Trabajo Fin de Grado

Grado en Ingeniería de las Tecnologías de Telecomunicación

Integración y Despliegue Continuo: Monitorización de y automatización de pruebas software

Autor: José Enrique Romero Bersabé

Tutor: Pablo Nebrera Herrara

Dep. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Integración y Despliegue Continuo: Monitorización de y automatización de pruebas software

Autor:

José Enrique Romero Bersabé

Tutor:

Pablo Nebrera Herrera

Profesor titular

Dep. de Ingeniería Telemática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2016

Proyecto Fin de Carrera: Integración y Despliegue Continuo: Monitorización de y automatización de pruebas software

Autor: José Enrique Romero Bersabé
Tutor: Pablo Nebrera Herrera

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2016

El Secretario del Tribunal

A mi familia

A mis compañeros

Agradecimientos

Presentando este proyecto pongo punto y final a una de las etapas más importantes en mi vida tanto a nivel personal como profesional. Este es el pequeño homenaje que tengo el honor de dedicarle a mis allegados, aquellos que han vivido y han sido partícipes de todo el esfuerzo que se ha dedicado en esta carrera.

En primer lugar agradecer a mis padres, Manuel y María José, y a mis hermanos, Manuel y Marta, que han sido los principales culpables de que llegue a este punto, sobre todo a mi madre, que ha estado ahí siempre que la he necesitado y ha hecho lo imposible para que pueda obtener este título. Es algo que jamás olvidaré y un simple gracias no sería justo.

Agradecer a mi abuela, Josefa, que tiene más ganas que yo de que termine la carrera y siempre estaba ahí recibiéndome con una sonrisa y dándome ánimos. Ya solo queda que me vea trabajar.

Agradecer, como no, a mis compañeros de clases el gran apoyo que me han ofrecido. Se puede decir que hemos sido una pequeña familia donde nos hemos ayudado cada vez que lo hemos necesitado. Cris, Vera, Morillo, Narváez, Ismael, Gerard, Luis, David, Juanan,…, gracias por esas salidas, esas cervezas, esos días de encerronas de estudio y sobre todo, gracias por esos suspensos juntos que hemos tenido. Nunca os olvidaré aunque nuestros caminos se hayan separado.

Del mismo modo es necesario agradecer el apoyo de mi tutor Pablo Nebrera por haberme proporcionado todo lo necesario para la realización de este proyecto en un gran ambiente de trabajo, y a su vez, me ha enseñado algo con lo que poder dedicarme profesionalmente y crecer como persona.

Por último y no menos importante, agradecer a Lola, que ella sí que ha estado siempre en los momentos más difíciles y ha estado apoyándome, aconsejándome y aguantándome. Llegarán mejores momentos y los disfrutaremos juntos.

José Enrique Romero Bersabé

Sevilla, 2016

Resumen

Las empresas actuales centran sus esfuerzos en conseguir una buena calidad del software que desarrollan, haciendo uso de metodologías, procedimientos y estándares adecuados para poder desarrollar y lanzar un producto software de calidad con la suficiente certeza de obtener un menor coste y riesgo finales. Es por ello que se han establecidos nuevos paradigmas de organización que proponen el uso de nuevas metodologías y procesos para permitir una puesta en producción ágil y menos arriesgada. En este trabajo, la implantación de estas técnicas se basa en llevar a cabo la automatización de todos los pasos entre el desarrollo y la puesta en producción de software.

Dentro de las tareas necesarias para implementar esta técnica, este trabajo lleva a cabo la identificación de las herramientas básicas que permiten la Integración Continua, así como el estudio de mecanismos de integración de todas estas herramientas, automatizando todo el proceso desde la construcción del proyecto hasta su puesta en producción. Se llevarán a cabo despliegues del producto software diarios que permitan recoger realimentación más rápidamente de las incidencias y poder reaccionar ágilmente en aquellas situaciones que la requieran.

Abstract

Nowadays, current companies are trying to achieve a developed software with a good quality. In order to make it, they are using appropriate methods, procedures and standars with wich they know for certain that it will cost less money and will have less final risks. For this reason, there have appeared new technologies and producer allowing a quicker and lees risky reléase. In this document, the implementation of this technique is about making all the steps between the development and the reléase of the software, automatic.

One of the necesary tasks to implement this method is to identify the basic tools wich permit the Continuous Integration. Besides this one, in this document we also can find the study of the mechanisms that integrates all these tools. These mechanism make the whole development porccess automatic. As a result, the companies can deploy the software daily in order to have a faster feedback from the incidents and to react easily in situations that require it.

Índice

Índice de tablas

Índice de figuras

1 INTRODUCCIÓN

1.1 Motivación

Muchos de los proyectos software que se desarrollan se acometen usando metodologías y procedimientos obsoletos, lo que provoca que el equipo encargado del proyecto acabe lanzando productos y servicios con retardos o con fallos.

Durante el ciclo de vida software, éste pasa por diferentes entornos (el entorno de desarrollo, de pruebas, de producción,…) y un error muy habitual sería desarrollar un producto dependiente de su entorno de trabajo, y a la hora de desplegarlo en producción donde el cliente interactúa con dicho producto o servicio, éste no funcione correctamente. Por ello, se hace necesario un paso intermedio entre estos dos entornos, que es el entorno de pruebas, donde se va a verificar el correcto funcionamiento del software.

Debido a la gran variedad de proyectos y lenguajes de programación con los que se trabaja, realizar manualmente el proceso de llevar el software hasta el entorno de producción (o cualquier otro entorno), se vuelve difícilmente repetible y poco fiable.

Todos estos inconvenientes provocan que el proceso que va desde el desarrollo software hasta su puesta en producción pueda llegar a tomar en la mayoría de los casos semanas, retardo que para muchas empresas puede traducirse en un alto coste de oportunidades y en un alto coste económico.

Es por ello que las empresas necesitan plantearse la calidad del software con una perspectiva que abarque tanto la calidad final del producto, como la calidad del mismo mientras se está desarrollando. Es necesario hacer uso de metodologías y procedimientos adecuados para poder desarrollar y lanzar un producto de calidad, con la suficiente certeza de obtener un menor coste y riegos finales.

El proceso de integración continua tiene como objetivo principal comprobar que cada actualización del código fuente no genere problemas en la aplicación que se está desarrollando. Se podrán automatizar tareas que abarcan desde que el desarrollador modifica el código fuente del repositorio software hasta su puesta en producción (o hasta el entorno que más le convenga a la organización).

Los ciclos de vida software cuentan con procesos que aíslan las tareas llevadas a cabo por los diferentes equipos de proyectos. Estos son el equipo de desarrollo, de pruebas, de sistemas,… de manera que existe una importante barrera entre estos. La integración Continua permite solucionar en parte la barrera mencionada entre los distintos equipos del proyecto, reduciendo el retardo entre fases que dependen de diferentes grupos, haciendo que dichas tareas puedan ser llevadas a cabo mediante la pulsación de un único botón o mediante la modificación del código fuente en el repositorio.

1.2 Objetivos del proyecto

El objetivo fundamental de este presente documento consiste en demostrar que se puede llevar a cabo la automatización de tareas que abarcan desde su recién desarrollo hasta su puesta en producción, facilitando una rápida realimentación a todos los miembros del equipo que intervienen en el proceso de creación del software. Se automatizarán tareas como la obtención y compilación del código fuente, despliegues, tests unitarios y de integración, así como la generación de paquetes RPM del servicio o producto y subirlo a un repositorio público con el propósito de que los clientes y la propia organización desplieguen los servicios que necesiten más cómodamente. Antes de realizar el último paso de puesta en producción, el servicio o producto desarrollado debe pasar ciertas pruebas que garanticen su correcto funcionamiento.

Esta automatización de tareas tiene el objetivo de mantener testeado el proyecto en todo momento y con cada cambio. Esto ayudará a mejorar la calidad del proyecto detectando errores lo más rápido posible, reduciendo procesos repetitivos manuales, permitiendo así tener una mejor visibilidad del estado del proyecto.

La siguiente imagen muestra los objetivos que se pretenden conseguir:

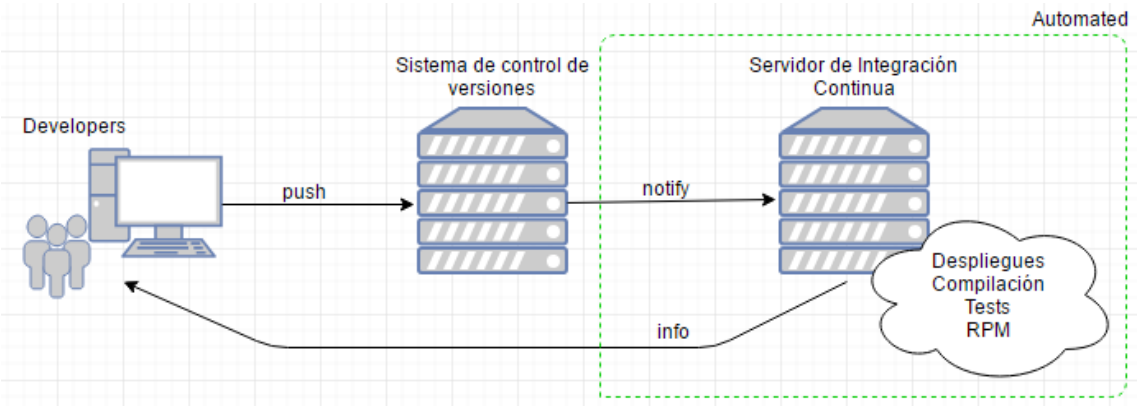


Figura 1-1. Objetivos

Tras un cambio en el código fuente del repositorio realizado por los desarrolladores, se automatizan las tareas citadas anteriormente para verificar el correcto funcionamiento del servicio.

2 INTEGRACIÓN Y DESPLIEGUE CONTINUO

La integración continua consiste en la ejecución programada de procesos que revisan periódicamente los cambios realizados en el código de un proyecto a través del sistema de control de versiones (git, 2enkins2ón, mercurial,…) para proceder si es necesario a la compilación del código, la realización de pruebas unitarias y de integración, cálculos de métricas de calidad, etc…

El objetivo final consiste en monitorizar constantemente los proyectos de forma que cualquier incidencia sea detectada automáticamente lo antes posible y comunicada a los responsables para proceder a su resolución.

La Integración Continua se centra en tener un feedback casi instantáneo del estado de cada entrega de código al repositorio, a través de la automatización del proceso de construcción del software. Nos da la posibilidad de detectar fallos de integración en las primeras etapas de la construcción de nuestro software, aumentando notablemente la calidad el software entregado.

Utilizar herramientas de integración continua en los procesos de desarrollo software y de supervisión de calidad trae consigo diversas ventajas:

- Teniendo en cuenta los errores que se pueden producir, no se espera que alguien descubra cualquier incidencia que se produzca puesto que la integración continua hace esto por nosotros.
- Como consecuencia de lo anterior, los problemas se pueden afrontar de inmediato.
- El origen de los problemas detectados es fácilmente identificable.
- Se agiliza el proceso de desarrollo ya que consiguen que algunas tareas se ejecuten de forma automática y cuando más nos convenga en función del proyecto. Esto libera a los desarrolladores de tener que realizar tareas manualmente repetitivas y que a menudo suelen ser tediosas.
- Se obtiene una gran visibilidad del estado del proyecto.

Se puede dar un paso más, y además de llevar a cabo procesos de integración continua realizar un Despliegue Continuo, que consiste en la publicación y entrega automática de nuevas versiones correspondientes a la última versión del código de la aplicación que haya superado satisfactoriamente las verificaciones que se hayan impuesto en la integración continua. Así pues, complementar los procesos de integración continua con procesos de entrega continua aporta las siguientes ventajas adicionales:

- El paso a producción puede ser totalmente programado y desatendido. Desde la primera línea de código, cada cambio pasa por el mismo conjunto de pasos que

- garantizan la robustez de la entrega. Su frecuente utilización le otorga estabilidad y confiabilidad, evitando largas puestas en producción y evitando sorpresas a última hora.
- Al igual que en el caso de la integración continua, se sigue automatizando tareas y liberando de su realización manual al equipo de desarrollo.
- Los problemas aparecen y son abordados en fases más tempranas.
- Total flexibilidad para adoptar nuevos cambios.

La diferencia entre la integración y despliegue continuo viene representado en la siguiente figura:

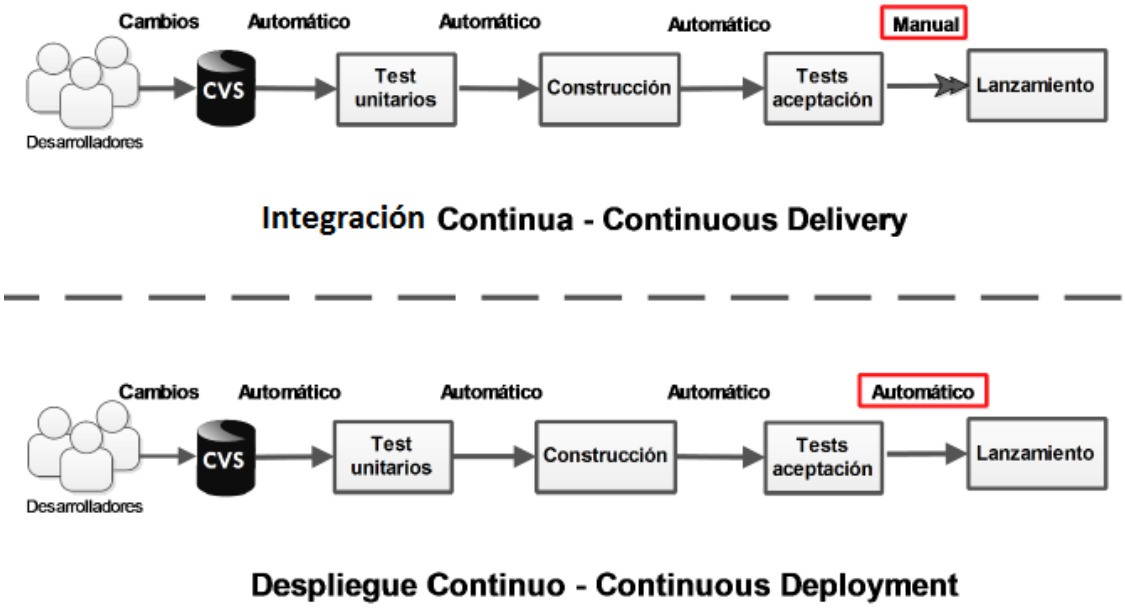


Figura 2-1. Diferencia entre Entrega y Despliegue Continuo

2.1 Automatización de pruebas

Siempre que se lleve a cabo un desarrollo software, para garantizar que se cumple un determinado nivel de calidad del mismo, es necesario verificar el conjunto de requisitos y funcionalidades que debe proveer éste. Esta verificación se lleva a cabo mediante la ejecución de unas pruebas que tratan de encontrar algunos errores en el software y en su funcionalidad, por lo que se podría decir que las pruebas constituyen una de las etapas más críticas dentro del ciclo de vida del proyecto.

El proceso típico de pruebas engloba las fases representadas en la siguiente figura:

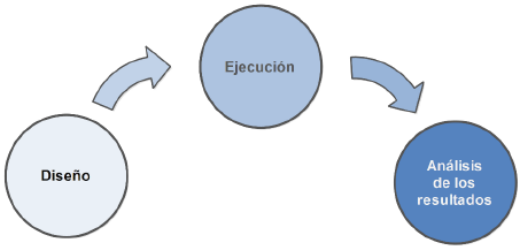


Figura 2-2. Proceso de automatización de pruebas

Lo que se pretende automatizar son las dos últimas fases de esta figura. Cuando sea necesario, se ejecutarán las pruebas diseñadas y se analizarán los resultados mediante la comparación de los resultados obtenidos con los esperados, informando al desarrollador en caso de incidencia.

Los beneficios de automatizar las pruebas serían los siguientes:

- Rapidez: Las herramientas de testing automatizado corren las pruebas significativamente más rápido que los seres humanos.
- Esto conlleva a un aumento de productividad por parte del equipo de pruebas, ya que el ahorro de tiempo debido a la automatización de pruebas y al análisis se dedican a la elaboración de nuevas pruebas.
- Fiabilidad: Las pruebas ejecutan precisamente las mismas operaciones cada vez que se ejecutan, eliminando el error humano.
- Se pueden realizar varias pruebas en paralelo en una o varias máquinas.
- Posibilidad de ejecución de pruebas más complicadas, ya que algunas pruebas son demasiado complejas para poder ser llevadas a cabo manualmente.
- Reusabilidad de los scripts.

Por mencionar algún contra de la automatización de pruebas, se podría decir que el esfuerzo inicial es mayor y el mantenimiento de los scripts puede ser costoso.

2.2 Pipeline

Una forma de implantar la integración y la entrega continua es mediante una abstracción denominada pipeline. Un pipeline trata de modelar el conjunto de fases y tareas automatizadas que se deben llevar a cabo en el proceso que va desde la obtención del software hasta que éste se encuentra en producción.

Dichas tareas y fases se definen en forma de código ejecutándose de manera secuencial, siendo la entrada de cada una la salida de la anterior, de manera que si se produce un fallo en cualquier punto del pipeline deja de ejecutarse los pasos posteriores.

En este proyecto, las fases por las que transcurre el pipeline viene representado por la siguiente figura:

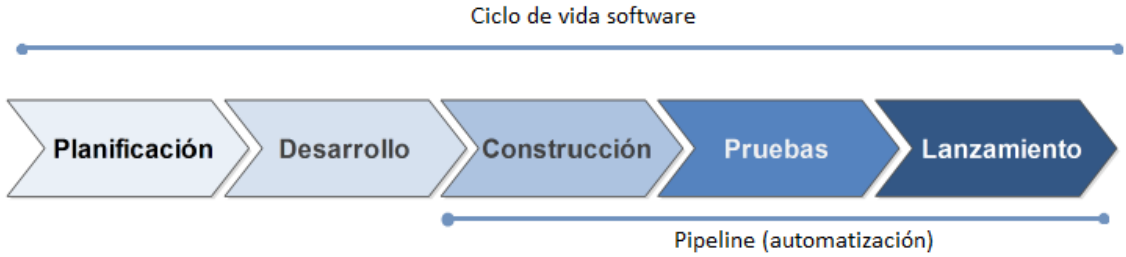


Figura 2-3.Fases por las que trascurre el pipeline

En el pipeline se indicarán los pasos que se desea realizar desde su construcción (build) hasta su lanzamiento. Podrás abarcar tareas como la obtención del código fuente, generar el build (compilación más ejecutables), realizar despliegue en distintos entornos, ejecutar pruebas, etc…

Cuando el sistema de control de versiones notifica al servidor de integración continua, éste realiza los pasos descritos en el pipeline, consiguiendo así una automatización de las tareas mencionadas.

