UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño e implementación de un sistema de entrega continua para aplicaciones web sobre contenedores Docker.

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Todea, Virgil Cristian

Tutor: Espinosa Minguet, Agustín Rafael
Cotutor: Hortelano Otero, Jorge

2015/2016

Resumen

El objetivo de este trabajo de fin de grado es reducir el tiempo necesario para desplegar las aplicaciones de la empresa BiiT Sourcing Solutions en los entornos de producción. Todo ello teniendo plenas garantías de que el *software* desplegado cumple con unos estándares de calidad mínimos. Para ello se ha implantado un sistema de entrega continua, donde una vez los desarrolladores han terminado una nueva versión de la aplicación, se realicen todos los tests necesarios automáticamente que aseguran el correcto funcionamiento del *software*. Además la solución está basada en virtualización mediante contenedores Docker para así asegurar su portabilidad y optimizar los recursos de la empresa.

Palabras clave: despliegue, integración continua, entrega continua, Docker, contenedores, Jenkins, automatización.

Abstract

The main objective for this degree final project is to reduce the necessary time for deploying any applications developed in BiiT Sourcing Solutions into a production environment. All this having a full guarantee that the generated software accomplish the minimum quality standards. In order to achieve this, we have implemented a continuous delivery system, where once the development team has finished a new version of the application, all necessary tests are performed automatically ensuring the correct operation of the software. In addition to this, the proposed solutions is based on virtualization using Docker containers to ensure the portability and the optimization of the company's resources.

Keywords: deploy, continuous integration, continuous delivery, Docker, containers, Jenkins, automatization.

Índice general

1. Introducción **1**

1.1. Motivación y objetivos de este trabajo 1

1.2. ¿Qué es BiiT? 2

1.3. Problemática dentro de la empresa 3

1.4. Estructura del trabajo 4

1.4.1. Contexto previo 4

1.4.2. Propuesta 4

1.4.3. Integración continua (IC) 4

1.4.4. Entrega continua (EC) 5

1.4.5. Contenedores como plataforma 5

1.4.6. Conclusión 5

2. Contexto previo de la empresa **7**

2.1. Desarrollo ágil: Scrum 7

2.2. Gestión del código fuente 10

2.2.1. Git 11

2.3. Gestión de dependencias 12

2.3.1. Maven 13

2.3.2. Artifactory 16
2.4. Despliegue de aplicaciones 16
2.4.1. Tomcat 17

3. Propuesta y pasos a seguir 19
3.1. Entrega continua (EC) 19
3.2. Virtualización 21
3.3. DevOps 22
3.4. Casos similares en otras empresas 23

4. Integración continua 25
4.1. Adaptación de las ramas de Git 28
4.2. Testeo 30
4.2.1. Tests Unitarios 31
4.2.2. Tests de Integración 31
4.3. Servidor de IC: Jenkins 32
4.3.1. Primer *job*: fase de *development* 34
4.3.2. Segundo *job*: fase de integración 38
4.3.3. Tercer *job*: fase de *release* 41
4.4. Extendiendo la `IC` 42

5. Entrega continua 43
5.0.1. Instalación automática 44
5.1. Ansible 44
5.2. Pasos para desplegar la aplicación 47
5.2.1. Despliegue del WAR 47

5.2.2. Sincronización de los ficheros de configuración 48
5.2.3. Actualización de URLs 49
5.2.4. Actualización de la base de datos 50
5.3. Despliegue desde Jenkins 51
5.4. Entrega continua frente a despliegue continuo 52

6. Virtualización como plataforma 55

6.1. ¿Qué es Docker? 56
6.2. Creación de imágenes 58
6.2.1. *Dockerfile* 58
6.2.2. `docker commit` 59
6.3. Virtualización del entorno de desarrollo 59
6.3.1. Configuración de red 60
6.3.2. Implantación de docker en la empresa 61
6.3.3. Supervisor 62
6.4. Arquitectura resultante 62

7. Evaluación, conclusiones y trabajo futuro 65

7.1. Valoración de resultados 65
7.1.1. Mejoras en los tiempos de despliegue 65
7.1.2. Mejora en la calidad del *software* 66
7.2. Conclusión 67
7.3. Trabajo futuro 68

Índice de figuras

2.1. *Product Objetc Model* (POM) de ejemplo con etiquetas básicas 13

2.2. Arquitectura de Maven utilizando un repositorio privado 15

3.1. Estructura de los roles tradicionales frente a DevOps 23

4.1. Riesgo respecto al tiempo mediante la integración tradicional 26

4.2. Riesgo respecto al tiempo mediante la integración continua 27

4.3. Proceso de integración y su relación con las ramas de Git 33

4.4. Configuración para que Jenkins pueda acceder a Git 35

4.5. Significado de los campos de Cron 36

4.6. Frecuencia con la que se consulta la presencia de nuevos *commits* en el SCV 37

4.7. Comandos para limpiar la base de datos (BD) 37

4.8. Configuración de los parámetros de Maven para ejecutar la compilación y los tests unitarios 38

4.9. Configuración del sistema de *feedback* de Jenkins 38

4.10. Configuración para que Jenkins suba el código a *integration* una vez acabe el *job* 39

4.11. Configuración del bloqueo entre *jobs* de integración y del mismo proyecto 40

4.12. Configuración de Maven para ejecutar los tests de integración 40

5.1. *Playbook* de ejemplo 45

5.2. Ejemplo del fichero `/etc/ansible/hosts` 46

5.3. Ejecución de un *job* parametrizado 51

5.4. Ejecución de comandos en función de una condición 53

5.5. Diferencia entre entrega continua y despliegue continuo 54

6.1. Distinción entre máquina virtual (MV) y contenedores 56

6.2. Ejemplo de *dockerfile* 58

6.3. Funcionamiento del *proxy* inverso 61

6.4. Comparativa entre la arquitectura previa y la arquitectura virtualizada . 63

7.1. Comparativa entre el tiempo de despliegue de *minor releases* 66

7.2. Comparativa entre el tiempo de despliegue de *major releases* 67

7.3. Número de *bugs* encontrados después de realizar un despliegue 67

Capítulo 1

Introducción

Este trabajo se ha realizado en colaboración con la empresa BiiT Sourcing Solutions, y su desarrollo se ha llevado a cabo durante los 4 meses de prácticas. Las prácticas fueron llevadas a cabo dentro del departamento de sistemas de la empresa, encargado de mantener y mejorar la infraestructura de esta.

Los avances realizados en la empresa durante este periodo están siendo continuados actualmente, gracias a un contrato para continuar con la estancia en la empresa después de finalizar el convenio de prácticas.

Motivación y objetivos de este trabajo

La finalidad de la empresa al plantear este trabajo es mejorar la experiencia de los clientes con las aplicaciones y en definitiva la calidad de los servicios ofrecidos.

A un nivel más técnico, estas ideas generales se traducen en cambios en diversos aspectos, que engloban desde el proceso de desarrollo, pasando por la instalación hasta la etapa de mantenimiento y postproducción. Describamos los aspectos más importantes que se pretenden mejorar:

- Actualizar los procesos de la empresa: adaptarse a las herramientas y metodologías más modernas nos reporta ventajas para todas las etapas de desarrollo de las aplicaciones.
- Aumentar la productividad de los trabajadores: esta idea da paso a la automatización de tareas repetitivas, uno de los aspectos clave durante este trabajo.

- Reducir los riesgos y en tiempo invertido en los despliegues: la instalación de la aplicación en las máquinas de producción es uno de los aspectos más propenso a errores. Los problemas a la hora de desplegar se traducen en tiempo y recursos que no invertimos en otras tareas más productivas.
- Mejorar la interacción con el cliente: esto se consigue mediante instalaciones más frecuentes y permitiendo una comunicación más directa entre cliente y equipo de desarrollo.
- Optimizar el uso de los recursos disponibles con la finalidad de ahorrar dinero (objetivo secundario).

Para conseguir los objetivos descritos, vamos a diseñar un sistema que permita realizar despliegues en producción *con un click*. De esta manera, invirtiendo tiempo y recursos en crear y mantener este sistema, conseguimos un ahorro temporal sustancial en todas las etapas del ciclo de la aplicación, sobretodo en las posteriores a la fase de desarrollo.

La rentabilidad de la inversión inicial se acentúa si pretendemos mantener o seguir desarrollando el producto durante un largo periodo de tiempo.

Una vez expuestos nuestros objetivos pasemos a introducir brevemente el contexto en el que desarrollaremos el trabajo.

¿Qué es BiiT?

Se trata de una empresa joven, con algo más de tres años de antigüedad, que basa su negocio en el desarrollo de aplicaciones *web* basadas en Java. Para el desarrollo, la mayoría de herramientas empleadas son de código libre, sobre estas hablaremos a medida que vayamos profundizando en los aspectos técnicos.

La productos ofrecidos por la empresa se basan en lo que se denomina *Business Inteligente* (BI)[1].

El objetivo del BI consiste en recabar información de distintas fuentes, procesarla, y reportar las conclusiones con la finalidad de ayudar en la toma de decisiones, todo ello basado en técnicas informáticas [14].

Este último concepto se puede aplicar en diversos campos, en nuestro caso particular son dos:

- Medicina, concretamente en centros fisioterapeuticos.

[1]Esto tiene una relación directa con el nombre de la empresa. BiiT son las siglas de *Business Inteligence and Information Technology*

- Instituciones públicas.

Esto último nos conduce a mencionar que la empresa tiene sede en Valencia, pero su mercado está orientado a los Paises Bajos.

Para todo ello la empresa se basa en una infraestructura en la nube, esto permite:

- Ahorrar en costes en mantenimiento y montaje: aspecto importante en una empresa de este tamaño.
- Ofrecer un servicio más estable: no se depende de un único servidor alojado en las instalaciones de la empresa.
- Mayor flexibilidad frente a las demandas de los clientes.
- Escalabilidad a un coste muy bajo.

Después de esta breve definición el contexto en el que nos desenvolveremos, pasemos a definir los problemas más comunes a los que se enfrentaba la empresa antes de la realización de este trabajo. A su vez, esto nos dará paso a introducir las soluciones propuestas.

Problemática dentro de la empresa

Como ya hemos descrito durante la sección de los objetivos a cumplir, uno de los aspectos más propensos a errores son los despliegues. Esto se debe a que las aplicaciones reales son elementos muy complejos. Es común que para su desarrollo se usen decenas de librerías (en ocasiones de terceros), que se basen en múltiples tecnologías o se empleen distintos *frameworks*[2] Esta complejidad se traduce en más errores potenciales.

Durante un despliegue normalmente es necesario realizar cambios en la BD, en ficheros de configuración, instalar nuevas dependencias, etc.

Por tanto es un proceso costoso y lento, y en muchas ocasiones muy repetitivo. Este proceso acababa tomando varias horas o incluso días en caso de complicarse.

Otro aspecto de los problemas detectados era el testeo de las aplicaciones. Este proceso llevado a cabo manualmente conlleva una gran inversión temporal. Además debemos tener en cuenta que los encargados de testear la aplicación también pueden

[2]Entornos de trabajo que facilitan el desarrollo mediante cierto nivel de abstracción, permitiendo programar a más alto nivel.

omitir errores o casos de prueba debido a la saturación. Esto último se traducía en *bugs* que se detectaban una vez la aplicación ya había pasado a producción.

Por último, cabe mencionar que aún contando con una arquitectura en la nube, el uso de los recursos no siempre era optimizado, dando lugar en muchos casos a máquinas sobredimensionadas. Lo que supone un gasto adicional que no reporta beneficios.

Estructura del trabajo

Contexto previo

En primer lugar, dedicaremos un capítulo al contexto técnico y a las metodologías de la empresa antes del inicio de nuestro trabajo. Enumeraremos las herramientas más importantes, y su finalidad. También profundizaremos en los detalles técnicos más relevantes, ya que son conceptos necesarios para entender los capítulos posteriores.

Propuesta

Dedicaremos un capítulo a presentar las soluciones propuestas partiendo de la base de la empresa.

En este capítulo presentaremos brevemente los conceptos de los capítulos siguientes, y describiremos cómo ayudan a conseguir cada uno de nuestros objetivos (mencionados en la sección 1.1) así como la solución a los problemas de la empresa descritos en la sección 1.3.

Integración continua (IC)

A continuación entraremos de pleno con nuestra solución propuesta, que como comentaremos más adelante, requiere de un paso previo: la instauración de un sistema de integración continua. En este capítulo explicaremos de forma teórica en que consiste la IC. Posteriormente hablaremos sobre los cambios necesarios en la arquitectura previa de la empresa, para que esta pueda adaptarse a los requisitos de la IC.

Finalmente comentaremos los pasos realizados para la implementación de dicho sistema. Para esto último emplearemos una herramienta llamada Jenkins.

Entrega continua (EC)

Destinaremos este capítulo a la entrega continua, mencionada durante el capítulo "Propuestas y pasos a seguir" como solución potencial a la mayor parte de los problemas descritos.

Esta guarda una relación directa con el sistema del capítulo de IC, por tanto se tomará como punto de partida el trabajo descrito en este.

Una vez más, dedicaremos una sección a los conceptos teóricos, para dar paso luego a la implementación de estos en el caso particular de la empresa.

Durante la introducción hemos mencionado algunos de los problemas comunes durante los despliegues. En este capítulo los detallaremos todos, explicando de forma más técnica por qué se producen. Esto nos conducirá a exponer las implementaciones necesarias para solucionarlos, todo ello mediante el uso de la herramienta propuesta. Para finalizar este capítulo, mencionaremos qué es el despliegue continuo (DC), y qué relación guarda con la EC.

Contenedores como plataforma

La virtualización como concepto es introducida durante la sección 3.2, pero de forma poco detallada. En este capítulo nos adentraremos mucho más en aspectos técnicos sobre su funcionamiento, y en concreto sobre el funcionamiento y características de los contenedores. Lo que nos llevará a presentar la herramienta que utilizaremos para este propósito: Docker.

Explicaremos qué es exactamente y cómo ha sido utilizada como plataforma. También comentaremos los problemas encontrados durante la virtualización de toda la infraestructura de desarrollo.

Como final de este capítulo ilustraremos la arquitectura resultante con la finalidad de compararla con la arquitectura anterior.

Conclusión

Después de la finalización de los capítulos más técnicos, dedicaremos este capítulo final a las conclusiones obtenidas durante el trabajo. También comentaremos las limitaciones que hemos encontrado y haremos una evaluación de como han repercutido los cambios en la empresa. Por último mencionaremos el trabajo futuro a realizar para seguir desarrollando nuestra solución.

Capítulo 2

Contexto previo de la empresa

Desarrollo ágil: Scrum

Vamos a dedicar esta sección a detallar la manera de trabajar de la empresa. Para ello debemos explicar que la empresa toma un enfoque ágil para el desarrollo del software. En dicho enfoque se pretende una adaptación rápida frente a los cambios en lugar de seguir un plan maestro, además de conseguir una alta interacción con los clientes y priorizar el desarrollo de software frente a una rigurosa documentación de los procesos [7].

Todo lo mencionado anteriormente se lleva a cabo en un *modus operandi* basado en Scrum, sin llegar a adoptar este sistema de forma estricta.

Antes de comenzar, debemos aclarar que lo que se pretende en este apartado es describir la manera de actuar de la empresa, aplicando ciertos patrones de *Scrum*, no es nuestro objetivo describir todos los aspectos técnicos de Scrum. Por tanto vamos a centrarnos en los detalles que más relación tienen con nuestro propósito dejando de lado algunos menos relevantes como los roles dentro de *Scrum* o los tiempos estipulados para cada evento.

El término "*Scrum*", sinónimo de melé, proviene del argot del rugby, y da nombre a la acción que reinicia el juego después de que este haya sido detenido por una infracción.

Dejando la etimología de lado, vamos a aclarar de qué estamos hablando realmente. *Scrum* no es una metodología, es decir, no pretende decirnos exactamente que debemos hacer en cada caso. En cambio, debe entenderse como un marco de trabajo, dentro del cual se emplean varias técnicas y procesos, con el propósito de entregar los productos (nuevas versiones del *software*) con la máxima productividad posible.

Este modelo está pensado para ser aplicado en equipos relativamente pequeños, para

que sean ágiles, pero suficientemente grandes como para que sean capaces de abordar una carga de trabajo considerable [28].

El tamaño óptimo varía entre 5 y 9 personas, teniendo en cuenta que los equipos deben ser independientes y auto organizados [20].

Uno de los fundamentos de *Scrum* es el empirismo, tomando así como premisa que el conocimiento es resultado de la experiencia y la toma de decisiones debe basarse en este. Por tanto, para optimizar la toma de decisiones, se debe contar con personal experimentado, capaz de gestionar situaciones adversas tomando las decisiones adecuadas y reduciendo al mínimo los riesgos asumidos.

El aspecto que acabamos de describir se asienta sobre tres bases [28]:

- Transparencia: es importante que todos los miembros del equipo puedan apreciar el progreso de los procesos. Para ello es recomendable fijar y estandarizar los baremos para que la percepción de cada individuo no sea subjetiva.
- Inspección: con la finalidad de detectar variaciones en el estado de los procesos respecto a los objetivos, se deben realizar inspecciones con frecuencia. Pero se debe tener en cuenta que estas no deben llegar a interferir con en la dinámica de trabajo. Las inspecciones deben llevarse a cabo en el mismo lugar de trabajo y son especialmente provechosas si son llevadas a cabo por revisores con una amplia experiencia.
- Adaptación: los procesos deben ser lo suficientemente flexibles para ajustarse a pequeños cambios en caso de que durante las inspecciones se estime que el resultado no será el esperado. Cuanto antes se realicen dichos ajustes menor será el riesgo de que surjan desviaciones entre la planificación y la ejecución.

Una vez conocemos los tres pilares fundamentales de los procesos de *Scrum*, vamos a definir el marco temporal en el que estos son llevados a cabo: el *sprint*.

Un *sprint* es un bloque temporal cuya duración puede variar dependiendo de las circunstancias del proyecto, oscilando entre un par de semanas y un mes [20]. El objetivo del *sprint* consiste en desarrollar un producto (o modificación sobre uno ya existente) completamente funcional y apto para ser puesto en producción.

Dentro de cada *sprint* se definen cuatro eventos principales, dichos eventos están ideados para cumplir con las bases descritas anteriormente: transparencia, inspección y adaptación.

