ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Grado

Construcción de entorno DevOps mediante arquitectura de contenedores de software

(Building DevOps environment using software container architecture)

Para acceder al Titulo de

Graduado en Ingeniería de Tecnologías de Telecomunicación

Autor: Alfonso Prieto Saiz

09 - 2021

E.T.S. DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACION

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE GRADO

Realizado por: Alfonso Prieto Saiz
Director del TFG: Rafael Menéndez De Llano Rozas
Título: "Construcción de entorno DevOps mediante arquitectura de contenedores de software"
Title: **"**Building DevOps environment using software container architecture**"**

Presentado a examen el día:

para acceder al Título de

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Composición del Tribunal:
Presidente (Apellidos, Nombre):
Secretario (Apellidos, Nombre):
Vocal (Apellidos, Nombre):

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: El Vocal

Fdo.: El Director del TFG
(sólo si es distinto del Secretario)

Vº Bº del Subdirector

Trabajo Fin de Grado Nº
(a asignar por Secretaría)

AGRADECIMIENTOS

A mis compañeros de trabajo que fueron el germen de abordar este proyecto. Gracias De Juan. Gracias Lidia.

A mi director de proyecto Rafael Menéndez, gracias por la libertad que me ha dado y por aceptar mi solicitud.

Agradecer al responsable académico de la titulación, Tomás Fernández Ibáñez, por su paciencia al responder a todos mi correos y dudas a lo largo de estos años de periplo por la Adaptación a Grado.

Y por qué no, a mí. Porque nunca creí cuando decidí empezar esto, me iba a llevar tanto. Y a pesar de que las circunstancias de la vida y el desámino pareciese que nunca iba a poder rematar el trabajo, no rendirse tarde o temprano da sus frutos. Eso sí, jamás en la vida me meteré de nuevo en una de estas…de momento.

Gracias a todos.

LISTA DE ACRÓNIMOS

ALM: application lifecycle management
API: application programming interface
CD: continuous deployment/delivery
CI: continuous integration
CLI: command-line interface
CPU: central processing unit
DC: despliegue continuo
DevOps: development & operations
IC: integración continua
POM: project object model
REST: representational state transfer
SCV: sistema de control de versiones
SCVD: sistema de control de versiones distribuido
VCS: version control system
YAML: yaml ain't markup language

ÍNDICE DE CONTENIDO

ÍNDICE DE ILUSTRACIONES

1. Introducción

La rápida evolución del desarrollo de software, aunque ha sido una constante desde su nacimiento, ha sufrido un cambio exponencial en los últimos años. No estamos hablando sólo de cambios a nivel técnico con la aparición y evolución de nuevos lenguajes, frameworks y tecnologías, sino también a nivel de metodologías de desarrollo con nuevos paradigmas, procesos y prácticas. Todo ello siempre encaminado y con objetivo de *generar un mejor software, más robusto ante el error y más seguro ante posibles ataques, y que pueda desarrollarse y adaptarse al cambio en el menor tiempo posible*.

Con esta máxima, aunque las tendencias actuales marcan las líneas claras de cómo debe ser o debería ser el desarrollo de nuevo software, no debemos nunca perder de vista aquel software que a lo largo de los años se ha ido desarrollando y que actualmente sigue en vigencia. En busca de esa máxima, nace la necesidad de integrar estos nuevos procesos y metodologías a todo ese software, en la medida que éste lo permite, para mejorar su mantenibilidad y corrección, y así solventar las carencias propias de las tecnologías, procesos y prácticas del momento en que fueron desarrolladas.

Por todo ello, conceptos como Agile, CI/CD o DevOps, cuyo objetivo es facilitar, mejorar y transformar los procesos "clásicos" de construcción y despliegue/entrega de aplicaciones, están a la orden del día dentro del panorama actual del desarrollo como estándares de facto dentro de la comunidad y de la industria del software. Muy ligado a ello, una tecnología, ya conocida pero que había pasado un tanto desapercibida hasta no hace tanto, como la de contenedores de software, ha aprovechado el momento para crecer, hacerse fuerte e integrarse completamente dentro del ecosistema como un elemento común más.

1.1. Motivación

Con estos antecedentes, es muy habitual que cualquier persona o equipo de desarrollo que se enfrenta a la tarea de iniciar el desarrollo de una nueva aplicación o el mantenimiento de una existente, pueda plantearse si posee las herramientas y utilizan las mejores prácticas ante los vertiginosos cambios que se han ido produciendo los últimos años o si los proyectos en los que trabajan simplemente las practican.

Así, es fácil encontrar *stacks* o entornos de desarrollo y despliegue de aplicaciones en los que diferentes agentes implicados en la arquitectura de servicios son desplegados cada uno de ellos en máquinas dedicadas o compartidas donde los procesos muchas veces son dependientes, no sólo de la interacción de ciertos perfiles de profesional, sino que además se encuentran poco automatizados y/o existe un bajo nivel de integración entre todos ellos. También es un problema clásico las posibles diferencias entre las máquinas de los diferentes entornos de despliegue y/o los entornos locales entre los desarrolladores, lo cual provoca verdaderos dolores de cabeza por incompatibilidades o

versionado no homogéneo de las dependencias así como de su configuración. Es cierto que en ese aspecto, el uso de máquinas virtuales ha permitido en gran medida solventar este problema (a costa de rendimiento y consumo de recursos extras en las máquinas), pero especialmente los contenedores software. Los contenedores de software son entornos de ejecución ligeros y aislados que contienen todos los archivos, variables y librerías que las aplicaciones que contienen necesitan para ejecutarse. Esto junto a otros aspectos, que veremos más adelante, los convierten en agentes muy interesantes dentro de entornos DevOps y CI/CD.

Por tanto, es interesante, si no se están aplicando estas buenas prácticas, el crear un entorno con las herramientas y servicios que permitan aplicarlas de la manera más sencilla y que facilite la vida a los desarrolladores aislándolos de temas secundarios no centrados en facetas propias del desarrollo o mantenimiento. Y no sólo hablamos, de implantar estos entornos, sino de evolucionar los ya presentes que pudieran estar implementando esas buenas prácticas para mejorarlos y mejorar a su vez los procesos en los que estuviesen interviniendo dentro de las diferentes fases del desarrollo del software.

1.2. Objetivos

Esta motivación hace plantearnos la construcción de una plataforma estándar de ALM (Application Lifecycle Management) que permita poner en práctica la metodología de integración continua, mediante un ecosistema de software abierto (con las máximas prestaciones de integración e interoperabilidad) y una arquitectura de contenedores de software.

Para ello, se pretende cubrir e integrar los métodos más comunes asociados a los entornos y procesos de integración continua mediante contenedores de software que cubran las siguientes necesidades:

- Gestión y control de versiones de código fuente, elementos de configuración y desarrollo de software colaborativo.
- Centralización y almacenamiento de paquetes software.
- Revisión y evaluación de calidad de software.
- Orquestación y automatización de tareas para el despliegue de software.

Al final, el objetivo que se persigue es la automatización, integración y simplificación de los elementos y procesos más comunes involucrados en el desarrollo de software, haciendo uso de las herramientas más demandadas y presentes actualmente en las empresas. Siendo más precisos, se plantea que la solución sea capaz de que ante cambios del código fuente por parte del desarrollador, el proceso por el que debe pasar el software hasta su despliegue sea lo más transparente y automático posible para este.

Aunque ya se verá en detalle en el capítulo dedicado a la implementación técnica de la solución propuesta, planteamos dos escenarios u objetivos que puedan poner en práctica el entorno y ver que es una solución válida para ello:

- Generación, centralización y almacenamiento de un paquete o dependencia software.
- Generación, centralización, almacenamiento y despliegue de una aplicación web.

1.3. Estructura documento

Teniendo en cuenta esto, el presente documento se estructura en cuatro capítulos.

Tras el presente y breve capítulo de introducción, comentando un poco el escenario sobre el que nos movemos y explicando la motivación y objetivos del trabajo, en el capítulo 2 nos centraremos en los aspectos teóricos más relevantes relacionados que permitan entender mejor, no sólo las metodologías o prácticas de las que ya hemos hablado y a las que el entorno objeto del trabajo quiere dar respuesta, sino también la tecnología en la que se va a basar el mismo, los contenedores de software.

El capítulo 3, lo dedicaremos a explicar y desgranar la implementación técnica propuesta junto con cada uno de los elementos que la componen y su integración para lograr el objetivo general del trabajo.

Finalizaremos el documento, con unas conclusiones generales sobre el desarrollo del trabajo realizado, analizaremos los resultados obtenidos y evaluaremos su conformidad con los objetivos marcados. Pondremos de manifiesto carencias o errores cometidos, planteando posibles soluciones, mejoras y/o líneas de investigación futuras.

2. Conceptos previos

2.1. DevOps [16][20]

DevOps es uno de los términos de moda en el entorno de las tecnologías de la información. Por lo general, se asocia a estrategias de transformación digital, y a metodologías como integración continua/despliegue continuo IC/DC o el desarrollo ágil.

Gran parte de la confusión con el término viene de mezclar o confundir lo que es DevOps con los requisitos necesarios o los beneficios obtenidos de implementar DevOps. Aunque es cierto que, a día de hoy, se trata de un término cuya definición y límites aún no han acabado de determinarse del todo.

Qué es DevOps

Si empezamos con lo que hoy en día podemos entender como definición “canónica”, Wikipedia dice: "DevOps (acrónimo inglés de development -desarrollo- y operations -operaciones-) es un conjunto de prácticas que agrupan el desarrollo de software (Dev) y las operaciones de TI (Ops). Su objetivo es hacer más rápido el ciclo de vida del desarrollo de software y proporcionar una entrega continua de alta calidad. DevOps es una práctica complementaria al desarrollo de software ágil ; esto debido a que varias de las características de DevOps provienen de la metodología Agile (término en inglés para la metodología de desarrollo ágil)" [22]. La metodología Agile, de manera resumida, enfoca el desarrollo en iteraciones rápidas y continuas con entregas constantes que vayan resolviendo las pequeñas piezas funcionales que conforman la solución software completa. Estos ciclos continuos y constantes permiten no sólo tener rápidamente algo tangible sino de generar flujos de retroalimentación que permiten mejorar el propio software.

La principal característica del movimiento DevOps es defender activamente la automatización y el monitoreo en todos los pasos de la construcción del software, desde la integración, las pruebas, el despliegue, hasta la implementación y la administración de la infraestructura. DevOps nace como respuesta a la dependencia entre del desarrollo de software y las operaciones IT. El objetivo es ayudar a producir software más rápidamente y de mejor calidad, y por lo tanto también con un coste menor. Los sistemas DevOps deben permitir entregas o despliegues muy frecuentes lo que se conoce como despliegue continuo (continuous deployment) o entrega continua (continuous delivery).

Tres ideas clave que definen DevOps:

- DevOps es una metodología para creación de software.
- DevOps se basa en la integración entre desarrolladores de software y administradores de sistemas.

- DevOps permite fabricar software más rápidamente, con mayor calidad, menor coste y una alta frecuencia de *releases*.

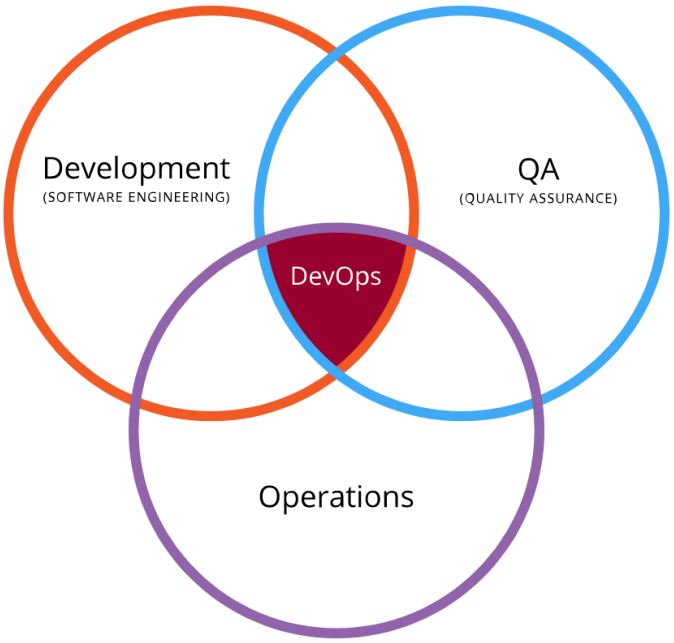

Ilustración 1 *Procesos involucrados en DevOps [22]*

Algunos entienden DevOps como "cultura", otros no. Lo que sí requiere es de un fuerte cambio en los procesos, en las herramientas y un cambio organizativo para su implementación. Un cambio hacia la colaboración, la comunicación, y en último término la completa integración entre las antiguas y habitualmente aisladas áreas de desarrollo y sistemas dentro de las empresas.

Esto no consiste en generar más responsabilidades de los desarrolladores haciendo que se conviertan en hombres orquesta, sino en liberar a los desarrolladores para que se centren en escribir código. DevOps debe eliminar el trabajo y las preocupaciones de la puesta en producción del software una vez que está escrito.

Una definición de DevOps con la que más o menos todos podemos estar de acuerdo podría ser: *DevOps es una metodología de desarrollo software basada en la integración entre desarrolladores y administradores de sistemas, que permite que los desarrolladores puedan enfocarse solo en desarrollar y puedan desplegar su código rápidamente*.

2.2. Integración continua [5][9][13]

Como requerimiento dentro de DevOps, el término Integración Continua aparece originalmente a finales de los 90 como una de las doce prácticas dentro de la metodología Programación Extrema definida por Kent Beck tras su experiencia como líder de proyecto para Chrysler. En dicha metodología, se definen como valores principales la simplicidad en el diseño, la comunicación, la retroalimentación, la valentía

y el respeto. Como vemos, no sólo trata aspectos técnicos sino también los propios procesos y la interacción entre todos los participantes en el desarrollo. Y de nuevo, todo ello tiene también que plasmarse de alguna manera en herramientas que nos ayuden a ese objetivo.

Qué es IC

La integración continua (IC) es una práctica de desarrollo de software donde los miembros de un equipo integran su trabajo de manera frecuente. Cada integración es verificada mediante un proceso de construcción automático (con test incluido) que permite la detección de errores de integración y su corrección lo antes posible.

La CI o IC es resultado de los viejos procesos donde erróneamente la fase de integración de código entre los diferentes desarrolladores de un proyecto se tomaba como una fase puntual dentro del proceso de desarrollo del software y no como una práctica continua durante el mismo. Este erróneo planteamiento provocaba grandes problemas no sólo por la cantidad de código y la complejidad que se podía dar a la hora de hacer la integración, sino también la incertidumbre en la estimación del tiempo o el tiempo real que al final requería la tarea.

Como ya hemos visto, dentro de DevOps la IC es uno de los requerimientos a implementar y, para ello, hace falta disponer de un entorno y una serie de herramientas que permitan:

- Mantenimiento de un repositorio único de código fuente, integración frecuente y diferentes líneas de desarrollo.
- Construcción automática del software (incluido testing) y despliegue.
- Información de todo lo que está ocurriendo disponible para todos los involucrados.
- Disponibilidad constante de las diferentes versiones del software.

Pero también es necesario una serie de buenas prácticas y su aceptación por los equipos de desarrollo, de manera que se conviertan en algo que se realice de forma automática, en un hábito.

Qué herramientas necesitamos en un entorno de IC

Evidentemente, pueden ser más, pero teniendo en cuenta todo esto, las herramientas o componentes que se espera encontrar al menos en un entorno de IC serían:

- Sistema de control de versiones (SCV o VCS) o también denominado repositorio de control de versiones. Herramienta básica que permite trabajar a los equipos de manera sencilla, organizada y, lo más importante, de manera sincronizada.

  Utilizando una arquitectura cliente-servidor, un servidor guarda las diferentes versiones de un proyecto, es decir, los diferentes cambios en ficheros y código fuente, generando a su vez un histórico que permite no sólo ver la evolución del

proyecto sino también recuperar versiones previas o realizar auditorías. Los clientes se conectan al servidor para obtener una copia actualizada del proyecto, sobre la que trabajan y más tarde integran todos sus cambios registrándolo en el sistema, de manera que esté a disposición de todos los desarrolladores del equipo.

Los CVS también pueden mantener distintas "ramas" de un proyecto. Esto significa que permiten, a partir de una línea base de código fuente, la convivencia entre diferentes líneas de desarrollo o ramas y que sus cambios puedan ir incorporándose entre estas y a la línea base o maestra según las necesidades del desarrollo.

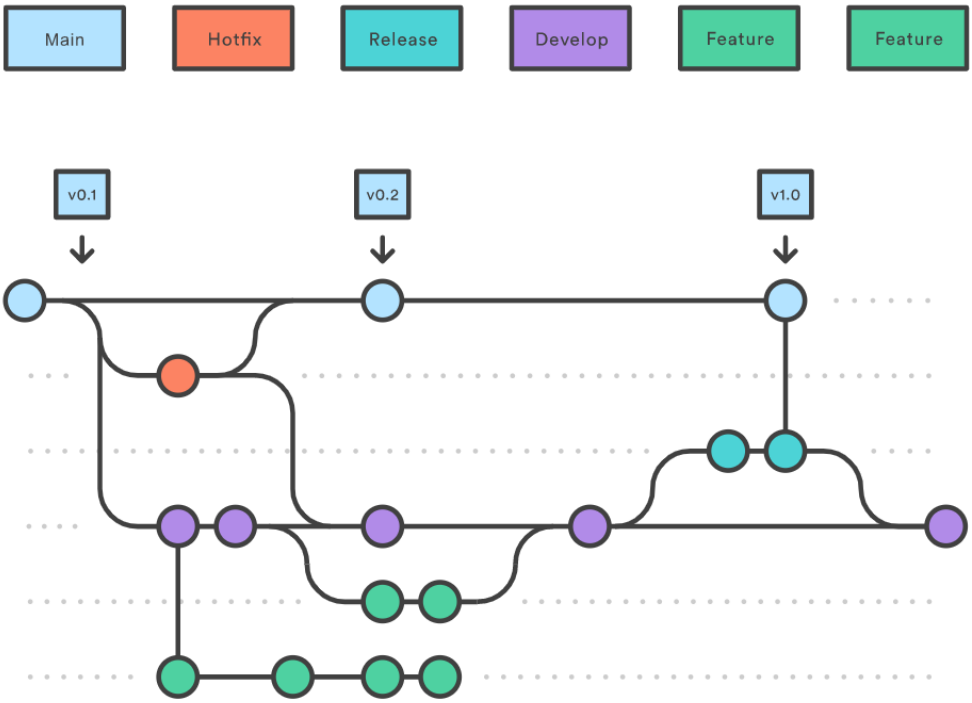

Ilustración 2 *Ejemplo flujo de trabajo basado en ramas [1]*

Existen diferentes tipos de SCV siendo actualmente los sistemas de control distribuidos (SCVD) los más utilizados, y especialmente Git. Existen diferentes herramientas que han nacido alrededor de este software para ampliar, organizar y facilitar los flujos de trabajo en los equipos encaminado a procesos de IC, siendo muy conocidas y ampliamente utilizadas herramientas como Github, Gitlab o Bitbucket.

