

FASTBOOK 05

CI/CD: herramientas para la integración y el despliegue continuos

Core Desarrollo

Qualentum

Qualentum.com



05. CI/CD: herramientas para la integración y el despliegue continuos

En este fastbook desgranaremos el concepto de CI/CD que corresponde por sus siglas en inglés a *continuous integration/continuous deployment*. Aprenderemos qué utilidades tiene en el mundo IT actual, ya que nos encontramos con una nueva filosofía que apuesta por los procesos de despliegue continuos en entornos tanto productivos como previos (preproducción, desarrollo, pruebas...), así como por la integración de los diferentes procesos que debe pasar un proyecto para mejorar su comunicación con otros ya existentes. Pero antes de abordar este método y sus herramientas de trabajo (Jenkins, Maven o XRay) resulta imprescindible entender en qué consiste la cultura DevOps y por qué las empresas adoptan este marco de trabajo.

Autor: Juan Arias, Arturo Mora y Jaime Morales

- La cultura DevOps
- CI/CD (*continuous integration/continuous deployment*)
- Pipelines de CICD
- Jenkins, herramienta clave para trabajar con CI/CD
- Otras herramientas para trabajar con CI/CD: Maven y XRay
- Conclusión

La cultura DevOps



DevOps (que nace de los términos *development* y *operations*, desarrollo y operaciones, respectivamente) es una revolución en la manera de trabajar de los equipos y las organizaciones. No debe entenderse solo en el **contexto técnico** sino también en el cultural, ya que su adopción promueve una corriente de cambio en la que prima la colaboración desde la gestión hasta el **desarrollo de software y las operaciones**.

¿Pero qué ha impulsado a las empresas hacia la adopción de DevOps?

La **respuesta** es muy sencilla: los largos plazos de entrega del software hasta su paso a producción, derivados de los sistemas de trabajo tradicionales que:

- Impiden a las empresas **brindar servicios** de vanguardia.
- Impiden a los **equipos TI** (tecnologías de la información) el trabajar de forma ágil.
- Desgastan la **experiencia** del cliente.

Para mantener el ritmo de las demandas del mercado, los equipos de TI deben construir, implementar, probar y lanzar software en ciclos cada vez más rápidos.

Debido a que DevOps mejora la forma en que una empresa entrega valor a sus clientes, proveedores y socios, podemos afirmar que es un **proceso esencial de negocio** y no solo una capacidad de TI.

DevOps no puede entenderse únicamente como un conjunto de tecnologías y mejores prácticas, sino como una **revolución cultural**. Es una corriente de cambio dentro de las organizaciones, en la que priman **tres factores**:

- 1 La **colaboración e interacción abierta** entre los equipos de operaciones y de desarrollo.
- 2 El **derribar los silos**, barreras artificiales creadas en las organizaciones que frenan la colaboración natural.
- 3 Los **ciclos cortos de diseño, implementación y despliegue**, aquellos que permiten anticiparse a los problemas y adaptarse al cambio.

Por lo general, solemos utilizar el término **cambio cultural** para implementar procesos de mejora continua, para emplear metodologías ágiles y para adoptar el enfoque DevOps. Esto obedece a la imposibilidad de adoptar estas transformaciones de forma superficial: estamos hablando de un cambio organizativo muy beneficioso a la vez que radical.

Si bien son las grandes organizaciones las que podrán beneficiarse de estos cambios de forma más evidente en términos cuantitativos, son las **nuevas organizaciones** las que tienen la posibilidad de abrazar la cultura DevOps y de la colaboración de una forma natural.

En la mayoría de las startups, la cultura de la colaboración, la visibilidad en contraposición con el control o la **automatización en oposición al trabajo manual** surgen de forma natural por la necesidad de optimizar los recursos.

DevOps es un término relativamente nuevo para describir lo que también ha sido llamado como 'administración de sistemas ágiles' y que se centra en el trabajo y la colaboración de los equipos de desarrollo y operaciones.

