Universidad de Oviedo

ESCUELA POLITÉCNICA DE INGENIERÍA DE GIJÓN.

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

ÁREA DE INFORMÁTICA

TRABAJO FIN DE MÁSTER

ESTUDIO, MEJORA E IMPLANTACIÓN DE UN SISTEMA DE INTEGRACIÓN CONTINUA

D. BARREDO GIL, PABLO
TUTOR: D. USAMENTIAGA FERNÁNDEZ RUBÉN

FECHA: 01/2019

Acknowledgements

Agradecimientos tanto a mi tutor Rubén Usamentiaga, como a los responsables de la empresa que me han ayudado, Juan Luis y Miguel Ángel

Abstract

En el proceso de Integración Continua, los cambios individuales de un desarrollador son fusionados y testados, validando que un cambio individual no afecte al resto del código.

En el contexto de este proyecto se estudiarán los conceptos de la integración continua así como sus ventajas y opciones que existen para materializarla. Se explicaran cada una de las fases habituales así como las tecnologías que se utilizan actualmente, el motivo por el cual se escogerá cada una de ellas y posibles mejoras sobre los procesos existentes. El objetivo principal del presente trabajo será pues la puesta en marcha de un sistema de Integración Continua a utilizar en un entorno de desarrollo real y totalmente productivo, siendo susceptible de aplicar mejoras a lo largo del tiempo, al enfocar la implementación como un proyecto incremental.

Un aspecto destacable es la aportación de una evolución del desarrollo habitual en los sistemas de integración continua, mediante un entorno de programación y las herramientas para incluir los patrones habituales de desarrollo.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Premisa

El TFM se sitúa en el contexto de una necesidad real que ha surgido en una empresa y que he tenido que acometer como empleado de la misma. Todo lo descrito a lo largo del documento son las labores de ese trabajo así como las conclusiones que se han ido extrayendo a lo largo de diversas iteraciones. La tarea propuesta por la empresa consistió en el estudio y puesta en marcha de un sistema de integración continua, con vistas a seguir trabajando con ellos e ir mejorándolo de forma continua. Uno de los principales requisitos era que el sistema debía poder evolucionarse en el tiempo adaptándose a las nuevas tecnologías que fueran apareciendo por lo que era importante no buscar un sistema excesivamente cerrado. Uno de los retos era que el sistema debía ponerse en marcha en muy poco tiempo por lo que se tenía que trabajar en incrementos y atendiendo a las necesidades de la empresa, que fueron variando a medida que se fue desarrollando el proyecto.
El proyecto coexistió con la adopción de scrum por parte del departamento en el que trabajo, es por ello que el desarrollo de esta implantación siguió una metodología ágil.

1.2. Situación inicial

La empresa parte de un acercamiento a un sistema de integración continua, disponiendo de un servidor de integración, en este caso Jenkins, un repositorio de artefactos Nexus, un repositorio de código Gitlab y un servicio propio en el que guardar las configuraciones. La forma de despliegue estaba basada en Docker aunque era directamente desplegando contenedores en una maquina sin ningún tipo de orquestación.
El entorno al ser una mera prueba de concepto presentaba diversos problemas.

- No poder lanzar por fases independientes las tareas (solo build, solo deploy, etc)
- Duplicación de código al ser cada proyecto una copia del anterior
- Cambios generales implican cambiar todos los proyectos
- No existe control de versiones
- Falta de comunicación de resultado
- No comparte ningún tipo de estructura entre proyectos de distintos ámbitos (Webs, IoT, Big Data, etc)
- Son tareas con seleccionables en vez de código

La idea general en sí, si estaba bien enfocada y sirvió como base a todos los cambios y mejoras que se aplicaron.

1.2.1. Servidor de integración continua

La configuración del servidor daba a posibilidades de interbloqueo debido a que se siguió un esquema de creación de trabajos parametrizables que actuaban de plantillas. El problema que presentaba este esquema era que sumado a que solo existían dos ejecutores para los trabajos y que tareas llamaran a otras tareas, se podían cometer errores si no existía una intervención manual para asegurar que solo se hacia un trabajo de cada vez.

1.2.2. Repositorio de artefactos

Aunque todas las herramientas se tenían que desplegar dockerizadas para facilitar tanto la instalación, la actualización como su migración a un nuevo equipo, en este caso la herramienta estaba instalada en el propio equipo por lo que una de las tareas debía ser conseguir dockerizar dicho servicio.
La configuración y el uso del repositorio era adecuado aunque solo se guardaba la ultima versión de los ensamblados evitando así poder volver a una versión anterior sin requerir de un nuevo proceso de compilación.

1.2.3. Repositorio de código

La herramienta escogida por la empresa para albergar el código fue Gitlab, el cual tenía una configuración correcta pero muchas de las opciones no se utilizaban, como pueden ser, incidencias o la conexión con jenkins para poder hacer una verdadera integración continua.

1.2.4. Servicio de configuración

La empresa optó por la buena práctica de generar un solo ensamblado que fuera parametrizable para que no estuviera atado a una configuración que se generara en tiempo de compilación, brindando la oportunidad de tener múltiples entornos para trabajar.
Es por ello que diseñaron un pequeño servicio que dado un nombre de proyecto devolviera la configuración de dicho servicio.

1.3. Objetivos

Se busca llegar a tener una infraestructura que dará soporte a proyectos de I+D+i y que en sí misma es un proyecto de investigación. Las ideas principales que debe cumplir este modelo de integración continua deben ser:

- Mejorar lo existente y no rehacerlo con una implementación conceptual totalmente distinta
- Ser escalable, en el sentido de poder añadir nuevos nodos para satisfacer la demanda computacional.
- Aprovechar las ultimas tecnologías (contenerización, IaC).
- Tener una estructura que permita el trabajo con proyectos de diversas índoles.
- Estar totalmente parametrizada para reducir código repetido, facilitando así la propagación de cambios.
- Ser incremental, se debe de trabajar de tal manera que se puedan ir acometiendo pequeños desarrollos para ir viendo como se trabaja con cada una de las partes que se quiere investigar.
- Contar con una plataforma de desarrollo, para evitar trabajar directamente en la parte productiva.
- Estar totalmente automatizada, evitando siempre en todo lo que se pueda, la interacción humana.
- Tener una interfaz sencilla con el programador para que este no requiera de un conocimiento experto en la materia
- Facilitar la comunicación, haciendo que toda la información llegue fácilmente a todos los responsables y claramente las personas entiendan que está pasando en todo momento.

Capítulo 2

Conceptos

2.1. Repositorio de código

Para poder trabajar en equipo de forma efectiva es fundamental tener un punto común donde depositar el código, una opción podría ser una carpeta compartida o algún sistema similar como los utilizados en empresas para compartir documentos, el problema de estos acercamientos es que en desarrollo de código es fundamental tener un control de versiones que permita ver los cambios realizados a lo largo del tiempo para poder volver a una versión en cualquier momento, ya sea por recuperar una funcionalidad eliminada o subsanar un defecto. Otra ventaja que se tiene al trabajar con estas herramientas es la capacidad de ver qué personas han realizado qué partes o cuándo se ha introducido un determinado fallo y quien ha sido el responsable.

Estos sistemas tienen la ventaja de que están totalmente preparados para trabajar con ficheros de texto plano, detectando que lineas de código han variado respecto a una versión y otra. Esto es fundamental porque al trabajar en equipo existe una circunstancia que se puede dar y es que varias personas modifiquen el mismo archivo, si se trabajase con ficheros absolutos se tendría el problema de tener que comparar las versiones e introducir los cambios de los dos ficheros en un tercero, por suerte la mayoría de esta herramientas automatizan estos paso y en caso de conflicto muestran las dos lineas que ocupan la misma posición en el fichero y manualmente se puede arreglar el problema, aunque por la forma de trabajo de la empresa, en este caso particular, la mayoría de las veces no se tendrán estos conflictos.

Los repositorios de código han sido algo tan fundamental que son aceptados por casi la totalidad de los programadores y han permitido realizar trabajos colaborativos entre personas desconocidas participando en el llamado “opensource”, que se trata de programas cuyo código está publicado en un repositorio de código de forma

totalmente pública y en el que cualquier persona puede colaborar y aportar sus mejoras, la plataforma más utilizada es github, que está basada en git que es el control de versiones por excelencia y que facilita mucho el trabajo en equipo distribuido con los conceptos de las ramas, que son básicamente crear versiones del código de forma aislada en un mismo repositorio pudiendo trabajar en distintas funcionalidades sin incurrir en los conflictos, una vez se quieren fusionar estas dos ramas se realiza un "merge" que compara los archivos y si no hay conflictos los une y si no, el usuario puede revisar los conflictos y arreglarlos. En Github normalmente se crean incidencias o "issues" en los que se indican defectos o mejoras y un programador puede crear una rama realizar la mejora y una vez subida al repositorio, realiza una "pull request", que es la petición de realizar un merge normalmente sobre la rama principal para introducir su cambio y que pase a formar parte del software.

2.2. Gestor de dependencias

En el desarrollo de código se generan las llamadas dependencias, estas dependencias son librerías de terceros realizadas por la comunidad o por empresas, y que normalmente se descargan de sus propios servidores o un repositorio central y pueden llegar a desaparecer. Un enfoque podría ser no hacer desarrollos que dependan de terceros pero eso en la mayoría de casos es inviable por tiempo, dinero y fiabilidad, algo desarrollado por un tercero y utilizado por mucha gente tiene bastante probabilidad de que su funcionamiento sea correcto.
Es por ello que muchos proyectos se realizan con multitud de librerías y es necesario poder gestionarlas de una forma cómoda y eficaz, un primer acercamiento podría ser trabajar directamente incluyendo la librería descargada manualmente, el problema de esto es cuando se trabaja con decenas de librerías y por si no fuera suficiente, las librerías pueden tener a su vez dependencias y necesitar de otras que no son parte del proyecto o de alguna que se usan pero una versión distinta. Eso consigue que al final manejarlo a mano sea una tarea imposible y es por ello que existen estas herramientas. Un ejemplo de este sistema lo podemos encontrar en los sistemas operativos linux, para poder instalar un programa en vez de descargarlo directamente de la web del fabricante, existe un repositorio centralizado en el que se puede descargar el software y sus dependencias, de forma sencilla.

2.3. Repositorio de artefactos

El código puede ser un producto en si mismo como pueden ser scripts utilizados para tareas de manejo de servidores, pero normalmente no es el caso y es necesario un proceso de compilación o empaquetado creando un ensamblado que será lo que se desplegará en un servidor o un ordenador. Existe una gran variedad de posibles ensamblados, desde un ".exe" de windows, a un ".jar" de java o su versión pensada para servidores web tomcat ".war".
Estos ensamblados normalmente ocupan un gran tamaño y son archivos binarios los cuales en general no se suelen subir a los repositorios de código porque todas las funcionalidades descritas no funcionan al no ser capaz de detectar que línea ha cambiado, porque cambian todas por ser binario. Además los repositorios de código se vuelven muy pesados en las descargas si se introducen archivos pesados.
Es por ello que como también es necesario tener un versionado de estos ensamblados existe el concepto de repositorio de artefactos en los que se pueden depositar y etiquetar pudiéndose descargar una versión antigua en cualquier momento. La ventaja de poder ir directamente al ensamblado y no al código que lo ha generado es fundamentalmente por dos motivos, en primer lugar la compilación es un proceso que tarda en proyectos grandes una cantidad de tiempo moderada, además si se suma la batería de test que garantizan que ese código funciona se puede incrementar bastante el tiempo, haciendo muchas veces que volver rápidamente a una versión anterior no sea posible. El segundo problema es que un código puede no compilar pasado un tiempo, aunque el código en sí mismo no ha cambiado si puede pasar que sus dependencias no estén disponibles.

2.4. Metodologías de desarrollo

El desarrollo de software al igual que cualquier otro proyecto, necesita de una planificación para poder ejecutarlo de forma efectiva, es por ello que a lo largo del tiempo han ido surgiendo diversas metodologías para acometer un proyecto software, una de las más representativa y que se ha usado mucho y sigue siendo usada es métrica 3, que es una metodología en cascada, la cual se planifica el proyecto en diversas etapas, asignando un tiempo a cada una de ellas, consiguiendo así una linea temporal efectiva, siempre y cuando los requisitos se hayan definido de manera correcta y las tareas estén bien estimadas. Esta metodología se asemeja mucho a como se hacen proyectos de ingeniería en los que es fundamental ese estudio y planificación previo. Aunque las metodologías cascada son muy sólidas a primera vista, tienen mucha posibilidad de que los proyectos que las utilizan fracasen por una mala planificación inicial. El software es algo muy cambiante y evoluciona constantemente

y es afectado por muchos agentes externos, es por ello que han ido surgiendo diferentes metodologías que buscan centrase en la agilidad del desarrollo y trabaja de forma incremental, son las llamadas metodologías ágiles, la diferencia principal con cascada es que mientras que en cascada el producto no es usable hasta que se terminan todas las fases, las metodologías ágiles buscan generar un producto funcional y testado en cada incremento. Esta forma de trabajo permite una mejor colaboración entre empresa y cliente ya que el producto se puede ir enseñando al cliente a medida que se le van añadiendo funcionalidades y se pueden cambiar detalles o ir añadiendo nuevos requisitos. Aunque este sistema no está exento de problemáticas, la colaboración con el cliente puede ser una ventaja o un inconveniente porque no todos lo clientes quieren involucrarse en el desarrollo y quieren algo llave en mano, entregan unos requisitos y quieren eso.
Dentro de las metodologías ágiles la más destacable es scrum cuyo funcionamiento consiste en primer lugar obtener los requisitos que quiere el cliente y priorizarlos, generando el llamado "backlog", a continuación se define un incremento o "sprint" que se abordará en un plazo de 1 a 4 semanas, y en el que se escogerá del backlog una serie de tareas para realizar en ese incremento y pasado ese plazo se debe generar un producto software testado y totalmente funcional que contenga los requisitos escogidos. Donde reside la utilidad es en la parte de producto funcional y testado, ya que se garantiza que el cliente va a tener siempre una versión funcional, sobre la que poder probar las funcionalidades y decidir si se deben mejorar o son correctas.

2.5. Tipos de trabajo (Ramas, Trunk)

Un control de versiones se puede utilizar de diversas formas, según convenga más por el equipo, la empresa o una cuestión personal de organización. Cuando se utiliza git surge el debate de si usar las ramas o trabajar directamente en la misma rama. La ventaja de trabajar siempre en la rama master es que la integración se realiza cada vez que se introduce código por lo que rara vez se introducen cambios muy grandes, el problema de fusionar dos ramas es que la integración puede ser muy compleja al incluir mucho código nuevo, aunque tiene la ventaja de que facilita trabajar de forma aislada. Existen diversas opciones para trabajar tanto si se quieren usar ramas como si no, aunque la empresa utiliza la variante centralizada [1].

2.5.1. Centralized Workflow

En este caso se trabaja solo con la rama principal llamada master, la idea es que todos los cambios van siempre al mismo punto. Supongamos dos programadores John y Mary, John realiza un cambio y lo sube al repositorio central.

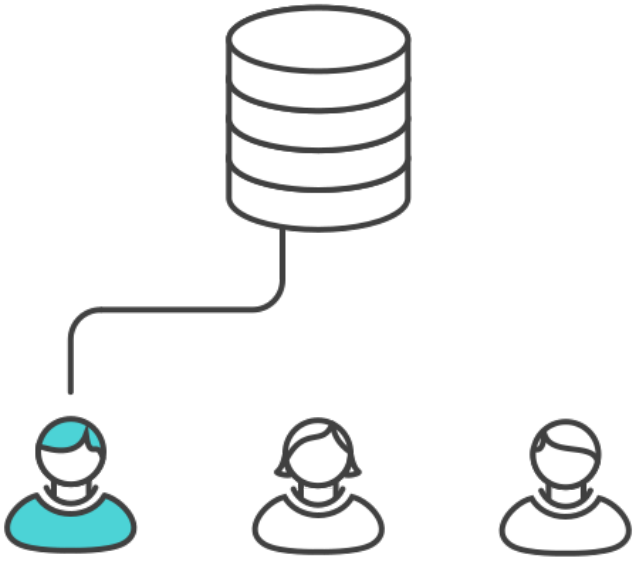

Figura 2.1: Ejemplo Centralized Workflow: Subida de John (Propiedad de Atlassian)

Al mismo tiempo la programadora Mary está trabajando en otra funcionalidad, una vez termine intentará subir los cambios al repositorio central, pero no podrá porque ya existen nuevos cambios que el no tiene en local.

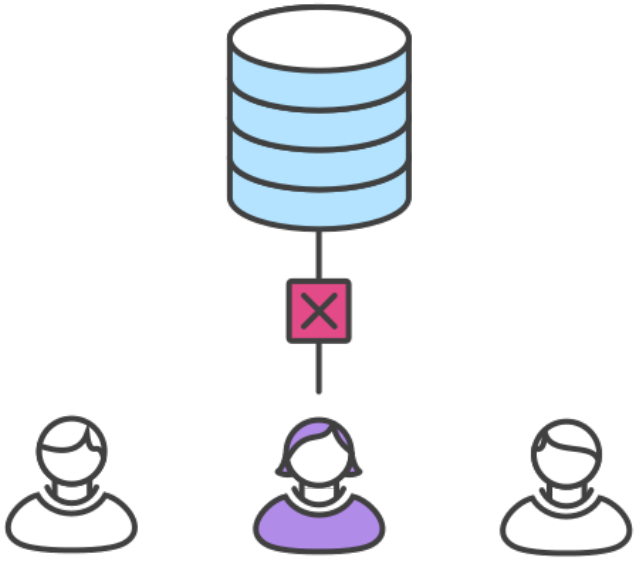

Figura 2.2: Ejemplo Centralized Workflow: Fallo de subida de Mary (Propiedad de Atlassian)

Para solucionar esto, debe de descargar los cambios que ha subido John, para luego poder subir los suyos.

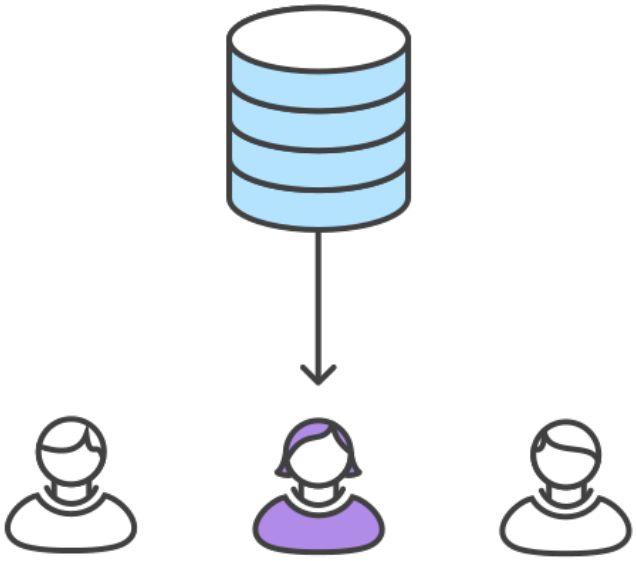

Figura 2.3: Ejemplo Centralized Workflow: Descarga de Mary (Propiedad de Atlassian)

En local realizará un nuevo commit quedando los cambios de las siguiente manera:

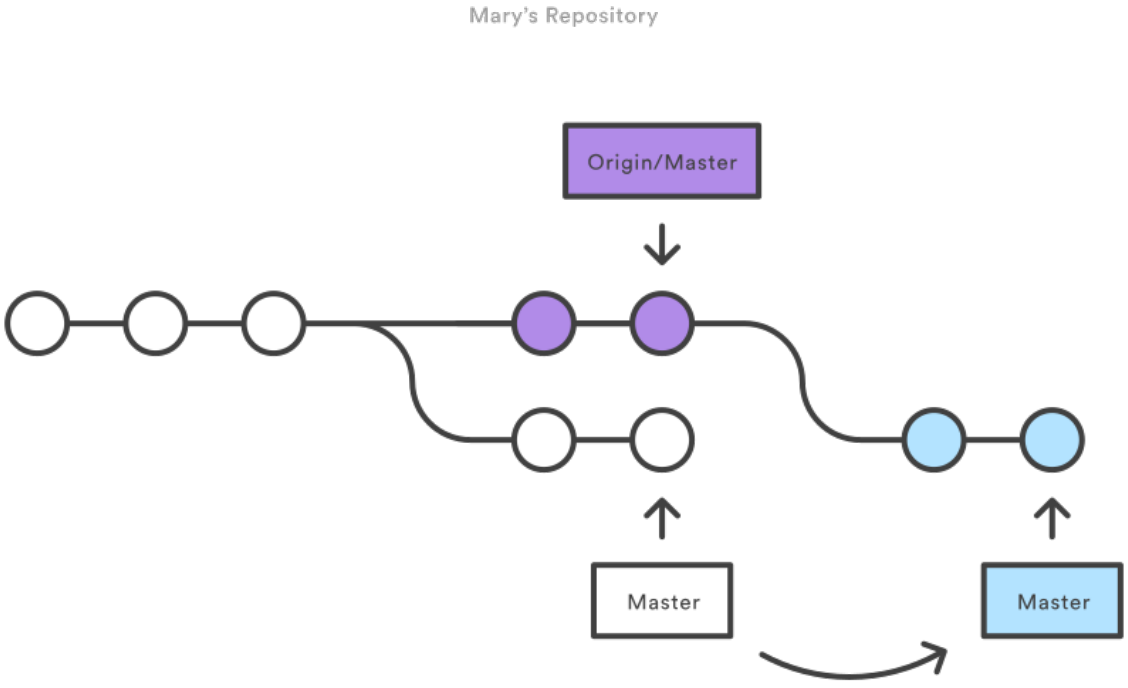

Figura 2.4: Ejemplo Centralized Workflow: Commit local (Propiedad de Atlassian)

Finalmente Mary puede subir sus cambios a local.