- El primer evento son las reuniones de planificación para las tareas que se realizaran en dicho *sprint*. En esta reunión se lleva a cabo al inicio, y en ella se fija cual

es el objetivo o meta del *sprint*. Además, como mencionamos al hablar sobre la transparencia en los procesos, también es necesario fijar unos límites comunes para homogeneizar el juicio de cuando un proceso se considera finalizado.

En el caso particular de la empresa, el objetivo de los *sprints*, su duración y punto de conclusión son definidos por las personas con más experiencia: el director y el analista.

- Reunión diaria (también denominada *daily scrum*) Como su nombre indica, este evento consiste en una reunión, de corte informal, realizada cada día durante el curso del *sprint*. Lo ideal es que dicha reunión sea llevada a cabo siempre en el mismo lugar y a la misma hora y su duración no debería exceder el cuarto de hora.

  En dicha reunión cada miembro del equipo de desarrollo lleva a cabo una corta exposición de que ha realizado desde la reunión del día anterior, que tarea realizará durante ese mismo día y si ha encontrado alguna dificultad o impedimento en la tarea que tiene asignada para que sea discutida o solicitar ayuda a otro miembro.

  Las principales finalidades de este evento son dos:

  - Sincronización del equipo de desarrolladores, se evita así que dos personas estén trabajando en la misma tarea sin saberlo.
  - Inspección por parte de los responsables. Estos pueden observar si las tareas avanzan al ritmo planeado o si requieren de más recursos o tiempo para ser completadas.

- El tercer evento consiste en revisar el *sprint* una vez este ha sido terminado.

  En esta reunión se pretende determinar si los objetivos planificados se han cumplido o no. También se ponen en en común las opiniones sobre los problemas que han surgido durante la realización de las tareas.

  En esta reunión se busca aprender de los errores, recopilar datos que puedan resultar útiles y en definitiva acumular experiencia para los *sprints* futuros, ya que como hemos dicho anteriormente, *Scrum* se basa en el empirismo.

- El último evento se denomina "retrospectiva del *sprint*" y consiste en una reunión entre la descrita en el punto anterior y la reunión para planificar el siguiente *sprint*.

  El objetivo de esta es inspeccionar los elementos más importantes en cuanto a personal, relaciones, herramientas y procesos, errores y mejoras del último *sprint*.

  Al igual que en la reunión para la examinación del *sprint*, lo que se pretende es aprender para poder mejorar en el futuro, pero en este caso nos centremos en mejoras relacionadas a la forma de trabajo del equipo de *Scrum*.

Gestión del código fuente

A continuación vamos a tratar uno de los aspectos más básicos durante el desarrollo de cualquier proyecto software: la gestión de su código fuente.

Para tratar esto, introduciremos uno de los pilares del entorno de desarrollo de cualquier empresa que se dedique al desarrollo software, el sistema de control de versiones (SCV).

Con el paso del tiempo distintas tendencias han surgido en el ámbito del desarrollo software, pero pocas han sido tan ampliamente aceptadas como los SCVs, que se ha convertido en un estándar *de facto*. Esta tecnología surgió a finales de los años 70, como solución que permitía a varios desarrolladores manejar el código fuente de forma concurrente y continua. Estos sistemas proporcionan la habilidad de ver la evolución del código a través del tiempo como si tratara de instantáneas, permitiendo la posibilidad de revertir los cambios a un punto concreto, una característica que los ha convertido en un componente esencial para los proyectos desarrollados en equipo [30].

Internamente, los SCVs guardan ficheros de código en lo que se conoce como repositorios. Desde un punto de vista práctico, un fichero de código puede ser entendido como un fichero de texto plano, esto es importante para definir cuál es la unidad atómica de este, es decir, la menor fracción que se puede modificar para considerar que el fichero ha sido actualizado. En la actualidad la mayoría de SCVs toman como unidad atómica la línea.

Arquitectónicamente existen cuatro grandes modelos, en los que se basan los SCVs para gestionar los repositorios [30]:

- Repositorio local aislado: un único repositorio en la máquina del desarrollador, solo se permite el acceso local, por tanto todos los cambios deben ser realizados desde esa máquina. Es el más primitivo y el menos útil a la hora de trabajar en equipo.
- Repositorio local compartido: un único repositorio en la máquina del desarrollador, pero en este caso con acceso a él mediante la red de área local. El repositorio actúa como si fuera una carpeta compartida entre varios usuarios.
- Repositorio remoto en un servidor: esta implementación sigue el esquema clásico de cliente/servidor. Se aloja el repositorio en un servidor al que acceden los desarrolladores para leer o escribir sus cambios.
- Repositorio distribuido: cada colaborador almacena una copia completa del repositorio en su máquina local.

La implementación más común actualmente es la basada en repositorios distribuidos,

con algún matiz que mencionaremos más adelante. Esta arquitectura otorga mayor independencia a los desarrolladores si el proyecto es grande, ya que, en una primera etapa sus modificaciones serán guardados solo en su repositorio local. Este sistema también permite que las operaciones sean más rápidas, ya que al trabajar en local no es necesario realizar constantemente operaciones a través de la red.

En la arquitectura de repositorios distribuidos, para poner en común el trabajo de todos los desarrolladores, es necesario combinar los cambios realizados sobre sus respectivos repositorios locales en un repositorio compartido entre todos ellos, vamos a denominar a este repositorio "remoto".

En en caso concreto de la empresa, como SCV se usa Git, que con alrededor de un 40 % de cuota de mercado [3] es de los más extendidos y utilizados. A continuación la ahondaremos sobre el uso de Git y la organización del código dentro de este.

Git

Para empezar, expliquemos como surgió esta herramienta. Git tiene una relación directa con el *kernel* de Linux, ya que nació como la alternativa para el desarrollo de este, además ambos fueron creados por Linus Torvalds.

Hasta el año 2005, los desarrolladores del *kernel* utilizaban un SCV propietario, pero en ese año, las relaciones entre los desarrolladores y la empresa dueña del SCV se rompieron. Esto impulsó a los desarrolladores, y en particular a Torvalds, a desarrollar su propio SCV [13].

Sobre los detalles técnicos de este, quedan fuera del alcanza de este trabajo, pero debemos saber que en Git existen tres comandos básicos:

- `git add`: marca los archivos modificados, para que sean subidos al repositorio en la siguiente sincronización (*commit*).
- `git commit`: guarda todos los archivos marcados en el repositorio local. Generalmente Git solo añade datos, por tanto después de realizar un *commit* es muy difícil que perdamos fragmentos de código.
- `git push`: guarda los cambios del repositorio local en el repositorio remoto

Estructuralmente Git utiliza un sistema de ramas, donde normalmente existe una rama principal, comúnmente denominada *master*. La función de esta rama es conservar una versión funcional, probada y estable del código.

Se pueden crear nuevas ramas a partir de una existente, y fusionarlas entre ellas, o bien borrarlas cuando se desee.

Internamente la rama *master* es idéntica a cualquier otra rama. Podríamos tener la versión estable del código en cualquier rama, simplemente que esta es la generada por defecto cuando se crea un repositorio, y tanto su finalidad como su nombre se han estandarizado con el paso del tiempo.

La metodología lleva a cabo a la hora de desarrollar nuevas características era:

1. Crear una rama secundaria aparte de la rama *master*.
2. Utilizar dicha rama para el desarrollo de la nueva funcionalidad.
3. Cuando el proceso de implementación y testeo de la nueva característica finalizaba, se fusionaba la rama secundaria con *master*.
4. La rama secundaria era borrada.

Una vez descrito esta metodología, introduzcamos otro de los aspectos potencialmente problemáticos a la hora de crear software: la gestión de dependencias.

Gestión de dependencias

Como ya hemos mencionado anteriormente, el software es un sistema complejo, donde se requiere que muchos componentes trabajen en sintonía. En general una aplicación acaba teniendo muchas dependencias, como por ejemplo: múltiples librerías, una versión específica de un sistema gestor de bases de datos, otra aplicación, etc.

Esto obliga a que la lista de características requeridas para el correcto funcionamiento de la aplicación sea común a todo el proyecto. Esto a menudo se vuelve complejo dentro del equipo de desarrolladores ya que cada uno trabaja en una versión distinta y acaba teniendo dependencias distintas. Si estas no gestionan correctamente, o no existe una sistema estandarizado para ponerlas en común se acaba convirtiendo en una fuente de errores. El problema se acentúa si tenemos en cuenta que puede haber dependencias anidadas, es decir, que las dependencias tengan a su vez más dependencias.

Como solución, se requiere de un sistema que estandarice y automatice la gestión de todos los conflictos descritos, para este propósito la empresa usa Maven. A esta herramienta dedicaremos la siguiente sección, pero antes debemos hacer un breve inciso. Para comprender mejor el contexto de la siguiente sección vamos a aclarar que la empresa basa sus aplicaciones en Java EE, utilizando un *framework* llamado Vaadin para la interfaz y MySQL para la gestión de bases de datos.

```
<modelVersion>4.0.0</modelVersion>
<artifactId>usmo-core</artifactId>
<packaging>jar</packaging>
<name>Uitgedbreid SportMedisch Onderzoek (Core)</name>

<parent>
  <groupId>com.biit</groupId>
  <artifactId>usmo</artifactId>
  <version>1.1.128-SNAPSHOT</version>
</parent>
```

Figura 2.1: POM de ejemplo con etiquetas básicas

Maven

Maven es una herramienta de código libre, creada por Apache y cuya primera versión fue lanzada en 2004. Esta herramienta está específicamente pensada para trabajar con proyectos en Java, lo que la convierte en adecuada para el entorno de la empresa.

Aunque con el tiempo su funcionalidad ha ido en aumento, durante su desarrollo inicial los dos objetivos principales fueron:

- 1: Estandarizar la estructura de directorios, para que fuera más fácil encontrar donde están los ficheros de código, las librerías, la documentación, etc
- 2: Crear un modelo de proyecto en el que se pudiera ver fácilmente todos los componentes que pertenecen a dicho proyecto.

Para la segunda finalidad Maven utiliza un fichero *eXtensible Markup Language* (XML) llamado POM. Todos los proyectos Maven tiene un POM que contiene toda la definición del proyecto, así como las dependencias de este.

Para resolver los problemas de dependencias, Maven recurre al concepto de artefacto. Podemos entender un artefacto como la extensión de una librería, ya que a parte de contener las clases de la librería propiamente dicha, el artefacto incluye cierta información para que sea gestionada correctamente. Con el fin de aclarar lo explicado, vamos a ver unos extractos de un POM real.

Pasemos a explicar las etiquetas para comprender la figura:

- groupId: identifica el proyecto a lo largo de la organización, por tanto, debe ser único. Además define la estructura del empaquetado y la ruta de este dentro del

repositorio. En la figura 2.3.1 apreciamos que este módulo forma parte de una jerarquía cuyo padre está bajo el groupID: `com.biit`.

- artifactId: hace referencia al nombre del artefacto, para ello se usa el nombre del fichero *Java ARchive* (JAR) que lo contiene, quitándo el número de version si lo tuviese.
- version: sirve para indicar la versión concreta del artefacto que queremos utilizar.
- packaging: indica el formato del artefacto. Los más comunes son *Web application Archive* (WAR) y JAR.

Lo siguiente de debemos saber es que Maven también define los ciclos de vida de la aplicación, es decir, existe un proceso definido para construir y distribuir artefactos con Maven.

La explicación de los ciclos de vida, implica a su vez explicar las etapas de cada uno, lo que acaba siendo un aspecto más relacionado al desarrollo *software* que a nuestros objetivos. Por tanto solo vamos a introducir brevemente tres de las fases, las que tienen relación directa con conceptos expuestos en capítulos posteriores (concretamente en los capítulos 4 y 5):

- `compile`: compilación de todo el código fuente del proyecto.
- `test`: ejecución de los tests unitarios y de integración.
- `package`: el código compilado y testeado se empaqueta para ser distribuido.
- `deploy`: se copia el paquete al repositorio remoto de artefactos.

Un último punto nos lleva a explica que Maven está pensando para trabajar con repositorios. Normalmente, Maven utiliza dos repositorios:

1. Repositorio remoto: Maven nos ofrece un repositorio central público en Internet [1]). Gracias a este podemos resolver todas las dependencias definidas en nuestro POM, dicho repositorio cuenta con más de 90.000 artefactos [31].
2. Repositorio local: con la finalidad de que las dependencias no sean descargadas de nuevo en cada compilación, Maven utiliza este repositorio a modo de caché.

Ahora bien, a nosotros nos interesa desplegar nuestros paquetes generados en la fase de *deploy* para que todos los desarrolladores puedan acceder a ellos. Para la empresa

[1] `http://search.maven.org/`

no es deseable hacer públicos los artefactos de sus aplicaciones, por tanto no podemos utilizar el repositorio central para este propósito.

Para solucionar esto último surgen los gestores de paquetes, que nos permiten permiten crear repositorios privados.

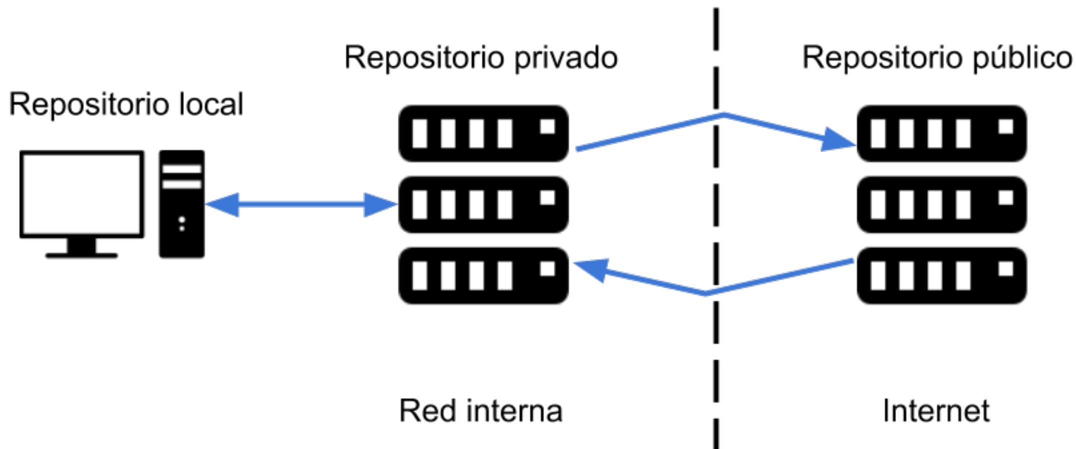

Figura 2.2: Arquitectura de Maven utilizando un repositorio privado

Estos repositorios privados actúan a modo de *proxy* (figura: 2.2) del repositorio central, lo que nos ofrecen varias ventajas [2]:

1. Aumenta la estabilidad debido a que ya no se depende siempre de la conexión al servidor central.
2. Nuestro gestor de paquetes accederá al servidor central solo en caso de que le pidamos un artefacto del que no dispone. Esto reduce en gran medida el número de descargas desde el servidor central, lo que se traduce en dependencias resueltas más rápidamente y menos ancho de banda consumido. El ancho de banda puede resultar un recurso a tener en cuenta si, como en nuestro caso, la infraestructura reside en la nube, donde los proveedores nos cobran si excedemos cierto límite.
3. Nos permite tener mayor control sobre los artefactos que utilizamos y consumimos.
4. Sirve de plataforma para intercambiar artefactos ya compilados entre los distintos proyectos si tener que recurrir al código fuente.
5. Nos permiten funcionalidades extra, como por ejemplo: almacenar documentación asociada a cada librería.

Tener un gestor de paquetes que actúe de repositorio privado nos obliga a seguir cierta organización a la hora de almacenar nuestro artefactos, de lo contrario podría resultar

incluso contraproducente. Evitar tener almacenados artefactos que ya no utilizamos o bien evitar duplicidades mejorando los tiempos de construcción del proyecto.

Como repositorio de artefactos la empresa utilizaba una herramienta de código libre llamada Artifactory, al que dedicaremos el siguiente apartado.

Artifactory

Artifactory, como su propio nombre indica, es una herramienta para gestionar repositorios de artefactos, cumpliendo con todas las funciones descritas en la sección anterior. Ha sido desarrollada por JFrog,es de código libre y fue lanzada en 2008.

Artifactory nos permite configurar parámetros de seguridad para controlar el acceso a nuestros artefactos, haciéndolos privados. Además está diseñado para asegurar la integridad de los datos que almacena así como soportar cargas de trabajo de forma concurrente [6].

Para interactuar con él, contamos con una interfaz *web* y una API REST. Esta última característica resulta especialmente útil si queremos manipular los artefactos mediante algún tipo de automatismo.

Pero también debemos tener en cuenta que disponer de un repositorio privado de artefactos privado nos obliga a mantener cierta organización, para que su funcionamiento no se vea ralentizado por jerarquías mal definidas o artefactos duplicados[2].

Para finalizar aclaremos que en Artifactory se despliegan los JARs y los WARs en la fase de *deploy* de Maven, no debemos confundir este proceso con el despliegue en producción. Este último era un proceso manual, y consistía en: acceder a Artifactory, descargar el paquete deseado e instalarlo en la máquina de producción.

Esto último nos lleva a presentar el proceso de despliegue de la empresa.

Despliegue de aplicaciones

Como se ha mencionado en la introducción, la empresa desarrolla aplicaciones *web* en un lenguaje de alto nivel. Tomando como punto de partida lo descrito en la sección anterior, debemos saber que el paquete se despliega sobre un contenedor de aplicaciones *web*, cuyas funciones son:

[2]La versión gratuita de Artifactory no ofrece herramientas para controlar esto, pero se puede solucionar usando la API REST, mediante un *script* podemos listar los identificadores de los artefactos y buscar duplicados

1. Descomprimir el paquete, en nuestro caso un WAR.
2. Montar la estructura de carpetas necesaria para la aplicación.
3. Publicar el contenido para que sea accesible desde la red.

Para desempeñar esta función la empresa empleaba Tomcat.

Tomcat

Esta herramienta *open source*, creada por Apache está especialmente diseñada para aplicaciones *web* basadas en Java.

Dichas aplicaciones, como hemos mencionado en varias ocasiones, las empaquetamos dentro de un WAR. Este archivo está formado por las clases Java que harán el papel de servidor (comúnmente conocidas como *servlets*) y ficheros que contienen información relativa a la aplicación. En estos ficheros se define el punto de entrada a la aplicación y el mapeado entre las URLs a las clases Java.

