DevOps y análisis de performance automáticos

Álvaro Olmedo Rodríguez
Administración de redes y sistemas operativos

Manuel Jesús Mendoza Flores

7 de Enero de 2018

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-CompartirIgual 3.0 España de Creative Commons

FICHA DEL TRABAJO FINAL

Título del trabajo:	**DevOps y análisis de *performance* automáticos**
Nombre del autor:	**Álvaro Olmedo Rodríguez**
Nombre del consultor:	**Manuel Jesús Mendoza Flores**
Fecha de entrega (mm/aaaa):	01/2018
Área del Trabajo Final:	**Administración de redes y sistemas operativos**
Titulación:	*Grado Ingeniería Informática*

Resumen del Trabajo (máximo 250 palabras):

Se tratará de hacer hincapié en el campo de los tests de performance y análisis de rendimiento de sistemas productivos, más concretamente en el despliegue de estos. Todo ello dentro del marco de la nueva metodología o filosofía organizativa conocida como DevOps, centrándose, por tanto, mucho más en la parte de Operaciones dentro de dicha filosofía.

Esta filosofía, que trata de reducir la distancia entre los grupos de desarrollo y operaciones tiene algunos aspectos como la integración y el despliegue continuo, automatización,etc es por ello que este trabajo profundizará también en la automatización y despliegues continuos de lo necesario para que ciertos tests de performance o rendimiento puedan ser realizados.

Además se realizará una simulación práctica en la que se desplegarán algunas de las herramientas más utilizadas así como el despliegue de una webapp con todo lo necesario para que, desde el momento del despliegue, poder tener métricas del uso de la aplicación.

Abstract (in English, 250 words or less):

This work deeps into performance's tests and analysis in production environments, focused in the deployment. This will be showed from a DevOps' philosophy. So, this work will be closer to Ops than Devs part.

This new methodology try to reduce the distance between dev and ops groups/work/interests/... with concepts like continuos integration and deployment, automation, etc. It's for this that this work speaks about the continuos integration and deployment (automation) of the performance's tests requirements.

A demo with an automatic deployment of a webapp with the performance tests and their requirements (*scripts* and the it self test) in a ready to product/service system will be provided too. So, the webapp and their metrics will be deployed automatically.

Palabras clave (entre 4 y 8):

DevOps, métricas, operación, ELK, despliegue, automatización, performance, tests.

Índice

1. Introducción..1
   1.1 Contexto y justificación del Trabajo...1
   1.2 Objetivos del Trabajo...1
   1.3 Enfoque y método seguido...2
   1.4 Planificación del Trabajo...2
   1.5 Breve sumario de productos obtenidos..2
   1.6 Breve descripción de los otros capítulos de la memoria............................3
2. Situación actual..4
   2.1 Filosofía DevOps...4
   2.2 Pruebas de rendimiento...7
3. Descripción herramientas DevOps..9
   3.1 Automatización..9
   3.2 Infraestructura / Gestión de la configuración..9
   3.3 Monitorización...11
   3.4 Seguridad...11
   3.5 Tratamiento de Logs...12
4. Descripción de infraestructura...14
5. Demostración *pipeline* DevOps...20
   5.1 Despliegue infraestructura: jenkins y docker registry...............................20
   5.2 Despliegue infraestructura ELK...26
   5.3 Despliegue webapp...34
   5.4 Uso de la webapp y acceso a la información...42
5. Conclusiones..44
6. Glosario...46
7. Bibliografía..48
8. Anexos...50

Lista de figuras

Figura I - Intersección DevOps....4
Figura II - Tendencias DevOps 1....5
Figura III - Tendencias DevOps 2....5
Figura IV - Spotify Feature Teams....6
Figura V - Tabla periódica DevOps....13
Figura VI - Linux Interfaces....15
Figura VII - Diagrama funcional....16
Figura VIII - Diagrama de red....17
Figura IX - Pipeline ELK....18
Figura X - Pipeline Webapp....18
Figura XI - Acceso a jenkins inicial....24
Figura XII - Jenkins instalación plugins inicial....25
Figura XIII - Jenkins preparado....25
Figura XIV - Configuración job despliegue ELK 1....27
Figura XV - Configuración job despliegue ELK 2....27
Figura XVI - Ejecución job despliegue ELK 1....28
Figura XVII - Ejecución job despliegue ELK 2....28
Figura XVIII - Ejecución job comprobación ELK....31
Figura XIX - Kibana status....32
Figura XX - Kibana sin configuración....33
Figura XXI - Elasticsearch desplegado....33
Figura XXII - Ejecución job build webapp 1....36
Figura XXIII - Ejecución job build webapp 2....36
Figura XXIV - Ejecución job despliegue webapp....40
Figura XXV - Smoke Tests Webapp....41
Figura XXVI - Kibana con index pattern configurado....43
Figura XXVII - Objetos en Kibana importados....43

1. Introducción

1.1 Contexto y justificación del Trabajo

Actualmente no hay una guía clara dentro de la filosofía de DevOps de cubrir las necesidades sobre tests de *performance* de forma general y automática tal y como se tratan otras series de tareas dentro de esta metodología. Estos test y análisis de rendimiento tienen su necesidad en sistemas en producción, donde Operaciones necesita saber de una forma clara y rápida si un sistema está funcionando como debe.
Esto es, parece claro que un desarrollador debe de facilitar o colaborar en la construcción de un paquete o imagen de docker, o tener alineados con su código, para que se ejecuten cuando corresponda, ciertos tests unitarios (típicamente antes de realizar cualquier construcción y/o empaquetado de software), pero no parece estar igual de implantado el hecho de que tiene que realizarse una tarea para el despliegue del sistema necesario así como de los componentes necesarios para poder realizar en producción tests de performance.
En este momento, por lo general, son los grupos de operaciones los que implementan normalmente esta adición y, en la mayoría de los casos, de forma manual o poco automatizada, ya que estos tests no han sido incorporados dentro del ciclo de desarrollo y, por tanto, los equipos de operaciones los van implementando según van sucediéndose las necesidades.

En este trabajo se pretende analizar si es viable realizar los despliegues necesarios para que este tipo de tests sean desplegados y actualizados automáticamente, como cualquier otro componente software.

1.2 Objetivos del Trabajo

Ante la problemática existente en las metodologías ágiles actuales y la operación posterior de los sistemas con esas aplicaciones desarrolladas y desplegadas de forma ágil, se desea cubrir el gap existente en la operación respecto al análisis de logs (métricas de uso y performance de la plataforma).
Para ello se analizarán diferentes herramientas, preferiblemente *open source* que permitan poner en funcionamiento estos análisis de rendimiento o métricas durante el despliegue y que informarán, a posteriori, si el sistema está respondiendo adecuadamente.
Además se realizará un montaje de referencia o demostración que incluya el despliegue automático (*pipeline*) de una webapp junto con todo lo necesario para analizar en tiempo real el *performance* o buen funcionamiento de la aplicación. Este montaje servirá para realizar ciertas pruebas funcionales y valorar la solución aportada.

1.3 Enfoque y método seguido

Dado que ya existen diferentes herramientas en el mercado que nos permiten poder valorar si el sistema está respondiendo a los usuarios como debe, es decir, que nos permiten analizar el rendimiento se procederá a analizar brevemente las diferentes soluciones aplicables: tanto en lo que a software se refiere, principalmente *open source*, como en la arquitectura a desplegar.

1.4 Planificación del Trabajo

Junto a este documento se adjunta, en formato pdf, un diagrama de gantt con la planificación del TFG tratado. En términos generales se diferencian las tareas de desarrollo, en las que se incluyen tanto el análisis previo de las diferentes soluciones así como la automatización y el despliegue de la solución definitiva.

Se han de tener en cuenta, además, algunos de los riesgos que pueden hacer variar esta planificación:

- Como ya se ha descrito anteriormente el asunto de los *performance tests* dentro de la filosofía conocida como DevOps es algo muy novedoso o poco tratado, debido a esto, la información concreta que se puede obtener del tema es escasa y, por tanto, podría darse el caso de que diferentes puntos muy concretos de este trabajo se retrasasen debido a la falta de información.
- La demostración de dicho trabajo se realizará levantando servicios utilizando la tecnología de contenedores docker. Esta tecnología permite un rápido despliegue, pero, en algún caso, se requerirá de algún tipo de personalización del contenedor para simular perdidas de servicio o deterioro del mismo. Estos desarrollos no están contemplados en la planificación y su inclusión, dependiendo de la necesidad de trabajo para llevarlos a cabo, suponen un riesgo para el proyecto.
- La tecnología de contenedores docker, como ya se ha comentado, permite un rápido y cómodo despliegue, por el contrario, se desconoce hasta donde podría afectar el uso de esta tecnología según la arquitectura que se planteé para responder a la necesidad del proyecto. Y, dado el caso, se podría dar la necesidad de un cambio en cuanto a la tecnología a utilizar y, por tanto, una replanificación del proyecto.

1.5 Breve sumario de productos obtenidos

No se obtendrá un producto como tal; este trabajo tratará de demostrar la viabilidad de incluir en los desarrollos realizados por los equipos de alto rendimiento a través de metodologías ágiles, como DevOps, los elementos necesarios tanto a nivel de código como de despliegue para que dentro de los despliegues habituales se incluyan los tests de performance que mostrarán en entornos productivos el buen funcionamiento del sistema.

1.6 Breve descripción de los otros capítulos de la memoria

Situación actual, introducirá, brevemente, al lector en la filosofía DevOps así como en el estado actual de las diferentes pruebas y métricas.
Descripción herramientas DevOps, se hará una descripción de las herramientas y tecnologías más utilizadas actualmente en el mundo DevOps: para automatizar las diferentes fases de una *pipeline*, para soportar los diferentes procesos, para desplegar servicios, monitorizarlos,... (algunas de ellas serán utilizadas en este trabajo).
Descripción de infraestructura hará un repaso de las diferentes tecnologías utilizadas en la demostración; desde las que permiten el propio despliegue o puesta en servicio de la aplicación como las que permiten el análisis del sistema. Además, se hará una descripción de la infraestructura desplegada, tanto a nivel de servicios como de red.
Demostración *pipeline* DevOps describe detalladamente la *pipeline* que, desde los procesos previos (puesta en marcha de la infraestructura que soporta el despliegue así como el resto de la infraestructura, construcción y empaquetado del software,...) hasta el propio despliegue, pone en servicio la infraestructura y el servicio de la webapp, así como las métricas que se utilizarán para analizar el rendimiento.

2. Situación actual

2.1 Filosofía DevOps

Con la llegada de las metodologías ágiles de desarrollo y las necesidades de realizar integración y entrega continua (CI, *continuous integration* y CD, *continuous delivery*) aparece una nueva corriente organizativa llamada DevOps, que, en resumidas cuentas pretende aunar en un único equipo a perfiles muy separados en organizaciones más tradicionales como puedan ser los desarrolladores y los equipos de operaciones, todo ello con el objetivo final de realizar despliegues en entornos productivos de forma más regular. Haciendo nuevas entregas del software de una forma regular (semanalmente, diariamente o, incluso, varias veces al día) se consigue dotar al proceso del paso a producción de más seguridad o estabilidad y más eficiencia. Cuanto más regularmente se haga una tarea, en este caso un despliegue en producción, menos «doloroso» será. Para ello se requiere un nivel muy alto en la automatización de los procesos de compilación, empaquetado, pruebas, despliegues, *smoke tests,…*

Como resumen de todo esto, una imagen, "Devops" de Rajiv Pant licenciada bajo CC BY 3.0:

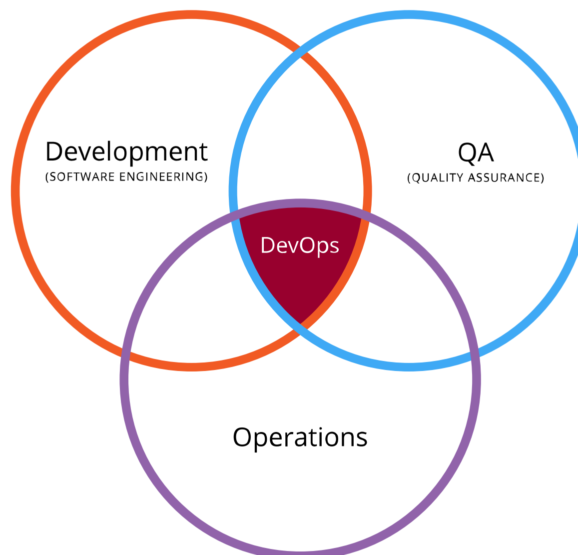

Figura I - Intersección DevOps

Según el estudio del estado DevOps realizado por Puppet y Dora[1] esta demostrado que organizaciones que utilizan metodologías ágiles de desarrollo y filosofía DevOps en su organización despliegan hasta 46 veces más frecuentemente que organizaciones más tradicionales, con tiempos de recuperación de fallos 96 veces más rápidos y con una tasa de fallo ante cambios 5 veces menor que organizaciones más tradicionales, no tan focalizadas al rendimiento. Los despliegues se hacen en producción mucho más a menudo (bajo demanda, varias veces al día) con un *Lead time* para los cambios de menos de una hora (tiempo que pasa desde que se hace un

cambio en el código hasta que se despliega en producción). Todos estos aspectos han sido mantenidos en valores similares a los de 2016 algo a destacar en aspectos como el ratio de fallos en cambios que, en el año anterior había subido notablemente (en gran medida debido al aumento en la frecuencia de los despliegues). Para visualizar estas tendencias una muestra de los gráficos en referencia a los aspectos comentados, obtenidos ambos del propio informe:

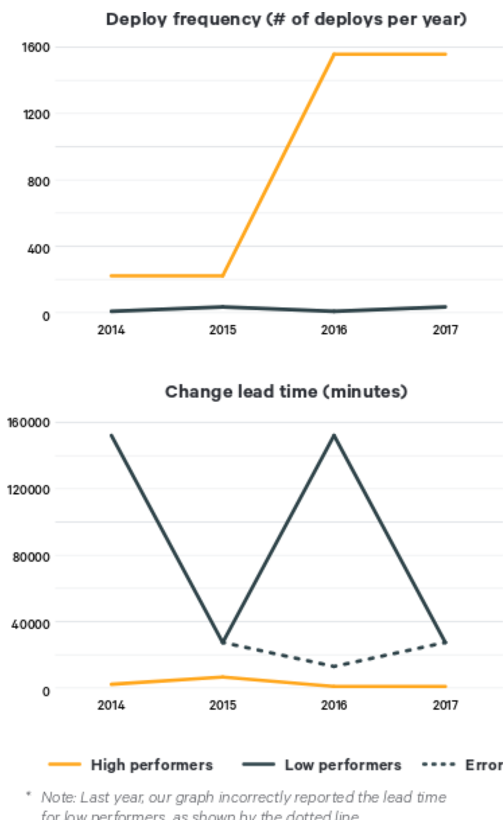

Figura II - Tendencias DevOps 1

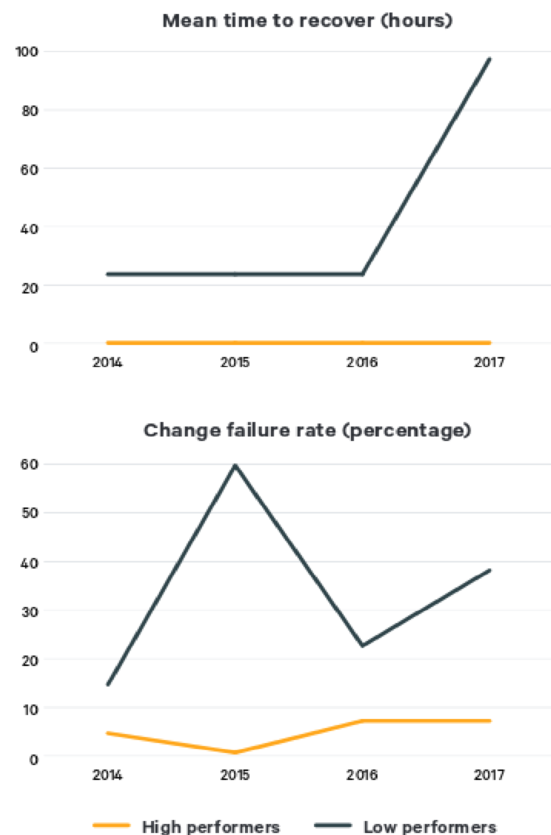

Figura III - Tendencias DevOps 2

Como últimas cifras del informe, indicar que los equipos IT de alto rendimiento (filosofía DevOps y metodologías de desarrollo ágiles) invierten un 21% menos de su tiempo en trabajos no planeados o repetir trabajos dando así hasta un 44% más de tiempo a nuevas *features*.

Otra innovación de esta filosofía es desechar el concepto de release de software ya que los despliegues se convierten en un proceso cíclico de mejora del servicio.
Como objetivo final, los despliegues más frecuentes se dan como respuesta a las necesidades u objetivos buscados desde la unidad de negocio:
- un rápido *time to market*, esto es, el tiempo que se necesita para poner en producción un nuevo desarrollo (ya sea el arreglo de un fallo, la incorporación de una nueva feature a un servicio ya existente o la implantación de un nuevo servicio)
- alta calidad
- Incremento de la efectividad de la organización

Entrando brevemente en lo que a la organización de los equipos se refiere, la filosofía o cultura DevOps propone diferentes aproximaciones[2].
Desde un tipo de organización más simple en el que exista una comunicación más directa y con mucho más contexto entre los equipos de desarrollo y operaciones, a una organización DevOps mucho más pura en el que se divide a todo el personal en equipos multidisciplinares, trabajando, cada uno de ellos, en una parte muy concreta de todo el proyecto, esto es, una nueva *feature* o servicio o uno ya existente y están formados por diferentes perfiles (desarrollo, QA, operaciones, seguridad...).

