MongoDB
- Visualización: Grafana, Kibana

4. Diseño

Tras la previa contextualización del proyecto, sus fases y tecnologías, se comenzará detallando el diseño de la arquitectura del sistema. Básicamente, exponiendo cómo se comunican todos los componentes del proyecto en cada una de las fases previas.

Después de conocer la visión general de la arquitectura, se procederá a explicar cada componente. De esta manera será más sencillo comprender el objetivo de cada herramienta.

4.1. Arquitectura

La arquitectura del proyecto está compuesta por las cinco fases explicadas anteriormente: ETL, encolamiento, procesamiento, persistencia y visualización, como se muestra en la figura 4.1. Todas ellas conforman una cadena (pipeline) y tienen dependencia de la fase anterior. Si el módulo anterior no hace su trabajo, la cadena se romperá y los datos dejarán de fluir.

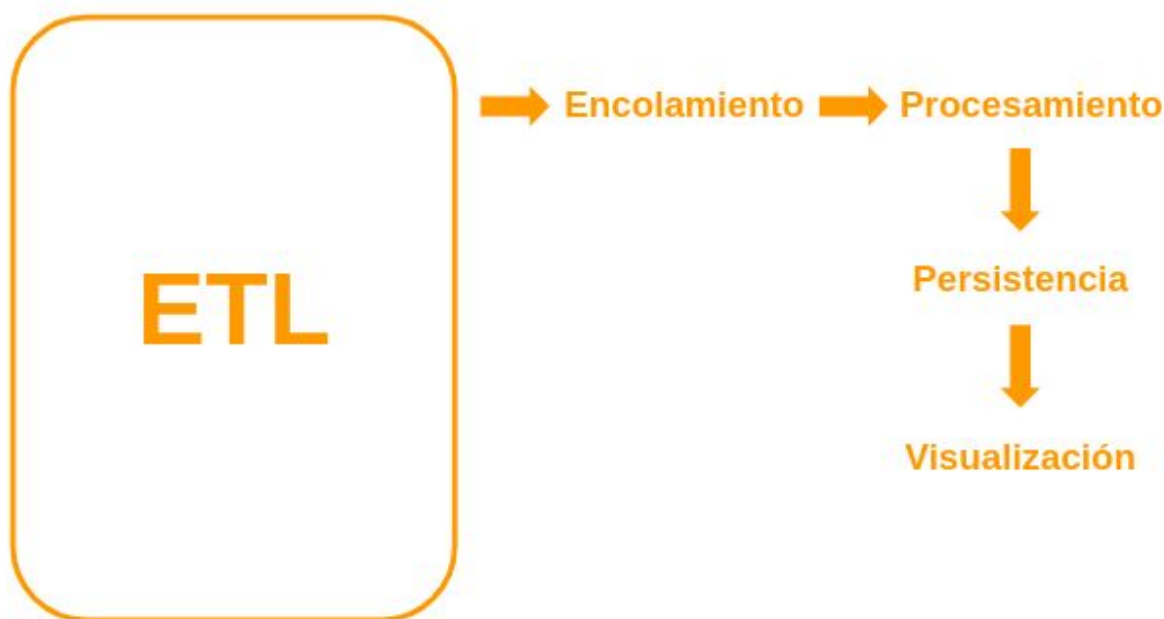


Figura 4.1 - Fases arquitectura

En la figura 4.2, la primera fase de generación del dato o ETL es la correspondiente a la caja de la izquierda. En este módulo, la herramienta de lectura y transformación de logs, obtendrá los datos de los ficheros generados por los servidores de la universidad en tiempo real. Los datos se encolan para, posteriormente, ser procesados por nuestro módulo de

enriquecimiento, el cual dará un sentido a dichos datos y los persistirá en el módulo siguiente. Una vez almacenados los datos, el módulo de visualización los mostrará de una forma amigable al usuario final.

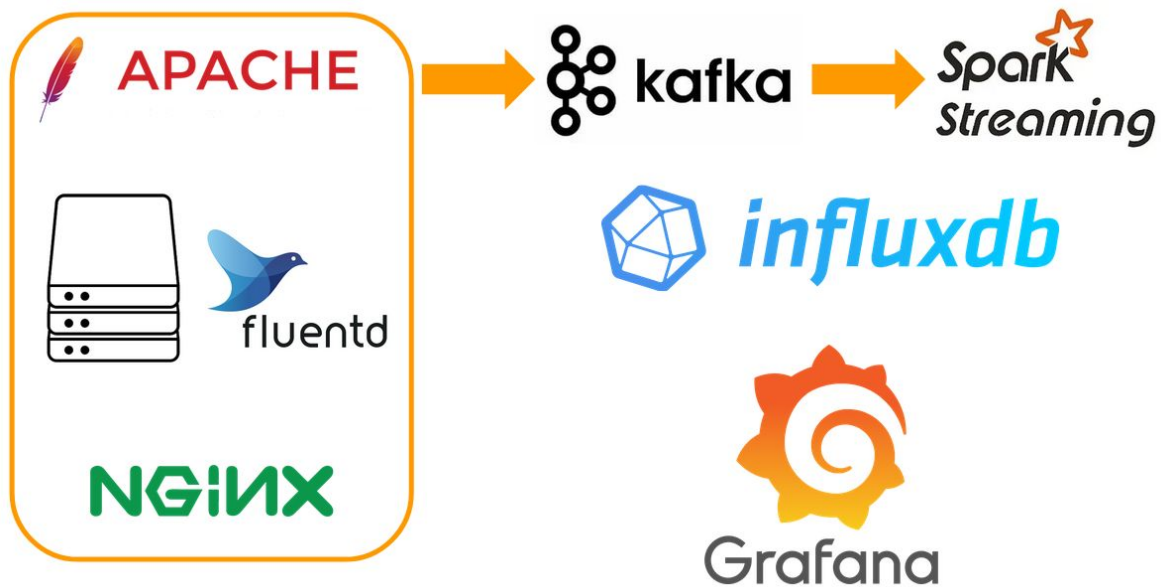


Figura 4.2 - Arquitectura

Así bien, se han reemplazado las fases por herramientas concretas, las cuales se detallarán en el punto siguiente. Se comentará el por qué de la elección de dichas tecnologías y se entrará en más detalle en su funcionamiento.

El componente de ingesta de datos se llama *fluentd*. Esta herramienta de código abierto se encarga básicamente de recopilar datos cuya fuente son ficheros. En concreto, en este proyecto se encargará de leer los ficheros de log generados por los servidores de *Apache* y *nginx*, los cuales se simularán debido a las limitaciones explicadas con anterioridad.

Para la fase de encolamiento se ha elegido Apache Kafka, una herramienta también *Open Source*. Permitirá retener los datos por un tiempo limitado hasta su consumo.

El procesamiento se ha delegado en Apache Spark con Python (*pyspark*). Spark es un *framework* de computación distribuida *Open Source*, lo cual permite procesar grandes cantidades de datos en tiempos muy pequeños. Este proceso consumirá los datos de Kafka según vayan estando disponibles.

Como base de datos no relacional se ha decidido usar InfluxDB. Es una base de datos orientada a monitoreo de operaciones, métricas de aplicaciones, datos de sensores de

Internet de las cosas y análisis en tiempo real, lo cual se adapta a la perfección con el caso de uso del proyecto.

Por último, el *frontend* de visualización será Grafana, una herramienta de código abierto que permite la visualización y el formato de datos métricos. Ofrece personalización a los usuarios de los cuadros de mando y de los gráficos, que es uno de los objetivos definidos del proyecto.

4.1.1. fluentd

Fluentd es un proyecto de software de recopilación de datos de código abierto multiplataforma. El lenguaje principal de programación en el que está escrito es Ruby [7].

El uso de esta herramienta es a través de un fichero de configuración que se divide en tres partes:

- Input: Origen de los datos, en este caso será un fichero, pero también permite escuchar en un socket, etc.
- Filter: Filtra y transforma los datos. Para este proyecto en concreto, filtra las trazas de log mal formadas y las transforma a formato JSON.
- Output: Envío de los datos a Kafka, concretamente.

En la figura 4.3, un ejemplo básico de configuración del agente *fluentd*.

```
<source>
  @type forward
</source>

<filter app.**>
  @type record_transformer
  <record>
    hostname "#{Socket.gethostname}"
  </record>
</filter>

<match app.**>
  @type file
  # ...
</match>
```

Figura 4.3 - Configuración *fluentd*

4.1.2. Apache Kafka

Apache Kafka es un proyecto *Open Source* creado por LinkedIn y donado a la Apache Foundation escrito en Java y Scala. El objetivo principal es la manipulación de datos en tiempo real, para ello debe ser una plataforma de alto rendimiento y baja latencia. Está desarrollada bajo el patrón de publicación-suscripción, como la gran mayoría de sistemas de colas de datos. Trabaja de manera distribuida en clúster, lo que hace que sea una herramienta muy poderosa dado que escala horizontalmente [8].

El patrón publicación-suscripción de Kafka se basa en topics (canales). Kafka permite crear esos canales distribuidos para que otros procesos publiquen y/o generen datos como se observa en la figura 4.4.

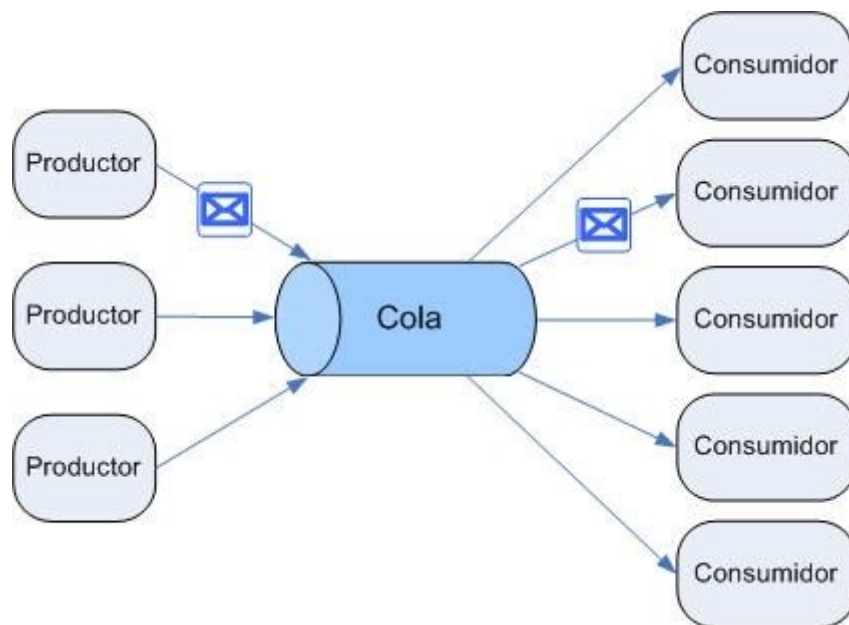


Figura 4.4 - Cola Kafka - jtech.ua.es

La figura anterior muestra un topic de Kafka, el cual tiene tres productores y cinco consumidores. Un topic tiene un nombre único que lo diferencie, así procesos externos sabrán de dónde consumir o dónde producir el dato.