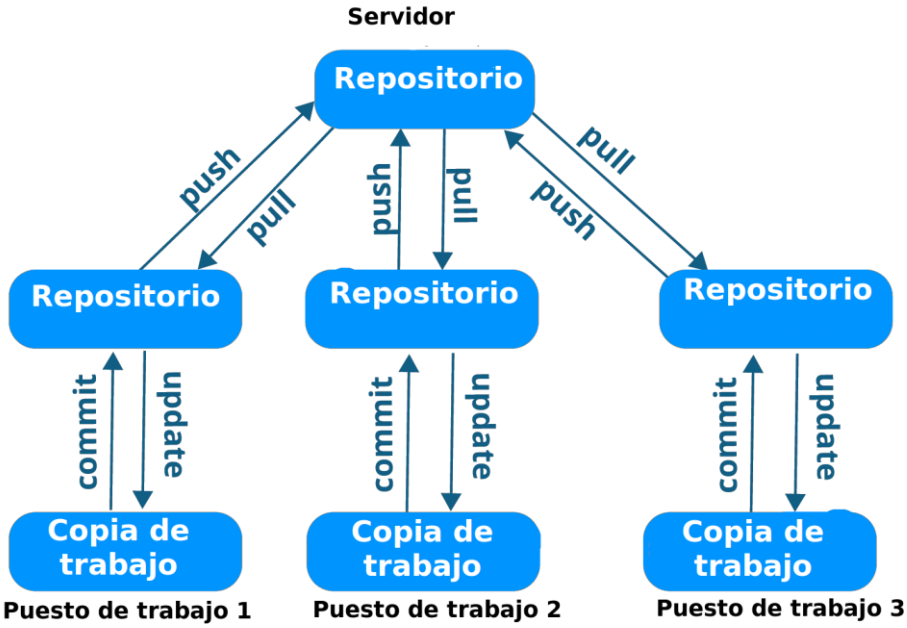

Ilustración 3 *Esquema funcional de un sistema de control de versiones distribuido [12]*

- Sistema de construcción automático que facilite la fase de la obtención del ejecutable, empaquetado o compilado necesario para la publicación del software. Desde los antiguos y complejos entornos de construcción y compilación, la necesidad ha provocado un alto desarrollo de herramientas que vengan no sólo a cumplir esa función sino a ampliarla. Cubriendo otros muchos aspectos, básicos a día de hoy, como son la gestión de dependencias o librerías, la ejecución de test (unitarios y/o integración), optimización, empaquetado e incluso despliegue.

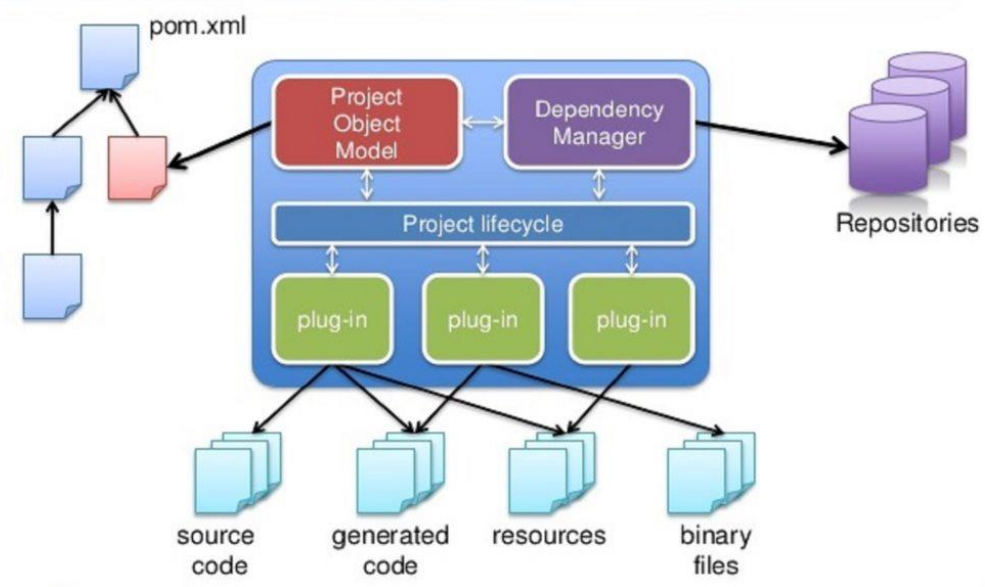

Ilustración 4 *Esquema funcional Maven [21]*

Existen muchas soluciones debido a la propia naturaleza de los diferentes lenguajes y arquitecturas, todo orientado a una configuración basada en fichero e interacción sencilla mediante una interfaz de comando. Así, son ejemplos bastante conocidos Ant, Maven o Graddle para los desarrolladores Java o por ejemplo Gulp y Grunt para aquellos que trabajan con Javascript. A día de hoy, es raro que no encontremos una herramienta de construcción automática en un lenguaje de programación ya maduro.

- Repositorio central de artefactos para el control, gestión y almacenamiento no sólo de las dependencias de los proyectos (propias o de terceros) sino de los artefactos generados, ya sean en forma de nuevas dependencias como de aplicaciones. En este caso, y a diferencia del sistema de control de versiones, no hablamos de un repositorio de código fuente, hablamos un repositorio de empaquetados o compilados.

  Son interesantes estos sistemas ya que no sólo permite almacenar y centralizar nuestras dependencias, sino también gestionar y controlar que dependencias están disponibles para los desarrolladores. Esto que puede parecer no muy relevante, en principio, es muy interesante para aquellas personas o equipos encargados de la seguridad, ya que permite gestionar qué dependencias o artefactos se consideran "seguros" para el desarrollo como para aplicar parches que corrijan errores de seguridad.

  Estos sistemas ayudan a la consistencia en el flujo dentro de la IC ya que evitan problemas con las diferentes dependencias que diferentes programadores podrían utilizar en un proyecto. Por citar alguno, ejemplos conocidos de administradores de repositorios de artefactos son Nexus OSS o JFrog.

- Servidor de integración, para "algunos" todavía no esencial, pero que nadie puede discutir las ventajas que ofrece. Actuando casi como un director de orquesta, nos permite la integración de cada uno de los componentes del sistema de IC a través de procesos centralizados y automatizados en el que cada una de las tareas que dichos sistemas cumplen dentro de la integración (código potencialmente liberable) quedan encapsuladas como un todo en un único proceso. Dicho proceso no sólo genera el posible entregable sino también proporciona retroalimentación de la ejecución de cada una de las tareas y del resultado final.

  Aunque esta tarea podría realizarse por parte de cada desarrollador, con las herramientas y scripts necesarios, no tiene sentido que sea así ya que puede llevar a discrepancias y errores en el proceso, y en el comportamiento final del software por la diferente naturaleza y configuración del entorno donde se lleva a cabo con respecto a otro. Para evitar todo esto, y que el entorno donde se ejecute el proceso sea siempre el mismo, es clave la figura del servidor de integración.

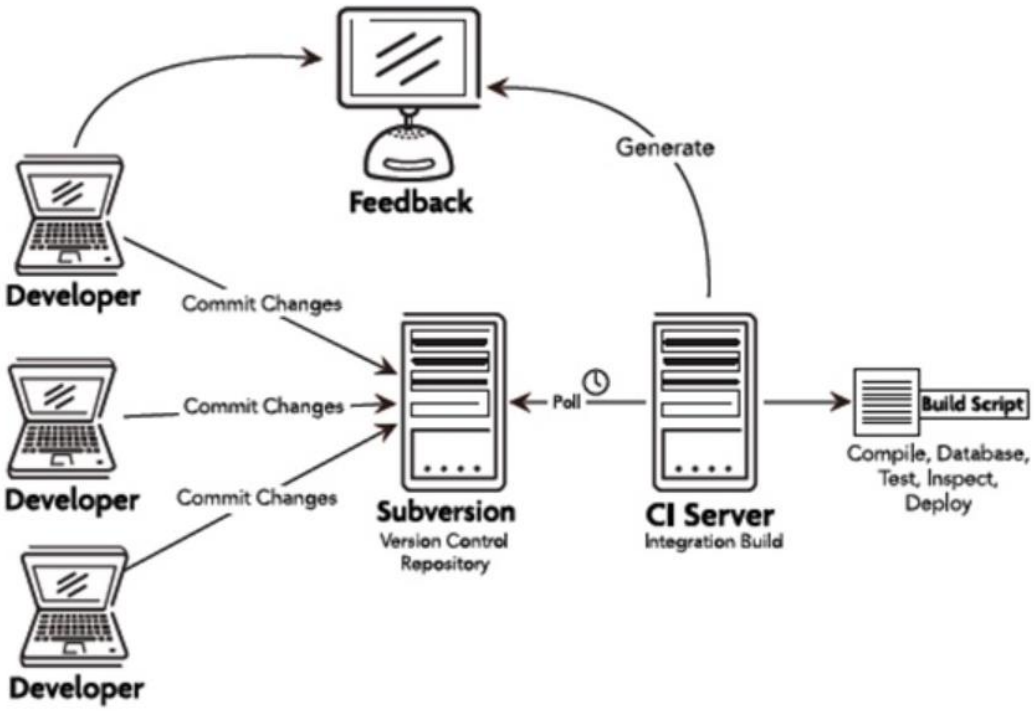

Ilustración 5 *Ejemplo entorno IC con servidor de integración [17]*

Como con el resto de elementos del entorno de IC que hemos visto, existe una variedad amplia de herramientas disponibles para cumplir esta función, en algunos casos orientadas a arquitecturas concretas de entorno o lenguaje, siendo ejemplos muy conocidos, por su alto uso en el sector, Jenkins o Bamboo. De hecho, incluso herramientas cuyo objetivo principal dentro del entorno IC no es la de ser servidores de integración comienzan a invadir las responsabilidades de estos proporcionando cada día más funcionalidades a los usuarios para la integración y automatización de procesos.

Aunque hemos comentado el mínimo esperado de herramientas o componentes en un entorno de IC (si partimos del objetivo de la propia definición de IC) es interesante ir al menos un poco más allá teniendo en cuenta las buenas prácticas y requerimientos si se quiere aplicar DevOps en un proceso de desarrollo de software. Así, hay que mencionar el análisis de código. Un aspecto interesante, y cada día más importante, no sólo para evitar malas prácticas de programación sino también para evitar posibles agujeros de seguridad. En este punto, las herramientas de análisis de código estático cobran protagonismo. No sólo por ser elemento evaluador del código o detector de errores de seguridad, sino también como elemento didáctico y mejora continua para los propios desarrolladores y del software en el que trabajan. Ejemplos de este tipo de herramientas ampliamente usado a nivel profesional son SonarQube o Kiuwan.

2.3. Contenedores software [4][23]

Hemos visto la cantidad de agentes que pueden llegar a intervenir en un entorno de IC y no siempre es posible que cada uno de ellos puedan estar hospedados en máquinas independientes. De hecho, en muchos casos sería un desperdicio de recursos para un

uso exclusivo. Por ello, es práctica habitual que diferentes herramientas o servicios compartan la misma máquina para optimizar recursos. Esta situación genera, sin embargo, la posibilidad real de problemas futuros o presentes tanto en la configuración de las diferentes herramientas como incompatibilidades entre dependencias y/o versiones. En esos casos, la práctica habitual era el uso de máquinas virtuales que permiten dentro de una misma máquina anfitrión, tener diferentes sistemas aislados corriendo su propio sistema operativo, dependencias y configuraciones. Sin embargo, esta práctica requiere de costes elevados de recursos, lo cual, en parte va en contra del objetivo de aplicarla. Aun así, existen situaciones en las que a día de hoy es la única técnica aplicable, pero cada vez son más escasas gracias a tecnologías como la de contenedores de software.

¿Qué son los contenedores de software?

Si buscamos una analogía en el mundo real para explicar qué son los contenedores software (y el objetivo detrás de esta tecnología) podemos hablar de esos típicos containeres que vemos siendo transportados en barco de un sitio a otro. No importa su contenido sino su forma modular para ser almacenados y transportados de un sitio a otro.

Ilustración 6 Analogía concepto contenedor software

Algo similar ocurre con los contenedores software. Dentro de ellos alojamos todas las dependencias que nuestra aplicación necesite para ser ejecutada: desde el propio compilado, empaquetado o ejecutable, a las librerías del sistema, hasta el entorno de ejecución y cualquier tipo de configuración. Desde fuera del contenedor no necesitamos mucho más. Dentro están aislados para ser ejecutados en cualquier lugar.

Los contenedores son la solución al problema habitual, por ejemplo, de moverse entre entornos de desarrollo como puede ser una máquina local o en un entorno real de

producción. Podemos probar de forma segura una aplicación sin preocuparnos de que nuestro código se comporte de forma distinta. Y esto es debido a que, como hemos dicho antes, todo lo que necesitamos está dentro de ese contenedor.

Los contenedores representan un mecanismo de empaquetado lógico donde las aplicaciones tienen todo lo que necesitan para ejecutarse. Pero entonces, ¿en qué se diferencia un contenedor software y una máquina virtual?

Contenedores VS Virtualización

Gracias a la virtualización somos capaces, usando una misma máquina, de tener distintas máquinas virtuales con su propio sistema operativo invitado. Todo ello ejecutándose en un sistema operativo anfitrión con acceso virtualizado al hardware. La virtualización es una práctica habitual en servidores para alojar diferentes aplicaciones o en nuestro propio entorno de trabajo para ejecutar distintos sistemas operativos. Un ejemplo de uso lo encontramos en muchos alojamientos de hosting que se han basado en crear máquinas virtuales limitadas sobre el mismo servidor para alojar servidores web de forma aislada, siendo compartido por decenas de clientes.

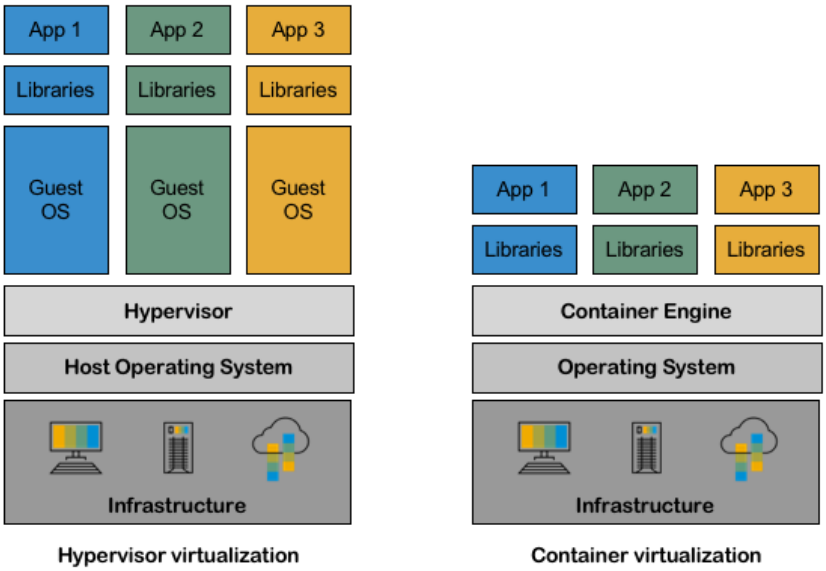

Ilustración 7 *Máquina virtual vs Contenedor software [15]*

En contraposición a las máquinas virtuales, los contenedores software se ejecutan sobre el mismo sistema operativo anfitrión de forma aislada, pero sin necesidad de un sistema operativo propio, ya que comparten el mismo Kernel o núcleo, lo que los hace mucho más ligeros. Al tener que emular todo un sistema operativo, esto puede suponer varios gigas de memoria, lo cual representa un primer punto en el ahorro de coste.

Básicamente, los contenedores software se basan en dos mecanismos para aislar procesos en un mismo sistema operativo. El primero de ellos, se trata de los "namespace" que provee Linux, lo que permite que cada proceso solamente sea capaz de ver su propio sistema "virtual" (ficheros, procesos, interfaces de red, hostname,

etc.). Y el segundo concepto son los “CGroups”, por el cual somos capaces de limitar los recursos que puede consumir (CPU, memoria, ancho de banda, etc).

Pero no todo son ventajas. A priori, y de un primer vistazo, podríamos llegar a la conclusión de que un problema o error en el kernel de la máquina anfitriona generaría la caída de todos los contenedores que estuviesen ejecutándose en ella, lo cual, sin embargo, no difiere de lo que ocurriría en un sistema anfitrión de máquinas virtuales en donde fallase el hipervisor o su sistema operativo. Lo que sí supone una desventaja, y ha introducido un nuevo reto en el uso de esta tecnología, son las políticas, configuración y buenas prácticas de seguridad en estos sistemas, ya que estamos compartiendo kernel, y una mala implementación puede suponer un riesgo para la máquina. Estos dos ejemplos, dan una idea de que no es una solución perfecta, evidentemente, pero aspectos derivados de su concepto como el rendimiento, la portabilidad, la gestión y administración o el escalado, hacen a esta tecnología muy atractiva.

De aplicaciones monolíticas a microservicios

El concepto de contenedor, y su desarrollo, ha llevado como consecuencia a la aparición de nuevos conceptos y paradigmas dentro del desarrollo de aplicaciones y romper con el pasado: los microservicios.

La definición clásica de una aplicación monolítica se refiere a un conjunto de componentes acoplados totalmente y que tienen que ser desarrollados, desplegados y gestionados como una única entidad. Prácticamente encajonados en un mismo proceso que puede ser muy difícil de escalar, ya no sólo de forma vertical añadiendo más CPU y memoria, sino también de manera horizontal con más máquinas. A todo esto hay que añadir el hecho de que para desarrollar, un programador necesita tener todo ese código y ejecutar las pruebas levantando un única instancia con todo, a pesar de que el cambio que quiera realizar sea mínimo. Sin hablar de lo costoso que se convierte cada vez que se quiera hacer una nueva entrega tanto en desarrollo, pruebas como despliegue.

En contraposición a esto, surgió el concepto de microservicios que permite que varias pequeñas aplicaciones independientes se comuniquen entre sí para ofrecer una funcionalidad específica concreta. Tenemos el bien conocido caso de Netflix, por ejemplo, una de las compañías que comenzó a hacer uso de forma más intensiva de los microservicios casi antes de su propia definición. Aunque no se conoce una cifra concreta, se estima a partir de algunos de sus datos en charlas técnicas que cuente con entre 700-1000 microservicios. Podríamos hablar de la presencia de un microservicio que se encargue de servir vídeo según la plataforma o dispositivo desde donde accedemos, otro que se encargue del historial de contenido que hayamos visto, otro para las recomendaciones, otro para el pago de la suscripción, etc. Todos ellos pueden convivir en una nube de microservicios y comunicarse entre sí. Este enfoque hace que no sea necesario modificarlos todos a la vez, ante evolutivos o correctivos de uno de ellos, y resultando en una administración o gestión más óptima y con mejores tiempos de respuesta, ya que podemos escalar algunos de los microservicios o reemplazar

contenedores prácticamente al vuelo en caso de fallo, según las necesidades de cada momento y de carga. Ese desacople y delegación de funciones entre diferentes microservicios proporciona esa flexibilidad [2].