Debido a las exigencias del mercado actual y la creciente competencia por crear productos y servicios atractivos y útiles para las personas, DevOps se ha convertido en un enfoque cada vez más extendido y popular para la entrega de software. Los equipos de desarrollo y operaciones utilizan esta metodología para **construir, probar, implementar y monitorizar** las aplicaciones debido a que les permite actuar con velocidad, mantener altos índices de calidad y controlar los cambios con suma rapidez.

DevOps es esencial para cualquier empresa que aspire a ser ágil y pretenda ser capaz de responder rápidamente a las demandas del mercado.

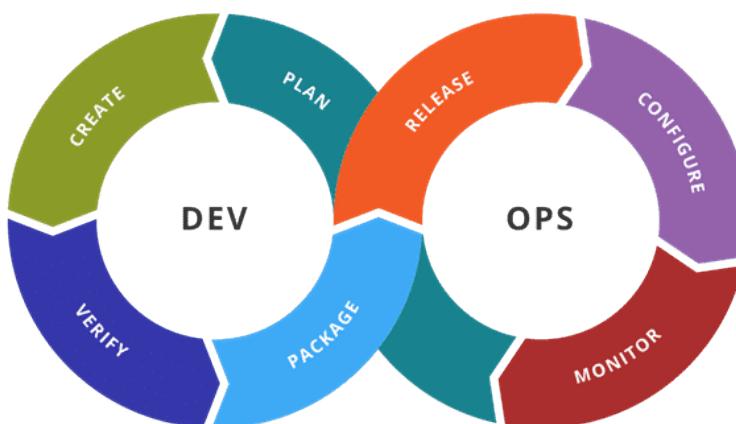
Por lo tanto, DevOps es un **enfoque hacia la entrega de software**, uno que promueve una colaboración más estrecha entre las líneas de negocio, desarrollo y operaciones al tiempo que elimina las barreras entre las partes interesadas y los clientes.

DevOps como herramienta de cambio

En general, realizar un cambio de la manera tradicional siempre es difícil y requiere de una inversión en tiempo, recursos humanos y económicos. Así que, cada vez que una organización adopta cualquier nueva tecnología, metodología o enfoque, la adopción de esta tiene que ser impulsada por una **necesidad de negocio**.

En un entorno DevOps, los equipos son responsables de ofrecer nuevas características, pero también estabilidad, escalabilidad, fiabilidad y otro sinfín de características comunes a un software de calidad. Es por ello por lo que las responsabilidades se equilibrان de manera más equitativa para garantizar que ambos equipos, desarrollo y operaciones, tengan visibilidad del rendimiento de la aplicación a través de todas las etapas del **ciclo de vida**.

Según la Wikipedia, el ciclo de vida está compuesto por una serie de fases, claramente definidas y diferenciadas, que son utilizados por ingenieros y desarrolladores de sistemas de información para **planificar, diseñar, construir, probar, entregar...**



Fuente: [Wikipedia](#).

El **objetivo del ciclo de vida del software** no es otro que producir sistemas de alta calidad que cumplan o excedan las expectativas del cliente y siempre dentro de las restricciones de tiempo y coste.

Para ello, las empresas que adoptan la cultura DevOps suelen **implementar una metodología Agile** en el trabajo de sus proyectos, ya que, como bien sabemos, es un método de trabajo esencialmente adaptativo que se centra en aplicar rápidamente los cambios de rumbo en un proyecto, modificando estimaciones, requisitos, alcance... y siempre con el objetivo de garantizar un producto final satisfactorio para negocio y/o el cliente.

Ahora que ya entendemos el marco de trabajo DevOps, podemos abordar dos procesos que marcan el core de este fastbook: la integración continua y el despliegue continuo. ¡Vamos a allá!

CI/CD (continuous integration/ continuous deployment)



Supongamos que ya hemos construido el software que nos han encargado, que hemos etiquetado el resultado y subido a nuestro almacén de artefactos. ¿Qué fase del proceso tocaría? **Desplegarlo**.