Los topics mantienen ordenados los datos por orden de llegada dado que son colas FIFO. El dato más antiguo en la cola será el primero en ser consumido.

4.1.3. Apache Spark Streaming

Apache Spark es un framework de código abierto de computación distribuida en clúster. Proporciona APIs en Java, Scala, Python y R. Consta de 4 grandes módulos:

- Spark SQL: para procesamiento de datos estructurados.
- MLlib: implementación de Machine Learning.
- GraphX: procesamiento de grafos.
- Spark Streaming: procesamiento en tiempo real de los datos.

Spark Streaming es un módulo de la API core de Spark que ofrece procesamiento de datos en tiempo real de manera escalable gracias a sus ventanas de tiempo, alto rendimiento y tolerancia a fallos. Los datos pueden ser ingestados de diferentes fuentes como Kafka, Flume, Kinesis, sockets TCP, etc.

4.1.4. InfluxDB

InfluxDB es una base de datos Open Source no relacional orientada a *time series*. El uso de este tipo de bases de datos han crecido exponencialmente en los últimos años debido al crecimiento del IoT y el Big Data.

Está optimizada para alta disponibilidad, almacenamiento rápido y recuperación de datos. Tres cualidades que son necesarias en este proyecto, sobre todo la de almacenamiento rápido. Se necesita un sistema que maneje de forma eficiente esas series de datos, con miles de datos por segundo, y de los que necesitamos hacer cálculos y agregar información de manera eficiente, como por ejemplo medias, máximos...

Al ser también una base de datos distribuida, permite trabajar con millones de puntos de datos clasificados en el tiempo y puede devolver resultados en tiempo real con una alta precisión [9].

La información almacenada en InfluxDB tiene el formato que se muestra a continuación.

```
<medida>[, <etiqueta-key>=<etiqueta-value>]
<campo-key>=<campo-value>[timestamp]
```

La medida hace referencia a la métrica del dato, se puede hacer una analogía con las tablas de las bases de datos relacionales. En el ejemplo, la métrica es la cpu.

La etiqueta hace referencia a categorizaciones de ese dato, como características. Columnas que identifican ese dato respecto al resto. El servidor se llama "serverA" y está en la región oeste de Estados Unidos.

El valor es el dato objetivo que se ha medido previamente. El valor de la cpu para el servidor A de la región oeste de Estados Unidos es 0.64.

Por último, se encuentra el timestamp en formato unix. Es un campo optativo. Si no se define en la propia métrica, InfluxDB tomará ese valor como la hora local del servidor donde

esté instalado. Cada métrica debe poder identificarse con un valor en el tiempo, para poder observar la evolución de dicho dato.

```
cpu,host=serverA,region=us_west value=0.64 1434067467000000000
```

4.1.5. Grafana

Grafana es una herramienta de código abierto que permite la visualización de datos en tiempo real almacenados en bases de datos. Comenzó como un componente de Kibana, el visualizador oficial de ElasticSearch del stack ELK.

Grafana es multiplataforma y permite gestionar paneles de control y realizar gráficos. Ofrece, también, administración de usuarios y grupos, lo cual satisface uno de los requisitos del proyecto que es poder dar cuadros de mandos personalizados por usuario.

Esta herramienta de visualización es compatible con numerosas bases de datos, ya sean relacionales o no como ElasticSearch, InfluxDB, MySQL, Prometheus... Y servicios de almacenamiento cloud como AWS CloudWatch y Azure Monitor entre otros.

En la figura 4.5 se muestra un ejemplo de un panel de control desarrollado en Grafana.



Figura 4.5 - Cuadro de mando Grafana - cdmconsultores.com

5.Desarrollo

En este capítulo se hablará en más detalle de cada uno de los componentes comentados anteriormente. Para una mejor comprensión, se estructurará el capítulo en los mismos bloques que se han ido presentando durante el presente documento:

- ETL.
- Encolamiento.
- Procesamiento.
- Persistencia.
- Visualización.

El completo desarrollo del proyecto será llevado a cabo sobre el sistema operativo Linux, cuya distribución será Ubuntu 18.04.

Por falta de recursos, todos los módulos serán desarrollados e instalados en la misma máquina, aún siendo herramientas distribuidas. El correcto desarrollo y montaje de estos módulos sería con hardware dedicado, en modo cluster y en alta disponibilidad (HA) para que el sistema sea eficiente, escalable y tolerante a fallos.

5.1. ETL

Para este módulo se han necesitado tres acciones completamente diferenciadas:

- Instalación y configuración del servidor.
- Desarrollo de la herramienta de carga.
- Definición de configuración del agente fluentd.

5.1.1. Instalación del servidor

El objetivo principal del proyecto es monitorear servicios digitales, para ello lo primero que se necesita es el propio servidor. Instalarlo, configurarlo y ver qué tipo de trazas de log pueden ser de utilidad para cumplir con el objetivo.

Se ha decidido instalar el servidor universal, el más usado, Apache Server. Al ser el más usado, por lo tanto, es con el que más documentación se cuenta.

Para realizar una instalación satisfactoria basta simplemente con ejecutar el siguiente comando:

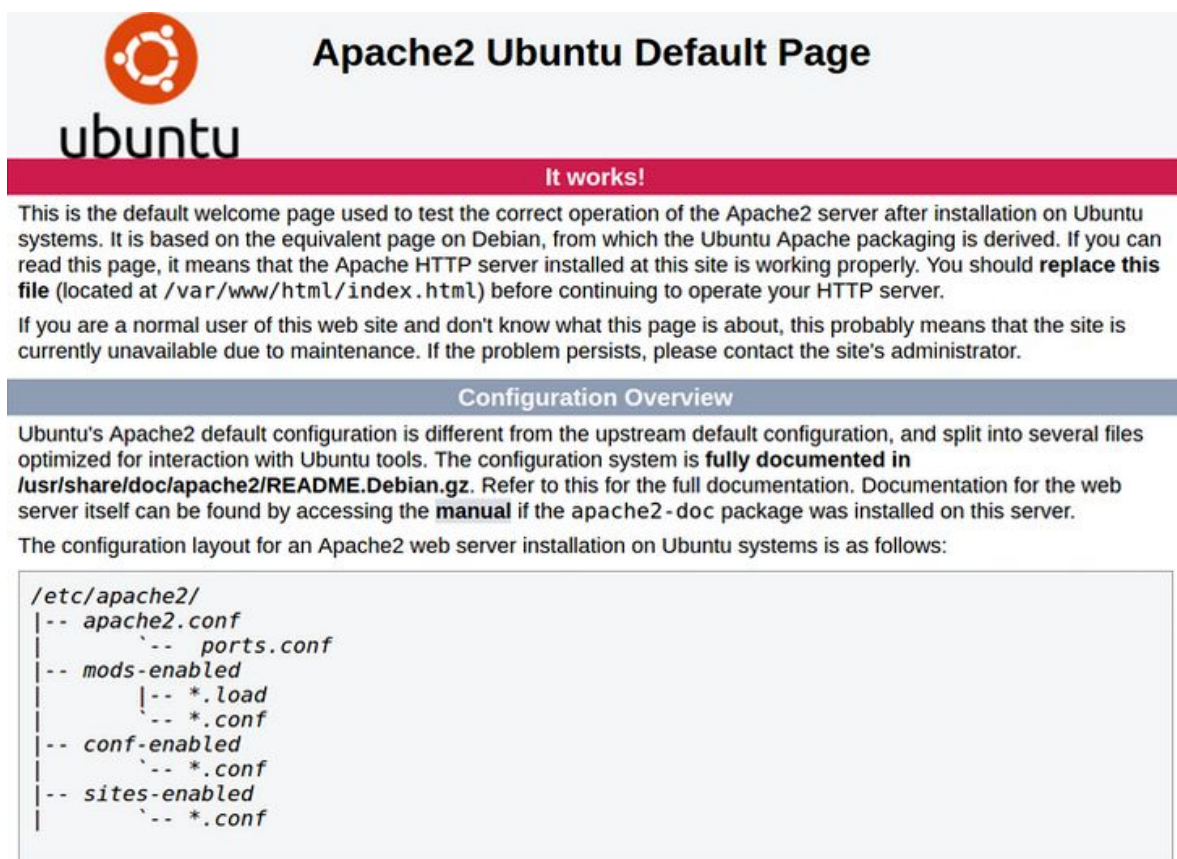
```
$ sudo apt install apache2
```

Una vez instalado, se puede llevar a cabo la configuración del cortafuegos, habilitación de Apache Secure que permite el tráfico únicamente por el puerto 443 (HTTPS), etc. Todos estos pasos son necesarios si al servidor se van a conectar usuarios externos a la red provenientes de Internet, pero este no es el caso. Únicamente se requiere la instalación como fuente de investigación, por lo que esos pasos anteriormente descritos, se obviarán.

Tras la instalación, se procede al encendido del servicio Apache:

```
$ sudo systemctl start apache2
```

Esto provoca que Apache lance un servicio web de bienvenida en el puerto 80, lo cual es accesible desde el navegador, como se muestra en la figura 5.1.



Apache2 Ubuntu Default Page

It works!

This is the default welcome page used to test the correct operation of the Apache2 server after installation on Ubuntu systems. It is based on the equivalent page on Debian, from which the Ubuntu Apache packaging is derived. If you can read this page, it means that the Apache HTTP server installed at this site is working properly. You should **replace this file** (located at `/var/www/html/index.html`) before continuing to operate your HTTP server.

If you are a normal user of this web site and don't know what this page is about, this probably means that the site is currently unavailable due to maintenance. If the problem persists, please contact the site's administrator.

Configuration Overview

Ubuntu's Apache2 default configuration is different from the upstream default configuration, and split into several files optimized for interaction with Ubuntu tools. The configuration system is **fully documented in [/usr/share/doc/apache2/README.Debian.gz](#)**. Refer to this for the full documentation. Documentation for the web server itself can be found by accessing the **manual** if the `apache2-doc` package was installed on this server.

The configuration layout for an Apache2 web server installation on Ubuntu systems is as follows:

```
/etc/apache2/
|-- apache2.conf
|   |-- ports.conf
|-- mods-enabled
|   |-- *.load
|   |-- *.conf
|-- conf-enabled
|   |-- *.conf
|-- sites-enabled
|   |-- *.conf
```

Figura 5.1 - Apache Server

Apache registra cada petición que se ha realizado contra el servidor.

```
/var/log/apache2/access.log
```

Se obtiene una traza de log por cada petición con un formato específico.

```
10.185.248.71 - - [09/Jan/2015:19:12:06 +0000] 808840 "GET  
/inventoryService/inventory/purchaseItem?userId=20253471&itemId=23434300  
HTTP/1.1" 500 17 "-" "Apache-HttpClient/4.2.6 (java 1.5)"
```

Se puede observar la IP de la máquina, la fecha y la hora de la petición, el método (GET), la URL, el código de respuesta, etc.