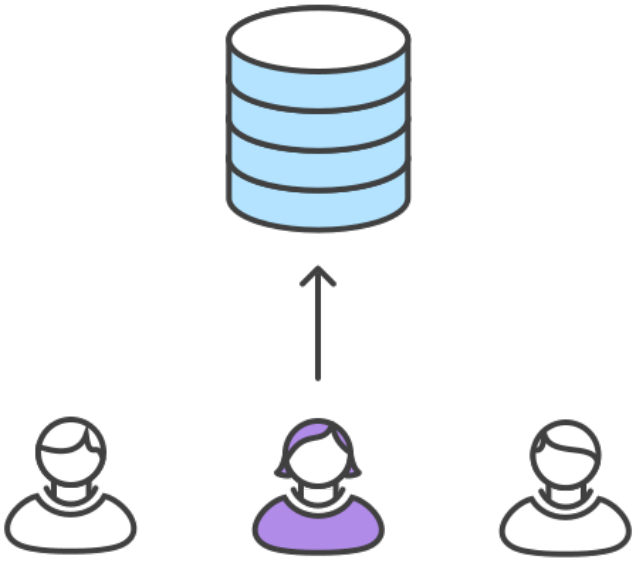

Figura 2.5: Ejemplo Centralized Workflow: Subida de los cambios (Propiedad de Atlassian)

2.6. Explicación CICD

En el software el despliegue de cualquier herramienta es una parte fundamental y crítica de cualquier desarrollo tanto software como hardware, es por ello que es fundamental que esta fase esté perfectamente diseñada y ejecutada ya que significa la culminación de un trabajo y es básicamente el producto que podrás ofertar, ya que una aplicación perfectamente desarrollada y diseñada pero que no despliega o lo hace con errores, no es vendible.
Tal es la importancia de esta fase que en muchas empresas se dedican días para hacerlo y verificar que todo ha salido de acuerdo a lo planificado, el problema de este sistema tan "mánual" es que requiere el esfuerzo de muchas personas en un determinado momento de tiempo incluido el estrés que genera y la preocupación del cliente.
En general suele ser una situación que genera incertidumbre ya que se hace poco y no se sabe si va a funcionar de forma correcta o no, y suelen aparecer dudas del tipo "¿La funcionalidad A se integrará bien con la funcionalidad B en producción?", Martin Fowler, Integración continua: [2]

2.6.1. Sistema de despliegue tradicional

Para poder explicar porque es necesario el uso de un sistema de integración continua primero se debe analizar como se ha estado haciendo durante muchos años en la gran mayoría de las organizaciones.

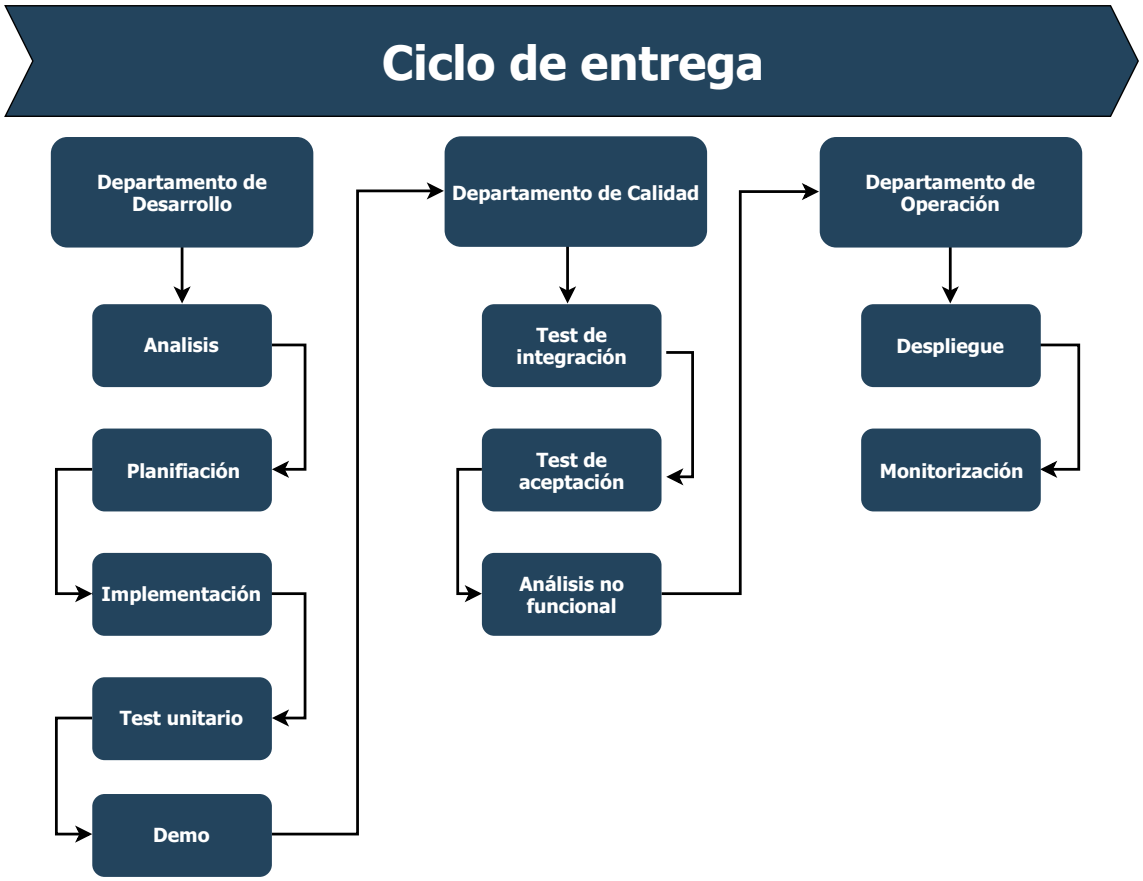

Figura 2.6: Esquema del desarrollo convencional

En primer lugar tenemos la figura del cliente que es la persona o empresa que definirá los requisitos que debe tener el producto, esta persona deberá hablar con la segunda figura implicada en el proyecto, que se trata del equipo de desarrollo, el cual tendrá que implementar todos los requisitos que pida el cliente. Se debe realizar una fase de análisis en la cual se estudiarán los requisitos, que quiere exactamente el cliente, la viabilidad del proyecto, las alternativas que existen para su ejecución y si tiene un coste factible.
Una vez analizado se procede a la siguiente fase que es la planificación, en este caso se debe de estimar la duración de cada uno de los requisitos y organizarlos en el tiempo, asignando unos plazos para cada uno de ellos consiguiendo así una fecha de entrega del producto.
Una vez se sabe que se debe implementar y en que orden, se procede con la fase de implementación en la que se codifica el proyecto software, pudiendo usarse distintas metodologías de desarrollo como las metodologías ágiles. La siguiente fase es la de testing unitario que dependiendo de la metodología utilizada se puede realizar al mismo tiempo que la implementación.

Una vez se tiene el código implementado y testado se puede generar una demo

para obtener una realimentación del cliente, por si fuera necesario realizar algún cambio. Una vez se ha terminado la programación es necesario facilitar el código al departamento de QA("quality assurance"), que es el encargado de garantizar que el producto tiene la calidad requeridas para cumplir las exigencias del cliente. En esta fase se realizan diversos tipos de testing, en primer lugar se deben hacer test de integración que comprueben que cada una de las partes del software funcionen con otra y que ningún cambio nuevo haya roto su funcionamiento.
Otro tipo de test que se realiza son los test de aceptación que verifican que el software hace lo que dicen los requisitos que debe hacer, ya que que el software funcione no quiere decir que, cumpla con lo que había pedido el cliente.
La ultima fase de test suele ser el testing no funcional en el que se revisa la seguridad, el rendimiento, la capacidad de recuperación, su disponibilidad y demás factores que pueden afectar a la calidad.

Finalmente con el producto desarrollado y verificada su calidad se puede proceder a su despliegue, entrando en la fase de operación, donde el equipo de operaciones por un lado realizan un despliegue en el que sustituyen la versión que actualmente está desplegada, si la hubiera y ponen en producción la nueva versión y en el caso de que falle el despliegue se ponen en contacto con el equipo de desarrollo para solucionarlo. Si el despliegue ha sido exitoso, el equipo de operaciones comienza la monitorización del software para garantizar que todo funciona de forma correcta a lo largo del tiempo.

Este proceso se realiza de forma cíclica y la duración de dichos ciclos varían dependiendo de la empresa, los sistemas y como se organice el equipo. En general el tiempo de duración de los ciclos suelen rondar entre una semana y pocos meses, aunque puede existir hasta ciclos anuales.
Este proceso de despliegue es muy común en toda la industria y es usado por un gran numero compañías, pero eso no hace que no esté libre de errores.

2.6.2. Problemas del sistema tradicional

Existen diversos problemas derivados del uso de este sistemas, los principales son los siguientes.

2.6.2.1. Lentitud en las entregas

El proceso al contar con tantas fases desde que se planifica hasta que se entrega finalmente al cliente, transcurre una gran cantidad de tiempo que hace que al final las mejoras y nuevos desarrollos tarden mucho en llegar a ser parte del producto

final, lo cual en algunos casos puede ser un problema ya que si se tarda en sacar una funcionalidad, se puede llegar tarde a una demanda del mercado, además el cliente tarda en poder ver el producto consiguiendo así que igual un ciclo no ha servido para nada debido a que no era lo que quería el cliente. Es por ello que el conseguir una velocidad en el desarrollo y despliegue es fundamental si se quiere conseguir ser competitivo en un mercado tan volátil como en el que se encuentra la industria.

2.6.2.2. Lentitud en la realimentación

El cliente no es el único que puede genera una realimentación, dentro del equipo de desarrollo se producen las realimentaciones, el problema de este enfoque es que al existir estás fase tan definidas desde que se desarrolla algo hasta que llega al equipo de QA pueden pasar meses, y eso puede hacer que solucionar ese defecto se convierta en una tarea ardua ya que igual muchos desarrollos posteriores dependen de esa funcionalidad incorrecta, por lo que el impacto de ese defecto es muy alto, y mientras se gasta tiempo en corregir errores de hace mese no se pueden realizar nuevos desarrollos.

2.6.2.3. Falta de automatización

En este sistema de trabajo, los despliegues no se producen de manera regular por lo que normalmente no se automatizan e implican mucha interacción humana que puede producir errores inesperados. Esto además se ve alimentado por la problemática psicológica de muchos equipos, ya que al estar trabajando con un producto que se ha tardado meses en desarrollar se tiene pánico a realizar un mal despliegue y se incurre en la falsa creencia que un despliegue supervisado por una persona tiene más probabilidad de éxito que algo automatizado, cuando normalmente la mayoría de errores son de naturaleza humana.

2.6.2.4. Riesgo de parches rápidos

Cuando se produce un error en producción se disparan todas las alarmas ya que eso puede significar un gran impacto económico y de prestigio dependiendo de la naturaleza del error detectado, es por ello que normalmente cuando se detecta este tipo de error se debe arreglar de forma rápida incurriendo en el llamado “hotfix”, que es básicamente un parche rápido que soluciona el error, el problema es que normalmente este tipo de soluciones al ser tan prioritarias no pasan por QA y directamente se despliegan en producción con todo el riesgo que esto conlleva y en muchas ocasiones el remedio en peor que la enfermedad y estos “hotfixes” llevan a más errores consiguiendo desestabilizar en gran medida el producto.

2.6.2.5. Estrés

Las personas que trabajan en estos desarrollos suelen estar sometidos a mucho estrés debido a los tiempos ajustados en estos ciclos ya que como se ha visto hay muchos puntos que pueden generar retrasos en las entregas debido a que un error en una fase ralentiza otra, el problema es que muchas veces estos ciclos tienen que realizarse en el tiempo pactado, porque si no, se puede incurrir en problemas contractuales, es por ello que el equipo está en constante tensión y este tipo de situaciones lo único que generan es un mal clima laboral, consiguiendo que el rendimiento de los trabajadores se vea mermado de forma aguda.

2.6.2.6. Fallos de comunicación

El producto pasa por unas fases muy marcadas y por diversas personas y equipos, es por ello que muchas veces se rompe la comunicación ya que muchas veces las personas que realizan las diversas tareas no se conocen porque trabajan en departamentos distintos o incluso empresas distintas, esto consigue que el trabajar como un solo conjunto no se hace y esto genera problemas de entendimiento, comunicaciones que llegan tarde e instrucciones incompletas.

2.6.2.7. No se comparte la responsabilidad

Cada persona realiza su tarea y esto produce que cuando algo falla las culpas se mueven de un punto a otro, ya que para el equipo de desarrollo, el producto está terminado cuando se ha codificado, para el equipo de QA, hecho significa probado y para el de operaciones es que está desplegado. Es por ello que si se detecta un fallo, en vez de asumir que ha sido un error de todos y arreglarlo, los esfuerzos se centran en buscar un culpable especifico.

2.6.2.8. Falta de satisfacción

Se producen insatisfacción tanto por parte del cliente como por el equipo, en primer lugar todos los problemas que se han comentado afectan de una manera u otra al cliente que al final se ve salpicado por conflictos, retrasos y productos que no funcionan como el esperaba. A su vez los trabajadores a pesar de que tienen mucho interés en su trabajo, se ven afectados por todos estos problemas y que muchas veces al depender del trabajo de otras personas pueden existir momentos en que realicen tareas consideradas menos interesantes como solucionar fallos, encontrarse parados porque un miembro de la cadena no ha terminado su parte o por otro lado verse saturados porque están siendo el eslabón débil de la cadena.

2.6.3. Nuevos modelos de despliegue

Para paliar los problemas anteriormente descritos han ido surgiendo nuevos modelos de despliegue el principal y que será el núcleo del trabajo es la Integración continua, en primer lugar es importante aclarar que le termino integración continua es algo concreto pero normalmente se usa para hablar de sus distintas variantes, que son la entrega continua (Continuous Delivery) y Desarrollo continuo (Continuous Delivery) [3].

2.6.4. Definición de "Continua"

En primer lugar se debe definir el termino continuidad que es la traducción del termino anglosajón "continuous", que en este caso significa automatización, en este caso la automatización de la mayoría de procesos que se encuentran en el desarrollo de productos software, tales como las pruebas del código, la creación de ensamblados, la monitorización, el despliegue configuración del mismo y la validación del producto final.

La palabra continua puede dar a confusión de su significado y se puede entender como algo que se repite de forma constante, pero en este caso se refiere a garantizar la fluidez del desarrollo, al poder hacer pruebas, compilaciones y despliegues, de manera automática del nuevo código que se incorpora a un proyecto. La idea es que si se garantiza que el coste de esas tareas no es muy alto y se pueden hacer de forma ágil garantizamos la continuidad del desarrollo, por lo tanto los errores en general son menos dañinos incluso en producción ya que la identificación de errores y su reporte se agiliza.

Existe una "metodología" de desarrollo conocida como "fail-fast"[4], que se fundamenta en que el software si tiene que fallar, falle rápido consiguiendo así la detección temprana de errores y que algo tan costoso como detectar fallos en producción sea bastante más sencillo. Esta técnica encaja perfectamente con la idea de continuidad ya que si se tiene un código que en el caso de tener un defecto, rápidamente falla y además tenemos un proceso que nos permite ir agregando mejoras de forma rápida, se consigue un desarrollo y realimentación del mismo de manera ágil y efectiva.

Como ya se adelantaba, la continuidad implica automatización, lo que conlleva que los cambios se pueden incluir sin intervenciones humana, esto garantiza varias propiedades:

- **Velocidad:** En primer lugar se consigue velocidad, un proceso automático se realiza mucho más rápido que uno hecho por una persona
- **Fiabilidad:** La mayoría de los fallos se producen por errores humanos, al automatizarlos reducimos en gran medida esa posibilidad.
- **Repetibilidad:** Al ser un proceso que siempre realiza las mismas acciones es fácil replicar el fallo, ya que el proceso suele ser determinista, ante la misma entrada tenemos la misma salida, mientras que en el caso de un proceso operado por humanos, no siempre tenemos esta posibilidad, ya que igual el error fue introducido por algo distinto que se hizo en ese momento en concreto.

Existen procesos que si pueden requerir de intervención humana, como puede ser dar el visto bueno para pasar algo a producción, ya que aunque se pueden validar los requisitos de forma automática, siempre un usuario final podrá dar una mejor valoración.

2.6.5. Integración continua

En primer lugar se tiene el término que da nombre al proyecto, Integración Continua, se trata de combinar el código de múltiples desarrolladores, realizarle pruebas y validarlo, consiguiendo así la detección de posibles errores en los cambios o bien la incompatibilidad de dos cambios, consiguiendo así no tener código no funcional durante mucho tiempo.
Esta fase prueba el código como ente único sin tener en cuenta la interacción con otro software, se centra en probar que las funciones producen las salidas esperadas, dadas las entradas adecuadas y que las incorrectas generan los errores correspondientes.

2.6.5.1. Test unitario

Este tipo de pruebas buscan cubrir las funcionalidades de la aplicación a un nivel más básico. Se prueba cada sección de código normalmente una función, de forma aislada y se verifica si dada una serie de condiciones, responde de la forma esperada. Descomponer las pruebas a este nivel, genera la confianza para afirmar que cada parte de la aplicación se comportará de la forma adecuada y permite comprobar los extremos que suele ser donde se genera el comportamiento más inesperado.

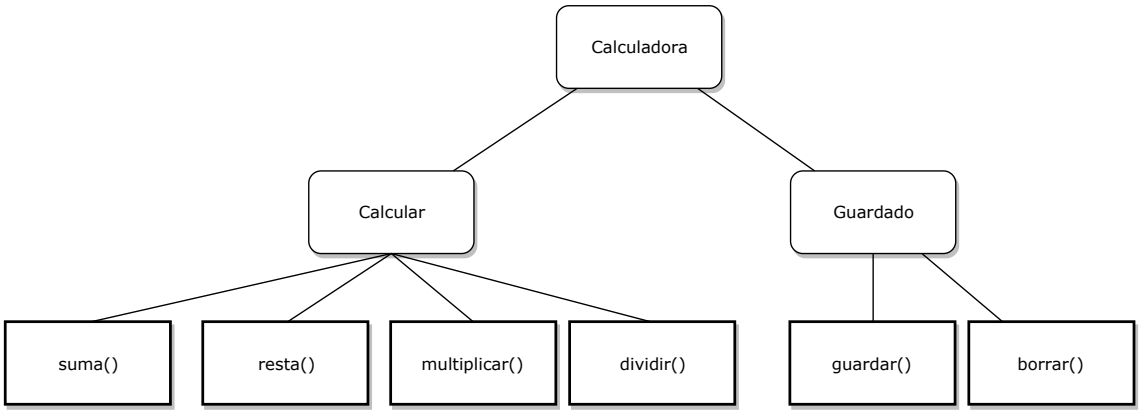

Figura 2.7: Ejemplo de aplicación dividida en funciones

Como se puede ver en la imagen aparece una calculadora y está descompuesta en unidades funcionales mínimas. La idea es que si se prueba cada una de las partes de la aplicación de esta manera se puede garantizar que la aplicación funcionará en un futuro. Esta separación se hace porque es mucho más fácil atacar pequeñas funciones que intentar probar toda la aplicación. En primer lugar se deben identificar las entradas, tanto las validas como invalidas, en segundo lugar los resultados posibles para tanto las entradas validas como las invalidas y una vez se tienen esos datos se pueden crear los test para garantizar que la calculadora funciona de forma correcta y es capaz de gestionar los casos de error como podría ser la división "0/0". Esta separación cumple un doble objetivo, en primer lugar permite saber si partes del código tienen una complejidad demasiado elevada, al necesitar de muchos test para probar la funcionalidad, esto es un indicativo de que se debe fraccionar más esa función y probablemente tener funciones diferentes. También funciona a la inversa se puede detectar una función demasiado simple que se puede fusionar con otra similar para crear una función más útil.

2.6.5.2. Integración

En este apartado se deben ejecutar los test unitarios descritos anteriormente para probar el código una vez se ha añadido código nuevo y detectar si esos cambios producen errores o efectivamente la aplicación funciona perfectamente y se puede considerar que todo está integrado. Un ejemplo puede ser dada una calculadora que solo suma enteros añadir la posibilidad de sumar decimales, el programador que ha realizado la suma con decimales puede haber verificado que su suma de decimales funciona, pero no detectar que la suma de enteros presenta un comportamiento anómalo por la inclusión de decimales. En este ejemplo los test unitarios en la fase de integración detectaría que la suma de enteros ha dejado de funcionar, avisando así que el código rompe funcionalidad y se debe revisar. La clave es que todo este proceso está automatizado consiguiendo así una gran agilidad.

2.6.6. Entrega Continua

La Integración Continua como se ha descrito solo comprueba que el código como ente aislado funciona, no se garantiza que ese código es desplegable. La garantía de que se cuenta con un elemento desplegable es lo que se realiza en esta fase.

2.6.6.1. Ensamblado

Como ya se adelantó al principio, el código para poder ser desplegado se debe de poder compilar o empaquetar generando así algo que ejecutar en un servidor de aplicaciones o distribuir a un cliente en un instalador. Uno de los objetivos por tanto que se debe alcanzar es la posibilidad de compilar el código y crear un ensamblado que sea perfectamente ejecutable.

2.6.6.2. Versionado

El código de una aplicación evoluciona en el tiempo y por lo tanto a lo largo del tiempo se van generando diferentes versiones de una aplicación, es por ello que es importante poder tener algún tipo de mecanismo para poder identificarla y tenerlas correctamente identificadas. Aunque no se quiera seguir un sistema de versionado muy estricto una practica habitual es al menos tener etiquetados los ensamblados pertenecientes a distintos entornos(producción, pre-producción, desarrollo, etc...).