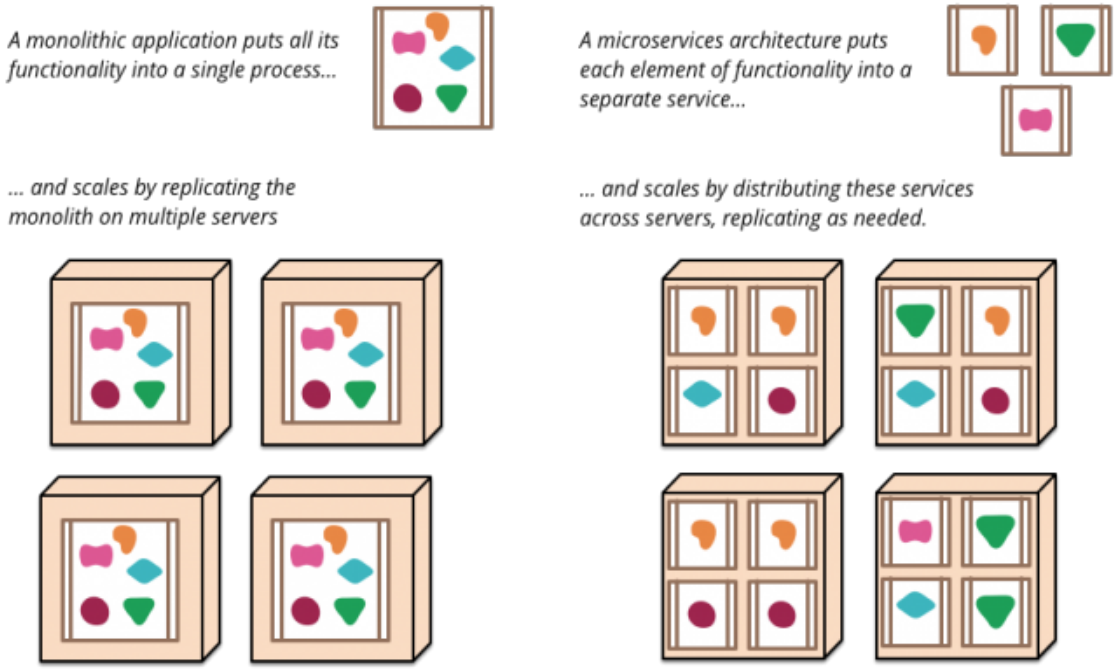

Ilustración 8 *Aplicación monolítica vs Arquitectura de microservicios [2]*

Docker

Aunque el concepto de los contenedores de software o la virtualización a nivel de sistema operativo tiene su génesis a principios de los 80, no es hasta relativamente hace poco, con la entrada Docker en 2013 que se inicia el despegue más global de esta tecnología. Tal es la importancia de la llegada de Docker que con el tiempo se ha convertido en el estándar de facto para construir y compartir aplicaciones contenerizadas. Eso no significa que no existan otros proyectos o alternativas a Docker, sin embargo, ni su uso ni su estado de madurez se puede comparar con el de este.

Docker es una plataforma abierta para el desarrollo, distribución y ejecución de aplicaciones. Separa las aplicaciones de la infraestructura proporcionando un entorno aislado para el empaquetado y ejecución de éstas denominado contenedor. Los contenedores son objetos ligeros y portables que contienen todo lo que la aplicación necesita, lo que permite la ejecución simultánea de muchos contenedores en una máquina anfitriona sin necesidad de preocuparse qué está instalado en ella. Además, Docker proporciona herramientas y una plataforma para la administración de todo el ciclo de vida de esos contenedores.

Utiliza una arquitectura cliente-servidor, donde el cliente se comunica con un demonio que se encarga de la construcción, ejecución y distribución de los contenedores. Tanto el cliente como el demonio puede ejecutarse sobre el mismo sistema o el cliente puede conectarse a un demonio remoto (lo cual da gran flexibilidad) cada uno en un sistema

diferente. En cualquiera de los casos, la comunicación entre ambos es a través de un REST API sobre sockets UNIX o una interfaz de red.

Es interesante diferenciar dentro los objetos o actores presentes en Docker a dos de ellos: imagen y contenedor. Una **imagen** es una plantilla de sólo lectura con instrucciones para la creación de un contenedor. Esto permite la creación de imágenes basadas en otra, pero con personalizaciones adicionales. Para construir una imagen hay crear un fichero Dockerfile con una sintaxis definida y sencilla que define cada uno de los pasos para crearla y se ejecute. Cada instrucción crea una capa en la imagen, y en caso de modificación, sólo aquellas capas que cambian son reconstruidas. Esto es lo que hace a las imágenes ligeras, pequeñas y rápidas comparadas con otras tecnologías de virtualización. En cambio, un **contenedor** es una instancia de una imagen. Un contenedor se puede crear, arrancar, detener, mover o borrar. Puedes conectarlos a una o varias redes, añadirles volúmenes de almacenamiento para persistencia o incluso crear nuevas imágenes basadas en su estado actual. Por defecto, un contenedor está relativamente bien aislado de otros contenedores y la máquina anfitriona. Pero lo que es importante es que un contenedor está definido por su imagen y las opciones de configuración que se proporcionan cuando se crea o se arranca. Cuando un contenedor se elimina, cualquier cambio en su estado no es almacenado de manera persistente y se pierde (evidentemente, es posible definir objetos en Docker que permiten la persistencia de datos cuando sea necesario).

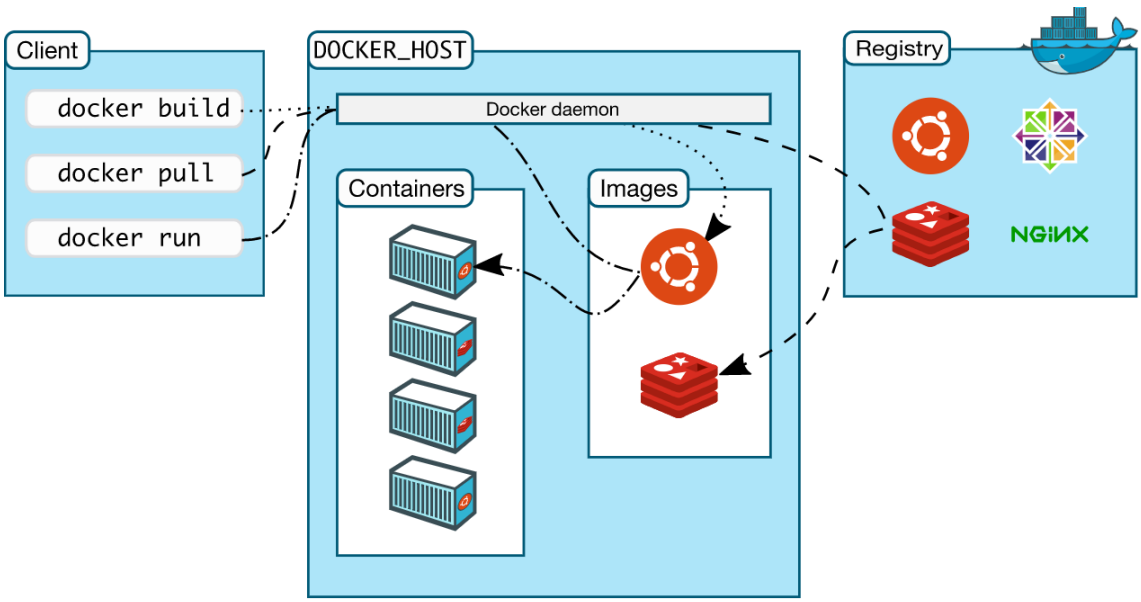

Ilustración 9 *Arquitectura de Docker (Docker, 2021)*

Para terminar, comentar dos elementos presentes en su arquitectura:

- **Registro Docker**, como repositorio para almacenar y distribuir imágenes siendo Docker Hub el registro público por defecto que el cliente utiliza para buscar imágenes. Es posible crear nuestros propios registros privados para almacenar nuestras imágenes, lo cual es muy interesante, pero sean públicos o privados, el

registro y los comandos del cliente de Docker nos va a permitir operaciones para obtener, publicar y actualizar imágenes de manera similar a como lo haríamos en un sistema de control de versiones.

- Docker Compose, como otro cliente Docker para la definición y ejecución de aplicaciones multicontenedor (o no). Mediante un fichero YAML (formato de serialización de datos amigable que destaca por su legibilidad y los tipos de datos con los que permite trabajar) es posible configurar en todos sus aspectos los servicios de una aplicación (como veremos más adelante), lo que permite de una manera sencilla y con un simple comando crear e iniciar una aplicación a partir de esa configuración.

3. Diseño y dockerización de servicios

Como ya se introdujo en el capítulo 1, dentro de la sección 1.2 Objetivos, el objetivo final del trabajo es implementar un entorno de integración continua mediante una arquitectura de contenedores de software. En este caso, y por tener un objetivo lo menos abstracto posible, se ha tomado como referencia un caso real de un entorno de IC con una arquitectura compuesta de varias máquinas y servicios. En la siguiente sección, comentaremos la arquitectura de herramientas o servicios que componen el entorno para posteriormente detallar un poco cada una de ellas y el proceso que se ha llevado a cabo para su contenerizado e integración.

El modo en que orientamos poder evaluar y construir ese entorno, viene dado por dar respuesta a dos flujos comunes de trabajo dentro del desarrollo de software en prácticas de DevOps como son:

- Generación, centralización y almacenamiento de un paquete o dependencia software.
- Generación, centralización, almacenamiento y despliegue de una aplicación.

Poniendo en práctica estas ideas, se trataría de conseguir que una vez que un desarrollador entregue sus cambios en el código fuente, todo el proceso de compilado, empaquetado, almacenamiento e incluso despliegue sea un proceso automático y transparente para este. Es decir, con la mínima interacción por parte del desarrollador. Todo un proceso, que como ya hemos visto en el anterior capítulo, es clave en IC y la práctica de DevOps.

La solución técnica seleccionada para la implementación de contenedores software ha sido Docker que, como ya vimos en el capítulo anterior, es una solución ampliamente utilizada y con todas las herramientas necesarias para ese objetivo. El proceso de dockerización del entorno pasará por la generación y configuración de las imágenes Docker necesarias para portar los servicios presentes, y la configuración de una aplicación multicontenedor que haga uso de estas y se comporte de manera análoga a su contraparte no virtualizada al ser ejecutada.

3.1. Arquitectura

La arquitectura de la que partimos es un caso real con varias máquinas anfitrión sobre sistema Windows Server no virtualizadas. Dicho entorno está compuesto por las siguientes herramientas:

- Gitlab, como herramienta de control de versiones sobre Git.
- Maven, como herramienta de construcción automática de artefactos y ejecución de test. Los principales desarrollos a los que está orientado este entorno son desarrollos en Java y dentro del ecosistema es la herramienta más extendida para dicho lenguaje.

- **SonarQube**, como herramienta de análisis y evaluación de código.
- **Nexuss OSS,** como herramienta de almacenamiento o repositorio central de artefactos.
- **JBoss EAP,** como servidor de aplicaciones (al tratarse de aplicaciones Java EE)
- **Jenkins,** como servidor de integración.

Salvo Jenkins y Maven, que compartían máquina, el resto se encontraban alojados en máquinas independientes.

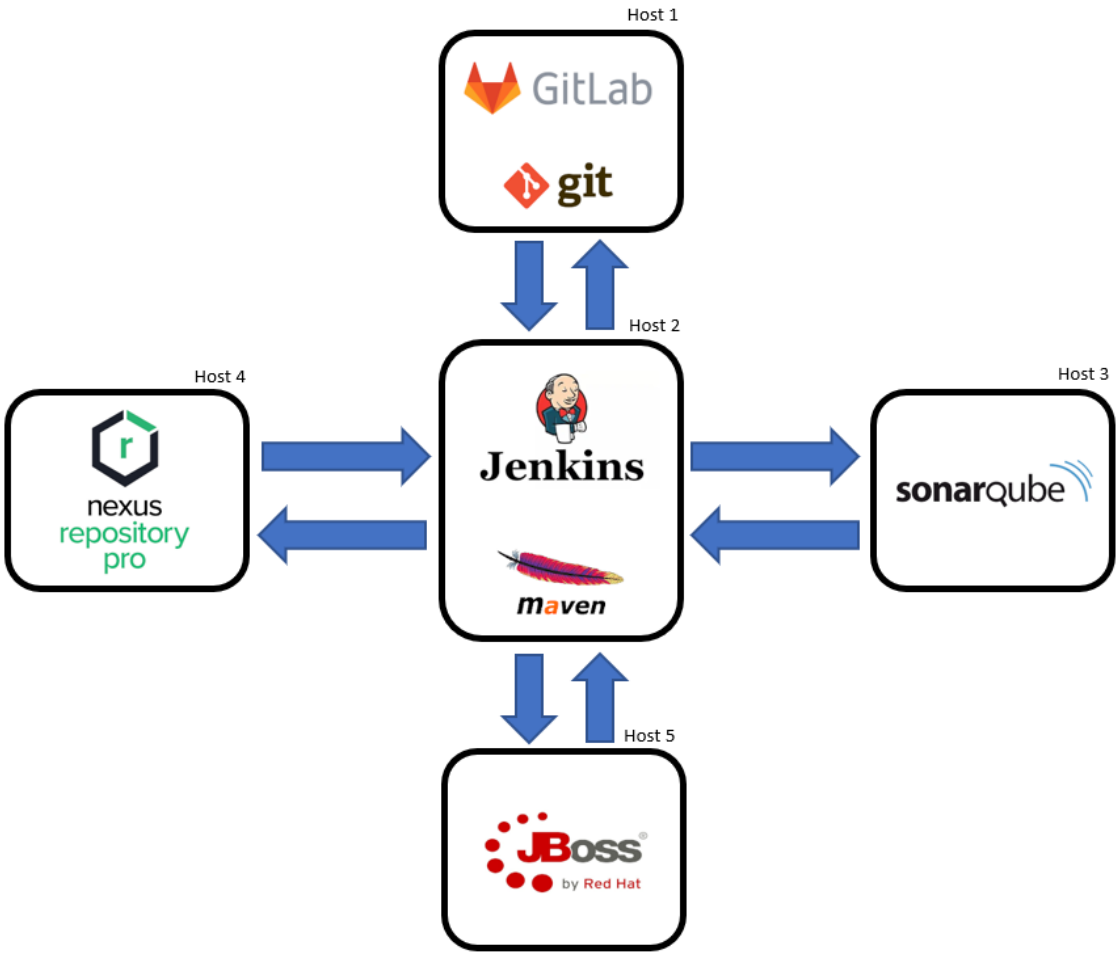

Ilustración 10 *Esquema entorno IC*

Como elemento central integrador tenemos a Jenkins que es el encargado de lanzar todas las acciones sobre los diferentes servicios o herramientas involucradas en el proceso. En dicho entorno, el proceso principal de IC constaba de la obtención de cambios del código fuente en Java en el sistema de control de versiones Git para realizar seguidamente la construcción y test del mismo (obteniendo las dependencias necesarias del sistema centralizado de dependencias o artefactos) tras lo que se realizaba un análisis de código estático y posterior despliegue de la aplicación.

En contraposición, lo que se pretende obtener es a una arquitectura análoga pero con un único host anfitrión de las versiones contenerizadas de las herramientas que

componen el entorno. A priori, podríamos pensar que el proceso de pasar a contenedores software va a ser una relación 1 a 1 por servicio que se vaya a utilizar, los propios requisitos de cada herramienta y/o el nivel de "microservicio" al que queramos llegar (a la hora de aislar procesos) puede hacer que esa relación crezca. Sirva de ejemplo lo que va a ocurrir con la dockerización de Maven. En ese caso, dadas las características y requisitos del proceso de construcción automática se entendió que sería interesante que Maven fuese un contenedor independiente más, y no fuese una dependencia intrínseca al contenedor de Jenkins. Más adelante lo veremos en detalle.

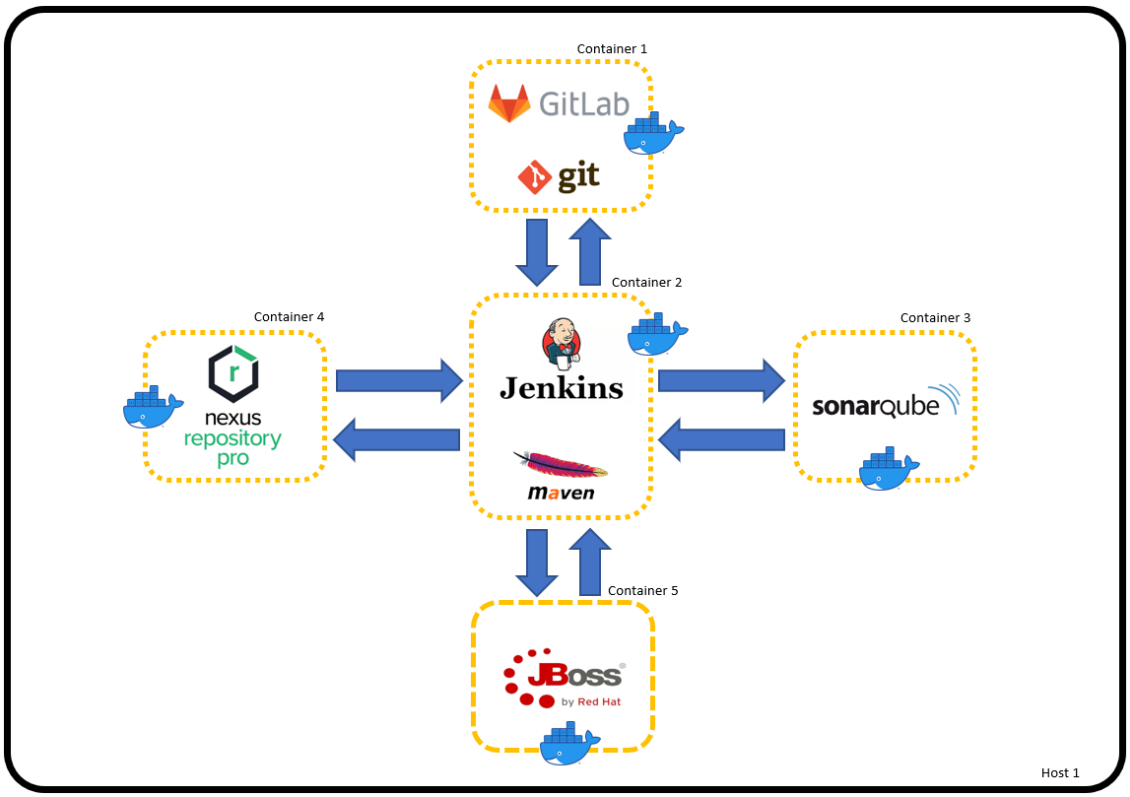

Ilustración 11 *Esquema simple entorno de IC dockerizado*

El proceso para obtener la versión virtualizada del entorno consistirá en la obtención de las imágenes Docker necesarias que representen a cada una de las herramientas o servicios de este y la generación de un fichero Docker Compose que defina y configure la ejecución de estas como una única aplicación multicontenedor o multiservicio. La implementación se ha realizado sobre un sistema Linux, más concretamente sobre una distribución CentOS 7, y evidentemente el prerrequisito para empezar es la instalación tanto el demonio de Docker (convirtiendo la máquina host en una máquina host de Docker), del CLI de Docker para poder interactuar con él, y Docker Compose como herramienta de configuración (mediante ficheros yml) y ejecución de servicios.