Todo el proceso previo al despliegue responde a un trabajo prácticamente manual, ¿cierto?; pero todos sabemos que es muy complicado **evitar un error humano en procesos tan complejos como el despliegue**. Por lo tanto, la automatización en esta fase del proyecto resulta fundamental.

De hecho, existen soluciones que se encargan de **automatizar** una serie de pasos por nosotros, son capaces de enviar feedback de cada paso que se da y tienen la capacidad de tomar decisiones en base a datos.

- Saber cuándo deben empezar a construir un ejecutable.
- Saber si un ejecutable cumple los mínimos exigibles para promocionar a un entorno diferente (pruebas, QA, producción...).
- Saber enviar un correo al responsable de una nueva característica porque rompe el test de integración, así como detener el proceso de *release*.
- Saber si debe hacer *rollback* de una *release* porque está fallando algo.

Pues bien, el CI/CD se apoya en un método para distribuir aplicaciones a los clientes de manera automatizada pasando por todas las etapas de desarrollo. Estos conceptos se aplican bajo el objetivo de conseguir la integración, la distribución y el despliegue continuo.

En concreto, **el proceso de integración y distribución continuas** incorpora la automatización y la supervisión permanentes en todo el ciclo de vida de las aplicaciones, desde las etapas de integración y prueba hasta las de distribución e implementación.



Para que lo entiendas mejor, te recomendamos que eches un vistazo a la conferencia de Jez Humble sobre el despliegue continuo: [Jez Humble - Continuous Delivery](#).

Jez Humble es uno de los padres de la entrega continua, quien habla de lo que implica adoptarla, de los beneficios que ofrece y las desventajas de no hacerlo.

Son muchas las **ventajas** de tener sistemas de *integración continua*, llamados así porque su labor es la de escuchar eventos en un VCS y, en caso de encontrar cambios en una o más ramas, empezar a ejecutar una serie de pasos:

- Test unitarios y de integración.
- Análisis de seguridad y calidad de código.
- Empaque.
- Publicación de resultado en un almacén.

En el caso de que, además, la herramienta sea capaz de **desplegar automáticamente** a entornos productivos y monitorizar el resultado para dar marcha atrás en caso de errores, estaríamos hablando de sistemas de despliegue continuo.

Antipatrones en el despliegue continuo

Antes de definir los **conceptos de integración y entrega continuas** (EIEC, denominados también como CICD, *continuous integration/continuous delivery*), vamos a repasar algunos patrones de despliegue opuestos a la metodología de CICD. Estos ejemplos servirán para **explicar las ventajas de CICD y los fallos** que pretende evitar.

1

Despliegue manual del software

Cualquier aplicación moderna tiende a ser compleja y, aun así, muchas organizaciones se empeñan en desplegar software manualmente. Incluso si pasos concretos del despliegue se llevan a cabo con scripts, se considera que un despliegue es manual si los pasos necesarios para llevarlo a cabo son atómicos, independientes y ejecutados por personas o grupos diferentes.

Aun cuando **se elimina el error humano** mediante scripts de automatización, el hecho de separar los pasos puede introducir diferencias en el orden que se ejecutan o en los parámetros de entrada. Esta aleatoriedad da lugar a un despliegue que no es determinista. Este antipatrón se da cuando...

- La **documentación** es extensa y detallada, con descripciones minuciosas de los pasos a seguir y los errores que pueden ocurrir.
- Los **grupos** de operaciones confían en pruebas manuales para confirmar que todo va bien.
- A pesar de la documentación, es necesario involucrar a los **desarrolladores** durante el despliegue o cambiar el flujo de despliegue al vuelo durante un pase a producción.
- Hay diferencias entre **servidores** que cumplen el mismo rol, por ejemplo, en la configuración del sistema de ficheros de nodos de un clúster de alta disponibilidad.