Para desplegar una aplicación en Tomcat, basta con copiar el WAR en una ruta definida (comúnmente `webapps`). Una vez hecho esto Tomcat se encarga de todos los pasos enumerados en la sección anterior, dejándola lista para ser utilizada.

Debemos mencionar que Tomcat también se asegura de que las peticiones a la aplicación solo sean respondidas si el cliente tiene los permisos adecuados.

Capítulo 3

Propuesta y pasos a seguir

Vamos a comenzar este capítulo partiendo de los problemas descritos (sección 1.3) y los objetivos a cumplir (sección 1.1). El objetivo de este capítulo es plantear una solución a estos aspectos, argumentando que beneficios aportan los distintos cambios.

Debemos aclarar que no es nuestro objetivo cambiar el marco de trabajo de la empresa, es decir, no se dejará de usar ninguna herramienta que la empresa ya usaba anteriormente para sustituirla por otra nueva. En cambio lo que se pretende es diseñar una solución que se adapte al marco de la empresa.

Dicho esto, debemos aclarar que para lograr nuestro fin, sí hay que dar un enfoque distinto a algunas de las herramientas utilizadas. Dichos cambios serán descritos cuando procedan.

Pasemos pues a describir en qué consiste nuestra solución.

Entrega continua (EC)

En primer lugar, vamos a introducir brevemente este concepto, al que se dedica todo el capítulo 5. Pero antes debemos saber que a la EC es la extensión de otro modelo: la IC.

Describamos brevemente ambos términos:

- IC: modelo para el desarrollo de software por el cual se pretende poner en común (integrar) el trabajo de los distintos desarrolladores lo más frecuentemente posible.
- EC: modelo para el despliegue de aplicaciones cuya finalidad es poder realizar

instalaciones automáticas bajo demanda y con garantías.

Pasemos a enumerar los argumentos de por qué nuestra propuesta parece adecuada:

- Nos permite desplegar cambios en la aplicación en un par de horas. Imaginemos que se ha corregido un *bug*, y que se quiere llevar a producción cuanto antes. Mediante la EC, el tiempo que se tardaría viene marcado principalmente por el tiempo que dure la compilación y el testeo del código para aplicar dicho cambio. Si el cambio es menor el despliegue se convierte en un proceso rutinario, siendo cuestión de minutos.

  Esto conecta directamente con el objetivo de reducir el tiempo de despliegue.

- Favorece la escalabilidad: una vez montada toda la infraestructura, el coste del mantenimiento es muy bajo. Esto permite que un mismo administrador puede gestionar muchos más proyectos que si lo tuviera que hacer de manualmente.

- Reduce costes: al reducir el tiempo dedicado a procesos repetitivos, el equipo técnico puede dedicarse a tareas más importantes, lo que se traduce en ahorro de dinero a la vez que una mejora en la productividad. Cumpliendo una vez más con uno de los objetivos propuestos

- Debido a que los despliegues de nuevas versiones son más frecuentes, el nivel de interacción con el cliente también mejora, ya que sus demandas son resueltas con mayor rapidez.

- Como se ha mencionado en los problemas comunes de la empresa (1.3), el testeo manual tiende a no depurar todos los errores. Mediante la EC, el número de errores que llegan a producción es mucho menor [1].

- El equipo de desarrollo ya presentaba cierta predisposición a invertir tiempo para conseguir una automatización de las tareas. Esto de debe a que en una empresa con poco personal, como es el caso, la automatización reporta beneficios más claros que en una empresa con decenas de empleados.

Como acabamos de enumerar, las ventajas potenciales son muchas, pero parar lograr implantarlo también hay que superar ciertos retos. En conjunto es un proceso muy complejo, con detalles dependientes de cada caso en particular, por tanto perfeccionarlo conlleva mucho tiempo.

La inversión inicial es alta, y con esto nos referimos también a la adquisición de *hardware* adicional, aspecto que explicaremos en el capítulo 4.

[1] Durante el capítulo de evaluación ofrecemos datos estadísticos sobre esto

Por último, hay que destacar que este modelo depende de todas las fases de los proyectos. La IC engloba las tareas de desarrollo y testeo, mientras que la EC extiende este proceso al despliegue. Por lo que todos los departamentos técnicos quedan implicados y tienen parte de responsabilidad del proceso total. En relación a esto último gira el tema de la sección 3.3.

Virtualización

Continuando con nuestra propuesta, aparte de construir una infraestructura para la EC, proponemos que todo esto se realice sobre una plataforma virtualizada. Esto tiene relación directa con el punto mencionado durante los objetivos para la optimización del uso de los recursos

Introduzcamos la virtualización brevemente. El objetivo primario de la virtualización es recrear un componente físico mediante software, emulando así todas las funciones de este. Este concepto es aplicable a múltiples ámbitos, podemos virtualizar: redes, componentes hardware, sistemas operativos, servidores, etc. Lo que se consigue son entornos aislados ejecutándose sobre la misma máquina física.

Las principales ventajas de esta tecnología son:

- Los entornos no comparten recursos, ni memoria, ni espacio de almacenamiento. Esto facilita que cada entorno tenga instaladas solo las dependencias necesarias para cumplir con su propósito, lo que proporciona ligereza.

- Los entornos virtualizados son autocontenidos, y por tanto fácilmente replicables. Esto mejora la portabilidad y la escalabilidad.

- Se aprovechan mejor los recursos. Si dedicáramos una máquina física a cada servicio/aplicación corremos el riesgo de sobredimensionarla, es decir, que existan recursos que no están siendo utilizados la mayoría del tiempo. Mediante la virtualización tenemos flexibilidad a la hora de asignar varios servicios a la misma máquina.

.

Dedicaremos el capítulo 6 a los detalles técnicos sobre esta tecnología.

DevOps

Para entender el concepto de DevOps debemos primero hablar del modelo de roles tradicional. Concretamente de las responsabilidades y funciones que toma cada rol.

Dentro de una empresa que se dedique al desarrollo software los roles necesarios son muchos: desde analistas, diseñadores, programadores, pasando por *testers*, técnicos de sistemas, etc.

Pero a nivel de finalidad podríamos clasificarlos todos en tres grandes grupos:

- Desarrolladores (*Developers*): responsables de diseñar y crear la aplicación propiamente dicha.
- Control de calidad o *Quality Assurance* (QA): responsables de realizar las pruebas pertinentes para verificar la funcionalidad de la aplicación, que los requisitos se hayan cumplido, detectar fallos, etc.
- Operaciones (*Information Technology Operations*): responsables de mantener y crear toda la infraestructura y el entorno para que la aplicación funcione correctamente. Este ámbito también es conocido comúnmente como administración de sistemas.

En este modelo, los roles son herméticos, y las responsabilidades de cada uno están bien definidas.

Nuestra propuesta consiste en pasar a un modelo mucho más comunicativo y con responsabilidades compartidas. El modelo propuesto se denomina comúnmente *DevOps*. El término fue acuñado durante una conferencia sobre desarrollo ágil en 2008, y fue ganando popularidad rápidamente.

El origen del término ya nos adelanta que este modelo tiene una relación directa con el desarrollo ágil de *software*. Por tanto encaja bien con los conceptos descritos en la sección 2.1, relacionados con la metodología de trabajo de la empresa.

DevOps pretende exportar las técnicas de desarrollo al ámbito de de la administración de sistemas, y en concreto las prácticas ágiles, como las descritas en la sección 2.1. A esto se le conoce como *Infrastructure as Code* [24].

Para implementar estos conceptos hay una serie de prácticas habituales entre las que se encuentran: el uso de la virtualización, la computación en la nube o la automatización de la fase de testeo. Estas tres prácticas encajan perfectamente con el marco de la empresa y las propuestas realizadas.

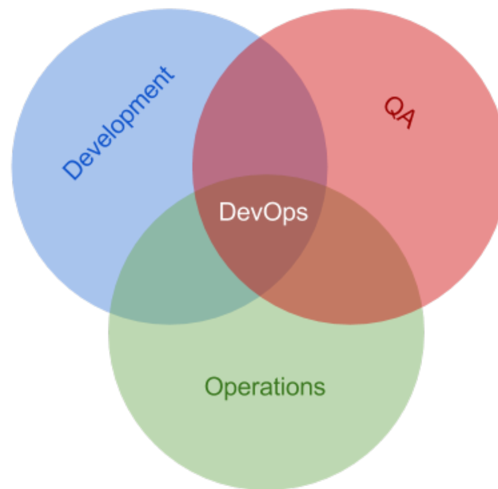

Figura 3.1: Estructura de los roles tradicionales frente a DevOps

En cuanto a la computación en la nube, esto encaja perfectamente con el marco de la empresa, ya que hemos explicado que toda la infraestructura de la empresa se basa en ella, por tanto el marco de la empresa encaja

Esto nos lleva a explicar que existe una clara relación entre el modelo DevOps y la EC. Como hemos mencionado en la sección 3.1, en la EC todos los roles se convierten es responsables del correcto funcionamiento del sistema. Por tanto, si las responsabilidades dejan de ser herméticas y se incrementa la comunicación, se facilita el proceso de implantación y el mantenimiento de la EC.

Casos similares en otras empresas

Después de realizar nuestras propuestas, mencionemos un par de casos reales para demostrar que estrategias similares han funcionado en otras empresas.

Mediante sistemas continuos como los propuestos [2] y pasando al modelo DevOps, Facebook consigue realizar dos despliegues en producción al día [27]. Pero el récord lo ostenta Amazon, que durante una conferencia en mayo del 2011, desveló que realizaba cambios en producción cada 11,6 segundos de media [21].

Otras empresas como Microsoft, Netflix, Mozilla o Flickr también han optado por estas soluciones.

[2] Nos referimos a la integración y entrega continua pero también al despliegue continuo, una sistema que describiremos con detalle en la sección 5.4, pero que es en definitiva muy similar a la EC

Datos analizados sobre los proyectos en grandes empresas muestran que el código se pone en producción 30 veces más rápido y los despliegues fallidos se han reducen en un 50 % [18]. Además queda reflejado que las empresas que más tiempo llevan utilizando estas prácticas son las que mejor rendimiento consiguen.

Dedicaremos la sección 7.1 para medir estos datos en nuestro caso particular.

Capítulo 4

Integración continua

En la sección 3.1 hemos presentado este concepto, pero durante este capítulo vamos a explicar en profundidad esta idea, cómo adaptarla a la arquitectura describa en el capítulo 2 y con qué herramientas contaremos para este propósito.

Implantar un sistema de IC estable es la antesala para poder evolucionar posteriormente hacia la EC. En primer lugar vamos a definir exactamente de qué estamos hablando al referirnos a IC.

Integración hace referencia al acto de combinar algo para que forme parte de un todo, en nuestro caso, ese algo algo es código Java y el todo es la aplicación final. Por otra parte "continuo/a" indica que una vez algo empieza, nunca acaba. Centrándonos más en nuestro contexto, podemos entender la IC como el proceso mediante el cual conseguimos que todos los cambios de cada desarrollador sean combinados constantemente para formar el producto final.

Debido a que los entornos de cada desarrollador van divergiendo a medida que se realizan modificaciones en el código, la frecuencia con la que se integran los cambios entre los desarrolladores es inversamente proporcional al riesgo de que durante dicha integración surjan fallos, así se ilustra en la figura 4.1.

Para solucionar esto último, diseñaremos e un sistema que:

1. Cada vez que un desarrollador haga un cambio en el código y lo suba al SCV, todo el código de la aplicación sea recompilado, con la finalidad de añadir dicho cambio.

2. Se realicen los tests que procedan para comprobar que dicho cambio no tiene consecuencias inesperadas.

3. Si todos los tests concluyen con éxito, se genere un paquete capaz de ser instalado

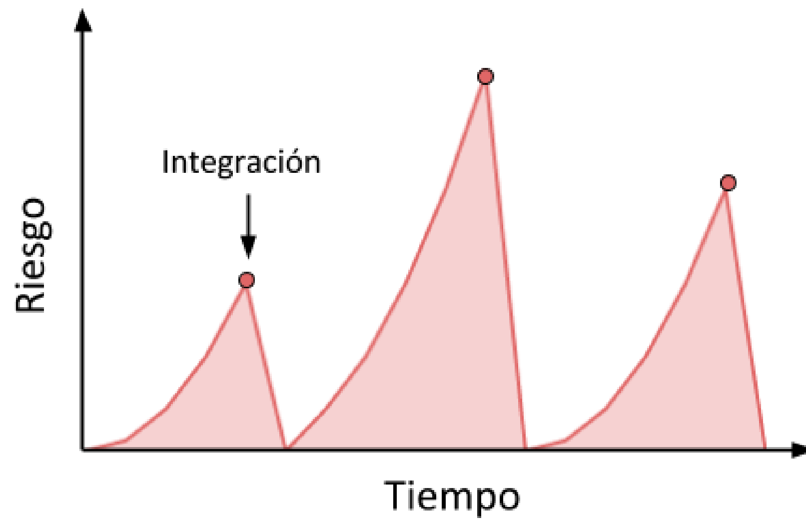

Figura 4.1: Riesgo respecto al tiempo mediante la integración tradicional

en producción.

Si realizásemos todo este proceso manualmente, invertiríamos demasiado tiempo y recursos en una tarea tan repetitiva. En cambio, si lo automatizamos, aparte de ahorrar tiempo, nos aseguramos de que el proceso es replicable, y que no depende de factores externos.

Un sistema de estas características nos proporciona múltiples ventajas, pero para llegar a implementarlo también encontraremos ciertos obstáculos, vamos a tratar ambos aspectos a continuación [25].

Empecemos por las ventajas:

- Los riesgos asumidos son menores: la IC sigue el principio: *"Si algo duele, hazlo con más frecuencia y adelanta el dolor"* [26] Al integrar el código varias veces al día, es más fácil que los errores sean detectados y resueltos en ese mismo momento. Cuanto más se tarde en descubrir un defecto, más costosa será su resolución. Mediante la IC conseguiremos un control de riesgos similar al de la figura 4.2

- Se mejora la visibilidad y la transparencia del proyecto: mediante la IC, los desarrolladores obtienen datos reales sobre el estado de la aplicación con más frecuencia, pudiendo usar estos para tomar decisiones más fundamentadas.

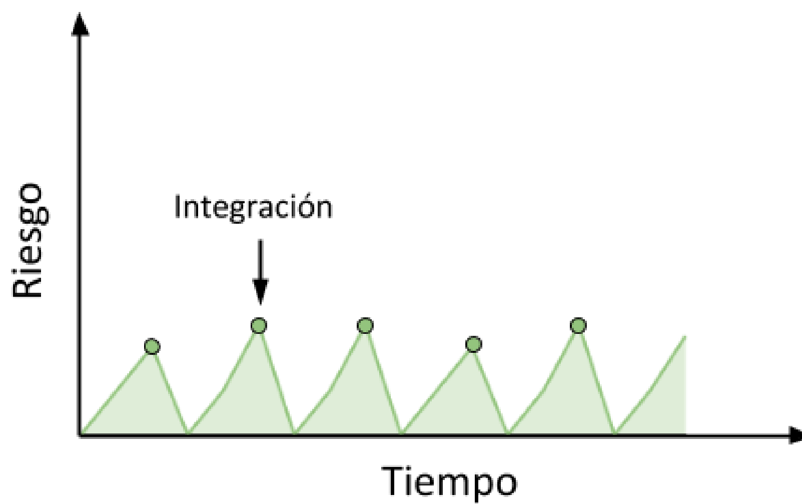

Figura 4.2: Riesgo respecto al tiempo mediante la integración continua

Un ejemplo clarificador: si durante las integraciones son muy frecuentes los errores provocados por un componente en concreto, esto puede llevar a los desarrolladores a prestar especial atención a este o bien sustituirlo por otro más estable.

Para conseguir este flujo de información sobre el estado de las integraciones, es necesario que nuestro sistema proporcione un *feedback* sobre el estado de cada proyecto. Normalmente esto es llevado a cabo mediante correos electrónicos enviados automáticamente desde del servidor de IC, que comunican al equipo de desarrollo si la integración ha fallado o ha terminado satisfactoriamente.

- Se mejora la confianza en el producto: al realizar todos los tests durante cada integración, los desarrolladores no deben preocuparse de que un cambio tenga un impacto catastrófico en la aplicación.

Como ya hemos anunciado, un sistema de IC puede conllevar ciertas dificultades durante su implantación, que de no ser abordadas correctamente puede convertir a este en inviable, enumeremos algunas de ellas:

- Dependiendo del sistema de partida, migrar hacia la IC puede suponer demasiados cambios en la metodología de trabajo. Sobretodo en proyectos que ya llevan mucho tiempo en marcha, establecer este tipo de cambios estructurales puede resultar demasiado costoso. En tales casos lo mejor es hacerlo paso a paso y de forma incremental para que el periodo de aclimatación de los equipos técnicos sea más llevadero.

- Se requiere de una inversión de tiempo y recursos, tanto para instaurar el sistema como para mantenerlo. Depender de un sistema de IC supone que si el proceso no está funcionando correctamente, el desarrollo de la aplicación queda bloqueado. Por tanto, mantener el proceso de integración libre de errores es más prioritario que el propio desarrollo de la aplicación.

  Recordemos que este sistema encaja bien con el modelo de responsabilidades de DevOps (sección 3.3), donde el correcto funcionamiento de la infraestructura es responsabilidad de todos.

- Requiere que los desarrolladores compilen y ejecuten la aplicación con sus cambios de manera local antes de integrarla con la de los demás. De lo contrario corremos el riesgo de que el filtro de todos los errores menores sea el sistema de IC. Tal y como acabamos de mencionar en el punto anterior, mantener el correcto funcionamiento del proceso se debe priorizar frente a otras tareas, así que se debe reducir al máximo los errores de este tipo.

- El sistema de IC suele requerir de una máquina dedicada debido a la utilización de recursos que supone estar compilando y lanzando tests continuamente, lo que supone un gasto adicional en hardware. Esta desventaja se ve mitigada si como en nuestro caso, contamos con una estructura de servidores en la nube, que nos aporta mayor flexibilidad y un gasto más contenido.

A pesar de todas esta desventajas, el potencial ahorro de tiempo sigue siendo mucho mayor al tiempo invertido, sobretodo si se desea mantener el proyecto durante un largo periodo de tiempo. El tiempo invertido disminuye a medida que se adquiere experiencia, mientras que el tiempo ahorrado es constante.