Estos son los datos que se deben extraer y formatear para el posterior procesado.

5.1.2. Herramienta de carga

Como se ha explicado anteriormente, al no contar con los sistemas reales, se han de simular todos estos datos como si provinieran de distintas máquinas y servicios. Recordemos que el objetivo real es monitorizar todos los servicios digitales de la Universidad Autónoma, los cuales son numerosos y están distribuidos en distintas máquinas.

Para ello se necesita una herramienta que simule todos esos datos y, además, tengan sentido para que el proyecto sea lo más real posible. La herramienta deberá generar trazas de log con el mismo formato que las generaría Apache y deberán tener sentido y cubrir un amplio abanico de posibilidades para darle credibilidad a los datos: Diferentes IPs, diferentes llamadas, diferentes usuarios, diferentes códigos...

El desarrollo de este módulo se ha realizado en Python. Se ha escogido este lenguaje debido a su sencillez a la hora de codificar y a las numerosas librerías que posee. Se ha codificado el módulo en forma de script que se ejecutará de manera indefinida para simular el tiempo real, los datos no deben nunca parar de llegar.

Con la herramienta de carga volcando los datos en el fichero de log, ahora se necesita el siguiente componente que los procese y los envíe.

5.1.3. fluentd

Esta herramienta de ETL irá instalada en todos los servidores que se quieran monitorizar. En este proyecto se ha realizado la instalación en la misma máquina donde se instalará todo lo relativo al proyecto.

El objetivo de fluentd es obtener las trazas de log generadas por la herramienta de carga y parsearlas en formato JSON para su posterior envío a Kafka.

JSON (JavaScript Object Notation) es un formato de texto sencillo de envío de datos, originado como alternativa a XML ya que este es muy tedioso y poco intuitivo.

En la figura 5.2 se muestra un ejemplo de este formato.

```
{ "empinfo" :  
  {  
    "employees" : [  
      {  
        "name" : "Scott Philip",  
        "salary" : f44k,  
        "age" : 27,  
      },  
      {  
        "name" : "Tim Henn",  
        "salary" : f40k,  
        "age" : 27,  
      },  
      {  
        "name" : "Long Yong",  
        "salary" : f40k,  
        "age" : 28,  
      }  
    ]  
  }  
}
```

Figura 5.2 - Formato JSON

Como se puede observar el dato está estructura en forma de diccionario entre llaves. Puede tener la profundidad que se quiera y soporta tipos complejos como arrays. En el ejemplo, la clave empleados tiene como valor un array para definir dentro cada empleado (nombre, salario y edad).

Este formato de texto cada día está más presente en el día a día ya que la mayoría de las nuevas herramientas lo usan debido a su gran potencial y sencillez.

Por supuesto, fluentd soporta JSON y el siguiente módulo de encolamiento, Kafka, basa su uso en dicho formato de texto. Por esta razón, se usará JSON como modelo de datos hasta el módulo de persistencia.

Como se explicó en el capítulo anterior, fluentd cuenta con tres fases diferenciadas, las cuales se ejecutan secuencialmente:

- Source: Origen y categorización vía etiqueta de los datos.
- Filter: Filtrado de etiquetas.
- Output: Envío de los datos filtrados y parseados.

Fluentd usa una notación específica para el fichero de configuración que le hará funcionar.

```
<source>
  ...
</source>

<filter>
  ...
</filter>

<match>
  ...
</match>
```

Estos tres grandes módulos son los descritos anteriormente: origen de datos, filtro de datos y destino de datos.

El origen de los datos será la ruta absoluta del fichero de log donde Apache debería volcar los datos.

```
/var/log/apache2/access_log
```

La primera parte del fichero de configuración quedaría de la siguiente manera.

```
<source>
  @type tail
  path /var/log/apache2/access_log
  pos_file /var/log/td-agent/apache2.access_log.pos
  <parse>
    @type apache2
  </parse>
  tag apache.access
</source>
```

Donde el tipo será *tail* que significa que a medida que Apache vaya añadiendo líneas al fichero, fluentd las irá consumiendo y no será necesario reprocesar toda la información cada vez.

El parámetro *pos_file* (fichero de posición) indica la ruta al fichero del cual se valerá fluentd para registrar la posición por la cual se quedó procesando el fichero de Apache.

El parámetro *parse* es *apache2*, fluentd en su propio desarrollo entiende el formato de apache de manera nativa por lo que no hace falta definir un parseador propio.

Por último el parámetro *tag* (etiqueta) le sirve a fluentd para etiquetar las trazas que ha conseguido consumir y parsear correctamente. En caso de no haber conseguido parsear una traza por malformación, fluentd no la etiquetará con el parámetro definido. Como se puede observar, se ha etiquetado a las trazas con el valor *apache.access*.

Una vez definido el origen de los datos, se procede a explicar el apartado de filtrado. El filtrado tiene el uso de filtrar y procesar de manera distinta las trazas previamente etiquetadas por el source dado que se puede dar la casuística de procesar un fichero con trazas de log totalmente distintas. En este caso no se tiene esa necesidad ya que todas las trazas de log tienen el mismo formato. Por lo tanto, no hará falta definir el apartado *filter* en el fichero de configuración.

El último apartado de envío, deberá enviar los datos a Kafka. Fluentd de manera nativa soporta el envío a Kafka gracias a su plugin *kafka2*. Básicamente habrá que indicarle los *brokers* (lista de nodos donde está instalado Kafka) y el *topic* dónde se quiere enviar, entre otras configuraciones opcionales.

```
<match apache.*>
  @type kafka2

  # list of seed brokers
  brokers <broker1_host>:<broker1_port>,<broker2_host>:<broker2_port>
  use_event_time true

  # data type settings
  <format>
    @type json
  </format>

  # topic settings
  topic_key apache_logs
  default_topic test

  # producer settings
  required_acks -1
```

```
compression_codec gzip
</match>
```

La etiqueta *match* que es donde se define el destino de los datos, viene acompañada de un *pattern* (patrón). Este patrón en forma de *wildcard* indica la etiqueta de los datos que se van a enviar. En este caso, todo lo que contenga en la etiqueta la palabra *apache* entrará al módulo de envío. Al haber etiquetado en el origen las trazas correctas con el valor *apache.access*, los datos entrarán a la fase de envío y se mandarán a Kafka.

La lista de brokers viene dada por la IP o hostname de la máquina, acompañada del puerto donde está escuchando Kafka. Para este proyecto habrá un único broker y será *localhost* dado que Kafka también se instalará en la misma máquina.

El formato de envío será JSON, como se explicó con anterioridad.

El topic al cual se enviarán los datos será *apache_logs*, previamente creado en Kafka. El topic por defecto, en caso de no existir el primero, será al que se envíen los datos.

Por último, configuraciones para hacer más eficiente el proceso y tolerante a fallos, como por ejemplo comprimir el dato o que se requiera un ACK por parte de Kafka.

Una vez instalado y configurado el agente de envío de datos y el simulador de trazas, en el siguiente capítulo se explicará cómo se encolan los datos mientras esperan a ser procesados.

5.2. Encolamiento

5.2.1. Apache Kafka

La instalación de Kafka se realizará en la misma máquina en la que se han instalado los componentes anteriores pero, en la práctica, debería ir instalado en un nodo o nodos diferentes.

Apache Kafka es una herramienta distribuida que trabaja en cluster, pero con un solo nodo es suficiente para hacer funcionar el proyecto. En caso de querer añadir más nodos en un futuro, la tarea es muy sencilla.

La instalación de Kafka en un solo nodo es muy sencilla, basta con ejecutar el siguiente comando, el cual descarga los archivos binarios de Kafka.

```
$ curl "https://www.apache.org/dist/kafka/<version>/kafka_<version>.tgz"
```

Una vez se tiene el directorio con Kafka, se cuenta tanto con el ejecutable de Zookeeper, como con el de Kafka.

Apache Zookeeper es el orquestador de Kafka, se usa para administrar el estado de sus clústeres y sus configuraciones. Se utiliza comúnmente en muchos sistemas distribuidos como componente integral. Para que el servicio Kafka funcione, es necesario que Zookeeper esté corriendo en la misma máquina.

Para facilitarnos el uso de ambos servicios, se registrarán en `systemctl` propio de Linux, que es el gestor de los servicios del sistema operativo. De esta manera, el proceso lo lanzará y gestionará `systemctl`.

En el directorio de instalación de Kafka en `/conf`, se encuentran numerosos ficheros de configuración para dicha herramienta. Los más destacados son:

- `zookeeper.properties`: Configuración del proceso Zookeeper.
- `server.properties`: Configuración del proceso Kafka.

En el directorio `/bin`, se encuentran los script para administración de Kafka, como por ejemplo:

- `zookeeper-server-start`: Inicia el servicio Zookeeper indicando las configuraciones.
- `zookeeper-server-stop`: Apaga el servicio Zookeeper.
- `kafka-server-start`: Inicia el servicio Kafka indicando las configuraciones.
- `kafka-server-stop`: Apaga el servicio Kafka.
- `kafka-console-consumer`: Lanza un proceso que consume de un topic Kafka y lo muestra por consola.
- `kafka-console-producer`: lanza un proceso que produce en un topic Kafka.
- `kafka-topics`: Crea, borra y edita los topics de Kafka.

Una vez instalado y entendido el contenido de la herramienta Kafka, se procede al inicio y configuración de esta.

Se inicia Zookeeper.

```
$ $KAFKA_HOME/bin/zookeeper-server-start.sh $KAFKA_HOME/config/zookeeper.properties
```

Se inicia Kafka, cuyo servicio estará escuchando en `localhost` en el puerto 9092.

```
$ $KAFKA_HOME/bin/kafka-server-start.sh $KAFKA_HOME/config/server.properties
```

Se crea el topic Kafka donde fluentd enviará los datos como productor llamado *apache_logs*.

```
$ KAFKA_HOME/bin/kafka-topics.sh --create --bootstrap-server localhost:9092  
--replication-factor 1 --partitions 1 --topic apache_logs
```

Se listan los topics para comprobar que existe el topic creado anteriormente.

```
$ KAFKA_HOME/bin/kafka-topics.sh --list --bootstrap-server localhost:9092
```

Ya se tendría preparado Kafka para la ingesta y consumo de datos, pero para que todo este proceso se ha ejecutado de manera satisfactoria, lo comprobaremos creando un consumidor y un productor para comprobar que todo está en orden.

```
Creación del productor:  
$ KAFKA_HOME/bin/kafka-console-producer.sh --bootstrap-server localhost:9092  
--topic apache_logs
```

```
Creación del consumidor:  
$ KAFKA_HOME/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092  
--topic apache_logs --from-beginning
```

En la figura 5.3 se puede ver que, efectivamente, produciendo datos vía terminal, el otro proceso los consume perfectamente.