En la siguiente figura se muestra un diagrama de mensajes que representa el proceso automatizado que se pretende conseguir con el pipeline:

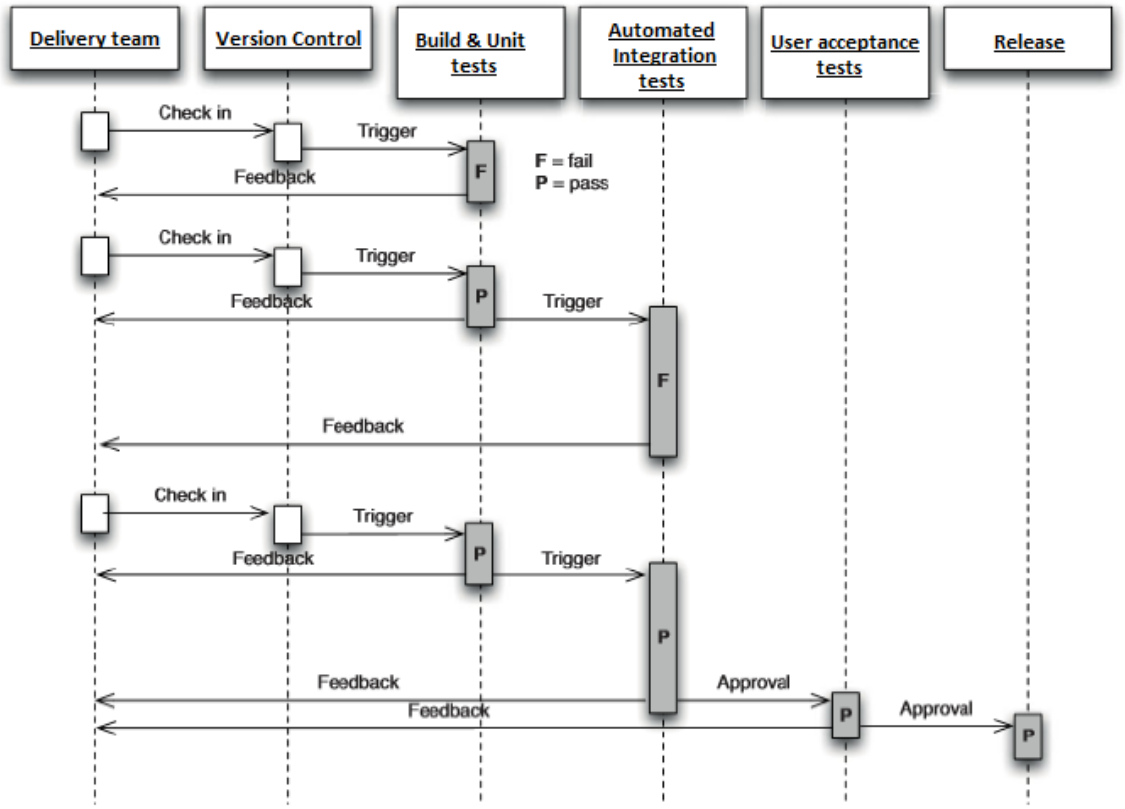


Figura 2-4. Diagrama de mensajes del proceso automatizado

El proceso se inicia cuando el equipo encargado del proyecto lleva a cabo una actualización en el sistema de control de versiones, momento en el que se dispara la fase de construcción y ejecución de pruebas unitarias. Una vez acabada, esta fase puede devolver dos resultados.

1. Se produce algún fallo en las pruebas unitarias y no se construye el proyecto, o las pruebas unitarias son correctas pero hay un fallo durante la construcción del proyecto, notificando a las partes interesadas de que ha fallado.
2. Las pruebas son correctas y se construye el proyecto, notificando a las partes interesadas de que se ha acabado la fase con éxito y se dispara la ejecución de la siguiente fase.

La siguiente fase será la ejecución de las pruebas de integración automatizadas, de nuevo esta fase puede devolver dos resultados:

1. Se produce algún fallo en las pruebas de integración, notificando a las partes interesadas.
2. Todas las pruebas de integración acaban con éxito, se notifica a las partes interesadas de los resultados de las pruebas y de que es posible pasar a la siguiente fase.

En este caso la fase no se dispara automáticamente, sino que será necesario que el equipo dé el visto bueno para pasar a la siguiente fase. Si es necesario se ejecutarán pruebas de manera manual para asegurar el correcto funcionamiento del servicio o producto.

Por último, en vista de todos los resultados obtenidos, se valorará si el software está listo para su puesta en producción, en cuyo caso se realizará el despliegue final de la aplicación.

Más adelante se detallará el diseño y el montaje de cada una de las tareas con varios escenarios de ejemplos.

2.3 Herramientas de construcción de proyectos

Uno de los problemas que intenta abordar la integración continua, es la gestión de dependencias, para ello existen una serie de herramientas las cuales se encargan de realizar este proceso.

El uso de estas herramientas hace más sencilla la manera de gestionar y construir un proyecto, ya que ejecutando un simple comando se pueden realizar diversos pasos como la compilación y generación de ejecutables, tests, instalación, etc…

Existen un gran número de estas herramientas como Maven, Ant, Gradle, PEAR,… con las que se podrá gestionar y construir el proyecto de una manera más rápida y eficaz.

Usar una herramienta de construcción de proyectos te proporciona los siguientes beneficios:

- Implementar integración y pruebas continuas al software.
- Mejorar la calidad del producto.
- Mejorar y acelerar el proceso puesta en marcha de la solución.
- Eliminar tareas redundantes.
- Contar con información de construcción y liberaciones de manera automática, donde todo el equipo estará enterado si algúna build ha fallado.

A continuación, se explicará algunas herramientas de este tipo. De dichas herramientas, en este proyecto, la que se ha usado es ***Maven***.

2.3.1 Maven

Apache Maven es una de las herramientas más usadas en la gestión y construcción de proyecto software Java creada en 2002 por Jason van Zyl, y cuya funcionalidad se asemeja a la de Apache Ant, make, PEAR, CPAN, etc…

Esta herramienta se fundamenta en la estructura del proyecto software que se va a gestionar y un fichero que describe todas las propiedades del proyecto denominado POM (Project Object Model). Este último fichero está escrito en formato XML y contiene información tal como un identificador único del proyecto, licencia, miembros del proyecto, dependencias del proyecto, repositorios remotos de artefactos Maven (de los cuales se pueden obtener las dependencias requeridas por el proyecto), plugins que permitan añadir funcionalidad al proyecto y muchos otros.

Maven define un ciclo de vida estricto para una aplicación de manera que la ejecución de una de las fases de este ciclo implica la ejecución de todas las fases anteriores. De manera ordenada las fases que definen el ciclo de vida del proyecto son:

- *Validación (validate):* Validar que el proyecto es correcto.
- *Compilación (compile):* Compila las fuentes .java generando los respectivos ficheros .class.
- *Test (test):* Probar el código fuente usando un framework de pruebas unitarias, como por ejemplo Junit, y si falla algún test no sigue con la siguiente fase.
- *Empaquetar (package):* Empaquetar el código compilado y transformarlo en añgún tipo .jar o .war, según lo especifique el fichero POM.
- *Pruebas de integración (integration-test):* Procesar y desplegar el código en algún entorno donde se puedan ejecutar las pruebas de integración.
- *Verificación (verify):* Verificar que el código empaquetado es válido y cumple los criterios de calidad.
- *Instalación (install):* Instalar el código empaquetado en el repositorio local de Maven, para usarlo como dependencia de otros proyectos.
- *Despliegue (deploy):* Desplegar el código a un entorno.

Para poder llevar a cabo alguna de estas fases en nuestro código, tan solo se tendrá que ejecutar mvn y el nombre de la palabra entre paréntesis. Además van en cadena, es decir, si se empaqueta el código, Maven ejecutará desde la fase de validación a la fase de empaquetación.

Por otra parte, con Maven la gestión de dependencias entre módulos y distintas versiones de librerías se hace muy sencilla. Solo se tiene que indicar los módulos que componen el proyecto, o qué librerías utiliza el software que se está desarrollando en el fichero POM.

Además, en el caso de las librerías, no hay que descargarlas a mano. Maven posee un repositorio remoto (Maven central) donde se encuentran la mayoría de librerías que se utilizan en los desarrollos software, y que la propia herramienta descarga cuando sea necesario.

2.3.2 Ant

Apache Ant es una herramienta usada en programación para la realización de tareas mecánicas y repetitivas, normalmente durante la fase de compilación y construcción (build). Es, por tanto, un software para procesos de automatización de compilación, similar a Make pero desarrollado en lenguaje Java y requiere la plataforma Java, así que es más apropiado para la construcción de proyectos Java.

Esta herramienta tiene la ventaja de no depender de las órdenes del Shell de cada sistema operativo, si no que se basa también en archivos de configuración XML.

La diferencia más notable entre Ant y Make es que Ant utiliza XML para describir el proceso de generación y sus dependencias, mientras que Make utiliza formato makefile. Por defecto, el archivo XML se denomina *build.xml.*

Ant es software open source, y se lanza bajo la licencia *Apache Software*.

Para utilizar Ant basta con disponer de una distribución binaria de ANT y tener instalado la versión 1.4 o superior del JDK. La distribución binaria consiste en la siguiente estructura de directorios: La carpeta *"ant"* contiene las carpetas *"bin"* (que asimismo contiene scripts de lanzador), *"lib"* (que contiene las dependencias necesarias y los .JAR de Ant), *"docs"* (que contiene la documentación de Ant, descripción, imágenes y un manual), y *"etc"* (que contiene los valiosos archivos .XSL para crear informe de mejora de la salida XML de varias tareas, migrar los archivos de creación, …), pero solo se necesitan los directorios bin y lib para ejecutar ant.

```
ant
  +--- bin  // contains launcher scripts
  |
  +--- lib  // contains Ant jars plus necessary dependencies
  |
  +--- docs // contains documentation
  |      +--- ant2    // a brief description of ant2 requirements
  |      |
  |      +--- images  // various logos for html documentation
  |      |
  |      +--- manual  // Ant documentation (a must read ;-)
  |
  +--- etc // contains xsl goodies to:
           //   - create an enhanced report from xml output of various tasks.
           //   - migrate your build files and get rid of 'deprecated' warning
           //   - ... and more ;-)
```

Figura 2-5. Estructura directorios Apache Ant

Para ejecuar Ant basta con escribir ant en la línea de comandos. Si se ha especificado la ejecución de un objetivo por defecto en la cabecera del proyecto del fichero build.xml no es necesario el uso de ningún parámetro ya que se ejecutará este por defecto. Para utilizar un buildfile diferente a build.xml hay que añadir el parámetro *–buildfile name_file*, donde name_file es el buildfile que se quiere utilizar

2.3.3 Gradle

Gradle es una herramienta de automatización de construcción de código basado en los conceptos de Apache ant y maven pero intenta llevarlo todo un paso más allá. Para empezar proporciona un lenguaje específico de dominio (DSL) en vez del tradicional y verboso XML como forma de declarar la configuración del proyecto, además proporciona una construcción declarativa, ya que permite usar scripts basado en DSL, el cual extiende de Groovy, o sea, permite extender y programar las propias tareas de construcción. Por otro lado dispone de una gran flexibilidad que permite trabajar con ella utilizando otros lenguajes y no solo Java.

Otra característica, es la convención sobre configuración, ya que con el poder del DSL, Gradle permite agregar y gestionar las dependencias, crear taras, declarar propiedades, detallar configuraciones, aplicar plugins, definir repositorios, empaquetar el proyecto, realizar pruebas unitarias y de integración,…

2.4 Herramientas para realizar la Integración Continua

En este apartado se van a presentar herramientas que permiten realizar la Integración Continua. Son muchas las herramientas existentes, entre las cuales, caben destacar a Jenkins, TeamCity, Travis y Go Continuos Delivery.

Estas herramientas o servicios permiten automatizar todas las tareas anteriormente descritas, apoyándose en los pipelines, que son los encargados de decirle al servidor los pasos a seguir.

Se hará una comparativa entre las herramientas mencionadas con el fin de estudiar los beneficios con cada una de ellas.

En este proyecto se ha trabajado con la herramienta Jenkins, y tras presentar dichas herramientas, se explicará el por qué se escogió.

2.4.1 Travis

Travis es un sistema de Integración Continua especialmente diseñado para construir y testear proyectos en Github de forma automática. La integración con Github se hace mediant**e webhooks**, de forma transparente para nosotros. Cada vez que se haga un push en el repositorio, Travis será notificado y comenzará con la ejecución del correspondiente job.

Actualmente soporta sistemas Ubuntu y OS X. Entre los lenguajes y tecnologías soportados se encuentran: Java, JavaScript, Node.js, iOS, PHP, Python, Ruby, C/C++, C#, Go, entre otros.

Para poder automatizar las tareas se debe crear un fichero YAML llamado .travis.yml y ubicarlo en el directorio raíz del proyecto. Este fichero le indicará a Travis las tareas a realizar. El ciclo de vida de un build consta de dos etapas: install y script. Cada etapa tiene varias sub etapas:

- Fase de install:
  - Before_install
  - Install
- Fase de script:
  - Before_script
  - Script
  - After_success / after_failure
  - Before_deploy

- Deploy
- After_deploy
- After_script

En la fase de install se prepará el entorno para la ejecución de la buid, es decir, se instalarán aquellas herramientas y dependencias de las que depende el proyecto que se quiere automatizar. En la fase de script es donde se realizarán las tareas que se ejecutarán en la build.

En la versión gratuita, Travis encolará todos los Jobs, realizándolos uno por uno, mientras que en la versión de pago se podrán lanzar en paralelo. Tiene una sencilla configuración y es muy usado, por lo que hay bastante documentación.

Se mostrará una tabla a continuación con las ventajas y desventajas que aporta esta herramienta:

• Fácil de preparar. Todo lo que necesita es un fichero YAML que le indique al servidor las tareas a realizar. • Es Open Source. Es gratis para todos los públicos, aunque con algunas limitaciones. • Soporta OS X & Ubunutu • Soporta diversos lenguajes y tecnologías. • Gran comunidad. Gran documentación.	• Está limitada solo para Github. Por lo que aquellas organizaciones que dependan de Gitlab o de cualquier otra alternativa a Github se ven obligados a depender de otra herramienta de IC. • Para disponer de su máxima capacidad de trabajo, se debe pagar y es relativamente caro. • No soporta Windows.

Tabla 2-1. Ventajas e inconvenientes de Travis

2.4.2 TeamCity

TeamCity es un servidor de integración continua escrito en Java por JetBrains, está integrado como un contenedor Tomcat servlet y donde su primer lanzamiento fue en Octubre de 2006.

Se puede integrar con entornos de desarrollo como Eclipse, IntelliJ IDEA y Visual Studio y las plataformas compatibles son Java, .NET y Ruby.

Soporta diversos sistemas de control de versiones como Subversion, CVS, Git, Mercurial, Microsoft Visual SourceSafe, entre otro.

Tiene más de 100 plugins listos para su uso, con los cuales podrá integrar herramientas que inicialmente no soportaba.

TeamCity proporciona una gestión de usuario flexible, incluyendo la asignación de funciones de usuario, clasificar los usuarios en grupos, diferentes formas de autentificación y un registro con todas las acciones de los usuarios para la transparencia de toda la actividad en el servidor.

Los buenos desarrolladores de software no les gusta la duplicación de código. Del mismo modo, los ingenieros de buena construcción no les gustan la duplicación de la configuración. TeamCity entiende esto y ofrece varias formas de reutilizar los ajustes.

Los proyectos en TeamCity se pueden anidar dentro de otros proyectos y por lo tanto forman un árbol y permite la ejecución de tareas en secuencia o en paralelo.

La siguiente tabla muestra las ventajas y desventajas de dicha herramienta:

• Soporta diferentes entornos de compilación, diversos lenguajes y herramientas. • Fácil instalación. • Posee una brillante interfaz, cuyo uso es intuitivo. • Buena documentación. • Soporta diversos Sistemas de Control de Versiones.	• Es caro. Inicialmente es gratuito, pero cuando sobrepasas un cierto número configuraciones builds, debes pagar.

Tabla 2-2. Ventajas e inconvenientes de TeamCity

2.4.3 Go Continuous Delivery

Go CD (Continuous Delivery) es un servidor de código abierto escrito en Java, cuyo objetivo es llevar a cabo un proceso de entrega continua, es decir, un proceso de integración continua junto con un sistema de administración de lanzamientos, basándose en un modelo Agentes – Servidor. Fue creado por el grupo ThroughtWorks y no fue hasta el 25 de febrero de 2014 cuando fue liberado bajo licencia Apache 2.0. Puede procesar proyecto escritos en todo tipo de lenguajes de programación, siendo básicamente solvente con proyectos desarrollados en Java.

El sistema modelo Agentes – Servidor permite distribuir el trabajo sobre los agentes. Esta distribución de trabajos se lleva a cabo con un conjunto de agentes que contactan con el servidor periódicamente, para que cuando haya una tarea que deba ser llevada a cabo por un agente en concreto, el cual cumple con la configuración esperada, sea asignada a éste. Cuando un agente crea una tarea se crea un directorio *sandbox*, sobre el cual se actualizan todos los materiales asociados a éste, como por ejemplo el contenido de un proyecto alojado en el repositorio.

Los pipelines son el núcleo de funcionamiento de Go CD. Un pipeline está compuesto por etapas las cuales se ejecutan secuencialmente. Cada una de estas etapas se compone a su vez de un conjunto de trabajos que se pueden llevar a cabo concurrentemente.

Cuando todas las tareas de una misma etapa hayan terminado con éxito, se considerará que esta etapa ha acabado con éxito y entonces se podrá ejecutar la siguiente etapa definida en este pipeline.

Los pipelines pueden ser disparados cuando haya una modificación del código fuente del repositorio, o cuando otro pipeline haya acabado con éxito, o incluso puede depender de los dos a la vez.

Este servidor provee un sistema de notificaciones basado principalmente en correo electrónico, medio por el cual, se avisa a los diferentes integrantes del proyecto de los estados alcanzado por las diferentes tareas del proceso de suministro continuo.

Además, esta herramienta puede extenderse mediante un sistema de plugins, aunque cabe decir que aún no se han desarrollado muchos, por lo que la integración con herramientas externas es ahora mismo pobre.

La siguiente tabla se mostrará las principales ventajas e inconvenientes de esta herramienta:

• Soporta pipelines. • Interfaz muy completa e intuitiva de usar. • Sencilla paralelización de trabajos. • Alta trazabilidad. Provee información de los cambios de todos los componentes que han provocado un estado concreto en un pipeline. • Sistema de usuarios con alta granularidad de permisos.	• No dispone de muchos plugins. Esto conlleva a que no soportará muchos más lenguajes o tecnologías de los que soporta inicialmente. • Algo difícil de configurar.

Tabla 2-3. Ventajas e inconvenientes de Go Continuous Delivery

2.4.4 Jenkins

Jenkins es un servidor de integración continua dirigido por CloudBees y escrito en Java, el cual deriva del proyecto Hudson. Es distribuido bajo licencia MIT y permite llevar a cabo las siguientes acciones:

- Construcción continua y pruebas automatizadas de proyectos software.
- Monitorización de la ejecución de servicios externos.
- Despliegue automático.

Puede trabajar con diversos lenguajes de programación como Java, Ruby, C, Python,… y permite usar un gran número de herramientas de construcción como Ant, Maven, Kundo, Gradle,… ya sea por soporte propio o mediante plugins.

El servidor Jenkins obtiene la información derivada de los fallos de construcción y permite enviar, a través de un sistema de notificaciones, esta información a los miembros encargados del proyecto. Este sistema cabe destacar por la cantidad de vías por las que se pueden enviar las notificaciones: Android, e-mail, Google Calendar, IRC, XMPP, RSS y Twitter.

A diferencia de otros servidores de integración continua, no es necesario hacer una compleja instalación, sino que para usarlo simplemente se debe desplegar un fichero WAR.