La alternativa a este antipatrón es tender a los despliegues completamente automatizados. La intervención humana se debe limitar a **tres operaciones: seleccionar el entorno** (desarrollo, prueba o producción), **seleccionar la versión** (el ID del commit o cualquier otra manera de identificar el código) y **presionar el botón de OK**. El resto de los pasos deben ser automáticos porque...

Los errores ocurren. La pregunta no es si ocurren sino cómo de importante será su impacto. Cuando más predecibles sean los despliegues, menos probabilidades habrá de que ocurran y menos tiempo se tardará en depurarlos.

Los despliegues manuales deben estar extensivamente documentados. Esta documentación es difícil de mantener y queda desfasada rápidamente. Cuando el código de un script (y el código de cualquier aplicación, en general) está bien escrito, sirve de documentación por sí mismo. Si hay que actualizar el script, la documentación se actualiza al mismo tiempo.

Además, los scripts son más fáciles de mantener por cualquier individuo del grupo, ya que la documentación suele estar sesgada por los conocimientos de quien la escribió: un experto puede obviar pasos que le han parecido evidentes, pero un script debe incluir todos los pasos de manera explícita. De la misma manera, la ejecución manual depende del nivel de experiencia del operador, por muy repetitivo que sea. Una tarea repetitiva y aburrida es la receta perfecta para un error humano por culpa de un despiste.

Escribir pruebas para un proceso automatizado es relativamente barato. De la misma manera, es más fácil de auditar. Probar un proceso manual implica ejecutarlo a mano, lo que implica tiempo de un personal cualificado que podría estar dedicado a tareas que aportan más valor.

El proceso de despliegue debe ser el mismo en todos los entornos. Si se cumple esta premisa, el proceso se habrá probado muchas veces antes de ejecutarlo en producción, lo que lo hace mucho más fiable.

2

Despliegue de producción una vez completado el desarrollo

A menudo, los equipos de operaciones no se plantean el despliegue de entornos de producción hasta que el equipo de desarrollo complete su parte. Hasta este punto, las pruebas se han llevado a cabo en entornos de desarrollo. El equipo de operaciones no toma contacto con la aplicación hasta que llega el momento de desplegar, bien porque nadie se ha molestado en implicarlos, o bien porque implementar un entorno similar al de producción es demasiado caro como para probarlo.

Esto fuerza a los desarrolladores a preparar los scripts de instalación y los binarios sin haberlos probado de manera fiable y a delegar esta tarea en unos operadores que tampoco han tenido oportunidad de familiarizarse con ellos. Algunos **problemas** que pueden ocurrir si se sigue este patrón son los siguientes:

- Si el equipo se forma expresamente para desplegar la aplicación, **la interacción entre desarrolladores y operadores** es menor que si ambos equipos han trabajado conjuntamente desde un primer momento. Una situación ideal sería que se formaran equipos multidisciplinares.

- A veces, **el diseño de la aplicación se construye sobre suposiciones válidas en entornos locales**, diferentes a un entorno de producción. Por ejemplo, el entorno de trabajo de un desarrollador puede simular un clúster con un único nodo, lo que puede encubrir problemas de comunicación entre múltiples nodos que no son visibles hasta que se instala la aplicación en un clúster real.
- Cuanto más se tarda en preparar ese primer entorno de producción, más **suposiciones incorrectas** pueden tomarse en el diseño y el desarrollo.

La solución a este antipatrón es incorporar las tareas de despliegue de entornos de prueba similares a producción lo más pronto posible en el ciclo de vida de la aplicación.

Si el día de lanzamiento ya se han ejecutado múltiples despliegues, la fiabilidad y la confianza en el sistema será mucho mayor.

3

Gestión de configuración manual de entornos de producción

Incluso cuando el despliegue de los diferentes entornos es **automático**, hay casos en los que las opciones de configuración se gestionan manualmente. Por ejemplo, si es necesario modificar los detalles de conexión a una base de datos, como nombre de host y credenciales, un operador se conectará a los servidores de la aplicación, editarán un fichero de configuración o una variable de entorno y, probablemente, reiniciará los procesos para que las nuevas opciones tengan efecto.