2.6.6.3. Test de integración

Una aplicación normalmente no es un ente único y tiene dependencias con otras partes, como pueden ser bases de datos u otras aplicaciones que trabajan conjuntamente con el software que se quiere desplegar.
A diferencia del test unitario en este caso las pruebas se realizan sobre el correcto funcionamiento de todas las partes, ya que el funcionamiento de las partes como entes únicos no garantiza que trabajando de forma conjunta funcionen.[5]

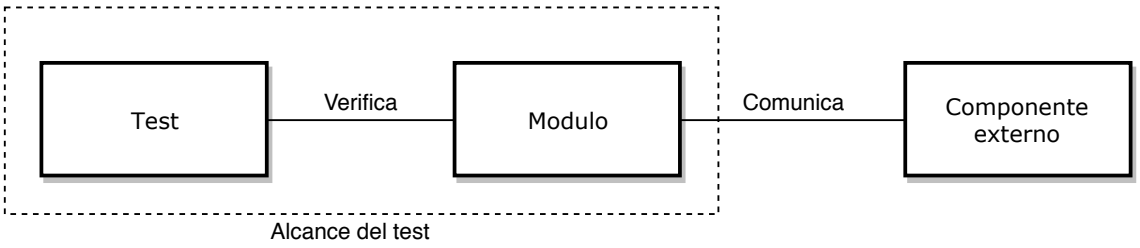

Figura 2.8: Ejemplo de test de la comunicación de una aplicación con otra

2.6.6.4. Test no funcional

Las aplicaciones normalmente no son utilizadas por un único usuario si no que son expuestas a un gran numero de personas. Otro aspecto importante es que los tests normalmente solo cubren que la funcionalidad funcione pero no verifican ni la velocidad ni la capacidad para trabajar con grandes volúmenes de datos. Es por ello que se debe de analizar este apartado, porque se puede dar el caso que se tiene una aplicación que muestra gráficas y efectivamente los test dicen que dando unos puntos los pinta, y otros test garantiza que el servicio de gráficas es capaz de conectarse a un servidor externo y obtener los puntos a pintar. El problema está en que en un entorno real en vez de pintar 10 puntos igual tiene que pintar 10000 puntos. Para este caso se realizan los llamados test de rendimiento en los que el objetivo es someter a la aplicación a una carga elevada y ver como funciona detectando problemas que puede provocar este uso, tales como cuelgues, cierres de la aplicación o funcionamiento indebido.

Otra parte importante que no se revisa en los test es la calidad del código. Que un código funcione no quiere decir que sea bueno ya que puede estar exponiendo múltiples vulnerabilidades o siendo propenso a dejar de funcionar a corto plazo por usar funciones que se van a quitar de librerías externas. Otro factor a tener en cuenta es que el código debe ser legible y debe evitarse el generar funciones muy complejas de entender ya que a lo largo del tiempo van a pasar muchas personas por ese código y no se puede dar el caso de tener que realizar un nuevo desarrollo ya que esa parte no se puede tocar al no ser totalmente indescifrable.

2.6.7. Despliegue continuo

La principal diferencia entre la Entrega Continua y el Despliegue Continuo es que en el primer caso solo garantizamos que se puede desplegar, no se llega a desplegar de forma automática. Es por ello que esta variante es hacer todo lo que se ha visto en el resto de apartados pero además desplegar el código y verificar que funciona.

2.6.7.1. Comprobaciones manuales

Según el tipo de producto se puede necesitar la aprobación de una persona para que se realice el despliegue, esto puede ser por muchos motivos, desde la espera a una campaña concreta, a la necesidad de pruebas de usuarios o cualquier otro motivo. Es por ello que muchas veces se realizarán todas las fase y el despliegue quedará esperando a una interacción humana que de el visto bueno y entonces se desplegará de forma automática.

2.6.7.2. Distintos entornos

Una practica habitual es tener diferentes entornos donde poder probar las aplicaciones, uno de los más habituales es pre-producción y producción, la idea de estos entornos es que se crea un ensamblado con las nuevas funcionalidades, se prueba en un entorno muy cercano a la realidad y si es correcto se mueve el mismo ensamblado a producción. Basándose en esta idea de entornos lo que se puede hacer, es que el proceso de despliegue en pre-producción está automatizado cada vez que se realiza un "commit" pero el paso a producción debe de lanzarse a mano, aunque lo que es el despliegue en sí sigue estando totalmente automatizado.

2.7. Contenerización

La virtualización es un concepto muy establecido y aceptado aunque de forma reciente ha tomado fuerza una variante que es la virtualización ligera, que consiste en solo virtualizar el código de la aplicación evitando tener que cargar un sistema operativo completo. Este concepto realmente no es nuevo ya que ya existía con los contenedores de linux pero gracias a la aparición de Docker y las arquitecturas de microservicios ha tomado muchísima relevancia.

2.8. Orquestación

Este tema enlaza con el anterior ya que una vez se tiene múltiples contenedores es fundamental poder manejarlos de forma distribuida para poder aprovechar maximizar la computación y disponibilidad. La orquestación consiste básicamente en una herramienta capaz de manejar múltiples contenedores ya sea de forma local o de forma distribuida.

Capítulo 3

Prospección tecnológica

3.1. Repositorio de código

3.1.1. Github

Github es uno de lo servidores de código más utilizados y populares. Su éxito se debe principalmente por alojar infinidad de código Open source y permitir la colaboración de diversas personas en un proyecto común a fin de proveer a la comunidad de un software accesible por todo el mundo y de forma gratuita. Github es propiedad de Microsoft[6] y tiene un modelo de negocio que consiste en ofrecer repositorios públicos de manera totalmente gratuita y que la opción de hacerlos privados sea de pago, estas dos opciones son en su propia infraestructura aunque ofrecen la opción de tener un Github en la infraestructura del cliente, que puede ser física o virtual, alojada en Azure, AWS u otro proveedor Cloud [7]. En general ofrece las opciones que se puede esperar de un repositorio de código, se puede subir código, crear ramas, incidencias, pull request y permite también una wiki para poder alojar documentación del proyecto. No ofrece "per se" ningún modulo de integración continua pero si ofrece opciones de add-ons para dotar de más funcionalidad a su plataforma integrándola con terceras partes con un coste adicional [8].

3.1.2. Gitlab

Se trata de un repositorio de código Open source muy completo que realiza mejoras de forma periódica, buscando ir adaptándose a la comunidad y sus necesidades. En cuanto al modelo comercial, tiene una gran variedad de productos, en primer lugar existe la "Community Edition" que es totalmente gratuita y se despliega en la infraestructura propia del cliente, también llamada "on-premise". Esta versión tiene casi todas las funciones más importantes y se han quitado las que están más relacionadas con la gestión o mejoras para la integración continua. La otra opción on-premise, es la llamada "Enterprise Edition", que cuenta con diferentes opciones según se necesite más o menos funcionalidades. El pago se realiza por número de

usuarios al mes, teniendo en cuenta el precio de la edición escogida. La otra opción es recurrir a su infraestructura, teniendo el mismo reparto que en la versión "on-premise", contamos con un plan gratuito que ofrece repositorios privados gratuitos y las mismas opciones que en la versión CE, y luego están las versiones comerciales que se pagan de la misma manera que lo ya visto [9].

Respecto a la funcionalidad, Gitlab a medida que se ha ido desarrollando se han introducido multitud de opciones, muchas de ellas no estaban disponibles cuando se empezó este documento. Gitlab intenta ser una herramienta que unifica todas las necesidades de un equipo de desarrollo, ofreciendo repositorio de código, de artefactos, servidor de integración continua, análisis estático de código, gestión de despliegues e incluso monitorización [10].

3.1.3. Bitbucket

Otro peso pesado de los repositorios de código, una de las razones a las que debe su popularidad es la opción de ofrecer repositorios privados totalmente gratuitos aunque en este caso a diferencia de Gitlab si que tiene un limite de colaboradores. Se trata de un producto propiedad de Atlassian que se integra con todas las herramientas que ofrecen para los equipos de desarrollo. En cuanto al modelo de pago, solo existe opción gratuita en su propia infraestructura. La opción "on-premise" tiene un coste simbolico de 10 euros para 10 usuarios, que dedican a beneficencia, aunque todo lo que pasa de 10 usuarios ya tiene un precio bastante más elevado. Tienen opción de una versión pensada para empresas que necesitan de redundancia con diferentes opciones para empresas de gran tamaño. Las mismas dos versiones se pueden encontrar en la opción de su infraestructura [11].

En cuanto a las opciones que ofrece se encuentran al igual que el resto de herramientas, guardar código, pull request, incidencias y ramas. El producto está muy pensado para ser usado con otras herramientas de la compañía y a diferencia de Gitlab, Atlassian cubre todas las necesidades del desarrollador, en vez de con una única herramienta, con el uso de su ecosistema [12].

3.1.4. Microsoft TFS

Es el acercamiento a una suite completa de herramientas para desarrollares de Microsoft, al principio no funcionaba con git, lo cual era un gran inconveniente aunque ahora mismo si lo utiliza por lo que puede resultar interesante. En cuanto funcionalidades, tiene al igual que Gitlab, todo el conjunto de herramientas que puede

necesitar un equipo de desarrolladores y cuenta con su propio modulo de integración continua. Este producto está pensado para instalarse en un servidor Windows, por lo que hay que tener en cuenta que se necesita una licencia de Windows server[13]

En cuanto al modelo de negocio, se puede pagar de forma mensual o realizar una compra de una licencia por 3 años [14].

3.1.5. Azure Devops

Microsoft ofrece una alternativa a su propio producto, y es la versión cloud, la cual mejora este ecosistema de productos y añade la ventaja añadida de que el cliente se despreocupa de la infraestructura al ser un servicio en la nube [15].

Respecto a los modelos de pago existen dos opciones, una gratuita con 5 usuarios y otra de pago con distinto numero de usuarios y coste por mes [16].

3.1.6. Tabla comparativa

Tecnología/Funcionalidad	Github	Gitlab	Bitbucket	TFS	Azure DevOps
Revisión de código	Si	Si	Si	Si	Si
Incidencias	Si	Si	Si	Si	Si
Wiki	Si	Si	Si	Si	Si
Proyectos privados (colaborativos)	De pago	Si	Si, hasta 5 usuarios	Si	Si
Proyectos personales	Si	Si	Si	Si	Si
Integración continua	Con plugins	Si	Con otra herramienta	Si	Si
Monitorización	No	Si	No	No	Si
Open source	No	Si	No	No	No
On-premise	De Pago	Si	De pago	Si	No
Cloud	Si	Si	Si	No	Si

Finalmente se escoge Gitlab principalmente, por ser una herramienta Open source, que tiene muchas funcionalidades interesantes, y la versión gratuita tiene todo lo necesario, además se integra muy bien con el resto de herramientas que se usarán.

3.2. Gestor de dependencias

3.2.1. Maven

Se trata principalmente de un gestor de dependencias, aunque también cumple el objetivo de gestionar los procesos de compilación y test. Se trata de una herramienta albergada bajo el amparo de la fundación Apache, y se ha convertido en una herramienta de referencia para los proyectos de Java, aunque se puede usar con multitud de lenguajes. Además de ofrecer una gestión de paquetes una de las ventajas que

tiene, es que "obliga" a tener una estructura de proyecto concreta, la cual permite trabajar de forma muy organizada [17].

3.2.2. Ant

Se trata de otro proyecto de Apache, que es muy similar a Maven, aunque en este caso no tiene ninguna imposición de estructura, permitiendo tener proyectos peor organizados. En este caso Ant como tal no gestiona dependencias y solo oferta la parte de hacer compilaciones y test unitarios [18], pero como está muy preparado para ejecutar tareas, se puede usar en combinación con Apache Ivy [19]. Apache ivy es un gestor de dependencias también parte de la fundación Apache, y está muy pensado para integrase con Ant.

3.2.3. Gradle

Gradle comparte muchas similitudes con Maven ya que tiene tanto el gestor de paquetes como la parte de compilación y test integrada. Cuenta con soporte para multiples lenguajes, incluyendo, Java, Groovy, C y C++. Se trata de proyecto Open source, pero en este caso no está albergado en la fundación Apache. Su popularización se dio en gran medida gracias a ser el gestor por defecto de los proyectos de Android [20].

3.2.4. NPM

Se trata de un gestor de dependencias para proyectos de Javascript y lenguajes de la misma familia. Permite al igual que sus competidores, la gestión de tareas de compilación y test. Si se quiere realizar cualquier proyecto Web con un framework moderno como puede ser Angualar, Vue o React, esta es la herramienta que se debe utilizar [21].

3.2.5. Tabla comparativa

Tecnología/Funcionalidad	Maven	Ant	Gradle	NPM
Gestor de dependencias	Si	Con Apache Ivy	Si	Si
Definir tareas	Si	Si	Si	Si
Open source	Si	Si	Si	Si
Multilenguaje	Si	Si	Si	Limitado a javascript

Las herramientas que se utilizarán será Maven y NPM, ya que la primera se utilizará para proyectos de Java y la segunda para los proyectos de Javascript.

3.3. Repositorio de artefactos

3.3.1. Nexus

Se trata de un repositorio de artefactos, el cual cuenta con bastantes funcionalidades que facilitan el almacenamiento de los artefactos, en primer lugar, tiene soporte nativo par docker registry, para maven, para python y para npm, lo cuál permite que la integración sea muy accesible. En segundo lugar cuenta con imagen de docker oficial y la versión gratuita cuenta con otodas las funcionalidades que se necesitán en un principio. La empresa que realiza este producto es Sonatype y cuentan con diversos productos complementarios y una versión Entrerprise [22].

3.3.2. Jfrog artifactory

Ofrece practicamente lo mismo que su competidor, en este caso cuentan con una versión Open source y varias de pago, la versión gratuita cumple con lo necesario aunque cuenta con algunas partes limitadas. La empresa responsable es Jfrog y al igual que Sonatype, tienen toda una suite de productos comerciales que complementan a sus otros productos. Ofrece una versión cloud, directamente gestionada por la empresa, lo cual puede ser una ventaja si no se quiere incurrir en el coste de infraestructura propia [23].

3.3.3. Tabla comparativa

Tecnología/Funcionalidad	Nexus	Jfrog
Variedad de artefactos	Si	Si
HA	Si	Si
Open source	Si	Si
Opción cloud	No	Si
Imagen de Docker	Si	Si

En general cualquiera de las dos opciones es válida, aunque en el caso de este proyecto se decantó por usar Nexus, dado que el personal por su experiencias en otros proyectos y empresas estaban acostumbrados a usar esta herramienta.

3.4. Servidores CICD

3.4.1. Jenkins

Se trata de uno de los servidores de integración continua más conocidos y extendidos, el proyecto empezó con Hudson pero terminó siendo abandonado y se empezó a utilizar un fork llamado Jenkins [24]. Una de las mayores ventajas, es la

gran cantidad de plugins que existen por lo que se puede dotar de casi cualquier funcionalidad que se busque. Dentro de sus cualidades, está el uso de los Jenkinsfile, la idea es tratar los pipeline como código, pudiendo almacenarlos en repositorios de código. El modo de funcionamiento es con un maestro que reparte tareas y con esclavos que las ejecutan, haciendo que sea un sistema distribuido aunque no cuenta con una solución de alta disponibilidad pura [25].

3.4.2. Bamboo

Se trata de la solución de integración continua ofrecida por Atlassian, la ventaja añadida, es la perfecta integración con el resto de herramientas de la compañía, por lo que si se trabaja con por ejemplo Bitbucket, es recomendable utilizar bamboo por su integración. En cuanto a funcionalidades, cuenta con un sistema de esclavos pero solo en una de las versiones de pago, además no cuenta con ninguna versión no comercial. Cuenta con sistema de plugins pero en este caso existen plugins que requieren de pago a terceros por lo que el precio final del sistema se puede encarecer. Cuenta con una versión cloud llamada bitbucket pipelines [26].

3.4.3. Travis

Es uno de los proyectos de integración continua más utilizados sobre todo por en el escenario Open source, ya que es totalmente gratuito para este tipo de proyectos. En cuanto a funcionalidades cuenta con las que cabria esperar, tiene su versión de Jenkinsfile para poder tener el código almacenado pero no cuenta con plugins. Solo ofertan versión cloud y para los desarrollos no Open source, se debe recurrir a la versión comercial, que cuenta con distintos precios dependiendo del numero de trabajos concurrentes que se quieran tener [27].

3.4.4. Tabla comparativa

Tecnología/Funcionalidad	Jenkins	Bamboo	Travis
Gratuito	Si	No	Solo Open source
HA	No	No	Si
Open source	Si	No	No
Opción cloud	No	Si	SI
Plugins	Si	Si	Si
Imagen de Docker	Si	No	No

Finalmente se escoge Jenkins ya que se quiere tener en infraestructura propia y tiene el añadido de ser gratuito y poder dockerizarse.

Capítulo 4

Diseño

4.1. Componentes de la integración continua

Se debe diseñar una infraestructura que permita realizar todas las tareas descritas anteriormente, es por ello que a continuación se planteará el esquema necesario para poder realizar la variante de "despliegue continuo", en la cuál cual el objetivo final es la automatización del despliegue.

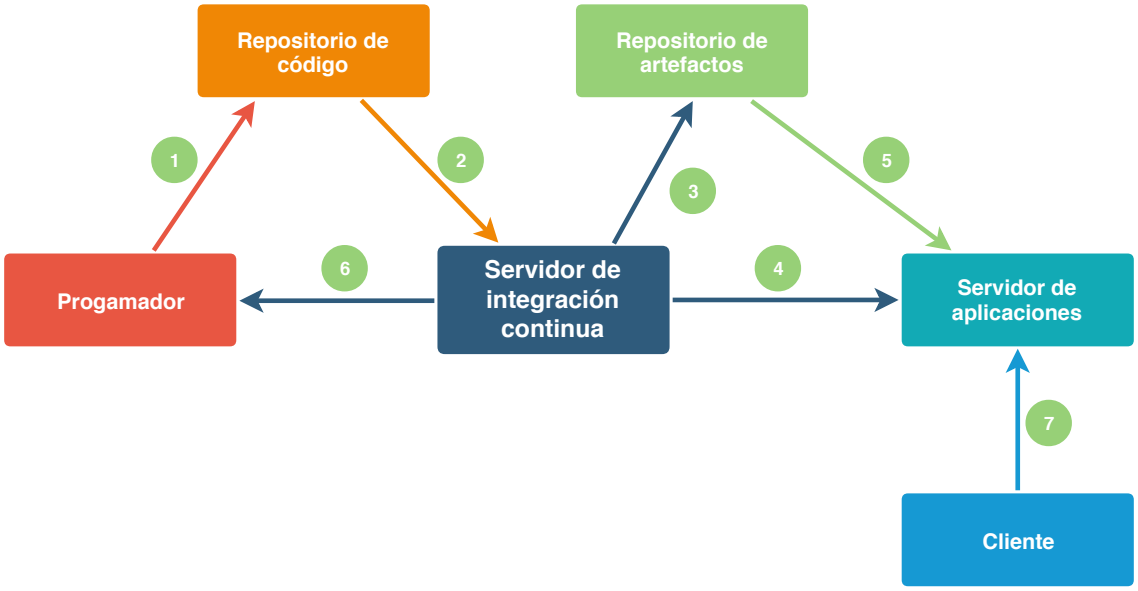

Figura 4.1: Esquema de diseño de la arquitectura CI

En primer lugar se cuenta con el programador, el cual tiene su entorno de desarrollo, donde irá acometiendo las tareas que le digan y realizando los cambios necesarios. El programador deberá tener realizados una serie de test unitarios que deberá pasar antes de subir los cambios para garantizar que no se sube código que no funciona. Esta medida es para evitar malgastar tiempo subiendo funcionalidades inacabadas, y ayuda a que las personas entiendan que deben de garantizar que lo que hacen funcione. Algo complementario que se debe intentar en todos los entornos que lo

permita, es tener también herramientas de análisis de calidad de código para poder en local ir intentando hacer código más eficiente.

Una vez el programador ha terminado sus cambios y realizadas todas las medidas que garantizan que su software en local funciona y pasa los test. Debe de subirlo a un repositorio como podemos ver en el paso 1 de la figura, previamente configurado. En este caso se sigue el modelo de una rama única, por lo que todo el código se debe de integrar de forma instantánea, este es parte del motivo por lo que se debe evitar subir código que claramente no funciona de forma intencional. El repositorio en este esquema es un ente que solo alberga código y delegará la parte de integración continua a un servidor de integración. Esta tarea de notificación se realiza cada vez que se recibe un commit, lo cual hace que todo código que se sube al repositorio va a ser compilado y desplegado.

El servidor de integración continua tiene una tarea que será capaz de bajarse el código, actividad que realiza en el paso 2, ejecutar los test unitarios y si son correctos intentará compilar el código. Una vez compilado, se hace una tarea adicional simplemente con carácter informativo, pero nunca restrictivo, que es el análisis de la calidad del código. Se escanea el código para detectar posibles vulnerabilidades, mandando a un servidor de calidad de código realizar un informe que permita descubrir si pueden surgir problemas a largo plazo. Además se evalúa la cobertura de test, para ver como de efectivos están siendo esos test, pero una vez más una cobertura de 0 o la detección de errores no bloquean el despliegue, aunque perfectamente se puede hacer y para proyectos más maduros es posible que sea necesario, aunque en un principio se prefiere hacer código que funcione aunque sea mejorable y luego con la ayuda de estas herramientas ir refactorizando, en parte esta decisión permite agilizar despliegues intentando no gastar mucho tiempo en el despliegue de funcionalidades, ya que muchas veces al trabajar con ideas y programas no finales se añaden y quitan funcionalidades, por lo que gastar mucho tiempo en algo que no se sabe si va a ser final es contraproducente.

El servidor de integración continua hace además la tarea de comunicarse con el repositorio de artefactos y guardar el ensamblado, esto corresponde al paso 3. El modelo que se seguirá para el versionado es el siguiente. Existe una versión final por entorno, es decir la ultima versión que compiló y se desplegó. Esta versión es remplazada por la nueva versión que se ha compilado y pasado los test. Para guardar la versión anterior lo que se tiene es una versión en cada entorno, que no solo compiló y desplegó si no que además pasó los test de integración, garantizando que ese ensamblado funciona. Este sistema tiene el problema de no tener todo el histórico

de versiones, pero en este caso no tiene interés y se prefiere simplemente tener la nueva versión y la ultima que funcionaba por si la nueva falla y se quiere volver atrás.