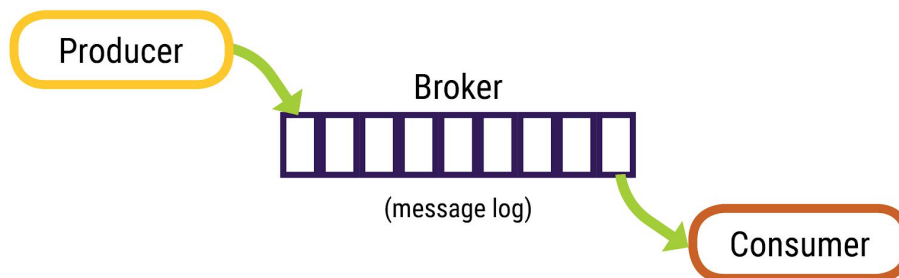


Figura 5.3 - Productor-Consumidor Kafka - datastax.com

Como refleja la figura anterior, el productor en el topic Kafka *apache_logs* sería fluentd con los datos formateados previamente en JSON y el consumidor sería el proceso del que se hablará a continuación, Apache Spark.

5.3. Procesamiento

En este capítulo se hablará de un proceso Spark que consumirá datos de la cola de Kafka y los procesará en tiempo real para su posterior persistencia en la base de datos InfluxDB.

5.3.1. Spark Streaming

Apache Spark es un framework de computación distribuida en cluster orientado al procesamiento de grandes cantidades de datos. Se usa en ámbitos Big Data de procesamiento y análisis de datos.

Apache Spark sirve para el procesamiento en batch de datos, es decir, procesar una gran cantidad de datos de una vez lo cual haría muy tedioso el desarrollo del proyecto. Existe un módulo de Spark llamado Spark Streaming, que permite desarrollar un proceso que se ejecuta a lo largo del tiempo y permite procesar los datos a medida que estos son generados por un tercero. Por lo tanto este es el módulo que se adapta a la perfección a los objetivos del proyecto.

Se ha desarrollado este módulo en el lenguaje de programación Python, dado que al ser de alto nivel, es más sencillo de programar. El desarrollo con Spark se basa en *lambdas*, programación funcional (orientado a funciones).

Spark cuenta con dos componentes básicos necesarios para entender este tipo de programación, el *driver* y los *executors*. Pueden ser tanto virtuales como máquinas físicas. Básicamente en el *driver* es donde se ejecuta el programa principal, es como el maestro de Spark. En los *executors*, esclavos, se ejecutarán las funciones propias del framework de Spark. Por lo que todo el procesamiento de datos, se realiza de manera distribuida en los nodos, físicos o no, definidos como *executors*.

Para dicho desarrollo, primero se procede a la instalación de Apache Spark en la máquina destino. Para conseguir esto, se descargará el binario con las librerías de igual forma que se hizo con Apache Kafka.

```
$ curl -O
https://www.apache.org/dyn/closer.lua/spark/spark-<spark-version>/spark-
<spark-version>-bin-hadoop<hadoop-version>.tgz
```

Será una instalación en un único nodo (*standalone*), por lo tanto basta con descomprimir el archivo y se lanzarán los procesos de maestro y esclavo. En el maestro se ejecutará el driver y en los esclavos se ejecutarán las funciones de procesado de datos de Spark.

```
# Lanzamiento del proceso maestro:
$ start-master.sh
```

```
# Lanzamiento del proceso esclavo:  
$ start-slave.sh spark://<hostname>:<port>
```

Estos procesos se registrarán en Zookeeper también para la gestión de cluster.

El ejemplo del contador de palabras en tiempo real es muy representativo de este tipo de programación. Es un programa que lee un fichero y saca la cuenta del número de palabras que existen en él.

```
sc = SparkContext("local", "PySpark Word Count Example")  
  
words = sc.textFile("D:/workspace/spark/input.txt").flatMap(lambda line:  
line.split(" "))  
  
wCounts = words.map(lambda word: (word, 1)).reduceByKey(lambda a,b: a+b)  
  
wCounts.saveAsTextFile("D:/workspace/spark/output/")
```

Para comenzar con el programa de Spark, se debe crear el **SparkContext**, que es el contexto básico de Spark, desde donde se crean el resto de variables. El contexto de Spark es propio del *driver*, y sólo se tiene en él, nunca en los ejecutores.

Una vez creado el contexto de Spark, se procede a la lectura de un fichero de texto que contiene palabras aleatorias con la función `textFile`. Esta función devuelve un objeto RDD (resilient distributed dataset) que es una colección de elementos que es tolerante a fallos y que es capaz de operar en paralelo. Es decir, que los datos consumidos del fichero se han distribuido por los ejecutores en forma de objeto. Así pues, `words` es un objeto especial que “físicamente” está repartido en particiones entre distintas máquinas.

Las funciones `flatMap` y `map`, son funciones propias de un RDD, las cuales lo transforman y devuelven otro RDD con dicha transformación. En el primer caso `flatMap`, gracias al `split`, se transforma el RDD en varias entradas divididas por un espacio (“ ”).

Tras una serie de transformaciones Spark ejecutadas en los *executors*, se obtiene el RDD final como una tupla de palabra y contador. Por último, esta tupla RDD se persiste en un fichero de salida gracias a la función `saveAsTextFile`.

En la figura 5.4 se muestra una síntesis del proceso explicado.

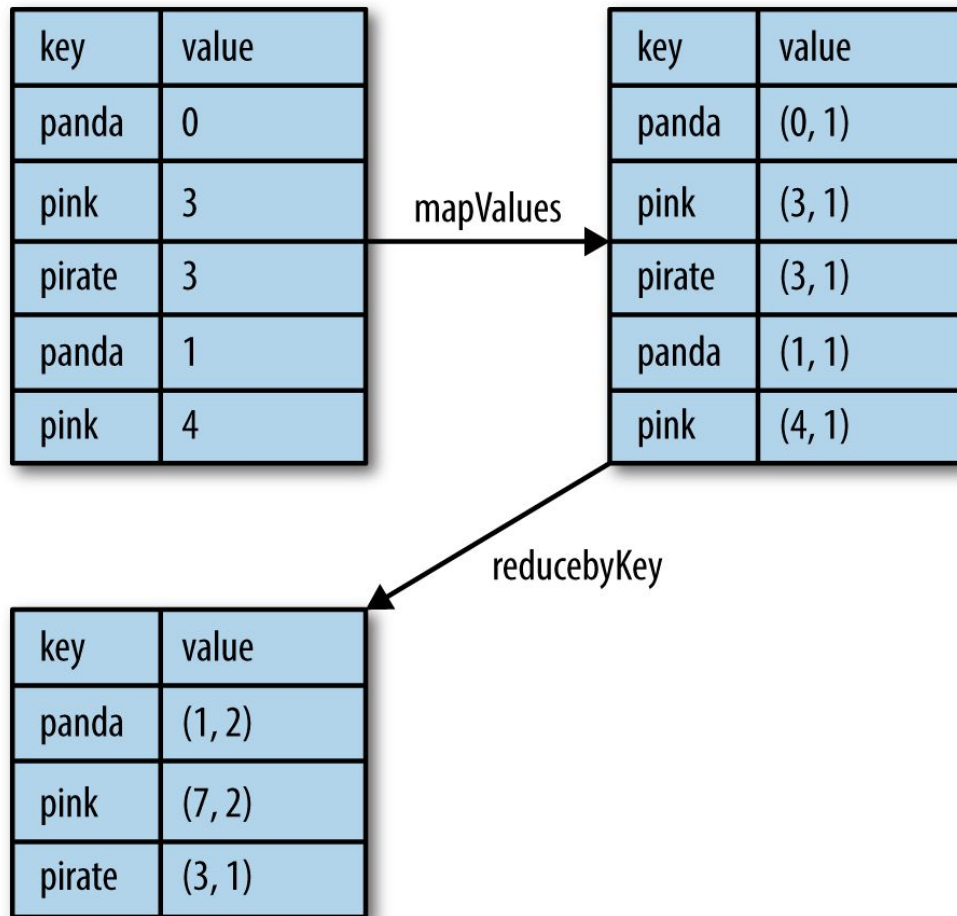


Figura 5.4 - Spark WordCount - databricks.com

Para este proyecto se ha usado Python 3 y Spark 2, es decir, las versiones más nuevas de ambas tecnologías dado que Spark 3 aún está en fase experimental. Se ha desarrollado un proyecto Python llamado *m_streaming*, el cual se compila generando un objeto .egg que contiene todas las librerías usadas. De esta manera, el compilado funcionará en cualquier otra máquina con Spark instalado.

Para ello, el proyecto cuenta con el fichero setup.py donde se declaran las dependencias y se compila el código Python.


```

from __future__ import print_function
from setuptools import setup, find_packages

__author__ = 'Abel Coronado López'
__email__ = 'abel.coronado@estudiante.uam.es'

setup(
    name='m_streaming',
    version='0.0.1',
    author='Abel Coronado López',
    author_email='abel.coronado@estudiante.uam.es',
    url='abel.coronado@estudiante.uam.es',

    packages=find_packages(exclude=['tests*']),
    install_requires=[
        'pyspark==2.4.4',
        'py4j==0.10.7',
        'PyYAML',
        'snappy',
        'influxdb'
    ],

    test_suite='tests',
    tests_require=[
        'pytest==5.2.3',
        'pytest-cov==2.8.1',
        'pylint==2.4.4',
        'pyspark==2.4.4'
    ],

    classifiers=[
        'Programming Language :: Python :: 3',
        'Operating System :: Unix'
    ],
    entry_points={

    },
    include_package_data=True,
    zip_safe=True
)

```

Este desarrollo también cuenta con ficheros de configuración YAML tanto de Spark, como de formato de entrada y salida de datos, como de conexiones con el resto de componentes.

```
app_name: StreamingLogProcessor
time_batch_window: 5
max_size_window: 100
kafka:
  brokers: localhost:9092
  topic: apache_logs
influx:
  host: localhost
  port: 8086
  user: user
  pass: pass
  db: apache_logs
```

Se ha elegido el fichero con formato YAML porque es bastante intuitivo y está bastante bien organizado. Cualquier modificación en el fichero se detectaría de una manera ágil y sencilla a diferencia de otros formatos.

El proyecto Python m-streaming cuenta con dos grandes módulos. El primero es donde se ubica el código base del proyecto llamado `m_streaming`. En este paquete se incluyen todas las clases Python y Spark las cuales realizarán todo el procesamiento de datos.