- En **el mejor de los casos**, la nueva configuración será permanente y el administrador registrará sus cambios en la documentación.
 - En **el peor**, no quedará registro alguno del cambio y en el siguiente pase a producción, la configuración antigua volverá a tener efecto, produciendo un fallo inesperado.
-

Esta gestión manual introduce riesgo en el despliegue y reduce su fiabilidad.

Puede producir fallos repentinos en despliegues que funcionan habitualmente y cuyos scripts de automatización, en principio, no han sufrido cambios. Además, no es posible recuperar una configuración anterior para poder hacer análisis forense (o solo es posible mediante una recuperación de copias de seguridad).

La manera de **evitar caer en este patrón** es mantener todos los elementos de los entornos de prueba y producción en el sistema de control de versiones. Esto significa que cualquier elemento de la infraestructura y de la configuración debe estar versionado y que, si fuera necesario, debería ser posible reproducir cualquier entorno, incluso producción, de manera automática.

Integración y entrega continua

Los antipatrones expuestos anteriormente se pueden evitar si se persiguen **dos objetivos**: desplegar a menudo y de manera automática.

"La integración continua es una metodología de desarrollo de software en la que los miembros de un equipo integran su trabajo de manera frecuente, incluso varias veces al día, por lo que el trabajo de todo el equipo se integra múltiples veces al día. Cada integración se verifica con una construcción y pruebas automáticas para detectar errores tan pronto como sea posible."

- Martin Fowler

La práctica de la integración continua no es nueva y es, sin duda, muy anterior a la revolución DevOps. El concepto de *nightly build*, en el que cada noche se construye el código completo incluyendo todos los cambios del día fue un primer intento. La **velocidad de retorno** de la información (es decir, si la construcción fue satisfactoria o si ha habido errores y cuáles han sido) era muy lenta. Además, esta técnica es difícil de adaptar en equipos con individuos en diferentes continentes. Actualmente, se entiende la integración continua como una metodología más agresiva. Tal como indica la cita de Martin Fowler, la construcción y las pruebas se ejecutan en cada cambio.

"La entrega continua es la capacidad de llevar los cambios de un sistema, sean los que sean (nuevas características, configuración, arreglos y experimentos) hasta el entorno de producción, de forma que aporten valor a los usuarios, de una manera segura y rápida. El objetivo es que los despliegues, ya sean grandes sistemas distribuidos, aplicaciones o sistemas embebidos, se puedan llevar a cabo de una manera predecible y rutinaria siempre que haga falta."

- Jez Humble

Ambos conceptos, integración y entrega, están **típicamente asociados** ya que la mejor forma de reducir el riesgo de un despliegue es haber comprobado exhaustivamente todas y cada una de las integraciones. Además, si las integraciones se hacen a menudo y los cambios son pequeños, cada entrega también será pequeña y, por tanto, más rápida.

Despliegue continuo

Utilizamos este término cuando hablamos de la facultad de poder desplegar en producción múltiples veces al día. Los **conceptos de entrega y despliegue** no tienen por qué estar siempre presentes de manera conjunta en todos los pipelines de CICD. La entrega llega hasta el momento en que **la aplicación está lista para ser desplegada**, pero puede haber razones para no desplegar tras cada integración; por ejemplo:

- Una nueva funcionalidad debe estar disponible a partir de una fecha concreta y no antes.

- Cada despliegue implica una parada de servicio y esto solo se permite cuando el volumen de usuarios es muy bajo o nulo.
- En un **software discreto** (es decir, una aplicación de escritorio o de móvil, por ejemplo), las funcionalidades nuevas deben ofrecerse en una versión nueva exclusivamente.