Teniendo el ensamblado listo la siguiente parte es la comunicación del servidor de integración continua con el servidor de aplicaciones, esta acción se realiza en el paso 4, en el caso de los despliegues automáticos por commit, la aplicación se despliega en los servidores de pre-producción para poder ver si la aplicación funciona con el resto de componentes o el sistema deja de funcionar. Para que esta idea funcione se debe tener un entorno muy similar al productivo ya que se trata de la última línea de defensa, esto es porque el código que funciona en este entorno muy probablemente va a ser desplegado a producción, así que si el entorno es muy diferente al productivo se puede subir código que no funciona bien a producción. Respecto al despliegue en si, en primer lugar se debe tener en cuenta que el ensamblado se diseña de forma que acepta parámetros de configuración, eso quiere decir que no tiene la conexión con las bases de datos, ni otros servicio escritos directamente en el código, esta decisión se toma para garantizar que el mismo ensamblado que funciona en pre-producción es el que se va a pasar a producción. Es por ello que antes de mandar la orden de despliegue se debe obtener información de otro componente y en este caso es el servidor de variables de despliegue, este servidor a partir del nombre de la aplicación junto con el entorno a desplegar devuelve las cadenas de conexión necesarias.

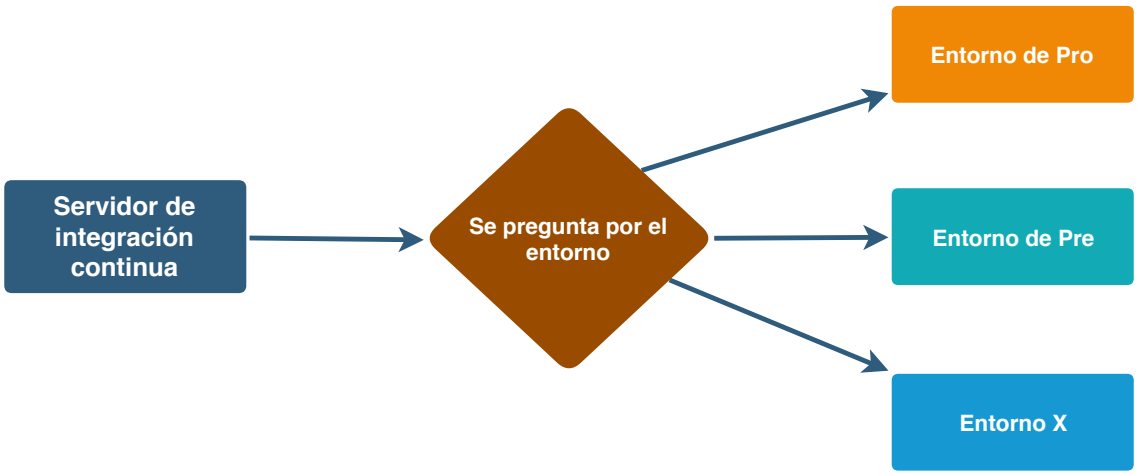

Figura 4.2: Esquema de múltiples entornos de despliegue

Teniendo toda la información necesaria el servidor de integración continua comunica al servidor de aplicaciones que se debe descargar la última versión del repositorio de artefactos, acción que corresponde con el paso 5 y desplegarlo con los parámetros

que le indica. El servidor de aplicaciones descarga dicho ensamblado y lo despliega delegándolo en algún servidor, ya que se trabaja con aplicaciones distribuidas, teniendo un cluster en vez de un único servidor, para garantizar disponibilidad. El servidor de aplicaciones, espera un tiempo marcado por proyecto, para una vez se entiende que el código ya debería estar desplegado y funcionando, realizar los test de integración, en este caso se lanza una aplicación que se descarga los test de integración de un repositorio común y los ejecuta, en el caso de pasarlos, se marca ese ensamblado como funcional y se sube al repositorio de artefactos para remplazar la ultima versión funcional, en caso negativo, no se hace nada y queda el código no funcional desplegado para poder analizar exactamente cual es el motivo aunque si se notifica a los programadores como se puede ver en el paso 6. En el caso de no saber solucionarlo siempre se puede volver a la ultima versión funcional, y poder investigarlo de forma más calmada. Aunque en pre-producción normalmente no importa que las aplicaciones no funcionen, ya que sirve como medio para probar las aplicaciones en real y ver exactamente que tal funcionan y que se debe de cambiar. Por último en el paso 7 el cliente podrá ver la nueva versión.

4.2. Distribución de fases

Una vez entendido todos los componentes analizaremos en la parte de integración continua como es exactamente un pipeline de trabajo.

Aunque el software y los desarrollos que se pueden realizar son muy diversos, trabajando con distintas tecnologías, lenguajes de programación y personas, es fundamental buscar una unificación de dichos flujos de trabajo.
Para ello se plantea un sistema que consiste en 4 fases que pueden aplicarse a cualquier desarrollo, son “build”, “deploy”, “test”,“notification”.

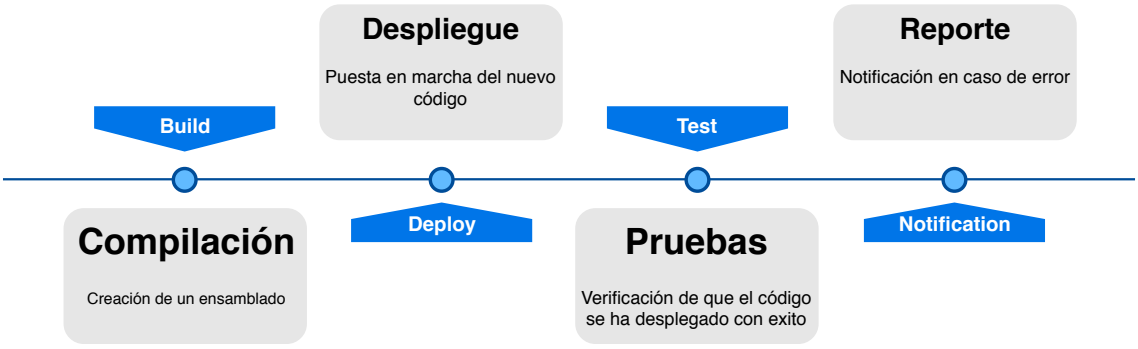

Figura 4.3: Mapa general de las fases

4.2.1. Fase de Build

Esta fase es la encargada de generar un ensamblado del código, normalmente esto se hará mediante un proceso de compilación con las herramientas que provee el lenguaje de programación o framework que se utilice.
Existen desarrollos que no requieren de un proceso de compilación como podrían ser los playbook de Ansible, para ello lo que se debe hacer es generar un comprimido del proyecto eliminando carpetas no esenciales para el funcionamiento como puede ser las generadas por herramientas de control de versiones.
Esta fase puede contener varias subfases que se deberán de solucionar de distintas maneras atendiendo a la tecnología que estemos tratando pero la idea general es la misma. Estas fases son "test unitario", "ensamblado", "publicación" y "análisis estático del código".

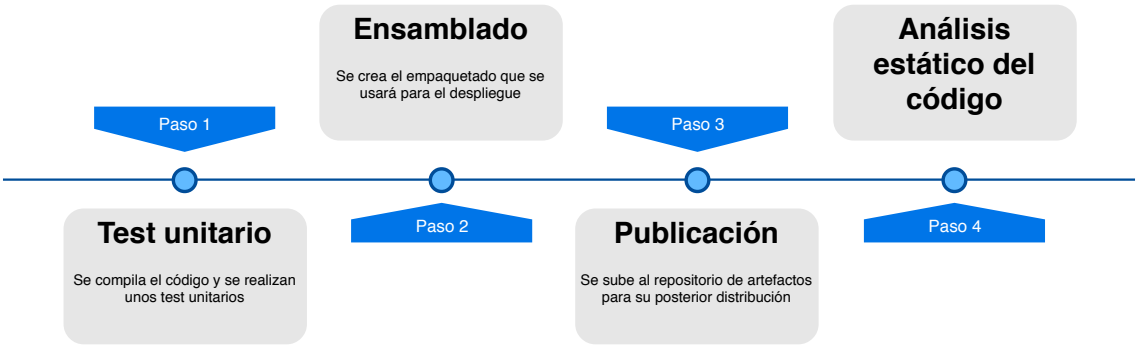

Figura 4.4: Subfases del paso de build

4.2.1.1. Test unitario

Antes de poderse crear un compilado es fundamental que se someta a unas pruebas para verificar que funciona de forma adecuada. Para evitar cualquier tipo de problema, los test no utilizan bases de datos centralizadas ni de ningún entorno, los test por si solos han de ser capaces de levantar sus propias bases de datos (usando docker) y en el caso de interacción con otros proyectos, sus llamadas será simuladas (mock). Una de las ventajas de plantear esto es que se unifica el poder probar el software, ya sea en un ordenador de desarrollo antes de subir el código, como en la máquina que compilará el código.
En general, cualquier proyecto que involucre desarrollo es susceptible de ser testado de una manera o de otra, por lo que esta subfase la podrán realizar todos los proyectos.

4.2.1.2. Ensamblado

El código se debe convertir a un formato entendible por el servidor de aplicaciones que además debe ser autocontenido para evitar la dependencia de otros componentes

y si necesitara de dichas dependencias irán empaquetadas de forma conjunta.

4.2.1.3. Subida a repositorio de artefactos

Una vez completado el test y el empaquetado es necesario archivar el compilado en algún punto para su posterior despliegue, esto es fundamental por varios motivos, en primer lugar la máquina que va a realizar el despliegue no tiene porque ser necesariamente la misma en la que se hace build, por lo que tener el ensamblado en un punto de acceso común permite la distribución de las tareas, otro factor importante es que el tener almacenados los diversos ensamblados que se han ido generados permite la vuelta atrás (rollback) sencilla independientemente del estado actual del código.
Un factor bastante importante a tener en cuenta es el uso de un sistema de versionado, el cual resulta fundamental si se quiere garantizar la vuelta atrás de cualquier servicio, así como identificar cada ensamblado. Para acometer esta tarea se puede recurrir a infinidad de sistemas, en primer lugar existe la opción manual, de por cada commit se suba la versión, el problema que tiene esto es que requiere bastante coordinación por parte de los programadores, ya que si dos suben personas suben la misma versión, una anula a la otra. Otra opción más automática es utilizar un sistema que etiquete por fecha, aunque es necesario hacer en algún punto una traza de cuales son las últimas versiones generadas para su posterior despliegue. Por último una opción que resulta muy interesante es un sistema simple de 2 versiones por entorno es decir, se tiene la versión actual y la anterior (funcional), esto permite que si una versión introduce algún problema se puede sustituir por una funcional rápidamente y al tener siempre los mismos nombres facilita bastante saber que versión es cual.

4.2.1.4. Análisis estático de código

Un software que funciona no significa necesariamente que esté bien programado, por diversos motivos, en primer lugar los test no son infalibles ya que no se analizan todos los posibles casos, por ello es conveniente revisar posible puntos que puedan generar fallos. Otro factor importante es la seguridad, un programa puede ser perfectamente funcional pero tener diversos agujeros de seguridad que pueden poner en compromiso tanto a clientes como a la propia empresa.
Esta subfase permite además una opción que puede ser muy útil para inculcar buenas practicas en los desarrolladores. Se trata de la cancelación del pipeline, es decir si se detecta que el código no tiene la calidad necesaria, se puede abortar su construcción y notificar tanto al programador como a las personas pertinentes.
La parte de cancelación requiere que la herramienta de análisis esté muy bien configurada para que no genere falsos positivos ya que perfectamente podría evitar el

despliegue de algo que en verdad no tiene ningún problema. La decisión de cancelación se puede hacer de diversas maneras o bien teniendo en cuenta el proyecto global, lo que se desaconseja para proyecto ya existentes, por el alto numero que puede tener de incidencias, que haría que no se puedan desplegar nuevas funcionalidades hasta solventar lo problemas, o bien la otra opción es tener en cuenta solo el código que se ha generado en el commit que se está desplegando, esto permite ir mejorando poco a poco los proyectos tanto nuevos como ya existentes.

4.2.2. Rollback

Esta fase, la cual no se realiza siempre, es la encargada de sustituir un compilado que no funciona por uno que si lo hace,esta fase es discutible si es necesaria o no, ya que el rollback se puede hacer a nivel de despliegue, pero en general no tiene ningún interés tener compilados que no funcionan subido en el repositorio así que este sistema es una manera de eliminar algo que no funciona, esta fase funciona muy bien con el sistema de dos versiones, donde se sustituiría la ultima versión por la anterior que si funciona, es cierto que con este sistema se realiza duplicación temporal pero se garantiza que cualquier versión descargada de ese repositorio es perfectamente funcional.

4.2.3. Fase de Deploy

El proceso de despliegue es algo que se ha de garantizar que se realiza siempre de forma correcta porque para según que sistemas puede ser muy perjudicial que esté parado, es por ello que se debe hacer de forma rápida y garantizando que incluso mientras se despliega se sigue dando servicio, intentando el llamado “zero-downtime deployment”. Aunque es cierto que esto no siempre es posible porque la tecnología no lo permite, pero lo que si se tiene que poder, es automatizar al máximo todo el proceso ya que todo procedimiento manual implica un posible error humano, mientras que algo automatizado, siempre que se haya preparado de forma correcta siempre generará el mismo resultado.
Al igual que en la fase de Build, esta fase cuenta con múltiples subfases.

4.2.3.1. Selección de entorno

En primer lugar es necesario saber en que entorno se va a desplegar, para poder obtener la información necesaria para el despliegue ya que las bases de datos cambian, rutas de accesos y otros posibles modificadores. Algo también fundamental es que el ensamblado sea parametrizable para en esta fase poder inyectarle los parámetros de despliegue adecuados.
Normalmente la obtención de el entorno de despliegue viene dado de manera manual

desde la herramienta de integración continua, aunque dependiendo del sistema que se haya escogido de uso de control de versiones, el entorno se puede obtener sabiendo en que rama nos encontramos.

4.2.3.2. Autorización

Aunque se busque la agilidad, también es fundamental garantizar que todo tiene un proceso de autorización para evitar que se despliegue sin el consentimiento de la persona a cargo. Es por ello que una vez se sabe a que entorno se quiere desplegar, se debe verificar si esa persona puede desplegar en ese entorno antes de dar paso al despliegue y en caso negativo se cancela, para a cometer esto existen diversas opciones. La primera y más simple es tener una tarea por entorno y dar permisos de ejecución a la personas pertinentes, esto puede resultar muy engorroso si existen muchos proyectos y muchos entornos ya que se pierde esa visión de conjunto. La otra alternativa es una tarea única que tenga algún mecanismo para saber si el usuario en cuestión puede desplegar en ese entorno y si es correcto lo permite, además esto permite registrar intentos de despliegue no autorizados. Por ultimo la opción más interesante es una variante de la anterior y dado el caso de que alguien despliegue en un entorno donde no tiene suficientes permisos, el despliegue queda a la espera de que alguien lo autorice, esta opción permite más agilidad en las comunicaciones, en primer lugar la persona que sube la funcionalidad no necesita comunicarla directamente enviando un correo o con otro sistema, y la persona que lo autoriza es notificado en el momento en el que se ha verificado ya que ese ensamblado está listo para ser desplegado.

4.2.3.3. Despliegue

La subfase encargada en si de ejecutar la puesta en marcha del nuevo código, dependiendo de la tecnología utilizada tendrá un sistema u otro, pero la idea principal es comunicarse con la maquina o máquinas en las que se va a desplegar, y parar lo que está funcionando y poner en funcionamiento el nuevo código, siendo esta parte critica, ya que una mala configuración puede hacer que se dejar el servicio caído. Para garantizar que esto no ocurra existen opciones de comprobar la salud del servicio y en caso de que no levante, se utilice la versión anterior de forma automática.

4.2.4. Test

Un programa rara vez será un ente único que no depende de nada, es por ello que se debe verificar que los cambios introducidos no rompen compatibilidad con ningún sistema o bien no estaba todo preparado para que se desplegara la nueva versión. La forma más sencilla de realizarlo es aplicando los test de integración que

nos permite claramente determinar si las funcionalidades de la aplicación funcionan conectándose a todas la componentes reales y no realizando mocks.

4.2.4.1. Rollback

Como en las demás fases, en caso de error siempre existe la opción de realizar una vuelta atrás automática cancelando todo lo anterior, dando tiempo al programador a revisar el motivo del fallo sin dejar no operativo el servicio. Este apartado como funciona exactamente es conectándose al repositorio de artefactos y remplazando la ultima versión, por la versión marcada como funcional, eliminando la verisón que no funciona porque no tiene ningún interés almacenarla.

4.2.5. Posibles mejoras: Actualización en cascada

La base principal del pipeline es la descrita anteriormente, pero dependiendo de la arquitectura que se utilice puede resultar interesante mejorar el pipeline con nuevas funcionalidades, en el caso de la arquitectura de microservicios, existe una que puede resultar muy interesante, y es la actualización en cascada. En el caso de los microservicios pueden existir dependencias entre ellos, por lo que igual para implementar una nueva funcionalidad se necesita modificar dos pero por la naturaleza del cambio, puede resultar en versiones no retro compatibles, por lo que una vez se actualiza un microservicio se deberá actualizar el otro, pero esto plantea un problema, si dados dos microservicios A y B, el A se despliega de forma correcta pero el B no, al hacerse un rollback automático quedarían tanto el A y el B en un estado no funcional, además hay que tener en cuenta que el A al depender del B cuando se desplegase, este fallaría al no poder pasar los test por su dependencia con B. Todo esto hace que automatizar este proceso sea muy complejo y solo es recomendable intentar automatizarlo una vez se tiene madurado el entorno. Para este tipo de situaciones lo más recomendable es fijar una ventana de mantenimiento y realizar un despliegue con supervisión manual, y si falla algo, se ralizarán de forma manual los rollback necesarios.

Capítulo 5

Implementación y casos de uso

Todas los apartados vistos anteriormente se han explicado de forma general, pero para poder llevarlo a un terreno concreto, se verán distintos ejemplos reales, en los que aparecen distintas tecnologías cada una con sus peculiaridades, las cuales obligan a adaptar algunos aspectos, por la naturaleza de la tecnología.
A continuación se pueden ver cuatro apartados, en primer lugar se explicará la implementación concreta de los conceptos y las tecnologías escogidas, seguido por un esquema del hardware que se utiliza, también se expondrá como son las interfaces que permite controlar todo y por ultimo dos casos de uso para ver todas estas componentes en funcionamiento.

5.1. Componentes de la integración continua

Anteriormente se ha analizado la idea general de cómo es la arquitectura que se ha diseñado para permitir el despliegue continuo. En este apartado se verán todas las herramientas concretas que se utilizan y como se concreta cada una para llegar desde el programador hasta tener una aplicación desplegada.

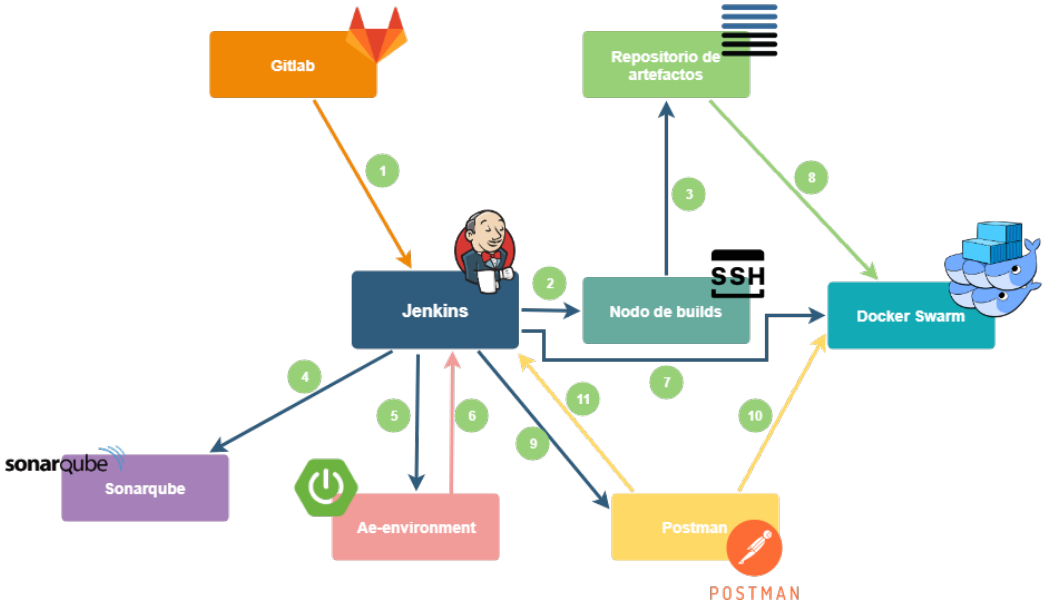

Figura 5.1: Esquema de implementación de la arquitectura CI

La figura del programador, es el punto de partida para iniciar todas las componentes siguientes una vez realiza un commit. Las herramientas con las que cuenta son en primer lugar con un portátil equipado con linux, esto se debe a que se llegó al consenso que para el tipo de tecnologías que se iban a trabajar era la herramienta más adecuada, respecto al IDE se utilizan diversos dependiendo del entorno que se trabaje, eclipse, vscode, intellij.

El programador normalmente trabaja directamente con la consola de comandos para hacer la subida de código, esto se hace para que el programador practique con git y que entienda los comandos y qué está haciendo, porque si se usa una interfaz gráfica se pueden llegar a resultados no deseados por las acciones que hace por debajo la herramienta, además al trabajar sin ramas los comandos a tratar no son muy complejos.