Esta organización en *squads* o *feature teams* permite que los diferentes roles tengan en todo momento un foco claro. Así un feature team es totalmente autosuficiente y totalmente responsable del servicio o parte de él que le ha sido encomendado, desde la codificación, hasta el despliegue y operación. Estos equipos pueden ser, incluso, temporales y, una vez la tarea ha finalizado las personas que formaban parte de él pueden formar un nuevo equipo en su totalidad o integrarse en otros equipos ya en funcionamiento.
Spotify lleva con esta organización de equipos desde 2014[3]:

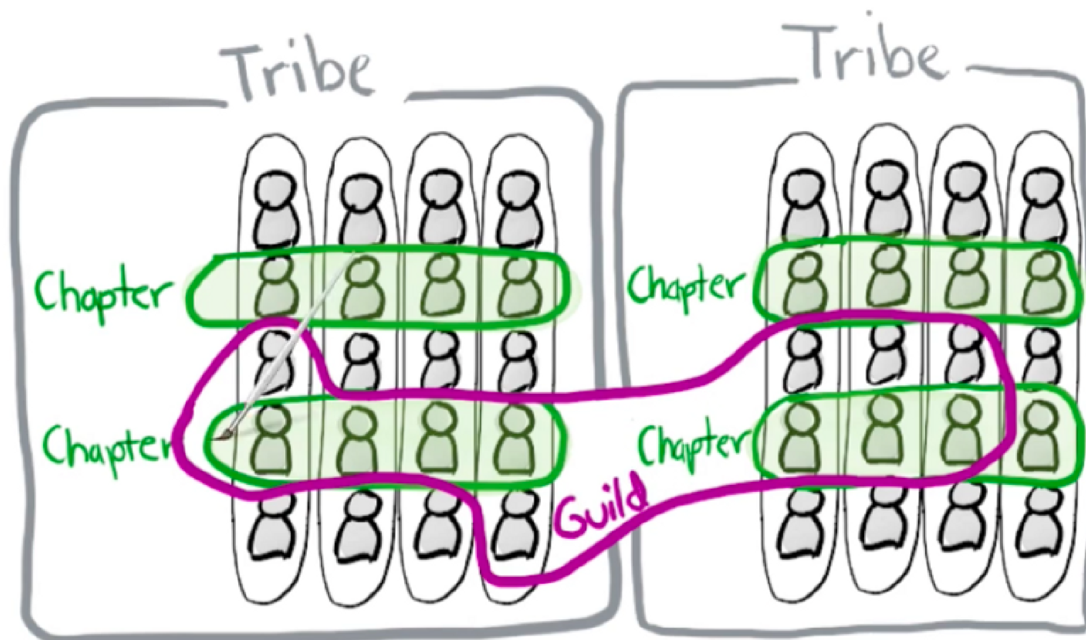

Figura IV - Spotify Feature Teams

Una solución intermedia, en la adopción de una filosofía DevOps, hasta conseguir esta organización pura de equipos multidisciplinares pasa por disponer de un equipo totalmente dedicado a "evangelizar" y ayudar con la adopción de la nueva forma de trabajo, normalmente comenzarán con las tareas más necesarias (empaquetamiento y versionado del software, automatización de despliegues, codificando la infraestructura,…) Por el tipo de tareas a realizar suele ser un equipo gente experimentada con una mezcla de habilidades o *skills* que van desde el uso de sistemas de control de versiones a la experiencia de implementación de *continuos delivery*.

Tradicionalmente los equipos de desarrollo y QA eran vistos de forma separada y, por tanto, se organizaban de forma separada: recursos diferentes, objetivos

diferentes, gestiones diferentes,… Aunque, desde afuera, equipos como operaciones veían a los equipos de desarrollo y QA mucho más alineados de lo que estaban, se pensaba en ellos como un único grupo buscando un objetivo común: entregar un software que funcione correctamente. Con la llegada de DevOps el modelo organizativo se adapta para, como ya se ha explicado, facilitar el despliegue de software de forma automatizada; como consecuencia los equipos desarrollo y operaciones se integran y alinean.
El equipo de QA, está encargado de automatizar todos los tests que hacen que el software pase por las distintas fases de despliegue en la *pipeline* y sea desplegado en los diferentes entornos. Es responsabilidad del equipo de desarrollo o un subconjunto de este (en algunos casos se entiende como un rol totalmente separado) conocido como Release Engineer, RSE o ingeniero DevOps facilitar las herramientas para que esos despliegues sean automáticos, chequeables, pueda realizarse rollback o marcha atrás,…

Por último, para reseñar un caso de éxito con la aplicación de la filosofía de DevOps, es muy conocido el caso del equipo de desarrollo de los firmwares de los productos LaserJet de HP[4].

2.2 Pruebas de rendimiento

En primer lugar ubicar las pruebas de rendimiento, dentro de la ingeniería del software, como las pruebas que determinan lo rápido que un sistema realiza una tarea en ciertas condiciones. Además, otros aspectos de la calidad como la escalabilidad, la fiabilidad y el uso de recursos pueden ser valorados con este tipo de tests. Dentro de todo el conjunto de pruebas de rendimiento podemos diferenciar entre:
- Prueba de carga: test más de sencillo que permite analizar el sistema o aplicación ante una demanda de rendimiento esperada, monitorizando elementos como la base de datos o servidor de aplicaciones situados en otras capas se puede llegar a determinar los cuellos de botella de la arquitectura diseñada e implementada.
- Prueba de estrés: destinada a buscar el limite de la aplicación o sistema, consiste, básicamente en realizar pruebas de cargas sucesivas aumentando la carga (peticiones, usuarios,…) en cada ejecución hasta romper el sistema. Con esta prueba se determina la solidez del desarrollo y si el rendimiento será suficientemente en caso de una carga superior a la esperada.
- Prueba de estabilidad: consiste en una prueba de carga durante un tiempo continuado, en busca de posibles fallos por el uso esperado y continuado de la aplicación. Durante la ejecución de los *soak testing* es habitual encontrar, por ejemplo, fugas de memoria en la aplicación, esto es, memoria RAM que no es liberada.
- Prueba de picos: busca determinar el comportamiento de la aplicación o sistema ante picos de uso, tanto aumentando como bajando drásticamente el número de usuarios o peticiones.

Con todo esto, es necesario remarcar, que el entorno o entornos donde se realizan este tipo de pruebas deben ser lo más parecidos al entorno productivo

y completamente aislado de otro tipo de tareas de desarrollo, integración con terceros,…
Las pruebas de rendimiento o *test performance* con la implantación de filosofías de despliegue mucho más continuo como DevOps han tenido una rápida evolución. Además de los habituales test funcionales en etapas tempranas de desarrollo para detectar posibles bugs en el uso o funcionamiento de la aplicación es importante realizar test de rendimiento que permitan dar un feedback adecuado al equipo de desarrollo sobre el *performance* de su aplicación. Esta retroalimentación es clave realizar en etapas tempranas del desarrollo ya que está ampliamente demostrado que arreglar un bug en producción tiene un coste altamente superior a hacerlo en entornos previos: se estima que arreglar un bug una vez el producto ha sido lanzado es 30 veces superior a solucionarlo durante la fase de diseño y arquitectura o 3 veces superior a solucionarlo durante la fase de tests de integración [5].

Para llevar a cabo unos test de rendimiento adecuados en un entorno de desarrollo ágil con CI en funcionamiento lo más habitual es realizar los siguientes tests:
- *smoke tests* en cada despliegue,
- tests de carga completos una vez al día (típicamente en el build nocturno o *Nightly Build)*
- y, por último, al final de cada sprint, test de stress sobre la plataforma desarrollada.

Obviamente todos estos tests han de ser reflejados en código ya que se requiere que su ejecución sea automática y desatendida.

En la mayoría de los casos la validación de muchos de estos test supone analizar los logs de ejecución de los diferentes sistemas y/o aplicaciones. Estos logs son diferentes entre sí en cuanto a estructura normalmente y, si además se tiene en cuenta sus tamaños, hacen de la automatización algo totalmente imprescindible para su análisis [6].

Para medir correctamente este rendimiento o performance se hace necesario definir unos Indicadores Clave de Rendimiento o *key performance indicators* (KPIs), estos indicadores pueden a su vez ser divididos en indicadores orientados al servicio y orientados a la eficiencia [7]. El primer tipo hace referencia a cómo se está dando el servicio al usuario (disponibilidad, tiempo de respuesta,…) el segundo, en cambio se refiere al uso de la infraestructura (rendimiento, capacidad,…).

Con todo esto se puede concluir que, además de los diferentes tests que se realizar en entornos previos a producción, son necesarios ciertos análisis en el entorno productivos. Estos análisis, definidos en función de los KPIs, son claves para, por un lado, analizar el funcionamiento de la aplicación en tiempo real por parte de operaciones y, por otro lado, como ya se ha comentado, dar retroalimentación al equipo de desarrollo sobre el performance de la aplicación. El despliegue automático de este tipo de análisis, conocidos como tests de *performance*, son los que se tratarán de llevar a cabo dentro de un marco de trabajo DevOps en este trabajo.

3. Descripción herramientas DevOps

Si bien el ecosistema de herramientas relacionadas con DevOps es cada vez más extenso ya que aparecen diariamente herramientas asociadas a tecnologías recién aparecidas que cubren más y mejor las diferentes necesidades se realizará un repaso rápido por diferentes categorías describiendo con cierto detalle algunas de las herramientas más utilizadas y simplemente nombrando algunas otras.

3.1 Automatización

La ejecución de trabajos automatizados y su orquestación se realiza con **jenkins**[8]. Es la piedra angular sobre la que se sustentan cualquier proceso DevOps: cualquier trabajo que vaya a ser ejecutado en numerosas ocasiones ha de ser configurado para ser ejecutado a través de jenkins ya que, además de la información de la propia ejecución aporta un gran valor en cuanto a metainformación se refiere: quién ha ejecutado el trabajo, por qué (en caso de ejecutarse por un trigger), cómo (con qué parámetros), cuándo, cuánto tiempo ha tomado la ejecución, el resultado de la misma,etc... Este servidor de automatización está desarrollado en java, es gratuito y es *open source*, dispone de cientos de plugins para poder ampliar su funcionalidad e integrarse con diferentes sistemas.
Hace poco más de un año se ha lanzado la versión 2, que como característica fundamental incorpora Jenkins Pipeline[9], un set de plugins predefinidos con el objetivo de permitir la configuración de *pipelines* de *Continuos Delivery* de forma sencilla, esto es, facilitar el despliegue del software desde el repositorio de código a los diferentes sistemas para que pueda ser utilizado por clientes y usuarios.
En el capítulo 4 se describirá más detalladamente el despliegue realizado de este servicio para la demostración y durante todo el capítulo 5 se describirá el uso de dicha herramienta para el caso concreto de la demostración que se realizará.

Otros sistemas como **RunDeck** (más centrado en la automatización) o **GoCD** (más centrado en las *pipelines* y *continuos deployment*) llevan tiempo algún tiempo intentando restar cuota de mercado a jenkins, pero se ha convertido en el standard de facto, la herramienta DevOps por excelencia, con un gran apoyo por parte de la comunidad.

3.2 Infraestructura / Gestión de la configuración

Vagrant es una herramienta de Hashicorp que permite levantar entornos ligeros, enfocados al desarrollo de una manera sencilla y rápida. Permite reproducir el entorno de desarrollo localmente con el foco puesto en la reproducibilidad y en la automatización. Se podría decir que es un “frontend” ya que, básicamente, levanta máquinas o recursos virtuales en diferentes sistemas (virtualbox, openstack, vmware, docker,…) a través de ficheros de

código. El hecho de que permita definir estos recursos en ficheros de código en lugar de en ficheros propietarios (OVA, VMDK,…) dota a vagrant de gran sentido dentro de la filosofía DevOps al permitir definir como código parte de la infraestructura (*Infraestructure as Code*, IaC)[10]. Es una herramienta gratuita, *open source* y desarrollada en ruby.

Chef y **Puppet** son los referentes en cuanto a automatización de la infraestructura (*IaC*), se trata de herramientas de gestión de la configuración. Existen algunas diferencias entre ambos, pero, en una visión general, se podrían comparar en cuanto a herramientas que posibilitan la codificación de la infraestructura y los despliegues sobre estas definiendo el estado final que se desea para los diferentes servidores. Ambas son gratuitas (chef con ciertas limitaciones de modo de uso), *open source* y desarrolladas en ruby.

Muy similar a las anteriores existe **ansible**, pero con la posibilidad de orquestar los despliegues desde el propio controlador, esto es, el nodo que ejecuta el código ansible (accediendo a los nodos por ssh), algo que las anteriores herramientas no pueden hacer por si solas y por lo que se podría considerar ansible como un herramienta de orquestación, aunque el uso habitual de ansible es utilizarlo como una herramienta de gestión de la configuración.
Tras la compra de la empresa que da soporte comercial (Ansible inc) por parte de Red Hat, promete convertirse en el próximo standard de la industria. Está escrita en python, es gratuita y *open source*.
Mencionar que para la orquestación anteriormente citada de Ansible, puppet ofrece un servicio de pago llamado Mcollective (Marionette Collective) que permite la orquestación de despliegues con puppet.

Los contenedores han cambiado la forma en la que el software es empaquetado, distribuido y desplegado, así como las infraestructuras que los hacen correr, dentro de toda la diversidad de tecnologías que permiten esta “virtualización ligera”, **docker** está siendo ampliamente aceptado por la industria frente a otras opciones como **LXC** o **rkt**. Es gratuito, *open source* y está escrito en Go.
Relacionado con la tecnología de contenedores se encuentra **Kubernetes**, desarrollado por Google para su propio uso, es un orquestador de contenedores, permitiendo una gestión más cómoda de estos, permitiendo cosas como autoescalado, mantener activos cierto número de contenedores,…
Al igual que docker está escrito en Go y es gratuito y *open source* (fue donado por Google a la Cloud Native Computing Foundation, parte de la Linux Foundation).
Otras opciones de orquestación de contenedores son **OpenShift** de RedHat y que básicamente se trata de un sistema Kubernetes recubierto con numeroso plugins o sistemas adicionales de gestión, **Nomad** de Hashicorp, **Rancher** o el propio **Swarm** de Docker.
Todos estos orquestadores permiten la definición de los servicios a través de ficheros con diferentes formatos, lo que facilita la distribución de los despliegues. Una solución para definir despliegues de servicios pero que no requieran de tanta infraestructura como para utilizar un orquestador es utilizar **docker-compose**[11], que permite la definición de servicios con múltiples contenedores, redes, etc. mediante ficheros de configuración en formato yaml.

En el capítulo 4 se describirá más detalladamente tanto la tecnología docker como el uso de docker-compose para realizar los despliegues y durante todo el capítulo 5 se describirá el trabajo tanto con los contenedores como los despliegues propiamente dichos para cada uno de los servicios desplegados y que son utilizados durante la presentación.

3.3 Monitorización

Nagios y Zabbix son las herramienta de monitorización clásicas que han conseguido evolucionar adaptándose a la filosofía de los equipos de trabajo DevOps, compitiendo ambos por convertirse en el standard de facto para la monitorización de la infraestructura. Ambos son gratuitos y *open source* y escritos principalmente en C.

Prometheus está tomando cierta relevancia, unido a **Grafana** permite la visualización de métricas almacenadas en su base de datos como series de tiempo. Es gratuita, *open source* y está escrita en Go.

Monit es usado como solución local para la monitorización de sistemas y recuperación de errores, de una forma sencilla se puede configurar tanto la monitorización como la posible recuperación en caso de fallo. Se trata de una aplicación gratuita, *open source* y escrita en C.

3.4 Seguridad

Durante la fase de compilación o empaquetado es recomendable correr ciertas herramientas de análisis como **Dependency Check** de OWASP, este scanner analiza los componentes *open source* utilizados buscando vulnerabilidades en la base de datos NIST. Su integración con jenkins y herramientas de construcción como maven, ant o grandle es total. Se trata de un scanner gratuito y *open source*.

Implantar, a nivel de sistema de control de versiones, chequeos para el código que se va a subir, es una buena política ya sea de forma automática con mecanismos como los pre-commit hooks o revisiones manuales a través Merge o Pull Requests.

Los **Controles Proactivos** recomendados por OWASP (OWASP *Proactive Controls*) es una iniciativa de 2016 donde se describen los 10 puntos que todo desarrollador y arquitecto deberían incluir y revisar en cada uno de sus proyectos. [12]

Snort es un IPS (Intrusion Prevention System) *open source* muy popular desde hace muchos años, su potencial radica en la capacidad de analizar tráfico en tiempo real. La utilidad de este tipo de aplicaciones en entornos productivos está ampliamente demostrada.