- `main.py`: Programa principal que se ejecuta en el driver de Spark. Orquesta el flujo principal del programa. Se crean los contextos de Spark Streaming y SQL para la creación de dataframes y RDDs y la sesión de Spark.

```
spark = SparkSession \
    .builder \
    .appName(config['app_name']) \
    .getOrCreate()
ssc = StreamingContext(spark.sparkContext, config['time_batch_window'])
sql_context = SQLContext(spark)
```

- `utils`: En este módulo se cuenta con clases genéricas para dar soporte a las distintas funciones Spark.
- `conf`: Directorio donde se guardan todas las configuraciones descritas anteriormente en formato YAML.
- `common`: Aquí encontramos las clases con funciones personalizadas que utilizará el framework de Spark que se ejecutará en los ejecutores. Funciones que se le pasarán a los `maps`, `reduceByKey`, etc. Como por ejemplo `load_json_from_string`.

```

@staticmethod
def load_json_from_string(json_str: str) -> dict:
    try:
        ret = json.loads(json_str)
    except json.JSONDecodeError:
        ret = dict()

    return ret

```

El segundo módulo es para los recursos, llamado *resources*. Entre otros, se puede encontrar:

- data.json: entradas JSON con las trazas de log parseadas de cara a los tests unitarios.

```

{"host":"127.0.0.1","user":"peter","method":"GET","path":"/sample-image.png","code":200,"size":1479}

```

- init.sh: Script para bash para iniciar todos los servicios como Kafka, Zookeeper, Spark, etc. También para la creación de topics.
- log-samples: Ejemplos de trazas crudas de Apache para realizar también tests.

```

127.0.0.1 - peter [9/Feb/2017:10:34:12 -0700] "GET /sample-image.png HTTP/2" 200 1479

```

Una vez transformados los eventos JSON provenientes de Kafka, el módulo de Spark Streaming, los convertirá en formato InfluxDB para su posterior inserción. Como se explicó en capítulos anteriores, InfluxDB tiene un formato concreto, pero no es necesario transformar las trazas a dicho formato.

```

<medida>[,<etiqueta-key>=<etiqueta-value>] <campo-key>=<campo-value>[timestamp]

```

No hay que parsear cada evento JSON al formato anterior, sino que Python cuenta con una librería de inserción en InfluxDB llamada *influxdb*. Esta librería, junto al resto de ellas definidas en el *setup.py*, son necesarias para el correcto funcionamiento de este módulo.

Esta librería es muy útil porque permite definir una conexión contra InfluxDB definiendo el host, las credenciales, etc. Para la inserción de los datos basta con parsear el JSON en un formato que la librería entienda. Campos obligatorios:

- Measurement (medida): "Tabla" destino del evento.

- Tags (etiquetas): Lista de etiquetas con sus respectivos valores.
- Fields (campos): Lista de medidas con sus respectivos valores para ese evento.

```
{
  'measurement': 'apache_logs',
  'tags': {
    ...
  },
  'fields': {
    ...
  }
}
```

Una vez *parseado* el evento a ese formato JSON de InfluxDB, el cliente InfluxDB para Python ofrece la función `write_points`, que es la encargada de persistir ese evento en la base de datos previamente definida.

5.4. Persistencia

En este capítulo se detallará el desarrollo del módulo de persistencia que viene dado por la herramienta InfluxDB.

5.4.1. InfluxDB

InfluxDB, como se ha hablado anteriormente, es una base de datos no relacional de código abierto orientada a *time series*. Es una herramienta de procesamiento y almacenamiento distribuida que trabaja en cluster. Esto permite que escale horizontalmente a través de nuevas máquinas que se vayan añadiendo a la infraestructura del proyecto. Permite replicación de los datos, lo cual es beneficioso en caso de catástrofe.

En esta base de datos se almacenarán todos los datos previamente enriquecidos por el módulo de Apache Spark y se insertarán los datos de una manera constante en el tiempo, ya que el pipeline del proyecto es en tiempo real.

La instalación de InfluxDB es sencilla, se descarga el paquete de la herramienta para Ubuntu del repositorio oficial y se instala con el comando `dpkg`.

```
$ wget
https://dl.influxdata.com/influxdb/releases/influxdb_<version>_amd64.deb
$ sudo dpkg -i influxdb_<version>_amd64.deb
```

La configuración de InfluxDB por defecto es lanzar el servicio en el puerto 8086 y tiene definido un usuario con una contraseña para la administración [10].

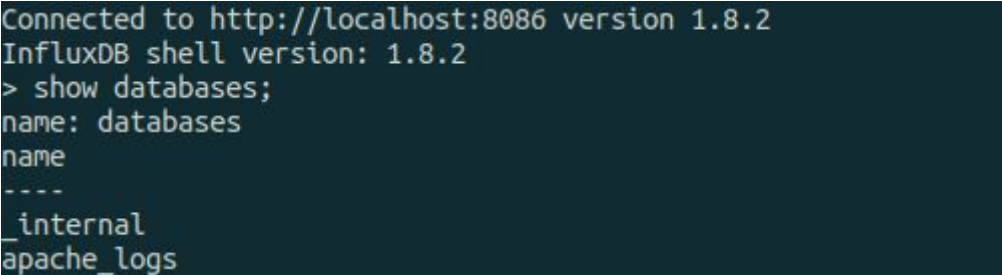
Lo primero que se hará es proceder a la creación de un usuario para la gestión de la base de datos de los logs de monitorización.

```
$ influx user create -n <username> -p <password>
```

Después se procede a la creación de la base de datos y dar permisos al usuario previamente creado.

```
> CREATE DATABASE <database>  
> GRANT [READ,WRITE,ALL] ON <database> TO <username>
```

De esta manera, como aparece en la figura 5.5, se tendrá un usuario propio para el manejo de esta base de datos, con los privilegios requeridos.



```
Connected to http://localhost:8086 version 1.8.2  
InfluxDB shell version: 1.8.2  
> show databases;  
name: databases  
name  
----  
_internal  
_apache_logs
```

Figura 5.5 - Influx Shell

Se ha realizado una conexión a InfluxDB vía terminal para realizar los comandos previamente descritos. Para comprobar que todo ha ido correctamente, se pueden listar tanto usuarios, como sus privilegios, como las bases de datos, etc. Para mayor comprobación, se puede crear una base de datos temporal para insertar algún dato sobre el que realizar búsquedas.

```
> INSERT weather,location=us-midwest temperature=82 1465839830100400200  
> SELECT * FROM "weather"
```

En la figura 5.6 se muestra la salida de los comandos explicados.

```
> INSERT weather,location=us-midwest temperature=82 1465839830100400200
> select * from weather
name: weather
time                location    temperature
----                -
1465839830100400200 us-midwest 82
```

Figura 5.6 - Influx data

5.5. Visualización

En este apartado de Visualización se explicará la instalación de la herramienta Grafana, su configuración, su conexión con InfluxDB y cómo se crean dashboards y paneles de control.

5.5.1. Grafana

Grafana es un sistema de visualización multiplataforma que permite crear visualizaciones de los datos de una base de datos. Grafana cuenta con múltiples conectores a diferentes bases de datos como, por supuesto, InfluxDB.

Grafana cuenta con dos versiones de la herramienta:

- Enterprise: De pago, con características añadidas y soporte en caso de incidencia. Enfocada a empresas.
- OSS: La versión básica para usuarios normales. Es gratis.

Para este proyecto, obviamente se usará la versión gratuita dado que uno de los requisitos era el coste cero.

Para la instalación basta con instalar las siguientes dependencias en la máquina Linux destino.

```
$ sudo apt-get install -y apt-transport-https
$ sudo apt-get install -y software-properties-common wget
```

Al ser código abierto, la propia empresa Grafana tiene repositorios oficiales donde poder descargar ambas versiones. Por lo tanto, se añadirá el repositorio de la versión OSS.

```
$ echo "deb https://packages.grafana.com/oss/deb stable main" | sudo tee
-a /etc/apt/sources.list.d/grafana.list
```

Esto provocará que cuando se instale grafana vía el instalador de Ubuntu, el sistema operativo reconozca este nuevo repositorio y busque el paquete de Grafana en él en vez de en los repositorios nativos de Linux.

```
$ sudo apt-get update
$ sudo apt-get install grafana
```

Una vez instalada la herramienta, se procede al encendido del servicio.

```
$ sudo systemctl start grafana-server
```

Automáticamente, si todo ha ido bien, se tendrá un servicio en el puerto 3000 corriendo en la máquina, que es el puerto por defecto de la instalación. Accediendo vía navegador web, como aparece en la figura 5.7, se encontrará la primera pantalla de Grafana.