En ningún caso, se va a detallar la configuración de los servicios a nivel administrativo a través de sus interfaces web, centrándonos exclusivamente en los aspectos relacionados con la arquitectura y la implementación del entorno mediante contenedores.

3.2. Gitlab

Gitlab es un servicio web de control de versiones y desarrollo de software colaborativo basado en Git. Además de gestor de repositorios, el servicio ofrece también alojamiento de wikis y un sistema de seguimiento de errores, todo ello publicado bajo una Licencia de código abierto [24].

Es una suite completa que permite gestionar, administrar, crear y conectar los repositorios con diferentes aplicaciones y hacer todo tipo de integraciones con ellas, ofreciendo un ambiente y una plataforma en cual se puede realizar varias etapas dentro del ciclo de vida de desarrollo de una aplicación y de DevOps [6].

Requisitos

A nivel de servicio dentro del entorno, Gitlab se está utilizando como repositorio central de código fuente. En principio no se utiliza como servidor de integración por lo que su instalación y configuración no va más allá de la estándar y la necesaria para dar acceso a los potenciales usuarios y al servidor integrador de nuestro entorno, en este caso la herramienta Jenkins.

Debe permitir lanzar la ejecución automática de procesos, ante eventos en el repositorio como acciones de *push* o *merge request*. Gracias a la funcionalidad de Webhooks (incluida ya en Gitlab), esto es posible realizando peticiones web con información hacia otros servicios que actúan como receptores desencadenando nuevos procesos o acciones.

Evidentemente, es necesaria la instalación de Git sobre la que actúa Gitlab.

Dockerización

Para la "dockerización" de la herramienta, como con el resto, nos apoyamos en Docker Hub como registro central público de imágenes. Antes que crear nuevas imágenes es conveniente hacer una búsqueda que nos ahorre tiempo, ya que es habitual encontrar imágenes ya preparadas de aplicaciones y publicadas por las propias compañías responsables de estas.

En el caso de Gitlab, existe una imagen de la versión Community Edition o CE, que se ajusta a los requerimientos a cubrir. Con ello, tan sólo necesitamos configurar nuestro servicio mediante Docker Compose teniendo en cuenta la documentación de la imagen.

docker.compose.yml

```
version: "3"

services:

  gitlab:
    image: 'gitlab/gitlab-ce:latest'
    restart: always
```

```
ports:
  - "8083:80"
# Persistencia de datos
volumes:
  - "/srv/gitlab/config:/etc/gitlab"
  - "/srv/gitlab/logs:/var/log/gitlab"
  - "/srv/gitlab/data:/var/opt/gitlab"
```

Como podemos ver, el contenedor hará uso de lo que se denomina *bind mount* para la persistencia de datos en contra de la práctica habitual que son los volúmenes. A parte de las ventajas o no de cada una de ellas, *bind mount* (como en este caso) monta una ruta absoluta de la máquina anfitriona en el contenedor sin que Docker intervenga en su administración. En cambio, en el caso de los volúmenes lo que se crea es un directorio en el directorio de almacenaje o persistencia de Docker en la máquina anfitriona, y Docker será el encargado de administrar el contenido.

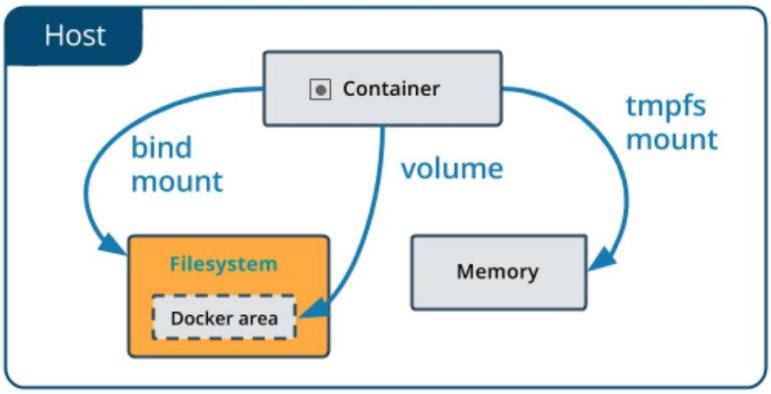

Ilustración 12 *Bind mount VS Volume [4]*

En este caso, vemos que se han montado las rutas correspondientes a la persistencia de los datos, logs y configuración de Gitlab; lógico ya que cuando un contenedor es eliminado los datos no presentes en la imagen se pierden.

En otros aspectos, se expone el puerto de protocolo web para acceder desde la máquina anfitriona al servicio.

3.3. Maven

Maven es una herramienta de software para la gestión y construcción de proyectos Java. Tiene un modelo de configuración de construcción simple, basado en un formato XML [26].

Maven utiliza un Project Object Model (POM) para describir el proyecto de software a construir, sus dependencias de otros módulos y componentes externos, y el orden de construcción de los elementos. Viene con objetivos predefinidos para realizar ciertas tareas claramente definidas, como la compilación y test del código y su empaquetado [10].

Requisitos

Dentro del flujo de IC, Maven va a ser invocado durante las fases de construcción, test y empaquetado de la aplicación. La naturaleza de su ejecución va a requerir dependencias concretas en cuanto a la versión de Maven que utilizamos como de la versión de Java que se va a utilizar en la compilación del artefacto a construir.

Es por ello, que puede resultar interesante la generación de imágenes basadas en versión de Maven y Java que se vayan a utilizar para cada proyecto. En este caso, los proyectos Java involucrados necesitan una versión de Maven 3 y Java 1.8.

Además, el servicio debe ser accesible desde el servidor de integración (que en este caso es Jenkins).

Dockerización

Teniendo en cuenta lo visto en el punto anterior, el primer paso que vamos a llevar a cabo es la creación de una imagen Maven basada en OpenJDK 1.8 sobre la que instalamos Maven.

Dockerfile

```
FROM openjdk:8-alpine
ARG MAVEN_VERSION=3.8.1
ARG USER_HOME_DIR="/maven"
# speed up Maven JVM a bit
ENV MAVEN_OPTS="-XX:+TieredCompilation -XX:TieredStopAtLevel=1"
RUN apk update && \
apk add --no-cache curl tar bash && \
mkdir -p /usr/share/maven && \
mkdir -p /maven && \
curl -fsSL https://ftp.cixug.es/apache/maven/maven-3/$MAVEN_VERSION/binaries/apache-
maven-$MAVEN_VERSION-bin.tar.gz | tar -xzC /usr/share/maven --strip-components=1 && \
ln -s /usr/share/maven/bin/mvn /usr/bin/mvn && \
#Añadimos usuario/grupo Jenkins para su consumo posterior por el servidor de
#integracion
addgroup -S --gid 1000 jenkins && \
adduser -S -D --uid 1000 jenkins jenkins
ENV MAVEN_HOME /usr/share/maven
ENV MAVEN_CONFIG "$USER_HOME_DIR/.m2"
USER jenkins
ENTRYPOINT ["/usr/bin/mvn"]
```

Mediante las herramientas de Docker, construimos la imagen de manera que esté disponible para ser ejecutada por el demonio. Es interesante, en este punto y con cualquier imagen que se genere, que esta esté disponible en un repositorio o registro Docker por lo que siempre es recomendable su publicación.

Con respecto a la configuración como servicio, al contrario que en el caso visto para Gitlab, la manera en que vamos a consumir Maven va a ser a demanda, es decir, crearemos o ejecutaremos un contenedor cada vez que lo necesitemos a partir de esta nueva imagen, de manera que cuando termine de ejecutar la tarea, que en cada

momento se necesite, este contenedor finalice y se elimine. Se verá en detalle dentro de la sección dedicada a la integración de servicios y Jenkins, posteriormente.

3.4. SonarQube

SonarQube es una plataforma para evaluar código fuente. Es software libre y usa diversas herramientas de análisis estático de código fuente para obtener métricas que pueden ayudar a mejorar la calidad del código de un programa [28].

Su arquitectura se basa en servicio web en donde se almacena, se consulta y está disponible toda la información de análisis de los proyectos software y desde donde se puede configurar los parámetros de evaluación y umbrales para los análisis, y por otro lado, la herramienta CLI propia que realiza o lanza el análisis.

Requisitos

Por un lado, necesitaremos lanzar el análisis de código Java desde el directorio de los fuentes de código. Y por otro, un servicio web que persista y permita consultar los análisis de los proyectos que se realicen.

Dada las características de SonarQube, es interesante diferenciar ambos servicios, ya que en el caso de la herramienta CLI existen diferentes versiones de esta en función de lenguaje a analizar, además de que como en el caso de Maven va a ser un servicio a demanda y no necesita estar levantado de manera permanente.

Dockerización

Diferenciaremos por un lado la dockerización del servicio web o del servidor SonarQube y por otro el del servicio de análisis o escáner de código.

En el primer caso, al igual que ya vimos con Gitlab, en el registro público de Docker tenemos una imagen oficial del servidor ya preparada para ser utilizada por lo que sólo tendremos que configurarla a nivel de servicio para ser ejecutada con Docker Compose teniendo en cuenta su documentación.

docker.compose.yml

```
version: "3"

services:

  sonarqube:
    image: sonarqube:8-community
    depends_on:
      - db_sonar
    environment:
      SONAR_JDBC_URL: jdbc:postgresql://db_sonar:5432/sonar
      SONAR_JDBC_USERNAME: sonar
      SONAR_JDBC_PASSWORD: sonar
    volumes:
      - sonarqube_data:/opt/sonarqube/data
```

```
      - sonarqube_extensions:/opt/sonarqube/extensions
      - sonarqube_logs:/opt/sonarqube/logs
    ports:
      - "9000:9000"

  db_sonar:
    image: postgres:12
    environment:
      POSTGRES_USER: sonar
      POSTGRES_PASSWORD: sonar
    volumes:
      - postgresql:/var/lib/postgresql
      - postgresql_data:/var/lib/postgresql/data
volumes:
  sonarqube_data:
  sonarqube_extensions:
  sonarqube_logs:
  postgresql:
  postgresql_data:
```

Cabe resaltar en la configuración la dependencia del servidor web con un servidor de base de datos postgres. Esto se realiza mediante la declaración de un servicio *db_sonar* y la declaración *depends_on* del servicio "sonarqube" que produce un comportamiento específico y necesario de orden de arranque y parada de los servicios [3]. Al contrario que en el caso de Gitlab, se han utilizado volúmenes para la persistencia de datos de cada uno de los servicios. Se expone el puerto de protocolo web para acceder desde la máquina anfitriona al servicio.

Para el caso de la herramienta de escáner, también tenemos en Docker Hub una imagen oficial para la ejecución de análisis sobre código Java, por lo que no necesitamos ningún proceso adicional nada más que ejecutarla de manera correcta dentro del flujo de IC y que veremos posteriormente.

3.5. Nexus OSS

Nexus es un repositorio web centralizado para la publicación y almacén de artefactos software. Es una suite completa que permite gestionar, administrar, crear y conectar repositorios y aplicaciones, haciendo todo tipo de integraciones con ellas.

Requisitos

Dentro del flujo de IC necesitamos que el servicio sea accesible por la máquina de integración para la obtención de dependencias que pudiesen estar almacenadas y ser necesarias durante la fase de construcción, y el posterior almacenaje de los artefactos que pudiesen generarse como resultado de todo el proceso.

Evidentemente, es necesario que el repositorio sea accesible por los desarrolladores para la descarga de dependencias de sus proyectos.

Dockerización

Como en casos anteriores, en el registro público de Docker tenemos una imagen oficial del servicio, por lo que sólo tenemos que llevar a cabo su configuración a nivel de

contenedor para posteriormente llevar a cabo una configuración a nivel de administración y garantizar acceso con los privilegios adecuados a Jenkins.

docker-compose.yml

```
version: "3"

services:

  nexus:
    image: 'sonatype/nexus3:latest'
    restart: always
    ports:
       - "8081:8081"
    volumes:
      - 'nexus-data:/nexus-data'

volumes:
  nexus-data:
```

Una configuración muy sencilla que expone el puerto protocolo web de la aplicación para el acceso desde la máquina anfitriona y la definición de un volumen para la persistencia de datos.

3.6. JBoss Wildfly

Anteriormente conocido como JBoss AS, o simplemente JBoss, es un servidor de aplicaciones Java de código abierto implementado en la especificación Java EE. Al estar basado en Java, JBoss puede ser utilizado en cualquier sistema operativo para el que esté disponible la máquina virtual de Java [29].

Requisitos

El servidor de aplicaciones debe ser capaz de desplegar de manera automática aplicaciones Java desarrolladas en JDK 8. Adicionalmente, por la arquitectura y el framework utilizado en las aplicaciones es necesario una configuración adicional a nivel de entorno sobre las propiedades o configuraciones de estas para su ejecución.

Dockerización

JBoss Wildfly posee un mecanismo por defecto de autodespliegue mediante el escaneo periódico de una ruta dentro del servidor [14]. Cualquier aplicación que se sitúe allí disparará de manera automática el despliegue de la misma, por lo que el requerimiento queda cubierto.

Por otro lado, aunque en el registro público de Docker existe una imagen ya preparada de JBoss Wildfly, esta está basada en OpenJDK11. Dado que el entorno en el que nos basamos utiliza Java 8, y para evitar problemas de incompatibilidad de dependencias, es lógico que tengamos que generar una imagen propia basada en OpenJDK8 y aprovechar para realizar las configuraciones adicionales que necesitamos para el despliegue de

aplicaciones. Así, basándonos en el Dockerfile de esta imagen generamos nuestra imagen personalizada

Dockerfile

```
# Use latest jboss/base-jdk:8 image as the base
FROM jboss/base-jdk:8

# Set the WILDFLY_VERSION env variable
ENV WILDFLY_VERSION 23.0.2.Final
ENV WILDFLY_SHA1 cd79cddc334cd58c7b9a8fc65439d4152c8d2fb8
ENV JBOSS_HOME /opt/jboss/wildfly

USER root

# Add the WildFly distribution to /opt, and make wildfly the owner of the extracted
tar content
# Make sure the distribution is available from a well-known place
RUN cd $HOME \
    && curl -O https://download.jboss.org/wildfly/$WILDFLY_VERSION/wildfly-
$WILDFLY_VERSION.tar.gz \
    && sha1sum wildfly-$WILDFLY_VERSION.tar.gz | grep $WILDFLY_SHA1 \
    && tar xf wildfly-$WILDFLY_VERSION.tar.gz \
    && mv $HOME/wildfly-$WILDFLY_VERSION $JBOSS_HOME \
    && rm wildfly-$WILDFLY_VERSION.tar.gz \
    && chown -R jboss:0 ${JBOSS_HOME} \
    && chmod -R g+rw ${JBOSS_HOME} \
    && $JBOSS_HOME/bin/add-user.sh admin-user jboss

# Configuracion específica por entorno para el framework
ADD properties /opt/jboss/wildfly/standalone/
COPY standalone.conf /opt/jboss/wildfly/bin/

# Ensure signals are forwarded to the JVM process correctly for graceful shutdown
ENV LAUNCH_JBOSS_IN_BACKGROUND true

USER jboss

# Expose the ports in which we're interested
EXPOSE 8080

# Set the default command to run on boot
# This will boot WildFly in standalone mode and bind to all interfaces
CMD ["/opt/jboss/wildfly/bin/standalone.sh", "-b", "0.0.0.0"]
```

Lo importante, fuera de la declaración de la imagen base JBoss con OpenJDK8, es la configuración personalizada del servidor

```
ADD properties /opt/jboss/wildfly/standalone/
COPY standalone.conf /opt/jboss/wildfly/bin/
```

Por ejemplificar la personalización, simplemente se está añadiendo nuestro propio fichero de configuración *standalone.conf*, con las modificaciones que hemos necesitado para la ejecución del servidor, y unas carpetas con ficheros de propiedades personalizadas necesarios para las aplicaciones.

Con esto, tan sólo nos queda la configuración del servicio, que queda reducida a la exposición del puerto del protocolo web y la definición de un volumen para la persistencia de datos, en este caso la ruta de despliegue automático de las aplicaciones.

docker-compose.yml

```
version: "3"

services:

  jboss:
    image: d3sarrollo/jboss-wildfly-amap:1.1
    container_name: jboss_amap
    volumes:
      - jboss_deployments:/opt/jboss/wildfly/standalone/deployments
    ports:
      - "8080:8080"

volumes:
  jboss_deployments:
```

3.7. Jenkins

Jenkins es un servidor de automatización open source que ayuda en la automatización de parte del proceso de desarrollo de software mediante integración continua y facilita ciertos aspectos de la entrega continua [25].

Será el elemento sobre el que pivote los servicios que hemos visto hasta ahora. El elemento integrador, generando tareas automáticas que haga uso de cada uno de ellos, según los pasos que necesitemos realizar y el resultado que queramos obtener al final de un proceso.

Requisitos

Dado el enfoque que se ha planteado en cuanto el consumo de algunos de los servicios del entorno, como por ejemplo Maven, es necesario tener acceso a la ejecución de la API de Docker por parte de Jenkins.

Evidentemente, es necesario que sea capaz de comunicarse o interaccionar con los servicios del entorno de IC.

Dockerización

Jenkins tiene una gran característica y es la extensión de sus funcionalidades mediante plugins. En este punto de los requerimientos, tenemos cubierta todas las necesidades mediante la instalación de plugins durante el proceso de instalación o una vez instalado el servidor.
Si miramos ya la comunicación de Jenkins con una máquina anfitriona de Docker es donde se hace necesario generar una imagen personalizada. Partiendo de la imagen oficial de Jenkins de Docker Hub, añadimos la instalación del CLI de Docker y generamos el mapa necesario de usuario y grupos para que pueda ser ejecutado el demonio de Docker desde un contenedor de esta imagen.