Muchas otras herramientas ayudan a la ejecución de Pen Testing de forma automática como una parte más del *Continuos Delivery* (gauntlt), securizar por defecto los diferentes servidores (hardening.io), test unitarios, análisis de vulnerabilidades (SonarQube),… Todo este interés por la integración de diferentes steps relacionados con la seguridad dentro de una *pipeline* típica DevOps ya sea de Integración o de Despliegue Continuo es conocido como DevSecOps o DevOpsSec [13].

3.5 Tratamiento de Logs

Logstash de Elastic forma parte de la pila ELK (Elasticsearch, Logstash y Kibana) y se encarga de la recepción centralizada, transformación y posterior envío para ser almacenada (típicamente a elasticsearch) de diversa información. Se trata de una herramienta desarrollada en java, *open source* y gratuita. Se hablará de ella más adelante en capítulos posteriores, ya que es una de las tecnologías utilizadas en la Demo de *pipeline*.

Splunk herramienta muy popular y similar en funcionalidad al *stack* ELK pero de pago.

Herramientas en la nube como papertrail (rsyslog en la nube) o logz.io (básicamente un ELK en la nube al que redirigir nuestros logs con filebeat) realizan funciones parecidas con la ventaja de no ser necesario infraestructura para el despliegue ni mantenimiento de la mismas.

En el capítulo 4 se describirá más detalladamente el despliegue realizado de este servicio ELK para la demostración, así como la función de cada uno de los componentes, en el capítulo 5.2 se describirá el despliegue realizado de la pila ELK para su uso en la demostración que se realizará, el uso de la herramienta es descrito en el capítulo 5.4.

Por último, hacer referencia a esta Tabla Periódica de las herramientas DevOps realizada por XebiaLabs (https://xebialabs.com/periodic-table-of-devops-tools/) que se ha convertido, gracias a sus continuas actualizaciones en una guía de herramientas de referencia o, ante nuevas necesidades, fuente de información para descubrir nuevas:

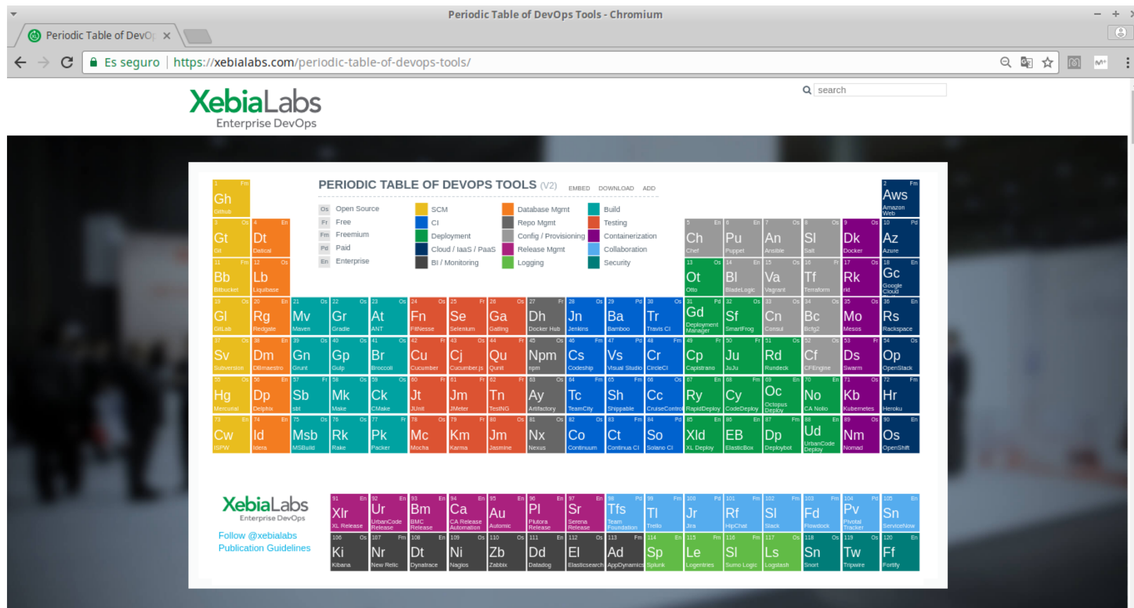

Figura V - Tabla periódica DevOps

4. Descripción de infraestructura

Este trabajo será acompañado de una demostración de una *pipeline* típica en un despliegue siguiendo la cultura DevOps, en este caso se ha optado por simular un aplicación web.

Debido a la importancia que está adquiriendo en el mundo DevOps la tecnología de contenedores, se decide desplegar cada uno de los componentes de esta infraestructura como un contenedor Docker, todos corriendo en el mismo host anfitrión. Además existen otros motivos que apoyan la decisión de utilizar docker para desplegar esta infraestructura frente a otras opciones de virtualización o despliegue como podrían ser vagrant, virtualbox o la propia nube (Amazon Web Services, Azure, Google Cloud Engine,…):

- Requerimientos de recursos no muy elevada: se corre toda la infraestructura en un ordenador personal con 8GB de RAM y un procesador de i5-6200U a 2.30GHz con un total de 4 núcleos.
- Costes de uso reducidos: instalar y usar docker no requiere del uso de licencias.
- Inmediatez en los despliegues
- Facilidad para simular entornos reales con la asignación de Ips y diferentes redes de forma clara y transparente: esto es una mala práctica en docker, ya que salvo en situaciones muy concretas, se ha de desestimar la asignación de Ips a contenedores y su acceso a sus servicios a través de esta (normalmente se hace uso de nombres, services discoveries u otras tecnologías más avanzadas) pero nos permite simular o hacer más entendible la arquitectura a visiones o aproximaciones más “clásicas”.

Docker es una tecnología que permite construir, distribuir y correr software a través de contenedores. Dota de una capa de abstracción y automatización de virtualización al sistema operativo Linux. Utiliza características del núcleo de dicho sistema operativo tales como *cgroups* y *namespaces* para que los contenedores (sistemas virtualizados) se mantengan independientes pero dentro de una única instancia de Linux (un único kernel) lo que provee de mayor ligereza que la clásica virtualización con máquinas virtuales que arrancan su propio SO. Hace uso de la biblioteca libcontainer (en primeras versiones se utiliza LXC, LinuX Container) para acceder a estas y otras funciones del kernel que permiten esta virtualización. [14]

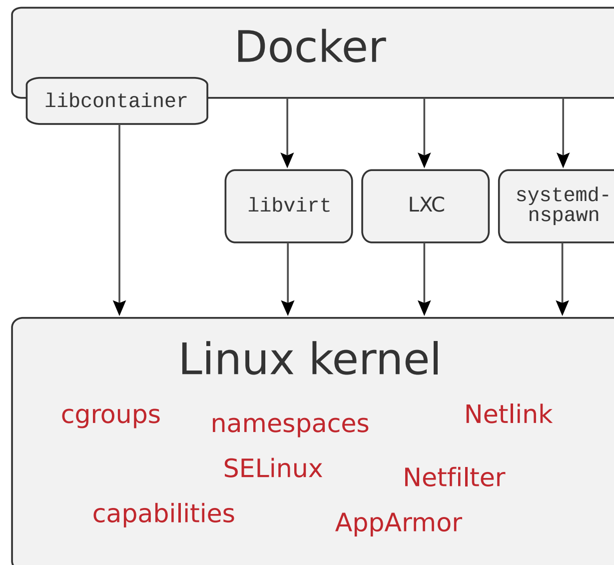

Figura VI - Linux Interfaces

En esta demostración veremos los diferentes pasos que se realizan en una *pipeline* de *Continuos Deployment* típica trabajando con docker:
- se construirá una imagen docker a través de un dockerfile[15]
- se pushea la imagen construida a un registry local para que esté disponible para los hosts que quieran correr un contenedor de dicha imagen [16]
- se corren las imágenes disponibles en el registry en diferentes contenedores, orquestados a través de docker-compose [11]

La arquitectura de la aplicación, propiamente dicha, se ha tratado de reducir, construyendo una infraestructura muy básica y sencilla, estaría formada por dos nodos en los que correría la aplicación web y, por delante de ellos, recibiendo las peticiones, un nodo que hace de balanceador y proxy inverso a través de un servicio nginx.
Nginx es un servidor web *open source* que ha ganado bastante popularidad en los últimos años gracias a su ligereza y alto rendimiento, puede actuar como proxy inverso. Realiza funciones también de balanceo de carga así como otro tipo de servicios tales como compresión, autenticación.
Respecto a la aplicación web, es muy sencilla, desarrollada en python y servida con Flask, apenas muestra un mensaje “Hola UOC!” y el identificador del contenedor donde está corriendo. El código está alojado en github (https://github.com/alvarolmedo/TFG-webapp) y, además del propio código de la aplicación, se aloja en el repositorio el Dockerfile junto con el fichero de librerías python a instalar que permiten la construcción de la imagen de docker.

Además del propio servicio de la webapp, formado por el proxy inverso y los dos nodos con el aplicativo se desplegaran diversos servicios que permitirán completar la demostración de la *pipeline*:
- Servicio ELK (ElasticSearch, Logstash y Kibana)[17] encargado de recibir los logs (logstash), almacenarlos (elasticseach) y permitir su consulta y visualización (kibana). Se elige ELK por ser *open source* y gratuita y por ser altamente aceptada en la industria.

- Servicio filebeat[17]. Es un accesorio a ELK, se trata de un servicio encargado de hacer llegar los logs al servicio ELK, típicamente, en una arquitectura más clásica, corre como un proceso (bastante ligero) en la máquina que posee logs y que se desean traspasar al stack ELK. En esta infraestructura concreta no sería necesario su uso, puesto que logstash podría acceder directamente a logs de la webapp (nginx), pero se ha preferido desplegar para, al igual que en el caso del setup de red, simular o acercar esta infraestructura a entornos más clásicos.

- Servicio jenkins encargado de orquestar los diferentes jobs para completar la *pipeline*. En él se definen las diferentes tareas o jobs de los que consta la *pipeline* a través de diferentes jobs (cuya configuración, mediante ficheros xml están disponibles en el repositorio https://github.com/alvarolmedo/TFG-deploy.

- Servicio registry. Un docker registry[16] es un servidor donde se almacenan imágenes docker que, posteriormente, correrán los contenedores.

A continuación se muestra un diagrama que representa los diferentes elementos funcionales así como los diferentes tipos de interacciones o comunicaciones que existen entre ellos:

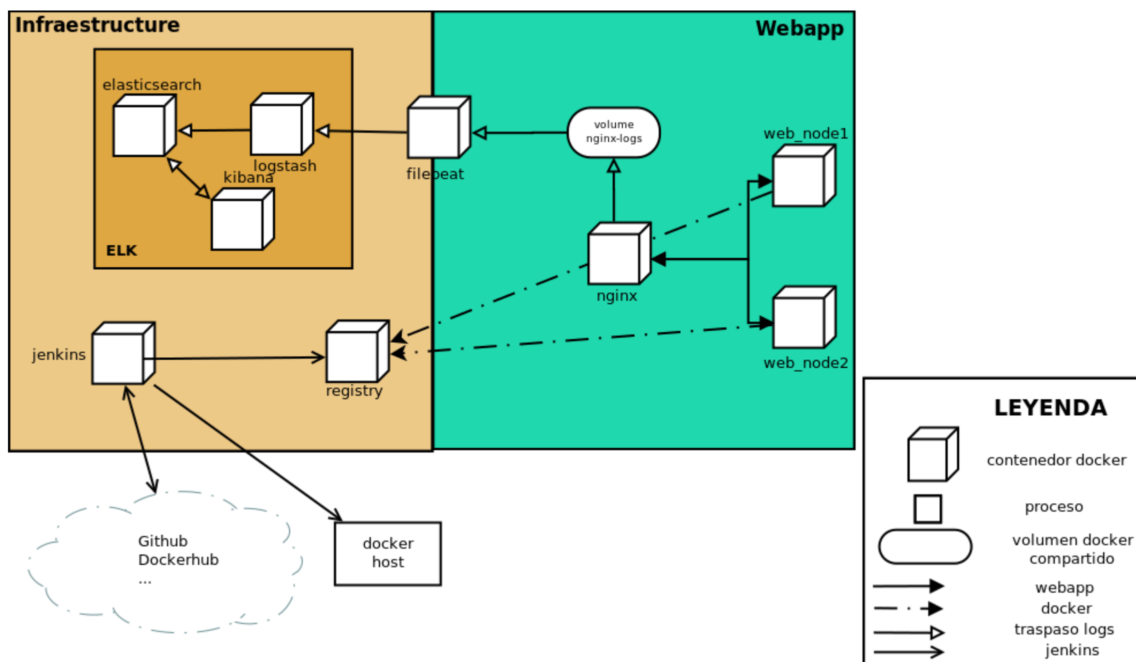

Figura VII - Diagrama funcional

En este diagrama se representa también el volumen compartido entre dos contenedores para el traspaso de la información (los logs) entre el servicio nginx y filebeat, encargado de hacérselo llegar, a su vez, a la pila ELK.
Otros contenedores importan volúmenes (elk, registry y jenkins para asegurar un almacenamiento persistente, filebeat y nginx por temas de configuración) pero, dado que son locales y no realizan ningún tipo de función más allá de la utilidad para sus propios servicios, no se ha considerado representarlos en este diagrama funcional ya que serán detallados más adelante.

A nivel de red se ha separado lo que sería el propio servicio web, dentro de la red private, con lo que serían servicios de infraestructura (registry y jenkins) desplegados en la red oam (*operation and maintenance*, operación y mantenimiento), por último el servicio ELK dispone de su propia red. En el siguiente diagrama se puede obtener una visión más detallada de cada componente y su direccionamiento:

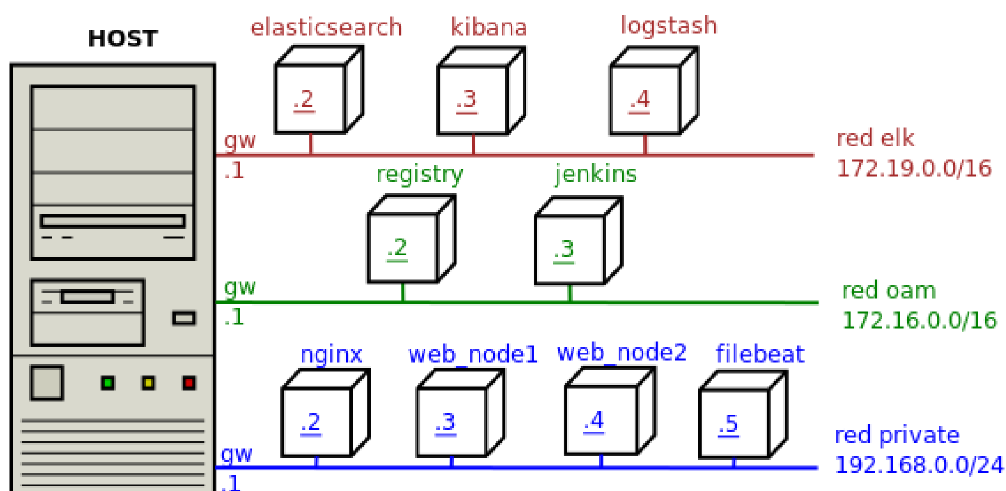

Figura VIII - Diagrama de red

Toda esta infraestructura, ha sido desplegada utilizando diferentes docker-compose[11], una herramienta oficial de docker que permite, de una manera sencilla definir aplicaciones o sistemas multi-contenedores a través de un fichero yaml y hacerlos correr o gestionar sus modificaciones con la propia herramienta.

El fichero yaml que define toda esta arquitectura así como otros ficheros necesarios (configuración de nginx para hacer de proxy inverso, configuración de filebeat,..) están almacenados en cada uno de los repositorios de github dedicados al proyecto:
https://github.com/alvarolmedo/TFG-deploy incluye todo lo relacionado con la infraestructura de O&M (*operation and maintenance*): jenkins y la configuración de los jobs, docker, registry,...
https://github.com/alvarolmedo/docker-elk incluye la infraestructura ELK: el build de las imágenes con configuración específica y deploy de las mismas y smoke test.
https://github.com/alvarolmedo/TFG-webapp incluye todo lo relacionado con la webbapp: build, deploy (configuración necesaria incluída) y smoke test.