Permite usar una gran cantidad de SCMs, algunos de ellos son CVS, Git, Bazaar, Integrity, Mercurial, Perforce, Subversion, Vault, etc…

Para extender su funcionalidad tiene un gran número de plugins. Estos plugins permiten integrar este servidor con diversas herramientas, como herramientas de construcción (Ant, Maven,…) o herramientas de pruebas (Jmeter, xUnit, MDTest TRX,…).

Dispone de una interfaz gráfica la cual facilita su uso y configuración mediante formularios webs. Permite crear y configurar los llamados ‘jobs’, que es el conjunto de tareas parametrizables que definirán el proceso de integración continua que queremos realizar en un proyecto concreto.

Además, Jenkins también puede trabajar con el pipeline. Cuando sea necesario, Jenkins ejecutará las tareas descritas en dicho fichero. Dichas tareas pueden ser ejecutadas en el mismo servidor de Jenkins o pueden ser desplegadas en *máquinas esclavas*. Puede realizar trabajos en paralelo en una misma máquina.

Se puede notar que Jenkins constituye uno de los servidores de integración continua más completos, que destaca debido a su alta funcionalidad, integración con muchos entornos y herramientas, una importante extensibilidad, que como se ha descrito en el documento se lleva a cabo mediante plugins y por la cantidad de documentación que existe de esta herramienta. A su vez este servidor se ve avalado por una gran comunidad.

La siguiente tabla muestra las principales ventajas e inconvenientes de usar esta herramienta:

• Es Open Source • Muchísimos plugins que extienden drásticamente su funcionalidad. • La interfaz es sencilla y la aplicación resulta fácil de utilizar. • Herramienta potente y configurable. • Es muy popular, por lo que es fácil encontrar documentación y ayuda en la red. • Paralelización de trabajos. • Escalable.	• Algunos plugins sin la calidad suficiente. Puedes encontrarte con algún bug, o que al cambiar de versión Jenkins, algún plugin que se encuentre en el repositorio oficial puede dejar de funcionar.

Tabla 2-4. Ventajas e inconvenientes de Jenkins

2.4.5 Tabla comparativa

El sistema de control de versiones que se usará en este proyecto en Gitlab, por lo que se deberá escoger un servidor de integración continua que se pueda integrar con dicha herramienta.

Se automatizan proyectos de distintos lenguajes de programación (C, Go, Java, Ruby,…) y se usan distintas herramientas (pipeline, Docker, docker-compose,…) por lo que se escogerá aquel servidor que soporte el mayor número de tecnologías, teniendo el menor coste posible.

	Jenkins	**Travis**	**TeamCity**	**Go**
Open Source	Si	A medias	No	Si
Plugins	Muchísimos	Pocos	Si	Si
Integración con Gitlab	Si	No	Si	Si

Tabla 2-5. Comparación de herramientas

Travis es eliminado de la lista al no poder integrarse con Gitlab, TeamCity no es Open Source y Jenkins posee muchísimos más plugins que Go, por lo que se puede integrar con un mayor número de herramienta y tecnologías. Es por ello, que Jenkins es el mejor candidato para realizar el proceso de Integración Continua en este caso.

2.4.6 Instalación de Jenkins

Como se ha comentado anteriormente, la instalación de Jenkins es un proceso bastante sencillo, ya que solo habría que desplegar un fichero WAR.

```
>> java –jar Jenkins.war
```

La instalación de Jenkins conllevará a lo siguiente:

- El servicio de Jenkins se pondrá en funcionamiento al arrancar el sistema.
- El usuario "Jenkins" será creado para ejecutar dicho servicio.
- Se creará el homedir para el usuario "Jenkins" en la carpeta /var/lib/1enkins donde se encontrará el archivo de configuración del servicio (config.xml).
- El log de Jenkins podrá encontrarse en la ruta /var/log/Jenkins.
- Por defecto Jenkins escucha en el puerto 8080.

Se puede cambiar el puerto de escucha de Jenkins de dos maneras:

1. Ejecutando el comando anterior con la opción *–httpPort=8089* por ejemplo.
2. Modificando el archivo de configuración del servicio.

Si se quiere cambiar el puerto de escucha de Jenkins al 80, no valdría lo anteriormente expuesto, ya que es un puerto privado. Para ello existen otras alternativas:

1. Usando reglas iptables por ejemplo, indicando al Kernell donde está instalado Jenkins, que las peticiones que le lleguen a dicho servidor por el puerto 80, las redirija al puerto 8080:

   ```
   #Requests from outside
   >> iptables –t nat –A PREROUTING –p tcp –dport 80 –j REDIRECT –to-ports 8080

   #Requests from localhost
   >>iptables –t nat –I OUTPUT –p tcp –d 127.0.0.1 –dport 80 –j REDIRECT –to-ports 8080
   ```

2. Instalando y configurando un proxy Apache o Nginx.

2.4.7 Plugins de Jenkins

Como se ha comentado ya, existe una gran cantidad de plugins (más de 800), y en este apartado se presentarán algunos plugins que se han usado en este proyecto y aquellos que son más usados por la comunidad.

Algunos de los plugins más interesantes usados en este proyecto son los siguientes:

- *Gitlab Plugin*: Gitlab es el sistema de control de versiones que se usará en este proyecto, por lo que será necesario el plugin que pueda integrar Jenkins con Gitlab.
- *Pipeline Plugin*: Gracias a este plugin Jenkins puede automatizar tareas en forma de código.

- *Cloudbees Docker Pipeline Plugin*: Permite usar la tecnología de los dockers en el pipeline. En apartados posteriores se explicará la importancia que tiene dicha herramienta en la integración continua.
- Swarm Plugin: Plugin para generar máquinas esclavas de Jenkins, en las cuales se desplegará los proyectos. Se detallará en apartados posteriores.
- *Maven Plugin*: Permite a Jenkins utilizar la herramienta de construcción maven para la compilación de proyectos Java. El uso de esta herramienta también se explicará en secciones posteriores.
- *ANSIColor*: Con este plugin puedes personalizar la interfaz gráfica de Jenkins. Las siguientes imágenes muestran el aspecto de Jenkins antes y después de usar este plugin:

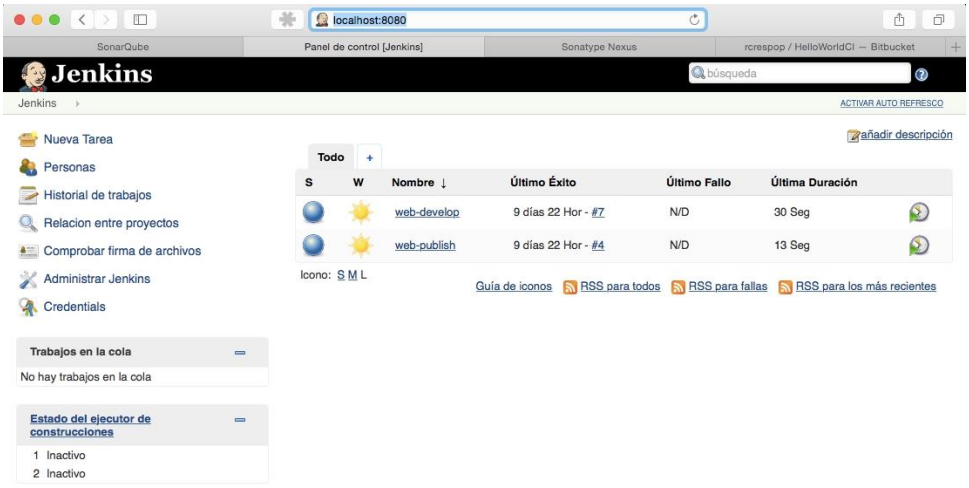


Figura 2-6. Interfaz gráfica de Jenkins por defecto

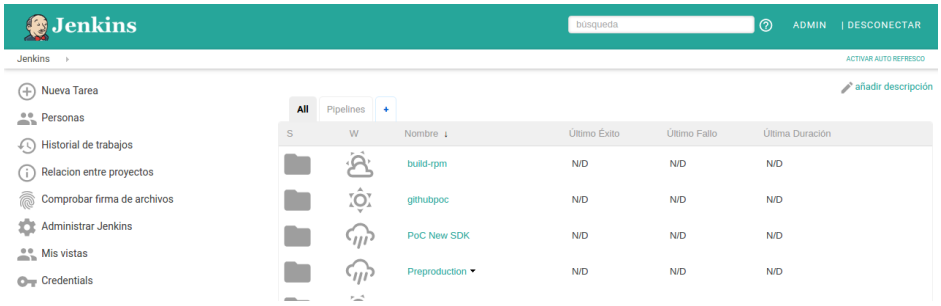


Figura 2-7. Interfaz de Jenkins personalizada

Los plugins que se nombran a continuación, son aquellos más usados por la comunidad en mayo de 2016:

- *Javadoc (127.784 instalaciones)*: Este plugin añade soporte Javadoc a Jenkins.
- *Mailer (127.725 instalaciones)*: Permite configurar las notificaciones de correo electrónico con los resultados de las builds. Para ello se debe configurar el servidor de correo. Si se ha configurado, Jenkins enviará un correo electrónico a los destinatarios especificados cuando se produce un cierto acontecimiento importante.

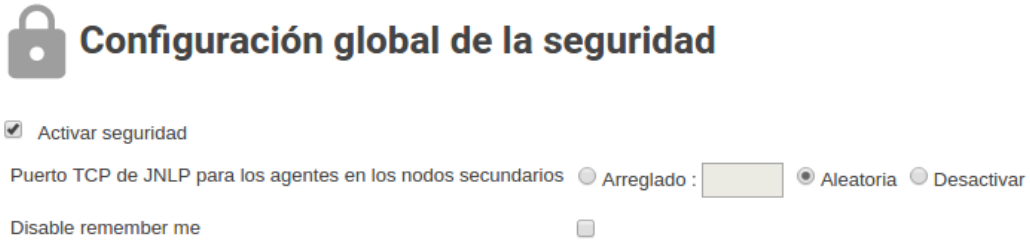
- *Credentials (127.086 instalaciones)*: Este plugin permite almacenar credenciales en Jenkins. Proporciona una API estándar para otros plugins para almacenar y recuperar diferentes tipos de credenciales.
- *SSH-Slaves (126.980 instalaciones)*: Este plugin permite administrar los esclavos a través de SSH. Añade un nuevo tipo de métodos de lanzamientos de esclavos.

2.4.8 Primeros pasos con Jenkins

Antes de empezar a usar Jenkins, debe configurarse conforme a tus requisitos, es decir, debes configurar variables globales, configurar la seguridad, configurar plugins y las herramientas a usar, etc…

Para realizar dicha configuración se debe seleccionar la opción del menú (véase figura 2-7) *"Administrar Jenkins"* y dentro de éste aparecerán varias opciones a poder realizar. Las opciones más importantes son las siguientes:

1. *Configuración del sistema*: Se configuran variables globales y rutas, como por ejemplo la ruta del donde se encuentra alojado el servidor Gitlab o para configura la automatización de envío E-mail.
2. *Configuración global de la seguridad*: Se configura la seguridad en Jenkins. Define quien tiene acceso al sistema (autenticación) y qué puede hacer (autorización). JNLP es el protocolo de comunicación que usa Jenkins con sus esclavos, por lo que si vas a usar esclavos debes activar esta opción.



3. Figura 2-8. Activación de puerto TCP de JNLP
4.

En esta misma opción podrás configurar los permisos de cada usuario. Jenkins dispone de una matriz propia para administrar permisos, y se muestra en la siguiente imagen:

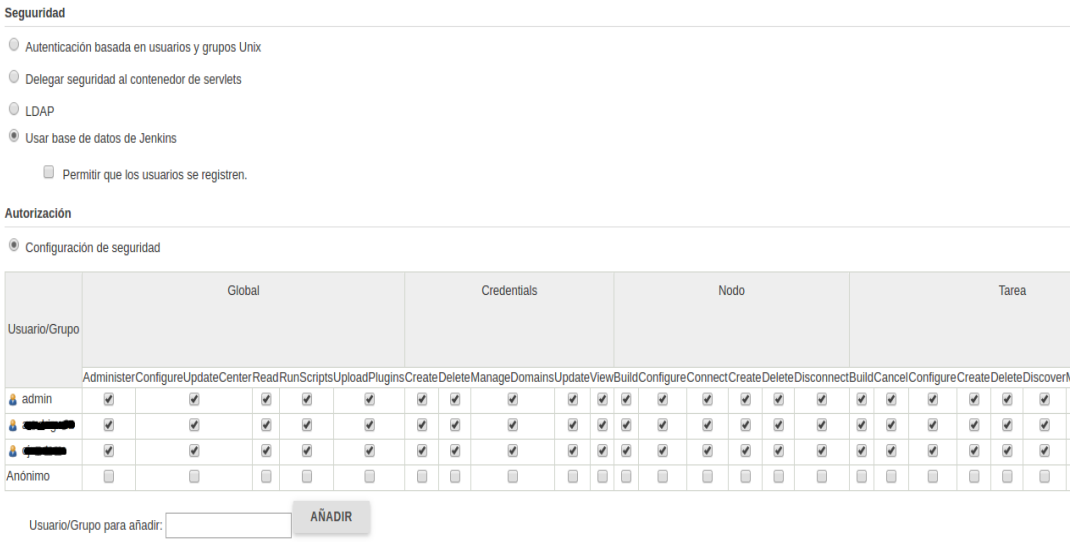


Figura 2-9. Administración de permisos

3. *Configuración Global de herramientas*: Se debe configurar Jenkins para que localice las rutas de instalación de los distintos componentes que forman parte de nuestro entorno de integración continua. Por ejemplo, para localizar el instalador de Maven se pone la ruta donde se encuentra el instalador de Maven:

Figura 2-10. Configuración localización de Maven

Y lo mismo sucede si queremos usar por ejemplo JDK:

Figura 2-11. Configuración localización JDK

4. *Administrar plugins*: En esta sección podrás añadir, borrar, desactivar y activar plugins que extienden las funcionalidades de Jenkins.

5. *System Log*. El Log del sistema captura la salida de la clase java.util.logging en todo lo relacionado con Jenkins.

Una vez realizado las configuraciones iniciales, Jenkins estará listo para poder ser usado. Si nuevamente en el menú inicial de Jenkins (figura 2-7) se selecciona la opción Nueva Tarea, aparecerán los tipos de configuraciones builds que puedes realizar. Algunas opciones son: Proyecto libre, Maven, Pipeline,… donde en este caso se ha trabajado siempre con la opción pipeline. Al seleccionar la opción *"pipeline"* se muestran unos formularios webs para poder configurar dicha build. Puedes realizar las siguientes configuraciones.

- Le indicas donde se encuentra el sistema de control de versiones, que al haberse configurado inicialmente en la sección de *"Configuración del Sistema"*, el campo "*GitLab connection*" se rellena automáticamente:

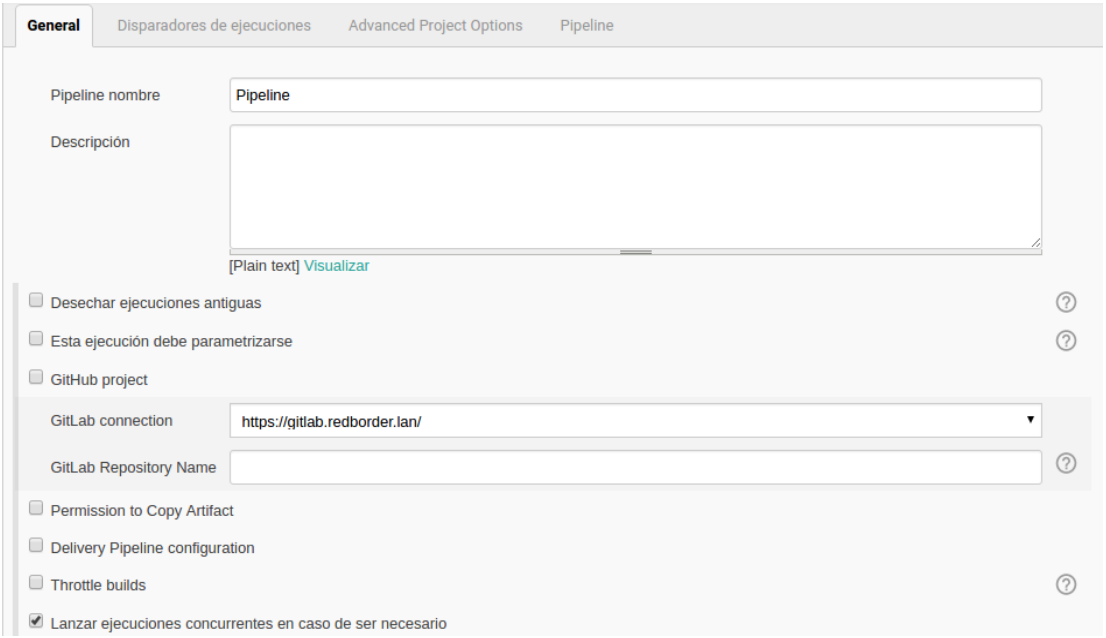


Figura 2-12. Configuración de nueva tarea (I)

- En cada tarea de Jenkins, se puede indicar cuando disparar las ejecuciones automáticas, es decir, puedes disparar las ejecuciones tras un periodo de tiempo constante o cuando se produzca algún cambio en el código fuente del repositorio. En este caso se ha se le ha indicado a Jenkins que se disparen las ejecuciones cuando se produzca un cambio en el repositorio:

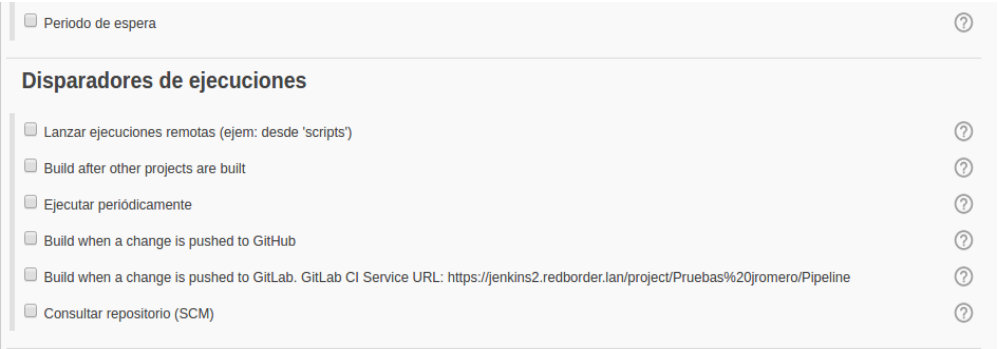


Figura 2-13.Configuración de nueva tarea (II)

Por último faltaría programar el *pipeline* para que Jenkins pueda automatizar las tareas deseadas. Jenkins tiene dos maneras de usar el pipeline:

1. En la misma interfaz de Jenkins.

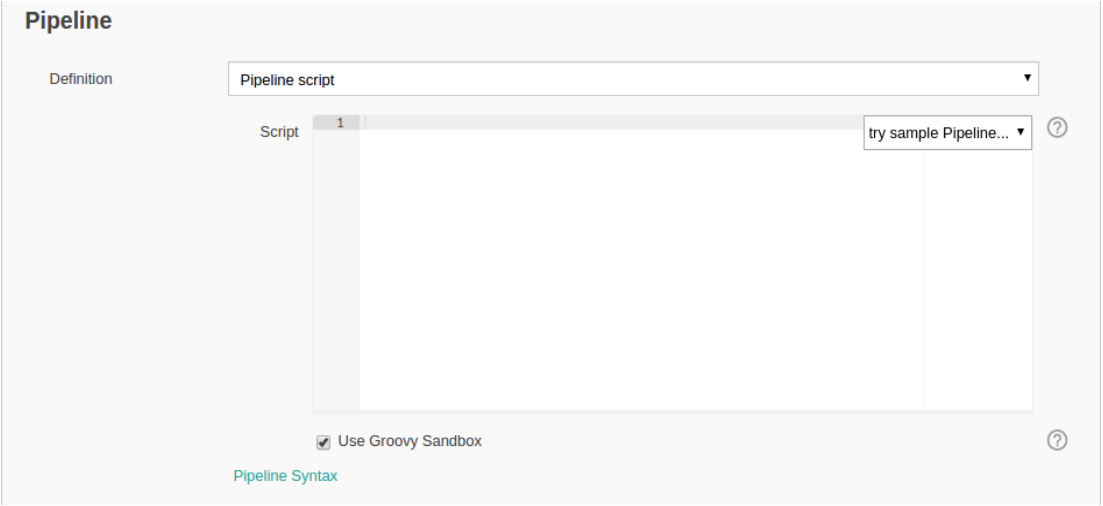


Figura 2-14. Configuración de nueva tarea (III)

2. En el directorio raíz de proyecto que se quiere automatizar. Se le indica a Jenkins el SCM donde está alojado el proyecto, así como la ruta del pipeline. Posteriormente, debes indicar la URL donde se aloja el proyecto y unas credenciales para establecer una conexión segura. En el siguiente apartado se profundizará sobre esta integración.