5.1.1. Repositorio gitlab

Propiedades	Descripción
Componente	Repositorio de código
Objetivo	Albergar el código de los programadores, para garantizar su versionado y acceso desde otras tecnologías
Tecnología seleccionada	Gitlab CE
Justificación	Es un producto opensource con infinidad de opciones e integraciones con otras plataformas
Instalación	Uso de la imagen oficial de Docker
Alta disponibilidad	Actualmente no y en un principio no está planificado
Interacción con otras tecnologías	Interactúa con Jenkins mediante el paso 1 que es el envió de una notificación de descarga del código mediante webhooks
Comunicación con el programador	Presenta una interfaz en la que ver el código subido y quien lo ha modificado e incluso editarlo

El servidor de Gitlab, se encuentra desplegado como contenedor de docker y actualmente cuenta con soporte HA, aunque se estudia la posibilidad para garantizar una alta disponibilidad porque se trata de una parte importante del desarrollo. De todas maneras en el caso de no estar disponible por una caída total del servidor, el tiempo en que se tarda en desplegar una nueva instancia y cargar la copia de seguridad, entra dentro de los margenes aceptables de tiempo, no tardando más de media hora. Es por ello que aunque es interesante dar disponibilidad el no hacerlo no supone un problema muy exagerado dado el numero de trabajadores actuales del departamento.

Gitlab cuenta con muchas opciones las cuales, no se utilizan como puede ser su repositorio de artefactos, su servidor de integración continua o la gestión de un cluster de kubernetes, funcionalidades que se fueron metiendo a medida que se desarrolló este documento y no estaban cuando se diseñaron algunas partes. Es por ello que lo único que se hace con gitlab es almacenar el código. Cuando se detecta un commit automáticamente utilizando los llamados webhooks, informa a jenkins de que debe de iniciar el proceso de integración continua.

5.1.2. Servidor de integración continua Jenkins

Propiedades	Descripción
Componente	Servidor de integración continua
Objetivo	Gestionar todos los procesos necesarios para la correcta compilación del código así como ejecutar las pruebas y desplegarlo. También debe comunicarse con el resto de herramientas que darán más información sobre el producto software
Tecnología seleccionada	Jenkins
Justificación	Es una de las herramientas de integración continua más utilizada y permite la instalación de infinidad de plugins para añadir o mejorar funcionalidades. Incorpora un sistema de librerías para poder evitar la repetición de código.
Instalación	Uso de la imagen oficial de Docker
alta disponibilidad	No, por falta de soporte pero se tienen desplegados múltiples esclavos, garantizando la ejecución de las tareas incluso con la perdida de un esclavo.
Interacción con otras tecnologías	Es el punto con más interacciones, recibe el código de gitlab en el paso 1, en el paso 2 ordena la compilación y ejecución de test unitarios, en el 4 manda analizar el código a Sonarqube, en el 5 y en el 6 ocurre la comunicación con ae-environment para obtener las variables de despliegue, en el 7 despliega contra el swarm de docker y finalmente en el 9 y 11 realiza la comunicación con postman para probar el ensamblado que se ha desplegado
Comunicación con el programador	Comunica al programador de posibles fallos mediante el correo y con herramientas de mensajería instantánea, ofrece además una interfaz en la que visualizar el histórico de los trabajos así como poder ejecutarlos de nuevo.

Jenkins se encuentra desplegado como contenedor de Docker y al igual que Gitlab no existe ningún tipo de montaje HA, pero en este caso es debido a que Jenkins está diseñado para tener exclusivamente un master, no tiene ningún tipo de sistema HA ya fabricado, en este caso normalmente se opta por la vía de lanzar una nueva instancia y cargar los datos de un volumen en red si se quiere rapidez o bien al igual que gitlab cargar la copia de seguridad. La unión con Gitlab se realiza con el plugin de Gitlab para Jenkins el cual genera una url donde gitlab puede mandar la información del proyecto mediante webhooks para que se inicie el pipeline correspondiente. De aquí en adelante Jenkins es el encargado de coordinar el resto de componentes para llegar a tener desplegada la aplicación en cuestión.

5.1.3. Repositorio de artefactos Nexus

Propiedades	Descripción
Componente	Repositorio de artefactos
Objetivo	Albergar los ensamblados de código para su posterior despliegue.
Tecnología selecionada	Sonatype Nexus 3
Justificación	Es una solución opensource que tiene las funcionalidades necesarias y cumple perfectamente con su objetivo.
Instalación	Uso de la imagen oficial de Docker
Alta disponibilidad	No y no se prevé implementarlo.
Interacción con otras tecnologías	Recibe el ensamblado del nodo de builds en el punto 3 y en el 8 se lo envía a Docker swarm para su despliegue
Comunicación con el programador	No comunica nada al programador, pero cuenta con una interfaz en la que ver los ensamblados disponibles y operar sobre ellos.

El encargado de almacenar todos los ensamblados es nexus, el cual estaba desplegado directamente instalado en un servidor pero se decidió mover a contenedor docker para facilitar las actualizaciones y la posibilidad de moverlo de maquina. En este caso no se realizan copias de seguridad de los ensamblados porque el perderlos no supone ningún problema más allá de tener que volver a compilar los proyectos.

La interacción que tiene con Jenkins la hace de distintas maneras, en el caso de imágenes docker, actúa como registry y trabaja como si de DockerHub se tratase. Para los proyectos de Java que no utilizan imágenes de Docker, funciona como repositorio Maven. Para las librerías propias de angular, actúa como repositorio de npm y por ultimo para todos los proyectos que no encajan en ninguna categoría tiene una API rest en la que permite subir cualquier tipo de archivo.

5.1.4. Servidor de Build

Propiedades	Descripción
Componente	Servidor de builds
Objetivo	Ejecutar las tareas de compilación y test unitarios
Tecnología selecionada	Centos 7
Justificación	El tener un servidor dedicado a builds garantiza la dedicación de recursos a una tarea concreta
Instalación	Instalado en un servidor dedicado
Alta disponibilidad	Actualmente solo existe un nodo de builds pero se estudia tener varios para dar mayor disponibilidad y tener más recursos
Interacción con otras tecnologías	Recibe el código de Jenkins en el paso 2 y envía el ensamblado a Nexus en el paso 3
Comunicación con el programador	No tiene ninguna comunicación con el programador.

Aunque Jenkins se puede utilizar directamente para compilar los proyectos, se utilizan otros ordenadores como esclavos, aunque en el caso particular de las compilaciones, no se utiliza directamente la opción de esclavo de jenkins habitual si no que existe un nodo el cual por ssh recibe las ordenes necesarias para compilar los proyectos. En el caso de los proyectos de microservicios lo hace con maven directamente instalado en la maquina y en el caso de angular, recurre a una imagen de Docker.

En este apartado se podría cambiar la manera de trabajo pero como el proceso de build necesita de crear contenedores docker se decidió no dockerizar esta parte por no complicarla, aunque siempre se estudia la posibilidad de aplicar mejoras en este apartado.

5.1.5. Servidor de configuraciones

Propiedades	**Descripción**
Componente	Servidor de configuraciones
Objetivo	Albergar los datos de despliegue para los distintos entornos
Tecnología seleccionada	Spring boot + Mongodb
Justificación	Al ser una pequeña utilidad se optó por un desarrollo interno para poder personalizarla totalmente a los proyectos y el sistema de integración continua
Instalación	Desplegado en el swarm de Docker
Alta disponibilidad	La parte de base de datos está montada como replicaset de 5 nodos garantizando que el sistema funciona con varios nodos caídos, además al estar desplegado el servicio en si en un swarm si una instancia falla se levanta otra.
Interacción con otras tecnologías	Recibe la petición de información de un entorno concreto por parte de jenkins en el paso 5 y en el 6 se la proporciona.
Comunicación con el programador	No tiene ninguna comunicación con el programador aunque existe una herramienta gráfica para introducir la variables.

Existe un servicio en la arquitectura, ae-environment, cuyo cometido es almacenar, las configuraciones de despliegue de otros servicios y páginas web. Tiene expuesta una sencilla api rest, en la que dado el nombre del servicio junto a su entorno, podemos obtener su configuración. Este servicio funciona almacenando la información en una base de datos mongo, se decidió hacerlo así ya que la información a almacenar es muy heterogénea.
Este servicio está desplegado como contenedor docker y de la misma manera que cualquier otro servicio ya que es un desarrollo interno.

5.1.6. Servidor Sonarqube

Propiedades	Descripción
Componente	Servidor de análisis estático de código
Objetivo	Realizar escáneres del código en busca de posibles fallos, malas practicas, partes de código de difícil comprensión, código duplicado o partes de sin cobertura de test
Tecnología seleccionada	Sonarqube
Justificación	Es una herramienta opensource que tiene diversos plugins para la gran mayoría de lenguajes utilizados en la industria y su incorporación al sistema de integración continua es sencilla
Instalación	Uso de la imagen oficial de Docker.
Alta disponibilidad	No y no se tiene interés al no ser un punto critico, ya que actualmente solo se utiliza con carácter informativo.
Interacción con otras tecnologías	Recibe la orden de analizar el código en el paso 4
Comunicación con el programador	Presenta una interfaz en la que consultar los resultados pero no realiza ninguna comunicación directamente al programador para informar de los resultados

Para poder ir verificando la calidad de los proyectos se decidió colocar un servidor Sonarqube, cuya misión es analizar el código de los servicios, esto se hace con un plugin de maven que permite seleccionar un servidor externo para realizar un análisis. El servidor se encuentra dockerizado y no tiene ningún tipo de HA y no se espera poner ya que, actualmente solo se utiliza con carácter informativo.

5.1.7. Herramientas de test

Propiedades	Descripción
Componente	Servidor de aplicaciones
Objetivo	Realizar los test de integración sobre el ensamblado desplegado en los servidores
Tecnología seleccionada	Postman
Justificación	Es la tecnología por excelencia para realizar los test de aplicaciones que exponen APIs REST
Instalación	Desplegada una imagen propia con la utilidad Newman en los diferentes swarms
Alta disponibilidad	No aunque se puede idear algún sistema para lanzar instancias a demanda
Interacción con otras tecnologías	Recibe la petición de ejecutar los test por parte de Jenkins en el paso 9, en el 10 los ejecuta contra el Swarm y en el 11 indica los resultados a Jenkins
Comunicación con el programador	No avisa al programador de ningún evento aunque se puede utilizar la aplicación de escritorio para confeccionar los tests

Para la parte de test, como es necesaria la comunicación directa con el servicio, algo que normalmente no se tiene por estar en redes aisladas sin puertos abiertos, se diseñó un esclavo de jenkins dockerizado, el cual reside en la misma red que todos los servicios y si tiene visibilidad y comunicación a ellos. En primer lugar nación para ejecutar newman sin tener que abrir puertos, pero luego se utilizó para las webs siguiendo el mismo principio.

5.1.8. Docker swarm

Propiedades	Descripción
Componente	Servidor de aplicaciones
Objetivo	Ejecutar las aplicaciones para poder ser consumidas por los clientes
Tecnología seleccionada	Docker swarm
Justificación	Es la herramienta nativa que ofrece Docker para montar clusteres de contenedores Docker distribuidos
Instalación	Instalación de múltiples nodos con Docker instalado
Alta disponibilidad	Si , cuenta con una solución con varios maestros que garantizan que aunque se caiga uno el otro puede seguir distribuyendo tareas entre los trabajadores
Interacción con otras tecnologías	Recibe la orden de despliegue por parte de Jenkins en el paso 7, en el 8 recibe el ensamblado al desplegar y por ultimo postman realiza los test contra el en el paso 10
Comunicación con el programador	No avisa al programador de ningún evento aunque existe una interfaz gráfica para visualizar su estado

Se tienen dos swarms, uno de pre y otro de pro, que son el punto donde se despliega todos los desarrollos, actualmente tienen múltiples nodos, pero los manager no están replicados por que no se trabaja a día de hoy con VIPs, que son ips compartidas por múltiples servidores. Lo que si tiene replicación son los trabajadores, por lo que si un nodo cae, sus servicios se balancean a otra maquina con una perdida de disponibilidad mínima. Por norma general los servicios no está abiertos al exterior y para poder acceder a ellos se diseñaron dos gateways, uno con seguridad y otro sin seguridad el cual se utiliza exclusivamente para consumo interno.

Los swarm están montados sobre maquinas físicas sin virtualizar, aunque se estudia la posibilidad de virtualizar esta infraestructura, la comunicación con ellos se hace mediante una api rest. El acceso a los swarm para tareas de gestión solo está disponible desde la red interna y en ningún caso desde el exterior.

5.2. Hardware

A la hora de implementar todas estas tecnologías es necesario tener una infraestructura hardware que la soporte, en este caso contamos con los siguientes equipos

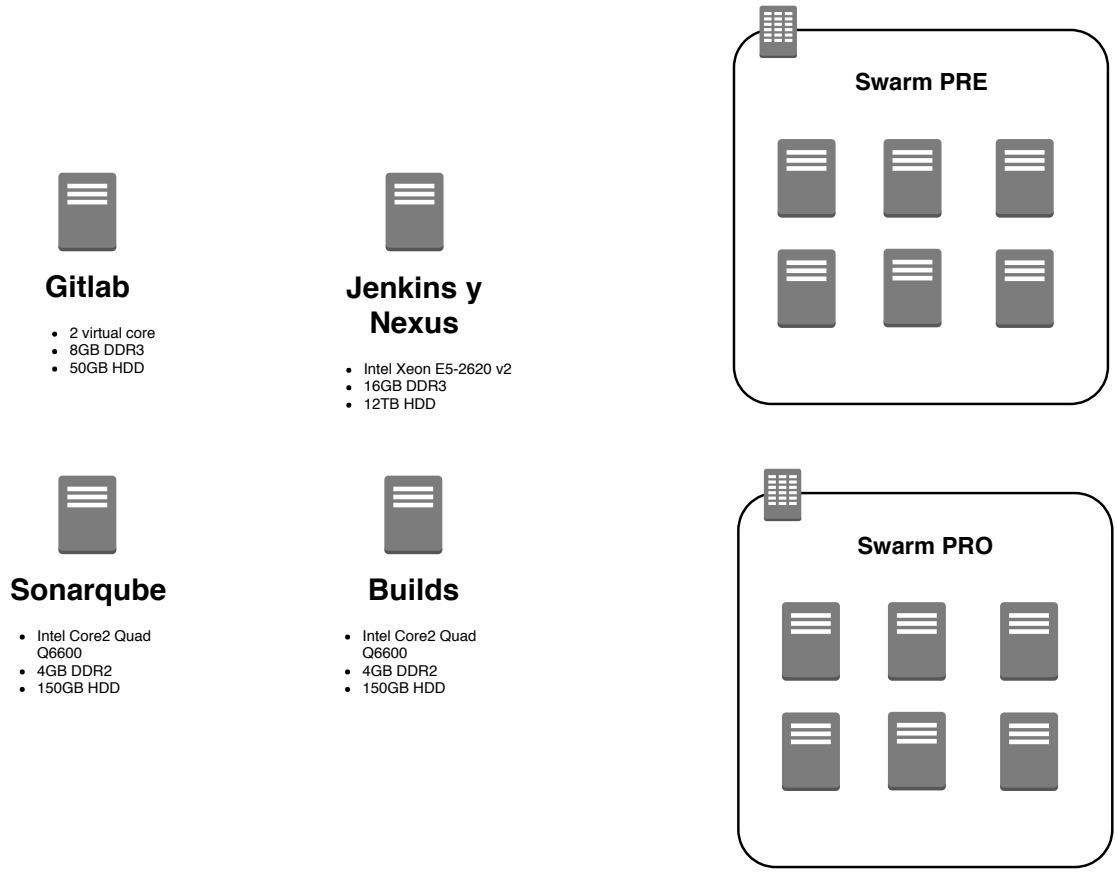

Figura 5.2: Esquema del hardware utilizado

Tecnología	CPU	RAM	Disco	Red	Uds
Jenkins y Nexus	Intel Xeon E5-2620 v2	16GB DDR3	12TB	Intel I350 Gigabit Network Connection	1
Gitlab	2 Virtual core	8GB DDR3	50GB	1Gb	1
Sonarqube	Intel Core2 Quad Q6600	4GB DDR2	150GB	Broadcom Limited NetXtreme BCM5754 Gigabit Ethernet	1
Builds	Intel Core2 Quad Q6600	4GB DDR2	150GB	Broadcom Limited NetXtreme BCM5754 Gigabit Ethernet	1
Swarm PRE	Intel(R) Core(TM)2 Duo CPU E7500 @ 2.93GHz	4GB DDR3	160GB	NetLink BCM57780 Gigabit Ethernet PCIe 1Gbit/s	1
Swarm PRE	Intel Core i3-2100	4GB DDR3	250GB	Realtek RTL8111/8168/8411 PCI Express Gigabit Ethernet	1

Swarm PRE	Intel Core i3-2120	4GB DDR3	250GB	Realtek RTL8111/8168/8411 PCI Express Gigabit Ethernet	2
Swarm PRE	Intel Core2 Duo E7400	4GB DDR2	160GB	Broadcom Limited NetXtreme BCM5754 Gigabit Ethernet	1
Swarm PRE	Intel Core2 Quad Q9300	4GB DDR2	80GB	Broadcom Limited NetXtreme BCM5754 Gigabit Ethernet	1
Swarm PRO	2xIntel Xeon E5260v1 @2.4Ghz	8GB DDR3	150GB	Broadcom Limited NetXtreme II BCM5716 Gigabit Ethernet	1
Swarm PRO	Intel Xeon X3323 @2.5Ghz	16GB DDR2	300GB	Broadcom NetXtreme BCM5722 Gigabit Ethernet	1
Swarm PRO	Intel Xeon X3360	4GB DDR2	160GB	Broadcom Limited NetXtreme BCM5721 Gigabit Ethernet	2
Swarm PRO	Intel Core2 Quad Q6600	4GB DDR2	150GB	Broadcom Limited NetXtreme BCM5721 Gigabit Ethernet	2

5.3. Interfaces

5.3.1. Interfaz de Jenkins

La web de Jenkins sirve como punto de control de todos los procesos de integración continua y será el lugar donde los programadores podrán verificar el estado de sus proyecto, lanzarlos manualmente si fuera necesario o bien revisar el estado de un despliegue en ejecución.

En primer lugar aparece una vista en la que se pueden ver todos los proyectos que podemos desplegar.

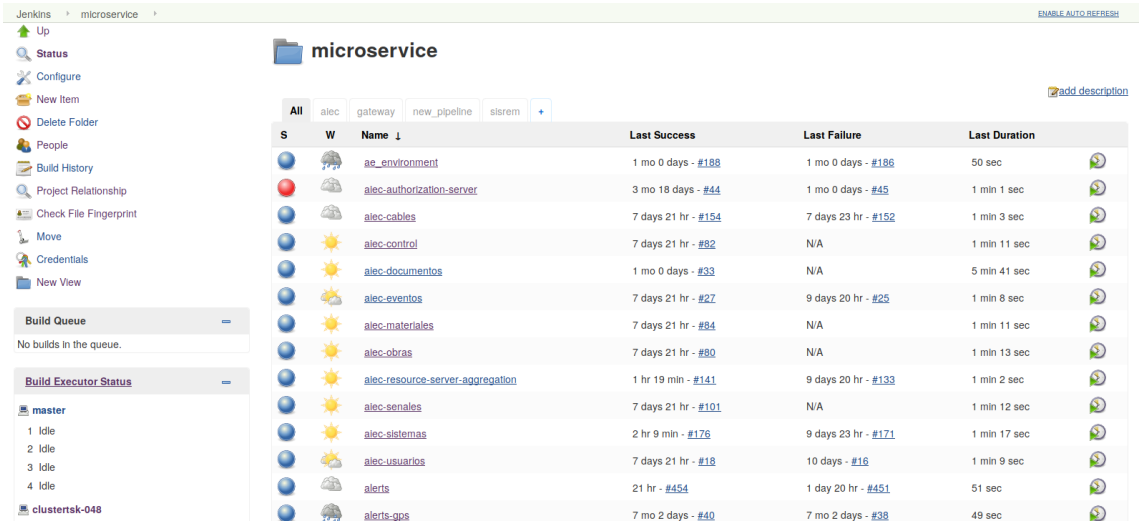

Figura 5.3: Vista general de la carpeta microservices

Aunque las tareas son idénticas gracias al sistema de librerías se decide tenerlas separadas para no mezclar información y poder hacer permisos más granulares si fuera necesario.

La vista de una tarea permite ver las ejecuciones recientes, el resultado de los test, que fases ha hecho, configurar la tarea o lanzarla.
Como se puede ver en la figura, las tareas fallidas aparecen en rojo, y los pasos que no se ejecutan aparecen vacíos para que el programador de un vistazo vea exactamente que pasos se han ejecutado.

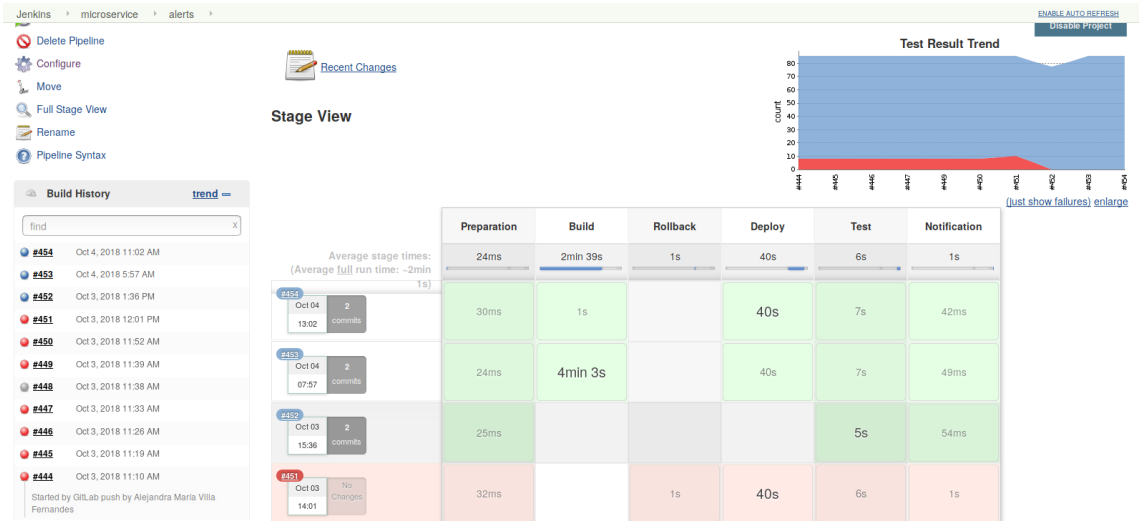

Figura 5.4: Vista de la tarea alerts

A la hora de lanzar la tarea aparecen varias opciones que permiten configurar que tareas se van a realizar y sobre que entorno.