En referencia a jenkins, indicar que en el capítulo siguiente se describirán detalladamente los jobs, pero como visión general, describir los diferentes tipos de jobs, al igual que existirían en cualquier *pipeline* típica: *build*, *deploy* y *smoke test*. Y, además, hay que diferenciar entre la *pipeline* de ELK (en la que no hay job de build, ya que no es necesario construir las imágenes) y la de la webapp:

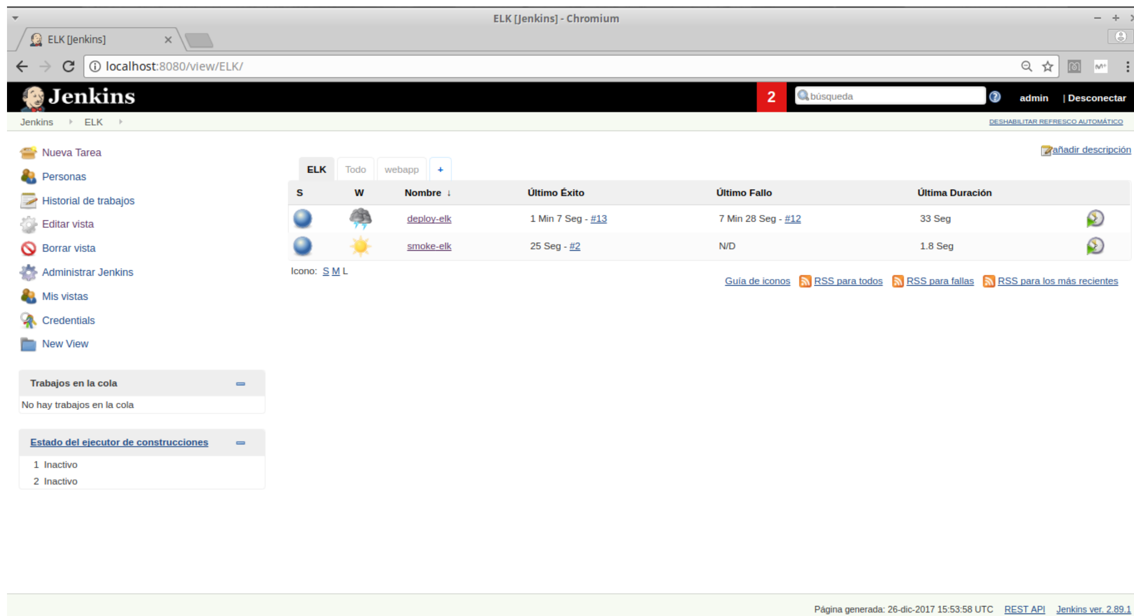

Figura IX - Pipeline ELK

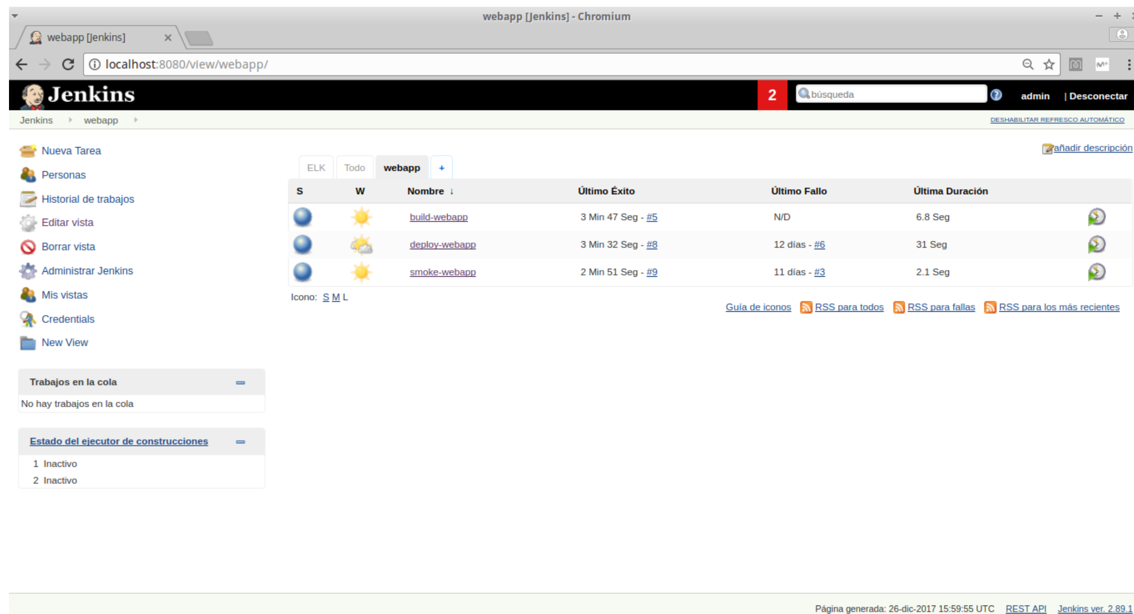

Figura X - Pipeline Webapp

Los jobs de jenkins se pueden consultar en el capítulo 5, donde se describe tanto su configuración como su ejecución y funcionalidad.

Por último, para describir brevemente la pila ELK, indicar la funcionalidad de cada uno de los componentes:

- Logstash es el encargado de recibir la información y, en caso necesario, formatear la información a través de plantillas, estableciendo campos, tipos de datos para estos,…. Una vez procesada, la información ha de ser traspasada para ser almacenada de forma persistente.
- Elasticsearch es el encargado de almacenar la información, una vez procesada por logstash, pero su funcionalidad es mucho mayor ya que no es una simple base de datos. El motor de búsquedas y analíticas provee búsquedas en tiempo real, funciones de autocompletado en búsquedas. Almacena la información en documentos formateados en JSON.
- Kibana permite visualizar la información almacenada en elasticsearch. Es necesario especificar qué índices almacenados en kibana pueden ser analizados, a través de un *index-pattern* o patrón de índices. A partir de ahí, la información puede ser consultada directamente, en crudo, e ir filtrando a través de búsquedas o se pueden realizar diferentes tablas, gráficos, mapas,… todas estas visualizaciones pueden ser juntadas en *dashboards*.

Se valoró y analizó la posibilidad de integrar el filebeat con el servicio logz.io (http://logz.io) básicamente, un ELK en la nube. En un principio era mucho más sencillo y cómodo que levantar la infraestructura localmente: la integración apenas supuso modificar el fichero filebeat y generar una nueva imagen que contuviera el certificado que cifra las comunicaciones con log.zio [18]. Una vez entregados los logs a logz.io, la visualización era mucho más rápida ya que logz.io incluye muchos plugins con búsquedas, visualizaciones y dashboards, lo que logz.io ha denominado ELK apps [19]. Cuando se intentó generar estos objetos dinámicamente, a través de ficheros json que los definan se observa que Logz.io únicamente da acceso al logstash vía para introducir información (logs) y vía web a kibana, para visualizar. Es imposible acceder a la API del stack ELK y, al no tener acceso directo a elasticsearch y, por tanto, no poder introducir ni index-pattern ni objetos como búsquedas, visualizaciones y dashboards de manera automática (tal y como se verá más adelante que hacemos en el entorno local) se descarta el uso de logz.io. Este problema se solventó con el despliegue en local de la pila ELK.

5. Demostración *pipeline* DevOps

En primer lugar se desplegará la infraestructura formada por el registry de docker y el jenkins por un lado, y, por otro, el stack ELK. Posteriormente se desplegarán los contenedores relacionados con la webapp.

Como paso previo en el host que correrá toda la infraestructura de docker se hacen las siguientes modificaciones para que el jenkins y el docker registry que se van a desplegar funcionen como se espera:
- Esto resolverá el problema que supone ejecutar docker desde un contenedor docker (los jobs de jenkins ejecutarán docker y docker-compose para construir y desplegar todos los servicios) atacando al demonio docker del anfitrión (Esta técnica es conocida como *docker in docker* y está ampliamente documentada y tratada[20]) y que estos contenedores que se despliegan usen volúmenes con ficheros que, originalmente, no están en el host anfitrión:

```
# ln -s $PWD/jenkins_home /var/jenkins_home
```

- Por un lado se introduce una entrada en el /etc/hosts que haga una conversión entre la IP y el nombre “registryuoc” como se va a identificar el registry desplegado:

```
# grep registryuoc /etc/hosts
172.16.0.1  registryuoc
```

- Por otro lado se configura el registry que se va a desplegar como insecure (http en lugar de https), es necesario, tras esto, que se reinicie el servicio docker:

```
# cat /etc/docker/daemon.json
{ "insecure-registries":["registryuoc:5050"] }
```

Obviamente, además, es necesario tener instalado docker y docker-compose:

```
# dpkg -l | grep -E "^ii  docker"
ii  docker-ce                                              17.09.1~ce-0~ubuntu
amd64        Docker: the open-source application container engine
ii  docker-compose                                               1.8.0-2~16.04.1
all               Punctual, lightweight development environments using
Docker
```

5.1 Despliegue infraestructura: jenkins y docker registry

El primer paso, antes de desplegar los contenedores jenkins y registry, alojado en el repositorio de git https://github.com/alvarolmedo/TFG-deploy tenemos el *script* y el Dockerfile asociado para construir la imagen de jenkins que utilizaremos posteriormente:

```
$ ls jenkinsuoc/
build.sh  Dockerfile
```

El Dockerfile es utilizado para la construcción de imágenes de docker, usando el comando docker build, toda la información al respecto puede ser encontrada en la web oficial de docker [15]. El contenido del Dockerfile es el siguiente:

```
FROM jenkins/jenkins:lts

# Installing docker via apt
USER root
RUN groupadd -g 135 docker
RUN apt-get update && apt-get install -y apt-transport-https ca-certificates wget software-properties-common sudo
RUN wget https://download.docker.com/linux/debian/gpg
RUN echo "jenkins ALL=(ALL:ALL) NOPASSWD:ALL" | tee -a /etc/sudoers
RUN apt-key add gpg
RUN rm -f gpg
RUN echo "deb [arch=amd64] https://download.docker.com/linux/debian $(lsb_release -cs) stable" | tee -a /etc/apt/sources.list.d/docker.list
RUN apt-get update && apt-get install -y docker-ce
RUN usermod -aG docker jenkins
RUN touch /var/run/docker.sock
RUN chown root:jenkins /var/run/docker.sock
# drop back to the regular jenkins user - good practice
USER jenkins
```

Básicamente, se utiliza la imagen oficial de jenkins pero se realizan las tareas oportunas para instalar docker community edition siguiendo el procedimiento oficial de docker [21] (la imagen de jenkins está basada en debian) y las recomendaciones de jenkins respecto a su imagen[22], además se realizan las tareas oportunas para que el usuario de sistema jenkins, el cual ejecuta el propio servicio docker dentro del contenedor, tenga permisos para poder ejecutar docker.

El *script* build.sh se encarga de construir la imagen de jenkins personalizada, construida a partir del Dockerfile descrito anteriormente, esta imagen será nombrada como jenkinsuoc, además, el *script* se encarga de limpiar la imagen jenkins oficial, para evitar confusiones (debido al almacenamiento de docker por capas, con esta limpieza, realmente, no se está ahorrando espacio):

```
#!/bin/bash

export IMAGE="jenkinsuoc"
export TAG="latest"

docker build -t ${IMAGE}:${TAG} ./
echo "Image Built: $IMAGE Tag: $TAG"
docker rmi jenkins/jenkins:lts
```

Ejecutando el *script*, obtenemos la imagen jenkinsuoc:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
$ cd jenkinsuoc
$ ./build.sh
Sending build context to Docker daemon  3.584kB
Step 1/13 : FROM jenkins/jenkins:lts
lts: Pulling from jenkins/jenkins
```

```
3e17c6eae66c: Pull complete
fdfb54153de7: Pull complete
(…)
Step 13/13 : USER jenkins
 ---> Running in 942236f7998a
 ---> f450e68c6fe4
Removing intermediate container 942236f7998a
Successfully built f450e68c6fe4
Successfully tagged jenkinsuoc:latest
Image Built: jenkinsuoc Tag: latest
Untagged: jenkins/jenkins:lts
Untagged:
jenkins/jenkins@sha256:643f147981c83f4654a1e94cb74f6aa4100da97951bb6c2
3f649f9338689cc56
$ docker images
REPOSITORY           TAG                    IMAGE ID            CREATED
SIZE
jenkinsuoc          latest                f450e68c6fe4        2 minutes
ago       1.24GB
```

Una vez construida la imagen, se pueden lanzar los contenedores, usando el fichero docker-compose.yml también disponible en el repositorio:

```
version: '2'

services:

  jenkins:
    container_name: jenkins
    image: jenkinsuoc:latest
    restart: always
    ports:
      - "8080:8080"
      - "50000:50000"
    volumes:
      - ./jenkins_home:/var/jenkins_home
      - /var/run/docker.sock:/var/run/docker.sock
    privileged: true
    networks:
      oam:
        ipv4_address: 172.16.0.3

  registry:
    container_name: registryuoc
    image: registry
    restart: always
    ports:
      - "5050:5000"
    volumes:
      - ./registry:/var/lib/registry
    networks:
      oam:
        ipv4_address: 172.16.0.2

networks:
  oam:
    driver: "bridge"
    ipam:
```

```
    config:
      - subnet: 172.16.0.0/16
        gateway: 172.16.0.1
```

En este fichero se describen los dos contenedores, en cada uno de ellos se configura la IP, dentro de la red oam así como el mapeo de puertos 8080 y 50000 para el contenedor de jenkins y el 5050 para el registry, que mapeará internamente con el 5000 (el cambio es debido a que el 5000 también es usado por el stack ELK, concretamente logstash). Respecto a los volumes, tanto el del registry como el jenkins_home son utilizados para que los datos generados en dichos contenedores persistan, siendo almacenados en el host anfitrión, el otro volumen del contenedor de jenkins, relacionado con el UNIX socket file del servicio docker viene dado por la necesidad del propio contenedor de ejecutar docker y de hacerlo en el anfitrión, en lugar de en el propio contenedor. La ya comentada técnica *docker in docker*[20].

Antes de ejecutar el docker-compose del repositorio https://github.com/alvarolmedo/TFG-deploy comprobamos el estado actual de docker:

- Ningún contenedor corriendo:

```
$ docker ps -a
CONTAINER ID          IMAGE                    COMMAND                 CREATED
STATUS                PORTS                    NAMES
```

- Imagen jenkinsuoc previamente generada disponible:

```
$ docker images
REPOSITORY            TAG                      IMAGE ID                CREATED
SIZE
jenkinsuoc            latest                   f450e68c6fe4            2 minutes
ago        1.24GB
```

- Únicamente las redes predefinidas disponibles:

```
$ docker network list
NETWORK ID            NAME                     DRIVER                  SCOPE
5c11d2ab15e8          bridge                   bridge                  local
357d61d04002          host                     host                    local
2719e724eedd          none                     null                    local
```

Al ejecutar docker-compose se genera los contenedores y redes descritos en el fichero y que han sido comentados anteriormente:

```
$ docker-compose up -d
Creating network "deploy_oam" with driver "bridge"
Pulling registry (registry:latest)...
latest: Pulling from library/registry
ab7e51e37a18: Pull complete
c8ad8919ce25: Pull complete
5808405bc62f: Pull complete
f6000d7b276c: Pull complete
f792fdcd8ff6: Pull complete
Digest:
sha256:9d295999d330eba2552f9c78c9f59828af5c9a9c15a3fbd1351df03eaad04c6
a
Status: Downloaded newer image for registry:latest
Creating registryuoc
Creating jenkins
```

Da como resultado la disponibilidad tanto del docker registry como del servicio jenkins:

```
$ docker network list
NETWORK ID          NAME                DRIVER              SCOPE
5c11d2ab15e8        bridge              bridge              local
25b1e46d1c52        deploy_oam          bridge              local
357d61d04002        host                host                local
2719e724eedd        none                null                local
$ docker ps
CONTAINER  ID              IMAGE                           COMMAND
CREATED                 STATUS                                  PORTS
NAMES
dfbf1a442e71         jenkinsuoc:latest   "/bin/tini -- /usr..."   About
a  minute  ago           Up  57  seconds           0.0.0.0:8080->8080/tcp,
0.0.0.0:50000->50000/tcp    jenkins
a131f61404f1         registry            "/entrypoint.sh /e..."   About
a minute ago    Up 57 seconds                      0.0.0.0:5050->5000/tcp
registryuoc
```

El servicio de jenkins ya es accesible a través del navegador web:

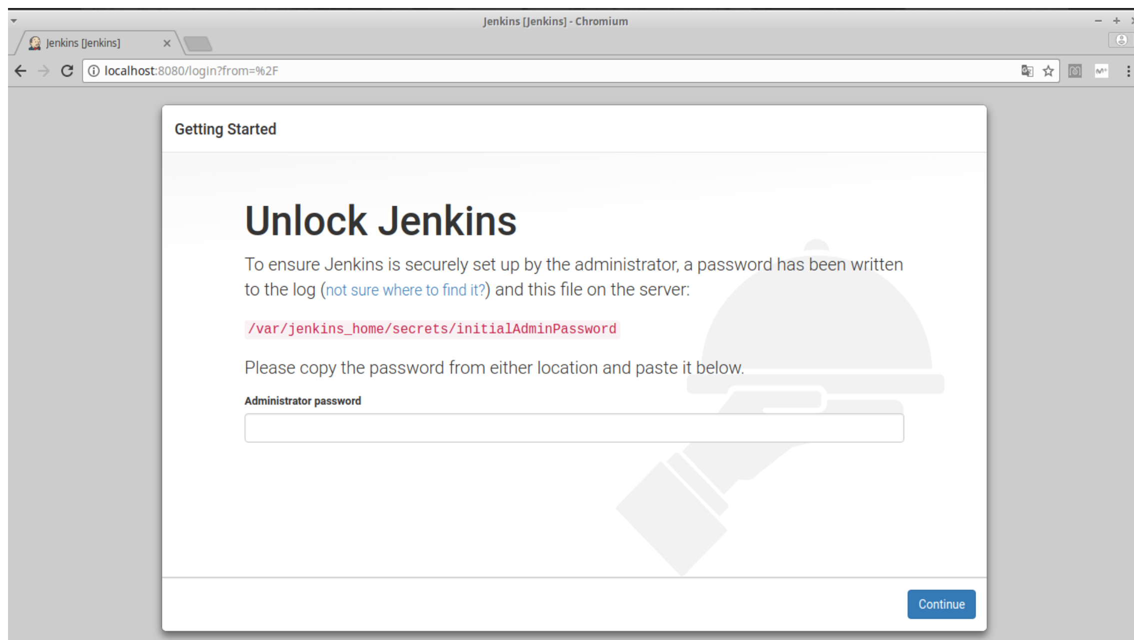

Figura XI - Acceso a jenkins inicial

Desbloqueado jenkins, se instala con los plugins habituales:

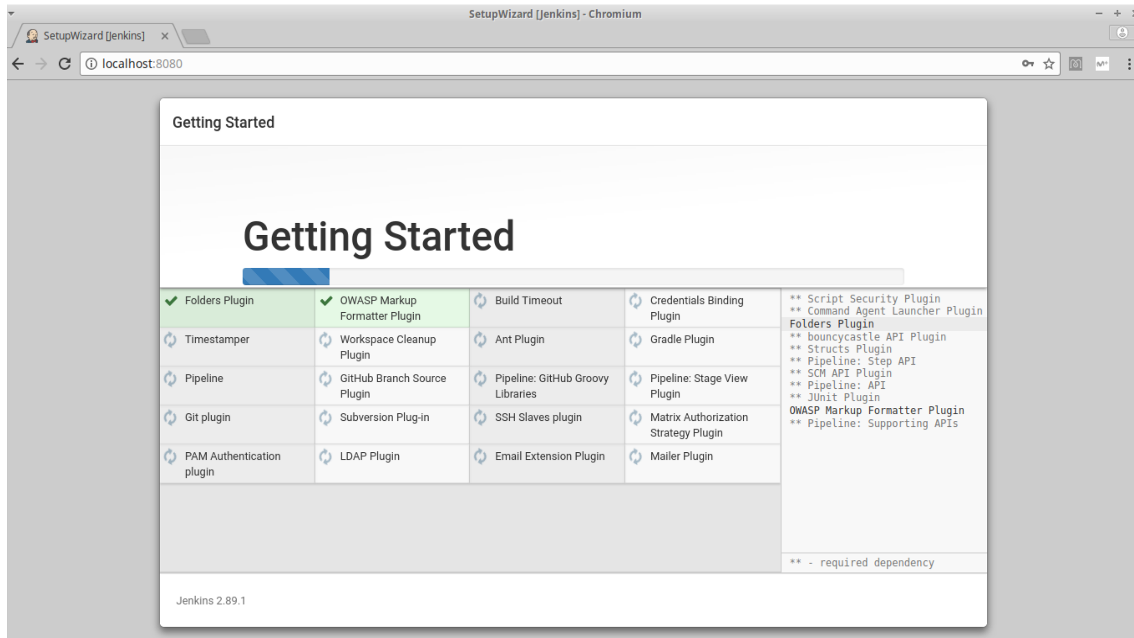

Figura XII - Jenkins instalación plugins inicial

Una vez instalados los plugins y creado el usuario administrador, jenkins está listo para usarse:

Figura XIII - Jenkins preparado

En este punto se describirán las diferentes jobs que se crean, en el repositorio https://github.com/alvarolmedo/TFG-deploy se incluyen los respectivos ficheros de configuración de cada job:

```
$ ls example_jenkins_config/jobs/*/config.xml
example_jenkins_config/jobs/build-webapp/config.xml
example_jenkins_config/jobs/deploy-elk/config.xml
example_jenkins_config/jobs/deploy-webapp/config.xml
```

```
example_jenkins_config/jobs/smoke-elk/config.xml
example_jenkins_config/jobs/smoke-webapp/config.xml
```

5.2 Despliegue infraestructura ELK

Se hará uso del jenkins recién instalado para realizar el despliegue del servicio ELK (ElasticSearch, Logstash y Kibana) para ello se crea un job que utilizará el repositorio https://github.com/alvarolmedo/docker-elk/ y el fichero docker-compose que contiene.

Este repositorio parte del repositorio oficial[23] que elastic, compañía detrás de ELK, facilita para el despliegue de su sistema utilizando docker. A este código se le han añadido algunas configuraciones para adaptarlo a las necesidades de este entorno:
- se añade un volumen al contenedor de elasticsearch para que la información que almacena sea persistente.
- se configura logstash para recibir y parsear correctamente los logs de nginx desde filebeat.
Ambos cambios son fácilmente localizables desde el listado de commits de la rama principal (master)[24] del repositorio git ya indicado.
- se añade un *script* para realizar un *smoke test* tras el despliegue para comprobar que el servicio está funcionando correctamente.

El job de despliegue del stack ELK está configurado con los siguientes pasos: clonado del repo, la ejecución del despliegue y la ejecución posterior del job de comprobación:

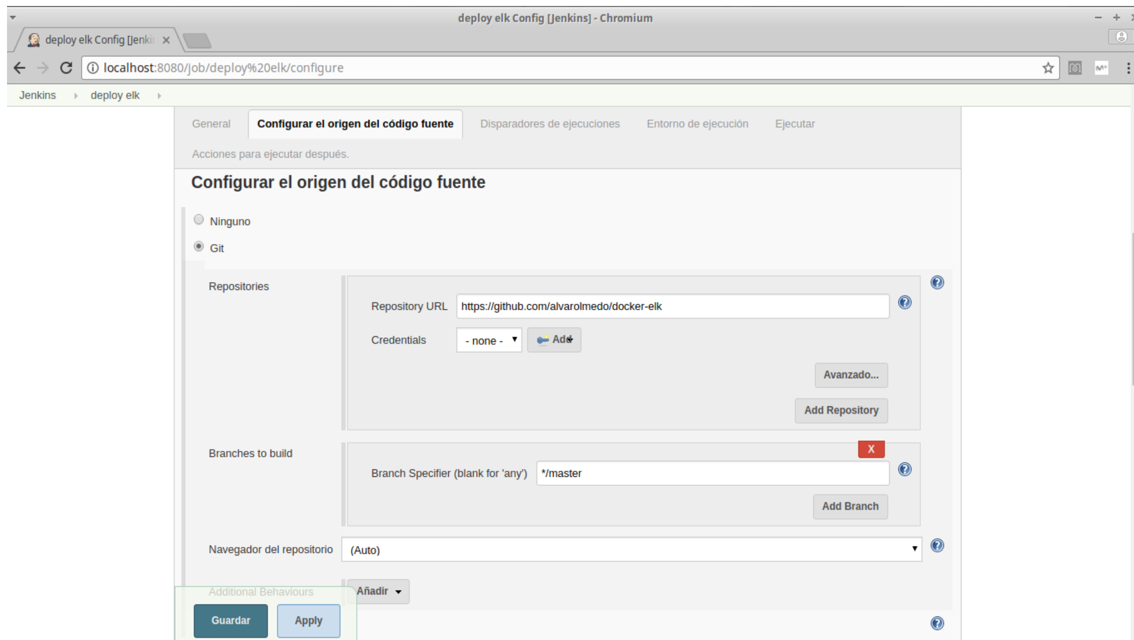

Figura XIV - Configuración job despliegue ELK 1

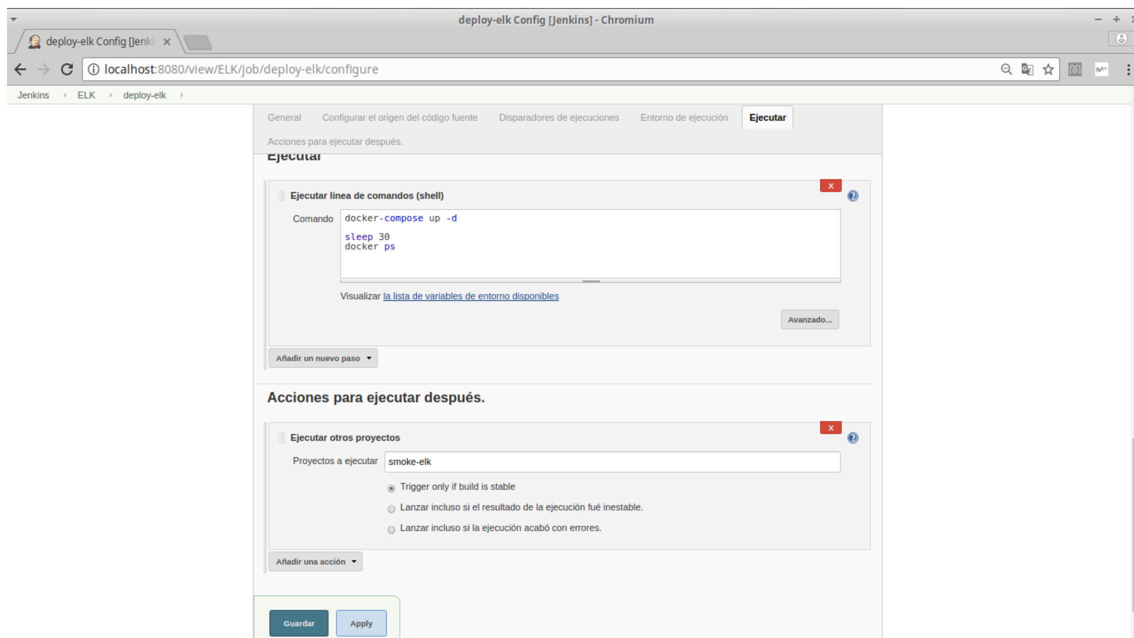

Figura XV - Configuración job despliegue ELK 2

La ejecución muestra, efectivamente, el clonado del repo y la ejecución posterior del despliegue:

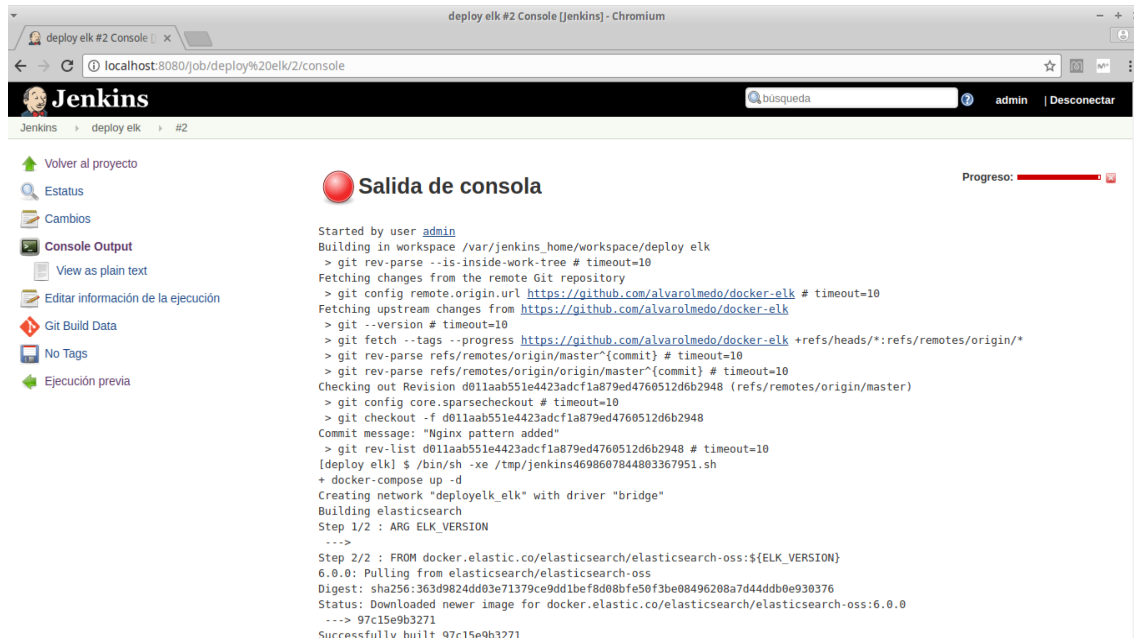

Figura XVI - Ejecución job despliegue ELK 1

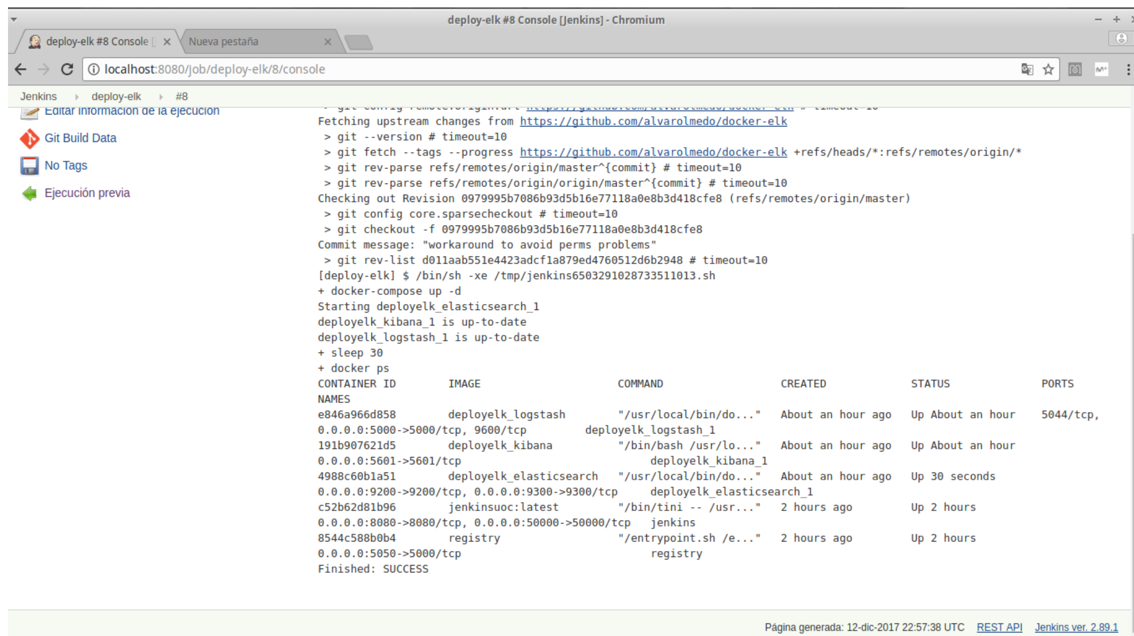

Figura XVII - Ejecución job despliegue ELK 2

Para desplegar todo el servicio, básicamente, se utiliza este fichero docker-compose.yml:

```
version: '2'

services:

  elasticsearch:
    build:
      context: elasticsearch/
      args:
        ELK_VERSION: $ELK_VERSION
    volumes:
      -
./elasticsearch/config/elasticsearch.yml:/usr/share/elasticsearch/conf
ig/elasticsearch.yml
      - ./elastic-data:/usr/share/elasticsearch/data
    ports:
      - "9200:9200"
      - "9300:9300"
    environment:
      ES_JAVA_OPTS: "-Xmx256m -Xms256m"
    networks:
      - elk

  logstash:
    build:
      context: logstash/
      args:
        ELK_VERSION: $ELK_VERSION
    volumes:
      -
./logstash/config/logstash.yml:/usr/share/logstash/config/logstash.yml
      - ./logstash/pipeline:/usr/share/logstash/pipeline
    ports:
      - "5000:5000"
    environment:
      LS_JAVA_OPTS: "-Xmx256m -Xms256m"
    networks:
      - elk
    depends_on:
      - elasticsearch

  kibana:
    build:
      context: kibana/
      args:
        ELK_VERSION: $ELK_VERSION
    volumes:
      - ./kibana/config/:/usr/share/kibana/config
    ports:
      - "5601:5601"
    networks:
      - elk
    depends_on:
      - elasticsearch

networks:
```

```
  elk:
    driver: bridge
    ipam:
     config:
       - subnet: 172.19.0.0/16
         gateway: 172.19.0.1
```

Básicamente, hace correr tres contenedores con sus respectivas imágenes, construyéndolas si es necesario, y expone una serie de puertos para cada uno de ellos además de hacer la configuración accesible (para poder ser modificada como en el caso de logstash) y los datos persistentes mediante volúmenes[25].

En el caso de logstash si ha sido necesario adaptar la configuración para que, al recibir los logs de nginx, pueda parsearlos correctamente. El logstash.conf queda de la siguiente forma (el output o salida no ha sido necesario modificarlo):

```
input {
      beats {
            port => 5000
                codec => plain {
                    charset => "US-ASCII"
                }
      }
}

## Add your filters / logstash plugins configuration here
filter {
  grok {
    patterns_dir => ["/opt/logstash/pattern"]
    match => { "message" => "%{NGINXACCESS}" }
  }
}

output {
      elasticsearch {
            hosts => "elasticsearch:9200"
      }
}
```

Donde se define, por un lado el endpoint para los beats (puerto 5000) así como el tipo de datos (plain) y codificación (US-ASCII) esperados y, por otro, el parseo de los campos, propiamente dicho, mediante grok y un patrón también generado y almacenado en el repositorio git (logstash/pattern/nginx) que mapea cada campo con el tipo de dato[26][27][28]:

```
NGUSERNAME [a-zA-Z\.\@\-\+_%]+
NGUSER %{NGUSERNAME}
NGINXACCESS %{IPORHOST:clientip} %{NGUSER:ident} %{NGUSER:auth} \[%{HTTPDATE:timestamp}\] "%{WORD:verb} %{URIPATHPARAM:request} HTTP/%{NUMBER:httpversion}" %{NUMBER:response} (?:%{NUMBER:bytes:float}|-) (?:"(?:%{URI:referrer}|-)"|%{QS:referrer}) %{QS:agent}
```

Una vez el job ha finalizado comprobamos la ejecución que se realiza, automáticamente, del *smoke test* que verifica que el despliegue ha finalizado correctamente:

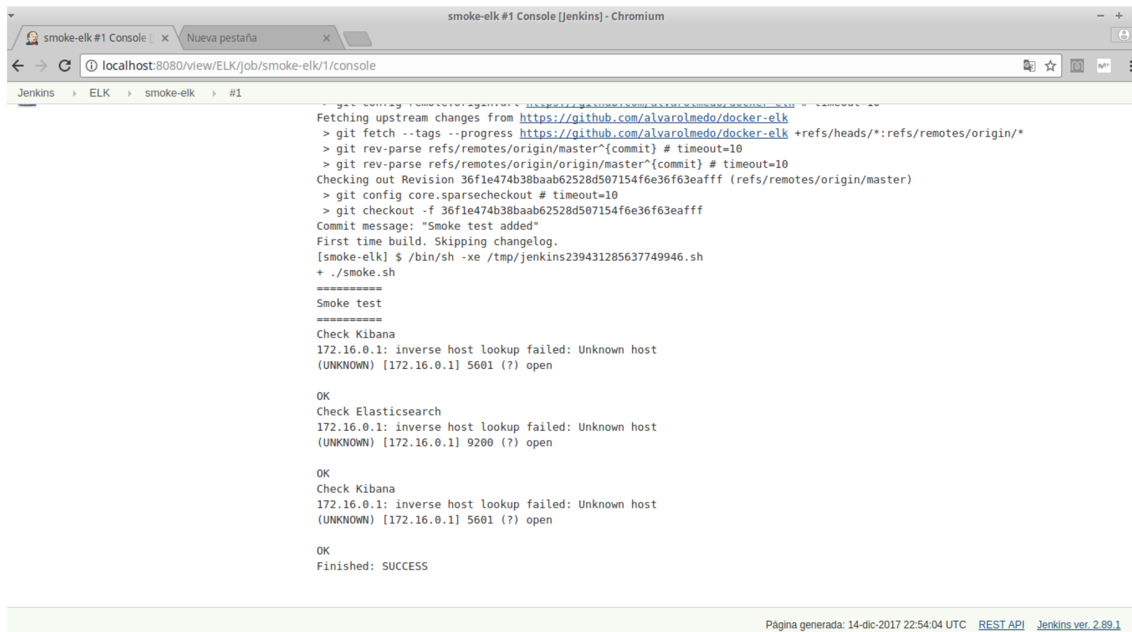

Figura XVIII - Ejecución job comprobación ELK

La configuración de este job es trivial, y fácilmente deducible de la propia ejecución: se clona el repo http://github.com/alvarolmedo/docker-elk y ejecuta el *script* smoke.sh:

```
#!/bin/bash

echo "=========="
echo "Smoke test"
echo "=========="

echo Check Kibana
nc -vz 172.16.0.1 5601
if [ $? == 0 ];then
	echo
	echo OK
else
	echo
	echo Fail in Kibana
	exit 1
fi


echo Check Elasticsearch
nc -vz 172.16.0.1 9200
if [ $? == 0 ];then
	echo
	echo OK
else
	echo
	echo Fail in Elasticsearch
	exit 1
fi
```

```
echo Check Logstash
nc -vz 172.16.0.1 5000
if [ $? == 0 ];then
	echo
	echo OK
else
	echo
	echo Fail in Logstash
	exit 1
fi
```

Adicionalmente, se verifica el funcionamiento correcto de kibana accediendo a la url de estado de Kibana:

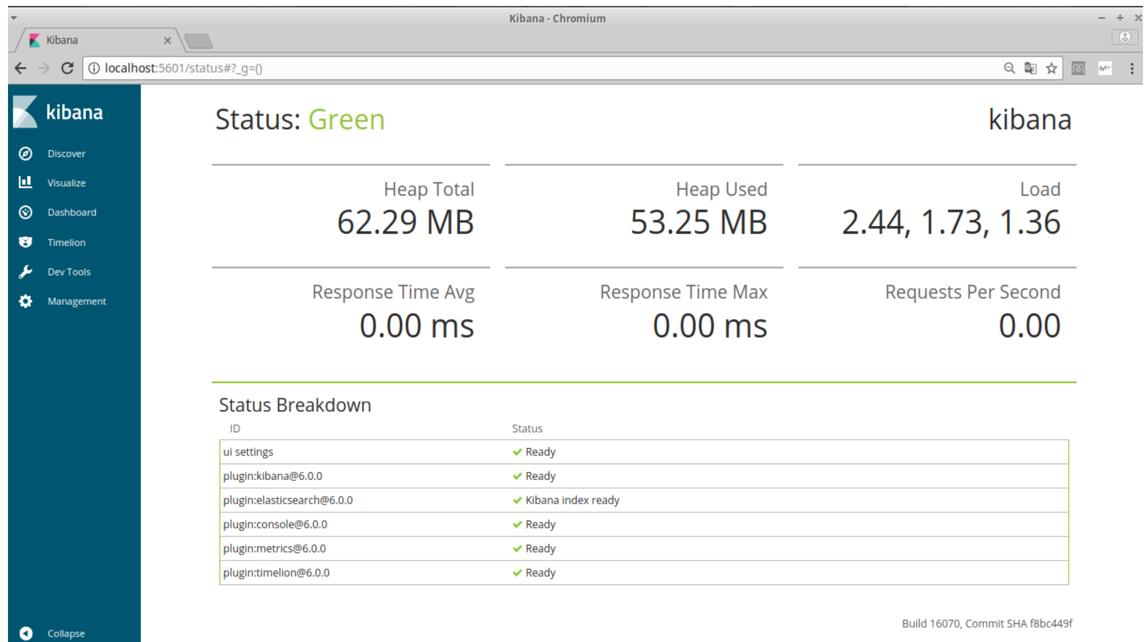

Figura XIX - Kibana status

El acceso a Kibana muestra que el servicio está disponible y que todo esta funcionando correctamente, aunque no tiene configurado ningún elemento: ni lo más básico (un index-pattern por defecto [31]) ni elementos de visualización de los datos más avanzandos como pueden ser visualizaciones, dashboards,...:

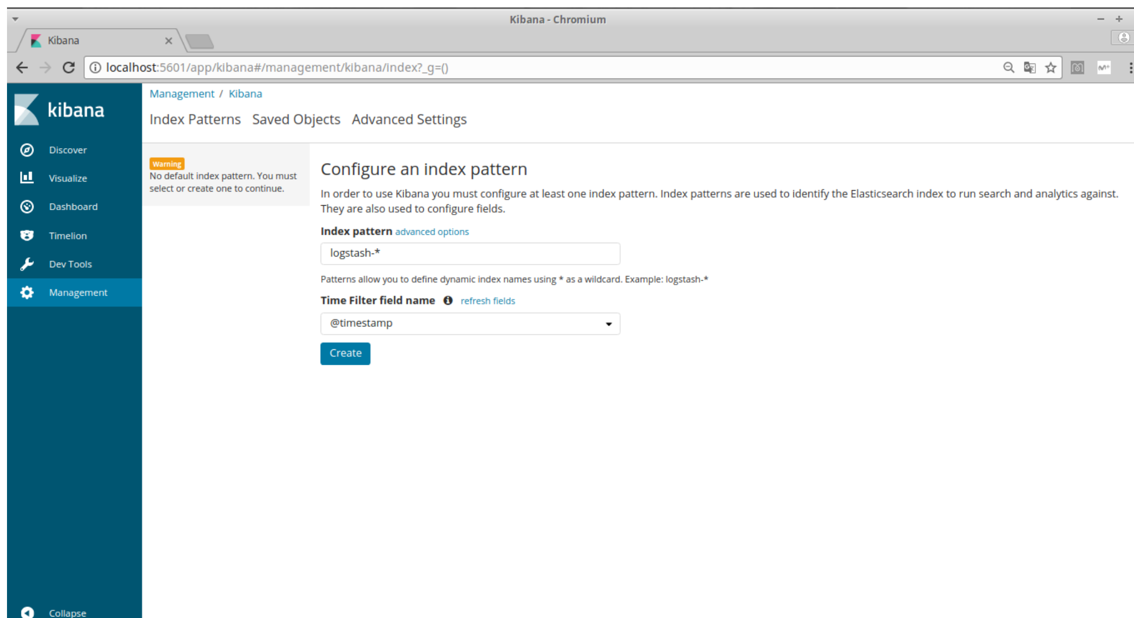

Figura XX - Kibana sin configuración

Más adelante, será configurado de forma automática para mostrar todas las métricas y poder realizar los análisis de rendimiento, alcanzando así el objetivo de este trabajo.

De igual modo se comprueba el funcionamiento de elasticsearch:

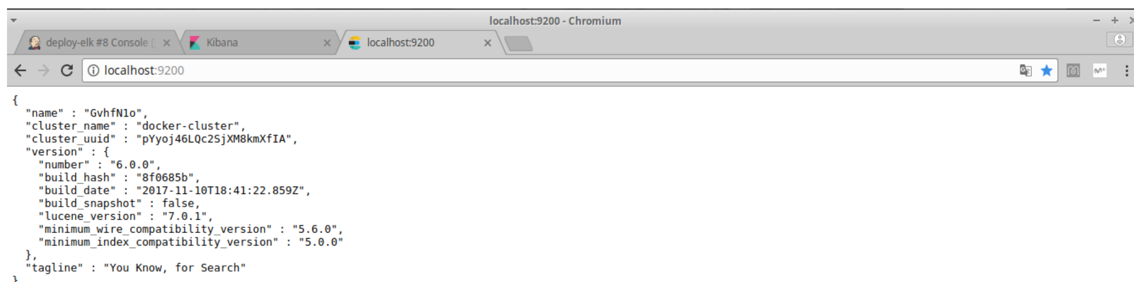

Figura XXI - Elasticsearch desplegado

5.3 Despliegue webapp

Una vez la infraestructura (jenkins y registry y ELK) está desplegada correctamente se puede realizar el despliegue webapp y sus “accesorios” (nginx y filebeat). Básicamente nginx, como ya se ha comentado, realizará tareas de proxy inverso, así los logs estarán centralizados en cuanto a tener un único punto de entrada. Por otro lado filebeat, en esta arquitectura podría haber sido eliminado y que logstash obtuviera del volumen de nginx los logs directamente, pero en entornos productivos, donde nginx corre como un proceso en máquinas, (normalmente de grandes recursos para poder dar servicio como proxy inverso al mayor número de usuarios) filebeat corre como un proceso en ellas para poder enviar los logs a ELK.

En este caso, la *pipeline* consiste en 3 jobs: build, deploy y *smoke test*. Cada uno de ellos ejecutará el siguiente job si ha finalizado correctamente. El contenido de todos estos jobs provienen del mismo repositorio http://github.com/alvarolmedo/TFG-webapp.

El primer paso, el build, se encarga de construir las imágenes de docker usando un Dockerfile que pondrá en ejecución la webapp, subirlas al docker registry (repositorio de imágenes) disponible y limpiarlas localmente de la máquina de integración continua que ejecuta jenkins (estas tareas de limpieza son muy importantes para que la máquina de IC no se vaya degradando con el paso del tiempo y la ejecución de los jobs).
La configuración del job, será por tanto, similar a la ya mostrada para otros jobs: clonado del repo, ejecución de la tarea concreta (en este caso, el *script* build.sh que se encarga de realizar las tareas que acabamos de mencionar) y ejecutar el siguiente job si todo ha ido correctamente (deploy).
El script build.sh realiza las tareas comentadas:

```
#!/bin/bash

export IMAGE="webapp"
export TAG="latest"
export REGISTRY="registryuoc:5050"

docker build -t ${IMAGE}:${TAG} ./
echo "============================================================="
echo "Image Built: $IMAGE Tag: $TAG"
echo "============================================================="
docker tag ${IMAGE}:${TAG} ${REGISTRY}/${IMAGE}:${TAG}
docker push ${REGISTRY}/${IMAGE}:${TAG}
echo "============================================================="
echo "Image $IMAGE pushed to $REGISTRY with tag $TAG"
echo "============================================================="
docker rmi ${IMAGE}:${TAG}
docker rmi python:3.6.0-alpine
```

El Dockerfile utilizado para realizar el docker build tiene el siguiente contenido:

```
FROM python:3.6.0-alpine

COPY requirements.txt /tmp/requirements.txt
RUN pip install -r /tmp/requirements.txt
```

```
COPY webapp.py /src/webapp.py

EXPOSE  8000
CMD ["python", "/src/webapp.py", "-p 8000"]
```

En resumen, la imagen de docker partirá de una imagen de python versión 3.6.0-alpine (Alpine es una imagen mínima de docker basada en linux muy popular por, principalmente, su ligereza y funcionalidad[33]). Copia el fichero de requisitos (requirements.txt) almacenado en el repositorio (básicamente se indica la necesidad de instalar Flask), con este fichero y instalará las dependencias de python de la webapp mediante pip, copia la propia webapp a la imagen (directorio /src/), expone el puerto 8000 y ejecuta “python /src/webapp.py -p 8000” para poner en ejecución la webapp.

La ejecución del job es la siguiente:

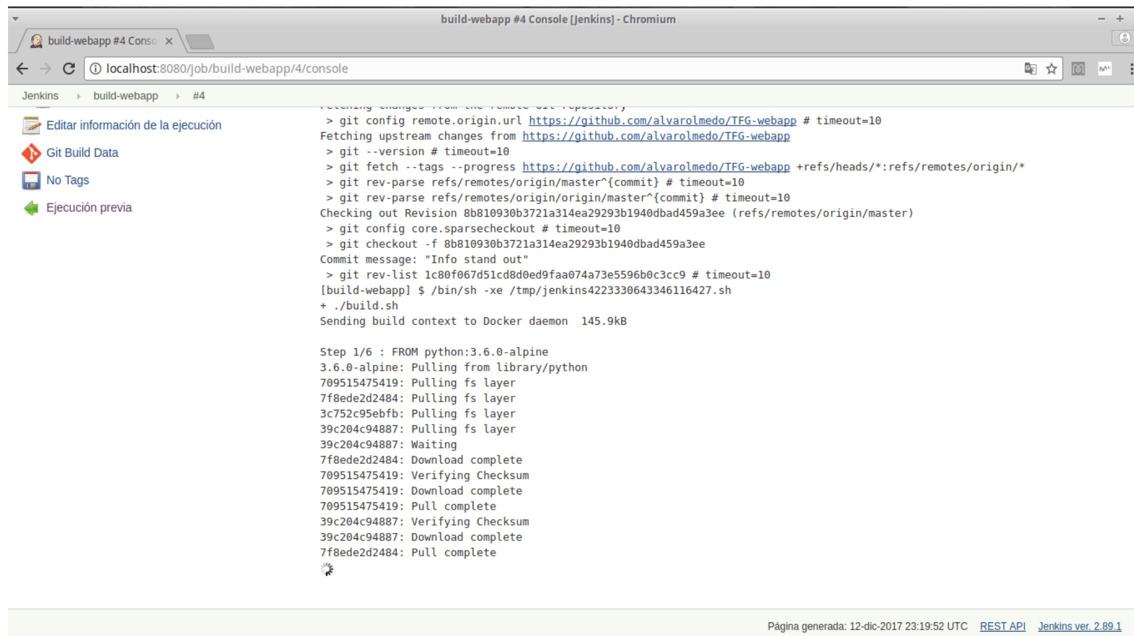

Figura XXII - Ejecución job build webapp 1

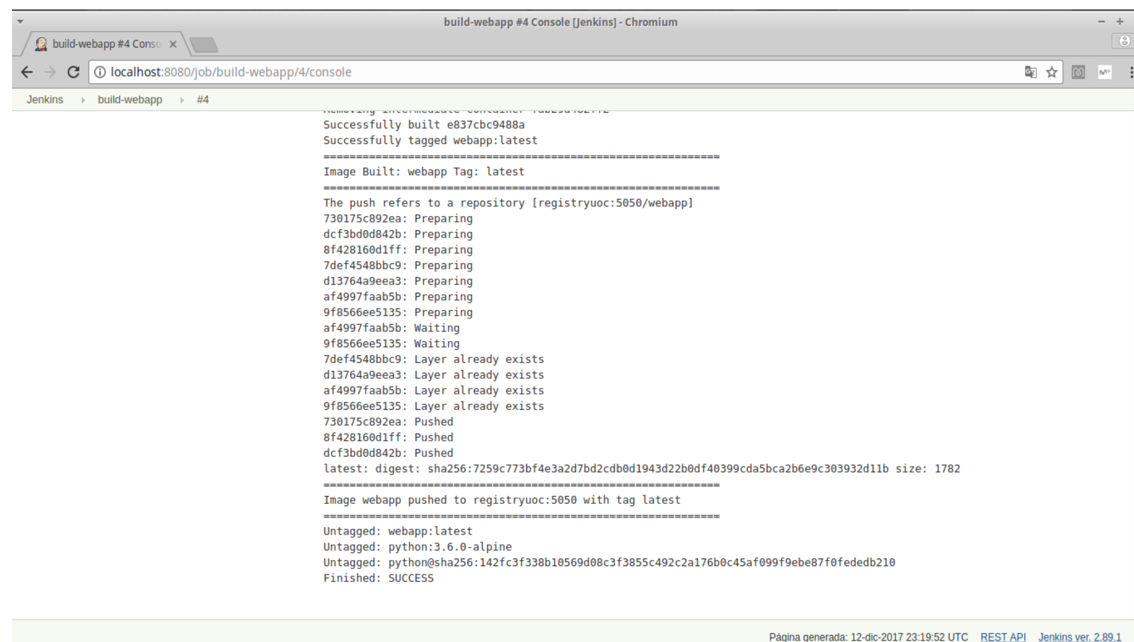

Figura XXIII - Ejecución job build webapp 2

Se puede observar como se inicia con el pull de la imagen base (indicada en el FROM del Dockerfile) y acaba haciendo push al docker registry y limpiando.

El siguiente job que se ejecuta es el deploy, que tiene una configuración similar: clonado del repo, ejecución de las tareas oportunas (*script* deploy.sh) y, si todo va correctamente, llamada al siguiente job (*smoke test*).

El *script* de despliegue, almacenado en el repositorio git, tiene 3 partes bien diferenciadas:

```
#!/bin/bash

echo "====================="
echo "== Deploying webapp =="
echo "====================="
docker-compose up -d

echo "==========================="
echo "== Putting index-pattern =="
echo "==========================="

curl    -XPUT    'http://172.16.0.1:9200/.kibana/doc/index-pattern:tfg-uoc' \
    -H 'Content-Type: application/json' \
         -d  '{"type":  "index-pattern",  "index-pattern":  {"title": "logstash-*", "timeFieldName": "@timestamp"}}'

echo ""
echo "=============================="
echo "== Importing kibana objects =="
echo "=============================="

cd kibana-objects
./kibana-importer.py      --json      everything.json      --kibana-url http://172.16.0.1:5601
```

a) El primer paso es levantar los contenedores mediante docker-compose, el siguiente paso es dejar disponible un patrón de índice (*index pattern*) en kibana para que los logs recibidos y parseados desde logstash sean correctamente almacenados[32] y, por último, se importan en kibana los objetos necesarios (visualizaciones y dashboards) para tener una visión rápida y amplia de la situación del sistema.

El fichero docker-compose.yml que despliega toda la infraestructura de la webapp es el siguiente:

```
version: '2'

services:

  nginx:
    container_name: nginx
    image: nginx:latest
    restart: always
    ports:
     - "80:80"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
      - ./nginx-logs:/var/log/nginx
    networks:
      private:
        ipv4_address: 192.168.0.2

  web1:
```

```
    container_name: web_node1
    image: registryuoc:5050/webapp:latest
    restart: always
    networks:
      private:
        ipv4_address: 192.168.0.3

  web2:
    container_name: web_node2
    image: registryuoc:5050/webapp:latest
    restart: always
    networks:
      private:
        ipv4_address: 192.168.0.4

  filebeat:
    container_name: filebeat
    image: docker.elastic.co/beats/filebeat:6.0.0
    restart: always
    volumes:
      - ./filebeat.yml:/usr/share/filebeat/filebeat.yml
    volumes_from:
      - nginx
    networks:
      private:
        ipv4_address: 192.168.0.5


networks:
  private:
    driver: "bridge"
    ipam:
     config:
       - subnet: 192.168.0.0/24
         gateway: 192.168.0.1
```

Básicamente, una red privada para gestionar las comunicaciones entre los componentes y los 4 contenedores (dos servidores con la webapp corriendo, un nginx a modo de balanceador y proxy inverso y el comentado filebeat que entrega los logs de nginx a logstash). Algunas particularidades de estos contenedores:

- el contenedor de nginx utiliza dos volúmenes, uno para que los logs sean almacenados de forma persistente en el host (y que además será aprovechado para que filebeat pueda acceder a ellos) y otro para montar la configuración almacenada en el repo en modo solo lectura debido al ":ro" indicado al final del mapeo.

Apoyado en documentación de nginx[34] se definen usuarios, logs, procesos, conexiones y, lo que nos ocupa, un proxy inverso que escucha en el 80 y redirige al puerto 8000 de uno de los nodos web_node (por nombre, no IP) que corren las webapp en dicho puerto.

```
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;
```

```
events {
    worker_connections  1024;
}


http {
    upstream web {
        server web_node1:8000;
        server web_node2:8000;
    }

    server {
        listen 80;

        location / {
            proxy_pass http://web;
        }
    }
}
```

- el contenedor de filebeat también utiliza dos volúmenes, el ya comentado de los logs de nginx (realmente monta los dos volúmenes del contenedor de nginx, pero la configuración, /etc/nginx/nginx.conf, no es útil) y otro para mapear la configuración desde el fichero filebeat.yml del repositorio:

```
filebeat:
  prospectors:
        -
            paths:
                - "/var/log/nginx/access.log"
            document_type: nginx-access
#############################                                    Output
##########################################
output:
  logstash:
    hosts: ["192.168.0.1:5000"]
    template.name: "filebeat"
    template.path: "filebeat.template.json"
    template.overwrite: false
```

Según la documentación de filebeat los prospectors permiten especificar los logs a mandar, además le es especificado un tipo, el output define el endpoint al que mandar la información, en este caso logstash, además se especifica una plantilla que, en caso de no existir, se establece en el endpoint [28][29][30].

b) Finalizado el despliegue de docker, como se ha comentado, se ejecuta la inserción del index-pattern. El *index-pattern* es insertado con un simple curl ya mostrado en el shell *script* de deploy.

c) Por útimo, se importan los objetos (búsquedas, visualizaciones y dashboards) de kibana[35][36]. Para esta importación de objetos, algo más compleja, se hace uso de un json fichero que define todos los objetos (búsquedas, visualizaciones y dashboards) y un *script* python (kibana-

importer.py) obtenido de [37] (licencia MIT) que utiliza el API de kibana para realizar la importación. Ambos ficheros son almacenados en el repositorio http://github.com/alvarolmedo/TFG-webapp dentro del directorio kibana-objects.

La ejecución del despliegue muestra todas las tareas:

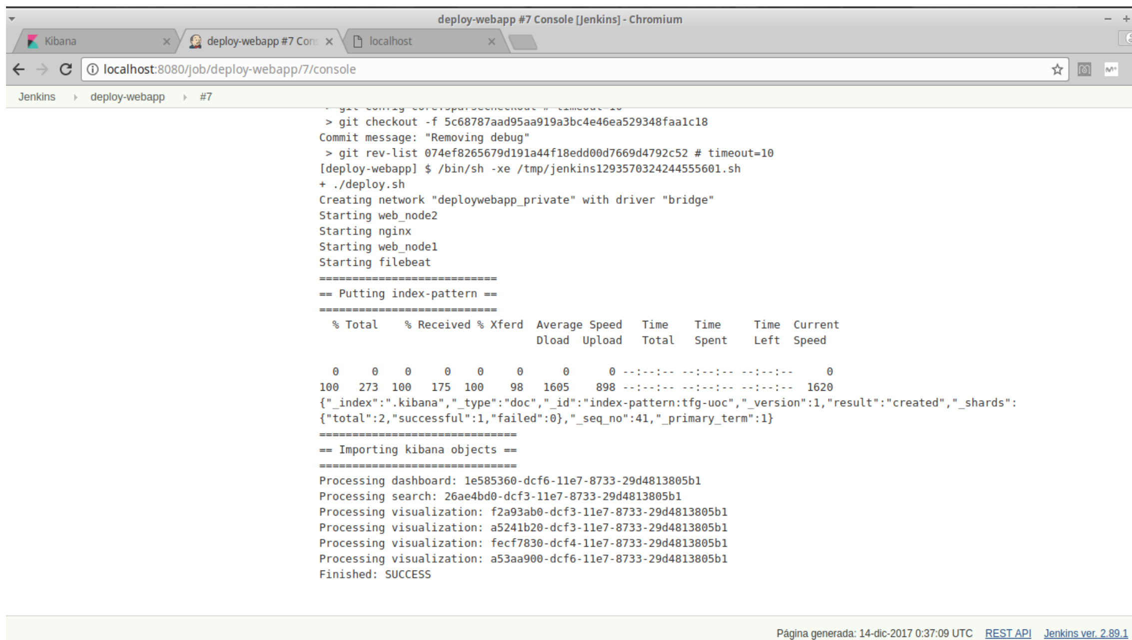

Figura XXIV - Ejecución job despliegue webapp

Estos puntos b y c son los que de forma efectiva, consiguen de forma efectiva el objetivo de este trabajo, realizar en el despliegue continuo, el despliegue de los elementos necesarios para que el análisis de performance y métricas estén disponibles de forma inmediata, en el propio despliegue de la webapp.

Para comprobar el estado del despliegue, a continuación ,se ejecuta el smoke test:

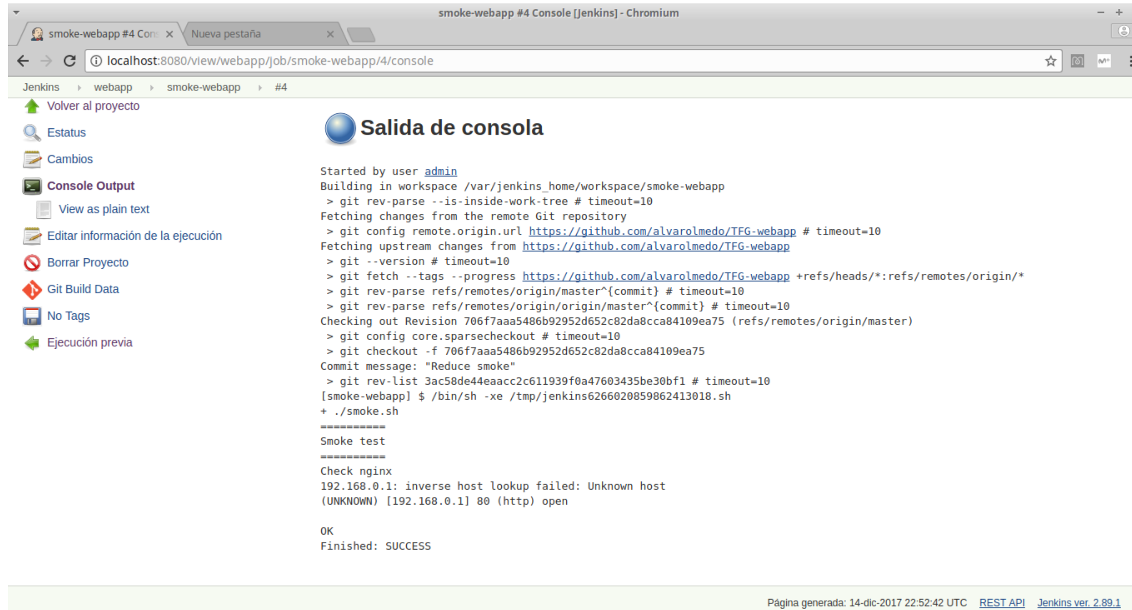

Figura XXV - Smoke Tests Webapp

La configuración de este job es bastante sencilla, se trata de chequear el correcto funcionamiento de la webapp tras el despliegue: se clona el repo http://github.com/alvarolmedo/TFG-webapp y se ejecuta el *script* smoke.sh que chequea el correcto funcionamiento de nginx:

```
#!/bin/bash

echo "=========="
echo "Smoke test"
echo "=========="

echo Check nginx
nc -vz 192.168.0.1 80
if [ $? == 0 ];then
      echo
      echo OK
else
      echo
      echo Fail in nginx
      exit 1
fi