Figura 2-15. Configuración de nueva tarea (IV)

3 ESTRUCTURA DEL SISTEMA

Gitlab es el sistema de control de versiones donde se almacenará el código fuente de la aplicación que se quiere automatizar. Para poder llevar un control del código fuente se hace necesaria la integración de Jenkins y Gitlab, de manera que Gitlab notifique a Jenkins de cualquier evento producido y Jenkins se descargue la última versión de la aplicación que se quiere automatizar. En secciones posteriores se detallará la integración de ambas herramientas.

Los desarrolladores pueden realizar ciertas acciones sobre el sistema de control de versiones (push, merge, comment,…) y dicha herramienta puede ser configurada para que notifique a Jenkins según tus requisitos.

En este caso, Gitlab avisará a Jenkins mediante **Web Hook**[1] cada vez que se produzca un cambio en el repositorio y, posteriormente, Jenkins desplegará los proyectos en máquinas esclavas, las cuales se encargarán de realizar las compilaciones y pruebas descritas en el pipeline.

La siguiente imagen muestra la estructura del sistema implantado:

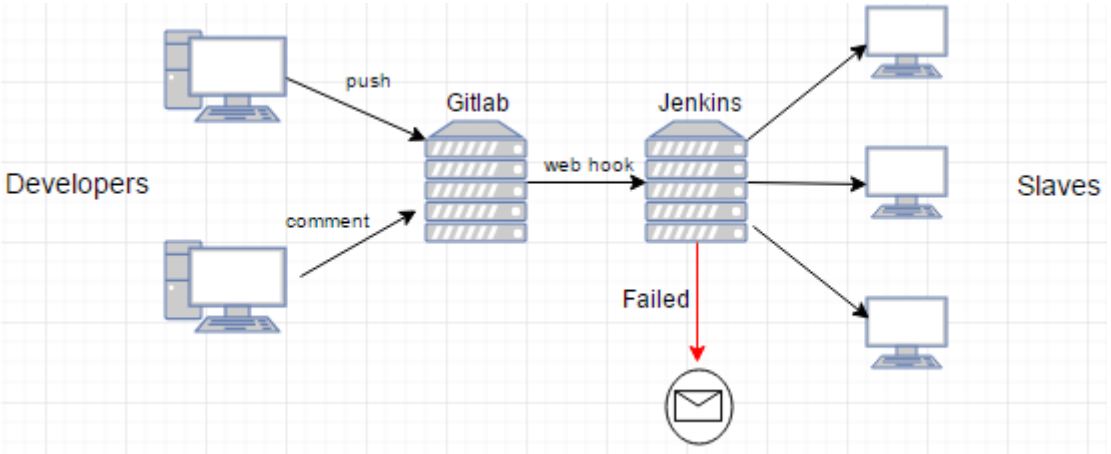


Figura 3-1. Estructura del sistema

Ante cualquier notificación, Jenkins informará a los desarrolladores. Podrá informarlos en la misma interfaz gráfica del servidor de Jenkins o a través de cualquier servidor de mensajería instantánea.

En este caso, se enviará un correo electrónico a los desarrolladores que formen parte del proyecto cuando se produzca alguna incidencia.

[1] Método de alteración del funcionamiento de una página o aplicación web, con callbacks personalizados.

En el siguiente apartado se hablará de los nodos encargados de realizar el trabajo de compilación y pruebas, así como las herramientas y plugins usados para el despliegue de dichas tareas.

3.1 Despliegue de tareas en entornos de pruebas

Para ofrecer una mayor escalabilidad, es normal dejar apartado a Jenkins en un único servidor y crear esclavos del mismo, con el objetivo de desplegar y realizar todo el trabajo en dichos esclavos. De manera que, cuantos más esclavos se tengan, más trabajos se podrán realizar en paralelo.

La generación y configuración de esclavos[2] puede realizarse mediante la interfaz web del propio Jenkins, sin embargo, existe un plugin para generar y configurar esclavos dinámicamente denominado **Swarm Plugin**. El uso de este plugin es bastante sencillo, ya que es un .jar que se ejecuta en el nodo que queremos convertir en esclavo, indicándole los parámetros que se crean más convenientes:

```
java -jar swarm-client-jar-with-dependencies.jar parameters
```

Donde la opción *"parameters"* puede ser cualquiera de las siguientes opciones:

```
-autoDiscoveryAddress VAL: Use this address for udp-based auto-discovery
                           (default 255.255.255.255)
-candidateTag VAL        : Show swarm candidate with tag only
-deleteExistingClients   : Deletes any existing node with the same name.
-description VAL         : Description to be put on the slave
-disableClientsUniqueId  : Disables Clients unique ID.
-disableSslVerification  : Disables SSL verification in the HttpClient.
-executors N             : Number of executors
-fsroot FILE             : Directory where Jenkins places files
-help (--help)           : Show the help screen
-labels VAL              : Whitespace-separated list of labels to be
                           assigned for this slave. Multiple options are
                           allowed.
-master VAL              : The complete target Jenkins URL like
                           'http://server:8080/jenkins/'. If this option
                            is specified, auto-discovery will be skipped
-mode MODE               : The mode controlling how Jenkins allocates
                           jobs to slaves. Can be either 'normal'
                           (utilize this slave as much as possible) or
                           'exclusive' (leave this machine for tied jobs
                           only). Default is normal.
-name VAL                : Name of the slave
-noRetryAfterConnected   : Do not retry if a successful connection gets
                           closed.
-password VAL            : The Jenkins user password
-passwordEnvVariable VAL : Environment variable that the password is
                           stored in
-retry N                 : Number of retries before giving up. Unlimited
                           if not specified.
-showHostName            : Show hostnames instead of IP address
-t (--toolLocation)      : A tool location to be defined on this slave.
                           It is specified as 'toolName=location'
```

[2] Para la configuración de esclavos, véase Jönsson, Peter. Swarm Plugin. Cloudbees. 2016

```
 -tunnel VAL               : Connect to the specified host and port,
                             instead of connecting directly to Jenkins.
                             Useful when connection to Hudson needs to be
                             tunneled. Can be also HOST: or :PORT, in
                             which case the missing portion will be
                             auto-configured like the default behavior
 -username VAL             : The Jenkins username for authentication
```

En mi caso, he usado el siguiente comando en concreto para la generación y configuración de esclavos:

```
      >>java -jar swarm-client-2.0-jar-with-dependencies.jar -master URL_SERVER_JENKINS
  -username NAME -password PASS -disableSslVerification -mode exclusive -labels TAG -
                                      executors 3
```

Se le deberá indicar la URL del servidor Jenkins (-máster), así como el login y el password con el que te conectas a dicho servidor.

En este caso, también se deshabilita la verificación ssl (-disableSslVerification), se le indica que ejecuten 3 tareas en paralelo como máximo (-executor 3), se etiqueta (-label) y se le indica el modo en que se asignan las tareas o jobs (-mode exclusive), donde se le indica que salga de esta máquina solo para los Jobs vinculados.

Una de las opciones más importantes en la generación de esclavos es el parámetro -label. Esta opción, te permite etiquetar los nodos esclavos que se levanten, con el objetivo de facilitar el despliegue en los mismos. Con la siguiente instrucción en el pipeline, se desplegará y se realizará el trabajo en el nodo con la etiqueta *'TAG'*:

```
node('TAG'){

    ...

}
```

El uso de máquinas esclavas aporta las siguientes ventajas:

- Escalabilidad.
- Fácil despliegue.
- Mayor paralelización de trabajos.

La mayoría de la veces, los esclavos necesitan preparar el entorno en el que se ejecutará la aplicación que se quiere automatizar. Para ello se hará uso de una herramienta denominada Docker.

3.2 Docker

Docker es un proyecto de código abierto desde marzo de 2013, que automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de virtualización a nivel de sistema operativo. Permite empaquetar una aplicación en una unidad estandarizada, que contiene todo lo que necesita para funcionar: código, herramientas, bibliotecas del sistema, etc…

Los dockers corren a partir de una imagen que se encuentra en un repositorio público, privado o en nuestra máquina local. Dicha imagen puede ser modificada añadiendo aquellas herramientas y dependencias que harán falta para realizar la integración continua.

Esta herramienta permite implementar las aplicaciones de forma rápida, fiable y sistemática, en cualquier entorno. Se puede decir que los dockers tienen un funcionamiento parecido a las máquinas virtuales.

3.2.1 Docker VS Máquinas Virtuales

Docker y las máquinas virtuales parecen ser idénticas. Para ver en qué se diferencian se partirá de la siguiente imagen:

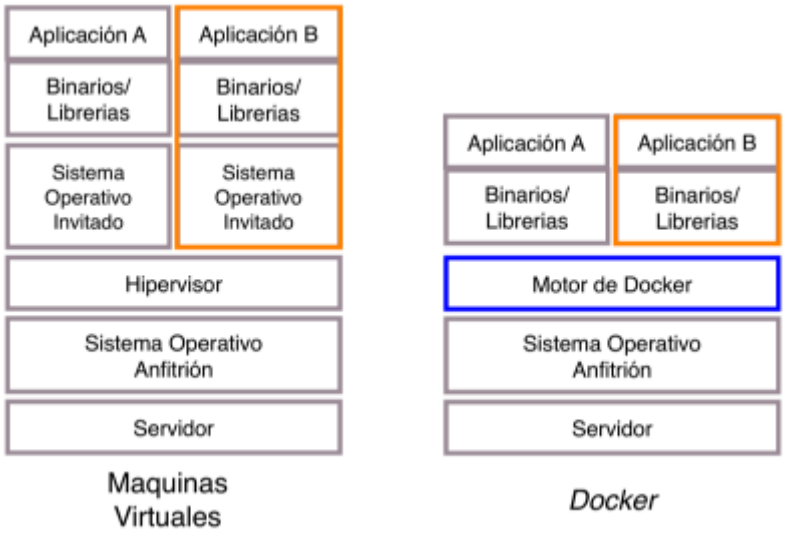


Figura 3-2. Docker VS Máquinas vistuales

En el gráfico, resaltados en naranja, se listan los componentes que se usan para ejecutar una aplicación tanto en máquina virtual como en Docker. Cuando se utilizan máquinas virtuales, es necesario virtualizar todo el sistema operativo, con el gasto de recursos que eso implica, minimizando la ram, disco y procesador que podrá usar la aplicación en sí, en cambio usando Docker, la cantidad de recursos para cada aplicación es mayor al usar directamente los binarios y librerías para ejecutar la aplicación dentro de una imagen.

Las principales ventajas de usar Docker son las siguientes:

- *Rapidez*: Docker y sus contenedores son capaces de compartir un solo núcleo y compartir bibliotecas de aplicaciones, esto ayuda a que los contenedores presenten una carga más baja de sistema que las máquinas virtuales. En comparación con las máquinas virtuales los contenedores pueden ser más rápidos y consumirán menos recursos. Una máquina virtual puede tardar hasta varios minutos para crearse y poner en marcha mientras que un contenedor puede ser creado y lanzados en unos pocos segundos.
- *Portabilidad*: Todas las aplicaciones tienes sus propias dependencias, que incluyen tanto los recursos de software como los de hardware. Docker es un mecanismo que ayuda a aislar las dependencias por cada aplicación mediante la creación de contenedores. Los

contenedores son escalables y seguros si se comparan con el enfoque anterior del uso de máquinas virtuales.

- *Fácil*: Mayor facilidad de automatizar e implantar en entornos de integración continua.
- *Seguridad*: Docker también ofrece grupos de control de acceso al demonio que controla la virtualización, restringiéndolo así sobre posibles cambios que puedan hacerse al sistema. Una lista completa sobre la seguridad Docker la puede encontrar aquí.
- *Administración*: Soluciones tales como Docker hacen más fácil la gestión de contenedores.

En ambos casos, las aplicaciones que se ejecuten dentro de la máquina virtual o de instancia de Docker, están aisladas del sistema operativo anfitrión, pudiendo ejecutarse tantas máquinas virtuales/instancias de contenedores como nos permita nuestro hardware, en el caso de Docker, al usarse menos recursos, es posible tener un mejor rendimiento de aplicaciones por servidor que las máquinas virtuales.

Como inconveniente de los dockers se puede decir que:

- Sólo puede usarse de forma nativa en entornos Unix aunque se puede virtualizar gracias a *"boot2docker"* tanto en OSX como en Windows.
- Las imágenes solo pueden estar basadas en versiones de Linux modernas (kernel 3.8 m
- mínimo).
- Como es relativamente nuevo, es más probable que haya errores de código entre versiones.

3.2.2 Primeros pasos con Docker

En este apartado se mostrará la facilidad que tiene usar los dockers. Se ejecutarán algunos comandos con el objetivo de entender más el funcionamiento de dicha herramienta.

Comentar que docker necesita los permisos de administrador para trabajar y los comandos de docker se muestran a continuación:

```
>> [sudo] docker [opción] [comando] [argumentos]
```

Donde la opción *"comando"* pueden ser las siguientes:

- *attach*: Adjunta a un contenedor corriendo.
- *build*: Construye la imagen que correrá el contenedor.
- *commit*: Crea una nueva imagen de los cambios del contenedor.
- *cp*: Copia archivos/carpetas de los contenedores del sistema de archivos a la ruta de host.
- *diff*: Inspecciona los cambios en el sistema de archivos de un contenedor.
- *events*: Obtiene eventos en tiempo real desde el servidor.
- *export*: Transmite el contenido de un archivo como un archivo tar.
- *history*: Muestra el historial de una imagen.
- *images*: Lista las imágenes.
- *import*: Crea una nueva imagen del sistema de archivos de los contenidos a partir de un archivo tar.

-
- *info*: Muestra el sistema de información por pantalla.
- *insert*: Inserta un archivo en una imagen.
- *inspect*: Regresa información de bajo nivel en un contenedor.
- *kill*: Mata a un contenedor en ejecución.
- *load*: Carga una imagen desde un archivo tar
- *login*: Registra la sesión para el servidor de registro de Docker.
- *logs:* Obtiene los registros de un contenedor.
- *port:* Busca el puerto público el cual está NAT-eado y lo hace privado.
- *ps:* Lista los contenedores.
- *pull:* Descarga una imagen o un repositorio del servidor de registros Docker.
- *push: Empuja una imagen o un repositorio del servidor de registro Docker.*
- *restart: Reinicia un contenedor en ejecución.*
- *rm: elimina uno o más contenedores.*
- *rmi:* Elimina una o más imágenes.
- *run*: Ejecuta un comando en un contenedor.
- *save*: Guarda una imagen en un archivo tar.
- *search:* Busca una imagen en el índice Docker.
- *start:* Inicia un contenedor detenido.
- *stop*: Detiene un contenedor en ejecución.
- *tag*: Etiqueta una imagen en el repositorio.
- *top:* Busca los procesos en ejecución de un contenedor.
- *versión*: Muestra la información de versión Docker.

Esta herramienta posee una gran lista de comandos, de entre los cuales, los más destacados o usados son los siguientes.

Para listar las imágenes disponibles en la máquina local se consigue con el siguiente comando:

```
kike@kike-PC:~$ sudo docker images
[sudo] password for kike:
REPOSITORY                    TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
<none>                        <none>              a2781e5ea59f        10 weeks ago        1.209 GB
ubuntu                        latest              7bd023c8937d        11 weeks ago        122 MB
arrysaa/rubyonrails4.2.4      latest              337e310aa95c        3 months ago        978.1 MB
zhengweiyu/docker-rubyonrails latest              7d45125dd9b5        15 months ago       592.5 MB
jessexu20/rubyonrails         latest              4f785f4dc066        16 months ago       1.119 GB
```

Figura 3-3. Ejemplo comandos docker (I)

Docker Hub es un repositorio de imágenes en el que cualquiera puede contribuir con las suyas. Esto hace que pueda encontrarse herramientas de las más populares y empezar a usarlas en muy pocos minutos. El comando *docker search* te lista las imágenes de dicho repositorio y necesita un argumento para poder ejecutarse, que es lo que deseas buscar.

```
kike@kike-PC:~$ sudo docker search maven
NAME                                DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
maven                               Apache Maven is a software project managem...   221       [OK]
stephenreed/jenkins-java8-maven-git Automated build that provides a continuous...  41                   [OK]
frekele/maven                       docker run --rm --name maven frekele/maven      2                    [OK]
pottava/maven                       Maven images based on Alpine Linux.             1                    [OK]
snowdream/maven                     Docker images for maven                         1                    [OK]
andreptb/maven                      Debian Jessie based image with Apache Mave...   1                    [OK]
stakater/maven                      Maven based on Ubuntu 14.04 and Oracle Java     0                    [OK]
vlatombe/maven-make                 Extends the official maven image with buil...   0                    [OK]
imxieke/maven                       Maven reverse Proxy!                            0                    [OK]
ryantheallmighty/maven-custom       This is a custom Docker image with Maven w...   0                    [OK]
bigtruedata/maven                   Docker image for Java development.              0                    [OK]
dirichlet/maven                     Maven container                                 0                    [OK]
theidledeveloper/maven              Latest maven builds using debian and alpine     0                    [OK]
rodrigosaito/maven                  Maven Docker image                              0                    [OK]
fabric8/maven-builder               Maven builder image used by Java projects ...   0                    [OK]
```

Figura 3-4. Ejemplo comandos docker (II)

Para correr un contenedor y situarse dentro de él con la imagen que desees se muestra a continuación:

```
kike@kike-PC:~$ sudo docker run -it arrysaa/rubyonrails4.2.4  /bin/bash
root@aa52b9fd3496:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@aa52b9fd3496:/# exit
exit
```

Figura 3-5. Ejemplo comandos docker (III)

Este contenedor corre una imagen que tiene instalado ruby on rails. Cuando nos salimos de dicho contenedor, al no haberlo ejecutado en segundo plano, el contenedor es destruido.