Una vez presentadas la IC junto con sus ventajas y desventajas, durante las siguientes secciones, vamos a hablar de todos los cambios que esta ha requerido. Entramos pues, en un contexto más específico y técnico ligado directamente al entorno y las herramientas propias de la empresa

Adaptación de las ramas de Git

En la sección 2.2.1 hemos explicado la metodología de trabajo de los desarrolladores, mediante el uso de una rama para cada nueva característica y una rama *master* como versión estable de la aplicación. Este sistema resulta poco útil para nuestro propósito, ya que debemos tener en cuenta que el proceso de IC es capaz de lanzarse de forma automática basándose en si ha habido cambios en el código. Por cambio entendemos un nuevo *commit* por parte de cualquier desarrollador a una rama específica del repositorio remoto. Si la única rama común a todos los desarrolladores es *master*, que además es la

encarga de mantener la versión estable, esta arquitectura se queda corta para nuestro objetivo.

Es necesario pues, crear un nueva rama común para todos los desarrolladores cuya finalidad es actuar de disparador para los procesos de integración. Por su cercanía al proceso de desarrollo, denominamos a esta rama "*development*".

Los desarrolladores deben seguir usando ramas locales durante la implementación de las tareas asignadas, donde pueden ir guardando el código a medio acabar. Una vez acabada cada tarea, se pueden subir su código a *development*, lo que lanzará el proceso de integración.

Como hemos descrito al inicio del capítulo, tratamos de conseguir un proceso continuo, por tanto es necesario realizar *commits* con frecuencia. Pero debemos tener en cuenta que no tiene sentido subir código a la rama *development* sin haberse asegurado antes de que el código compile correctamente y parezca tener los efectos deseados. De lo contrario estaríamos disparando procesos de integración con una alta probabilidad de fallo. En definitiva, hay que alcanzar un equilibrio entre evitar subir código sin acabar, propenso a errores, y una frecuencia de *commits* relativamente alta.

El siguiente paso es ahondar en las acciones llevadas a cabo durante el proceso de integración. Si lo lleváramos a cabo de forma manual, después de la compilación deberíamos realizar los tests. Sobre la creación, ejecución de estos profundizaremos más adelante en este capítulo en la sección 4.2, de momento vamos a centrarnos solo en el orden de ejecución y en que momento se lanzarán.

Después de la compilación ejecutaremos los tests unitarios y solo si estos acaban satisfactoriamente se lanzarán los tests de integración. En general podemos dividir proceso total de compilación y testeo en dos subprocesos:

1. Compilación y tests unitarios
2. Tests de integración.

El primer subproceso emula lo que suele ejecutar cualquier desarrollador de forma local, es decir, compilar su propio código y probar de forma aislada los cambios en los que trabaja, este proceso es relativamente rápido y sencillo. Por otro lado, el segundo subproceso, encargado de validar si los cambios funcionan bien en sintonía con todos los otros componentes de la aplicación es una tarea mucho más costosa.

Siguiendo este planteamiento se decidió crear una rama llamada *integration*, cuya funcionalidad consiste en albergar el código que ha pasado el primer subproceso, es decir, ha sido compilado y ha pasado los test unitarios correctamente. Y otra rama para

almacenar el código que ha superado todos los tests de integración, a dicha rama la llamaremos *release*.

Para que el código que llega a esta última rama se convierta finalmente en una versión potencialmente desplegable, solo resta incrementar el número de versión y el empaquetado. Una vez se ha cambiado el número de versión, el código ya puede pasar a la rama *master*. En la sección 4.3, veremos como se ha implementado todo el proceso que estamos describiendo.

Testeo

Es obvio que para que los procesos de integración se ejecuten sin intervención humana, todos los tests deben de ejecutarse automáticamente. La automatización del testeo requiere definir los casos de prueba, y el mejor momento para hacer esto es justo después de haber finalizado la implementación de la funcionalidad a testear.

Una buena política es que no sea el mismo desarrollador que ha implementado la funcionalidad el que diseñe los tests. De esta manera conseguimos que estos sean definidos de manera mucho más objetiva, evitando que comprueben solo los casos para los que el código está preparado.

A la hora de definir los casos de prueba debemos también tener cierta intención de conseguir que el código falle, reduciendo al máximo las asunciones. El código que solo es capaz de tratar como entrada los casos lógicos acabará fallando en algún momento.

Por otra parte debemos ver el testeo como un proceso atómico, es decir, o se completa totalmente con éxito o se considera fallido, no existe término medio. En nuestro caso, el encargado de abortar el proceso en caso de que un test falle es Maven.

Hasta el momento, nos hemos centrado solo en los tests unitarios y de intragración, pero para entregar un producto es necesario también realizar unos tests de aceptación, es decir, comprobar si lo que hemos implementado es lo que el cliente realmente esperaba. Este proceso tiene una fuerte relación con Scrum (sección 2.1. El cliente [1] participa en la reunión para revisar el *sprint* con la finalidad de decidir si el entregable cumple o no con sus expectativas.

[1] Cumpliendo el rol de *Product Owner*, que por simplicidad, durante la sección dedicada a Scrum no hemos explicado

Tests Unitarios

Este tipo de tests busca probar la funcionalidad de cada componente de forma aislada. Un claro ejemplo: si una para la implementación de un funcionalidad se necesita escribir en la BD, su test unitario consistiría en comprobar que dicha escritura se ha ha reflejado en la BD. Estos test son imprescindibles durante el proceso de implementación. Al tratarse de un aspecto más propio de la fase de desarrollo y al estar ya implantado en la empresa antes de nuestro trabajo, no vamos a profundizar más en él.

El encargado de ejecutar los tests unitarios es Maven, y dicho proceso será lanzado desde nuestro servidor de IC.

Tests de Integración

Es en este tipo de tests donde entra en juego la automatización. Hemos de aclarar que la creación de tests de integración automáticos no se ha resuelto como parte de este trabajo. Debido a su conocimiento sobre la aplicación, se ha delegado esta tarea al equipo de desarrolladores.

Por tanto no detallaremos como se han realizados dichos tests, pero si mencionaremos los aspectos básicos y las herramientas utilizadas, para poder integrarlas posteriormente en nuestro sistema de IC.

Para la creación de los tests de integración se ha requerido de dos herramientas fundamentales:

- Vaadin Testbench[2]: la empresa utiliza Vaadin para el desarrollo de la capa de presentación, un *framework* que pasa de código Java a HTML, CSS y JavaScript, los lenguajes que entienden los navegadores web.

  Pues bien, esta herramienta directamente relacionada está específicamente diseñada para automatizar las tareas de testeo en interfaces creadas con Vaadin [10]. Esta automatización se realiza emulando las acciones que realizaría un usuario real sobre la aplicación, moviendo el cursor por la pantalla, cambiando de ventana, haciendo *clicks*, etc. Con esto se consigue poner a prueba el funcionamiento conjunto de todos los componentes de la aplicación.

- PhantomJS: Vaadin Testbench requiere de un navegador web para poder ejecutar los tests, y además lo hace utilizando la Interfaz gráfica de usuario o *Graphic User Interface* (GUI). Esto supone un problema si pretendemos que este proceso se lleve a cabo en un servidor en la nube, sin GUI.

[2]Esta herramienta está basada en Selenium, un entorno ideado testear aplicaciones web interactuando con el navegador web mediante *scripts*

> PhantomJS [3] es una herramienta de código libre que permite emular en segundo plano un navegador web completo, simulando incluso la interacción con la GUI.

Conectando Vaadin Testbench con PhantomJS conseguimos que los tests de integración funcionen incluso en un entorno sin interfaz, como suele ser el caso de los servidores.

Vamos a mencionar que si se desea, los tests de integración se pueden ejecutar en la máquina local durante el desarrollo. Esto permite ver en cada momento como el cursor se mueve automáticamente por la pantalla realizando los tests definidos. Este proceso resulta útil a la hora de depurar errores, ya que podemos ver el momento exacto en el que falla un test.

Como acabamos de mencionar, las herramientas utilizadas para la creación están pensadas para poder ejecutarse en un servidor, sobre este servidor y su relación con la IC vamos a hablar en la siguiente sección.

Servidor de IC: Jenkins

Durante el capítulo hemos estado describiendo la IC y los cambios necesarios para poder implantarla, pero no hemos mencionado todavía la herramienta encargado de llevarla a cabo. Este será el objetivo de Jenkins, la herramienta escogida para para ejercer de servidor de IC

Jenkins es una herramienta de código libre escrita en Java, nació a finales del 2010 como un *fork*[4] de Hudson, debido a las tensiones entre los desarrolladores y Oracle, después de que dicha empresa comprara a la empresa desarrolladora de Hudson. Después de su lanzamiento los sondeos muestran que el 75 % de usuarios de Hudson migraron a Jenkins [29], lo que se traduce en una ámplia comunidad de usuarios.

Gracias a esa ámplia aceptación Jenkins cuenta con una gran variedad de *plugins* para añadirle funcionalidades, personalizarlo, o bien conseguir que funcione en sintonía con otras herramientas. Dichos *plugins* son la clave para adaptar Jenkins al entorno de nuestra empresa, permitiendo integrarlo con Maven, Git, Docker e incluso con algunos servicios de los proveedores de *cloud computing* como *Amazon Web Services* (AWS).

La principal funcionalidad de Jenkins son las tareas, o *jobs* en inglés. El concepto detrás de los *jobs* es muy sencillo: se trata de una serie de acciones configurables que pueden lanzarse de forma automática.

A nivel teórico, podemos concebir a Jenkins como un miembro más del equipo de

[3] `http://phantomjs.org/`
[4] Ramificación de otro programa

desarrollo, al cual delegaremos todas las tareas repetitivas[5]. Pero esto nos obliga también a mantener la máquina que hospeda Jenkins con las mismas herramientas que cualquier otro desarrollador, nuestro caso Java, Maven, MySQL, etc. También debemos tener en cuenta que Jenkins se conectará a nuestro SCV mediante SSH como un desarrollador más, por lo que hay que concederle acceso. Dicha configuración queda fuera del contenido de este trabajo.

Para explicar como funciona Jenkins, vamos a definir una aplicación de ejemplo, cuya finalidad es puramente didáctica, sobre la que crearemos un conjunto de *jobs* y poder así explicar los distintos aspectos a configurar.

Para dicha aplicación vamos a tomar como base el sistema de ramas descrito en la sección 4.1 y a crear los *jobs* de manera que cumplan con los procesos de la figura 4.3[6].

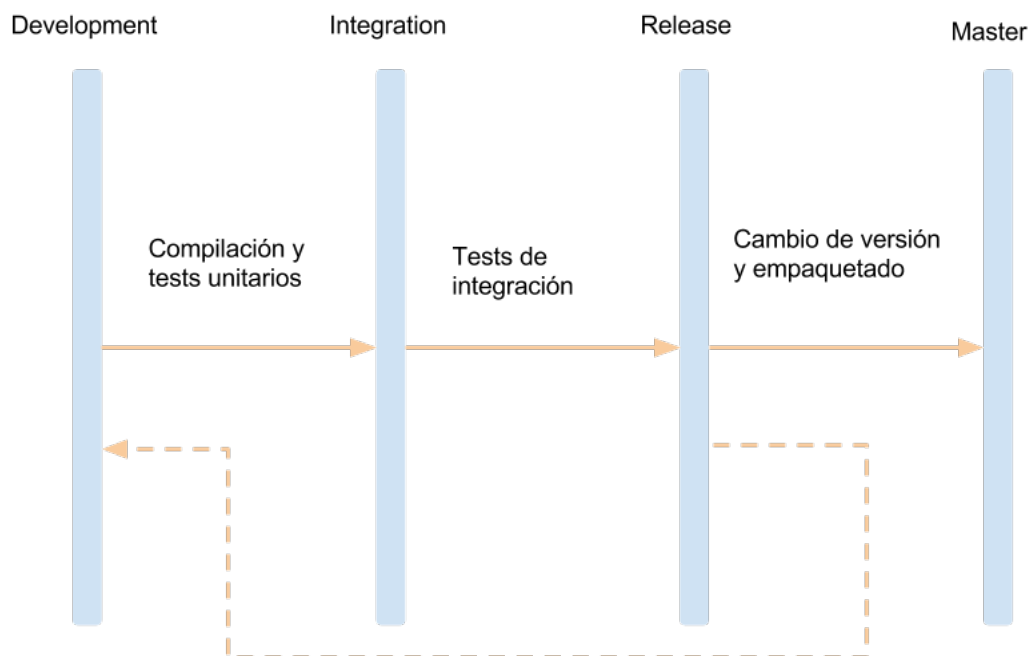

Figura 4.3: Proceso de integración y su relación con las ramas de Git

Antes de pasar a nuestro primer ejemplo debemos saber que Jenkins divide cada *job* en distintas fases, cada una con una finalidad concreta.

[5] Por este motivo el icono de Jenkins es un mayordomo

[6] La explicación de cada proceso se realiza durante las secciones 4.3.1, 4.3.2 y 4.3.3

Las más importantes son:

- General: aspectos generales como el nombre del *job*, una descripción y en qué situaciones no debe lanzarse. Esto último lo explicaremos con detalle en la sección 4.3.1.
- *Source Control Management*: definir la configuración de acceso al SCV.
- *Build Triggers*: definición de que eventos capaces de disparan el *job*.
- *Pre Steps*: pasos previos a realizar para preparar el entorno.
- *Build*: configuración del objetivo fundamental del *job*.
- *Build Settings*: configuración sobre las alertas para estar al corriente del estado del proceso.
- *Post Steps*: acciones a ejecutar después del *build*, normalmente interacciónes con el SCV o el lanzamiento de otros *jobs*.

A continuación introducimos los distintos *jobs* que se utilizan para la ejecución del proceso de integración.

Primer *job*: fase de *development*

En esta sección vamos a crear un *job* que compile la aplicación y lance los tests unitarios. Para ello, el primer paso, y común en todos los *jobs* es configurar la sección *Source Code Management* donde deberemos darle acceso a Jenkins para que se descargue el código.

Para ello debemos definir la URL del repositorio, seleccionar unas credenciales preconfiguradas para que Jenkins tenga acceso como cualquier otro desarrollador.

En la imagen observamos que existe un botón (*Add*) para añadir más detalles sobre acceso al SCV, en nuestro caso vamos a configurar dos opciones extra. En primer lugar definimos que la rama de la que queremos que Jenkins se descargue el código es *development*. En segundo lugar, configuramos mediante una expresión regular que queremos que Jenkins ignore todos los *commits* que contengan cierto mensaje, la utilidad de este último campo cobra sentido al configurar la siguiente fase: los disparadores del proceso.

Jenkins llama a los eventos que pueden lanzar un *job*: "*Build triggers*". El *trigger* más básico consiste en planificar los *jobs* para sean lanzados empleando la notación del programador de tareas Cron de Unix [4]. Dicha nomenclatura consiste en una serie de

Figura 4.4: Configuración para que Jenkins pueda acceder a Git

campos para configurar cuando debe ser lanzado un comando, veamos esto en detalle en la figura 4.5.

Esta característica permite programar los *jobs* en periodos de tiempo donde la máquina está ociosa, habitualmente por la noche. Pero, como hemos explicado previamente en la sección 4.1, el *trigger* que realmente resulta útil para nuestro propósito es el que dispara el proceso cuando ha habido un cambio en una rama de git.

Para ello debemos seleccionar la casilla "*Poll* SCM" y configurar mediante la nomenclatura de la figura 4.5[7] cada cuanto deseamos que se compruebe si ha habido cambios en el SCV, tal y como se muestra en la siguiente imagen 4.6.

En la figura 4.6 podemos observar que se ha añadido el prefijo `H/` delante de los minutos, la consecuencia de esto es que el *trigger* se active de forma aleatoria en cualquier minuto dentro del rango definido. Añadiendo este modificador, conseguimos distribuir más la carga de trabajo, evitando una concurrencia excesiva en caso de que todos los *jobs* se lanzaran a la vez.

En la sección *general*, hemos configurado que todos los *commits* que contengan cierto patrón sean ignorados (pág.: 34), para entender el propósito de esto debemos anticipar que al finalizar todo el proceso de integración, Jenkins subirá el código a la rama *development*. Lo hará mediante un *commit* con un mensaje automático (configurable), para que sea fácilmente identificable. Pues bien, si no queremos que Jenkins entre en bucle

[7] El campo "[comando a ejecutar]" se omite en Jenkins, ya que la finalidad es lanzar el *job*, no un simple comando

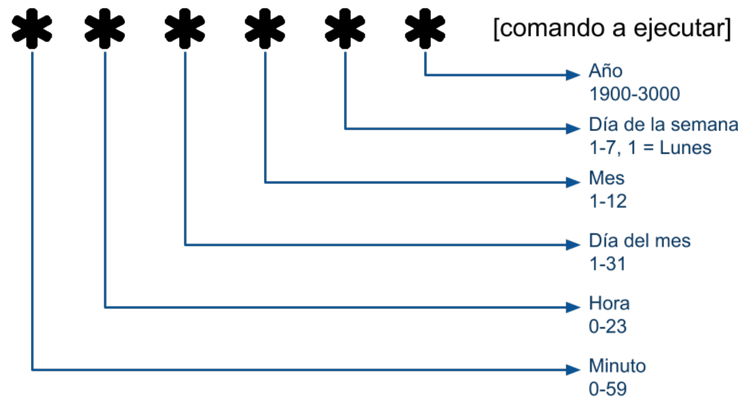

Figura 4.5: Significado de los campos de Cron

y detecte este *commit* como si se tratara de un cambio por parte de un desarrollador, volviendo a lanzar así otro proceso de integración, debemos configurar cada *job* para que se ignoren todos los *commits* con dicho prefijo, que es precisamente lo que se configura en la figura 4.4.

A continuación vamos a preparar el entorno para que la compilación y los tests unitarios sean ejecutados. La finalidad de esta fase es asegurarse de que el entorno donde se llavarán a cabo está limpio y no contiene ningún detalle que pueda afectar a estos haciendo que no fueran replicables en otros entornos.

Para ello Jenkins nos permite definir múltiples acciones en la fase "*Pre Steps*", en nuestro caso particular, como el objetivo es limpiar la BD para asegurarnos de que no quedan datos de integraciones anteriores, vamos a lanzar un *script* en BASH, tal y como se muestra en la siguiente imagen.