Madurez de la organización

No todas las organizaciones están preparadas para aplicar CICD en sus proyectos de manera inmediata. El modelo de madurez definido por Jez Humble y David Farley en su magnífico libro *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* (Addison-Wesley Professional, 2010) ayuda a **identificar en qué estado se encuentra** una organización en términos de los procesos y prácticas. Además, define el progreso que debe seguir para mejorar, que debe ser el objetivo final de aplicar CICD. Los **resultados deseables** son:

- 1 **Reducir el tiempo** entre despliegues (también denominado *cycle time*) para aumentar la velocidad con la que se aporta valor a los clientes.
- 2 **Reducir los incidentes** para mejorar la eficiencia y dedicar menos tiempo al soporte.
- 3 Hacer que el ciclo de vida del software sea **más predecible**.
- 4 Adaptarse a las **regulaciones** de manera natural.
- 5 Determinar y gestionar los **riesgos** de manera efectiva.
- 6 **Reducir el coste** gracias a una correcta gestión del riesgo.

La tabla 1 facilita la **identificación del nivel de madurez** de una organización y sirve de guía para alcanzar estos objetivos. Una organización puede estar en niveles diferentes en grupos y proyectos diferentes, por lo que deberá aplicar las mejoras de manera particular en cada uno.

Debería empezar por las áreas en las que sufra más: por ejemplo, si la construcción automática es fiable, pero no se detectan suficientes errores en las primeras fases, el primer objetivo debería ser **automatizar** también las pruebas. Si, además, desplegar un entorno de desarrollador es una tarea manual, ineficiente y llena de errores, el siguiente objetivo debería ser automatizar también ese paso hasta que sea **fiable y repetible sin esfuerzo**.

Nivel de madurez de una organización:

		Construcción y CI	Entornos y despliegues	Liberación de versiones	Pruebas	Gestión de DB
3	Optimización	Los equipos se reúnen habitualmente para resolver problemas en la integración y los resuelven con automatización, feedback y visibilidad.	Todos los entornos se provisionan automáticamente y se administran de forma eficiente.	Los equipos de operaciones colaboran de manera habitual para reducir el riesgo y reducir el tiempo de entrega.	Las marchas atrás en producción son poco habituales. Las pruebas detectan los errores rápido y los equipos los arreglan al instante.	El rendimiento de las bases de datos se reporta de manera consistente.

2	Administración cuantitativa	Las métricas de la construcción se recopilan, son visibles y se actúa para corregirlas. Una construcción errónea se arregla rápidamente.	Los despliegues están correctamente orquestados. El despliegue y la marcha atrás se prueban de manera consistente.	La salud de la aplicación y el tiempo de despliegue se monitorizan.	Las métricas de calidad se reportan. Los requisitos no funcionales se definen y miden.	Las actualizaciones y marchas atrás de las bases de datos se prueban en cada despliegue.
1	Consistente	Cada cambio arranca una construcción y una batería de pruebas automáticas. Se administran correctamente las dependencias.	Los despliegues se consiguen de manera automática, con un único proceso para todos los entornos.	Se aplica gestión del cambio y proceso de aprobación.	Las pruebas unitarias y de aceptación son automáticas y se implica al equipo de probadores y QA.	Los cambios en la base de datos se aplican como parte del despliegue.
0	Repetible	La construcción y las pruebas se ejecutan de manera regular. Las construcciones se pueden repetir a partir de un número de versión.	Algunos entornos se pueden desplegar de manera automática. La configuración está versionada.	Los despliegues a producción son pocos, pero fiables.	Las pruebas automáticas se escriben como parte del desarrollo habitual.	Los cambios en la base de datos se ejecutan con un script automático que está versionado.
-1	Regresión	La construcción es manual. No hay gestión de los artefactos, paquetes e informes de pruebas.	El despliegue es manual. Hay paquetes específicos en cada entorno.	Los pasos a producción son escasos y poco fiables.	La fase de pruebas es manual e independiente de la fase de desarrollo.	Las migraciones de las bases de datos no están versionadas y se ejecutan manualmente.

Pipelines de CICD



El objetivo principal de **implementar CICD en un proyecto de software** es ofrecer software de calidad a los usuarios partiendo de los cambios de un desarrollador de una manera rápida y eficaz. El código escrito por el desarrollador atraviesa múltiples fases: compilación, empaquetado, pruebas de múltiples tipos, despliegues en múltiples entornos y, finalmente, despliegue en producción.