Para ejecutarlo en segundo plano basta con añadirle la opción –d:

```
kike@kike-PC:~$ sudo docker run -d -it arrysaa/rubyonrails4.2.4 /bin/bash
72f614a74a65e67027f1213b93b1b5dcb0fc40291d94aecdd3fb38a3cc9f8450
kike@kike-PC:~$ sudo docker ps
CONTAINER ID        IMAGE                          COMMAND             CREATED             STATUS              PORTS               NAMES
72f614a74a65        arrysaa/rubyonrails4.2.4:latest   "/bin/bash"         13 seconds ago      Up 10 seconds                           boring_ar
```

Figura 3-6. Ejemplo comandos docker (IV)

Como puede observarse, al ejecutar el comando *docker ps* nos muestra la información del contenedor levantado.

Para poder acceder al contenedor como antes se hizo (sin la opción -d) se ejecutará lo siguiente:

```
kike@kike-PC:~$ sudo docker ps
CONTAINER ID        IMAGE                          COMMAND             CREATED             STATUS              PORTS               NAMES
916c68149892        arrysaa/rubyonrails4.2.4:latest   "/bin/bash"         24 seconds ago      Up 23 seconds                           modest_fe
rmat
kike@kike-PC:~$ sudo docker exec -it 916c68149892 bash
root@916c68149892:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@916c68149892:/#
```

Figura 3-7. Ejemplo comandos docker (V)

Se ejecuta el comando *docker exec* con el identificador del contenedor que se quiere acceder.

Por último se verá cómo se eliminarán aquellos contenedores que se han levantado en segundo plano. Es necesario parar los contenedores primero y después borrarlos:

```
kike@kike-PC:~$ sudo docker ps
CONTAINER ID        IMAGE                          COMMAND             CREATED             STATUS              PORTS               NAMES
916c68149892        arrysaa/rubyonrails4.2.4:latest   "/bin/bash"         6 minutes ago       Up 6 minutes                            modest_fe
rmat
kike@kike-PC:~$ sudo docker stop 916c68149892
916c68149892
kike@kike-PC:~$ sudo docker rm 916c68149892
916c68149892
kike@kike-PC:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
kike@kike-PC:~$
```

Figura 3-8. Ejemplo comandos docker (VI)

Como se ha comentado con anterioridad, esta herramienta permite crear nuevas imágenes y correr los contenedores sobre dichas imágenes creadas, para ello se usa un fichero denominado *Dockerfile*.

3.2.3 Dockerfile

Un Dockerfile es un fichero de texto donde se indica los comandos a ejecutar sobre una imagen base para crear una nueva imagen. El comando *docker build* construye la nueva imagen leyendo

las instrucciones de dicho fichero con el objetivo de preparar el entorno del que depende el software, es decir, se instalarán en la nueva imagen aquellas dependencias y herramientas de las que depende el software.

A continuación se mostrará las instrucciones que admite el Dockerfile:

- *FROM*: Lo primero que hay que hacer es elegir una imagen base de la que se partirá para posteriormente añadirle las herramientas y dependencias necesarias. Buscará si la imagen se encuentra localmente, si no la descargará de internet.

```
FROM <imagen>:<tag>
```

- *MAINTAINER*: Referencia al autor que genera la imagen.

```
MAINTAINER <nombre> <correo>
```

- *RUN*: Permite correr cualquier comando en la imagen base. Tiene dos formatos:
  - *En modo Shell*:

```
RUN comando
```

  - *En modo ejecución:*

```
RUN ["ejecutable", "parámetro1", "parámetro2"]
```

- *ADD*: Esta instrucción permite añadir un archivo o directorio en la imagen. Tiene dos formas:

```
ADD <src> <dest>

ADD ["<src>" "<dest>"]
```

  Ojo, sólo pueden copiar archivos que se encuentran en paralelo al directorio en el que se encuentra el Dockerfile.

- *ENV*: Esta instrucción configura las variables de ambiente para la nueva imagen.

```
ENV <key>=<value>
```

- *EXPOSE*: Esta instrucción permite exponer por defecto un puerto del contenedor, ya sea entre el local o entre otros contenedores al momento de arrancar.

```
EXPOSE <puerto1> <puerto2>
```

- *VOLUME*: Permite crear un punto de montura en un directorio especificado y esto permite compartir dicho punto de montura con otros contenedores y la máquina anfitrión.

```
VOLUME ["ruta"]
```

- *WORKDIR*: Indica el directorio en el que se situará al arrancar el contenedor.

```
WORKDIR "ruta"
```

- *CMD*: Esta instrucción es de cierta manera parecida a RUN, con la diferencia que esta instrucción no se ejecuta con el comando *docker build*, si no cuando se inicia el contenedor. Tiene tres formatos:
    - *Formato de ejecución:*

```
CMD ["ejecutable", "parametro1", "parametro2"]
```

    - *Modo Shell:*

```
CMD comando paramtro1 parametro2
```

Una vez diseñado tu fichero Dockerfile se pasa a la generación de la nueva imagen. Como se ha comentado, se genera con el comando docker build, cuyo uso viene descrito a continuación:

```
Usage: docker build [OPTIONS] PATH | URL | -
Build an image from a Dockerfile
Options:
      --build-arg value       Set build-time variables (default [])
      --cgroup-parent string  Optional parent cgroup for the container
      --cpu-period int        Limit the CPU CFS (Completely Fair Scheduler)
                              period
      --cpu-quota int         Limit the CPU CFS (Completely Fair Scheduler)
                              quota
      -c, --cpu-shares int    CPU shares (relative weight)
      --cpuset-cpus string    CPUs in which to allow execution (0-3, 0,1)
      --cpuset-mems string    MEMs in which to allow execution (0-3, 0,1)
      --disable-content-trust Skip image verification (default true)
      -f, --file string       Name of the Dockerfile (Default is
                              'PATH/Dockerfile')
      --force-rm              Always remove intermediate containers
      --help                  Print usage
      --isolation string      Container isolation technology
      --label value           Set metadata for an image (default [])
      -m, --memory string     Memory limit
      --memory-swap string    Swap limit equal to memory plus swap: '-1' to
                              enable unlimited swap
      --no-cache              Do not use cache when building the image
      --pull                  Always attempt to pull a newer version of the
                              image
      -q, --quiet             Suppress the build output and print image ID on
                              success
      --rm                    Remove intermediate containers after a
                              successful build (default true)
      --shm-size string       Size of /dev/shm, default value is 64MB.The
                              format is `<number><unit>`. `number` must be
                              greater than `0`. Unit is optional and can be
                              `b` (bytes), `k` (kilobytes), `m` (megabytes),
                              or `g` (gigabytes). If you omit the unit, the
                              system uses bytes.
      -t, --tag value         Name and optionally a tag in the 'name:tag'
                              format (default [])
      --ulimit value          Ulimit options (default [])
```

En mi caso, para la generación de la imagen me ha bastado usar el siguiente comando:

```
>> docker build –t nombreImagen .
```

Decir que el comando anterior se debe ejecutar en el directorio donde se encuentre el Dockerfile y que el argumento –t especifica el nombre de la nueva imagen.

3.3 Docker-compose

La mayoría de las aplicaciones requieren de servicios externos para su correcto funcionamiento. *Docker-compose* es una herramienta que facilitará enormemente la administración de aplicaciones compuestas por varios contenedores relacionadas entre sí. Se define en un archivo de texto los contenedores que forman parte de una aplicación y se podrán iniciar, parar, eliminar o ver sus estados como si de una unidad se tratase.

Con esta herramienta, en lugar de estar levantando todos los contenedores de los que depende dicha aplicación uno a uno, los levanta todos a la vez con un simple fichero de texto y se podrán configurar cada uno de los contenedores conforme a tus requisitos.

En este proyecto, esta herramienta se usa en las pruebas de integración, levantando aquellos servicios de los que depende el proyecto que se está monitorizando y comprobando el correcto funcionamiento del mismo.

Se mostrará un pequeño ejemplo con la finalidad de que se entienda mejor el uso de esta herramienta. Se creará un fichero de texto llamado *docker-compose* con formato *yaml* y el objetivo consiste en compilar y ejecutar las pruebas de un proyecto web programado en ruby on rails, donde dicho proyecto depende de una base de datos postgres y aerospike.

```
services:
  postgres:
    image: 'postgres:9'     #Imagen en repositorio público
    environment:
      POSTGRES_USER: 'USER'
      POSTGRES_PASSWORD: 'PASSWORD'
      POSTGRES_DB: 'NAME_DB'

  aerospike:
    image: 'aerospike:latest' #Imagen en repositorio público

  rubyonrails:
    image: 'ruby-manager' #Imagen creada a partir del Dockerfile
    links:
      - aerospike
      - postgres
```

Primero, se define el servicio de postgres, el cual corre la imagen denominada postgres:9, y se definen unas variables de ambiente para poder acceder a la base de datos.
Posteriormente, se define el servicio de aerospike.
Por último, se define el contenedor donde se compilarán y se ejecutarán las pruebas del proyecto, conectando dicho contenedor con los otros dos[3].

[3] Para más información acerca del docker-compose, véase Docker Compose. 2016

De esta manera se podrán levantar los contenedores descritos en el fichero ejecutando un simple comando:

```
>> docker-compose up [-d]
```

La opción –d es para ejecutar los contenedores como demonios, es decir, en segundo plano.

Para parar y borrar los contenedores levantados:

```
>> docker-compose down
```

Esta herramienta dispone de muchas opciones de configuración por servicio (volume, expose, build, command,…) y permite configurar la red en la que estarán dichos servicios. Si no se configura ninguna, se creará una por defecto. En el siguiente apartado se profundizará sobre el uso de docker-compose con otro ejemplo.

4 DISEÑO DEL PROCESO

En este apartado del proyecto se va a describir el diseño realizado del proceso automatizado que se ha descrito en el apartado de integración y entrega continua (poner número apartado), intentando que este proceso se asemeje lo máximo posible al descrito por el pipeline (figura de diagrama de paso de mensajes pipeline).

Se propondrá un ejemplo el cual requiere de seis servicios en total, es decir, sería necesario levantar seis contenedores cada uno con un servicio. Estos servicios son Apache *zookeeper*, *Apache Kafka*, *k2http y n2kafka*.

Apache Zookeeper

Apache Zookeeper es un servicio centralizado para almacenar y gestionar información relativa a la configuración y a la sincronización de aplicaciones distribuidas. Básicamente actúa como punto de coordinación para servicios distribuidos que cooperan entre sí. Además confiere características como la alta disponibilidad y una alta tolerancia ante posibles fallos gracias a la redundancia.

Para poder coordinar esta redundancia a nivel de nodos servidores se tiene el rol de líder. Este es elegido automáticamente por Zookeper. El resto de nodos son una copia de este.

Apache kafka

Apache define Kafka como un servicio particionado y orientado a la replicación cuyo propósito principal es proveer un sistema de cola de mensajes. Es muy utilizado en situaciones en las que es necesario comunicar flujos de información entre varias aplicaciones.

Utiliza una arquitectura de sistema distribuido basándose en los conceptos de publicador y subscriptor. Los mensajes que comprenden la información compartida se categorizan en topics. Los procesos publicadores de estos mensajes se conocen con el nombre productores de topics. Análogamente, los subscriptores son los que consumen estos topics. Además, Kafka se ejecuta en clústeres compuestos de uno o varios servidores de Kafka donde cada uno de ellos se conoce como broker. Para funcionar, Kafka hace uso de un protocolo propio basado en TCP y se apoya en Apache Zookeeper para almacenar el estado de los broker. La siguiente imagen muestra la estructura que tendrá este ejemplo:

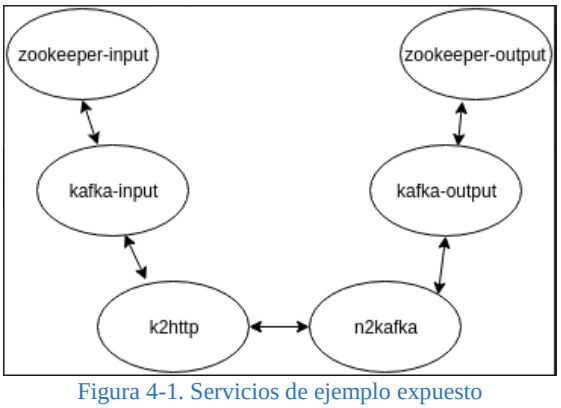


Figura 4-1. Servicios de ejemplo expuesto

K2http y n2kafka

k2http es un servicio consumidor de Kafka-input y la información que reciba se la enviará al servicio n2kafka que es un servicio productor de Kafka-output. K2http envía mensajes recibidos de Kafka por HTTP.

El objetivo que se pretende con este escenario es demostrar que un mensaje enviado desde el Kafka-input sea recibido por Kafka-output, simulando como lo haría de manera real y comprobando si se produce alguna incidencia.

Decir que k2http y n2kafka son proyectos desarrollados por la empresa RedBorder por lo que estos servicios son los que se monitorizarán. Para simplificar la explicación, en este documento, se explicará el proceso diseñado para el proyecto de k2http. N2kafka seguirá el mismo proceso.

4.1 Integración de Jenkins y Gitlab

Para poder integrar Jenkins con Gitlab, lo primero de todo es instalar el plugin de Gitlab. Una vez instalado, al crear una nueva tarea del tipo *"pipeline"* se debe rellenar aquellos campos necesarios que realicen esta integración. En la figura tal (figura apartado anterior), en la selección de SCM, se debe de seleccionar *"Git"* y una vez seleccionada dicha herramienta se debe indicar la URL (formato SSH) del proyecto a automatizar y unas credenciales para establecer una conexión segura con Gitlab:

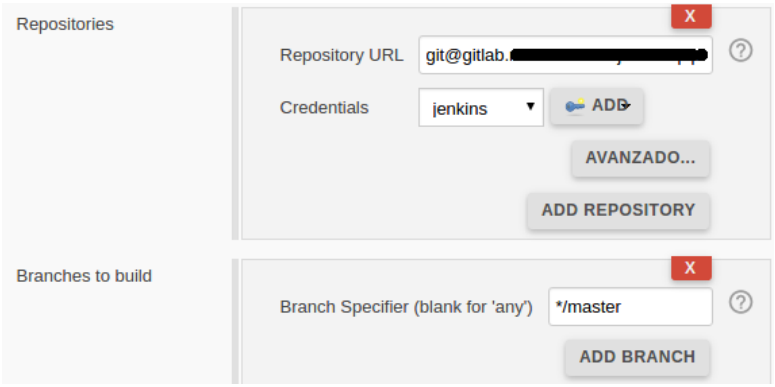


Figura 4-2. Integración Jenkins y Gitlab

Se deberá seleccionar la rama que deseas realizarle la integración continua. Si no se selecciona ninguna, se tomará por defecto la rama master.

Para las credenciales, es necesario crear una clave pública y otra privada en la máquina donde está instalado Jenkins, que se consigue ejecutando el siguiente comando:

```
>> ssh-keygen -t rsa -C "your_email@example.com"
```

Se creará una clave pública y privada del tipo *rsa* en un directorio oculto denominado *".ssh"*.

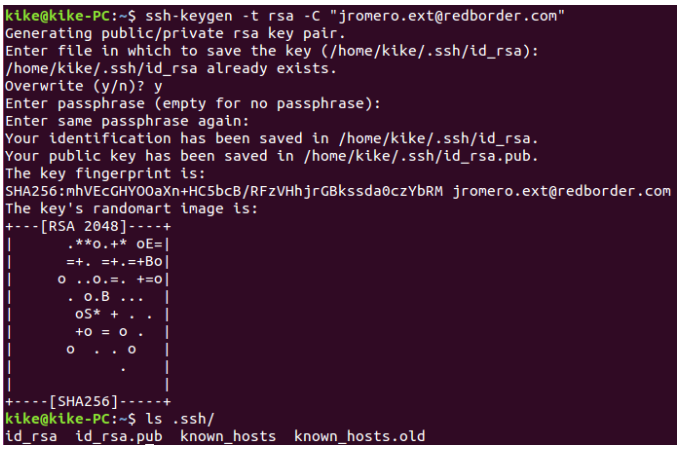


Figura 4-3. Generación clave pública y privada

La clave pública generada (id_rsa.pub) debe copiarse en el servidor Gitlab. En el proyecto que se va a integrar, en la sección de *"Deploy Keys"*, se almacenará dicha clave con el objetivo de que Jenkins asocie la clave pública con la privada y tener así una conexión segura. La Deploy es solo de lectura y sirve para los Hooks, ya que en el servidor web se puede poner un script automatizado que haga peticiones al servidor Gitlab sin necesitar contraseña, es decir, usando la Key, como sucede en este caso.

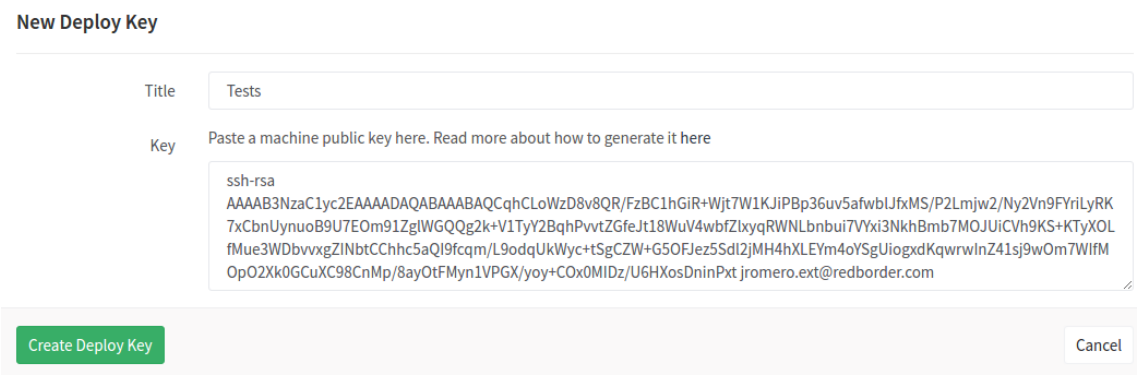


Figura 4-4. Deploy Key de Gitlab

Y a la hora de añadir las credenciales en la interfaz gráfica de Jenkins, debe aparecer lo siguiente:

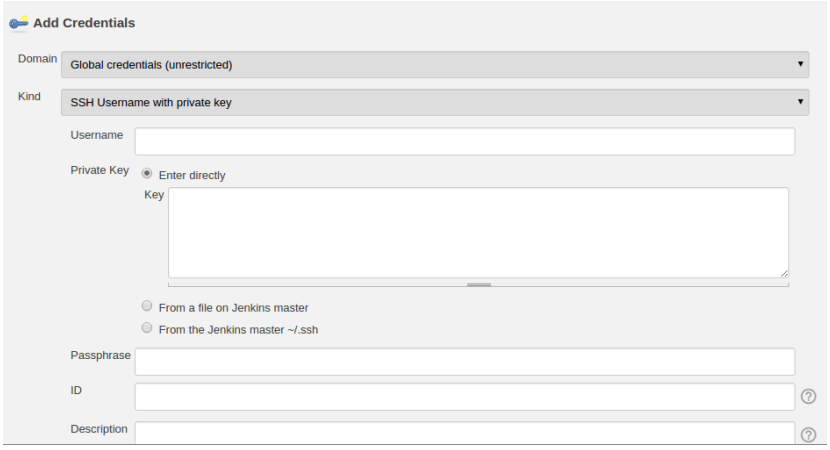


Figura 4-5. Añadir credenciales en Jenkin

Se debe seleccionar el tipo *"SSH Username with private key"* e introducir la clave privada o indicarle la ruta donde se encuentra.

Por último, faltaría configurar el Web Hook de Gitlab para que dicha herramienta notifique a Jenkins según tus requisitos. Se le indica la URL del job de Jenkins que se quiere ejecutar automáticamente, así como el motivo por el que Jenkins será avisado.