Pasemos ahora a la acción principal del *job*, el *build*, como se ha mencionado en la sección 2.3.1 al describir la arquitectura de la empresa, el ciclo de vida la aplicación lo gestiona Maven, por tanto en esta fase solo tenemos que delegar la compilación y la ejecución de los tests a Maven. Para ello debemos aclarar que previamente hemos instalado un *plugin* para que integrar Jenkins con Maven, y ejecutar comandos de este en los *jobs*. Como se muestra a continuación basta con que configuremos la ruta del POM, que en nuestro caso se encuentra en la raíz del directorio del proyecto, y los parámetros a aplicar al comando de Maven[8].

Una vez Maven finalice su trabajo, el código ya podrá pasar a la siguiente rama, pero

[8]La definición de los parámetros empleados se realiza en conjunto con el equipo de desarrollo y son específicos para cada aplicación, por tanto no entraremos en más detalles acerca de estos

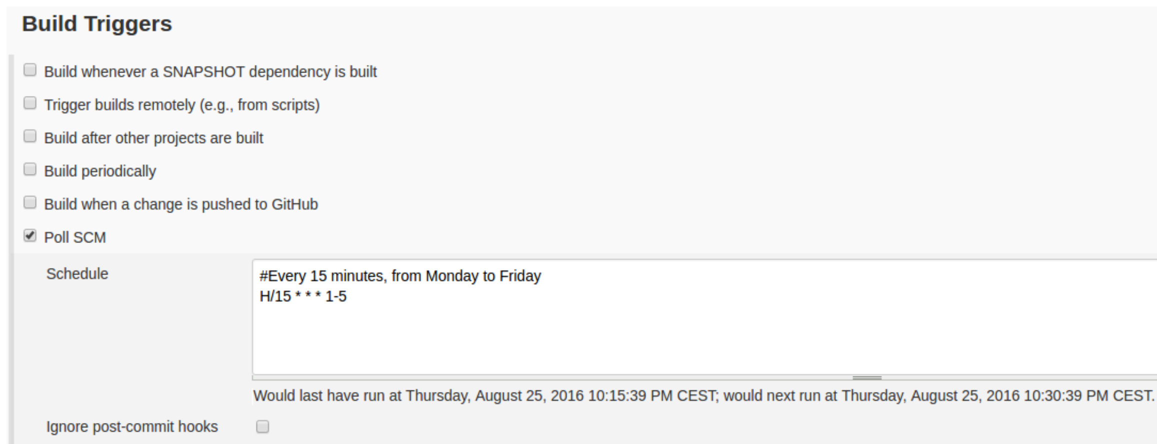

Figura 4.6: Frecuencia con la que se consulta la presencia de nuevos *commits* en el SCV

Figura 4.7: Comandos para limpiar la BD

antes todavía debemos configurar algunas funciones adicionales.

Entre ellas se encuentra el sistema de *feedback* del que hemos hablado al inicio de este capítulo (concrétamente en la pág.: 27). Jenkins facilita esta comunicación mediante correos electrónicos enviados automáticamente, en los que podemos configurar las direcciones a las que queremos que sean enviados.

Tal y como se aprecia en la imagen 4.9, las acciones posteriores pueden ser configuradas para que sean ejecutadas si Maven finaliza correctamente, si falla o independientemente del resultado.

Ahora ya podemos definir las acciones finales del *job*, que en nuestro caso consistirán en:

Figura 4.8: Configuración de los parámetros de Maven para ejecutar la compilación y los tests unitarios

Figura 4.9: Configuración del sistema de *feedback* de Jenkins

- subir el código a la rama *integration* del nuestro repositorio.
- lanzar otro *job*, que será el encargado que tomar el relevo en el proceso de integración, realizando los tests de integración.

Para lo primero, podemos hacer que Jenkins suba el código a un repositorio configurado en un *plugin* que integra Jenkins con Git, además de elegir si queremos que esto se realice solo en caso de que la compilación y los tests unitarios han finalizado correctamente. Finalmente, para lanzar el *job* que continúe con el proceso de integración basta con definir otra acción llama *Build other projects* configurando el nombre del *job* que deseamos lanzar. Esto queda ilustrado en la figura 4.10.

Segundo *job*: fase de integración

En esta sección vamos a continuar desarrollando el proceso de integración, contruyendo esta vez un *job* que ejecute los tests de integración.

El *job* a construir es muy similar al descrito en la sección anterior, por tanto no

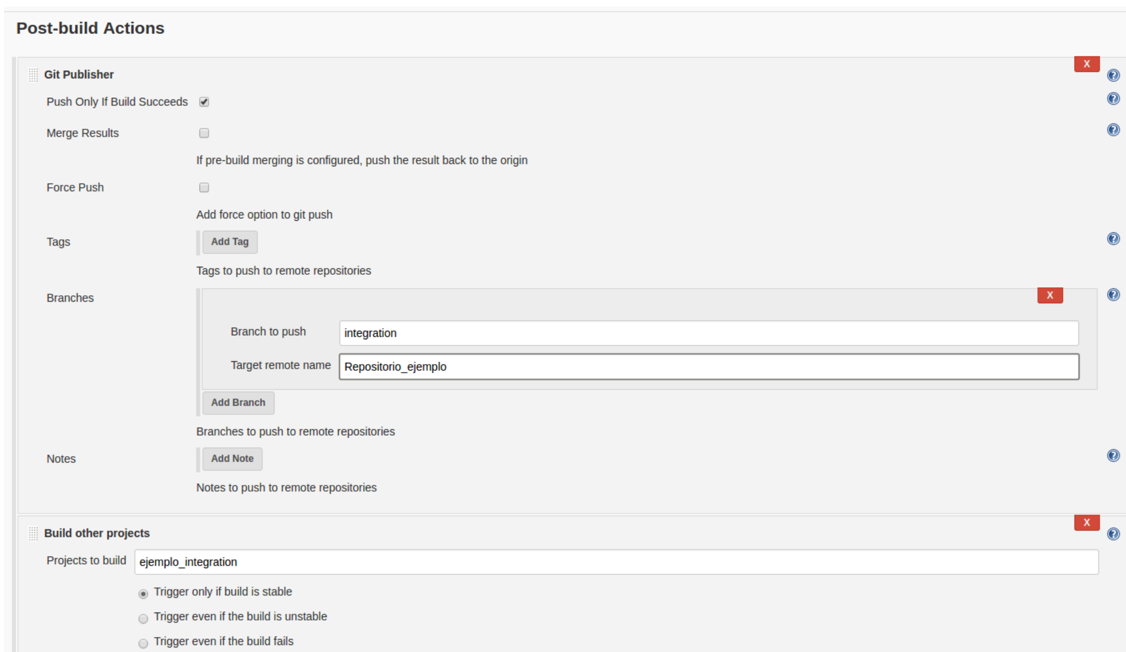

Figura 4.10: Configuración para que Jenkins suba el código a *integration* una vez acabe el *job*

volveremos a mencionar todos los detalles ya explicados anteriormente.

No obstante, existe una diferencia importante a tener en cuenta respecto al *job* de desarrollo. Los tests de integación requieren el uso exclusivo de algunos recursos para ser ejecutados, lo que nos obliga a asegurarnos de solo hay un *job* realizando tests de integración al mismo tiempo. Entre estos recursos se entra la herramienta PhantomJS, introducida en la seccion 4.2.2.

Para ello Jenkins nos permite bloquear un *job* en caso de que otro esté ejecutándose. Como se muestra en la figura 4.11, esto lo podemos configurar marcando la casilla "*Block build if certain jobs are running*" y configurando una expresión regular para que nuestro *job* no se lance si ya hay uno ejecutándose cuyo nombre encaje con dicho patrón.

En la imagen observamos que utilizamos una expresión para bloquear cualquier otro *job* de integración, pero también los otros *jobs* relacionados con el mismo proyecto, para asegurarse de que el proceso finaliza completamente antes de que se inicie otro.

Las razones de este bloqueo las explicaremos durante la siguiente sección, ya que son consecuencia directa de lo que se desarrolla en esta.

La configuración para el acceso al código fuente es igual a la de la sección anterior

Figura 4.11: Configuración del bloqueo entre *jobs* de integración y del mismo proyecto

con la diferencia de que ahora el código debe ser descargado de la rama *integration*.

En este *job*, también vamos a ejecutar un *script* para limpiar la BD, en este caso totalmente idéntico al que ya hemos explicado en la figura 4.7.

En cuanto a la fase *Build*, el comando a ejecutar varía levemente ya que ahora este debe invocar a los tests de integración en vez de compilar y ejecutar los tests unitarios. Como ya hemos mencionado, los parámetros de este comando y por consecuencia la configuración de Maven es responsabilidad del equipo de desarrollo.

Figura 4.12: Configuración de Maven para ejecutar los tests de integración

En la figura 4.8 podemos observar que en el comando se indica explícitamente que se salten los tests unitarios (ya realizados anteriormente) y se ejecuten los de integración.

En cuanto a los correos electrónicos de *feedback* se mantienen exactamente igual que los descritos en la sección anterior. Finalmente, debemos tener en cuenta que ahora el *job* que debe lanzarse cuando el actual acabe es “ejemplo_release”, para pasar así al último paso del proceso de integración.

Tercer *job*: fase de *release*

En este último paso crearemos un *job* que al igual que realice las últimas modificaciones necesarias y empaquete la aplicación para poder ser desplegada.

En líneas generales la configuración es muy similar a la descrita en las dos secciones anteriores.

En concreto, los objetivos de este *job* son:

1. Cambiar el número de versión en los ficheros pertinentes.
2. Empaquetar la aplicación.
3. Subir el paquete a Artifactory.
4. Subir el código a las ramas *master* y *development*.

Los tres primeros puntos se delegan a Maven, al igual que en los dos primeros *jobs*.

Concretamente el primer punto, es el responsable de que los *jobs* no puedan ser concurrentes y requieran de bloqueos. Durante la sección anterior hemos ilustrado como configurar reglas (figura 4.11) para que las distintas fases sean secuenciales. Este proceso es necesario para evitar colisiones en los números de versión. Para entender esto último debemos tener en cuenta que el código entre las ramas de *development* e *integration* no varía, ya que ni la compilación ni los tests realizan ningún cambio. Debido al cambio del número de versión, el código si que varía al finalizar esta fase. Si permitiésemos que dos *jobs* del mismo proyecto se lanzasen de forma concurrente ambos acabarían produciendo un paquete con el mismo número de versión. Obviamente esto último es totalmente indeseable.

Sobre la configuración de la cuarta acción ya hemos hablado en anteriores ocasiones (pág.: 37). En cuanto a su finalidad hay dos aspectos a explicar:

- Se sube el código a dichas ramas añadiendo una etiqueta junto al mensaje del *commit*. En dicha etiqueta consta la versión generada de la aplicación. En Git esto sirve poder identificar a que versión corresponde cada *commit*, permitiendo identificar que cambios se realizaron en el código entre cada versión.
- El *push* a *development* sirve para que si los desarrolladores se descargan de nuevo el proyecto, lo hagan de la versión publicada más reciente.

Un aspecto que no hemos mencionado hasta el momento, es que los *jobs* de Jenkins se ejecutan en un entorno de trabajo o *workspace*. Dicho *workspace* es utilizado por

Jenkins como directorio para guardar archivos relacionados con la ejecución de los *jobs*. Mencionamos esto ahora, para recalcar que aparte de subir el WAR a Artifactory, este también se encuentra presente en el *workspace*. Este aspecto nos será útil más adelante, en la sección 5.2.1 le daremos uso a esta particularidad.

Una vez el paquete ha sido generado y subido a Artifactory el proceso de integración ha finalizado.

Extendiendo la IC

De todo el proceso descrito durante este capítulo, el rediseñado de las ramas de Git, los tests unitarios y de integración y las tareas relacionadas con Maven han sido llevadas en gran parte por el equipo de desarrollo.

Por contrapartida, toda la configuración de Jenkins, así como de sus detalles de funcionamiento: accesos al SCV, planificación de *triggers*, configuración de bloqueo, *feedback*, etc. han sido desarrolladas en solitario.

Durante este capítulo nos hemos centrado en conseguir mediante las herramientas que nos ofrece Jenkins que el trabajo de los desarrolladores se integre de forma continua.

Pero el proceso no debe que detenerse aquí, el siguiente paso es extender esta automatización a la fase de despliegue, creando un sistema que facilite la instalación al igual que hemos creado un sistema que facilita la integración. Este proceso es lo que nos lleva a la EC y el DC, ya mencionados anteriormente, y a los que vamos a dedicar el siguiente capítulo.

Capítulo 5

Entrega continua

En este capítulo vamos a describir como conseguir, mediante la integración de ciertas herramientas con nuestro servidor de IC, que nuestros despliegues se puedan realizar de forma automática y bajo demanda.

Cuando nos referimos a despliegue, debemos saber que este puede ser en un entorno de testeo o de desarrollo. La máquina donde se realizará el despliegue no afecta a la ejecución de este [19], es decir, en rasgos generales[1] el proceso de instalación será el mismo independientemente de que en que entorno despleguemos. Además debemos asegurarnos de que este proceso es idempotente, para ello debemos dejar el entorno de despliegue siempre en el mismo estado [11].

Como explicamos al final del capítulo anterior, debemos considerar la EC como la evolución natural de la IC. Por tanto debemos tener este último sistema bien implementado, lo que a su vez implica tener un conjunto de tests eficaces a la hora de detectar errores.

El objetivo de la EC no se limita a automatizar el despliegue con la única finalidad de evitar todos los errores que este pueda producir, sino que además se pretender acelerar al máximo, para que el equipo técnico pueda dedicarse a seguir desarrollando. Con despliegues más rápidos también conseguimos agilizar el *feedback* por parte el cliente, que es uno de los pilares para realizar Scrum correctamente, como se ha explicado en la sección 2.1.

La EC es un método que encaja con la filosofía DevOps, ya que tanto el proceso de integración como el despliegue requiere de conocimientos técnicos sobre las distintas partes del proyecto, así como la colaboración y el entendimiento entre desarrolladores y

[1] Es posible que la ejecución no sea exactamente idéntica debido a pequeñas variaciones, como por ejemplo la ruta donde está instalado Tomcat o distintos usuarios en la BD

el equipo de sistemas [11]. Esto provoca que el sistema de EC sea responsabilidad de todos los miembros del equipo.

Para definir como llevar a cabo todo esto vamos a explicar en primer lugar como conseguir una instalación de forma automática, haciendo hincapié en las herramientas que intervienen en dicho proceso. Posteriormente seguiremos el esquema del capítulo anterior creando un *job* de ejemplo en Jenkins que lance dicha instalación y explicaremos las configuraciones a medida que vayamos definiendo el flujo de tareas.

Instalación automática

Desde la introducción de este trabajo no hemos dejado de reincidir en la importancia de la automatización como paso necesario para alcanzar nuestro objetivo. Ahora vamos a enfocar esta idea en como conseguir que el código resultante del proceso de integración pase manos de los clientes, con la mínima intervención humana.

Para ello necesitamos de una herramienta que se ajuste a nuestro entorno particular y que nos permita instrumentar todo este proceso en múltiples servidores. En la empresa se decidió utilizar Ansible, una herramienta a la que dedicaremos la siguiente sección. Pero debemos saber que no es la única herramienta de este tipo, existe todo un abanico de herramientas muy similares como *Chef*, *Puppet* o *Salt*. La elección de usar Ansible fue por su simplicidad.

Ansible

Para entender el propósito y la finalidad de Ansible lo mejor es empezar por su etimología. El término proviene del ámbito de la ciencia ficción [12], donde da nombre a una tecnología que permite realizar comunicaciones instantáneas.

En un nivel más técnico y dejando las metáforas de lado, podríamos decir que Ansible permite crear una lista ordenada de tareas y delegarlas para que sean ejecutadas en una máquina remota.

Antes de continuar, vamos a aclara los motivos por los que nos decantamos por Ansible frente a las otras herramientas mencionadas. El motivo de más peso es la facilidad para iniciarse, ya que desde RedHat, la empresa desarrolladora, se ha priorizado la simplicidad por encima de otros aspectos. Esto se traduce en que podemos crear nuestra primera tarea con un par de nociones básicas pero también en una instalación realmente sencilla: lo único que debemos tener instalado en el servidor remoto para gestionarlo desde Ansible es un servidor SSH y Python, mientras que en la máquina desde la que

```
---
- name: Ansible example
  hosts: server_1
  user: ana
  sudo: yes

  tasks:
    - name: Simple hello world
      shell: echo "Hello world" > example.txt

    - name: Loop hello world
      shell: echo "{{ item }}" >> example.txt
      with_items:
        - Hello
        - World
```

Figura 5.1: *Playbook* de ejemplo

se dirigen los despliegues podemos instalar Ansible con un par de comandos[2]. Otras herramientas similares cuentan con arquitecturas más complejas, donde se requieren la instalación de un agente en el servidor remoto para poder ser gestionado.

Esto último nos da paso a explicar como funciona internamente Ansible, pero antes de entrar en materia vamos a definir el vocabulario necesario para entender la explicación. En primer lugar, Ansible denomina a los *scripts* que ejecuta *playbooks*. En estos se utiliza la sintaxis YAML, que es similar a XML o a *JavaScript Object Notation* (JSON) pero que está especialmente concebida para que sea fácilmente legible por humanos [15].

Como su nombre indica, los *playbooks* contienen *plays*. Un *plays* es una estructura que consta de dos objetos:

- Una definición de a que servidor se delegarán los comandos y con que rol o usuario se ejecutarán los comandos finales[3].
- Un o más tareas o *tasks* en inglés. Que es donde se programan las comandos a ejecutar.

Como vemos en la figura 5.1, entre los comandos ejecutados dentro de una tarea podemos utilizar funciones ya definidas, como por ejemplo bucles. Esto es porqué Ansible

[2]Ansible se puede instalar desde `pip`, el gestor de paquetes de Python

[3]Es importante que nos aseguremos de que dicho usuario tiene todos los permisos necesarios para ejecutar los comandos

cuenta con una serie de módulos que nos sirven de librería para facilitarnos la implementación y que no tengamos que programarlas a tan bajo nivel. Por otra parte, se ha intentado que esta capa de abstracción sea lo más fina posible. Un ejemplo clarificador: si tuviéramos que instalar un paquete en dos máquinas cuyo gestor de paquetes no es el mismo no podríamos usar la misma instrucción para ambos, sino que deberíamos recurrir a módulos o instrucciones distintas [16]. Esto puede sonar como una desventaja, pero como ya hemos mencionado, en Ansible cobra mucho peso la simplicidad, y tener una capa de abstracción muy amplia obligaría a programar los *playbooks* como si de un lenguaje de alto nivel se tratara, teniendo una curva de aprendizaje mucho más pronunciada.