No es habitual encontrar el concepto de pipeline traducido al español, incluso en documentación en español. Por tanto, esta asignatura usará exclusivamente el término pipeline.

Un pipeline es un proceso que permite **encadenar estas fases** cumpliendo dos características fundamentales:

1

El código y los artefactos generados en cada fase se mueven de una fase a otra de manera **automática**.

2

El flujo se repite continuamente, en base a eventos o de forma temporizada.

Fases de un pipeline

Aunque cada proyecto tendrá sus **necesidades específicas** en cuanto a métodos de construcción, tipos de artefactos, batería de pruebas, casuísticas de despliegue..., es posible encontrar una serie de patrones comunes que, de uno u otro modo, se van a repetir en la **gran mayoría de los casos**:

- Validación y construcción.
- Control de calidad.
- Publicación de artefactos.
- Despliegue.

1

Validación y construcción

Sin lugar a duda es **la fase más importante** de cualquier pipeline. El sistema de CICD iniciará el proceso cuando detecte algún evento o cambio significativo en el repositorio:

- Modificaciones en alguna rama.
- Creación de *pull request*.
- Tarea planificada (por ejemplo, cada noche).

El objetivo de esta fase del pipeline es el de generar un **artefacto**, es decir, un paquete o empaquetado o ejecutable de cualquier clase que sirva para poder lanzar nuestra aplicación en alguna plataforma de ejecución: un jar/war, en el caso de una aplicación Java, o un Wheel, en el caso de Python, son ejemplos de artefactos.

Pero para llegar a tener un artefacto que contenga los cambios que han originado esta ejecución del pipeline, lo habitual es que se ejecuten **una serie de pasos**:

- El análisis estático de código.
- La ejecución de los test unitarios.
- La descarga de paquetes.
- El lanzamiento de los scripts.
- Los comandos de compilación.

Para poder ejecutar esto en los **servidores de integración continua**, vamos a tener que ejecutar de forma remota comandos y operaciones que, normalmente, se quedan en el ámbito de los desarrolladores: lo que en cualquier IDE puede hacerse con un botón o incluso de forma automática, en CICD debemos poder expresarlo por medio de scripts que se ejecuten de manera declarativa e independiente.

Además, debemos tener en mente que se debe fallar lo más rápido posible y dando todo el feedback al usuario que sea factible conseguir. Esto es porque cuanto antes se vea un error, antes puede el equipo arreglarlo y tratar de pasar a la siguiente fase del pipeline.

Por último, hay que comentar que es **habitual añadir/inyectar flags o modificadores a esta y otras fases de CICD**, donde tengamos información sobre el tipo de *build*, y usarlo para tener comportamientos diferentes según qué tipo de construcción estemos haciendo:

- Cuando se construye una *pull request*, normalmente, no interesa generar un artefacto.
- En los eventos de construcción de una rama de desarrollo no suele interesar gastar recursos en ejecutar costosos tests de integración.
- Cuando se construye una *nightly* nos puede interesar añadir un sufijo al artefacto generado.
- Cuando se construye la rama *master/main* es posible que queramos taguear la construcción si va bien y que sea **la única manera** de que un artefacto se promocione al entorno real de producción.

2

Publicación de artefactos

Una vez que tenemos generada una **versión de nuestro software** y hemos usado las herramientas de construcción apropiadas para obtener un ejecutable, surgen más preguntas:

- ¿Dónde podemos almacenar sucesivas versiones del mismo software?
- Si mi software tiene dependencias de otras piezas, ¿dónde se almacenan?
- ¿Puedo llevar el ejecutable a su destino? ¿O necesito una pieza intermedia?

Las herramientas que cumplen estas características son los almacenes de artefactos: existen varios tipos de almacenes según el tipo de paquete en el que se especializan: Docker Registry, Maven Central, PyPi... También existen herramientas que los aúnan todos tras una interfaz común, como [Artifactory](#) o [Nexus](#).