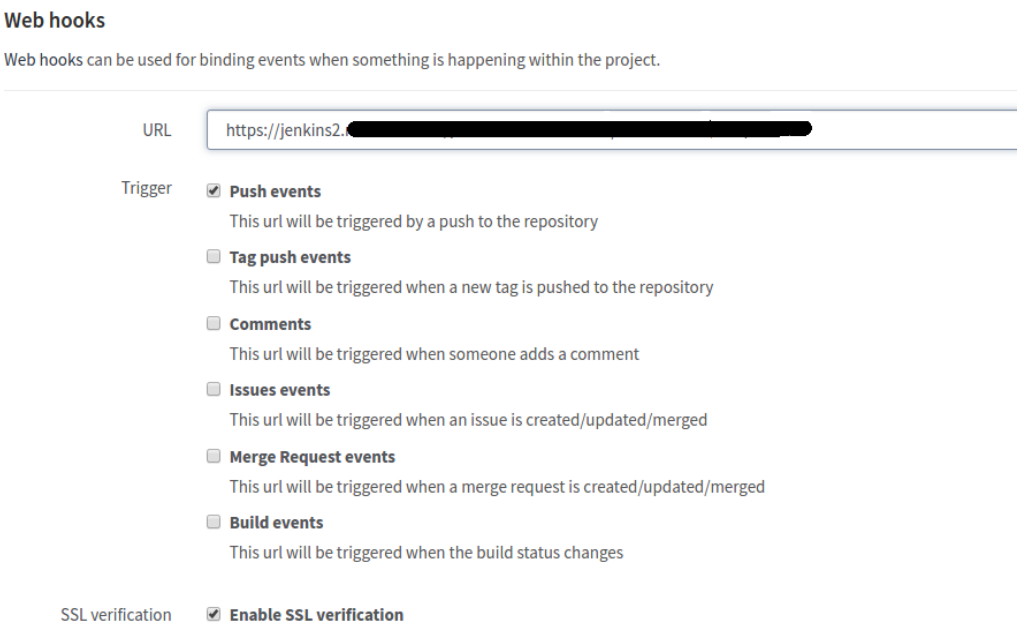


Figura 4-6. Configuración Web Hook de Gitlab

Una vez realizado todo lo anterior, la integración de Jenkins con Gitlab estaría completada.

4.2 Automatización de envío de E-mail

Como se ha mencionado con anterioridad, Jenkins puede ser configurado para que envíe un correo electrónico a aquellas personas que formen parte del proyecto que se está automatizando, para que cuando un desarrollador haga un cambio en el código, si éste falla, inmediatamente se le envíe un correo al desarrollador indicándole que ha fallado, teniendo así una mayor realimentación y una reacción más rápida y ágil.

En la sección de *Configuración del Sistema* en Jenkins, se podrá configurar que se envíen correos automáticos según tus requisitos. En este caso, se enviará un correo en caso de fallo.

Bastará con introducir la siguiente configuración:

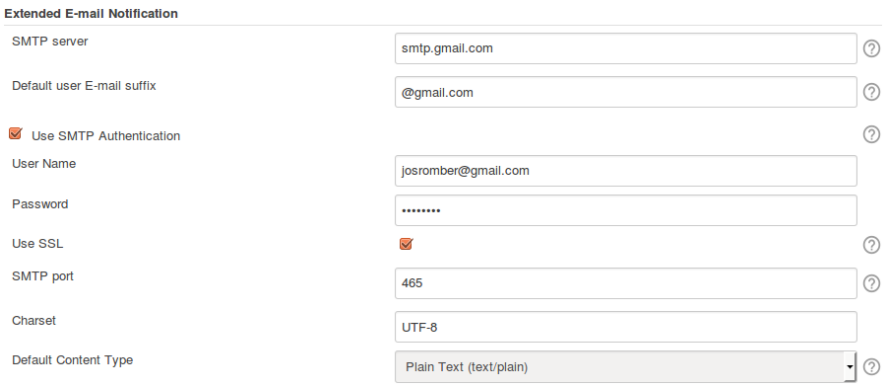


Figura 4-7. Configuración automatización E-mail

Se le indica la persona a la que se le enviará el correo y el SMTP server que se usará, así como el puerto y la contraseña de dicho correo. Ya sólo quedaría indicar cuando se dispararán estos envíos de correos.

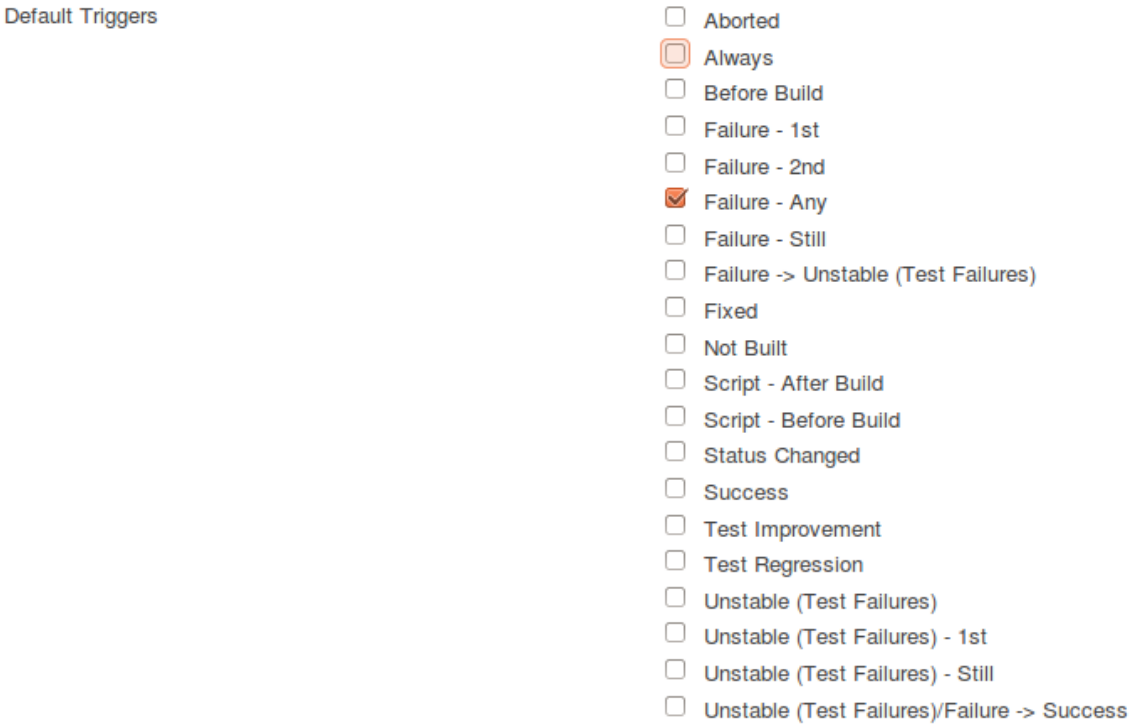


Figura 4-8. Disparadores E-mail

Con esta configuración, cada vez que una build falle, se le enviará un correo electrónico a la persona indicada.

4.3 Fases del proceso

Como se ha descrito con anterioridad, el pipeline es el flujo de tareas por el que transcurre un proyecto en la integración continua. Dichas tareas se definen en forma de código (groovy) y podrán abarcar tareas de tests, builds y despliegues. Con build se refiere a la compilación más la generación de ejecutables.

La automatización de tareas que se pretende conseguir con los pipelines viene representada en la siguiente figura:

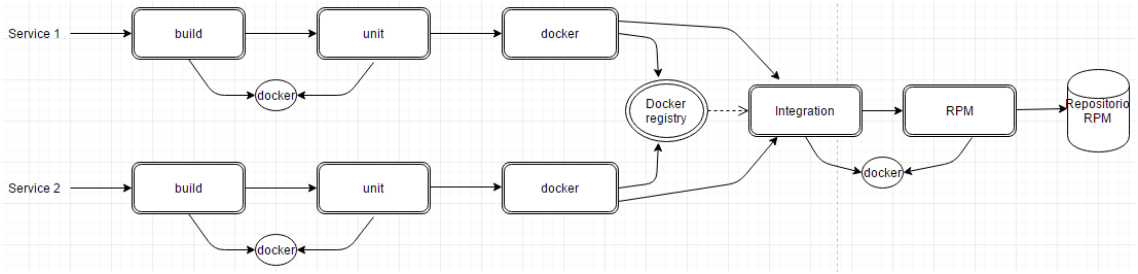


Figura 4-9. Fases del proceso

Cuando se produzca un cambio en el código del servicio (k2http en este caso) se automatizarán todos los pasos indicados en la figura. Se empezará realizando la build del proyecto y tests unitarios dentro de un contenedor docker. Si fue pasada con éxito, se genera una nueva imagen docker con el servicio incluido y se almacena en el repositorio público de dockers, para posteriormente poder usar dichas imágenes en las pruebas de integración. Pasadas las pruebas de integración, se pasará a la generación de paquete RPM del servicio y se subirá a un repositorio público.

Para todo este proceso de usarán tres pipelines como lo muestra la siguiente figura:

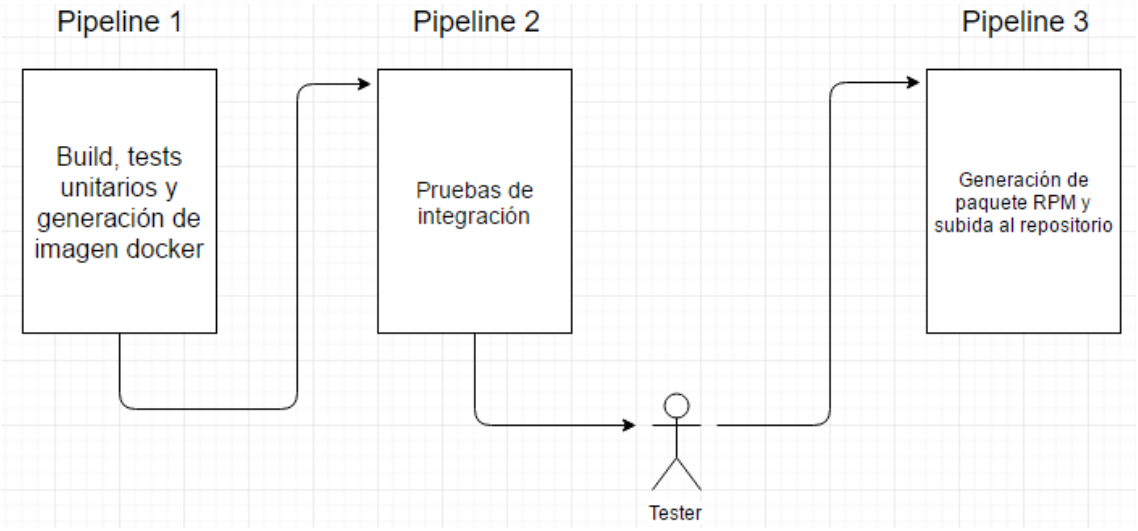


Figura 4-10. Diseño del proceso

El primer pipeline avisará automáticamente al segundo pipeline, es decir, se realizarán tareas de builds, tests, generación de imagen docker y subida a repositorio automáticamente. El segundo pipeline no avisará al tercero automáticamente, el tester o desarrollador deberá comprobar si el servicio actúa como lo esperado. Una vez comprobado el correcto funcionamiento, el tester/desarrollador podrá generar el paquete RPM y subirlo al repositorio público con un simple click.

Recordar que cuando ocurre alguna incidencia, no siguen ejecutándose las tareas, ya que el pipeline se ejecuta de manera secuencial.

4.3.1 Pipeline 1: Build, Tests Unitarios y Generación de Imagen Docker

Como se ha comentado, el primer paso será realizar la build y tests unitarios del servicio usando dockers, es decir, se levantará un contenedor docker con la imagen adecuada y dentro de él, se compilará el proyecto y se ejecutarán los tests unitarios.

Las pruebas unitarias son una forma de comprobar el correcto funcionamiento de un módulo de código. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado.

Una vez se ejecuten las pruebas unitarias, es posible obtener dos resultados:

- Las pruebas dan resultado positivo, en este caso se dispara la ejecución de la etapa de empaquetamiento.
- Las pruebas unitarias fallan, en este caso se notificará los resultados de las pruebas unitarias al desarrollador, lo que permite saber al mismo que su cambio ha producido un error y actuar en consecuencia para resolverlo.

La fase de empaquetado permite generar una aplicación en forma de paquete que contenga todos los ejecutables de la aplicación, todos los recursos necesarios y todas las librerías de las que depende para su ejecución.

Si en los pasos anteriores no hay ninguna incidencia se creará una nueva imagen docker con el servicio k2http y se subirá al repositorio público de imágenes dockers.

La estructura del pipeline 1 para el servicio k2http es el siguiente:

```
//Las tareas se ejecutarán en el nodo cuya etiqueta se "docker"
node ('docker') {


    //Se corre un contenedor y dentro de él se ejecutarán las tareas.
    docker.image('golang:1.6').inside {

        sh '''
          ...         //Clonación de proyectos, comandos que compilan el proyecto
                      // tests unitarios
        '''

    } //Se sale del contenedor y se destruye

      ...   //Se crea y se sube la imagen docker al repositorio

    //Si todo fue correcto, se llama al job que contiene el segundo pipeline
    Build job: '/Shared Jobs/integration k2http', quietPeriod: 0
}
```

El pipeline llama al Dockerfile para construir la imagen docker. A la nueva imagen se le añaden todas la herramientas y dependencias necesarias así como el servicio que se está monitorizando (k2http). Posteriormente se etiqueta dicha imagen y se sube al repositorio público para poder usarla en las pruebas de integración.

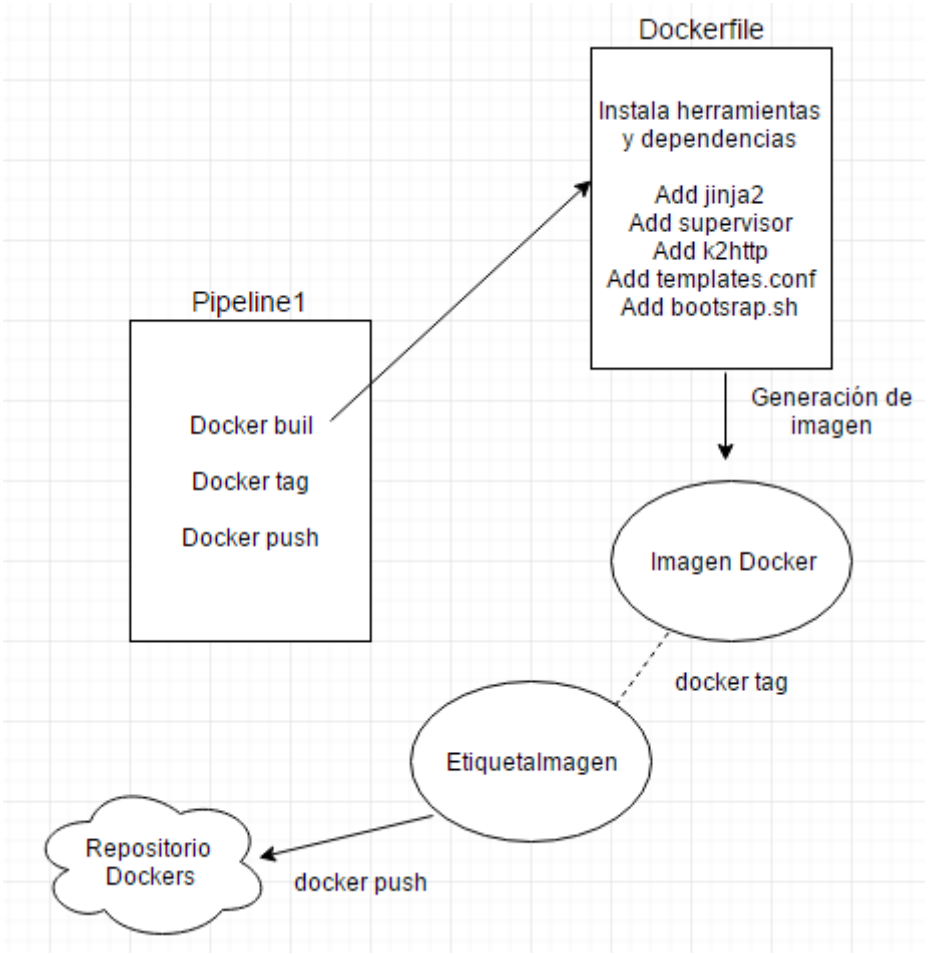


Figura 4-11. Fases pipeline 1

En el Dockerfile se observa que se añaden ciertas herramientas como jinja2 y supervisor, y ciertos ficheros como templates y bootstrap.sh. Jinja2 se usará para poder cambiar la configuración de las plantillas (templates) de configuración, supervisor se encargará de ejecutar el servicio k2http una vez que se arranque la máquina y bootstrap.sh será el script encargado de iniciar el supervisor y ejecutará el comando de jinja2. En el siguiente apartado se detallará el proceso realizado con estas herramientas.

4.3.2 Pipeline 2: Pruebas de integración

Una vez pasado los tests unitarios y la subida de la nueva imagen docker, se pasará a la pruebas de integración. Estas pruebas verifican que los componentes de la aplicación funcionan correctamente actuando en conjunto. En este caso, se encargarán de comprobar el correcto funcionamiento de los distintos servicios enlazados entre sí como viene representado en la

figura 4-1, es decir, la información que se envíe a través del *fafka-input* debe ser recibida por *Kafka-output*.

La estructura del pipeline 2 viene representado a continuación:

```
node ('docker') {

   ...

   sh 'docker-compose up -d'
   sleep 20


   sh '''
      //Se identifican el número de procese de los dos kafka
      KAFKA_INPUT=$(docker ps | grep k2httpn2kafka_kafka-input_1 |
                  awk '{print $1}')
      KAFKA_OUTPUT=$(docker ps | grep k2httpn2kafka_kafka-output_1 |
                  awk '{print $1}')

      //Se usa el servicio kafka dentro de cada contenedor
      docker exec -i $KAFKA_INPUT sh -c "timeout 10 echo '{}' |
                                        /opt/kafka_2.11-0.9.0.1/bin/kafka-
                                        console-producer.sh --topic testing
                                        --broker-list 172.16.238.100:9092"

      docker exec -i $KAFKA_OUTPUT timeout 30 /opt/kafka_2.11-
                                          0.9.0.1/bin/kafka-console-
                                          consumer.sh --topic testing --
                                          zookeeper zookeeper-output:2181 -
                                         -max-messages 1
   '''
}
```

En este caso, el pipeline llama al docker-compose.yml. Como se describió con anterioridad, esta herramienta levantará todos los contenedores que vengan descritos en el fichero, que son seis en este ejemplo. Cada contenedor se enlazará y se configurará conforme a tus requisitos.

El docker-compose se estructura en dos partes, la primera en la que se definen y se configuran todos los servicios y la segunda en la que se define y se configura la red de componentes. Si no se configura ninguna red se creará una por defecto.

```
services:
       service1:
       service2:
            …..

networks:
      network1:
            ….
```

La configuración elegida para los servicios siguiendo con la topología mostrada en la figura 4-1 se muestra a continuación.

- *Zookeeper-input y output:*

```
zookeeper-input:
    image: 'wurstmeister/zookeeper'
    networks:
      - jenkins_network
```

Se le indica la imagen de zookeeper que correrá y la red a la que pertenece. No requiere de más configuración ya que es un servicio usado por Kafka.