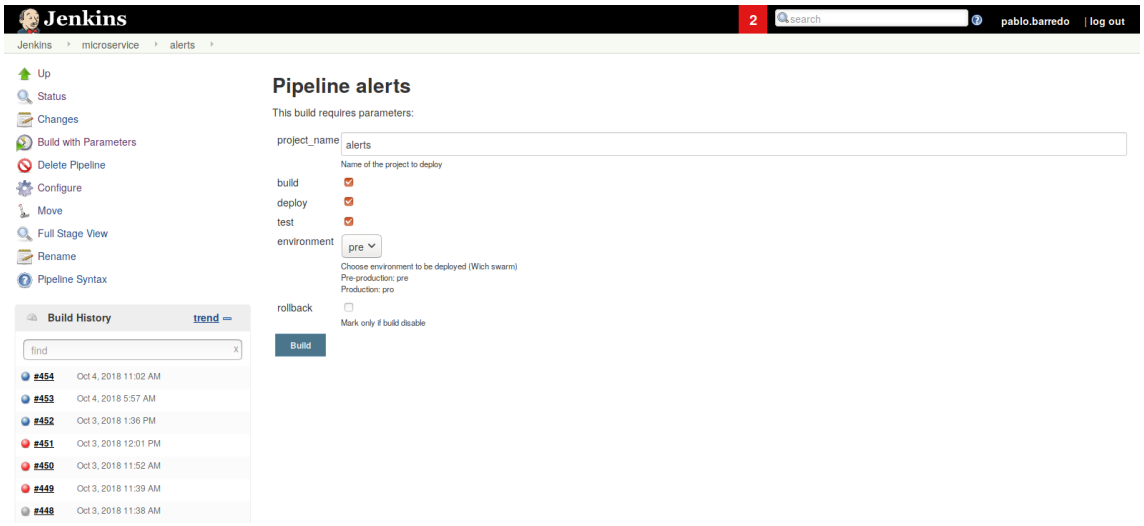

Figura 5.5: Vista de la tarea alerts a la hora de ejecutarla

Las tareas se pueden configurar por si necesitaran parámetros extra tal y como aparece en la siguiente figura.

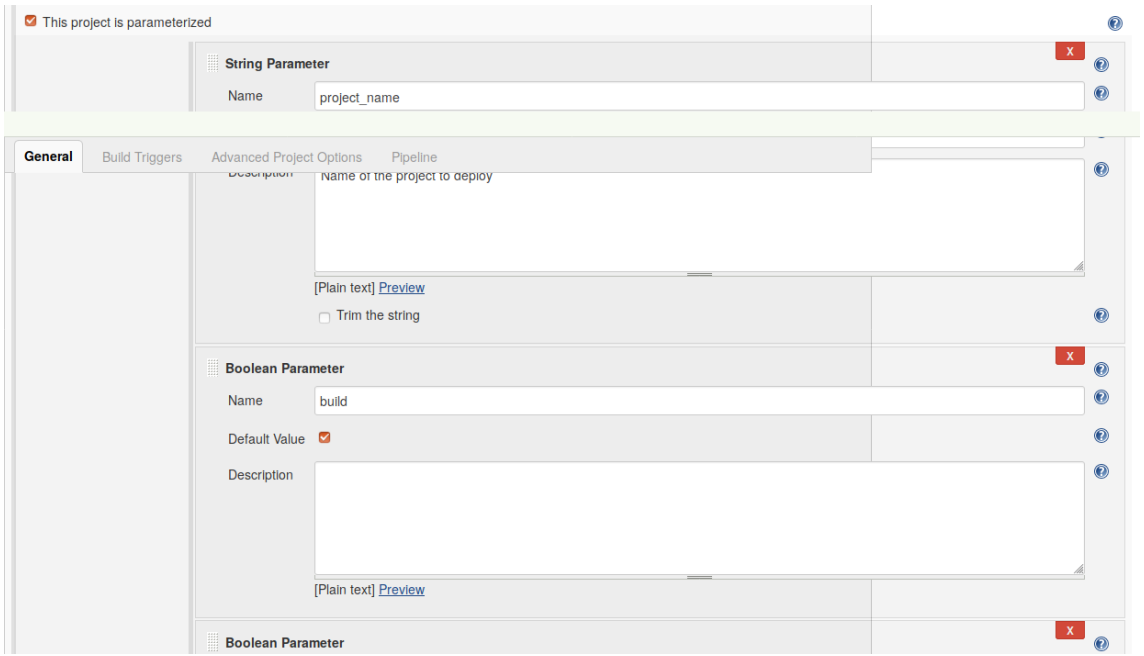

Figura 5.6: Parametros de la tarea

Los entornos al poder variar entre proyectos son perfectamente configurables en cada tarea como se puede ver en la figura.

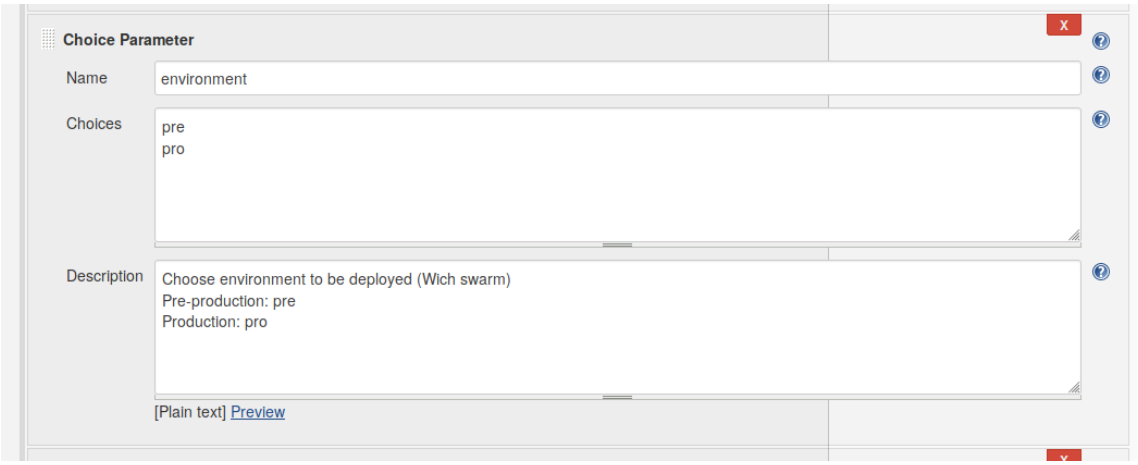

Figura 5.7: Configuración de los entornos

El enlace con Gitlab se hace con un parámetro de configuración que da una url webhook que debemos configurar en el servidor Gitlab, una vez enlazamos las dos herramientas, cada vez que se realice un commit se lanza la tarea con los parámetros por defecto.

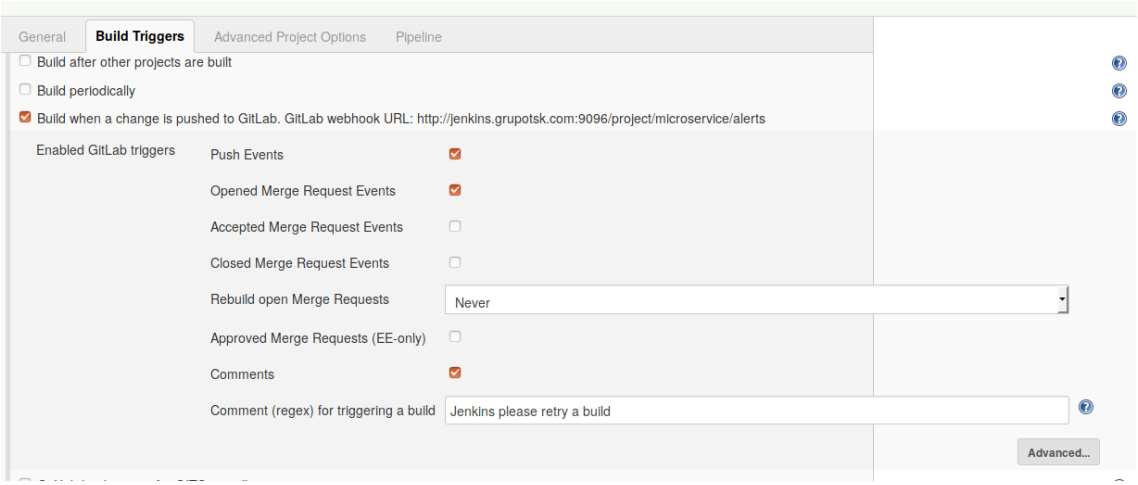

Figura 5.8: Unión con Gitlab

En caso de error Jenkins envía un correo al programador indicando cual ha sido el fallo y si ha sido capaz de ejecutar los test, indica cuales han provocado el fallo.

BUILD FAILURE

URL:	http://jenkins.grupotsk.com:9096/job/microservice/job/alerts/451/
Project:	alerts
Date:	Wed, 03 Oct 2018 12:01:20 +0000
Duration:	49 sec and counting
Cause:	Started by user pablo.barredo

CHANGES

No Changes

Test Results

Name	Failed	Passed	Skipped	Total
(root)	10	32	0	42
Alerts / GetAlertsSummarizedWithStatusAndType.result is one (Age: 1)				
Alerts / GetAlertsSummarizedWithType.result is one (Age: 1)				
status / 002_putAlertsStatus.204 code put (Age: 8)				
status / 003_getAlertsStatus.body contains key (Age: 8)				
status / 005_getAlertsStatus.body contains key (Age: 8)				
status / 006_deleteAlertsStatus.200 code delete (Age: 8)				
type / 002_putTypeAlerts.204 code put (Age: 8)				
type / 003_getAllTypeAlerts.body contains UUID (Age: 8)				
type / 005_getTypeAlertList.body contains UUID (Age: 8)				
type / 006_deleteTypeAlerts.200 code delete (Age: 8)				
notes / 001	0	1	0	1
notes / 002	0	4	0	4
notes / 003	0	4	0	4
notes / 004	0	4	0	4
notes / 005	0	7	0	7
notes / 006	0	4	0	4
notes / 007	0	3	0	3
notes / 008	0	8	0	8
notes / 009	0	4	0	4
notes / 010	0	4	0	4

Figura 5.9: Ejemplo de correo fallido

5.4. Casos de uso

5.4.1. Microservicios

En primer lugar se tratará una de las piezas más importantes con las que se trabaja en la empresa, se trata del componente de "servicio", en este caso particular se utiliza el concepto de microservicio, que consiste en evitar crear grandes aplicaciones monolíticas en las que una mismo servicio hace muchas acciones haciendo cada vez más complejo su desarrollo, la idea del microservicio es reducir su funcionalidad al mínimo teniendo una pequeña aplicación, que puede ser instanciada múltiples veces si se necesita de mayor capacidad de computo. Algo también importante en nuestro caso es que se intenta que los componentes sean reutilizables, por lo que si por ejemplo se tiene un microservicio de alertas, este ha de poder ser utilizado en múltiples proyectos, consiguiendo así evitar ir replicando funcionalidades en distintos proyectos, ya que con un estilo tradicional si se quisiera hacer una aplicación de monitorizar paneles fotovoltaicos, crearíamos una aplicación que además de su lógica particular, tendrá un modulo de alertas, pero si además hacemos otro proyecto que monitoriza gaseoductos, pues tendríamos que implementar la lógica propia y además otro modulo nuevo de alertas, duplicando así código.
Desacoplando este modulo, conseguimos poder usarlo en dos proyectos y garantizamos que módulo funciona por si mismo.

5.4.1.1. Maven

En primer lugar en esta implementación concreta, tenemos como participante principal el gestor, maven, que nos permite descargar las dependencias necesarias que necesita el proyecto, y ejecutar tanto los test unitarios como la fase de compilación, este proceso se deja definido directamente en el archivo pom de cada proyecto, se podría decir que cada proyecto de maven tiene internamente unas instrucciones que le dicen que es exactamente la fase de compilar, que normalmente es la ejecución, de los test unitarios y en caso de pasar, realiza el empaquetado en imagen de docker.

5.4.1.2. Fase de preparación

En primer lugar antes de iniciar el pipeline debemos realizar unas tareas de obtención de datos e inicialización. Para poder aplicar tareas de analítica, se ha montado un tópico de Kafka en el que podemos inyectar los resultados de los despliegues. Es por ello que en primer lugar averiguamos quien es el responsable del despliegue y se decide a que tópico se tiene que mandar la información, en este caso particular se trata de "jenkinsbuildlogs", la información que mandamos es la siguiente:

- Fecha de envió del mensaje (23/07/2018T06:19:49.229Z)
- Desarrollador (pablo.barredo@****.com)
- Nombre del proyecto (alerts)
- Tipo de componente (microservice)
- Entorno (pro)
- Fase (Deploy)
- Resultado (SUCCESS)
- Tiempo de ejecución (50.0)

Otro aspecto es la obtención de información única del proyecto, como puede ser el repositorio en el que está almacenado, donde tiene los test de postman, nombre de la imagen que se ha de generar, etc.

Esta fase es fundamental ya que permite que todas las tareas partan de una plantilla ya que si no se optara por esta opción tendríamos n tareas iguales pero con los parámetros cambiados, siendo un cambio una tarea tediosa al tener que actualizarlo en todos los proyectos. El uso de plantillas a veces supone un problema si se quiere

hacer algo distinto en solo un proyecto porque se debe de adaptar la plantilla para albergar ese posible caso.

5.4.1.3. Fase de build

En este apartado lo que se hace es ejecutar una tarea auxiliar que utiliza un servidor externo de compilación donde se realizará el proceso de test, aunque normalmente solo se ejecutan test unitarios lo que si se realiza son test con bases de datos reales, para ello con la ayuda de contenedores docker se levanta una base de datos con la información de test. La parte de conexión con otros servicios no es real y se simula las respuestas que debería recibir. Si el test unitario falla se aborta toda la ejecución y se genera un mensaje de error indicando que se ha producido un fallo en build y se comunica a los programadores responsables de este proyecto. En el caso de funcionar se realiza la parte de ensamblado,m en este caso lo que se hace es la generación de una imagen de docker que será el elemento que será el elemento que posteriormente se desplegará. El dockerfile en este caso si es único para cada proyecto, aunque si es cierto que se puede externalizar y estar parametrizado, si se opta por tenerlo repetido para que el desarrollador pueda hacer la fase de compilación en local. Una vez se tiene el ensamblado se debe de subir a nexus, remplazando la ultima versión, para poder desplegarla posteriormente.

El caso de la compilación en pro es distinta a la de pre, porque no se realiza, en el caso de los microservicios, como el ensamblado está perfectamente parametrizado, lo que se realiza es simplemente es etiquetar la imagen de dopcker de pre como la de pro, haciendo que cuando llegue la fase de despliegue se despliegue este nuevo ensamblado que ya se sabe que pasa el pipeline completo.
También destacar que la fase de build en pro no la pueden realizar todas las personas y si lo realiza una persona que no tiene permiso para ello, el pipeline falla y lo comunica a los responsables del microservicio.

5.4.1.4. Fase de deploy

Este apartado es el encargado de realizar el despliegue del ensamblado generado en la fase de build. Para ello en primer lugar debe de averiguar en que entorno lo tiene que averiguar, esto lo recibe como parámetro, toda esta información ya fue recabada en la fase de preparación, ya que la tarea tiene un selector de entornos. Conociendo que servicio tiene que desplegar y en que entorno realiza una llamada rest al microservicio “ae-environment”, el cual le devolverá las variables de entorno que indicará al servidor de aplicaciones. La ventaja de este sistema es que se puede llamar por separado a esta tarea y cambiar sin necesidad de una nueva compilación, la url de bases de datos, el nivel de log u otros apartados de la aplicación como

podría ser una funcionalidad, que se tiene preparada para ser activada solo en un entorno especifico.
Respecto a la comunicación con el servidor de aplicaciones, se tiene un swarm de docker, el cual expone una api rest donde se le debe indicar, el nombre del servicio a desplegar, la imagen que debe de utilizar y parámetros de configuración como puede ser la memoria RAM que debe de consumir cada contenedor como máximo, numero de replicas y las variables de entorno, este ultimo apartado es fundamental porque es donde le pasaremos las conexiones de bases de datos y demás datos únicos del entorno.

Docker swarm permite indicar que estamos realizando una actualización del servicio, pero en esta implementación concreta, se opta por primero destruir la versión que está desplegada, si la hubiere, y posteriormente decir que despliegue un nuevo servicio, esto en parte se hace por el sistema de versionado escogido, perfectamente se podría cambiar pero actualmente no se contempla.

La fase de despliegue termina una vez ha pasado el tiempo que el programador ha fijado previamente que tarda en desplegar su aplicación, en esta fase no se comprueba que lo desplegado funciona ya que para eso existe la fase de test, lo máximo que se hace es esperar a que pase el tiempo necesario para considerar que esa aplicación debería haber desplegado.

Respecto a los puntos de fallo posibles, son la descarga de información de despliegue y la comunicación con el servidor de aplicaciones que en el caso de no estar disponible fallaría. El fallo en esta fase suele ser una situación que rara vez se da porque solo se ve afectada por la disponibilidad de la infraestructura.

5.4.1.5. Fase de test

Esta fase es la encargada de verificar que efectivamente el servicio está perfectamente desplegado y funciona de acuerdo a unos test, para ello contamos con unos test de Postman que ejecutamos de forma automática gracias a la versión de linea de comando de Postman, Newman. En primer lugar estos test detectan si la aplicación ha desplegado, ya que en caso de que el despliegue haya fallado por algún motivo, como puede ser que la aplicación no arranca porque no conecta con la base de datos o con otro servicio, esta fase lo indica porque intenta conectarse con un puerto que no le contesta. Un dato a tener en cuenta es que Newman está integrado directamente en la misma red que los microservicios, por lo que existe un newman de pre y un newman de pro, esto es necesario porque los servicios como tal no tiene puertos abiertos al exterior, salvo los llamados gateway o en el caso de los servicios

de autorización y autenticación.

Una vez se ejecutan todos los test, pueden ocurrir dos alternativas, que hayan fallado o que hayan pasado. En el primer caso se debe de marcar el pipeline como fallido y comunicar el error a todos los programadores que tengan relación con el servicio para que tomen las medidas adecuadas.
La segunda alternativa es que el servicio ha desplegado perfectamente por lo que en este caso se realiza la tarea de marcar el ensamblado como funcional, por lo que la ultima imagen y la ultima funcional son la misma, este sistema elimina la vuelta atrás de una versión funcional a otra funcional, pero por decisión de la empresa, se ha aceptado que este caso no tiene aplicación y que solo interesa garantizar la vuelta atrás de una versión que no funciona a otra que si funciona.

5.4.1.6. Fase de rollback

Ya que todo lo que se sube al repositorio de código se despliega, es fácil que ocurra que se sube algo que no funciona, que por como está diseñado el sistema no se considera un error, la idea de poder ser tan ágiles en los despliegues permite que esto sea tolerable y se pueda ver los fallos rápidos para que no lleguen a producción. Pero algunas veces lo subido no se puede solucionar tan rápido y en el caso de que este fallo ha llegado a producción tiene que existir un mecanismo extremadamente ágil que permita en cuestión de pocos minutos volver a la versión que funcionaba.

Esta fase es una que se debe de desplegar de forma manual, ya que para evitar complejidad, no se ha optado por la opción de rollback automático aunque si que existe la posibilidad, pero como puede generar la problemática de no saber exactamente que está desplegado y que no, se decidió que se haría solo de forma manual.

El funcionamiento es prácticamente a como es un despliegue en pro, ya que lo unico que se hace es sustituir en el repositorio de artefactos las imágenes de docker, en este caso se sustituye la ultima imagen, digamos que estamos en el entorno de pre, por lo que sustituimos la imagen pre, por la pre-stable que es aquella que se garantizó que funcionaba.

El siguiente paso es llamar a la fase de deploy y de test para que trabajen como si de un despliegue normal se tratase. Esta fase es incompatible con build, por lo que existe una protección para que no se puedan realizar build y rollback a la vez, en el caso de indicar las dos fases siempre prevalece build y rollback se ignora.

5.4.1.7. Notificación

La ejecución de despliegues se realiza de forma bastante habitual por lo que es fundamental una buena comunicación para detectar posibles problemas, es por ello que además del correo se cuenta con la herramienta de chat Mattermost, en la que se tiene un canal dedicado exclusivamente a despliegues y otro a fallos de los mismos. Por lo que si falla un despliegue todo el equipo lo sabe, para poder actuar con agilidad y solucionarlo, lo que no se suele hacer es la comunicación positiva, es decir no se indica que se ha desplegado algo satisfactoriamente, ya que en general no aporta mucho. En parte no importa ya que el despliegue no tarda mucho y la persona que lo realiza normalmente revisa que todo ha funcionando verificando que no tiene ningún correo de error, aunque obviamente se puede notificar de que se ha desplegado con éxito, pero la empresa prefiere optar por no hacerlo.

5.4.2. Aplicaciones web

El siguiente ejemplo practico a examinar es el despliegue de aplicaciones web que aunque comparte similitudes con los microservicios, muchas de las fases cambian como funcionan internamente.

5.4.2.1. Angular cli

En primer lugar la herramienta encargada de descargar los paquetes, realizar los test y compilar, se trata de Angular-cli, se utiliza esta herramienta debido a que es la que está diseñada para trabajar de forma eficiente con cualquier proyecto de Angular2 en adelante. Con ella se hace la obtención de dependencias, la compilación y los test unitarios.