Dockerfile

```
FROM jenkins/jenkins:lts
USER root
RUN apt-get update -y && \
apt-get update && \
apt-get install -y apt-transport-https ca-certificates curl gnupg lsb-release && \
curl -fsSL https://download.docker.com/linux/debian/gpg | gpg --dearmor -o
/usr/share/keyrings/docker-archive-keyring.gpg && \
echo   "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/debian $(lsb_release -cs) stable" |  tee
/etc/apt/sources.list.d/docker.list > /dev/null && \
apt-get update -y && \
apt-get install -y docker-ce-cli && \
groupadd -g 982 docker && \
usermod -aG docker jenkins
USER Jenkins
```

Una vez obtenida la imagen con estas características, tan sólo tenemos que configurar el servicio dotándole de persistencia mediante un volumen, exponiendo su puerto web y montando el socket de Docker de la máquina anfitriona con el socket de la imagen de Jenkins

docker.compose.yml

```
version: "3"

services:

  jenkins:
    image: d3sarrollo/jenkins-docker-cli:v1.0
    volumes:
      - jenkins_data:/var/jenkins_home
      #Montamos Docker Socket para tener acceso desde el contenedor al Docker Daemon
del host
      - /var/run/docker.sock:/var/run/docker.sock
    ports:
       - "8082:8080"

volumes:
  jenkins_data:
```

Con esta configuración, el contenedor tendrá acceso al demonio de Docker de la máquina anfitriona y le permitirá consumir aquellos servicios dockerizados no persistentes que se necesiten a demanda.

3.8. Nginx

Nginx es un servidor web y proxy inverso ligero de alto rendimiento, libre y de código abierto. Su implantación ha crecido año tras año, y actualmente es la opción dominante entre los sitios web más visitados de Internet, por delante ya incluso de Apache [27].

Las ventajas que ofrece Nginx sobre otras soluciones van desde el balanceo de carga a una seguridad mejorada, un sistema potente de caché y compresión, cifrado optimizado, etc. [11]

Requisitos

Exponer los servicios y/o herramientas con acceso web para los potenciales usuarios del entorno de IC. Cada servicio expuesto accesible bajo un subdominio y segurizar la conexión entre cliente y servicios.

Dockerización

En este caso, el uso de Nginx va dirigido a servir de fachada y punto de entrada a los servicios web de entorno de IC, es decir, como proxy inverso. De manera que todas las peticiones de clientes web a servicios del entorno pasarán por él, enrutando adecuadamente cada petición al servicio que corresponda y permitiendo segurizar la comunicación de una manera sencilla.

Como primer paso, configuraremos el servicio usando como base la imagen del registro público de Docker para Nginx.

docker-compose.yml

```
version: "3"

services:

  nginx:
    image: nginx:alpine
    volumes:
     - /srv/nginx/templates:/etc/nginx/templates
    ports:
     - "80:80"
     - "443:443"
```

La configuración se resume en exponer el puerto web seguro y no seguro (443 y 80, respectivamente) y montar un directorio para la configuración “en caliente” de Nginx. La razón de exponer ambos puertos es simplemente ante la previsión de que servicios no segurizados del entorno puedan segurizarse fácilmente.

Tanto la propia segurización como la resolución de las peticiones de clientes web a los servicios web basadas en subdominio se verá en el siguiente punto.

4. Integración de servicios

En este apartado, veremos no sólo la integración de los diferentes servicios como aplicación multicontenedor sino la resolución de los dos ejemplos de procesos de IC propuestos al inicio del capítulo 3. En ningún momento, se va a entrar en detalle en la instalación, configuración u operativa básica de gestión de cada herramienta ya que queda fuera del alcance del trabajo, pero si se mencionarán algunas de las acciones realizadas con ellas dentro del proceso de integración.

4.1. Stack de servicios dockerizado

Como hemos visto en anteriores apartados, hemos ido "dockerizando" y configurando los servicios y herramientas que vamos a necesitar en el entorno. El siguiente paso será unificar esos servicios de manera que con Docker Compose podamos gestionar global o individualmente cada uno de ellos, si fuese necesario. La ventaja que nos da Docker Compose es que simplemente fusionando los archivos de configuración en un solo fichero podemos, de manera rápida y sencilla, obtener un *stack* de aplicaciones que pueden ser arrancadas con un solo comando. Y no sólo eso, además Docker Compose al ejecutar dicho fichero no sólo iniciará la ejecución de un contenedor de cada uno de los servicios declarados y montará las rutas del anfitrión y volúmenes en los servicios correspondientes, sino que generará además una red que los servicios compartirán permitiendo que puedan comunicarse entre ellos.

Así, recuperamos los ficheros de configuración docker-compose.yml de los diferentes servicios que vimos en el capítulo previo y fusionamos adecuadamente para cumplir con la sintaxis de Docker Compose, declarando un único fichero de configuración con todos los servicios (y sus dependencias) y los volúmenes para la persistencia de datos que se van utilizar.

docker-compose.yml

```
version: "3"

services:

  jenkins:
    image: d3sarrollo/jenkins-docker-cli:v1.0
    volumes:
      - jenkins_data:/var/jenkins_home
      #Montamos Docker Socket para tener acceso desde el contenedor al Docker Daemon
del host
      - /var/run/docker.sock:/var/run/docker.sock
    ports:
       - "8082:8080"

  gitlab:
    image: 'gitlab/gitlab-ce:latest'
    restart: always
    ports:
      - "8083:80"
    volumes:
```

```
      - '/srv/gitlab/config:/etc/gitlab'
      - '/srv/gitlab/logs:/var/log/gitlab'
      - '/srv/gitlab/data:/var/opt/gitlab'

  nexus:
    image: 'sonatype/nexus3:latest'
    restart: always
    ports:
       - "8081:8081"
    volumes:
      - 'nexus-data:/nexus-data'

  sonarqube:
    image: sonarqube:8-community
    depends_on:
      - db_sonar
    environment:
      SONAR_JDBC_URL: jdbc:postgresql://db_sonar:5432/sonar
      SONAR_JDBC_USERNAME: sonar
      SONAR_JDBC_PASSWORD: sonar
    volumes:
      - sonarqube_data:/opt/sonarqube/data
      - sonarqube_extensions:/opt/sonarqube/extensions
      - sonarqube_logs:/opt/sonarqube/logs
    ports:
      - "9000:9000"

  db_sonar:
    image: postgres:12
    environment:
      POSTGRES_USER: sonar
      POSTGRES_PASSWORD: sonar
    volumes:
      - postgresql:/var/lib/postgresql
      - postgresql_data:/var/lib/postgresql/data

  nginx:
    image: nginx:alpine
    volumes:
     - /srv/nginx/templates:/etc/nginx/templates
     - /srv/nginx/certs:/etc/nginx/certs
    ports:
     - "80:80"
     - "443:443"

  jboss:
    image: d3sarrollo/jboss-wildfly-amap:1.1
    container_name: jboss_amap
    volumes:
      - jboss_deployments:/opt/jboss/wildfly/standalone/deployments
    ports:
      - "8080:8080"
    command: /opt/jboss/wildfly/bin/standalone.sh -b 0.0.0.0 -bmanagement 0.0.0.0

volumes:
  nexus-data:
  jenkins_data:
  sonarqube_data:
  sonarqube_extensions:
  sonarqube_logs:
  postgresql:
  postgresql_data:
  jboss_deployments:
```

Las referencias a las imágenes de cada uno de los servicios se corresponden a aquellas que hemos generado o hemos localizado en el registro Docker.

4.2. Proxy inverso

Ya vimos en el apartado dedicado a Nginx, cual era la misión de este como fachada ante las peticiones sobre los servicios del entorno de IC. En ese aspecto, quedó pendiente comentar cómo se llevaría a cabo el enrutado de peticiones de clientes web y la segurización que ahora trataremos.

El objetivo es asignar a cada servicio un subdominio sobre el que responder ante peticiones de clientes web. No entraremos en la configuración o mecanismos para la asignación de dominios y resolución DNS, partiendo de una situación en que está resuelto, donde a la máquina anfitriona del entorno se le ha asignado un dominio *example.com* y la resolución de los subdominios *.*example.com,* que utilizaremos para acceder a cada servicio web expuesto.

Con esa asignación ya realizada es cuando el proxy inverso entra en escena. El primer paso que realizamos es el redireccionamiento de todo el tráfico no seguro a protocolo seguro

default.conf.template

```
server {
        listen 80 default_server;
        server_name _;

        return 301 https://$host$request_uri;
}
```

Si recordamos (del apartado 3.8 Nginx), el servicio se había configurado exponiendo los puertos web 80 y 443 previendo casos de servicios no segurizados y poder de esta manera solucionarlo. Con este redireccionamiento del tráfico damos el primer paso para ello. El planteamiento es que el entorno de manera externa sea como una *caja negra* cuyo punto de entrada sea Nginx y cualquier comunicación entre un cliente y un servicio web del entorno se realice a través de él. Esto hace que no sea necesario segurizar la comunicación entre los servicios del entorno, mejorando la respuesta entre ellos ya que se elimina el coste de recursos que supone la encriptación de comunicaciones.

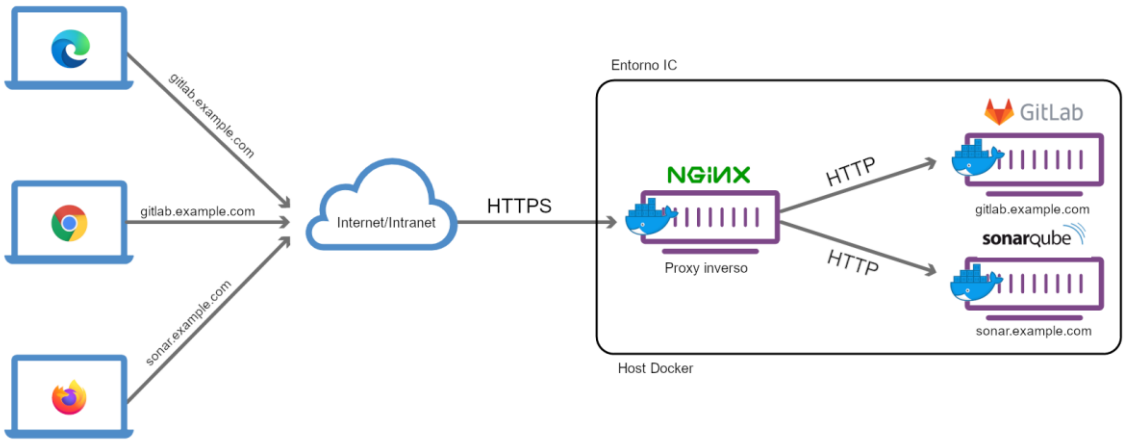

Ilustración 13 *Esquema comunicación entorno IC con proxy inverso y clientes web*

Para configurar la capa de seguridad necesitaremos un certificado de clave pública y una clave privada (que instalaremos en nuestro proxy inverso), y configuraremos para cada servicio el enrutado de peticiones basado en el subdominio que les hemos asignado. Sirva de ejemplo, el servidor de aplicaciones JBoss que será consumido desde el subdominio *deploy.example.com*

```
default.conf.template

upstream deploy{
        server jboss:8080;
        keepalive 10;
}

server {
        listen 443 ssl http2;
        server_name         deploy.example.com;
        ssl_certificate     /etc/nginx/example.com.crt;
        ssl_certificate_key /etc/nginx/example.com.key;
        ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
        ssl_ciphers 'ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-
ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-
SHA384:ECDHE-RSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-
SHA384:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-
SHA:ECDHE-RSA-AES256-SHA384:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA384:ECDHE-
ECDSA-AES256-SHA:ECDHE-RSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-
RSA-AES256-SHA256:DHE-RSA-AES256-SHA:ECDHE-ECDSA-DES-CBC3-SHA:ECDHE-RSA-DES-CBC3-
SHA:EDH-RSA-DES-CBC3-SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:AES128-SHA256:AES256-
SHA256:AES128-SHA:AES256-SHA:DES-CBC3-SHA:!DSS';
        ssl_prefer_server_ciphers on;
        ssl_dhparam /etc/nginx/dhparams.pem;

        proxy_http_version 1.1;
        proxy_set_header Connection "";
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

        location / {
                proxy_pass http://deploy;
                proxy_redirect http://deploy.example.com https://deploy.example.com;
        }
}
```

Por un lado, estamos configurando tanto las capas SSL/TLS para saber dónde localizar el certificado de clave pública y la clave privada, como los protocolos y cifrados a utilizar. Y por otro, estamos estableciendo el protocolo y el servicio que va a responder a la petición que está manejando el proxy para el host indicado en las cabeceras de esta (hay que recordar que Docker Compose genera una red que permite comunicarse a los servicios y pueden simplemente descubrirse por su nombre).

De los servicios que componen el entorno se establecieron los siguientes para su acceso web y bajo los siguientes subdominios:

Servicio	Subdominio
Gitlab	gitlab.example.com
Nexus OSS	nexus.example.com
Jenkins	jenkins.example.com
SonarQube	sonarqube.example.com
JBoss	deploy.example.com

La configuración en el proxy inverso para cada uno de ellos sería análoga a la vista para el caso dado como ejemplo, redirigiendo cada petición al servicio correspondiente (ver Anexo 1).

4.3. Tareas Jenkins

Jenkins nos permite de manera sencilla organizar y crear tareas simples y complejas de manera automatizada. Esto lo hace a través de una serie de características básicas que pueden extenderse, como ya dijimos, mediante plugins. La manera en que Jenkins nos permite definir cada uno de esos procesos o tareas automatizadas es lo que denomina Pipeline. Un Pipeline no es más que la definición de un proceso completo mediante una sintaxis propia (siguiendo las reglas del lenguaje Groovy) que puede ser de tipo declarativa o programada ("scripted"). Para ello, y para facilitar su creación, Jenkins proporciona herramientas para acceder de manera sencilla a las funcionalidades básicas y aquellas extendidas mediante plugins, que permitan describir las acciones a realizar en el Pipeline.

El modo en que se organiza un Pipeline es a través de diferentes fases (stages) y los pasos (steps) que componen cada una de ellas. Mediante estos elementos modelamos nuestros flujos de trabajo para llevar a cabo las tareas a completar. Veamos esto, aplicado a los dos casos de uso que se propusieron como objetivo dentro del trabajo.

- *Generación, centralización y almacenamiento de un paquete o dependencia software.*

Partiendo de un código ejemplo o *"dummy"* de un artefacto Java desarrollado en Spring Boot se quiere obtener el compilado y empaquetado de este, la ejecución de test unitarios, análisis de código estático y publicación en repositorio central de artefactos.

Teniendo en cuenta todas las tareas a realizar es fácil definir 4 fases en el proceso:

1. Obtener fuentes actualizadas del código.

```
stage('Gitlab checkout') {
   steps {
            checkout([$class: 'GitSCM', branches: [[name: '*/master']],
   extensions: [], userRemoteConfigs: [[ credentialsId: 'ssh-jenkins-gitlab-
   key',url: 'git@gitlab:jenkins-group/hello-world.git']]])
        }
  }
```

El primer paso será comprobar cambios y descargar fuentes desde el repositorio, que habremos configurado en nuestro servicio Gitlab con el código fuente del artefacto. En este caso, lo que se hace es comprobar cambios en la rama *master* del repositorio, sobre el que hemos dado permisos de acceso a un usuario para Jenkins. Esas credenciales pueden almacenarse en Jenkins y ser referenciadas (credentialsId) en el Pipeline.

2. Compilación, empaquetado y ejecución de tests.

```
stage('Docker Maven build') {
        steps{
                  sh 'docker run --rm --volumes-from ic_jenkins_1 -w ${PWD}
   d3sarrollo/maven-3.8.1-open-jdk-8 clean package'
                   archiveArtifacts artifacts: 'target/*.jar', followSymlinks:
   false, onlyIfSuccessful: true
        }
  }
```

En esta fase, diferenciamos dos pasos. Por un lado, la generación de compilado, empaquetado y ejecución de tests, y por otro, el guardado local del artefacto (en ocasiones es interesante tener copias de seguridad de los compilados de rápido acceso). Destacamos el primer paso haciendo uso del servicio Maven a través del consumo de un contenedor Docker partiendo de la imagen que generamos para Java 8 y montando los volúmenes del servicio Jenkins de manera que el directorio local, en donde están los fuentes resultado de la fase previa, estén disponibles para Maven. Mediante los parámetros *"clean package",* que pasamos a Docker para el punto de entrada del contenedor, nos aseguramos de limpiar los ficheros generados en anteriores construcciones y se lance el compilado, empaquetado y ejecución de tests.

3. Análisis de código.

```
stage('Docker Sonarqube analysis') {
        steps{
                  sh 'docker run --network ic_default --rm --volumes-from
   ic_jenkins_1 -w ${PWD} -e SONAR_HOST_URL="http://sonarqube:9000" -e
   SONAR_LOGIN="e5503766c1d1282e83b25c2afae003fc944d1047" sonarsource/sonar-
   scanner-cli -Dsonar.projectKey=IC_CD_artifact_demo -Dsonar.sources=src -
   Dsonar.java.binaries=target'
      }
  }
```

Con el servicio web SonarQube activo, esta fase consistirá en ejecutar el escáner de análisis sobre los fuentes para que luego vuelque los resultados en el servidor del servicio. Como ya comentamos en la sección dedicada a SonarQube, tenemos

disponible una imagen en Docker Hub para lanzar este paso, por lo que de manera similar al caso de la fase anterior, consumimos un contenedor que ejecute la tarea. Habremos generado previamente un usuario en SonarQube para el servicio Jenkins y generado unas credenciales (en este caso en forma de token) que junto con el *host* de SonarQube, el nombre del proyecto, el perfil de análisis y el resto de parámetros que necesitemos se utilizarán en la ejecución del contenedor.

4. Publicación artefacto.

```
stage('Nexus publishing') {
      steps{
          nexusArtifactUploader artifacts: [[artifactId: 'demo', classifier: '',
   file: 'target/demo-0.0.1-SNAPSHOT.jar', type: 'jar']], credentialsId:
   'jenkins-nexus', groupId: 'com.example', nexusUrl: 'nexus:8081', nexusVersion:
   'nexus3', protocol: 'http', repository: 'my-repo/', version: '1.0.0'
      }
  }
```

Aquí, en un proceso inverso al inicial de obtención de fuentes, publicamos en el servicio Nexus OSS, en un repositorio que habremos creado previamente en el servicio, el artefacto obtenido durante las fases previas. Como en el caso de Gitlab, crearemos un usuario y generaremos unas credenciales de acceso para Jenkins en el repositorio que proporcionaremos junto con todos los datos necesarios para que se ejecute la publicación.

Aunque el Pipeline podría quedar finalizado así, podemos dar valor añadido al mismo con un par de detalles extras:

- Notificación resultado ejecución Pipeline.

```
post {
    always {
        emailext attachLog: true, to: 'd3sarrollo@gmail.com', body:
   "${currentBuild.currentResult}: Job ${env.JOB_NAME} build
   ${env.BUILD_NUMBER}\n More info at: ${env.BUILD_URL}", subject: "Jenkins Build
   ${currentBuild.currentResult}: Job ${env.JOB_NAME}"
    }
}
```

La sintaxis de Jenkins para los Pipeline permite crear una sección para añadir pasos adicionales una vez que una fase del Pipeline o todas se han ejecutado. En este caso, lo utilizamos para añadir una notificación correo electrónico con información de ejecución del Pipeline.