Para poder desplegar, lo primero que debemos hacer es definir la lista de servidores en la que queremos lanzar los *playbooks*. Para ello debemos comprender que Ansible realizará las conexiones con los servidores remotos mediante SSH, y que es necesario definir las ruta del servidor (ya sea mediante una URL o bien su IP) junto con el usuario que deseamos utilizar para conexión, todo ello en un fichero que Ansible denomina *hosts*.

```
[production]
server1.example.com ansible_ssh_user=ana

[testing]
server2.example.com ansible_ssh_user=bob
```

Figura 5.2: Ejemplo del fichero `/etc/ansible/hosts`

En la figura vemos como a cada entrada se le asigna un nombre o etiqueta, utilizados posteriormente en los *plays*. Ansible denomina al conjunto de sistemas configurados donde se pueden ejecutar *playbooks* inventario o *inventory* en inglés, y también nos da la posibilidad de organizar los elementos del inventario en grupos, para que compartan variables comunes y puedan ser dirigidos despliegues contra dicho grupo como si se tratara de un *host* único. Esta arquitectura junto con un optimizado sistema para gestionar un gran número de conexiones SSH simultaneas [17] hacen de Ansible una herramienta diseñada pensando en la escalabilidad.

Durante la ejecución de un *playbook* que implica a múltiples servidores, Ansible realiza una conexión SSH a todos de forma concurrente y ejecuta las tareas en el orden definido y de forma simultanea. Es importante saber que Ansible no pasa a ejecutar la siguiente tarea hasta que la actual haya sido concluida en todos los *hosts*.

Una vez explicados todos los conceptos básicos pasemos a definir lo pasos a realizar para conseguir automatizar el despliegue. Para ello es necesario que fijemos otra vez el contexto y lo acotemos al de la empresa.

Pasos para desplegar la aplicación

En esta sección y las que vienen a continuación volvemos acotar el contexto al de la empresa, por tanto, los pasos seguidos tienen relación directa con las herramientas utilizadas en el desarrollo del nuestros proyectos software.

En las siguiente secciones nos centramos en explicar la finalidad de cada *task* que forma parte del *playbook* del despliegue, entraremos en detalles técnicos para entender el proceso, pero no en el código YAML.

Antes de comenzar hay un detalle importante a tener en cuenta: nuestro sistema automatizado está ideado para realizar despliegues incrementales, es decir, funciona siempre que haya previamente una versión de la aplicación instalada. Esto obliga a que la primera vez que instalemos la aplicación debemos hacerlo de manera manual. El motivo de este comportamiento se debe a que, como explicaremos a continuación, nuestras tareas de Ansible están diseñadas para encontrarse el sistema ya funcionando, es decir, con una versión ya instalada, la base de datos en funcionamiento, el fichero de configuración en su ruta predefinida, etc.

En nuestro caso esto no ha supuesto ningún problema ya que cuando empezamos a implementa el sistema de EC la aplicación ya llevaba meses instalada en los servidores.

Aun así, si quisiéramos desplegar en una máquina nueva, basta con que despleguemos el WAR e instalemos la base de datos. A partir de dicho momento ya podremos utilizar el sistema de despliegue automático, ya que aunque este falle, como explicaremos durante las siguientes secciones, el sistema está preparado para darnos directivas sobre como solucionar los errores. Lo cual nos conducirá a configurar todos los aspectos necesarios.

Vamos pues a explicar los pasos requeridos para alcanzar nuestra meta.

Despliegue del WAR

Durante la sección 2.4.1 hemos dicho ya, que la empresa emplea Tomcat como contenedor de aplicaciones *web*, y que para desplegar una aplicación en este, basta con situar el fichero WAR que contiene la contiene en la carpeta `webapps`. A partir de ese momento Tomcat se encarga de construir toda la estructura necesaria. También hemos explicado en el capítulo anterior que una vez Jenkins finaliza el proceso de integración sube el WAR resultante a Artifactory.

Por tanto, puede parecer lógico que para instalar nuestra aplicación baste con crear una tarea que acceda a Artifactory, descargue el WAR y otra que lo copie en el servidor deseado. Aunque esto sería correcto, no es lo más óptimo, ya que como explicaremos

más adelante en la sección 5.3, Jenkins y Ansible se ejecutan sobre la misma máquina. Por tanto resulta más cómodo y eficiente obtener el WAR directamente desde Jenkins. Para esto rescatamos un concepto mencionado en la sección 4.3.3, donde explicamos la existencia de un *workspace* donde Jenkins guarda el WAR antes de subirlo a Artifactory. Tomando el WAR del *workspace* de Jenkins en vez de descargarlo de Artifactory nos ahorra transferencias innecesarias entre máquinas.

Por tanto crearemos nuestra primera tarea que accederá a la ruta de Jenkins donde se almacena el WAR y lo comprimirá junto a otros ficheros en un ".zip", para acelerar su transmisión a través de la red. Una segunda tarea lo copiará al servidor deseado, pero a una ruta temporal, ya que debemos descomprimirlo antes de hacerlo llegar a Tomcat. Adicionalmente, en vez de sustituir la versión instalada con la nueva, vamos a mover la antigua a una carpeta a modo de *backup*, para poder volver a la versión anterior en caso de que algo no saliera bien.

Por último, solo nos falta copiar el WAR de la nueva versión a la carpeta *webapps* dentro de la carpeta de instalación de Tomcat.

Pero nuestro despliegue no acaba aquí, habitualmente las aplicaciones dependen de más componentes en las máquinas en las que se instalan, y existen detalles a configurar para su correcto funcionamiento.

El siguiente paso consiste en configurar uno de esos detalles.

Sincronización de los ficheros de configuración

Es común que las programas utilicen uno o más ficheros para almacenar la configuración general, como por ejemplo el idioma de la interfaz, servicios web a los que acceder, usuario de la BD,

En nuestro caso, además la empresa sigue una política que consiste en hacer que ciertas funcionalidades de la aplicación se puedan deshabilitar mediante los ficheros de configuración. Esto es útil para poder ofrecer servicios distintos dependiendo del cliente pero también para poder desactivar una funcionalidad en la que se ha descubierto algún error hasta que sea arreglada y desplegada de nuevo.

Todo esto es importante porque una de las fuentes de errores más frecuente a la hora de realizar despliegues se encuentra en los ficheros de configuración. Si se desea añadir una funcionalidad nueva a la aplicación y para implementarla se requiere modificar el fichero de configuración, debemos conseguir que dicha modificación sea trasladada también a los entornos de despliegue. Pero como pasa cierto tiempo entre el desarrollo y la instalación, es común que se olvide replicar las nuevas configuraciones en todos los entornos

Para solucionar esto, vamos a diseñar un mecanismo que nos compare el fichero de configuración del servidor en el que vamos a desplegar con el fichero de configuración de la aplicación durante su desarrollo. En caso de que los ficheros coincidan en cuanto a parámetros de configuración, el despliegue puede continuar sin problemas. Pero en caso de que el fichero que se halla en el servidor carezca o le sobre algún parámetro, el despliegue debe abortarse ya que debido a estas diferencias la aplicación podría no funcionar correctamente.

Debemos tener en cuenta que habitualmente las configuraciones son tuplas del estilo "clave = valor". A nosotros solo nos interesa comprobar las claves, ignorando el valor que se les da, ya que este podría requerir un valor distinto al que nosotros hemos configurado en el fichero de configuración de desarrollo.

Por otra parte también debemos saber que las aplicaciones de la empresa tienen dos ficheros de configuración uno con la configuración común de todas las aplicaciones y otro específico para cada una de ellas.

Para cumplir con nuestro propósito vamos debemos crear un conjunto de tareas que extraiga todas las claves de las configuraciones de ambos ficheros, y las compare con todas las claves del fichero de configuración de desarrollo. En nuestro caso dicha comparación la realizamos mediante la utilidad *diff* de Linux

En caso de que los ficheros difieran, el despliegue se aborta y se imprimen por pantalla las diferencias para que el administrador pueda resolverlas manualmente. Las discrepancias solo pueden ser resueltas por un humano. Desde la dirección de la empresa se decidió no automatizar este mecanismo para asegurarse de que se da un trato más personalizado a cada cliente, ya que no todos tienen porqué tener el mismo valor en todas las *settings*.

En la siguiente sección hablaremos sobre otro detalle importante, esta vez referente al acceso del cliente a la aplicación.

Actualización de URLs

Una vez desplegada la aplicación en Tomcat, debemos realizar ciertas configuraciones para que esta sea fácilmente accesible desde el exterior. Dado que la aplicación contiene el número de la versión dentro del WAR, la URL generada por Tomcat cambiará con cada nuevo despliegue. Lo que resulta incómodo a la hora de acceder a esta. Para facilitar el acceso a los clientes usamos un redireccionamiento de URLs, usando un nombre genérico y fácil de recordar, el cual enlazamos con la URL cambiante.

Esto lo conseguimos mediante una conjunto de tareas en Ansible que obtienen el número de versión de la aplicación a partir del WAR, y que en cada despliegue modifican el redireccionamiento para que la URL genérica apunte a la nueva versión.

En la siguiente sección abordaremos los problemas que pueden surgir durante los despliegues en relación a la BD.

Actualización de la base de datos

Por último, otro de los principales focos de problemas a la hora de realizar el despliegue tiene que ver con la estructura de la base de datos. Es común que para desarrollar ciertas características sea necesario realizar modificaciones o añadir tablas o columnas a la BD. De no replicar dichas modificaciones en la BD de producción al realizar un despliegue, provocaremos que la aplicación falle.

Para solventar este problema vamos a seguir una estrategia similar a la utilizada en la sección anterior, pero en vez de crear dos tareas que recolecten la estructura de las BD para compararlas en busca de diferencias vamos a utilizar una herramienta llamada SchemaSync [9] que lo hará por nosotros. Esta herramienta aparte de encontrar las diferencias entre los esquemas nos va a facilitar un archivo con sentencias *Structured Query Language* (SQL) (*script: patch* o parche) para resolver dichas discrepancias. Es decir, si en la BD de desarrollo tenemos una tabla que no se encuentra en la BD de producción el SchemaSync nos creara una sentencia del tipo: `create table [nombre];`, así solo tenemos que ejecutar el parche para resolver las diferencias.

Aparte de generarnos un parche, como acabamos de explicar, SchemaSync también nos genera otro archivo SQL (*script: revert* o deshacer) con las sentencias inversas del parche por si queremos deshacer los cambios aplicados. Así pues, si en el parche hemos creado una tabla, en el *revert* tendremos una sentencia para borrarla.

Ahora bien, esta funcionalidad resulta muy útil si solo queremos añadir tablas o columnas, ya que podríamos automatizar el proceso de sincronización, lanzando el parche que generar SchemaSync y resolviendo así las diferencias. Pero en la vida real también es común renombrar o eliminar una columna o una tabla por motivos de diseño. En SQL renombrar una columna implica borrarla y volver a crearla, lo que implica que en el proceso se pierden los datos almacenados. SchemaSync nos proporciona las sentencias para realizarlo, pero la BD de producción contiene datos importantes de clientes reales, y por seguridad no podemos ejecutar automáticamente las sentencias que borren datos.

Esto nos lleva a la conclusión de que en caso de que exista alguna acción que borre datos, se requiere la intervención humana para tomar la decisión de si la columna o tabla debe ser eliminada completamente o se trata solo de un renombrado.

Si se trata de un renombrado debemos crear una columna adicional, copiar todos los datos de la columna original y posteriormente borrarla.

Por tanto, el automatismo referente a este proceso se limita a comprobar si las BBDD

son iguales estructuralmente, y en caso de no serlo se aborta el despliegue indicando la diferencias en las dos BD.

Despliegue desde Jenkins

Durante el capítulo nos hemos centrado en la automatización del despliegue, pero es necesario que definamos también como encaja dicho proceso con la IC definida en el capítulo anterior.

Una vez más el encargado de orquestar el todo esto es Jenkins. Al igual que desde Jenkins hemos delegado el proceso de compilación y testeo a Maven, haremos lo mismo con Ansible.

Por tanto vamos a crear un *job* desde el que se lanzarán todos los pasos del proceso de despliegue. En primer lugar, al igual que hemos hecho en algunos *jobs* del capítulo anterior, vamos a definir unas reglas de bloqueo para que el *job* se encole en vez de lanzarse si ya hay otro ejecutándose. En concreto el *job* que podría provocar problemas es el que genera el WAR. La configuración es similar a la mostrada en la figura 4.11 del apartado 4.3.3, pero en este caso el patrón utilizado es: `ejemplo_release`.

Lo siguiente a configurar es el servidor sobre el que se va a realizar el despliegue. Jenkins nos ofrece la posibilidad de crear *jobs* con diversos parámetros de entrada, nosotros configuraremos una lista de posibles valores junto con el nombre de una variable. Por ejemplo, podemos definir una variable que defina el servidor destino sobre el que desplegar. De este modo cuando lancemos el *job* se nos solicitará que elijamos uno de los servidores disponibles, tal y como se ilustra en la Figura 5.3

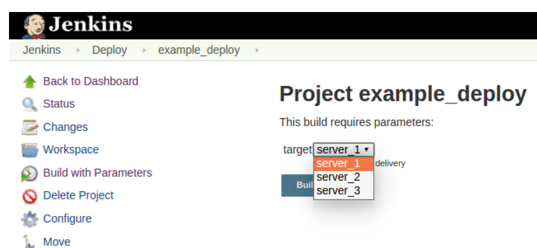

Figura 5.3: Ejecución de un *job* parametrizado

El siguiente paso es la configuración relativa al acceso al SCV, ya que los *playbooks* de Ansible, como el resto del código, también se almacenan en Git, y para poder lanzarlos Jenkins debe descargárselos previamente. Vamos a omitir más detalles ya que ya hemos explicado este tipo de configuración anteriormente, en la sección 4.3 y en la Figura 4.4.

A continuación vamos a lanzar las tareas explicadas en las secciones anteriores para ejecutar el despliegue. Para ello a lanzaremos tres *playbooks* que se encargarán de:

1. Comprobar el estado de los ficheros de configuración (Sección 5.2.2).
2. Comprueba que las estructuras de las BBDD son iguales (Sección 5.2.4)
3. Copiar el WAR y modificar las URLs (Secciones 5.2.1 y 5.2.3)

Podemos lanzar estos *playbooks* mediante comandos en BASH, pero en vez de lanzar los tres *playbooks* a la vez, vamos a lanzarlos por orden, y configurar que solo se lance el siguiente en caso de que el anterior ha finalizad satisfactoriamente. Esto lo conseguimos mediante la opción "Conditional step", que nos permite ejecutar un *script* en función de si se cumple cierta condición. Jenkins ofrece un amplio abanico de condiciones posibles, nosotros usaremos como condición el estado actual del *job*.

En la Figura 5.4 podemos ver como el primer comando se lanza sin condición, ya que no viene precedido de ningún otro que pueda fallar, y los siguientes solo se lanzan si el anterior ha terminado con éxito.

Ahora solo nos falta decidir el *trigger*, este aspecto de configuración cobra especial importancia en los *jobs* de despliegue. Ya que dependiendo de como lo configuremos estaremos escogiendo entre un sistema de entrega continua o uno de despliegue continuo. Vamos a dedicar la siguiente sección a este tema.

Entrega continua frente a despliegue continuo

En esencia, el DC es una extensión mínima de la EC, la única diferencia que existe entre ambos procesos es que en la EC el objetivo es dejar el despliegue preparado para hacerlo bajo demanda, tal y como hemos explicado durante este capítulo, mientras que en el DC el proceso de despliegue se lanza sin intervención humana. Es decir, una vez se inicia el un proceso de IC, y siempre que este concluya con éxito se realizará un despliegue de una nueva versión del producto.

Al final de la sección anterior hemos mencionado la importancia de la configuración del *trigger* del *job*, ya que si deseamos tener un sistema de EC no debemos configurar este aspecto, lo que se traduce en que el *job* solo puede ser ejecutado manualmente. Mientras que si nuestro objetivo es el DC deberemos lanzar el *job* de despliegue cuando finalice la fase de IC.

Dependiendo del caso, el DC puede no resultar apropiado. Vamos a mencionar las razones por las que en la empresa se tomó la decisión de no lanzar los despliegues automáticamente:

Execute shell
Command
```
ansible-playbook -e "target=$target" check_config_files
```
See the list of available environment variables

Conditional step (single)
Run? Current build status
Worst status Success
Best status Success
Advanced...
Builder Execute shell
Command
```
ansible-playbook -e "target=$target" check_database
```
See the list of available environment variables

Conditional step (single)
Run? Current build status
Worst status Success
Best status Success
Advanced...
Builder Execute shell
Command
```
ansible-playbook -e "target=$target" deploy_war
```
See the list of available environment variables

Figura 5.4: Ejecución de comandos en función de una condición

- Se desea tener el control sobre la frecuencia de despliegue. Nuestros clientes prefieren lidiar con despliegues programados en vez de estar cambiando de versión constantemente.

- El DC es más sensible a errores, ya que su estabilidad depende mucho más de la eficacia del conjunto de tests. Mientras que en la EC es más tolerante a errores, ya que da margen para realizar pruebas manuales antes de desplegar en producción.

- Como hemos comentado en la sección 5.2.4, ciertas decisiones no pueden ser tomadas por una máquina, ya que para tomar la decisión acertada se requiere conocer la finalidad de cada modificación. Los desplieguen que afectan a BBDD son uno de los grandes escollos de la DC.

- En general, los beneficios del DC no son tan notables si ya contamos con un

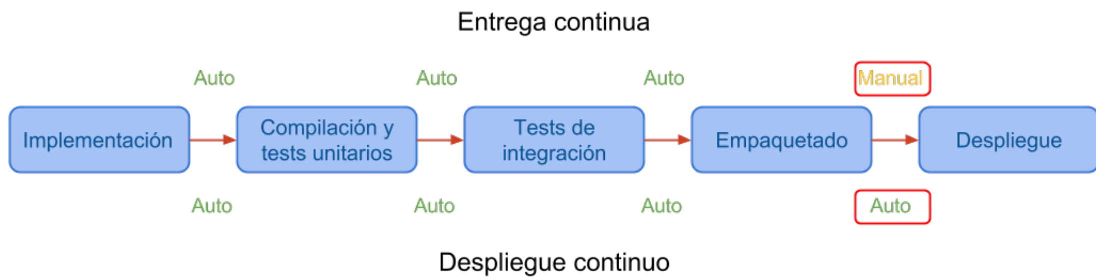

Figura 5.5: Diferencia entre entrega continua y despliegue continuo

sistema de EC. Es decir, no hay una gran ahorro en los costes de un sistema frente al otro. Los beneficios del DC se acentúan si realizamos decenas o incluso cientos de despliegues al día. Pero en nuestro caso, el desarrollo de una nueva funcionalidad suele tomar como mínimo una semana.