- *Kakfa-input y output*:

```
kafka-input:
    image: 'wurstmeister/kafka'
    expose:
        - '9092'
    links:
        - zookeeper-input
    environment:
          KAFKA_ADVERTISED_HOST_NAME: '172.16.238.100'
          KAFKA_ADVERTISED_PORT: '9092'
          KAFKA_ZOOKEEPER_CONNECT: 'zookeeper-input:2181'
          KAFKA_CREATE_TOPICS: 'testing:1:1'
    networks:
          jenkins_network:
             ipv4_address: 172.16.238.100
```

Este servicio requiere de más configuración. Primero se le indica la imagen Kafka que correrá, su puerto de escucha (Kafka escucha por el puerto 9092) y se conecta con zookeeper-input. Se crean variables de ambientes para posteriormente modificar las plantillas de configuración y poder conectarse correctamente a los servicios externos. Por último, se le indica la red de la que forma parte y se le asigna una dirección IP.

La configuración de Kafka-output es igual pero con la IP 172.16.238.101 y conectado con zookeerper-output.

- *K2http:*

```
k2http:
    image: 'resources.redborder.lan:5000/k2http:latest'
    links:
      - n2kafka
      - kafka-input
    environment:
      KAFKA_BROKER: '172.16.238.100:9092'
      KAFKA_CONSUMERGROUP: 'k2http-n2kafka'
      HTTP_URL: 'http://n2kafka:2057/rbdata/abc'
    networks:
      - jenkin
s_network
```

Se le indica que corra la imagen k2http que se creó y se conecta con Kafka-input y n2kafka. Se crean de nuevo variables de ambientes conforme en la plantilla de

configuración (API) para configurar dinámicamente el contenedor de k2http con la herramienta jinja2. Por último, se le indica la red que pertenece.

- *N2kafka:*

```
n2kafka:
    image: 'resources.redborder.lan:5000/n2kafka:latest'
    environment:
      BROKERS: "172.16.238.101:9092"
      TOPIC: "testing"
    links:
      - kafka-output
    expose:
      - '2057'
    networks:
      - jenkins_network
```

Mismo procedimiento que los anteriores. Se configura conforme tus requisitos.

- Networks:

```
networks:
  jenkins_network:
    driver: bridge
    driver_opts:
     com.docker.network.bridge.enable_ip_masquerade:"false"
    ipam:
      driver: default
      config:
      - subnet: 172.16.238.0/24
        gateway: 172.16.238.1
```

Se configura la red conforme tus requisitos. En este caso por ejemplo se deshabilita la opción *ip_masquerade*, se le asigna dirección IP a la red y se define la puerta de enlace.

Como se ha comentado, las variables de entorno se usan para configurar dinámicamente los servicios que lo requieran. El contenedor lee del docker-compose dichas variables de entorno para sustituirlas en su plantilla de configuración. Una de las maneras de modificar la plantilla de configuración es con la herramienta jinja2.

Cuando se inicia el contenedor de k2http, en su Dockerfile se le indicó que cuando arranque el contenedor ejecute el script *bootstrap.sh*. El contenido de este script es el siguiente:

```
#!/bin/bash

j2 /app/config.template > /app/config.yml

supervisord -n
```

Es un simple script que ejecuta la herramienta de jinja2 y el supervisor. El supervisor se encargará de levantar el servicio k2http, si no lo consigue lo vuelve a intentar las veces que haga falta. El contenido del fichero de configuración del supervisor es el siguiente:

```
[program:k2http]
command=/app/k2http --config /app/config.yml --debug
autostart=true
autorestart=true
```

Jinja2 usa la plantilla de configuración *"config.template"*, donde en dicha plantilla se encuentran las variables de entorno que usa el docker-compose.

El contenido del *"config.template es el siguiente":*

```
backend:
  workers: 1
  queue: 100
  retries: 3
  backoff: 5
  showcounter: 5
kafka:
  # begining: true
  broker: {{KAFKA_BROKER}}
  consumergroup: {{KAFKA_CONSUMERGROUP}}
  topics:
    - input
http:
  url: {{HTTP_URL}}
  batchsize: 100
  batchtimeout: 500
  deflate: true
```

Cuando se ejecuta el comando de jinja2, sustituirá las variables de ambientes por el correspondiente valor leído del docker-compose, pudiendo así configurar dinámicamente el contenedor.

Una vez realizado todo este proceso será el pipeline el que se encargue de ejecutar las pruebas de integración. Con el comando *docker exec* se podrá ejecutar comandos dentro del contenedor deseado y se comprobará así su correcto funcionamiento.

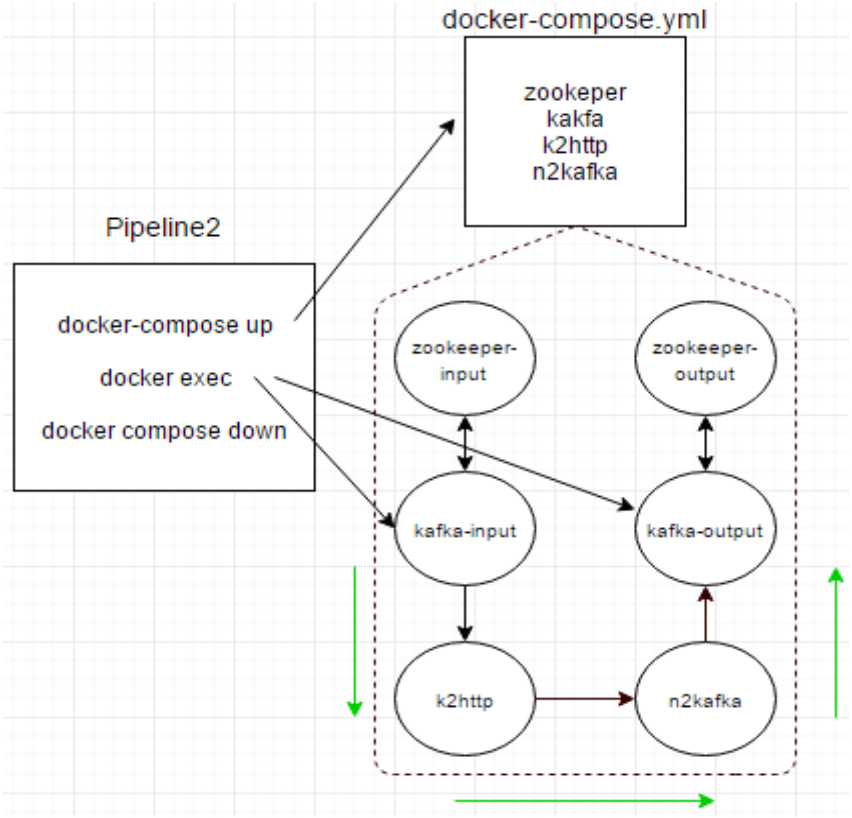


Figura 4-12. Fases pipeline 2

4.3.3 Pipeline 3: Generación de paquete RPM y subida al repositorio

Por último, si el resultado de todas las pruebas anteriores es positivo, se pasará a la generación automática del paquete RPM del servicio correspondiente (k2http). Estos paquetes se almacenarán en un repositorio con el objetivo de poder desplegar el servicio de forma más sencilla. Dichos servicios podrán ser desplegados tanto por los clientes como por la propia organización.

RPM es una herramienta de administración de paquetes pensada básicamente para GNU/Linux. Es capaz de instalar, desinstalar, verificar y solicitar programas. Originalmente desarrollado por Red Hat para Red Hat Linux y en la actualidad existen muchas distribuciones GNU/Linux que lo usan, dentro de las cuales las más destacadas son *Fedora Linux*, *Mageia*, *PCLinuxOS*, *openSUSE*, *SuSE Linux.*

Para el administrador de sistemas que realice mantenimiento y actualización de software, el uso de gestor de paquetes en vez de construirlos manualmente tiene ventajas como simplicidad, consistencia y la capacidad de que aquellos procesos se automaticen.

Entre las características de RPM están:

- Los paquetes pueden ser cifrados y verificados con GPG y MD5.
- Los archivos de código fuente están incluidos en SRPMs, posibilitando una verificación posterior.
- PatchRPMs y DeltaRPMs, que son equivalentes a ficheros parche, pueden actualizar incrementalmente los paquetes RPM instalados.
- Las dependencias pueden ser resueltas automáticamente por el gestor de paquetes.

Para poder comenzar a generar el paquete RPM se deberá crear una estructura de directorios necesaria para la gestión de rpms, código fuente, specs, etc…

Esta estructura de directorios se puede crear a mano o ejecutando el siguiente comando:

```
>> rpmdev-setuptree
```

En mi caso se ha usado la siguiente estructura de directorios:

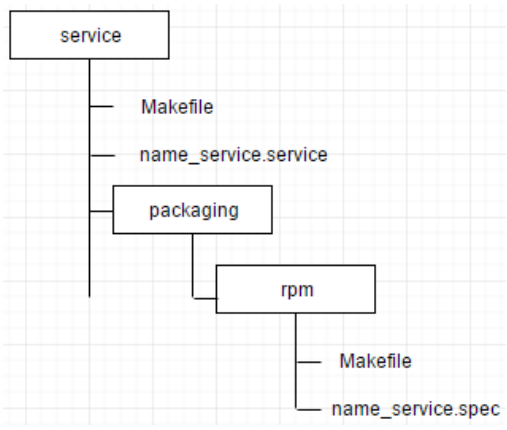


Figura 4-13. Estructura de ficheros para generación paquetes RPM

El fichero *name_service.service* del directorio raíz, es un fichero que se encargará de levantar el servicio. Se configurará para que, en caso de fallo lo vuelva a intentar levantar automáticamente, indicándole el usuario y el grupo que pertenece.

El Makefile del directorio raíz es el Makefile principal del servicio. En dicho Makefile se debe añadir la opción de generar paquetes rpm y se realiza de la siguiente manera:

```
rpm: clean
	$(MAKE) -C packaging/rpm
```

Esta sentencia llamará al *Makefile* del directorio rpm que, junto con su respectivo .spec, realizará lo necesario para generar el paquete rpm.

El Makefile del directorio *rpm* se encargará de obtener el código y preparar el entorno para que se pueda generar el paquete RPM, llamando al fichero .spec.

El fichero que se tiene que crear para poder construir el paquete RPM es el fichero .spec. Este fichero contiene toda la información del paquete rpm: nombre, número de versión, arquitectura, descripción, dependencias, órdenes de ejecución para la instalación,…

No es necesario crearlo desde cero, se puede ejecutar el siguiente comando:

```
>> rpmdev-newspec
```

Este comando genera una plantilla con parámetros la cual debes completar con la correcta información para la generación del paquete rpm. Los parámetros más importantes son:

- *Name:* El nombre del paquete rpm.
- *Version:* Número de versión.
- *Release:* Release de la distribución.
- *Summary:* Simplemente descriptivo, indica el contenido del paquete RPM. No es obligatorio.
- *Group:* Hace referencia al grupo al que pertenece el paquete.
- *License:* Se pueden especificar términos de licencia del paquete, debe ser una licencia de software de código abierto.
- *URL:* La dirección URL completa para obtener más información sobre el programa.
- *Source0:* La dirección URL completa para el archivo comprimido que contiene el código fuente.
- *BuildRequires:* Una lista separada por comas de paquetes requeridos para la construcción (compilación) del programa.
- *Requires:* Una lista separada por comas de los paquetes que se requieren cuando se instala el programa. Hay que tener en cuenta que BuildRequires lista los que se necesita para construir el RPM binario, mientras que la etiqueta Requires lista lo que se requiere para la instalación/ejecución del programa.
- *%description:* Una descripción más detallada.
- *%prep:* Comandos de script para preparar el programa, por ejemplo para descomprimirlo, y que pueda estar listo para la construcción.
- *%build:* Comandos de script para construir el programa, por ejemplo para compilarlo y prepararlo para la instalación.

- *%check:* Comandos de script para probar el programa. Estos se ejecutan entre los procedimientos %build e %install, así que debe colocarlo allí si se tiene este parámetro.
- *%install:* Comandos de script para instalar el programa.
- *%clean:* Instrucciones para limpiar la raíz de construcción.
- *%changelog:* Cambios en el paquete.

Dicho todo esto, ya se podría generar el paquete RPM y subirlo al repositorio. La estructura del pipeline 3 sería la siguiente:

```
node ('docker') {

    ...          //clonación del proyecto

    sh '''
       make rpm
       package_cloud push username/myrepo /path/to/packagefile
    '''

}
```

En este caso, el pipeline 3 llama al Makefile del directorio raíz, y éste a su vez llama al Makefile del directorio 'rpm'. Este último Makefile generará, junto con su respectivo spec, el paquete RPM correspondiente. La siguiente imagen muestra las fases por las que pasa este proceso.

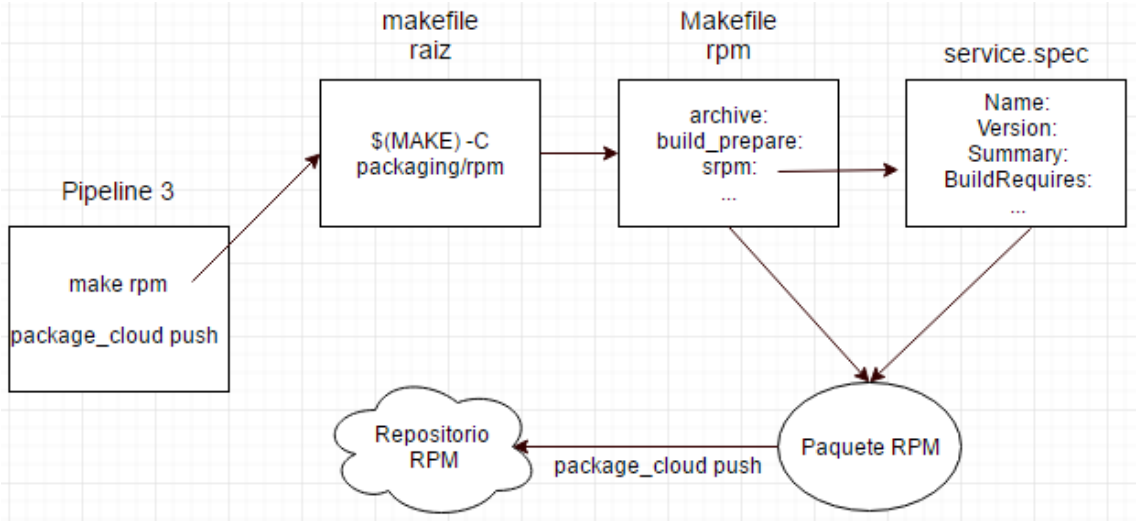


Figura 4-14. Fases pipeline

Una vez que se modifique el código del repositorio, se realizarán todos los pasos descritos de manera automática, realimentando a los desarrolladores de las incidencias ocurridas de una manera más ágil, rápida y eficaz.

5 CONCLUSIONES

Montar un servicio de integración continua puede resultar un poco complicado al principio, pero donde realmente reside la complicación es en estructurar tu proyecto para que todas las tareas sean fácilmente automatizables. Estructurando bien tu proyecto, en muchos casos, podrás reutilizar las técnicas de integración de un proyecto a otro.

Aunque es esfuerzo inicial sea más elevado, personalmente aconsejo montar un servidor de integración continua en las empresas, ya que podrás tener testeado tu proyecto en todo momento, ya sea mediante la pulsación de un botón, cuando se produzca un cambio, cuando se genere el build de un proyecto,… y, a su vez, se obtendrá una mayor realimentación al equipo encargado del proyecto.

Existe una gran diversidad de herramientas de integración continua como Travis, Baamboo, Go,... pero en comparación con Jenkins estas herramientas no disponen de tantos plugins como Jenkins, por lo que a su vez disponen de una menor integración con otras herramientas y SCV.

Por otro lado, se podría decir que los Dockers han revolucionado el mundo de la integración continua, ya que teniendo instalada únicamente dicha herramienta se podrá compilar y construir los ejecutables de cualquier proyecto, así como poder simular un escenario real con múltiples contenedores y ver su funcionamiento.

5.1 Objetivos iniciales y resultados finales

Como se describió en la introducción del trabajo, el objetivo fundamental consiste en demostrar que se puede llevar a cabo una automatización por completo del ciclo de vida de una aplicación, haciendo una integración de herramientas existentes, de manera que se pueda facilitar la implantación de un modelo de entrega continua.

Tras el estudio que se ha detallado en este documento, tanto en la parte del diseño como de pruebas, se ha podido demostrar que se ha cumplido el objetivo propuesto. Ha sido posible poner en contexto el conjunto de herramientas elegidas llevando a cabo un montaje de cada una de las tareas permitiendo procesar una aplicación de ejemplo desde su desarrollo hasta su preparación para la puesta en producción, facilitando además una rápida realimentación de los resultados de pruebas y el estado de la aplicación a todos los miembros de los equipos que intervienen en el proceso de creación software a través del sistema de notificaciones provisto por el servidor de entrega continua.

5.2 Aportación del trabajo a nivel personal

Llevar a cabo este trabajo fin de grado ha resultado ser a la vez una experiencia enriquecedora y un reto, que me ha brindado la oportunidad de poder conocer con profundidad conceptos y filosofías de trabajos claves a la hora de trabajar en un proyecto de desarrollo software de calidad, además de poder poner en práctica, mediante la construcción de un proceso automatizado, como sería posible facilitar la implantación de algunos de estos conceptos.

Además, la integración continua es un proceso en las que no todas las organizaciones lo han implantado. Aunque no soy un experto en este campo, recomiendo que se implante cuanto antes, ya que se podrá monitorizar los proyectos automáticamente y se podrá tener una mayor calidad del producto que se ofrece. En la actualidad, existen diversos puestos de trabajo en este campo, un campo bastante interesante en el que poder crecer personal y profesionalmente, pudiendo formar parte en el desarrollo software de calidad.