- Ejecución automática por cambios en el repositorio.

Se trata de una mejora que realmente automatiza todo el proceso, ya que permite que ante cambios en una rama seleccionada se lance en respuesta la ejecución del Pipeline. Es decir, permite crear flujos en el desarrollo en el que una vez el desarrollador deje su código en el SCV, todo el proceso fijado en la tarea de Jenkins

se ejecute por sí solo sin interacción de este. Evidentemente, la estrategia que se lleva a cabo a nivel de repositorio de código suele implicar revisión de los cambios propuestos por el desarrollador, pero eso depende ya de la organización de los equipos y las buenas prácticas que se lleven a cabo.

Para ello, Gitlab tiene dentro de la configuración de cada repositorio la posibilidad de activar un mecanismo, denominado *Webhook,* por el que puede informar a Jenkins de que quiere lanzar la ejecución de un Pipeline ante cambios en el repositorio. Evidentemente, Jenkins debe permitirlo, para lo cual a través de la configuración del Pipeline podemos indicar ante qué cambios del repositorio se va a responder y generar un token que de acceso al Webhook para lanzar la ejecución.

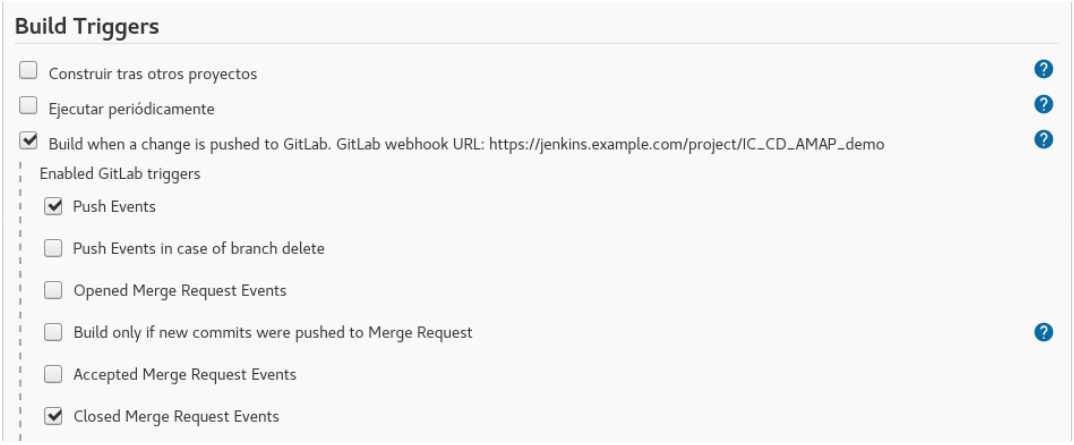

Ilustración 14 *Ejemplo configuración disparadores de ejecución Pipeline Jenkins*

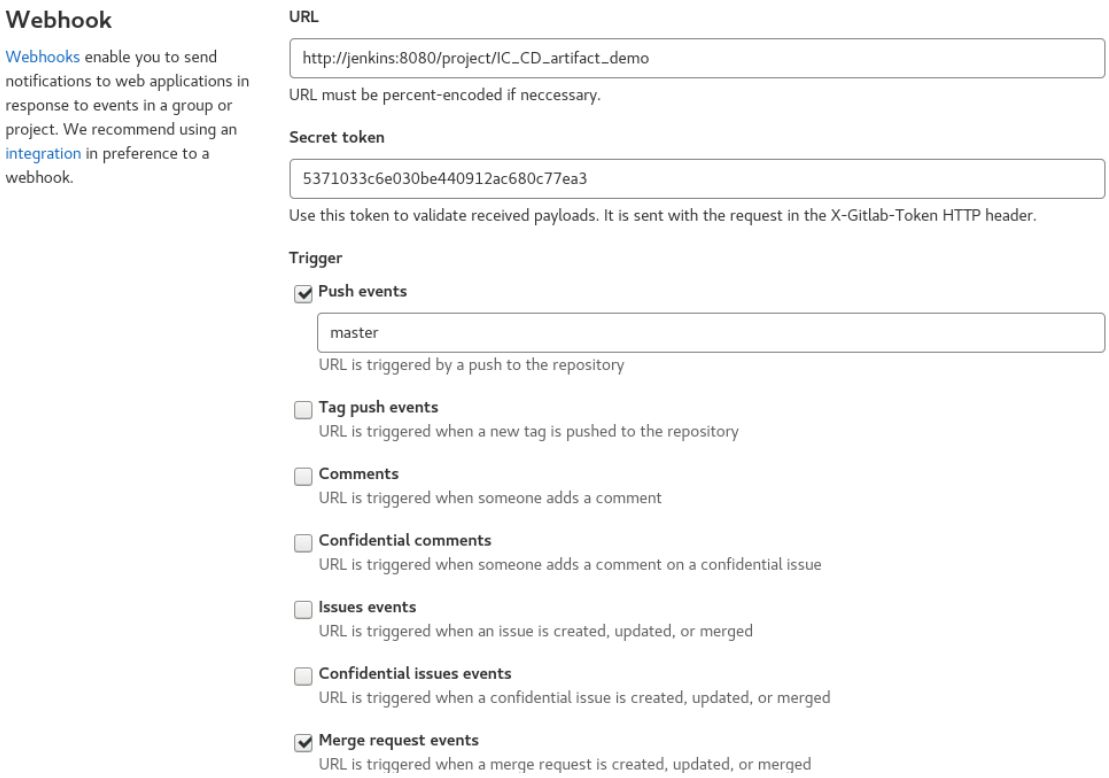

Ilustración 15 *Ejemplo configuración Gitlab para Webhook en un repositorio*

La configuración del Pipeline completa puede consultarse en el Anexo 2. Veamos un ejemplo de ejecución del Pipeline, con algunos detalles de su salida:

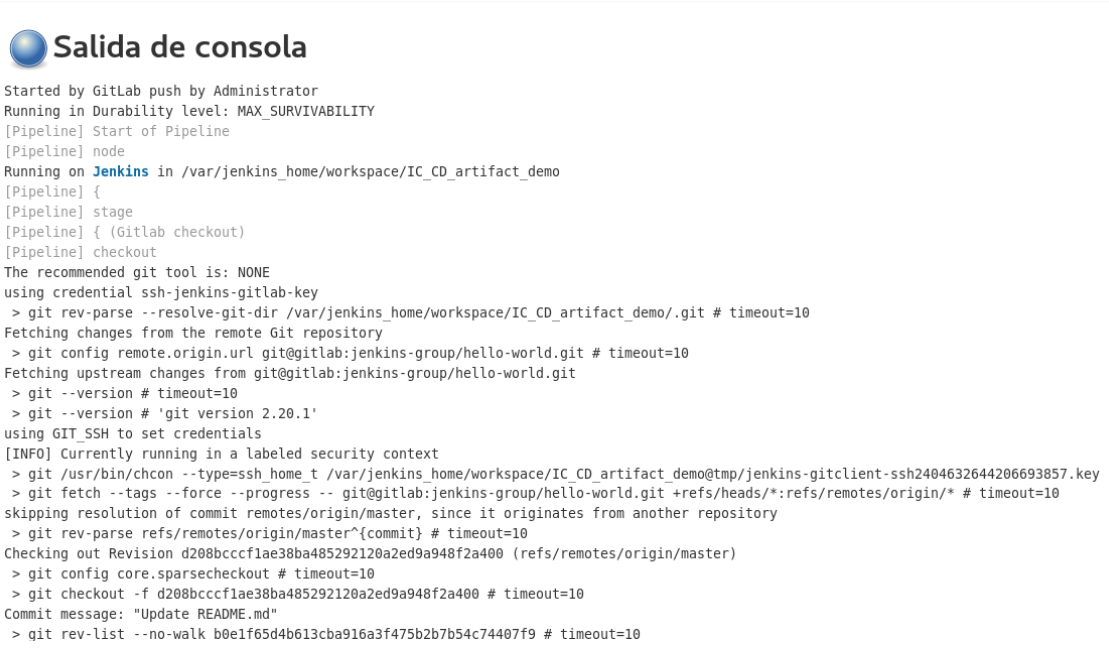

Ilustración 16 *Salida fase recuperación código repositorio Git*

```
[Pipeline] stage
[Pipeline] { (Docker Maven build)
[Pipeline] sh
+ docker run --rm --volumes-from ic_jenkins_1 -w /var/jenkins_home/workspace/IC_CD_artifact_demo d3sarrollo/maven-3.8.1-open-jdk-8 clean package
[INFO] Scanning for projects...
Downloading from central: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-starter-parent/2.0.1.RELEASE/spring-boot-starter-parent-2.0.1.RELEASE.pom
Progress (1): 2.7/12 kB
Progress (1): 5.5/12 kB
Progress (1): 8.2/12 kB
Progress (1): 11/12 kB
Progress (1): 12 kB

Downloaded from central: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-starter-parent/2.0.1.RELEASE/spring-boot-starter-parent-2.0.1.RELEASE.pom (12 kB at 6.9 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-dependencies/2.0.1.RELEASE/spring-boot-dependencies-2.0.1.RELEASE.pom
Progress (1): 2.7/133 kB
```

Ilustración 17 *Salida fase construcción Maven inicio (descarga de dependencias)*

```
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire-junit4/2.20/surefire-junit4-2.20.jar (82 kB at 601 kB/s)
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running com.example.demo.DemoApplicationTests
11:28:20.970 [main] DEBUG org.springframework.test.context.junit4.SpringJUnit4ClassRunner - SpringJUnit4ClassRunner constructor called with [class com.example.demo.DemoApplicationTests]
11:28:21.061 [main] DEBUG org.springframework.test.context.BootstrapUtils - Instantiating CacheAwareContextLoaderDelegate from class [org.springframework.test.context.cache.DefaultCacheAwareContextLoaderDelegate]
11:28:21.141 [main] DEBUG org.springframework.test.context.BootstrapUtils - Instantiating BootstrapContext using constructor [public org.springframework.test.context.support.DefaultBootstrapContext(java.lang.Class,org.springframework.test.context.CacheAwareContextLoaderDelegate)]
11:28:21.318 [main] DEBUG org.springframework.test.context.BootstrapUtils - Instantiating TestContextBootstrapper for test class [com.example.demo.DemoApplicationTests] from class [org.springframework.boot.test.context.SpringBootTestContextBootstrapper]
```

Ilustración 18 *Salida fase construcción ejecución tests*

```
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 32.251 s - in com.example.demo.DemoApplicationTests
2021-09-04 11:28:50.957  INFO 63 --- [       Thread-3] ConfigServletWebServerApplicationContext : Closing org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@2e385cce: startup date [Sat Sep 04 11:28:28 GMT 2021]; root of context hierarchy
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- maven-jar-plugin:3.0.2:jar (default-jar) @ demo ---
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-archiver/3.1.1/maven-archiver-3.1.1.pom
```

Ilustración 19 *Salida fase construcción resultado tests*

```
Downloaded from central: https://repo.maven.apache.org/maven2/com/google/guava/guava/11.0.2/guava-11.0.2.jar (1.6 MB at 403 kB/s)
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:40 min
[INFO] Finished at: 2021-09-04T11:29:08Z
[INFO] ------------------------------------------------------------------------
[Pipeline] archiveArtifacts
Archiving artifacts
[Pipeline] }
[Pipeline] // stage
```

Ilustración 20 *Salida fase construcción resultado construcción Maven y archivado de empaquetado*

```
[Pipeline] stage
[Pipeline] { (Docker Sonarqube analysis)
[Pipeline] sh
+ docker run --network ic_default --rm --volumes-from ic_jenkins_1 -w /var/jenkins_home/workspace/IC_CD_artifact_demo -e SONAR_HOST_URL=http://sonarqube:9000 -e
SONAR_LOGIN=e5503766c1d1282e83b25c2afae003fc944d1047 sonarsource/sonar-scanner-cli -Dsonar.projectKey=IC_CD_artifact_demo -Dsonar.sources=src
-Dsonar.java.binaries=target
INFO: Scanner configuration file: /opt/sonar-scanner/conf/sonar-scanner.properties
INFO: Project root configuration file: NONE
INFO: SonarScanner 4.6.2.2472
INFO: Java 11.0.11 AdoptOpenJDK (64-bit)
INFO: Linux 3.10.0-1160.25.1.el7.x86_64 amd64
INFO: User cache: /opt/sonar-scanner/.sonar/cache
INFO: Scanner configuration file: /opt/sonar-scanner/conf/sonar-scanner.properties
INFO: Project root configuration file: NONE
INFO: Analyzing on SonarQube server 8.8.0
INFO: Default locale: "en_US", source code encoding: "UTF-8" (analysis is platform dependent)
INFO: Load global settings
INFO: Load global settings (done) | time=489ms
INFO: Server id: D1BE41C1-AXnNVrHF6d_U6RuaMIn8
INFO: User cache: /opt/sonar-scanner/.sonar/cache
```

Ilustración 21 *Salida fase análisis de código estático inicio*

```
INFO: Analysis report generated in 262ms, dir size=95 KB
INFO: Analysis report compressed in 31ms, zip size=15 KB
INFO: Analysis report uploaded in 270ms
INFO: ANALYSIS SUCCESSFUL, you can browse http://sonarqube:9000/dashboard?id=IC_CD_artifact_demo
INFO: Note that you will be able to access the updated dashboard once the server has processed the submitted analysis report
INFO: More about the report processing at http://sonarqube:9000/api/ce/task?id=AXuwkmMHB6toeQfSw-Uz
INFO: Analysis total time: 20.381 s
INFO: ------------------------------------------------------------------------
INFO: EXECUTION SUCCESS
INFO: ------------------------------------------------------------------------
INFO: Total time: 32.685s
INFO: Final Memory: 9M/119M
INFO: ------------------------------------------------------------------------
[Pipeline] }
[Pipeline] // stage
```

Ilustración 22 *Salida fase análisis de código estático resultado*

```
[Pipeline] stage
[Pipeline] { (Nexus publishing)
[Pipeline] nexusArtifactUploader
Uploading artifact demo-0.0.1-SNAPSHOT.jar started....
GroupId: com.example
ArtifactId: com.example
Classifier:
Type: jar
Version: 1.0.0
File: demo-0.0.1-SNAPSHOT.jar
Repository:my-repo/
Uploading: http://nexus:8081/repository/my-repo/com/example/demo/1.0.0/demo-1.0.0.jar
10 % completed (1.6 MB / 16 MB).
20 % completed (3.2 MB / 16 MB).
30 % completed (4.8 MB / 16 MB).
40 % completed (6.4 MB / 16 MB).
50 % completed (8.1 MB / 16 MB).
60 % completed (9.7 MB / 16 MB).
70 % completed (11 MB / 16 MB).
80 % completed (13 MB / 16 MB).
90 % completed (14 MB / 16 MB).
100 % completed (16 MB / 16 MB).
Uploaded: http://nexus:8081/repository/my-repo/com/example/demo/1.0.0/demo-1.0.0.jar (16 MB at 4.8 MB/s)
Uploading artifact demo-0.0.1-SNAPSHOT.jar completed.
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Declarative: Post Actions)
[Pipeline] emailext
Sending email to: d3sarrollo@gmail.com
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Ilustración 23 *Salida fase publicación artefacto y notificación de correo electrónico ejecución Pipeline.*

- *Generación, centralización, almacenamiento y despliegue de una aplicación.*

Para ejemplificar este caso, partimos de un arquetipo de aplicación con una serie de dependencias propias (repositorio de dependencias externo) y una configuración a nivel de servidor de aplicaciones que ya vimos en la sección dedicada a JBoss. Se trata de una aplicación Java desarrollada en Spring MVC de la que se quiere obtener el compilado y empaquetado, la ejecución de tests, análisis de código estático y despliegue en un servidor de aplicaciones.

Teniendo en cuenta todas las tareas a realizar se definen 4 fases en el proceso:

1. Obtener fuentes actualizadas del código.

```
steps {
            checkout([$class: 'GitSCM', branches: [[name: '*/master']],
   extensions: [], userRemoteConfigs: [[ credentialsId: 'ssh-jenkins-gitlab-
   key',url: 'git@gitlab:jenkins-group/demo-amap.git']]])
        }
   }
```

Igual a la misma fase ya vista en el caso del Pipeline anterior, particularizando para el repositorio del proyecto con el que trabajamos.

2. Compilación, empaquetado y ejecución de tests.

```
stage('Docker Maven build and test') {
        steps{
                    configFileProvider([configFile(fileId: 'a952bafc-5676-4a87-
   8161-f3cf19434538', targetLocation: './settings-amap.xml')]) {
                        sh 'docker run --rm --volumes-from ic_jenkins_1 -w ${PWD}
   d3sarrollo/maven-3.8.1-open-jdk-8 clean package -s settings-amap.xml'
                        archiveArtifacts artifacts: 'target/*.war',
   followSymlinks: false, onlyIfSuccessful: true
            }

        }
    }
```

Fase exactamente igual a la vista en el Pipeline anterior, pero al que se le ha añadido un paso más para poder incorporar en la ejecución de Maven los repositorios particulares en donde se encuentran las dependencias de la aplicación y que añadiremos como parámetro de entrada adicional.

Para ello, Maven permite mediante un fichero de configuración xml (por defecto, settings.xml) configurar los repositorios que se van a utilizar para resolver las dependencias en una construcción. Usando Jenkins podemos crear ficheros a los que podemos referenciar en nuestros Pipeline para ser utilizados durante su ejecución.

```
configFileProvider([configFile(fileId: 'a952bafc-5676-4a87-8161-f3cf19434538',
targetLocation: './settings-amap.xml')]
```

Con esto hacemos referencia al fichero ***fielId,*** que habremos creado previamente, y copiamos en el directorio de trabajo de nuestra tarea (settings-amap.xml). De esta manera, lo dejamos accesible para Maven en la ejecución del contenedor y que indicamos como parámetro de entrada.

```
sh 'docker run --rm --volumes-from ic_jenkins_1 -w ${PWD} d3sarrollo/maven-3.8.1-
open-jdk-8 clean package -s settings-amap.xml'
```

3. Análisis de código.

```
stage('Docker Sonarqube analysis') {
        steps{
                    sh 'docker run --network ic_default --rm --volumes-from
   ic_jenkins_1 -w ${PWD} -e SONAR_HOST_URL="http://sonarqube:9000" -e
   SONAR_LOGIN="e5503766c1d1282e83b25c2afae003fc944d1047" sonarsource/sonar-
   scanner-cli -Dsonar.projectKey=amap_demo -Dsonar.sources=src -
   Dsonar.java.binaries=target'
        }
    }
```

Exactamente igual a la misma fase ya vista en el caso en el Pipeline del artefacto, particularizando para el proyecto con el que trabajamos.

4. Despliegue aplicación.