5.4.2.2. Fase de preparación

La fase de preparación es exactamente igual que en la fase de microservicios, se configura la conexión con kafka para almacenar la información del despliegue y se obtiene un objeto Proyecto con todos los datos que se necesitarán luego para las distintas fases. La conexión con kafka en este caso queda de la síguete forma.

- Fecha de envió del mensaje (23/07/2018T06:19:49.229Z)
- Desarrollador (pablo.barredo@****.com)
- Nombre del proyecto (wot-dashboard)
- Tipo de componente (ui)
- Entorno (pro)

- Fase (Deploy)
- Resultado (SUCCESS)
- Tiempo de ejecución (50.0)

5.4.2.3. Fase de build

La idea general es la misma que en la parte de microservicios, pero en este caso cambian las herramientas utilizadas y algunas pecurialidades. En primer lugar actualmente no se tienen activados los test unitarios en los proyectos webs, algunos proyectos se han activado para ver como son las pruebas en ese tipo software pero no es algo generalizado, como muchos otros aspectos, en algún momento del tiempo se añadirá. Tampoco cuenta con el análisis estático de código aunque se ha estudiado la opción y se ha planificado añadirlo, pero en el momento en que se redacta este documento no está implementado.

Lo que si se hace es el proceso de compilación aunque en este caso la configuración para el despliegue debe ir en el propio ensamblado, esto se hace así por una limitación del lenguaje, aunque existen soluciones los desarrolladores de la parte que compila y empaquetan en código, creen en que se debe hacer un ensamblado por entorno, es por ello que el ensamblado de pre y de pro no son el mismo, aunque el código si está parametrizado, para luego descargar la información de pre o pro según se esté compilando un entorno u otro, evitando subir el código directamente con las cadenas de conexión. Esto por desgracia hace que los pasos a pro requieran de una compilación total y no se pueda cambiar parámetros de forma rápida en la fase de despliegue.
Al igual que en microservicios se sube a Nexus la imagen de docker de la ultima versión, y el Dockerfile una vez más se encuentra en cada proyecto.

5.4.2.4. Fase de despliegue

Esta fase es prácticamente igual que la de microservicios con la diferencia de que en este caso solo se escoge en que entorno se quiere desplegar, no se manda ningún tipo de información al servidor de aplicaciones porque en tiempo de despliegue no se puede alterar el ensamblado. Aunque a diferencia de los microservicios, en estos proyectos el tiempo de espera es fijo para todos los proyectos ya que levantar un servidor web es una tarea rápida y no depende de tantos factores como un servidor de java.

5.4.2.5. Fase de Test

Un añadido relativamente reciente en el momento en el que se escribe este documento, fue añadir la parte de test al pipeline de las webs, esto se debió a querer que todos los pipelines tengan las misma fases y se comporten de la misma manera para que los programadores entiendan claramente que debería ocurrir cuando hacen un commit.

En este caso la herramienta encargada de los test no es postman, si no karma, que realiza los llamados test e2e, estos test utilizando por debajo selenium, realizan una serie de pruebas utilizando un navegador web y simulando que hacen click e interactúan como un humano con la aplicación. Al igual que con los microservicios, si el resultado del test es negativo lo comunican a los programadores responsables y si funciona, sustituyen la imagen estable anterior por la nueva.

5.4.2.6. Fase de rollback

Es exactamente igual a su homologa de microservicios, en este caso si que no existe ninguna diferencia, porque el proceso se gestiona igual sustituyendo las imágenes de docker y desplegando otra vez.

5.5. Ejemplo de una aplicación concreta

5.5.1. Contexto

Para poder comprender mejor el funcionamiento de la plataforma y como es utilizada por un programador se utilizará como ejemplo el servicio Jenkins-data el cual permite conectarse a una base de datos para obtener datos que se han ido guardando de las ejecuciones de los proyectos en Jenkins. Este programa está desarrollado en Java utilizando el framework de Spring Boot, que permite generar API RESTs de forma sencilla.

5.5.2. Creación del proyecto en Gitlab

En primer lugar se debe de crear el proyecto en Gitlab por parte de un administrador, ya que los programadores no pueden crear proyectos por motivos de seguridad. Una vez se tiene el repositorio se le comunica la dirección al programador para que pueda clonar el repositorio y depositar el código.

5.5.3. Inicialización del código

Se debe de crear el proyecto de Spring utilizando una plantilla, y posteriormente se deben de hacer dos modificaciones, en primer lugar se debe de añadir una serie de plugins al pom.xml para poder generar las imágenes de docker y poder también realizar el análisis estático de código.

```
<!-- DOCKER -->
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.4.13</version>
  <configuration>
    <imageName>${docker.image.prefix}/${project.artifactId}</imageName>
    <dockerDirectory>src/main/docker</dockerDirectory>
    <resources>
      <resource>
        <targetPath>/</targetPath>
        <directory>${project.build.directory}</directory>
        <include>${project.build.finalName}.jar</include>
      </resource>
    </resources>
  </configuration>
</plugin>

<!-- Jacoco -->
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.0</version>
  <executions>
    <execution>
      <id>default-prepare-agent</id>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>default-report</id>
      <phase>prepare-package</phase>
      <goals>
```

```
35          <goal>report</goal>
36        </goals>
37      </execution>
38    </executions>
39  </plugin>
```

La segunda modificación es la inclusión del Dockerfile.

```
1  FROM frolvlad/alpine-oraclejdk8:slim
2  #FROM openjdk:8-jdk
3  VOLUME /tmp
4  ADD  jenkins-data-0.0.1-SNAPSHOT.jar app.jar
5  EXPOSE 80
6  #Add bash to run wait-for-it.sh
7  #RUN apk add --update bash && rm -rf /var/cache/apk/*
8  RUN sh -c 'touch /app.jar'
9  ENV JAVA_OPTS=""
10 ENTRYPOINT [ "sh", "-c", "java $JAVA_OPTS
11 -Djava.security.egd=file:/dev/./urandom -jar /app.jar" ]
```

5.5.4. Plantilla de pipeline

Se dispone de un repositorio centralizado donde están todas las configuraciones de los pipelines. El programador debe copiar el de ejemplo y rellenarlo con los parámetros propios de su proyecto. El archivo debe de estar en una carpeta con el nombre del proyecto que luego se deberá indicar a Jenkins.

El archivo en cuestión es el siguiente:

```
1  name : jenkins-data
2  git_project : http://olaf.empresa.com/sisplant/jenkins-data.git
3  image_name: jenkins-data
4  image_tag : latest
5  postman_collection : jenkins-data.postman_collection.json
6  postman_environment : jenkins-data.postman_environment.json
7  service_name : jenkins-data
8  wait_for_deploy : 30
9  memorybytes: 524288000
```

El fichero permite en primer lugar indicar el nombre del proyecto así como los datos que serán necesarios para la compilación, despliegue y pruebas. En la parte de despliegue solo existen los parámetros relativos a la configuración de docker swarm ya que los parámetros de la aplicación se introducen en otra herramienta.

5.5.5. Creación del proyecto en Jenkins

Para poder ejecutar el proyecto se debe crear una tarea en Jenkins de tipo pipeline poniendo como fuente un repositorio. Para facilitar la tarea al programador existe una tarea plantilla con todo preparado que simplemente deberá clonar y cambiar el valor por defecto de la variable "proyect_name", que es la responsable de seleccionar que proyecto se ejecuta de todos los disponibles en el repositorio. Es fundamental que el nombre de la variable y la carpeta que contiene el archivo "config.yml" sea el mismo ya que si difieren el pipeline es incapaz de descargarse las propiedades del proyecto.

La tarea de jenkins cuenta además con cinco variables más que son:

- **build:** Booleano que indica si se ejecuta el paso de build (por defecto activado)
- **deploy:** Booleano que indica si se ejecuta el paso de deploy (por defecto activado)
- **test:** Booleano que indica si se ejecuta el paso de test (por defecto activado)
- **rollback:** Booleano que indica si se ejecuta el paso de rollback (por defecto desactivado y no se debe marcar a la vez que el de build)
- **environment:** selector que permite escoger un entorno en el que desplegar el proyecto (por defecto está marcado el entorno de pre)

5.5.6. Archivos de test

Para realizar los test de los microservicios se utiliza la herramienta postman, que cuenta con una interfaz gráfica en la que poder escribir y probar los test. Una vez el programador los tiene listos debe subirlos a un repositorio centralizado de archivos postman al que accederá el pipeline cuando se encuentre en la fase de test. El funcionamiento es exactamente el mismo que el del archivo de configuración. El programador debe crear una carpeta con el nombre del proyecto y en este caso crear dos archivos uno son los test en si y el otro son las variables de entorno que se van a sustituir en el test como puede ser la cadena de conexión al servicio.

5.5.7. Configuración de variables de entorno

Ya que se cuenta con distintos entornos en fundamental el poder modificar las configuraciones de forma sencilla por entorno. Se ha desarrollado una herramienta que cumple esta función. Utilizando esta herramienta se deben de crear dos configuraciones que siguen la siguiente regla "nombre de servicio-entorno" en este caso son jenkins-data-pre y jenkins data pro. En la siguiente figura podemos ver la herramienta y un ejemplo de la configuración del servicio:

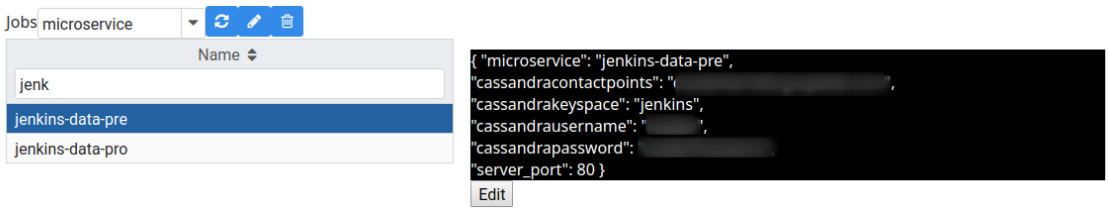

Figura 5.10: Herramienta de configuración de variables de entorno

5.5.8. Enlazado de Gitlab y Jenkins

Por ultimo se debe de unir el proyecto de Jenkins con el repositorio de Gitlab para que cuando se realice un commit se despliegue automáticamente. Este paso solo lo puede hacer un administrador, y se realiza de la siguiente manera:

Se debe configurar en el proyecto la opción que aparece en la figura que genera una url en la que gitlab podrá mandar un mensaje que hará que Jenkins comience a ejecutar el pipeline

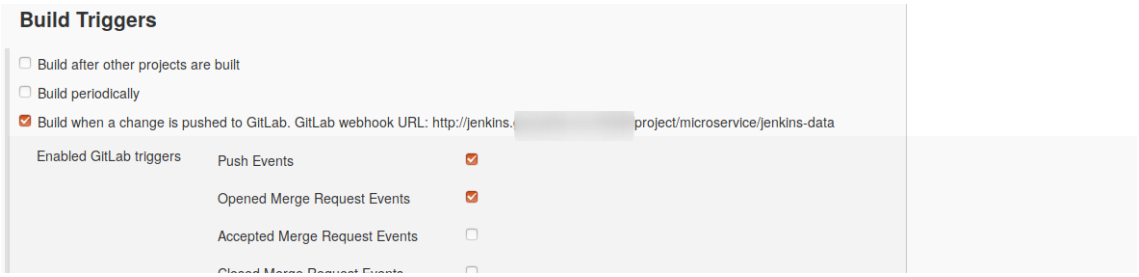

Figura 5.11: Plugin de Gitlab en Jenkins

En gitlab la opción es igual de sencilla de configurar, simplemente se debe de pegar la url que ha generado jenkins e indicar que solo se ejecute el pipeline en el caso de un push.

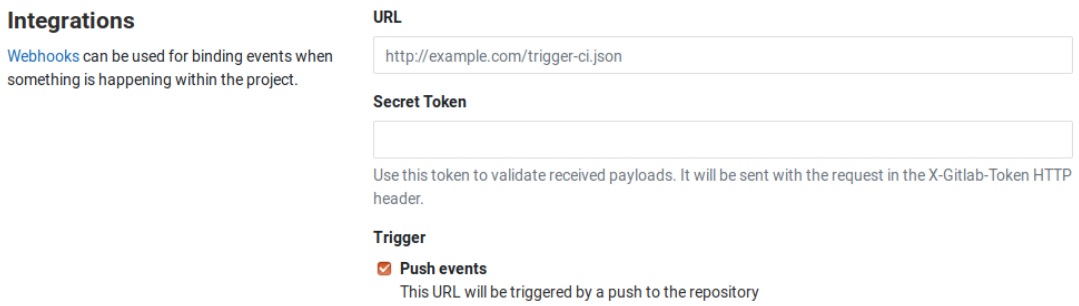

Figura 5.12: Configuración de un webhook en Gitlab

5.5.9. Ejecución

Con todos estos pasos realizados ya estaría preparado el proyecto para cuando se realice un commit se despliegue de forma automática aunque también existe la posibilidad de ejecutar el pipeline manualmente. Si se quiere por ejemplo redesplegar el microservicio sin volver a hacer una compilación o realizar un rollback porque el código no funcione correctamente una vez desplegado.

5.5.10. Errores

Los pipelines no siempre funcionarán es por ello que deben de estar preparados para ser capaces de comunicar como han terminado y cual ha sido el fallo. En primer lugar se puede ver de forma visual ya que saldrá que todo el pipeline ha fallado y se podrá ver exactamente cual ha sido la fase que ha fallado porque al ser un pipeline no pasa a la siguiente fase si ha ocurrido un error por lo que la ultima fase que se ha ejecutado es la que ha fallado.

La otra vía de comunicación es mediante un correo al programador que indica los test fallidos, si los hubiera, y una copia del log para detectar el fallo.

Capítulo 6

Entorno de desarrollo

6.1. Necesidad

La integración continua no es algo estático que se realiza una vez y queda montado de manera indefinida, si no que se va adaptando y mejorando, introduciendo nuevas partes u opciones para poder implementar una nueva tecnología o mejorar una ya existente, es por ello que es fundamental generar un entorno que nos permita programar dicha entorno y poder realizar pruebas para garantizar que una nueva mejora no deje inutilizado el sistema de despliegues.

6.2. Configuración automática

Existen muchas opciones a la hora de generar un entorno de pruebas, una muy habitual es la creación de una copia espejo de la de producción para realizar pruebas, y en general puede resultar atractiva, pero si varias personas quieren poder trabajar de forma cómoda puede no resultar viable el tener un entorno por cada programador, es por ello que la alternativa más viable es la creación de una imagen de docker ya configurada y lista para funcionar, que se pueda personalizar con las credenciales del usuario que la va a utilizar. Esto además tiene utilidad para la propia versión productiva porque rápidamente se puede generar una instancia de Jenkins con todo listo, siempre y cuando se tengan los datos de producción disponible en algún sitio. La imagen automatizada se ha podido realizar gracias a los scripts de inicialización de groovy [28], que permiten realizar unas operaciones al arrancar jenkins.

6.2.1. Dockerfile

La imagen de jenkins que se va a ser utilizada por los programadores, es una versión propia, por lo que se genera a partir de un dockerfile, en primer lugar se utiliza la imagen oficial lts como base [29], se instala los mismos plugins que se usarán en producción, se desactiva el asistente de instalación y finalmente se añaden

los scripts de inicialización para personalizar jenkins, siendo imprescindibles, el que genera usuarios y el que añade las credenciales para acceder a los repositorios.

```
FROM jenkins/jenkins:lts
RUN /usr/local/bin/install-plugins.sh git build-pipeline-plugin \
    workflow-cps-global-lib matrix-auth credentials-binding \
    workflow-aggregator swarm sonar cucumber-reports kafkalogs \
    filesystem_scm pipeline-utility-steps build-user-vars-plugin \
    mattermost ssh-agent

ENV JENKINS_USER admin
ENV JENKINS_PASS admin


# Skip initial setup
ENV JAVA_OPTS -Djenkins.install.runSetupWizard=false


COPY default-user.groovy /usr/share/jenkins/ref/init.groovy.d/
COPY add-credentials.groovy /usr/share/jenkins/ref/init.groovy.d/
```

6.2.2. Docker-compose

Al ser necesarios varios componentes para la ejecución del entorno lo más efectivo es recurrir al uso de un fichero docker-compose, con el que se podrá levantar un contenedor jenkins, un esclavo y un registry de docker para pruebas de subidas.

```
version: '3'
services:
  jenkins-dev:
    build: .
#    ports:
#      - 9096:8080
    environment:
      - GITLAB_USER=${GITLAB_USER}
      - GITLAB_PASS=${GITLAB_PASS}
      - DEFAULT_BRANCH=${DEFAULT_BRANCH}
      - REPOSITORY=${REPOSITORY}
    network_mode: "host"
    volumes:
```

```
14        - "${HOST_PATH}:/home/"
15        - "${LIBRARY_PATH}:/var/jenkins_home/workflow-libs"
16      deploy:
17        placement:
18          constraints:
19            - node.hostname == swarm-1
20    jenkins-slave:
21      image: registry.empresa.com:19003/jenkins-slave
22      volumes:
23        - "/var/run/docker.sock:/var/run/docker.sock"
24      environment:
25        - "COMMAND_OPTIONS=-master http://localhost:8080 \
26        -username ${MASTER_USER:-admin} -password ${MASTER_PASS:-admin} \
27        -labels ${SLAVE_LABEL:-test} -executors 5 -mode exclusive"
28      network_mode: "host"
29
30    registry:
31      image: "registry:2"
32      ports:
33        - "19003:5000"
```

El archivo contine varios puntos importantes, en primer lugar, la imagen de jenkins se construye desde el propio compose, todos los parámetros sensibles y únicos del programador se pasan a través de variables de entorno con un fichero que no se subirá al control de versiones, la red se utiliza en modo host por lo que no es necesario abrir ningún puerto, se utilizan dos volúmenes, el primero es para poder d utilizarlo de carpeta compartida con el contenedor, para conveniencia del programador y el segundo es para poder compartir la librería de jenkins que es la que se desarrollará con ayuda del entorno, se hace mediante este sistema para evitar el uso de un repositorio intermedio.
En el caso del esclavo se aprecia que tiene un volumen un tanto peculiar, esto es para poder ejecutar docker desde un contenedor docker, utilizando del demonio de la maquina host.

```
1 # .env.sample
2 GITLAB_USER=CHANGE
3 GITLAB_PASS=CHANGE
4 DEFAULT_BRANCH=master
5 REPOSITORY=CHANGE
```

```
6 LIBRARY_PATH=CHANGE
7 HOST_PATH=CHANGE
```

El archivo de docker-compose ha de ir acompañado de un .env, como el mostrado anteriormente para poder inyectar las variables de entorno de forma sencilla.
Una vez preparados los ficheros necesarios se debe ejecutar el entorno con "docker-compose up". El entorno una vez arrancado nos permitirá ir haciendo pruebas y desarrollando el código que queramos añadir en un futuro a la integración continua.

6.3. Biblioteca de Jenkins

6.3.1. Utilidad

A medida que se vaya desarrollando distintos flujos de integración continua se irá notando que se empieza a repetir código y muchos proyectos podrían compartir una serie de funciones comunes. Al igual que cualquier otro lenguaje de programación se pueden realizar funciones centralizadas en librerías que permitan compartir código entre proyectos. Es por ello que se debe diseñar una librería única donde poder almacenar todo el código de integración continua consiguiendo así un doble objetivo, en primer lugar se tendrá un control de versiones de la integración continua y en segundo lugar se permite simplificar los archivos de configuración por proyecto, siendo solo necesario modificar los parámetros propios al proyecto, pero si se requiere un cambio generalizado al tener un único punto se puede hacer de forma sencilla.

6.3.2. Estructura

Jenkins nos dota de una herramienta para poder crear librerías que el mismo carga dinámicamente cada vez que lanza una ejecución de un trabajo, descargando de forma automática de un repositorio de código. Esta herramienta llamada "shared libraries"[30], cuenta con una estructura concreta que se ha de respetar para poder garantizar el correcto funcionamiento.

- root
- src (Archivos de código)
  - com
    - empresa
      - Foo.groovy(Archivo de clase Foo)
- vars
  - foo.groovy(Paso foo)
  - foo.txt (Documentación del paso)
- resources
  - com
    - empresa
      - bar.json(Archivo json auxiliar)

En primer lugar tenemos el directorio de "src" en el que depositaremos el código como si de un proyecto de java se tratase. En general esta carpeta se utiliza simplemente para crear clases, aunque para poder intentar conseguir un desarrollo más estructurado, la implementación particular, ha consistido en utilizar patrones ya conocidos como DAO, DTO, Factories y otros muchos que podemos encontrar en cualquier otro proyecto. Aunque la librería está más pensada para intentar ir a un estilo que más directo, como si scripts de bash se tratasen, el utilizar patrones permite mejorar la calidad del software al ser más mantenible en el tiempo y permitiendo simplificar funcionalidades más complejas.

La siguiente carpeta es "vars", el objetivo de esta carpeta es albergar funciones conocidas como "steps" que se podrán llamar desde el pipeline de jenkins. El mayor problema que presenta este sistema es que no se pueden hacer subcarpetas, lo cual puede dificultar bastante que el código sea comprensivo, es por ello que se debe intentar que las funciones sean muy parametrizable para no tener muchas funciones

que realicen acciones similares.

Por ultimo existe la carpeta resources, que actualmente no se le da ninguna utilidad pero permite tener archivos que pueda necesitar algún apartado, como una archivo de configuración o cualquier otro tipo de archivo.