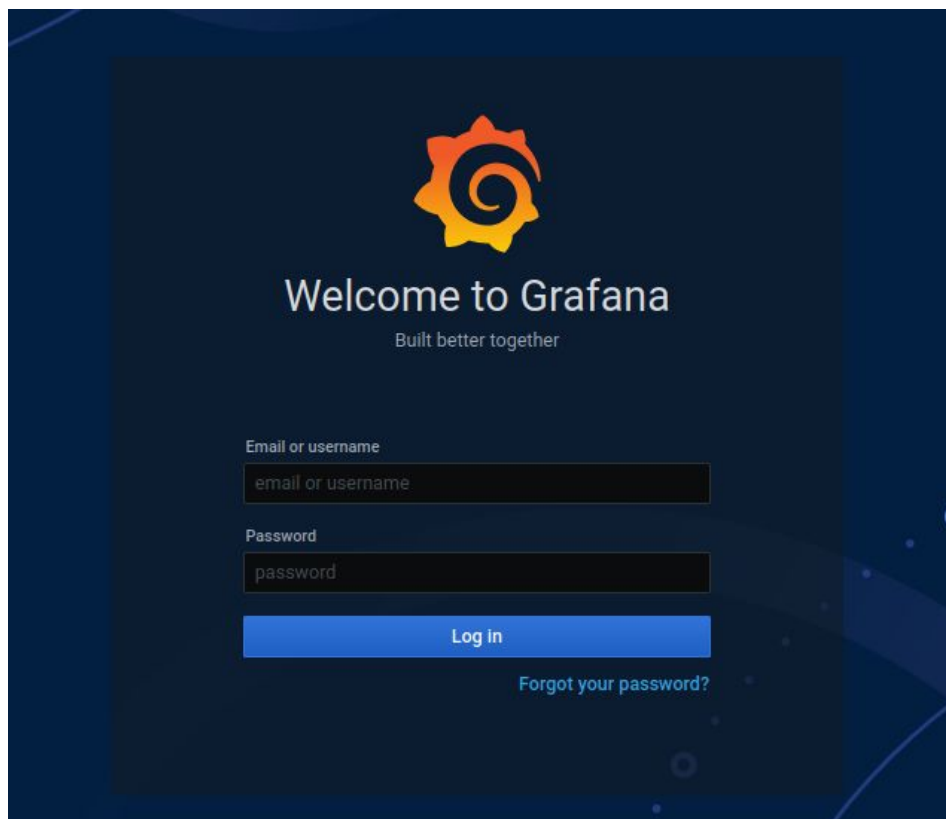


Figura 5.7 - Grafana login

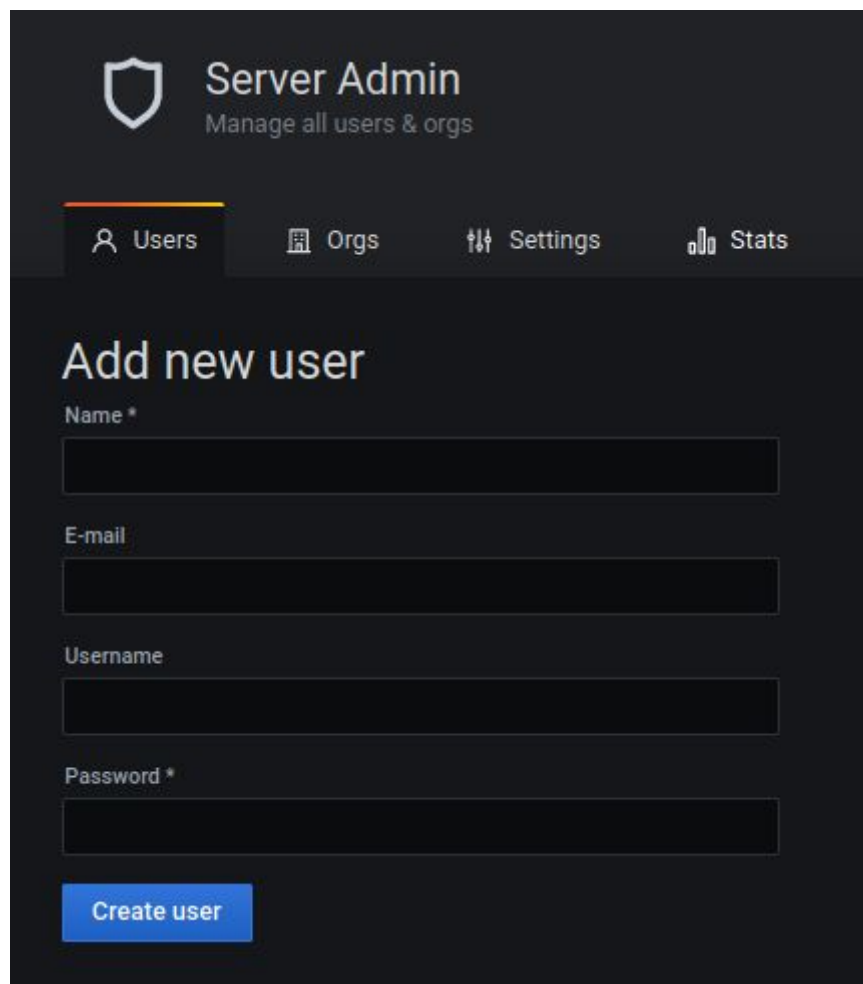
Las credenciales por defecto de Grafana son usuario *admin* y contraseña *admin*, las cuales se pueden ver en el fichero de configuración de la herramienta.

```
$WORKING_DIR/conf/defaults.ini
```

La primera vez que se inicia sesión, Grafana te obliga a cambiar la contraseña para el usuario admin por temas de seguridad, dado que este usuario va a ser el que administre todo por completo.

Para no usar este usuario de manera habitual, se han creado usuarios específicos con privilegios concretos. Por lo tanto, se creará un nuevo usuario para esta herramienta que tenga visibilidad sólo al *workspace* y a los datos que se han estado desarrollando.

En la figura 5.8 aparece la pantalla de creación de usuarios de Grafana.



The screenshot shows the 'Server Admin' interface with the 'Users' tab selected. The main heading is 'Add new user'. Below this, there are four input fields: 'Name *', 'E-mail', 'Username', and 'Password *'. At the bottom left, there is a blue button labeled 'Create user'.

Figura 5.8 - Nuevo usuario

Ahora, para ver los datos cargados en InfluxDB, se ha de configurar el *datasource* o conector a la base de datos. Como se comentó anteriormente, Grafana cuenta en sus propias distribuciones con múltiples conectores a diferentes bases de datos, ya sean *time series*, relacionales, cloud, etc.

En la figura 5.9, se observa que Grafana tiene categorizados sus conectores por los tipos de bases de datos y sus casos de uso. En este caso, en el apartado que interesa, time series, se encuentran estas 4 bases de datos que vienen de manera nativa, sin plugins.

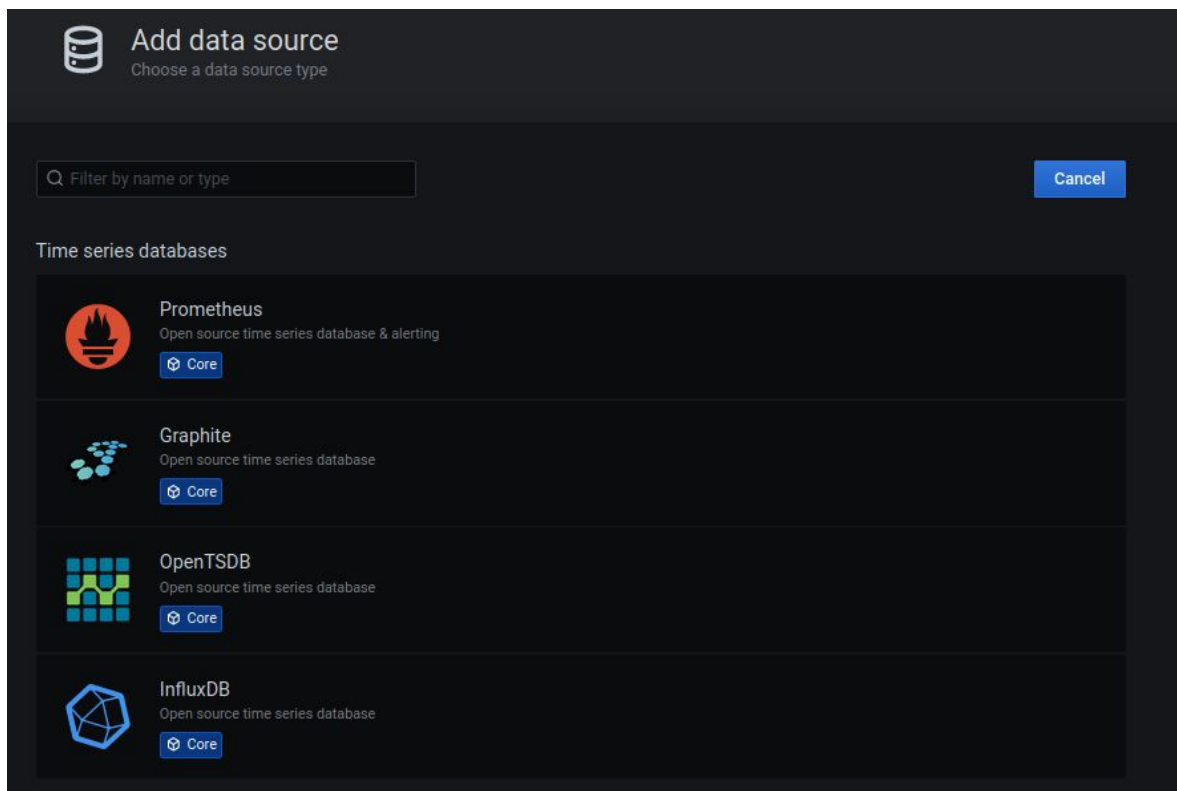


Figura 5.9 - Crear datasource 1

Se elige InfluxDB y nos redirige a un *wizard* de creación para esta base de datos, figura 5.10. Aquí aparecerá el nombre que se le quiere asignar al datasource, el lenguaje sobre el que se quieren hacer las consultas a la colección de InfluxDB, la URL junto con el puerto de la base de datos, credenciales de usuario, etc.

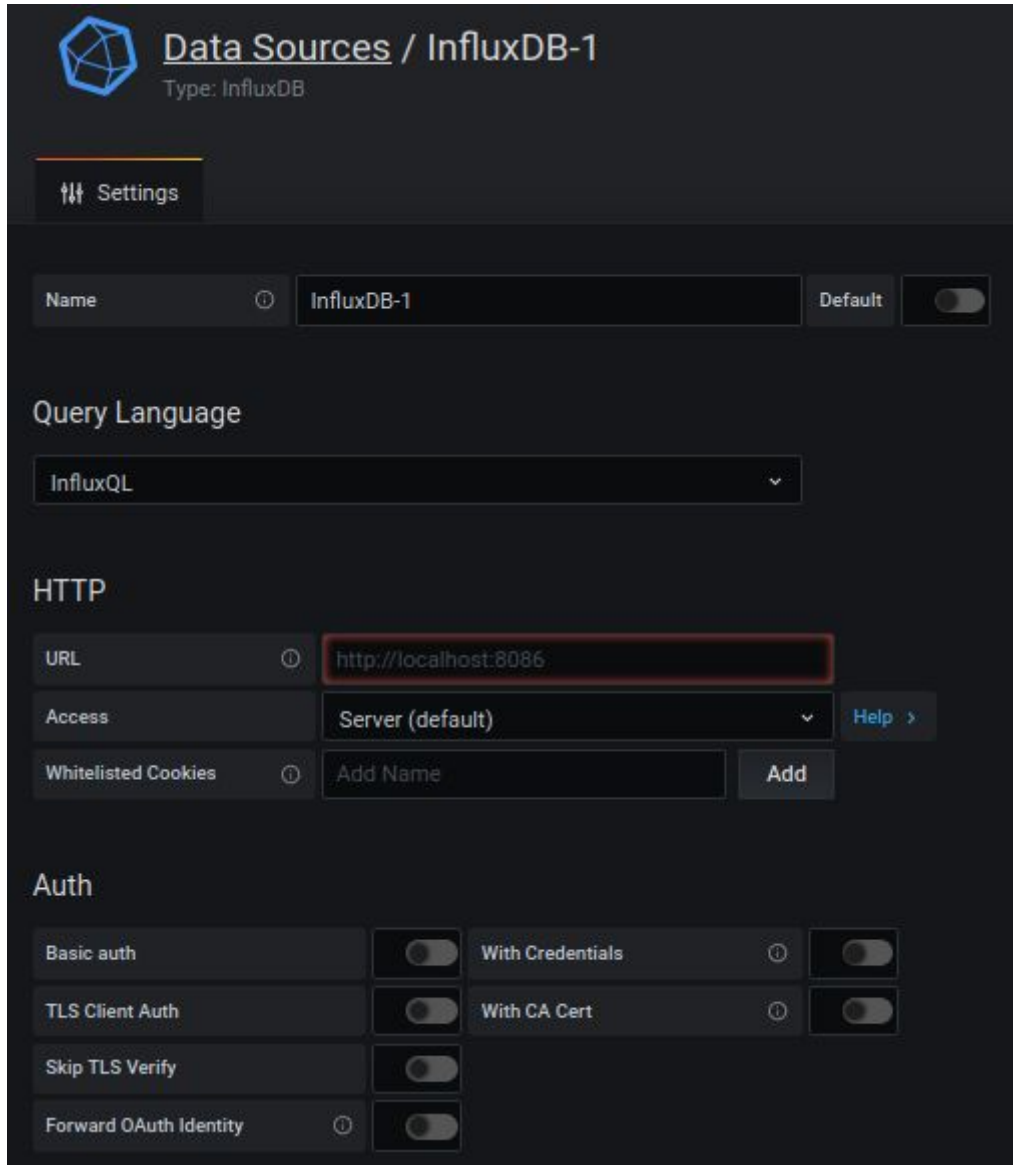


Figura 5.10 - Crear datasource 2

Con el datasource ya creado, ya se puede proceder a la creación del primer dashboard del proyecto. Se probará a calcular los códigos 200 provenientes de los distintos servidores virtuales que se han generado con la herramienta de carga como se muestra en la figura 5.11.