```
stage('Jboss war deploy') {
        steps{
            sh 'docker cp target/demo-1.0.0.war
   jboss_amap:/opt/jboss/wildfly/standalone/deployments'
        }
    }
```

Aprovechando el mecanismo por defecto de autodespliegue, mediante escaneo periódico de una ruta de JBoss Wildfly, podemos resolver esta fase simplemente copiando el empaquetado de la aplicación usando Docker hacia el volumen que tiene mapeado el contenedor del servidor de aplicaciones y que se corresponde con la ruta que utiliza para ese mecanismo. Ya ahí, el servicio se encargará de realizar las acciones necesarias para inicializar la aplicación.

Aunque el Pipeline podría quedar finalizado así, también se integran de manera análoga el par de detalles extras vistos en el caso anterior con el artefacto Java:

- Notificación resultado ejecución Pipeline.
- Ejecución automática por cambios en el repositorio.

La configuración del Pipeline completa puede consultarse en el Anexo 3. Veamos un ejemplo de ejecución del Pipeline, con algunos detalles de su salida:

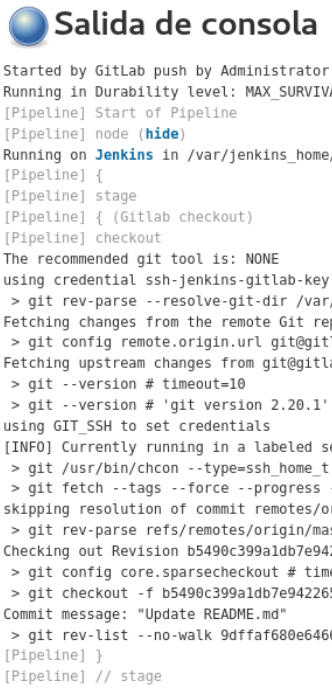

Ilustración 24 *Salida fase recuperación código repositorio Git*

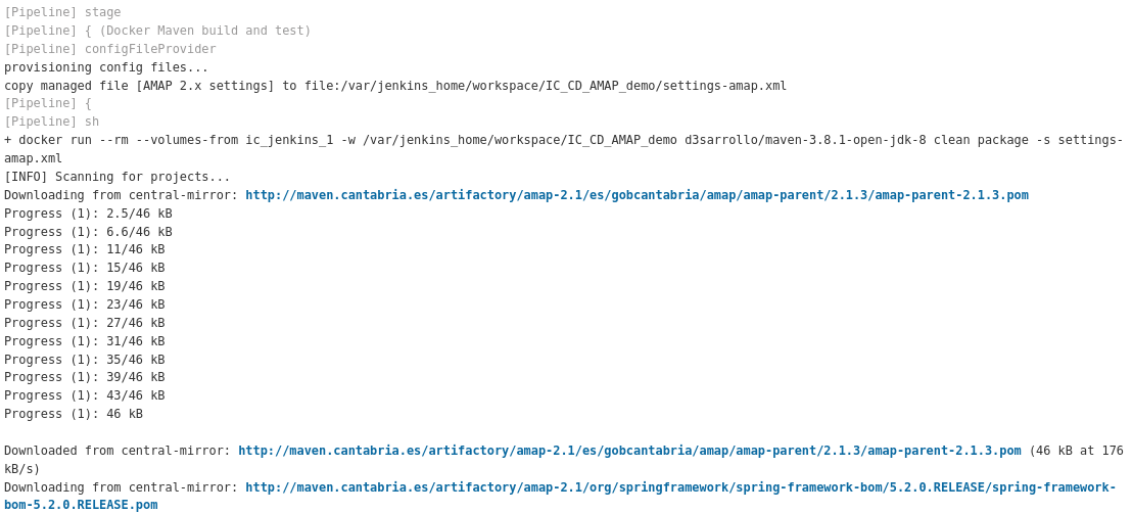

Ilustración 25 *Salida fase construcción Maven inicio (configuración repositorio dependencias y descarga)*

Ilustración 26 *Salida fase construcción ejecución tests*

```
[2021-09-04 11:58:21,932]  INFO (AbstractEntityManagerFactoryBean.java:598) - Closing JPA EntityManagerFactory for persistence unit 'default'
[2021-09-04 11:58:21,935]  INFO (EmbeddedDatabaseFactory.java:221) - Shutting down embedded database: url='jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-
1;DB_CLOSE_ON_EXIT=false'
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 19.581 s - in es.gobcantabria.aplicaciones.demo.business.service.impl.PortadaServiceTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jacoco-maven-plugin:0.7.9:report (report) @ demo ---
[INFO] Loading execution data file /var/jenkins_home/workspace/IC_CD_AMAP_demo/target/jacoco.exec
[INFO] Analyzed bundle 'Demo arquetipo AMAP' with 36 classes
```

Ilustración 27 *Salida fase construcción resultado tests*

```
Downloaded from plugins-mirror: http://maven.cantabria.es/artifactory/plugins/com/thoughtworks/xstream/xstream/1.4.10/xstream-1.4.10.jar (590 kB at 1.9 MB/s)
[INFO] Packaging webapp
[INFO] Assembling webapp [demo] in [/var/jenkins_home/workspace/IC_CD_AMAP_demo/target/demo-1.0.0]
[INFO] Processing war project
[INFO] Copying webapp resources [/var/jenkins_home/workspace/IC_CD_AMAP_demo/src/main/webapp]
[INFO] Webapp assembled in [406 msecs]
[INFO] Building war: /var/jenkins_home/workspace/IC_CD_AMAP_demo/target/demo-1.0.0.war
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:15 min
[INFO] Finished at: 2021-09-04T11:58:28Z
[INFO] ------------------------------------------------------------------------
[Pipeline] archiveArtifacts
Archiving artifacts
[Pipeline] }
[Pipeline] // configFileProvider
[Pipeline] }
```

Ilustración 28 *Salida fase construcción resultado construcción Maven y archivado de empaquetado*

```
[Pipeline] stage
[Pipeline] { (Docker Sonarqube analysis)
[Pipeline] sh
+ docker run --network ic_default --rm --volumes-from ic_jenkins_1 -w /var/jenkins_home/workspace/IC_CD_AMAP_demo -e SONAR_HOST_URL=http://sonarqube:9000 -e
SONAR_LOGIN=e5503766c1d1282e83b25c2afae003fc944d1047 sonarsource/sonar-scanner-cli -Dsonar.projectKey=amap_demo -Dsonar.sources=src -Dsonar.java.binaries=target
INFO: Scanner configuration file: /opt/sonar-scanner/conf/sonar-scanner.properties
INFO: Project root configuration file: NONE
INFO: SonarScanner 4.6.2.2472
INFO: Java 11.0.11 AdoptOpenJDK (64-bit)
INFO: Linux 3.10.0-1160.25.1.el7.x86_64 amd64
INFO: User cache: /opt/sonar-scanner/.sonar/cache
INFO: Scanner configuration file: /opt/sonar-scanner/conf/sonar-scanner.properties
INFO: Project root configuration file: NONE
INFO: Analyzing on SonarQube server 8.8.0
INFO: Default locale: "en_US", source code encoding: "UTF-8" (analysis is platform dependent)
INFO: Load global settings
INFO: Load global settings (done) | time=323ms
INFO: Server id: D1BE41C1-AXnNVrHF6d_U6RuaMIn8
INFO: User cache: /opt/sonar-scanner/.sonar/cache
```

Ilustración 29 *Salida fase análisis de código estático inicio*

```
INFO: Analysis report generated in 480ms, dir size=727 KB
INFO: Analysis report compressed in 1318ms, zip size=259 KB
INFO: Analysis report uploaded in 280ms
INFO: ANALYSIS SUCCESSFUL, you can browse http://sonarqube:9000/dashboard?id=amap_demo
INFO: Note that you will be able to access the updated dashboard once the server has processed the submitted analysis report
INFO: More about the report processing at http://sonarqube:9000/api/ce/task?id=AXuwrdyuB6toeQfSw-U0
INFO: Analysis total time: 1:21.734 s
INFO: ------------------------------------------------------------------------
INFO: EXECUTION SUCCESS
INFO: ------------------------------------------------------------------------
INFO: Total time: 1:28.713s
INFO: Final Memory: 18M/119M
INFO: ------------------------------------------------------------------------
[Pipeline] }
[Pipeline] // stage
```

Ilustración 30 *Salida fase análisis de código estático resultado*

```
[Pipeline] stage
[Pipeline] { (Jboss war deploy)
[Pipeline] sh
+ docker cp target/demo-1.0.0.war jboss_amap:/opt/jboss/wildfly/standalone/deployments
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Declarative: Post Actions)
[Pipeline] emailext
Sending email to: d3sarrollo@gmail.com
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Ilustración 31 *Salida fase despliegue aplicación y notificación de correo electrónico ejecución Pipeline*

Como se habrá podido comprobar en la definición de los diferentes Pipeline se hace referencia a llamadas que se corresponden con funcionalidades que aporta por defecto Jenkins o añadidas a través de plugins mediante su sintaxis de Pipeline. Podemos hacernos una idea de lo básico que son para la configuración echando un vistazo a los plugins instalados, ya sea por defecto de la instalación de Jenkins o por los que se hayan tenido que instalar para ampliar funcionalidades, en el Anexo 4.

5. Conclusiones

Es evidente la importancia que está teniendo la tecnología de contenedores en el desarrollo de software actual, y como las prácticas de DevOps deben ser práctica común si queremos aportar verdadero valor añadido a los procesos del ciclo de vida de una aplicación. Aunque es necesario una inversión en formación y migración de entornos, en aquellos casos en que no está aplicando, es innegable los beneficios que aportan a largo plazo.

Definida la implementación técnica veamos ahora su resultado, y en qué grado hemos dado respuesta a cada uno de los objetivos que nos marcamos al inicio, y detectar posibles mejoras y líneas futuras de investigación o desarrollo.

5.1. Resultados

Arquitectura de contenedores de software para procesos de integración continua

Hemos conseguido generar un entorno de IC basado en contenedores Docker con herramientas web que permite:

- Gestión y control de versiones de código fuente, elementos de configuración y desarrollo de software colaborativo.
- Centralización y almacenamiento de paquetes software.
- Revisión y evaluación de calidad de software.
- Orquestación y automatización de tareas para el despliegue de software.

Se ha dado acceso web y segurizada la comunicación con las herramientas para su explotación, configuración y personalización. Dicha segurización mediante un esquema de red con proxy inverso es algo que no estaba planificado inicialmente pero que se ha descubierto como una manera muy eficaz de llevarlo a cabo y dirigir la manera en que los usuarios acceden a las herramientas del entorno IC.

Generación, centralización y almacenamiento de un paquete o dependencia software.

Se ha conseguido mediante el entorno de IC configurar una tarea automatizada que permite la construcción completa, test y almacenamiento de un artefacto, proporcionando información de calidad del código y resultado del proceso completo, que puede invocarse de manera manual mediante un servidor de integración (Jenkins) o de manera automática ante cambios en el repositorio de código.

Generación, centralización, almacenamiento y despliegue de una aplicación web.

Se ha conseguido mediante el entorno de IC configurar una tarea automatizada que permite la construcción completa, test y despliegue de una aplicación, proporcionando información de calidad del código y resultado del proceso completo, que puede invocarse de manera manual mediante un servidor de integración (Jenkins) o de manera automática ante cambios en el repositorio de código.

Con todo esto podemos decir que hemos cumplido de manera general con los objetivos que se habían planteado al inicio del trabajo. Hemos podido comprobar como los entornos de IC permiten aislar completamente a los desarrolladores de software de los procesos más tediosos y que pueden provocar más problemas en el ciclo de vida de aplicaciones cuando los equipos y las aplicaciones crecen en tamaño. Y hemos podido evaluar el potencial de la tecnología de contenedores de software como solución de arquitecturas o entornos multiservicio facilitando la integración, administración e instalación.

5.2. Líneas futuras

A pesar de cumplir los objetivos principales marcados, hemos detectado posibles mejoras en los procesos o nuevas vías a investigar e incorporar. A nivel de contenedorización:

- Optimización de imágenes Docker.
- Configuración de servicios para carga de secretos externa.
- Configuración de cuotas de recursos dedicados a cada servicio.
- Seguridad de los contenedores.
- Orquestación de contenedores y escalado de servicios.
- Registro privado de imágenes Docker.
- Dockerización de entornos locales de desarrollo.

En cuanto a la Integración Continua:

- Parametrización de Pipelines.
- Mejora de logs de ejecución de algunas fases.
- Generación de flujos alternativos de ejecución en casos de error durante una fase del Pipeline.
- Migración de Pipeline a modelo “In SCM” mediante JenkinsFile y repositorio de control de versiones.
- Pipelines multirama.
- Mejora de las fases que consumen contenedores a demanda (Docker plugins).
- Viabilidad de utilizar Gitlab como servidor de integración.
- Configuración de flujos de trabajo dentro de cada una de las herramientas web del entorno.

La implementación de buenas prácticas, aunque no eran protagonista de este trabajo, podrían ser perfectamente objeto de uno aparte. Las posibilidades de integración y

aplicación de estas, tienen la suficiente complejidad e independencia de la arquitectura como para tener entidad propia.

6. Anexos

6.1. Anexo 1

#Configuracion completa Nginx proxy inverso

```
upstream gitlab{
        server gitlab:80;
        keepalive 10;
}

server {
        listen 443 ssl http2;
        server_name         gitlab.example.com;
        ssl_certificate     /etc/nginx/example.com.crt;
        ssl_certificate_key /etc/nginx/example.com.key;
        ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
        ssl_ciphers 'ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-
ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-
SHA384:ECDHE-RSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-
SHA384:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-
SHA:ECDHE-RSA-AES256-SHA384:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA384:ECDHE-
ECDSA-AES256-SHA:ECDHE-RSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-
RSA-AES256-SHA256:DHE-RSA-AES256-SHA:ECDHE-ECDSA-DES-CBC3-SHA:ECDHE-RSA-DES-CBC3-
SHA:EDH-RSA-DES-CBC3-SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:AES128-SHA256:AES256-
SHA256:AES128-SHA:AES256-SHA:DES-CBC3-SHA:!DSS';
        ssl_prefer_server_ciphers on;
        ssl_dhparam /etc/nginx/dhparams.pem;

        proxy_http_version 1.1;
        proxy_set_header Connection "";
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;


        location / {
                proxy_pass http://gitlab;
                proxy_redirect http://gitlab.example.com https://gitlab.example.com;
        }
}


upstream jenkins{
        server jenkins:8080;
        keepalive 10;
}

server {
        listen 443 ssl http2;
        server_name         jenkins.example.com;
        ssl_certificate     /etc/nginx/example.com.crt;
        ssl_certificate_key /etc/nginx/example.com.key;
        ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
        ssl_ciphers 'ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-
ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-
SHA384:ECDHE-RSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-
SHA384:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-
SHA:ECDHE-RSA-AES256-SHA384:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA384:ECDHE-
ECDSA-AES256-SHA:ECDHE-RSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-
RSA-AES256-SHA256:DHE-RSA-AES256-SHA:ECDHE-ECDSA-DES-CBC3-SHA:ECDHE-RSA-DES-CBC3-
```

```
SHA:EDH-RSA-DES-CBC3-SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:AES128-SHA256:AES256-
SHA256:AES128-SHA:AES256-SHA:DES-CBC3-SHA:!DSS';
        ssl_prefer_server_ciphers on;
        ssl_dhparam /etc/nginx/dhparams.pem;

        proxy_http_version 1.1;
        proxy_set_header Connection "";
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Forwarded-Port $server_port;

        location / {
                proxy_pass http://jenkins;
                proxy_redirect http://jenkins.example.com
https://jenkins.example.com;
        }
}

upstream nexusoss{
        server nexus:8081;
        keepalive 10;
}

server {
        listen 443 ssl http2;
        server_name         nexus.example.com;
        ssl_certificate     /etc/nginx/example.com.crt;
        ssl_certificate_key /etc/nginx/example.com.key;
        ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
        ssl_ciphers 'ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-
ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-
SHA384:ECDHE-RSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-
SHA384:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-
SHA:ECDHE-RSA-AES256-SHA384:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA384:ECDHE-
ECDSA-AES256-SHA:ECDHE-RSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-
RSA-AES256-SHA256:DHE-RSA-AES256-SHA:ECDHE-ECDSA-DES-CBC3-SHA:ECDHE-RSA-DES-CBC3-
SHA:EDH-RSA-DES-CBC3-SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:AES128-SHA256:AES256-
SHA256:AES128-SHA:AES256-SHA:DES-CBC3-SHA:!DSS';
        ssl_prefer_server_ciphers on;
        ssl_dhparam /etc/nginx/dhparams.pem;

        proxy_http_version 1.1;
        proxy_set_header Connection "";
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        add_header Content-Security-Policy "upgrade-insecure-requests";

        location / {
                proxy_pass http://nexusoss;
                proxy_redirect http://nexus.example.com|http://nexus:8081
https://nexus.example.com;
        }
}


upstream sonar{
        server sonarqube:9000;
        keepalive 10;
}

server {
        listen 443 ssl http2;
        server_name         sonarqube.example.com;
        ssl_certificate     /etc/nginx/example.com.crt;
```

```
        ssl_certificate_key /etc/nginx/example.com.key;
        ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
        ssl_ciphers 'ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-
ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-
SHA384:ECDHE-RSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-
SHA384:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-
SHA:ECDHE-RSA-AES256-SHA384:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA384:ECDHE-
ECDSA-AES256-SHA:ECDHE-RSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-
RSA-AES256-SHA256:DHE-RSA-AES256-SHA:ECDHE-ECDSA-DES-CBC3-SHA:ECDHE-RSA-DES-CBC3-
SHA:EDH-RSA-DES-CBC3-SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:AES128-SHA256:AES256-
SHA256:AES128-SHA:AES256-SHA:DES-CBC3-SHA:!DSS';
        ssl_prefer_server_ciphers on;
        ssl_dhparam /etc/nginx/dhparams.pem;

        proxy_http_version 1.1;
        proxy_set_header Connection "";
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

        location / {
                proxy_pass http://sonar;
                proxy_redirect http://sonarqube.example.com
https://sonarqube.example.com;
        }
}


upstream deploy{
        server jboss:8080;
        keepalive 10;
}

server {
        listen 443 ssl http2;
        server_name         deploy.example.com;
        ssl_certificate     /etc/nginx/example.com.crt;
        ssl_certificate_key /etc/nginx/example.com.key;
        ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
        ssl_ciphers 'ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-
ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-
SHA384:ECDHE-RSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-
SHA384:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-
SHA:ECDHE-RSA-AES256-SHA384:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA384:ECDHE-
ECDSA-AES256-SHA:ECDHE-RSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA:DHE-
RSA-AES256-SHA256:DHE-RSA-AES256-SHA:ECDHE-ECDSA-DES-CBC3-SHA:ECDHE-RSA-DES-CBC3-
SHA:EDH-RSA-DES-CBC3-SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:AES128-SHA256:AES256-
SHA256:AES128-SHA:AES256-SHA:DES-CBC3-SHA:!DSS';
        ssl_prefer_server_ciphers on;
        ssl_dhparam /etc/nginx/dhparams.pem;

        proxy_http_version 1.1;
        proxy_set_header Connection "";
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

        location / {
                proxy_pass http://deploy;
                proxy_redirect http://deploy.example.com https://deploy.example.com;
        }
}
```