HTTP_CODE=`curl -s -o /dev/null -w "%{http_code}"  http://192.168.0.1`
if [ $HTTP_CODE == '200' ];then
      echo
      echo OK
else
      echo
      echo Fail in nginx
      exit 1
fi
```

La webapp no puede ser comprobada directamente en cada uno de los dos contenedores que están corriendo el aplicativo debido a que desde jenkins únicamente es accesible el FE o nginx (debido a la red en la que están corriendo cada contenedor). Es por esto que todos los chequeos del *script* se centran en el nginx.

5.4 Uso de la webapp y acceso a la información

Con la creación del *index-pattern* Kibana ya está listo para recibir logs:

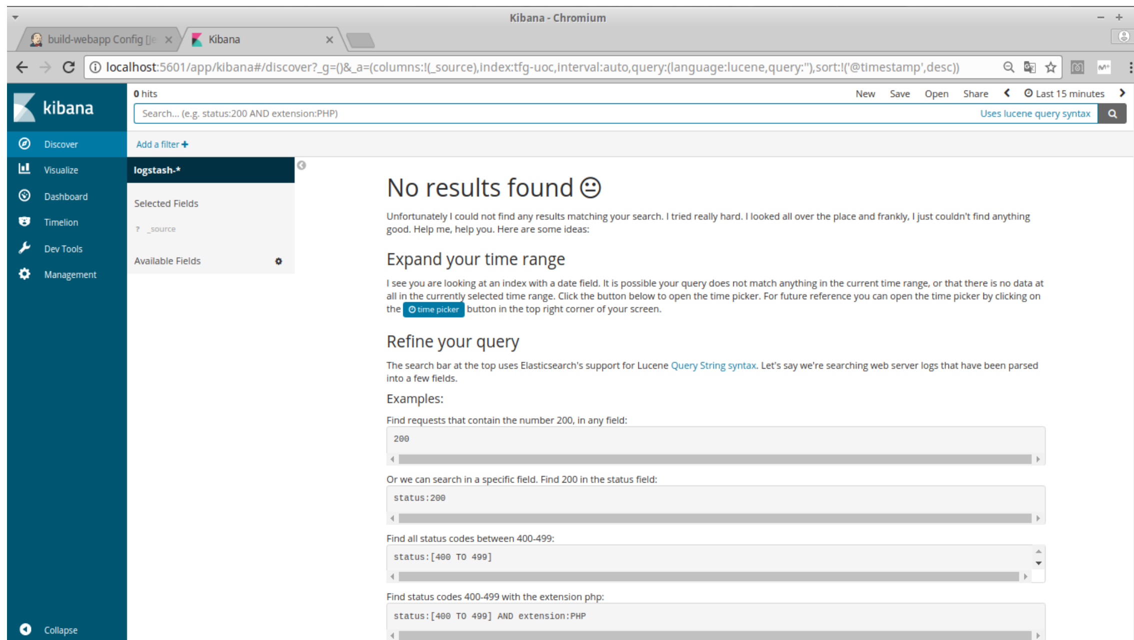

Figura XXVI - Kibana con index pattern configurado

Además se han creado búsquedas (all), visualizaciones (Agents, AverageBytes, counts, requests, Responses y URL) y un dashboard (Main dashboard):

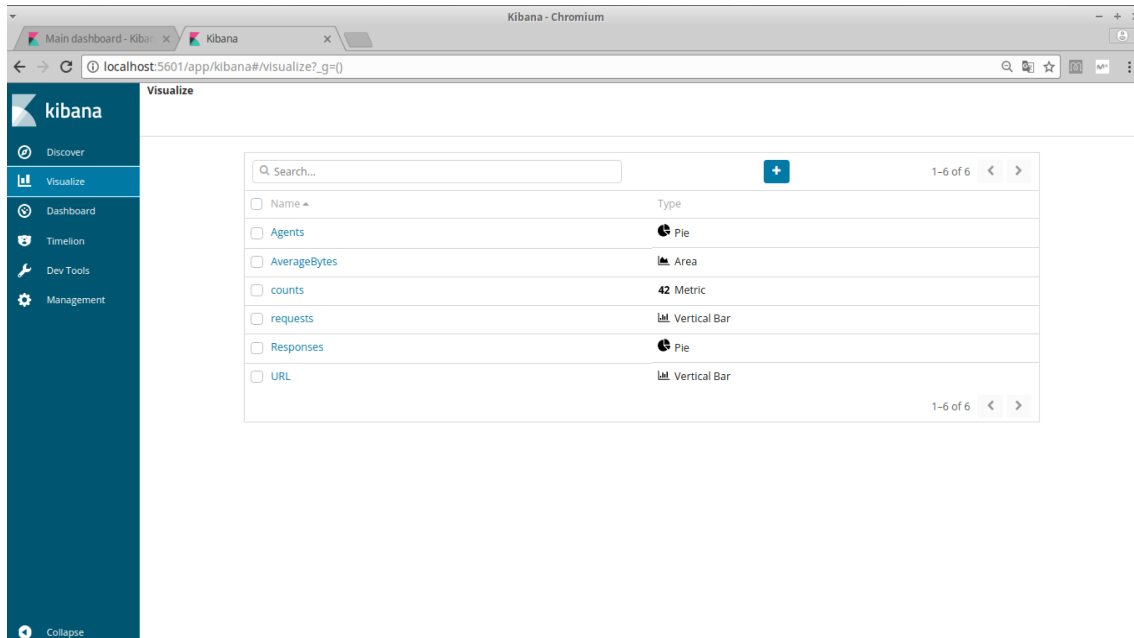

Figura XXVII - Objetos en Kibana importados

Se comienzan a realizar peticiones a nginx desde distintos clientes (Firefox, Chromium, curl y wget) está es la información almacenada en los últimas 15 minutos:

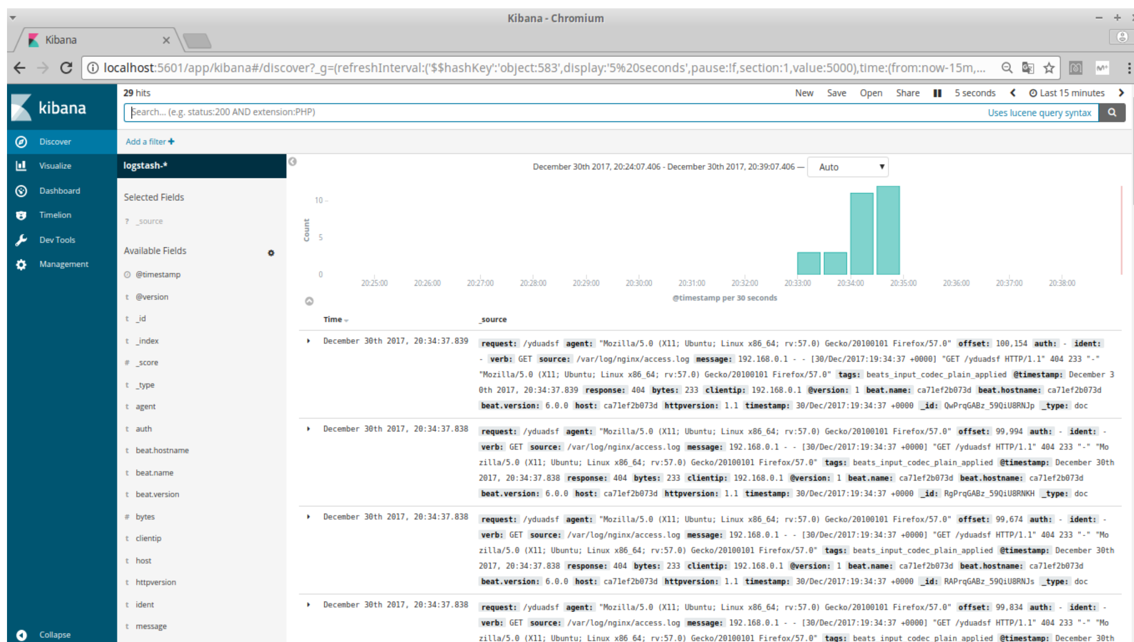

Figura XXVIII - Información desde Kibana Discover

Y así es vista desde el dashboard creado que incorpora las seis visualizaciones:

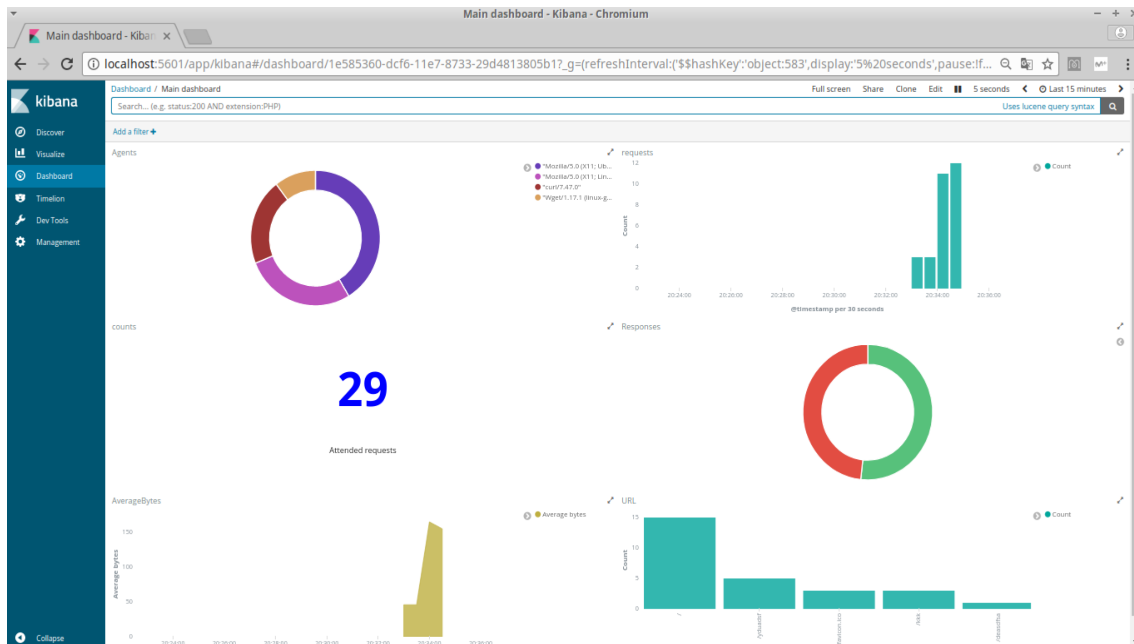

Figura XXIX - Dashboard y visualizaciones en Kibana

Hay muchos y muy diferentes gráficos/métricas que se pueden generar con los datos ofrecidos por los logs de nginx para analizar su rendimiento, en este caso se ha optado por algunos de los recomendados por logz.io [38].

5. Conclusiones

Queda demostrado con este trabajo que es viable automatizar el despliegue de las métricas y elementos necesarios para poder realizar un análisis de rendimiento o *performance* desde el mismo momento en el que se realice el despliegue de la propia aplicación o servicio, reduciendo así el gap que existe actualmente con este tipo de tareas.

Basta con:
1) Incorporar a los equipos de operaciones en los procesos de desarrollo para que puedan aportar toda la información sobre la métricas, análisis, los sistemas que los soportan, así como de los actuales procesos de despliegue de estos elementos.
2) A partir de aquí, trabajar para añadir como una parte más del código, los elementos que soportan las métricas y visualizaciones en el formato requerido por el sistema en el que van a ser importadas.
3) Asegurar, durante la fase de despliegue, la importación, actualización,creación,... de dichos elementos (generalmente a través de los APIs que ofrezca el sistema donde van a ser visualizados posteriormente los datos).

Obviamente estas tareas han de ser enfocadas para que puedan ser ejecutadas idempotentemente y de manera automatizada (tal y como se hace con cualquier otro código o tarea que vaya a ser ejecutada dentro de una *pipeline* de despliegue continuo).

Posiblemente sea necesario dentro del equipo de desarrollo, evangelizar más con la filosofía DevOps, haciendo hincapié en que se ha de hacer un esfuerzo en impulsar este tipo de actividades más relacionadas con la operación (métricas, monitorización,backups…) dentro del ciclo de desarrollo. Esto es, no basta ya con desplegar las aplicaciones con sus configuraciones de forma automática sino que, además, se ha de complementar el despliegue con “anexos” como el despliegue de los servicios necesarios y sus respectivas configuraciones para que la aplicación sea correctamente monitorizada, respaldada, analizada,…

Respecto a la planificación planteada se ha tratado de seguir en la medida de lo posible pero muchas tareas han tenido más trabajo del previsto. Por ejemplo, la tarea 9, donde se resuelve la disponibilidad de un servicio como ELK para poder visualizar en ese sistema datos y métricas, ha sufrido grandes cambios: se comenzó con la idea de desplegar el servicio, se pasó por la idea de desplegar las métricas y llevar los datos a un servicio en la nube como logz.io pero, finalmente, ante la imposibilidad de poder interactuar con el API de ELK en dicho servicio en la nube, se vuelve y finalmente se implementa el ELK a nivel local, via docker. Además, a posteriori, se observa que para todas las tareas, la planificación fue muy arriesgada, acortando los plazos demasiado. Se deberían haber puesto, para todas las tareas, plazos más amplios. Es por esto que la planificación no ha sido modificada en tareas, aunque si en fechas.

Por último, algunas líneas de trabajo se abren a partir de este trabajo. Estas podrían ir en varias direcciones:

- Ampliar las propias métricas o visualizaciones para un sistema similar al construido en este trabajo.
- Explorar las métricas necesarias para otro tipo de aplicaciones. Aquí se ha utilizado una webapp con un proxy inverso para balancear pero se deben analizar otros tipos: webapp distribuidas, aplicaciones más clásicas en cuanto a su arquitectura,etc...
- Analizar otros posibles sistemas donde poder llevar los datos o logs y tratar de implementar, al igual que en este trabajo, el despliegue automático de dichas métricas.
- Indagar en otros posibles sistemas con operativas similares pero funciones diferentes, por ejemplo, desplegar alarmado en zabbix para monitorizar la plataforma, desplegar configuración de bacula para respaldar de forma automática,..
- Incluir en los smoke test de la *webapp* el chequeo de las propias métricas, esto es, de los elementos desplegados en Kibana para el análisis de la aplicación (visualizaciones, dashboards,...)

6. Glosario

Anfitrión (*host*): Sistema encargado de hospedar una o más máquinas virtuales o, en el caso de dcoker, de correr contenedores. Para realizar esta tarea ha de poseer un software específico corriendo.

API: Interfaz de programación de aplicaciones, es una capa de abstracción facilitada por un software exponiendo ciertas funciones para poder ser utilizado por otro software.

Aplicación web (o webapp): Herramienta que los usuarios pueden utilizar usando un navegador y accediendo a un servidor web, es decir, es un programa codificado en un lenguaje que es entendido por el navegador.

Biblioteca: Conjunto de funciones que ofrece una interfaz definida, la diferencia fundamental con un programa al uso es que una biblioteca no se ejecuta de forma autónoma sino que se espera que sea utilizada por otros programas.

Despliegue: El *despliegue* comienza cuando el código ha sido suficientemente probado, ha sido aprobado para su liberación y ha sido distribuido en el entorno de producción. Se podría decir que un despliegue consiste en todas las actividades que hacen que un software quede listo para ser usado.

Empaquetado de software: Proceso por el cual se unen diferentes ficheros que forman un programa junto a ciertas instrucciones para su instalación en un único fichero, normalmente binario y dependiente del SO.

Entrega continua: Metodología de desarrollo de software ágil que se basa en ciclos de desarrollo breve dando como resultado software confiable en un espacio corto de tiempo. Por tanto, la liberación del software es más rápida y frecuente. Consta de 3 etapas de automatización: compilación, pruebas y despliegue. Amplía la integración continua.

Features: Características del software claramente distinguidas, tanto funcionales como no funcionales.

Integración continua: Metodología de desarrollo de software que se basa en la combinación del software de los diferentes desarrolladores de un equipo de desarrollo en un repositorio central (CVS como git, subversion,...) generando las oportunas versiones del software empaquetado y las correspondientes pruebas automatizadas.

Logs: Registro secuencial, normalmente en un fichero aunque podría ser en otro sistema como una base de datos, de todos los acontecimientos relacionados con un proceso. Además de la propia información del evento es habitual encontrar en las misma traza de un evento información adicional como el momento (fecha y hora en la que se produce) así como una categorización que indica la importancia del evento.

Metodología ágil: Basado en el desarrollo iterativo o incremental, el desarrollo del software es realizado por equipos de trabajo auto-organizados y multidisciplinares produciendo software usable cada ciclo de desarrollo.

Nube (*cloud*): Los servicios en la nube o *cloud computing* es un modelo de arquitectura que permite ofrecer diferentes servicios a través de Internet.

Núcleo (*kernel*): Conjunto de procesos que gestionan el acceso al hardware de manera segura y ordenada a través de llamadas al sistema.

Software libre (*open source*): Software licenciado de tal manera por su autor que permite 4 libertades a cualquier usuario: copia, distribución, modificación y utilización.

Pipeline: De forma genérica se podría decir que es un proceso compuesto por diferentes pasos ejecutados de forma secuencial y en los que la salida o productos de un paso n es la entrada o parte de ella para un paso n+1. En el caso concreto de lo

QA (*Quality assurance* o aseguramiento de la calidad): Engloba desde el rol de las personas que ejercen dicha actividad hasta la propia actividad en sí, consistente en diversas actividades que tienen como objetivo certificar y asegurar ciertos requisitos de calidad de un producto o servicio.

Rendimiento (*performance*): Comportamiento de, en sentido genérico, un producto o servicio ante determinadas situaciones y con determinados recursos.

Repositorio: De manera general se podría definir como un lugar donde almacenar cosas. Más concretamente se refiere al lugar donde almacenar artefactos (normalmente paquetes o binarios), o código (CVSs normalmente).

Script: Programa compuesto de una secuencia de instrucciones que normalmente son interpretadas. Normalmente es un programa más o menos simple. Al ser interpretado no ha de ser compilado (para ser ejecutado por el procesador directamente) y, por tanto, normalmente se trata de un fichero en texto plano.

Servidor web: Proceso corriendo en el lado servidor y que atiende peticiones HTTP (por norma en el puerto TCP número 80).

Smoke tests: Se podría resumir como una revisión ligera del producto de software para comprobar que su funcionamiento es el esperado.

Versionado software: Proceso por el que asigna a un software un identificador (nombre, código o número, o una combinación de estos) para indicar su nivel de desarrollo.

Yaml: Formato de serialización de datos facilmente leíble por humanos. Inspirado en lenguajes como XML, en este caso no es un lenguaje de marcado.

7. Bibliografía

[1] https://puppet.com/resources/whitepaper/state-of-devops-report (11 de diciembre de 2017)
[2] https://puppet.com/blog/what%E2%80%99s-best-team-structure-for-devops-success (11 de diciembre de 2017)
[3] https://labs.spotify.com/2014/03/27/spotify-engineering-culture-part-1/ (27 de diciembre de 2017)
[4] https://itrevolution.com/the-amazing-devops-transformation-of-the-hp-laserjet-firmware-team-gary-gruver/ (10 de diciembre de 2017)
[5] Varios, *Minimizing code defects to improve software quality and lower development costs*. Development solutions White paper (IBM). Octubre 2008.
[6] Syer, M.D., Shang, W., Jiang, Z.M. *Continuous validation of performance test workloads* et al. Autom Softw Eng (2017) 24: 189. https://0-doi-org.cataleg.uoc.edu/10.1007/s10515-016-0196-8
[7] Ian Molyneaux, *The Art of Application Performance Testing*, 2nd Edition, O'Reilly Media, Sebastopol, CA, Diciembre 2014
[8] https://jenkins.io/doc/ (24 de diciembre de 2017)
[9] https://jenkins.io/doc/book/pipeline/ (24 de diciembre de 2017)
[10] Kief Morris, *Infraestructure as Code*, 1st Edition, O'Reilly Media, Sebastopol, CA, Junio 2016
[11] https://docs.docker.com/compose/ (25 de diciembre de 2017)
[12] https://www.owasp.org/index.php/OWASP_Proactive_Controls?tab=OWASP_Proactive_Controls_2016#tab=OWASP_Proactive_Controls_2016 (24 de diciembre de 2017)
[13] Jim Bird, *DevOpsSec*, 1st Edition, O'Reilly Media, Sebastopol, CA, Junio 2016
[14] https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/ (25 de diciembre de 2017)
[15] https://docs.docker.com/engine/reference/builder/ (12 de diciembre de 2017)
[16] https://docs.docker.com/registry/ (25 de diciembre de 2017)
[17] https://www.elastic.co/products (25 de diciembre de 2017)
[18] https://logz.io/blog/shipping-logs-filebeat/ (23 de noviembre de 2017)
[19] https://logz.io/blog/introducing-elk-apps/ (23 de noviembre de 2017)
[20] http://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/ (12 de diciembre de 2017)
[21] https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/ (12 de diciembre de 2017)
[22] https://github.com/jenkinsci/docker/blob/master/README.md#installing-more-tools (12 de diciembre de 2017)
[23] https://github.com/deviantony/docker-elk (12 de diciembre de 2017)
[24] https://github.com/alvarolmedo/docker-elk/commits/master (12 de diciembre de 2017)
[25] https://github.com/deviantony/docker-elk#how-can-i-persist-elasticsearch-data (15 de diciembre de 2017)
[26] https://www.elastic.co/guide/en/logstash/current/logstash-config-for-filebeat-modules.html#parsing-nginx (15 de diciembre de 2017)
[27] https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html (15 de diciembre de 2017)

[28] https://www.digitalocean.com/community/tutorials/adding-logstash-filters-to-improve-centralized-logging (15 de diciembre de 2017)
[29] https://www.elastic.co/guide/en/logstash/current/advanced-pipeline.html (15 de diciembre de 2017)
[30] https://www.elastic.co/guide/en/beats/filebeat/1.2/filebeat-template.html (15 de diciembre de 2017)
[31] https://www.elastic.co/guide/en/kibana/current/index-patterns.html#settings-create-pattern (15 de diciembre de 2017)
[32] https://www.elastic.co/guide/en/kibana/current/connect-to-elasticsearch.html (15 de diciembre de 2017)
[33] https://wiki.alpinelinux.org/wiki/Docker (15 de diciembre de 2017)
[34] Derek DeJonghe, *Complete NGINX Cookbook*, 1st Edition, O'Reilly Media, Sebastopol, CA, Marzo 2017.
[35] https://www.digitalocean.com/community/tutorials/how-to-use-kibana-dashboards-and-visualizations#prerequisites (15 de diciembre de 2017)
[36] https://www.elastic.co/guide/en/kibana/current/tutorial-visualizing.html (15 de diciembre de 2017)
[37] https://github.com/nh2/kibana-importer (15 de diciembre de 2017)
[38] https://logz.io/blog/nginx-access-log-monitoring-dashboard/ (30 de diciembre de 2017)

8. Anexos