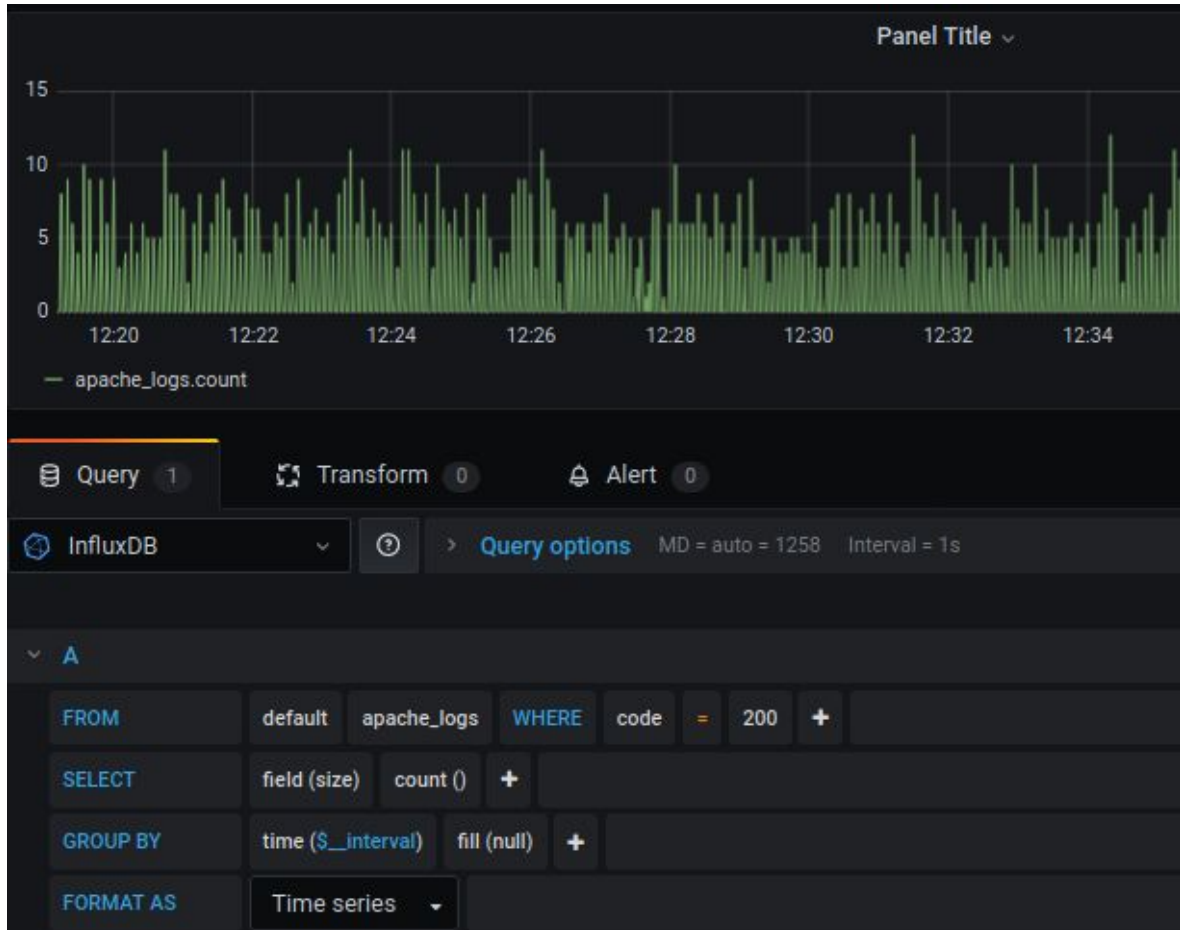


Figura 5.11 - Crear dashboard

6.Resultados

Como se ha ido detallando durante todo el documento, los resultados finales serán dashboards (paneles de mando) donde se mostrarán los datos transformados durante todo el pipeline. Los dashboards diseñados tan sólo serán unos pocos de los infinitos que se puedan desarrollar. Como se especificó, los dashboards serán personalizables por cada usuario de manera que cada uno de ellos tenga su visión personal del estado de los servicios que mejor le ayude a desempeñar su trabajo. Eso es porque cada persona entiende unos gráficos mejor que otros o entienden los datos de maneras distintas.

Las imágenes que se mostrarán a continuación son capturas de pantalla reales del proyecto del módulo frontend Grafana, ya que es el resultado visible que el usuario puede percibir. Estas imágenes muestran los datos enriquecidos y transformados que han pasado por todo el pipeline del proyecto hasta que desembocan en InfluxDB y Grafana los expone.

Un gráfico concreto por sí solo no da toda la información necesaria para saber concretamente qué está ocurriendo. El conjunto de varios sí, por eso se agrupan las visualizaciones en dashboards que tengan relación entre ellos y aporten muchísima más información extra.

La figura 6.1 muestra un ejemplo del cuadro de mando desarrollo en este proyecto. Se irá panel a panel explicando cada uno de los KPIs que se han creído útiles para el entendimiento de la monitorización de los servidores.



Figura 6.1 - Ejemplo de cuadro de mando

El primer gráfico o grupo de gráficos, figura 6.2, hace referencia a la media de los códigos problemáticos de respuesta [11].

- 300: Redirecciones.
- 400: Errores de cliente.
- 500: Errores de servidor.

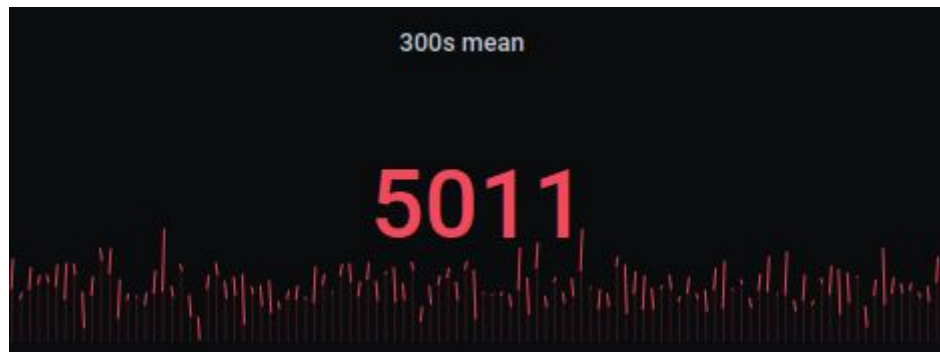


Figura 6.2 - Códigos 300

Este tipo de gráfico que da un único valor, proporciona la utilidad de poner *thresholds* (límites) al dato y en función de los rangos dados, que el gráfico visualmente aporte más información cambiando de color, como se muestra en la figura 6.3.

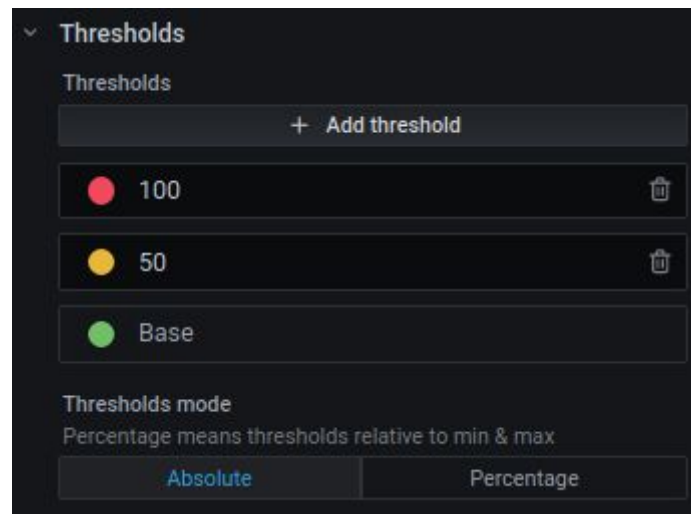


Figura 6.3 - Thresholds personalizables

Una manera intuitiva de asignar los colores y los rangos es simular un semáforo. De esta manera si el dato está verde no indica peligro pero, si está rojo sí. Colocar un color intermedio como el ámbar puede ayudar a la prevención de riesgos.

En este caso, si la media de códigos supera los 50 en valor absoluto, el gráfico se pondrá de color ámbar y si pasa de los 100, se considera que es un valor crítico y el gráfico se tornará rojo, indicando peligro. Si el dato se mantiene por debajo de 50, el dato será verde, indicando que ese KPI está en orden.

Estos límites son personalizables por cada usuario dado que para este proyecto se han simulado los datos. Se ha creído conveniente que los thresholds sean 0, 50 y 100, pero esto variará en función del servicio a monitorizar y la definición de dichos límites la deberán dar los usuarios porque son los que, en teoría, deberán conocer el servicio.

La figura 6.4 es un gráfico de líneas que agrupa el dato por los distintos códigos generados por el servidor. De esta manera, de una manera visual, se podrá saber qué códigos de respuesta concretos son los que se están generando.

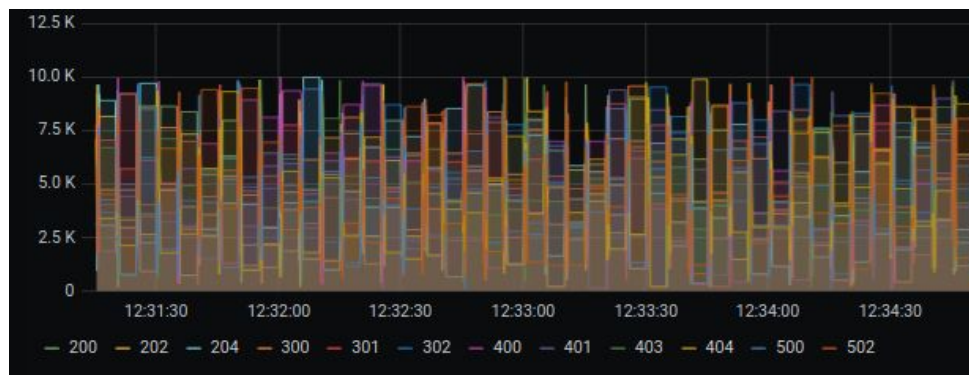


Figura 6.4 - Contador de códigos

Otra manera de mostrar el dato es con mapas de calor, en concreto la figura 6.5 muestra el número de peticiones GET que se están enviando contra los servidores. Este gráfico en conjunto con el resto puede dar una visión mucho más completa del estado de los servicios.