Capítulo 6

Virtualización como plataforma

Hasta ahora hemos descrito todas las herramientas y como funcionan en conjunto para alcanzar el objetivo que proponíamos en la introducción: la EC. Este ha sido uno de los temas troncales durante el trabajo. Durante este capítulo vamos a tratar otro tema troncal: la plataforma sobre la que hemos contruido todo lo descrito anteriormente.

En nuestro caso, nos hemos centrado en virtualizar las herramientas usadas durante el desarrollo y el proceso de EC: desde Git, pasando por MySQL, Jenkins, Artifactory, etc. Durante las siguientes secciones explicaremos los conceptos necesarios para dicha virtualización, pero no entraremos en detalles de como ha sido el proceso con cada herramienta, ya que esto requiere de un conocimiento más profundo sobre cada herramienta en particular, lo que queda fuera del ámbito de este trabajo.

Antes de continuar, hagamos hagamos un breve inciso para distinguir entre los dos grandes modelos de virtualización actuales: las máquina virtual y los contenedores.

En primer lugar, hay que destacar que ambos necesitan de un motor para ser gestionados, este será el encargado de asignar los recursos de la máquina física a cada entorno así como de gestionar su ciclo de vida. En el caso de las MV este se denomina *hypervisor* y también se encarga de comunicar el sistema operativo (SO) anfitrión con el de la MV.

En segundo lugar, ambos modelos nos permiten tener entornos aislados donde instalar solo las librerías y dependencias requeridas para cumplir con el servicio al que están destinados.

Pero la gran distinción entre ambos modelos es la presencia del sistema operativo en cada entorno. Mientras que las MV cuentan con SO propio [1], los contenedores se ejecutan sobre el sistema operativo de la máquina anfitrión. En términos de escalabilidad

[1] Este SO puede ser distinto al del anfitrión.

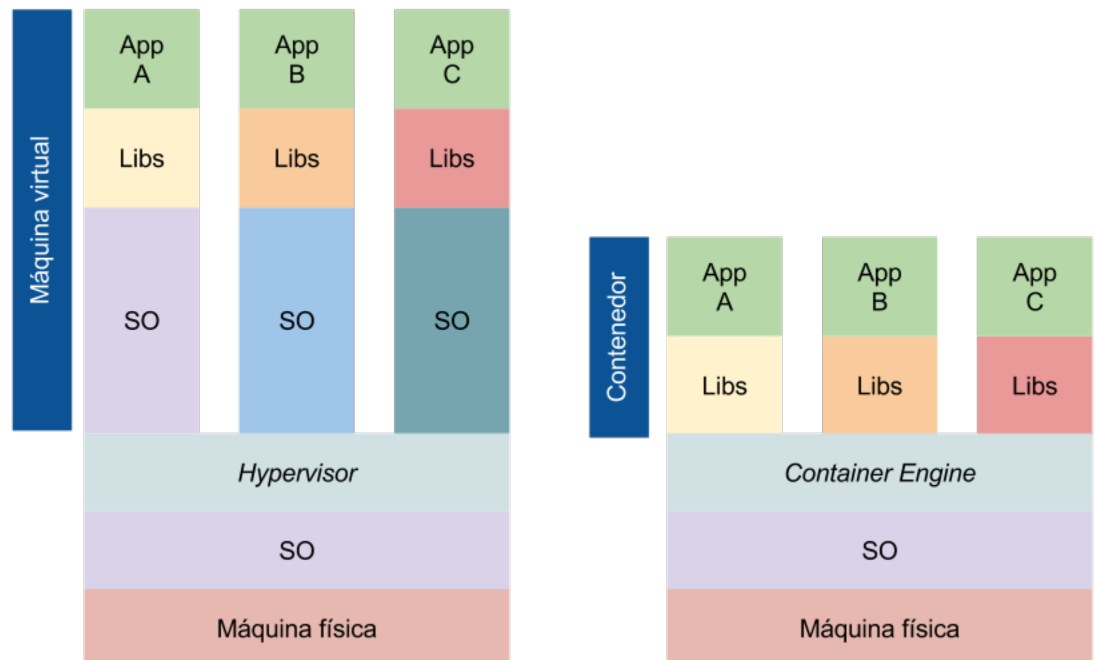

Figura 6.1: Distinción entre MV y contenedores

y rendimiento los contenedores son mucho más óptimos que las MV, ya que ejecutar varios SO de forma concurrente acarrea una importante sobrecarga sobre los recursos.

¿Qué es Docker?

Docker es una herramienta de virtualización mediante contenedores basada en el *kernel* de Linux. Fue anunciada a inicios de 2013, en una exposición de cinco minutos durante una conferencia de desarrolladores en California, donde ganó popularidad e interés por parte de los asistentes. A las semanas de este anuncio su código se liberó para que todo el mundo pudiera contribuir con el proyecto.

Ya hemos introducido la virtualización mediante contenedores en el apartado 3.2 de la introducció. Por tanto en el actual vamos a centrarnos en como implementa Docker dicho modelo, recalcando sus ventajas. Pero antes de continuar, definamos con detalle los dos términos que más utilizaremos durante este capítulo:

- Imagen: a nivel técnico, una imagen es un conjunto de capas superpuestas de un sistema de archivos. De una manera más simple, podríamos definir una imagen como una instantánea tomada sobre un sistema de archivos en cierto instante, por lo que una imagen es inmutable y no tiene estado. Gracias a la estructura en capas, las imagenes pueden ser superpuestas unas encima de otras. Una imagen se identifica mediante un nombre y una etiqueta, donde la etiqueta se suele para utilizar para expresar la versión concreta de la imagen [22]. Ejemplo: `image1:2.8`

- Contenedor: instancia de una imagen en tiempo de ejecución. Los contenedores tienen estado, una vez creado a partir de una imagen podemos encenderlo o apagarlo a voluntad.

  Un contenedor consta de tres componentes [5]:

  1. Una imagen Docker.
  2. Un entorno de ejecución.
  3. Un conjunto de instrucciones.

  Este último componente es especialmente importante ya que las instrucciones que definimos al arrancar un contenedor influyen en su ciclo de vida. Un contenedor se apaga cuando completa los comandos definidos cuando fue arrancado, si por ejemplo definiéramos como comando al iniciar un contenedor: `sleep 20`, el contenedor se apagaría tras veinte segundos.

  Cuando decimos que un contenedor se apaga nos estamos refiriendo a que su ejecución se ha detenido o pausado. No debemos confundir este término con el borrado de un contenedor, proceso que solo se puede realizar manualmente.

Podemos entender a Docker como una plataforma para desarrollar, transportar y ejecutar aplicaciones [1]. Para ofrecer estas funcionalidades Docker se ha convertido en una herramienta con una complejidad relativamente alta. Aún así, arquitectónicamenrte el núcleo de Docker se basa en el modelo cliente/servidor. Donde el servidor, se encarga de lanzar y gestionar los contenedores y desde el cliente se pueden lanzar instrucciones directamente al servidor. Docker también nos permite que desde un único cliente se pueda comunicar con varios servidores, además de ofrecernos herramientas como “Docker Swarm” para gestionar cientos de nodos en caso de que nuestra arquitectura requiera de una alta escalabilidad. Existe un tercer componente opcional en la arquitectura de Docker, el *registry* o registro de imágenes, cuya finalidad es servir de repositorio para nuestras imágenes.

Existe un *registry* público y gratuito llamado DockerHub, donde podemos subir nuestras imágenes o utilizar las creadas por otros usuarios. En DockerHub también podemos encontrar imágenes oficiales de los desarrolladores de algunas herramientas que ofertan como método de instalación una imagen ya preconfigurada.

Volviendo al cliente de Docker, hemos de explicar que la línea de comandos es la principal interfaz para la interacción con este. Existen clientes Docker para la mayoría de arquitecturas y sistemas operativos actuales[2]. Adicionalmente, Docker también cuenta con una API, que es utilizada para la comunicación entre el cliente y el servidor que acabamos de mencionar. Pero como dicha API es de código libre y está documentada también se suele utilizar desde herramientas externas o *scripts* [23].

Demos paso ahora a una sección donde explicaremos las distintas maneras de construir una imagen.

Creación de imágenes

Existen dos maneras de crear imágenes Docker, y en función de nuestras necesidades será conveniente que utilicemos una u otra.

Vamos a dedicar una subsección a cada una, explicando el proceso de creación y exponiendo las ventajas e inconvenientes de cada una.

Dockerfile

Este método consiste en crear las imágenes automáticamente, mediante lo que se denomina un *dockerfile*, que no es más que un *script* donde se parte de una imagen base y se van añadiendo capas a esta mediante la ejecución de comandos. Pongamos un breve ejemplo para ilustrar esta idea.

```
FROM ubuntu:14.04
ENV MYSQL_VERSION=5.6
RUN apt-get install mysql-server-${MYSQL_VERSION}
```

Figura 6.2: Ejemplo de *dockerfile*

Como resultado de la ejecución del *dockerfile* anterior obtendríamos una imagen basada en la versión 14.04 de Ubuntu[3] a la que se le añade una capa con la instalación de la versión 5.6 de MySQL.

El potencial de los *dockerfiles* reside en la idempotencia y en su portabilidad. Al ser un *script* podemos usar un SCV para distribuirlo, facilitando la replicación de las

[2]Para OS X y Windows la instalación de Docker debe realizarse mediante boot2docker (`http://boot2docker.io/`), una ligera MV que emula un entorno Linux

[3]La imagen base debe encontrarse en el conjunto de imágenes locales, en caso de no encontrarse se intenta descargar de DockerHub

imágenes en cualquier máquina.

Por otra parte, la creación del *dockerfile* puede resultar tediosa, ya que debemos comprobar que los comandos provocan los cambios deseados en cada paso,y además hay ciertas tareas que resultan difíciles de automatizarla debido a que están pensadas para interaccionar con el usuario mediante mensajes o ventanas emergentes.

`docker commit`

El segundo método para crear imágenes consiste en realizar un *commit* sobre un contenedor. Esta práctica, que ya hemos explicado en el apartado 2.2.1, trata al contenedor como un SCV lo haría con un fichero de texto.

Ejecutando `docker commit [container_name | ID] [image_name:tag]` crearemos una imagen con el nombre y etiqueta especificadas a partir del estado actual del contenedor. Durante este proceso el contenedor queda pausado para evitar que en caso de que haya datos cambiantes estos no acaben corruptos durante el guardado [8].

Este proceso es más rápido y sencillo que el descrito en la sección anterior, y nos permite crear distintas versiones de las imágenes en función del estado del contenedor.

Pero también es más propenso a errores ya que si la imagen va evolucionando corremos el riesgo de acabar arrastrando errores, ya que los *commits* son incrementales.

Después de explicar todos estos conceptos técnicos, centrémonos en como hemos llevado a cabo nuestro proceso particular para crear las imágenes de las herramientas del entorno de desarrollo.

Virtualización del entorno de desarrollo

Para iniciar la migración de las herramientas a Docker, se decidió empezar por las herramientas cuyo funcionamiento es mas sencillo, o cuyo número de dependencias fuera menor.

Básicamente, esto se decidió para que a la falta de experiencia inicial en entornos Docker no se sumara la dificultad para comprender el funcionamiento de la herramientas más complejas.

Por otro lado, en un inicio se escogió como estrategia para crear las imágenes se tomo la vía manual, habitualmente partiendo de la imagen de Ubuntu 14.04 y añadiendo capas a esta mediante *commits*. A medida que la experiencia con los detalles de funcionamiento

de Docker fue en aumento esto dinámica fue cambiando hacia el uso de *dockerfiles*.

Entre estos detalles se encuentra un aspecto que no hemos mencionado anteriormente pero que resulta básico a la hora de hacer que un contenedor sea accesible desde el exterior de la máquina que lo hospeda: la configuración de red.

Configuración de red

Debemos saber que al arrancar un contenedor, hay ciertos parámetros referentes a la conectividad de este que podemos editar. En primer lugar, podemos vincular un puerto de la máquina huésped a un puerto del contenedor, para que todo el tráfico que llegue a la máquina huésped a traves de ese puerto sea redirigido directamente al contenedor[4].

En segundo lugar, Docker también nos permite crear una Red Privada Virtual o *Virtual Private Network*s (VPNs) entre nuestros contenedores, asignando el rango de IPs deseado o directamente una IP estática. Si no definimos nada, por defecto todos los contenedores al crearse se conectan a una VPN con la máquina huésped, para que sean accesibles desde ella en todo momento, ya que de lo contrario ni siquiera se podrían redirigir los puertos.

Tener varios contenedores a modo de servidor en una misma máquina nos supone un problema. Por definición todos los servidores deben ser accesibles desde el exterior, pero al encontrarse en la misma máquina solo disponemos de una IP.

Para solucionar este problema tenemos dos alternativas:

- Mapear cada servicio en un puerto distinto. Esta solución es sencilla de llevar a cabo, ya que como hemos explicado podemos redirigir puertos directamente a los contenedores, pero conlleva una desventaja importante.

  Como toda la infraestructura de la empresa ya estaba en funcionamiento antes de que empezáramos con este proyecto, todo estaba configurado para acceder a los servidores con dominios distintos, ya que se encontraban en máquinas distintas. Por tanto no era deseable modificar todas esas configuraciones, presentes incluso en las aplicaciones, para que ahora accedieran a la misma IP.

- La utilización de un servidor *proxy* inverso. Para que este se encargara de redirigir cada conexión al contenedor pertinente en función del dominio con el que se realizaba la petición. En este caso solo hay que cambiar la IP que apunta cada dominio en el proveedor al que hemos contratado el *hosting*.

[4]Debido a la estrecha relación entre Docker y el *kernel* de Linux, al redirigir un puerto, si usamos `iptables` como *firewall* no es necesario que añadamos una regla para que el tráfico de ese puerto no sea descartado, este proceso se realiza automáticamente

Nosotros nos decantamos por la segunda opción, creando dicho *proxy* inverso en un contenedor adicional usando un servidor web, concretamente Nginx, para configurar el mapeado entre los dominios y las IPs. Adicionalmente, para que este sistema funcione, es necesario que las IPs de los contenedores sean accesibles desde el *proxy*. Con la finalidad de tener siempre localizados todos los servidores, se decidió crear una VPN y unir todos los nuevos contenedores a esta, asignándoles una IP estática y conocida por todos.

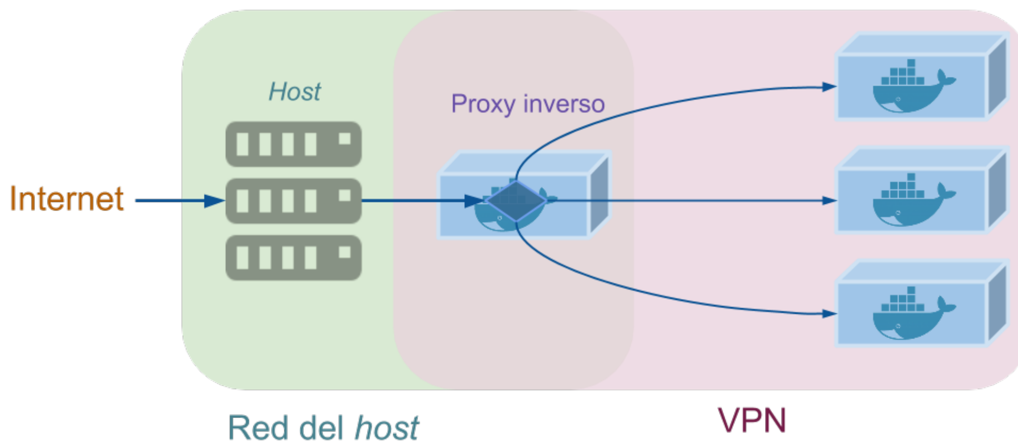

Figura 6.3: Funcionamiento del *proxy* inverso

Implantación de docker en la empresa

Volviendo a la idea de empezar la migración con las herramientas más simples, se decidió que Git es el mejor candidato, ya que, como apuntamos en 2.2.1, está estrechamente ligada a Linux y no requiere de aplicaciones externas. Para su correcto funcionamiento solo se requiere copiar las carpetas que contienen los proyectos Git desde el servidor antiguo, instalar un servidor ssh y configurar ambos para los desarrolladores puedan conectarse y escribir sobre las rutas del repositorio.

El resto de servicios tiene un proceso de una instalación más complejo, en el caso de Artifactory, Taiga y Liferay se requiere de una BD para que funcionen, mientras que Jenkins la necesita para realizar los tests.

A la hora de instalar las BBDD una opción habría sido dedicar un contenedor solo a estas, y hacerlo común a todos los servidores. Esto no ha sido posible debido a que no todos los servidores trabajan con el mismo gestor de BD, algunos están preparados por tabajar con MySQL mientras que otros lo hacen con PostgreSQL así que por simplicidad se optó por instalar cada BD en el mismo contenedor que su servidor.

Supervisor

Durante la migración de las herramientas más complejas, hay un aspecto especialmente problemático. Como hemos mencionado anteriormente, normalmente estos servidores requieren de varios procesos para funcionar correctamente, a esto se le añade que nuestra intención es que los contenedores estén totalmente operativos cuando se crean, sin necesidad de tener que arrancar los gestores de BBDD u otros procesos manualmente *a posteriori*.

La dificultad reside en que, como hemos comentado en la introducción de este mismo capítulo, los contenedores están diseñados para ejecutar un comando y apagarse después, si por ejemplo configuramos que dicho comando sea el que arranca la BD, cada vez que quisiéramos reiniciar el gestor de BD por cualquier motivo nuestro contenedor se apagaría.

Para evitar este problema, se decidió usar Supervisor, un sistema de control de procesos [5]. La idea detrás de su funcionamiento es sencilla, en vez ejecutar los procesos o servicios directamente, estos se delegan a Supervisor, desde el que podemos arrancarlos, reiniciarlos o detenerlos.