6.2. Anexo 2

#Generación, centralización y almacenamiento de un paquete o dependencia software.

```
pipeline {
    agent any

    stages {
        stage('Gitlab checkout') {
            steps {
                checkout([$class: 'GitSCM', branches: [[name: '*/master']],
extensions: [], userRemoteConfigs: [[ credentialsId: 'ssh-jenkins-gitlab-key',url:
'git@gitlab:jenkins-group/hello-world.git']]])
            }
        }
        stage('Docker Maven build') {
            steps{
                sh 'docker run --rm --volumes-from ic_jenkins_1 -w ${PWD}
d3sarrollo/maven-3.8.1-open-jdk-8 clean package'
                archiveArtifacts artifacts: 'target/*.jar', followSymlinks: false,
onlyIfSuccessful: true
            }
        }

        stage('Docker Sonarqube analysis') {
            steps{
                sh 'docker run --network ic_default --rm --volumes-from ic_jenkins_1
-w ${PWD} -e SONAR_HOST_URL="http://sonarqube:9000" -e
SONAR_LOGIN="e5503766c1d1282e83b25c2afae003fc944d1047" sonarsource/sonar-scanner-cli
-Dsonar.projectKey=IC_CD_artifact_demo -Dsonar.sources=src -
Dsonar.java.binaries=target'
            }
        }

        stage('Nexus publishing') {
          steps{
              nexusArtifactUploader artifacts: [[artifactId: 'demo', classifier: '',
file: 'target/demo-0.0.1-SNAPSHOT.jar', type: 'jar']], credentialsId: 'jenkins-
nexus', groupId: 'com.example', nexusUrl: 'nexus:8081', nexusVersion: 'nexus3',
protocol: 'http', repository: 'my-repo/', version: '1.0.0'
          }
        }

    }

    post {
        always {
            emailext attachLog: true, to: 'd3sarrollo@gmail.com', body:
"${currentBuild.currentResult}: Job ${env.JOB_NAME} build ${env.BUILD_NUMBER}\n More
info at: ${env.BUILD_URL}", subject: "Jenkins Build ${currentBuild.currentResult}:
Job ${env.JOB_NAME}"
        }
    }
}
```

6.3. Anexo 3

```
#Generación, centralización, almacenamiento y despliegue de una aplicación


pipeline {
    agent any

    stages {
        stage('Gitlab checkout') {
            steps {
                checkout([$class: 'GitSCM', branches: [[name: '*/master']],
extensions: [], userRemoteConfigs: [[ credentialsId: 'ssh-jenkins-gitlab-key',url:
'git@gitlab:jenkins-group/demo-amap.git']]])
            }
        }
        stage('Docker Maven build and test') {
            steps{
                configFileProvider([configFile(fileId: 'a952bafc-5676-4a87-8161-
f3cf19434538', targetLocation: './settings-amap.xml')]) {
                    sh 'docker run --rm --volumes-from ic_jenkins_1 -w ${PWD}
d3sarrollo/maven-3.8.1-open-jdk-8 clean package -s settings-amap.xml'
                    archiveArtifacts artifacts: 'target/*.war', followSymlinks:
false, onlyIfSuccessful: true
                }

            }
        }

        stage('Docker Sonarqube analysis') {
            steps{
                sh 'docker run --network ic_default --rm --volumes-from ic_jenkins_1
-w ${PWD} -e SONAR_HOST_URL="http://sonarqube:9000" -e
SONAR_LOGIN="e5503766c1d1282e83b25c2afae003fc944d1047" sonarsource/sonar-scanner-cli
-Dsonar.projectKey=amap_demo -Dsonar.sources=src -Dsonar.java.binaries=target'
            }
        }

        stage('Jboss war deploy') {
            steps{
                sh 'docker cp target/demo-1.0.0.war
jboss_amap:/opt/jboss/wildfly/standalone/deployments'
            }
        }
    }

    post {
        always {
            emailext attachLog: true, to: 'd3sarrollo@gmail.com', body:
"${currentBuild.currentResult}: Job ${env.JOB_NAME} build ${env.BUILD_NUMBER}\n More
info at: ${env.BUILD_URL}", subject: "Jenkins Build ${currentBuild.currentResult}:
Job ${env.JOB_NAME}"
        }
    }
}
```

6.4. Anexo 4

```
#Listado de plugins de Jenkins disponibles durante la implementación (no quiere decir
#que se hayan utilizado todos). Algunos de ellos forman parte de la funcionalidad
#básica de Jenkins y otros han tenido que instalarse para añadir nuevas.


Ant Plugin
Adds Apache Ant support to Jenkins
       1.11

Apache HttpComponents Client 4.x API Plugin
Bundles Apache HttpComponents Client 4.x and allows it to be used by Jenkins plugins.
       4.5.13-1.0

Bootstrap 4 API Plugin
Provides Bootstrap 4 for Jenkins plugins.
       4.6.0-3

bouncycastle API
This plugin provides a stable API to Bouncy Castle related tasks.
       2.20

Branch API
This plugin provides an API for multiple branch based projects.
       2.6.4

Build Timeout
This plugin allows builds to be automatically terminated after the specified amount
of time has elapsed.
       1.20

Checks API
This plugin defines an API for Jenkins to publish checks to SCM platforms.
       1.7.0

Command Agent Launcher Plugin
Allows agents to be launched using a specified command.
       1.6

Config File Provider Plugin
Ability to provide configuration files (e.g. settings.xml for maven, XML, groovy,
custom files,...) loaded through the UI which will be copied to the job workspace.
This plugin is up for adoption! We are looking for new maintainers. Visit our Adopt a
Plugin initiative for more information.
       3.8.1

Credentials
This plugin allows you to store credentials in Jenkins.
       2.3.18

Credentials Binding
Allows credentials to be bound to environment variables for use from miscellaneous
build steps.
       1.24

Deploy to container Plugin
This plugin allows you to deploy a war to a container after a successful build.
Glassfish 3.x remote deployment
       1.16

Display URL API
Provides the DisplayURLProvider extension point to provide alternate URLs for use in
notifications
       2.3.4
```

Durable Task
Library offering an extension point for processes which can run outside of Jenkins yet be monitored.
	1.36

ECharts API
Provides ECharts for Jenkins plugins.
	5.1.0-2

Email Extension
This plugin is a replacement for Jenkins's email publisher. It allows to configure every aspect of email notifications: when an email is sent, who should receive it and what the email says
	2.82

Folders
This plugin allows users to create "folders" to organize jobs. Users can define custom taxonomies (like by project type, organization type etc). Folders are nestable and you can define views within folders. Maintained by CloudBees, Inc.
	6.15

Font Awesome API
Provides the free fonts of Font Awesome for Jenkins plugins.
	5.15.3-2

Generic Webhook Trigger
Can receive any HTTP request, extract any values from JSON or XML and trigger a job with those values available as variables. Works with GitHub, GitLab, Bitbucket, Jira and many more.
	1.72

Git
This plugin integrates Git with Jenkins.
	4.7.1

Git client
Utility plugin for Git support in Jenkins
	3.7.1

GIT server Plugin
Allows Jenkins to act as a Git server.
	1.9

GitHub
This plugin integrates GitHub to Jenkins.
	1.33.1

GitHub API Plugin
This plugin provides GitHub API for other plugins.
	1.123

GitHub Branch Source Plugin
Multibranch projects and organization folders from GitHub. Maintained by CloudBees, Inc.
	2.10.2

GitLab Plugin
This plugin allows GitLab to trigger Jenkins builds and display their results in the GitLab UI.
	1.5.20

Gradle
This plugin allows Jenkins to invoke Gradle build scripts directly.
	1.36

Jackson 2 API
This plugin exposes the Jackson 2 JSON APIs to other Jenkins plugins.
	2.12.3

Java JSON Web Token (JJWT) Plugin
Bundles the Java JSON Web Token (JJWT) library.
	0.11.2-9.c8b45b8bb173

JavaScript GUI Lib: ACE Editor bundle plugin
JavaScript GUI Lib: ACE Editor bundle plugin.
	1.1

JavaScript GUI Lib: Handlebars bundle plugin
JavaScript GUI Lib: Handlebars bundle plugin.
	3.0.8

JavaScript GUI Lib: Moment.js bundle plugin
JavaScript GUI Lib: Moment.js bundle plugin.
	1.1.1

JQuery3 API
Provides jQuery 3.x for Jenkins plugins.
	3.6.0-1

JSch dependency plugin
Jenkins plugin that brings the JSch library as a plugin dependency, and provides an
SSHAuthenticatorFactory for using JSch with the ssh-credentials plugin.
	0.1.55.2

JUnit
Allows JUnit-format test results to be published.
	1.49

LDAP
Adds LDAP authentication to Jenkins
	2.6

Lockable Resources
This plugin allows to define external resources (such as printers, phones, computers)
that can be locked by builds. If a build requires an external resource which is
already locked, it will wait for the resource to be free.
	2.10

Mailer Plugin
This plugin allows you to configure email notifications for build results
	1.34

Managed Scripts
This plugin allows to centrally manage shell scripts and reference these as build
steps in your builds.
This plugin is up for adoption! We are looking for new maintainers. Visit our Adopt a
Plugin initiative for more information.
	1.5.4

Matrix Authorization Strategy
Offers matrix-based security authorization strategies (global and per-project).
	2.6.6

Matrix Project Plugin
Multi-configuration (matrix) project type.
	1.18

Nexus Artifact Uploader
This plugin to upload the artifact to Nexus Repository.
	2.13

OkHttp Plugin
This plugin provides OkHttp for other plugins.
	3.14.9

Oracle Java SE Development Kit Installer Plugin
Allows the Oracle Java SE Development Kit (JDK) to be installed via download from Oracle's website.
	1.5

OWASP Markup Formatter Plugin
Uses the OWASP Java HTML Sanitizer to allow safe-seeming HTML markup to be entered in project descriptions and the like.
	2.1

PAM Authentication plugin
Adds Unix Pluggable Authentication Module (PAM) support to Jenkins
	1.6

Pipeline
A suite of plugins that lets you orchestrate automation, simple or complex. See Pipeline as Code with Jenkins for more details.
	2.6

Pipeline Graph Analysis
Provides a REST API to access pipeline and pipeline run data.
	1.10

Pipeline: API
Plugin that defines Pipeline API.
	2.42

Pipeline: Basic Steps
Commonly used steps for Pipelines.
	2.23

Pipeline: Build Step
Adds the Pipeline step build to trigger builds of other jobs.
	2.13

Pipeline: Declarative
An opinionated, declarative Pipeline.
	1.8.4

Pipeline: Declarative Extension Points API
APIs for extension points used in Declarative Pipelines.
	1.8.4

Pipeline: GitHub Groovy Libraries
Allows Pipeline Grrovy libraries to be loaded on the fly from GitHub.
	1.0

Pipeline: Groovy
Pipeline execution engine based on continuation passing style transformation of Groovy scripts.
	2.90

Pipeline: Input Step
Adds the Pipeline step input to wait for human input or approval.
	2.12

Pipeline: Job
Defines a new job type for pipelines and provides their generic user interface.
	2.40

Pipeline: Milestone Step
Plugin that provides the milestone step

```
       1.3.2

Pipeline: Model API
Model API for Declarative Pipeline.
       1.8.4

Pipeline: Multibranch
Enhances Pipeline plugin to handle branches better by automatically grouping builds
from different branches.
       2.23

Pipeline: Nodes and Processes
Pipeline steps locking agents and workspaces, and running external processes that may
survive a Jenkins restart or agent reconnection.
       2.38

Pipeline: REST API Plugin
Provides a REST API to access pipeline and pipeline run data.
       2.19

Pipeline: SCM Step
Adds a Pipeline step to check out or update working sources from various SCMs
(version control).
       2.12

Pipeline: Shared Groovy Libraries
Shared libraries for Pipeline scripts.
       2.19

Pipeline: Stage Step
Adds the Pipeline step stage to delineate portions of a build.
       2.5

Pipeline: Stage Tags Metadata
Library plugin for Pipeline stage tag metadata.
       1.8.4

Pipeline: Stage View Plugin
Pipeline Stage View Plugin.
       2.19

Pipeline: Step API
API for asynchronous build step primitive.
       2.23

Pipeline: Supporting APIs
Common utility implementations to build Pipeline Plugin
       3.8

Plain Credentials Plugin
Allows use of plain strings and files as credentials.
       1.7

Plugin Utilities API
Provides several utility classes that can be used to accelerate plugin development.
       2.2.0

Popper.js API Plugin
Provides Popper.js for Jenkins plugins.
       1.16.1-2

Resource Disposer
Dispose resources asynchronously. Utility plugin for resources that require more
retries or take a long time to delete.
       0.15
```

```
SCM API
This plugin provides a new enhanced API for interacting with SCM systems.
       2.6.4

Script Security
Allows Jenkins administrators to control what in-process scripts can be run by less-
privileged users.
       1.76

Snakeyaml API
This plugin provides Snakeyaml for other plugins.
       1.27.0

SonarQube Scanner for Jenkins
This plugin allows an easy integration of SonarQube, the open source platform for
Continuous Inspection of code quality.
       2.13.1

SSH Build Agents plugin
Allows to launch agents over SSH, using a Java implementation of the SSH protocol.
       1.31.5

SSH Credentials Plugin
Allows storage of SSH credentials in Jenkins
       1.18.1

Structs
Library plugin for DSL plugins that need names for Jenkins objects.
       1.22

Timestamper
Adds timestamps to the Console Output
       1.12

Token Macro
This plug-in adds reusable macro expansion capability for other plug-ins to use.
       2.15

Trilead API Plugin
Trilead API Plugin provides the Trilead library to any dependent plugins in an easily
update-able manner.
       1.0.13

Workspace Cleanup Plugin
This plugin deletes the project workspace when invoked.
       0.39
```

7. Referencias y bibliografía

[1] Atlassian (2021) *Flujo de trabajo de Gitflow.* Disponible en agosto 2021 en: https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow

[2] Blanco, A. (2018) *Características principales de arquitecturas orientadas a microservicios*. Disponible en agosto 2021 en: https://www.autentia.com/2018/09/10/caracteristicas-principales-de-arquitecturas-orientadas-a-microservicios/

[3] Docker (2021) *Docker Compose*. Disponible en agosto 2021 en: https://docs.docker.com/compose/

[4] Docker (2021) *Docker Docs*. Disponible en agosto 2021 en: https://docs.docker.com/get-started/overview/

[5] Fowler, M. (2006) *Continuous Integration*. Disponible en agosto 2021 en: https://martinfowler.com/articles/continuousIntegration.html

[6] Gitlab (2021) *Gitlab Docs*. Disponible en agosto 2021 en: https://docs.gitlab.com/ee/index.html

[7] Iglesias, D. (2016) *Aceleración SSL/TLS con NGINX*. Disponible en agosto 2021 en: https://enmilocalfunciona.io/aceleracion-ssl-tls-con-nginx/

[8] Jenkins (2021) *Jenkins Handbook*. Disponible en agosto 2021 en: https://www.jenkins.io/doc/book/

[9] Kabir, A. (2017) *Continuous Integration*: A "Typical" Process, , Disponible en agosto 2021 en: 2021, de https://developers.redhat.com/blog/2017/09/06/continuous-integration-a-typical-process

[10] Maven (2021) *Maven Use*. Disponible en agosto 2021 en: https://maven.apache.org/users/index.html

[11] Nginx (2021) *Nginx Documentation*. Disponible en agosto 2021 en: https://nginx.org/en/docs/

[12] Opentix (2019) *¿Por qué deberías usar un sistema de control de versiones si eres desarrollador?* Disponible en agosto 2021 en: https://www.opentix.es/sistema-de-control-de-versiones/

[13] Red Hat (2021), *¿Qué son la integración/distribución continuas (CI/CD)?* Disponible en agosto 2021 en: https://www.redhat.com/es/topics/devops/what-is-ci-cd

[14] Red Hat (2021), *Wildfly Documentation*. Disponible en agosto 2021 en: https://docs.wildfly.org/23/

[15] Braun, A. (2018) *Cloud Native with Containers and Kubernetes – Part 2.* Disponible en agosto 2021 en: https://blogs.sap.com/2018/06/05/cloud-native-with-containers-and-kubernetes-part-2/

[16] Shash, M. (2020) *Stop messing up with CI/CD vs. DevOps and learn the difference finally*. Disponible en agosto 2021 en: https://www.itproportal.com/features/stop-messing-up-with-cicd-vs-devops-and-learn-the-difference-finally/

[17] Sites Google (2021), *Continuous Integration.* Disponible en agosto 2021 en: https://sites.google.com/site/practicadesarrollosoft/temario/continuous-integration

[18] Sonarsource (2021) *SonarQube Docs*. Disponible en agosto 2021 en: https://docs.sonarqube.org/latest/

[19] Sonatype (2021) *Repository Manager 3 Sonatype Documentation*. Disponible en agosto 2021 en: https://help.sonatype.com/repomanager3

[20] Steven, J. (2018) *What's the difference between agile, CI/CD, and DevOps.* Disponible en agosto 2021 en: https://www.synopsys.com/blogs/software-security/agile-cicd-devops-difference/

[21] Webipedia (2018) *Curso de Maven*. Disponible en agosto 2021 en: http://webipedia.es/tecnologia/cursos/maven-tutorial-que-es/

[22] Wikipedia (2021) *DevOps*. Disponible en agosto 2021 en: https://es.wikipedia.org/wiki/DevOps
[23] Wikipedia (2021) *Docker*. Disponible en agosto 2021 en: https://es.wikipedia.org/wiki/Docker_(software)
[24] Wikipedia (2021) *Gitlab*. Disponible en agosto 2021 en: https://es.wikipedia.org/wiki/GitLab
[25] Wikipedia (2021) *Jenkins*. Disponible en agosto 2021 en: https://es.wikipedia.org/wiki/Jenkins
[26] Wikipedia (2021) *Maven*. Disponible en agosto 2021 en: https://es.wikipedia.org/wiki/Maven
[27] Wikipedia (2021) *Nginx*. Disponible en agosto 2021 en: https://es.wikipedia.org/wiki/Nginx
[28] Wikipedia (2021) *SonarQube*. Disponible en agosto 2021 en: https://es.wikipedia.org/wiki/SonarQube
[29] Wikipedia (2021) *Wildfly*. Disponible en agosto 2021 en: https://es.wikipedia.org/wiki/WildFly