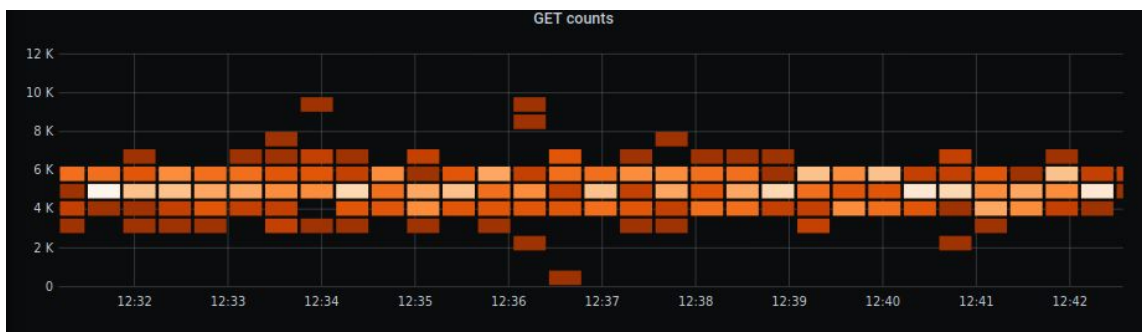


Figura 6.5 - Mapa de calor

Una idea, como se muestra en la figura 6.6, podría ser saber qué servicios en un servidor concreto son más problemáticos. Así pues, se ha creado un gráfico por cada tipo de código (200, 300, 400, 500) agrupando por las distintas IPs de los servidores involucrados. En este caso, el generador de trazas explicando anteriormente, genera trazas con estas tres IPs: 10.0.0.2, 10.0.0.3, 127.0.0.1.

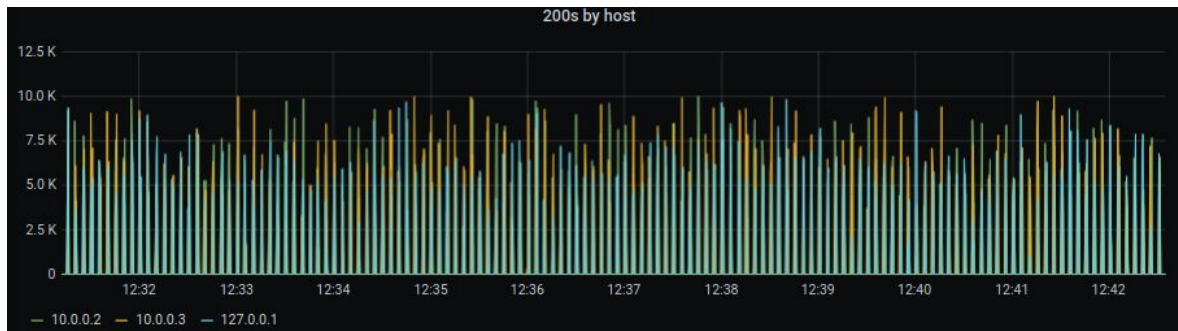


Figura 6.6 - Códigos 200 por host

7. Conclusiones

Para concluir, en este último apartado se detallarán los aprendizajes e impresiones personales que se han obtenido durante el desarrollo del proyecto y las conclusiones una vez finalizado.

El proyecto permite a los usuarios finales, tener una visión centralizada del estado de los servicios que ellos administran. Esto ahorra así una búsqueda exhaustiva y tediosa a través de ficheros de logs y demás fuentes de datos, así como un tiempo muy valioso que se podría invertir en la mejora de dichos servicios. Con un simple vistazo se puede llegar a realizar un diagnóstico del problema y una resolución más rápida y eficaz.

Durante la realización de este Trabajo de Fin de Máster, se ha aprendido a desarrollar un *pipeline* de datos complejo a través de unas de las herramientas más punteras de Big Data del mercado como pueden ser Kafka o Spark. La instalación, configuración y desarrollo de estas herramientas ha servido de un gran aprendizaje del mundo de la transformación y análisis del dato.

8. Trabajo futuro

Los objetivos y requisitos del proyecto han sido satisfechos en su totalidad pero hay mucho trabajo futuro de mejora que hacer. El desarrollo de este proyecto tan sólo es la punta del iceberg de todo lo que se puede llegar a hacer debido a su potencial y a las diversas posibilidades y frentes abiertos que tiene, lo cual es algo muy positivo.

Para no romper con la línea argumental de este documento, se procederá a indicar las mejoras y el futuro trabajo de cada uno de los módulos, empezando por el de ETL para finalizar con el de visualización.

La instalación del agente fluentd es sencilla y de fácil configuración, no supondría un problema hacerlo de forma manual en tres o cuatro ocasiones. En cuanto un mismo proceso se tiene que repetir en el tiempo, es que hay algo que no está bien planteado y/o desarrollado. Por ello, se propone automatizar el despliegue tanto de la instalación como de la configuración de esta herramienta. Así, con tan sólo darle a un botón, el agente se desplegará en tantas máquinas como se quieran monitorizar.

Herramientas como Ansible o Puppet para la automatización de tareas remotas, podrían ser de ayudas para cumplir dicho objetivo. También Jenkins para la orquestación de dichas ejecuciones.

De esta manera, cuando los servicios digitales de la Universidad Autónoma de Madrid escalen, añadiendo más y más servidores, éstos entrarán en la rueda de la monitorización, suponiendo un avance enorme de cara a la calidad de dichos servicios y ayuda a su operación.

En lo que se refiere al módulo de encolamiento Kafka, al final todo avanza al *cloud* debido a la delegación de todas las tareas de administración y mantenimiento de los llamados *clusters*. Suelen ser servicios caros pero está demostrado que, si se les da un buen uso, terminan compensando porque son servicios inteligentes que se adaptan a la carga a la que están sometidos, crecen o decrecen en función de si hace falta o no.

Tanto AWS (Amazon Web Services) como Azure o Google Cloud, proveen de servicios de encolamiento como Kafka, cómodos de usar y con cero mantenimiento. Suelen cobrar por evento ingestado y/o por almacenamiento, aunque este sea volátil.

Bastaría con que los agentes instalados *on-premise* envíen los eventos de log a alguno de los servicios *cloud* previamente nombrados.

Para Spark, la mejora propuesta también será llevarlo al cloud. Al igual que con Kafka, existe su homónimo en la nube. Por ejemplo, AWS cuenta con un servicio Big Data llamado Amazon EMR, que es un clúster analítico para el uso de herramientas tales como Apache

Spark. Se lanzan los trabajos de tiempo real, eligiendo la versión de Spark y su ejecución es transparente para el usuario. Sin preocupaciones de librerías, versionado, capacidad, número de *cores* de las máquinas y un largo etcétera.

Todos los proveedores cloud cuentan con data warehouses y data lakes, lo cual supone un problema menos a tener en cuenta. Por lo tanto, llevar InfluxDB al cloud sería uno de los trabajos futuros siempre y cuando se cuente con el presupuesto necesario.

Por último, la visualización. Grafana cuenta de manera nativa con conectores para los servicios cloud de almacenamiento de los distintos proveedores, por lo que no parece muy necesario tener la visualización en un servidor ajeno cuando el *grosso* del cómputo ya está realizado. Al no tener muchos usuarios, no parece que el servidor donde esté ubicado Grafana vaya a tener mucho uso.

En vista a un posible trabajo futuro, sería estudiar y analizar el dato, explorando diferentes casos de uso para dar más valor a éste. Explorar dónde se podría obtener más dato que complementa y nuevas fuentes.

Por ejemplo, las métricas físicas como *cpu*, *throughput*, memoria... Este tipo de dato, mezclado con el que ya se tiene, podría avisar y dar más información ante posibles problemas gracias a las tendencias que vayan teniendo.

Además de la mejora y de los procesos ya existentes, se pueden crear otros nuevos para añadirlos al pipeline con el que se cuenta.

Por ejemplo, un sistema de alarmas, ya sea a nivel de base de datos o a nivel de visualización. Este sistema configurado con una serie de rangos en las métricas, podrían alertar de situaciones no controladas para que se remedien manualmente antes de que pasen. Kapacitor del stack de InfluxDB o Zabbix, son herramientas para este caso de uso concreto.

Teniendo un sistema de alarmas, se podría configurar otro módulo que ante una alarma dada, ejecute una serie de pasos para la resolución del problema. Alertas muy controladas por problemas de fácil resolución, serían un buen caso de uso para este módulo. El objetivo final es resolver la incidencia de manera automática sin intervención humana.

Se ha hablado de dos nuevos módulos de alarmado y resolución de problemas pero, ¿y si el sistema se pudiera anticipar a los problemas para que nunca ocurran? Un posible trabajo futuro es el desarrollo de un sistema automático que aprenda de las tendencias de los datos y cuándo van a desembocar en un caso problemático. En un año se obtendrían muestras de datos suficientes como para entrenar un sistema de *Machine Learning* para que cumpla con los requisitos propuestos y, de esta manera, predecir un posible problema antes de que ocurra.

Bibliografía

- [1] *Backup Software & Data Protection Solutions - Acronis*. (2020, November 20). Retrieved from <https://www.acronis.com/es-es>
- [2] *Observability in One Place (music, no auto start)*. (2020, July 18). Retrieved from <https://newrelic.com>
- [3] *Top 10 Server & Application Monitoring Tools | Acronis.com*. (2020, October 08). Retrieved from <https://www.acronis.com/es-es/articles/monitoring-tools>
- [4] *BIG DATA: EXTRAER, TRANSFORMAR Y CARGAR LOS DATOS - Instituto Internacional de Ciencia de Datos*. (2020, March 25). Retrieved from <https://i2ds.org/2016/05/04/big-data-extraer-transformar-y-cargar-los-datos>
- [5] *Colaboradores de los proyectos Wikimedia*. (2020, June 13). Cola (informática) - Wikipedia, la enciclopedia libre. Retrieved from [https://es.wikipedia.org/w/index.php?title=Cola_\(inform%C3%A1tica\)&oldid=126910643](https://es.wikipedia.org/w/index.php?title=Cola_(inform%C3%A1tica)&oldid=126910643)
- [6] *PowerData, R*. (2020, October 07). Los 3 principales tipos de técnicas de procesamiento y análisis de datos. Retrieved from <https://blog.powerdata.es/el-valor-de-la-gestion-de-datos/los-3-principales-tipos-de-tecnicas-de-procesamiento-y-analisis-de-datos>
- [7] *Fluentd Project*. (2020, November 21). Fluentd | Open Source Data Collector. Retrieved from <https://www.fluentd.org>
- [8] *Apache Kafka*. (2020, October 25). Retrieved from <https://kafka.apache.org/intro>
- [9] *Instalación y primeros pasos con InfluxDB en Ubuntu 18.04 - Nociones.de*. (2018, March 14). Retrieved from <https://www.nociones.de/instalacion-y-primeros-pasos-con-influxdb-en-ubuntu-18-04>
- [10] *InfluxDB v1.8 Documentation*. (2020, October 23). Retrieved from <https://docs.influxdata.com/influxdb/v1.8>
- [11] *Códigos de estado de respuesta HTTP*. (2020, November 09). Retrieved from <https://developer.mozilla.org/es/docs/Web/HTTP/Status>