Una vez hecho esto, configuramos que el proceso que ejecute el contenedor sea Supervisor. Con esto conseguimos solucionamos el problema que los contenedores se apaguen si nos vemos obligados a reiniciar algún proceso y además conseguimos que todos los procesos requeridos para que el contenedor funcione se lancen automáticamente cuando este se arranca.

Esto último nos permite gestionar mucho más fácilmente nuestros contenedores, pudiendo hacerlo incluso desde un móvil.

Arquitectura resultante

Vamos a ilustrar brevemente cuál ha sido el resultado de la virtualización de la infraestructura de desarrollo. El fin de esta sección es comparar este resultado con el presentado durante el capítulo 2.

En la figura 6.4 observamos como antes cada servicio estaba hospedado en una instancia *cloud* distinta, con los recursos justos para su funcionamiento. No se han nombrado todos los servicios migrados debido a que algunos de ellos no tienen relación con nuestros objetivos, como por ejemplo: herramientas administrativas, herramientas de apoyo para las reuniones de *Scrum*, entornos para realizar demostraciones a los clientes, etc.

[5] http://supervisord.org/index.html

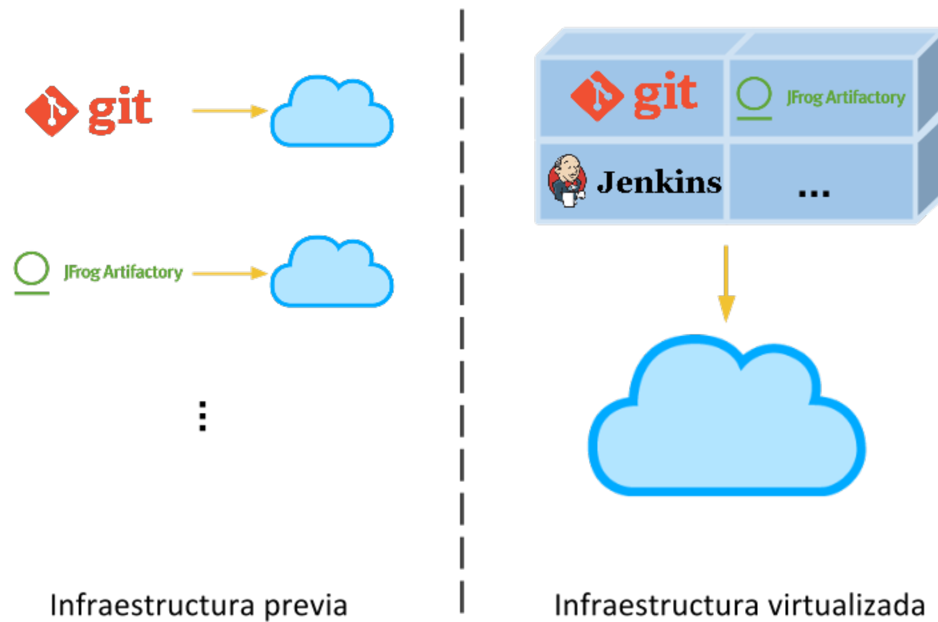

Figura 6.4: Comparativa entre la arquitectura previa y la arquitectura virtualizada

Para la virtualización se optó por prescindir de esas pequeñas instancias y agrupar todos los servicios en una sola. Para esto se decidió contratar con otro proveedor de *cloud computing* que ofrecía instancias más grandes y caras pero con una relación precio/recurso más rentable.

Por último, cabe destacar que se ha creado un sistema de *backups* automático que una vez al día genera una imagen a partir del estado de cada contenedor. Para este proceso se ha utilizado Cron (mencionado en: sección 4.3.1 y figura 4.5) junto con los conceptos descritos en la sección 6.2.2 para la creación de imágenes mediante *commits*.

Capítulo 7

Evaluación, conclusiones y trabajo futuro

Valoración de resultados

Procedamos a presentar algunos datos sobre los despliegues de las aplicaciones con la finalidad de comparar el escenario anterior y posterior a este trabajo. Vamos a evaluar el trabajo realizado, comprobando así si hemos alcanzado los objetivos propuestos. Para ello nos vamos a centrar en las mejoras de:

- Los tiempos de despliegue.
- La calidad del *software*.

Mejoras en los tiempos de despliegue

En primer lugar comparemos los tiempos que conlleva realizar un despliegue con el sistema aquí desarrollado frente al escenario previo.

Debido a la gran diferencia de tiempos que puede existir entre cada despliegue, los vamos a dividir en dos grandes grupos:

1. *Minor release*: sin modificaciones importantes en la BD ni los ficheros de configuración. Solo se ha de desplegar el WAR con la nueva versión y actualizar los ficheros para actualizar las URLs, tal y como se ha mencionado en la sección 5.2.3.

2. *Major release*: con cambios en la BD, nuevas configuraciones o nuevas librerías que deben ser instaladas para que las nuevas características de la aplicación funcionen.

Para obtener estos datos, antes de implantar el sistema de EC se anotaron los tiempos necesarios para realizar un despliegue. Esto tiempos han sido comparados con los actuales aportados por Jenkins, tal y como se presenta en las figuras 7.1 y 7.2.

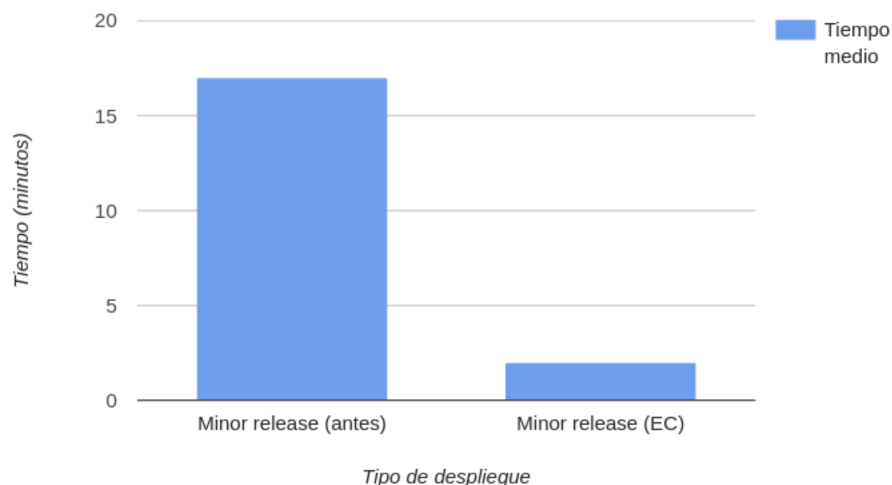

Figura 7.1: Comparativa entre el tiempo de despliegue de *minor releases*

En la figura 7.1 vemos que hay un ahorro sustancial en el tiempo, reduciéndose a la octava parte.

En la gráfica de la figura 7.2 vemos como el ahorro temporal en el caso de *major releases* está alrededor del 90 %.

Mejora en la calidad del *software*

El alto coste de los despliegues manuales venía acentuado por la necesidad de desplegar en algunos casos hasta dos o tres veces la aplicación para corregir fallos que no habían sido detectados previamente. Esto lo trataremos con más detenimiento en la siguiente gráfica (figura 7.3).

Mediante el testeo automático la inmensa mayoría de estos fallos han sido solucionados, lo que se traduce en una reducción del número de problemas reportados por los clientes de alrededor de un 80 % al mes.

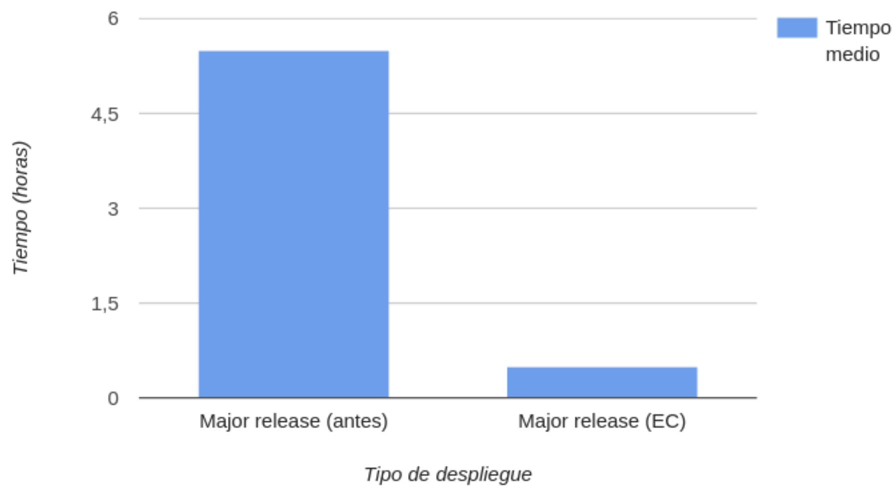

Figura 7.2: Comparativa entre el tiempo de despliegue de *major releases*

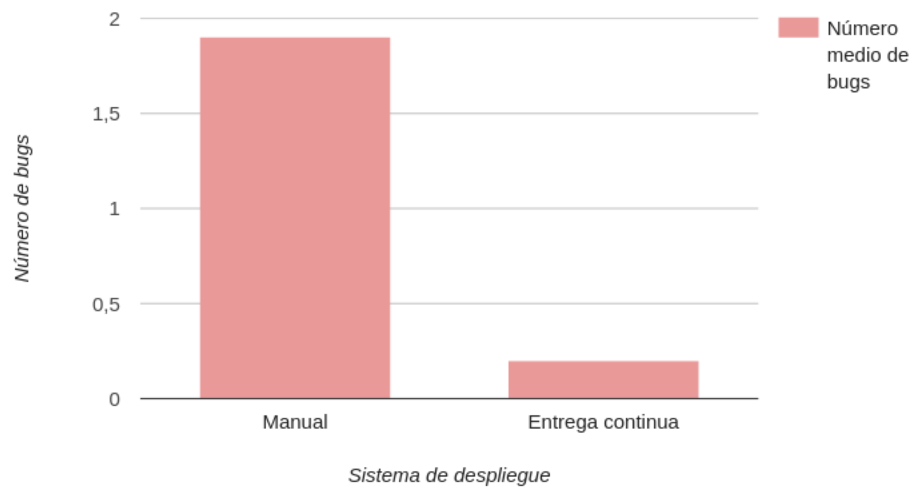

Figura 7.3: Número de *bugs* encontrados después de realizar un despliegue

Conclusión

El objetivo principal de este trabajo ha sido mejorar los tiempos de despliegue de las aplicaciones de la empresa. Originalmente la empresa sufría numerosos problemas a la

hora de desplegar sus productos debido a que el testeo se realizaba de forma manual y la falta de un sistema para sincronizar los cambios realizados entre el entorno de desarrollo y el de producción.

Para solucionar estos problemas se ha implantado un sistema de entrega continua que automatiza al máximo todo el proceso posterior a la fase de implementación. Ello ha supuesto cambios en la estructura organizativa, política de testeo y arquitectura de los servidores.

Además de las mejoras en los tiempo de despliegue, también se ha trabajado en la virtualización de toda la infraestructura necesaria para el desarrollo, basándose en contenedores Docker. Mejorando así la portabilidad de la solución, así como la gestión de recursos dentro del entorno de desarrollo.

Como se ha visto en la sección de evaluación, el tiempo de despliegue ha sido reducido de forma notable, aumentando así la productividad de la empresa. Además, también queda demostrado que la calidad del *software* se ha incrementado, mejorando la percepción que los clientes tienen de la misma.

Trabajo futuro

Durante el desarrollo de este trabajo han surgido aspectos mejorables, que quedan fuera de los objetivos iniciales.

Mencionemos algunos de los más importantes:

- Creación de microcontenedores: debido a la falta de experiencia inicial, al crear las imágenes Docker se acabaron instalando más dependencias de las estrictamente necesarias para la ejecución de las herramientas. En consecuencia, al ejecutar estas imágenes algunos de los contenedores ocupan varios GB. La creación de microcontenedores consiste en instalar solo las dependencias estrictamente requeridas, consiguiendo así tamaños mucho más contenidos. Esto favorece el rendimiento y facilita aún más la portabilidad.

- Creación de un Docker *registry* privado: durante el capítulo dedicado a Docker, hemos mencionado que existe un repositorio de imágenes público (DockerHub). Para almacenar las imágenes de la empresa sería interesante disponer de un repositorio privado, ya que DockerHub no permite hacerlo gratuitamente.

- Arquitectura maestro/esclavo en Jenkins: como hemos mencionado durante el capítulo 4 (concretamente en la sección 4.3.2) algunos de los recursos utilizados durante los *jobs* de Jenkins obligan a que estos deban ser ejecutados de forma

secuencial. Como solución a esto, Jenkins permite delegar los *jobs* a otros contenedores, que al estar aislados entre si, si que permitirían integrar varios proyectos a la vez.

- Despliegues *en caliente*: al igual que se ha virtualizado el entorno de desarrollo, se pretende también virtualizar los entornos de producción. Lo que permitiría actualizar las aplicaciones sin detener el servicio ofrecido a los clientes.

Dado que el periodo de prácticas en el que se ha realizado este trabajo de final de grado ha desembocado en un contrato laboral, se trabajará en los aspectos aquí mencionados en los meses sucesivos.

Bibliografía

[1] About docker engine. https://docs.docker.com/engine/. Accessed: 26-08-2016.

[2] Best practice - using a repository manager. https://maven.apache.org/repository-management.html. Accessed: 2016-07-26.

[3] Compare repositories. https://www.openhub.net/repositories/compare. Accessed: 21-08-2016.

[4] Cron format. http://www.nncron.ru/help/EN/working/cron-format.htm. Accessed: 24-08-2016.

[5] Docker glossary. https://docs.docker.com/engine/reference/glossary/. Accessed: 26-08-2016.

[6] Jfrog artifactory. https://www.jfrog.com/open-source/. Accessed: 2016-09-2.

[7] Manifesto for agile software development. http://agilemanifesto.org/. Accessed: 21-08-2016.

[8] Reference docker commit. https://docs.docker.com/engine/reference/commandline/commit/. Accessed: 30-08-2016.

[9] Schema sync: a mysql schema versioning and migration utility. http://mmatuson.github.io/SchemaSync/. Accessed: 29-08-2016.

[10] Vaadin testbench. https://vaadin.com/docs/-/part/testbench/testbench-overview.html. Accessed: 31-08-2016.

[11] Kellet Atkinson. Guide to continuous delivery. 2014.

[12] Orson Scott Card. In *El juego de Ender*. Tor Books, 1985.

[13] Scott Chacon and Ben Straub. Pro git. page 30. ”https://git-scm.com/book/en/v2”, apress, 2014.

[14] Cebotarean Elena. Business inteligence. Bucarest, Romania. Titu Maiorescu University.

[15] Loring Hochstein. Ansible up & running. pages 28–30. O'Reilly, May 2015.

[16] Loring Hochstein. Ansible up & running. pages 7–8. O'Reilly, May 2015.

[17] Loring Hochstein. Ansible up & running. pages 161–164. O'Reilly, May 2015.

[18] Jez Humble. The case for continuous delivery. `https://www.thoughtworks.com/insights/blog/case-continuous-delivery`, February 2014. Accessed: 02-09-2016.

[19] Jez Humble and David Farley. Continuous delivery: reliable software releases through build, test, and deployment automation. pages 115–117. Addison-Wesley, 2011.

[20] Michael James. *Scrum reference card*, 2010.

[21] Jon Jenkins. Velocity 2011: Jon jenkins, "velocity culture". `https://www.youtube.com/watch?v=dxk8b9rSKOo`, May 2011. Accessed: 02-09-2016.

[22] Karl Matthias & Sean P. Kane. Docker up & running. pages 25–26. O'Reilly, June 2015.

[23] Karl Matthias & Sean P. Kane. Docker up & running. pages 10–13. O'Reilly, June 2015.

[24] Ernest Mueller. What is devops. `https://theagileadmin.com/what-is-devops/`. Accessed: 02-09-2016.

[25] Steve Matyas Paul M. Duvall and Andrew Glover. Continuous integration: improving software quality and reducing risk. pages 29–33. Addison-Wesley, June 2007. Forewords by Martin Fowler and Paul Julius.

[26] Steve Matyas Paul M. Duvall and Andrew Glover. Continuous integration: improving software quality and reducing risk. page 26. Addison-Wesley, June 2007. Forewords by Martin Fowler and Paul Julius.

[27] Chuck Rossi. The facebook release process. `https://www.infoq.com/presentations/Facebook-Release-Process`, Febrero 2007. Accessed: 02-09-2016.

[28] Ken Schwaber and Jeff Sutherland. *La Guía Definitiva de Scrum: Las Reglas del Juego*. Scrum.Org and ScrumInc, July 2013.

[29] John Ferguson Smart. Jenkins: The definitive guide. pages 1–8, July 2011.

[30] Abdullah Uz Tansel. A survey of version control systems. https://www.researchgate.net/publication/266866968_A_Survey_of_Version_Control_Systems, 1 2015.

[31] Jason Van Zyl. Getting started with maven repository management.

Acrónimos

API *Application Programming Interface*. 16, 58

AWS *Amazon Web Services*. 32

BASH *Bourne Again Shell*. 36, 52

BD base de datos. 3, 31, 36, 37, 40, 43, 48, 50–53, 61, 62, 65, 66

BI *Business Inteligente*. 2

DC despliegue continuo. 5, 23, 42, 52–54

EC entrega continua. 5, 19–21, 23, 25, 42–44, 47, 52–55, 66

GUI Interfaz gráfica de usuario o *Graphic User Interface*. 31, 32

IC integración continua. 4, 5, 19, 21, 25–28, 31, 32, 43, 51, 52

IP *Internet Protocol address*. 46, 60, 61

JAR *Java ARchive*. 14, 16

JSON *JavaScript Object Notation*. 45

MV máquina virtual. 55, 56, 58

POM *Product Objetc Model*. 13, 14, 36

QA Control de calidad o *Quality Assurance*. 22

REST *Representational State Transfer*. 16

SCM *Source Control Management*. 34, 35

SCV sistema de control de versiones. 10, 11, 25, 33–35, 42, 51, 58, 59

SO sistema operativo. 55, 56

SQL *Structured Query Language*. 50

SSH *Secure Shell*. 33, 44, 46

URL *Uniform Resource Locator*. 17, 34, 46, 49, 52, 65

VPN Red Privada Virtual o *Virtual Private Network*. 60, 61

WAR *Web application Archive*. 14, 16, 17, 42, 47–49, 51, 52, 65

XML *eXtensible Markup Language*. 13, 45

YAML *Yet Another Markup Language/YAML Ain't Markup Languge*. 45, 47