6 REFERENCIAS

[1] Docker Compose. 2016 (https://docs.docker.com/compose/)

[2] Docker Pipeline Plugin. Cloudbees. 2016 (https://go.cloudbees.com/docs/cloudbees-documentation/cje-user-guide/chapter-docker-workflow.html#chapter-docker-workflow_docker-workflow)

[3] Hannah Inman. Get Started with Jenkins 2.0 with docker. Cloudbees. 2016 (https://www.cloudbees.com/blog/get-started-jenkins-20-docker)

[4] How to create an RPM package. Red Hat. 2016 (https://fedoraproject.org/wiki/How_to_create_an_RPM_package)

[5] Jönsson, Peter. Swarm Plugin. Cloudbees. 2016 (https://wiki.jenkins-ci.org/display/JENKINS/Swarm+Plugin)

[6] Pipeline as code with Jenkins. Cloudbees. 2016 (https://jenkins.io/solutions/pipeline/)

[7] Apache Kafka.Oficial (http://kafka.apache.org/)

[8] Ronacher, Armin. Welcome to Jinja2. 2008 (http://jinja.pocoo.org/docs/dev/)

[9] Apache Zookeeper ™.Oficial (https://zookeeper.apache.org/)

[10] Integración y Despliegue Continuo. Wordpress. 2014 (https://devopsti.wordpress.com/2014/09/26/integracion-continua-ci-entrega-continua-cd-y-despliegue-continuo-cd/)

[11] Docker VS Máquina Virtual. Andrés Rosales.2016 (https://guiadev.com/docker-vs-maquinas-virtuales-mejor/)

[12] De la Integración Continua a la entrega Continua. Eder Castro Lucas. 2014 (http://atsistemas.com/wp-content/uploads/downloads/2014/04/articulo_integracion_continua_entrega_continua.pdf)

[13] Home. Travis. 2016 (http://www.travisonline.com/)

[14] Home. TeamCity. 2016 (https://www.jetbrains.com/teamcity/)

[15] Home. Go CD.2016 (https://www.go.cd/)

[16] Home. Jenkins. Oficial.2016 (https://jenkins.io/)

[17] Docker security. 2016 (https://docs.docker.com/engine/security/security/)

[18] Home. Docker. Oficial. 2016 (https://www.docker.com/)

ANEXO A: CÓDIGO DEL EJERCICIO EXPUESTO

- *Pipeline 1 de k2http*

```
node ('docker') {
    stage 'Get GO docker image'
    docker.image('golang:1.6').inside {
        stage 'Compilation'
        git branch: 'continuous-integration',
                    credentialsId: 'jenkins_id',
                    url: 'git@gitlab.redborder.lan:core-
                          developers/k2http2.git'

        sh '''
           export GOPATH=/golang
           export PATH=$PATH:$GOPATH/bin
           mkdir -p /golang/src/k2http2/
           mv * .git /golang/src/k2http2/
           cd /golang/src/k2http2/
           make get
           make get_dev
           make
        '''
        sh 'cp /golang/src/k2http2/k2http .'
        stage 'Unit Tests'
        sh '''
           export GOPATH=/golang
           export PATH=$PATH:$GOPATH/bin
           cd /golang/src/k2http2/
           make fmt
           make vet
           make test
           pwd && ls -l
        '''
        stash includes: 'k2http', name: 'k2http-bin'
        archive 'k2http'

    }
    stage 'Generate Docker Image'
    git branch: 'continuous-integration',
                credentialsId: 'jenkins_id',
                url: 'git@gitlab.redborder.lan:core-
                developers/k2http2.git'
    unstash 'k2http-bin'
    sh '''
      docker build -t k2http .
      docker tag k2http
             resources.redborder.lan:5000/k2http:latest
      docker push resources.redborder.lan:5000/k2http:latest
    '''
}
```

- *Pipeline 2 de k2http*

```
node ('docker') {
   git branch: 'feature/docker-compose-support',
               credentialsId: 'jenkins_id',
               url: 'git@gitlab.redborder.lan:jenkins/pipeline-
                     jobs.git'
   dir ('integration-tests/k2http-n2kafka') {
     try {
         /*****************************/
         stage 'Environment preparation'
         /*****************************/
         sh 'docker-compose down'
         sh 'docker-compose up -d'
         sleep 20

         /*****************************/
         stage 'Integration tests'
         /*****************************/
         sh '''
            KAFKA_INPUT=$(docker ps | grep k2httpn2kafka_kafka-
                        input_1 | awk '{print $1}')
            KAFKA_OUTPUT=$(docker ps | grep k2httpn2kafka_kafka-
                        output_1 | awk '{print $1}')

         docker exec -i $KAFKA_INPUT sh -c "timeout 10 echo '{}'
                     | /opt/kafka_2.11-0.9.0.1/bin/kafka-console-
                     producer.sh --topic testing --broker-list
                     172.16.238.100:9092"

         docker exec -i $KAFKA_OUTPUT timeout 30 /opt/kafka_2.11-
                     0.9.0.1/bin/kafka-console-consumer.sh
                     --topic testing --zookeeper zookeeper-
                     output:2181 --max-messages 1
         '''
         } finally {
            /*****************************/
             stage 'Cleaning up'
             /*****************************/
             sh 'docker-compose down'
         }
   }
}
```

- *Dockerfile k2http*

```
FROM Ubuntu

RUN apt-get update && \
    apt-get install -y supervisor && \
    apt-get install -y python-pip && \
    rm -rf /var/lib/apt/lists/* && \
    apt-get clean

RUN pip install j2cli

ADD supervisor/k2http.conf /etc/supervisor/conf.d/

ADD k2http /app/
ADD templates/config.template /app/

ADD bootstrap.sh /app/

CMD ["/app/bootstrap.sh"]
```

- *Docker-compose.yml*

```
version: '2'

services:
  zookeeper-input:
    image: 'wurstmeister/zookeeper'
    networks:
      - jenkins_network

  zookeeper-output:
    image: 'wurstmeister/zookeeper'
    networks:
      - jenkins_network

  kafka-input:
    image: 'wurstmeister/kafka'
    expose:
      - '9092'
    links:
      - zookeeper-input
    environment:
      KAFKA_ADVERTISED_HOST_NAME: '172.16.238.100'
      KAFKA_ADVERTISED_PORT: '9092'
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper-input:2181'
      KAFKA_CREATE_TOPICS: 'testing:1:1'
    networks:
      jenkins_network:
        ipv4_address: 172.16.238.100

  kafka-output:
    image: 'wurstmeister/kafka'
    expose:
```

```
      - '9092'
    links:
      - zookeeper-output

    environment:
      KAFKA_ADVERTISED_HOST_NAME: '172.16.238.101'
      KAFKA_ADVERTISED_PORT: '9092'
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper-output:2181'
      KAFKA_CREATE_TOPICS: 'testing:1:1'
    networks:
      jenkins_network:
        ipv4_address: 172.16.238.101

  n2kafka:
    image: 'resources.redborder.lan:5000/n2kafka:latest'
    environment:
      BROKERS: "172.16.238.101:9092"
      TOPIC: "testing"
    links:
      - kafka-output
    expose:
      - '2057'
    networks:
      - jenkins_network

  k2http:
    image: 'resources.redborder.lan:5000/k2http:latest'
    links:
      - n2kafka
      - kafka-input
    environment:
      KAFKA_BROKER: '172.16.238.100:9092'
      KAFKA_CONSUMERGROUP: 'k2http-n2kafka'
      HTTP_URL: 'http://n2kafka:2057/rbdata/abc'
    networks:
      - jenkins_network

networks:
  jenkins_network:
    driver: bridge
    driver_opts:
      com.docker.network.bridge.enable_ip_masquerade: "false"
    ipam:
      driver: default
      config:
      - subnet: 172.16.238.0/24
        gateway: 172.16.238.1
```

- Makefile del directorio *rpm*

```
PACKAGE_NAME?=      k2http

VERSION?=       $(shell git describe --abbrev=6 --tags HEAD --always | sed 's/-/_/g')

BUILD_NUMBER?= 1

MOCK_CONFIG?=default

RESULT_DIR?=pkgs

all: rpm

SOURCES:
        mkdir -p SOURCES

archive: SOURCES
        cd ../../ && \
        git archive --prefix=$(PACKAGE_NAME)-$(VERSION)/ \
                -o packaging/rpm/SOURCES/$(PACKAGE_NAME)-$(VERSION).tar.gz HEAD

build_prepare: archive
        mkdir -p $(RESULT_DIR)
        rm -f $(RESULT_DIR)/$(PACKAGE_NAME)*.rpm

        srpm: build_prepare
        /usr/bin/mock \
                -r $(MOCK_CONFIG) \
                --define "__version $(VERSION)" \
                --define "__release $(BUILD_NUMBER)" \
                --resultdir=$(RESULT_DIR) \
                --buildsrpm \
                --spec=k2http.spec \
                --sources=SOURCES
        @echo "======= Source RPM now available in $(RESULT_DIR) ======="

rpm: srpm
        /usr/bin/mock \                     -r $(MOCK_CONFIG) \
                --define "__version $(VERSION)"\
                --define "__release $(BUILD_NUMBER)"\
                --resultdir=$(RESULT_DIR) \
                --rebuild $(RESULT_DIR)/$(PACKAGE_NAME)*.src.rpm
        @echo "======= Binary RPMs now available in $(RESULT_DIR) ======="
        clean:
        rm -rf SOURCES pkgs

distclean: clean
        rm -f build.log root.log state.log available_pkgs installed_pkgs \
                *.rpm *.tar.gz
```

- *K2http.spec*

```
Name:   k2http
Version: %{__version}
Release: %{__release}%{?dist}

License: GNU AGPLv3
URL: https://github.com/redBorder/k2http
Source0: %{name}-%{version}.tar.gz

BuildRequires: gcc librd-devel net-snmp-devel json-c-devel librdkafka-devel
                    libmatheval-devel

Summary: …
Group:   Development/Libraries/C and C++
Requires: librd0 libmatheval librdkafka1 net-snmp
Requires(pre): shadow-utils

%description
%{summary}

%prep
%setup -qn %{name}-%{version}

%build
./configure --prefix=/usr
make

%install
DESTDIR=%{buildroot} make install
mkdir -p %{buildroot}/usr/share/k2http
mkdir -p %{buildroot}/etc/k2http
install -D -m 644 k2http.service %{buildroot}/usr/lib/systemd/system/k2http.service
install -D -m 644 config.yml %{buildroot}/usr/share/k2http

%clean
rm -rf %{buildroot}

%pre
getent group k2http >/dev/null || groupadd –r k2http
getent passwd k2http >/dev/null || \
      useradd -r -g k2http -d / -s /sbin/nologin \
      -c "User of k2http service" k2http
exit 0

%post -p /sbin/ldconfig
%postun -p /sbin/ldconfig

%files
%defattr(755,root,root)
```

```
/usr/bin/k2http
%defattr(644,root,root)
/usr/share/k2http/config.yml
/usr/lib/systemd/system/k2http.service

%changelog
* Wed May 11 2016 José E. Romero <josromber@gmail.com> - 1.0.0-1
- first spec versión
```

ANEXO B: PIPELINE 1 Y DOCKERFILE DE N2KAFKA

- *Pipeline 1 de n2kafka*

```
gitlabCommitStatus {
    node ('docker') {
        stage 'Build'
        sh "echo ${env.gitlabSourceBranch}"
        docker.image('resources.redborder.lan:5000/rb-cbuilds-
                 n2kafka:1.1').inside {
            git branch: "${env.gitlabSourceBranch}",
                         credentialsId: 'jenkins_id',
                     url:'git@gitlab.redborder.lan:
                              dfernandez.ext/n2kafka.git'
            sh './configure --disable-optimization'
            sh 'make'
            stage 'Unit Tests'
            sh 'make tests'
        }

        stage 'Create Docker Image'
        git branch: 'continuous-integration', credentialsId:
                    'jenkins_id', url: 'git@gitlab.redborder.lan:
                     dfernandez.ext/n2kafka.git'
        sh '''
          docker build -t n2kafka .
          docker tag n2kafka
resources.redborder.lan:5000/n2kafka:latest
          docker push resources.redborder.lan:5000/n2kafka:latest
        '''
    }
}
```

- *Dockerfile n2kafka*

```
FROM resources.redborder.lan:5000/rb-cbuilds-n2kafka:latest
MAINTAINER Diego Fernández

RUN yum install -y supervisor && \
    yum install -y python-pip && \
    yum clean all

ADD . /app/
RUN cd /app; /app/configure; make;

RUN pip install j2cli
ADD supervisor/n2kafka.ini /etc/supervisord.d/
ADD templates/config.template /app/
ADD bootstrap.sh /app/
WORKDIR /app

CMD ["/app/bootstrap.sh"]
```

ANEXO C: DOCKERFILE Y PIPELINE PROYECTO WEB

En esta sección se expondrá el código de automatización para un proyecto web programado en ruby on rails. Se creará una imagen docker en el que se instalarán todas las dependencias y herramientas que necesita para poder compilar y ejecutar dicho proyecto.

Esta página web depende de dos bases de datos externas, una base de datos postgres y otra aerospike.

Las tareas que se abarcan en este apartado son hasta las pruebas unitarias.

- *Dockerfile web*

```
FROM ruby:2.1
MAINTAINER Jose Enrique Romero
# Install main dependencies
# Debian image, we use apt-get to install those.

RUN apt-get update && apt-get install -y \
  build-essential \
  nodejs \
  r-base r-base-dev \
  git

# Install dependencies of phantomJS
RUN apt-get install -y g++ flex bison gperf perl libsqlite3-dev \
  libfontconfig1-dev libicu-dev libfreetype6 libssl-dev \
  libpng-dev libjpeg-dev python libx11-dev libxext-dev

# Install phantom JS
RUN cd /tmp && \
    git clone --recurse-submodules
              git://github.com/ariya/phantomjs.git && \
              cd phantomjs && git checkout -f 2.1.1 && python
              build.py
RUN cp /tmp/phantomjs/bin/phantomjs /usr/bin/

# Install aerospike
RUN curl -L
  http://www.aerospike.com/download/tools/3.6.3/artifact/debian7
 | tar xvz && \
  cd aerospike-tools-3.6.3-debian7 && ./asinstall

# Delete TMP files
RUN rm -fr /tmp/*

# Add test files
ADD ./test-files /test-files

# Run tests by default!
CMD ["bundle", "exec", "rake", "test"]
```

- *Pipeline web*

```
node('docker') {

    git branch: 'development', credentialsId: 'jenkins_id',
           url: 'git@gitlab.redborder.lan:arodriguez/redborder-
                 manager.git'

    def postgres = docker.image('postgres:9')
    postgres.run('--name postgres -e POSTGRES_USER=redborder
                 -e POSTGRES_PASSWORD=redborder
                 -e POSTGRES_DB=redborder')

    def aerospike = docker.image('aerospike:latest')
    aerospike.run('--name aerospike')


   def rubyonrails = docker.build "rb-cbuild-bundler"
   rubyonrails.inside('--link postgres --link aerospike') {
      git branch: 'development', credentialsId: 'jenkins_id',
             url: 'git@gitlab.redborder.lan:arodriguez/redborder-
                   manager.git'


      stage 'test:api'
      sh '''
            cp /test-files/license/* ./lib/modules/aalicense/lib
            cp /test-files/config/* ./config
            rm ./Gemfile.lock
            bundler install --jobs $(nproc) --retry 2
                            --path=/cache/bundler --quiet
            /test-files/prepare_database.sh
            mkdir -p ./tmp/cache
            bundle exec rake test:load_fixture_modules
            bundle exec rake test:api
         '''


      stage 'test:non-integration'
      sh '''
            cp /test-files/license/* ./lib/modules/aalicense/lib
            cp /test-files/config/* ./config
            rm ./Gemfile.lock
            bundler install --jobs $(nproc) --retry 2
                             -path=/cache/bundler --quiet
            /test-files/prepare_database.sh
            mkdir -p ./tmp/cache
            bundle exec rake test:load_fixture_modules
            /test-files/aerospike.sh
            bundle exec rake test:non-integration
         '''

      stage 'test:precompile'
      sh '''
            cp /test-files/license/* ./lib/modules/aalicense/lib
```

```
            cp /test-files/config/* ./config
            rm ./Gemfile.lock

            bundler install --jobs $(nproc) --retry 2
                            --path=/cache/bundler --quiet
            /test-files/prepare_database.sh
            mkdir -p ./tmp/cache
            bundle exec rake test:load_fixture_modules
            NO_MODULES=1 RAILS_ENV=production bundle exec rake
                       redBorder:move_assets_module
            RAILS_ENV=production rake assets:precompile &>
                      /dev/null
         '''

      stage 'linter'
      sh '''
            cp /test-files/license/* ./lib/modules/aalicense/lib
            cp /test-files/config/* ./config
            rm ./Gemfile.lock
            bundler install --jobs $(nproc) --retry 2
                            --path=/cache/bundler --quiet

            mkdir -p ./tmp/cache
            bundle exec rake test:load_fixture_modules
            gem install rubocop --no-ri --no-rdoc -q
            rubocop -f s --fail-level E app config lib
         '''

   }

  sh 'docker stop postgres'
  sh 'docker rm postgres'

  sh 'docker stop aerospike'
  sh 'docker rm aerospike'

}
```

ANEXO D: EJEMPLOS DE PIPELINES

- Rb_nmsp: Servicio programado en Java y se usa maven para su compilación. Se compila dicho proyecto y se despliega para comprobar su correcto funcionamiento.

```
node ('docker') {
      stage 'Compilation'
         stage 'Run maven docker image'
         def maven = docker.image('maven:latest')
         maven.inside('-v /tmp/docker-m2cache:/root/.m2:rw'){
                 git branch: 'continuous-integration',
                     credentialsId: 'jenkins_id',
                     url: 'git@gitlab.redborder.lan:bigdata/rb-
                           nmsp.git'
            sh 'mvn clean package'
            stash includes: 'target/nmsp*-nodependecy.jar',
                            name: 'rb-nmsp'
         }
}
node ('prep-manager') {
    stage 'Copy to preproduction'
      unstash 'rb-nmsp'
      stage 'Update service'
      sh 'rm -f /opt/rb/var/nmspd/app/nmsp*-nodependecy.jar'
      sh 'cp target/nmsp*-nodependecy.jar
          /opt/rb/var/nmspd/app/'
      sh '/opt/rb/bin/rb_manager_scp.sh all
          /opt/rb/var/nmspd/app/nmsp*-nodependecy.jar'
      sh '/opt/rb/bin/rb_service all restart nmspd'
}
```

- *Rb_social:* Igual que el anterior, servicio programado en java y se usa maven para su compilación. En este caso se agranda la memoria del contenedor, ya que la memoria del mismo era insuficiente. Se vuelve a desplegar el servicio para comprobar su correcto funcionamiento.

```
node ('docker') {
      stage 'Compilation'
         stage 'Run maven docker image'
         def maven = docker.image('maven:latest')
         maven.inside('-v /tmp/docker-m2cache:/root/.m2:rw'){
            git branch: 'continuous-integration',
                        credentialsId: 'jenkins_id',
                        url: 'git@gitlab.redborder.lan:
                              bigdata/rb-social.git'
            sh 'export MAVEN_OPTS="-Xmx512m -Xms256m -Xss10m
                -XX:MaxPermSize=512m" && mvn clean package'
            stash includes: 'target/rb-social*-selfcontained.jar',
                            name: 'rb-social'
         }
}
node ('prep-manager') {
      stage 'Copy to preproduction'
```

```
      unstash 'rb-social'
      stage 'Update service'
      sh 'rm -f /opt/rb/var/rb-social/app/rb*-selfcontained.jar'
      sh 'cp target/rb-social*-selfcontained.jar /opt/rb/var/rb-
          social/app/'
      sh '/opt/rb/bin/rb_manager_scp.sh all /opt/rb/var/rb-
           social/app/rb*-selfcontained.jar'
      sh '/opt/rb/bin/rb_service all restart rb-social'
}
```

- *Samza_pms:* Idem anteriores.

```
node('docker') {
     stage 'Compilation'
     def maven = docker.image('maven:latest')
     maven.inside('-v /tmp/docker-m2cache:/root/.m2:rw'){
        git branch: 'continuous-integration', credentialsId:
                    'jenkins_id',
                    url: 'git@gitlab.redborder.lan:bigdata
                         /rb-samza-pms.git'
        sh 'mvn clean package'
        stash includes: 'target/rb-samza-pms*.tar.gz',
                         name: 'rb-samza-pms'
     }
}


node('jenkins-premanager1') {
      stage 'Copy to preproduction'
      unstash 'rb-samza-pms'
      stage 'Update service'
      sh 'rm -f /opt/rb/var/rb-samza-pms/app/rb-samza-pms*.tar.gz'
      sh 'cp target/rb-samza-pms*.tar.gz /opt/rb/var/rb-samza-
          pms/app/rb-samza-pms.tar.gz'
      sh '/opt/rb/bin/rb_upload_cookbooks.sh manager'
      sh '/opt/rb/bin/rb_wakeup_chef -c'
}
```