6.3.3. Detalles de Src

El apartado de src de la librería fue una mejora no solo sobre lo que estaba actualmente en la empresa si no sobre el resto de proyectos que habitualmente se pueden encontrar de librerias de jenkins. Normalmente el acercamiento recae en hacer muchos steps como funciones sueltas, ero en este caso se quiso aplicar un enfoque de programación clásica aplicando herencia, polimorfismo, diferentes patrones, consiguiendo así un código que se reutiliza de forma cómoda y se puede invocar en cualquier step. Un ejemplo de esto lo podemos encontrar en el siguiente código:

```
1  package com.empresa.dto.microserviceenvironment;
2
3  public class MicroserviceEnvironmentFactory{
4
5          public MicroserviceEnvironment getEnvironment(String environment) {
6
7                  switch (environment) {
8                  case "pre":
9                          return new MicroserviceEnvironmentPre();
10                 case "pro":
11                         return new MicroserviceEnvironmentPro();
12                 case "intranet":
13                         return new MicroserviceEnvironmentIntranet();
14                 default:
15                         return new MicroserviceEnvironmentPre();
16                 }
17         }
18 }
```

```
package com.empresa.dto.microserviceenvironment;

public class MicroserviceEnvironmentPre extends MicroserviceEnvironment {

        public MicroserviceEnvironmentPre() {
                super();
                swarm_manager_ip = "swarm-pre.empresa.com";
                swarm_manager_hostname = "cluster-019.empresa.com";
                jenkins_slave = "newman-pre";
        }

        public String toString() {
                return "MicroserviceEnvironmentPre [swarm_manager_ip="
                        + swarm_manager_ip + ", swarm_manager_hostname="
                        + swarm_manager_hostname + ", jenkins_slave="
                        + jenkins_slave + "]";
        }
}
```

```
package com.empresa.dto.microserviceenvironment;

public class MicroserviceEnvironment implements Serializable {

        public String swarm_manager_ip
        public String swarm_manager_hostname
        public String jenkins_slave
}
```

Como se puede ver, en este caso se ha optado por realizar un patrón Factory.

6.3.4. Detalles de vars

Esta carpeta es donde se deposita todo el código que Jenkins va a ejecutar, la idea general es que existe una tarea global que empieza con el nombre pipeline y luego existen pequeñas tareas intermedias que se pueden reutilizar en diversos pipelines o lanzarse por separado.
En este caso el código es groovy y si se utilizan bastantes peculiaridades del lenguaje.

A continuación se verá el código encargado de desplegar los microservicios, aunque puede parecer bastante extraño, algunas de las peculiaridades vienen impuestas por Jenkins como la forma en la que se reciben los parámetros.

La idea principal de la tarea es obtener los parámetros de despliegue, retirar el servicio actual si lo hubiese y desplegar el nuevo mediante el uso de la api rest de docker swarm.

```
1  import groovy.json.JsonOutput
2  def call(body){
3      def config = [:]
4      if ( body != null){
5          body.resolveStrategy = Closure.DELEGATE_FIRST
6          body.delegate = config
7          body()
8      }
9      call(config)
10 }
11
12 def call(Map config = [:]){
13     def service_name = config.service_name
14     def image_name = config.image_name
15     def image_tag= config.image_tag?:'latest'
16     def swarm_manager_ip = config.swarm_manager_ip?:env.PRE_SERVER
17     def assembly_configuration_mongo=
       ↪  config.assembly_configuration_mongo
18     def ports = config.ports?:null
19     def memorybytes = config.memorybytes?:0
20
21     def deployConf = httpRequest "http://swarm-pro.empresa.com:8900/"
22         +"aeinfrastructure/configuration/microservice?name="
23         +"${assembly_configuration_mongo}"
24     echo deployConf.getContent()
25     echo ports
26     def portjson = JsonOutput.toJson(ports:[])
27     if(ports){
28         def split_ports=ports.tokenize(',')
29
30             def port = split_ports[0]
31
```

```
32
33              portjson = JsonOutput.toJson(Ports: [[Protocol:
                ↪ "tcp",PublishedPort: port[0].toInteger(), TargetPort:
                ↪ port[1].toInteger()]])
34      }
35      httpRequest httpMode: 'DELETE', responseHandle: 'NONE', url:
        ↪ "http://${swarm_manager_ip}:4243/services/${service_name}",
        ↪ validResponseCodes: '100:600'
36      sleep 10
37      def XRA='{ "username": "****", "password": "****", "email":
        ↪ "*****" }'.bytes.encodeBase64().toString()
38      def containerEnv=deployConf.getContent()
39      echo containerEnv
40      if (containerEnv!="")
41      {
42          containerEnv="${containerEnv},
            ↪ \"JAVA_OPTS=-XX:+UnlockExperimentalVMOptions
            ↪ -XX:+UseCGroupMemoryLimitForHeap\",
            ↪ \"zipkin_base_url=http://cluster-033.empresa.com:9411/\""
43      }
44
45      // DEPLOY json
46      def deployJson = """{
47      "Name": "${service_name}",
48      "TaskTemplate":
49      {
50          "ContainerSpec": {
51              "Image":
    ↪ "${env.REGISTRY_SERVER}/${image_name}:${image_tag}",
52              "Env": [${containerEnv}]
53          },
54          "Resources":  {
55              "Limits": {
56              "MemoryBytes": ${memorybytes}
57              }
58          }
59
60      },
61      "Networks": [{ "Target": "empresa-swarm" }],
62      "EndpointSpec": ${portjson}
```

```
63 }"""
64
65     echo deployJson
66     def deployRequest = httpRequest(url:
       ↪ "http://${swarm_manager_ip}:4243/services/create", httpMode:
       ↪ "POST", requestBody: deployJson, customHeaders: [[maskValue:
       ↪ true, name: 'X-Registry-Auth', value: XRA]])
67     echo deployRequest.getContent()
68
69
70
71 }
```

El código del pipeline de microservicios es bastante extenso por lo se muestra una figura con todos los elementos implicados en el pipeline donde en primer lugar se puede ver que el pipeline necesita definir el entorno en el que lo va a desplegar, utilizando un factory se crea un objeto de uno de los tres posibles entornos(pre, pro o intranet). El siguiente elemento implicado es el injector de kafka, cuyo cometido es la inserción de objetos "pipelineSteps" que contiene información de tiempos, resultado, fases y demás información relevante de la ejecución del pipeline. El objeto "MailNotification" permite obtener la lista de correo a la que debe mandar los mails. Por ultimo se cuenta con "ConfigurationDao" que permite obtener el objeto de proyecto correspondiente, en este caso "ProjectMicroservice" que contiene todo la información necesaria para posteriormente compilar, desplegar y testar el proyecto.

En el apartado de steps, que son pequeñas funciones de Jenkins, se cuenta con las propias del sistema y con unas que se han creado especificamente para el entorno de la empresa.
Las que se utilizan propias del sistema son las siguientes:

- **ssh:** Permite la conexión por ssh con otra maquina para enviar comandos
- **git:** Descarga un repositorio de código
- **httpRequest:** Realiza peticiones http y retorna un objeto con la respesta
- **readYaml:** Lee de un archivo YAML y retorna un objeto con todas sus propiedades
- **mattermostSend:** Envía un mensaje al sistema de mensajería instantánea Mattermost

- **emailText:** Envía correos con una determinada plantilla

En el caso de los pasos custom cabe destacar que algunos llaman a pasos del sistema, lo que se han creado y se utilizan en el pipeline son los siguientes:

- **buildMicroservice:** Compila el código, crea una imagen Docker del microservicio y lo sube a nexus (Utiliza los steps ssh y git)
- **deployMicroservice:** Despliega un microservicio
- **testMicroservice:** Prueba el microservicio que está desplegado con newman
- **replaceDockerImage:** Permite crear una copia de una imagen de Docker con otra etiqueta
- **when:** Una función que se usa como sustituta del if para que cuando no se realiza una fase se marque como saltado, haciendo que visualmente se vea claramente que no se ha ejecutado esa fase

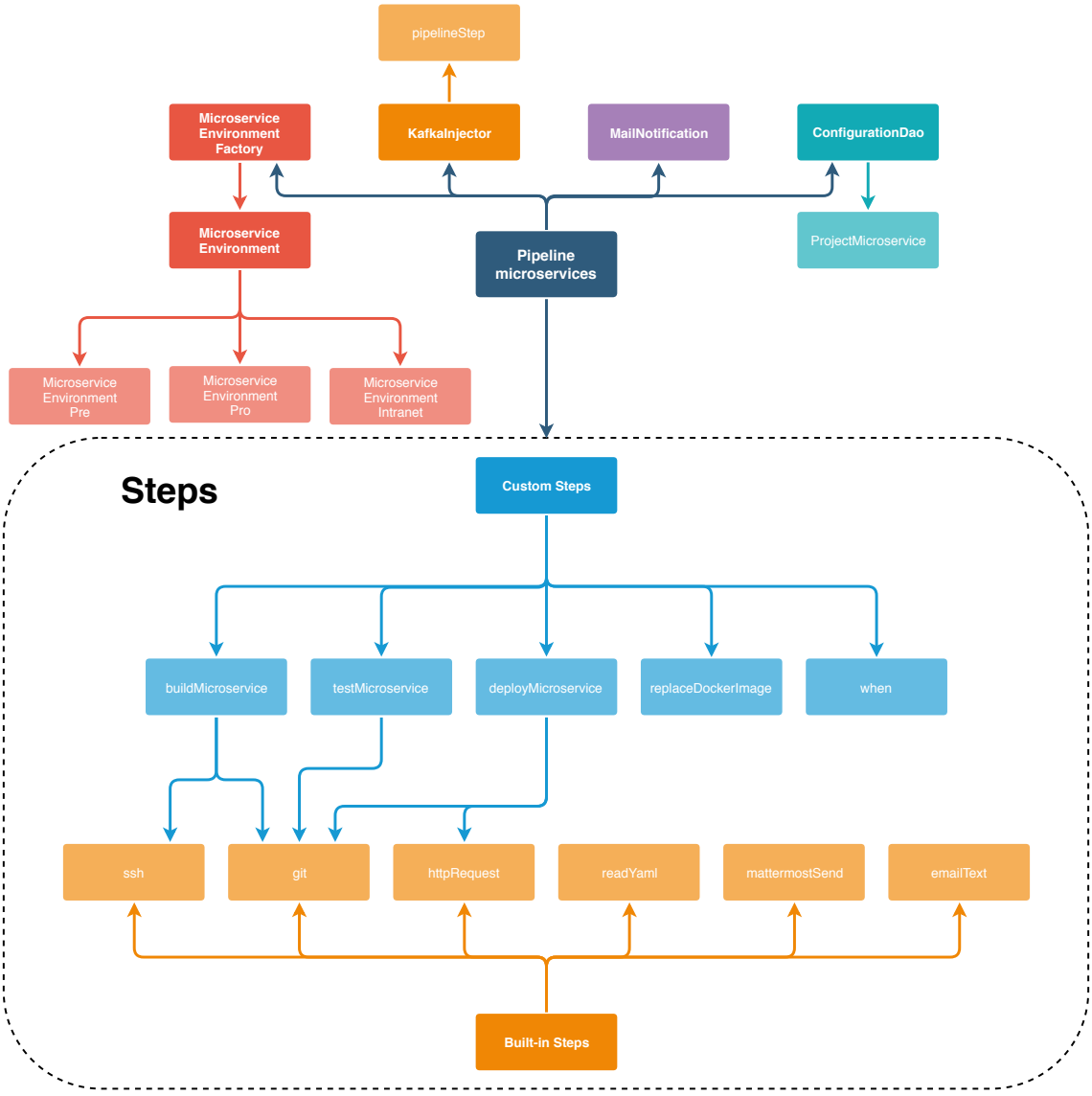

Figura 6.1: Diagrama del pipeline de microservicios

Capítulo 7

Conclusiones

7.1. Proyecto

Aunque el proyecto como tal sigue en evolución y se van poco a poco incorporando mejoras, lo que se tiene hasta el momento ha supuesto una gran mejora frente a la forma en la que se trabajaba anteriormente. Este proyecto ha permitido ver como es posible automatizar la mayoría de los procesos que se realizan habitualmente en los despliegues, consiguiendo así que cualquier persona que se incorpore tenga una curva de aprendizaje mucho más liviana, evitando tener que formar a todas las personas en todos los entresijos de la integración continua que está montada.

Este proyecto además ha permitido de forma cómoda ir incorporando nuevas ideas como la monitorización del las llamadas a APIs REST que se realizan en todos los proyectos, que con el sistema antiguo se tendrían que alterar todos los proyectos y todas las tareas mientras que en este caso solo se tuvo que cambiar en un punto central y se replicó rápidamente a todas las tareas.

El proyecto ha supuesto un cambio en el funcionamiento y velocidad de los desarrollos, al poder desplegar un nuevo servicio en un intervalo de tiempo muy corto y sin apenas interacción o conocimiento por parte del programador, ayuda a que el personal vea resultados pronto lo cual supone un incentivo. Para las personas que se incorporan este sistema permite que la curva de entrada no sea muy compleja ya que solo deben de seguir una serie de instrucciones.

Al tratarse de un proyecto modular permite de forma cómoda el poder añadir nuevas mejoras que pueden ser definitivas o permanentes e invita a la mejora por parte de los propios programadores ya que si aparece una necesidad o mejora para un proyecto, al estar centralizada esa mejora se propaga para todos los proyectos.

7.2. Valoración personal

Desde mi punto de vista personal, este proyecto me ha permitido enfrentarme a un entorno real con sus diversos problemas y dificultades. El proyecto al ser mejorar y no remplazar me pareció bastante interesante porque me permitió experimentar de primera mano lo que es analizar algo que no has hecho tu e intentar comprenderlo, analizar sus fallos y mejorarlo incluyendo nuevas funcionalidades pero si que sea tan disruptivo que las personas que ya lo están utilizando tengan que aprender un sistema completamente distinto.

El trabajo incremental me pareció muy interesante ya que es otra filosofía de trabajo y me parece muy acertada, ya que en un primer lugar haces un acercamiento a la solución del problema, pruebas como funciona y vas mejorándola hasta dar con lo que mejor se adapta, ya que el ir directamente a buscar el óptimo muchas veces puede resultar en un fracaso porque o bien falta conocimiento o la solución óptima es demasiada disruptiva y es mejor buscar algo que pueda ir poco a poco hacia el óptimo.

Apéndice A

Presupuesto

Concepto	Precio unitario	Unidad	Unidades	Total
Equipo X9DR7/E-(J)LN4F	€ 1.600,00	cantidad	1	€ 1.600,00
Precision WorkStation T3400	€ 900,00	cantidad	4	€ 3.600,00
Dell Optiplex 380	€ 600,00	cantidad	2	€ 1.200,00
Dell Optiplex 390	€ 700,00	cantidad	2	€ 1.400,00
PowerEdge R410	€ 3.000,00	cantidad	1	€ 3.000,00
PowerEdge R300	€ 2.200,00	cantidad	1	€ 2.200,00
PowerEdge R200	€ 1.900,00	cantidad	1	€ 1.900,00
Graduado en ingeniería	€ 20,75	horas	2080	€ 43.160,00
Total				€ 58.060,00

Apéndice B

Planificación

Fecha de inicio	Fecha de Fin	Descripción	días hábiles
04/09/2017	15/09/2017	Estudio de situación actual	10
18/09/2017	29/09/2017	Estudio de 1ª Versión de pipeline	10
02/10/2017	13/10/2017	Creación de paso de build	10
16/10/2017	03/11/2017	Creación del paso de deploy	15
06/11/2017	17/11/2017	Creación del paso de test	10
20/11/2017	01/12/2017	Webhook gitlab-Jenkins	10
04/12/2017	15/12/2017	Sistema de mensajeria	10
18/12/2017	29/12/2017	Sistema de inyección de variables	10
01/01/2018	12/01/2018	Analisis estatico de código	10
15/01/2018	09/02/2018	Creación de entorno de desarrollo	20
12/02/2018	09/03/2018	Creación de biblioteca de código de código	20
12/03/2018	06/04/2018	Creación de pipelines configurables	20
09/04/2018	27/04/2018	Adaptación de tareas a steps	15
30/04/2018	11/05/2018	Creación de sistema de rollback	10
14/05/2018	08/06/2018	Creación de pipeline de UI	20
11/06/2018	06/07/2018	Instrumentalización de los pipeline	20

09/07/2018	03/08/2018	Sistema de testing para los pipelines	20
06/08/2018	31/08/2018	Refactorización de la biblioteca	20
Total			260

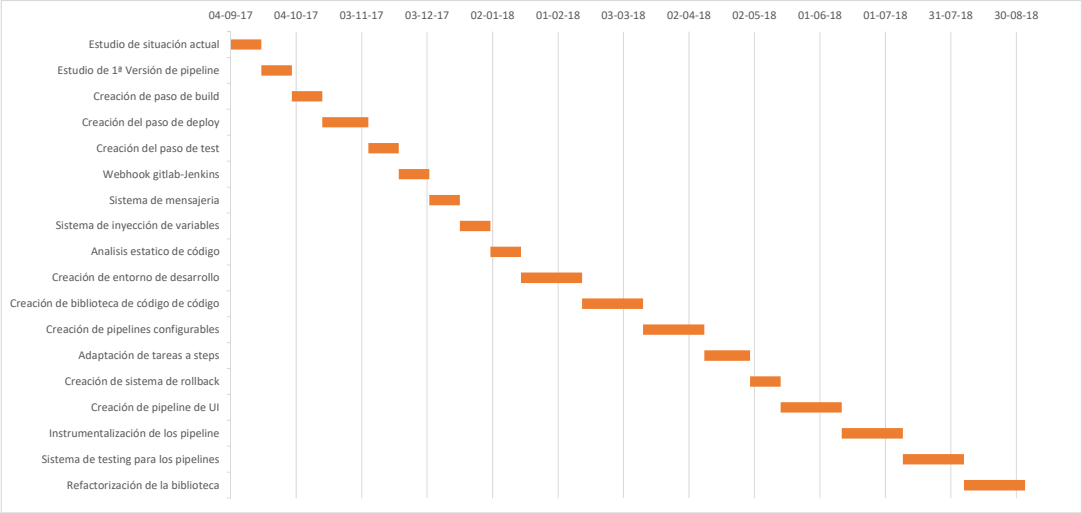

Figura B.1: Diagrama Gantt

Bibliografía

[1] Atlassian, "Comparing workflows." https://www.atlassian.com/git/tutorials/comparing-workflows, (Accedido el 24 de Septiembre de 2018).

[2] M. Fowler, "Continuous integration." https://www.martinfowler.com/articles/continuousIntegration.html, 2006, (Accedido el 23 de Marzo de 2018).

[3] B. Laster, *Continuous Integration vs. Continuous Delivery vs. Continuous Deploymnet*. O'Reilly Media, 2017.

[4] J. Shore, "Fail fast," *IEEE Software*, vol. 21, pp. 21–25, 09 2004.

[5] A. S. B. y. J. P. Andy Gumbrecht, *Testing Java Microservices: Using Arquillian, Hoverfly, AssertJ, JUnit, Selenium, and Mockito*. Manning Publications, 2018.

[6] M. Blog, "Microsoft to acquire github for $7.5 billion." https://news.microsoft.com/2018/06/04/microsoft-to-acquire-github-for-7-5-billion/, (Accedido el 24 de Septiembre de 2018).

[7] Github, "Pricing." https://github.com/pricing, (Accedido el 24 de Septiembre de 2018).

[8] Github, "Features." https://github.com/features, (Accedido el 24 de Septiembre de 2018).

[9] Gitlab, "Price." https://about.gitlab.com/pricing/, (Accedido el 24 de Septiembre de 2018).

[10] Gitlab, "Features." https://about.gitlab.com/product/, (Accedido el 24 de Septiembre de 2018).

[11] Bitbucket, "Price." https://bitbucket.org/product/pricing, (Accedido el 24 de Septiembre de 2018).

[12] Bitbucket, "Features." https://bitbucket.org/product/features, (Accedido el 24 de Septiembre de 2018).

[13] M. TFS, "Features." https://visualstudio.microsoft.com/es/tfs/, (Accedido el 24 de Septiembre de 2018).

[14] M. TFS, "Price." https://visualstudio.microsoft.com/es/team-services/tfs-pricing/, (Accedido el 24 de Septiembre de 2018).

[15] M. A. DevOps, "Features." https://azure.microsoft.com/es-es/services/devops/, (Accedido el 24 de Septiembre de 2018).

[16] M. A. DevOps, "Price." https://azure.microsoft.com/es-es/pricing/details/devops/azure-devops-services/, (Accedido el 24 de Septiembre de 2018).

[17] A. Maven, "Info." https://maven.apache.org/what-is-maven.html, (Accedido el 24 de Septiembre de 2018).

[18] A. Ant, "Info." https://ant.apache.org/, (Accedido el 24 de Septiembre de 2018).

[19] A. Ivy, "Info." https://ant.apache.org/ivy/features.html, (Accedido el 24 de Septiembre de 2018).

[20] Gradle, "Info." https://gradle.org/features/, (Accedido el 24 de Septiembre de 2018).

[21] NPM, "Info." https://www.npmjs.com/get-npm, (Accedido el 24 de Septiembre de 2018).

[22] S. Nexus, "Info." https://www.sonatype.com/nexus-repository-oss, (Accedido el 24 de Septiembre de 2018).

[23] J. Artifactory, "Info." https://jfrog.com/open-source/, (Accedido el 24 de Septiembre de 2018).

[24] Eclipse, "Jenkins as an alternative." https://wiki.eclipse.org/Jenkins#About_Jenkins, (Accedido el 24 de Septiembre de 2018).

[25] Jenkins, "Info." https://jenkins.io/, (Accedido el 24 de Septiembre de 2018).

[26] Bitbucket, "Bitbucket pipelines." https://bitbucket.org/product/features/pipelines, (Accedido el 24 de Septiembre de 2018).

[27] Travis, "Info." https://travis-ci.com/, (Accedido el 24 de Septiembre de 2018).

[28] O. J. Docs, "Post-initialization script." `https://wiki.jenkins.io/display/JENKINS/Post-initialization+script`, (Accedido el 23 de Marzo de 2018).

[29] Jenkins, "Jenkins oficial docker image." `https://hub.docker.com/r/jenkins/jenkins/`, (Accedido el 23 de Marzo de 2018).

[30] O. J. Docs, "Extending with shared libraries." `https://jenkins.io/doc/book/pipeline/shared-libraries/`, (Accedido el 23 de Marzo de 2018).