El hecho de **alojar una solución propia** en lugar de depender de un externo es algo que dependerá totalmente de nuestro proyecto y de nuestra organización, pero que sobre todo vendrá determinado por los requisitos de seguridad y conectividad que tengamos:

Seguridad

- ¿Se puede establecer una comunicación segura con el servicio?
- ¿Qué garantías tenemos de que los artefactos están protegidos de modificaciones en este alojamiento?
- ¿Podemos perfilar y auditar los accesos de alguna manera?

Conectividad

- ¿Es posible acceder desde la infraestructura de CICD? ¿Incluso desde un contenedor?
- ¿Se necesitará acceso vía proxy? ¿Podemos configurar nuestro CICD para eso?
- ¿Pueden llegar los desarrolladores desde sus estaciones locales para descarga de dependencias durante el desarrollo?

3

Control de calidad

Aunque el control de calidad tiene muchas formas, recordemos de lo que sabemos de testing que ciertas pruebas son mucho más costosas que otras en términos de tiempo de ejecución, pero también en recursos.

Sin olvidar nunca que el objetivo primordial de un pipeline es ofrecer feedback lo más rápido posible, habrá test o pruebas que tengan que esperar a que el artefacto haya sido generado o incluso publicado. Existen muchos casos, que dependen siempre de las aplicaciones o métodos que existan en la organización, pero podemos dar algunos ejemplos:

Antes de la compilación

- Un análisis estático del código en busca de errores de formateo y desviaciones de las convenciones de código es relativamente rápido y se puede ejecutar antes de la compilación.
- Los test unitarios deberían ser rápidos por definición y no requerir recursos externos para su ejecución.

Antes de la publicación

- Un análisis con herramientas tipo [SonarQube](#), donde se aplican ciertos heurísticos que dependen del lenguaje empleado y gracias a los cuales se obtienen análisis increíblemente detallados de deuda técnica, empleo de código deprecado, duplicaciones de código, etc. Estos análisis pueden tomar su tiempo, pero son, sin duda, necesarios antes de que el artefacto llegue al almacén correspondiente.
- Análisis de vulnerabilidades de seguridad como [Docker Scan](#), por ejemplo, en donde se analiza una imagen o un empaquetado que contenga librerías de terceros en busca de vulnerabilidades conocidas como, por ejemplo, aquellas reportadas por [OWASP](#) Foundation (Open Source Foundation for Application Security).
- Test de integración que, cuando el paquete cumple con los mínimos exigibles para su construcción, ya tiene sentido ejecutar.

Después de la publicación

- Cuando el artefacto ya está publicado se puede simular su implantación usando entornos **preproductivos** en los que dispongamos de forma de verificar que hace lo que tiene que hacer, con unas condiciones lo más parecidas a las reales.

4

Despliegue

Una vez llegados a esta fase del pipeline, ya deberíamos tener bastantes garantías, **mayores cuanta mayor sea la confianza en nuestras pruebas automáticas**, de que el cambio que queríamos desplegar está listo para hacerlo.

Aquí, una vez más, dependemos totalmente del tipo de entorno en el que nos encontramos y del tipo de cambio que queramos desplegar. Aunque no nos detendremos en este tema en profundizar en las maneras que existen de desplegar un cambio, pero podemos adelantar que existen dos tipos de despliegues, dependiendo de si existe o no pérdida de servicio (**downtime**):

SIN DOWNTIME

CON DOWNTIME

Son cambios que no implican pérdida de servicio de la aplicación y se pueden desplegar sin que los usuarios 'noten nada'.

SIN DOWNTIME

CON DOWNTIME

Si para poder hacer el despliegue hay que reiniciar el servicio, lo que conlleva que algún usuario puede tener afectación o perder datos, debemos tener mucho más cuidado.

En general, la mayoría de los cambios suelen **tener asociado algún tiempo de pérdida de servicio**. Incluso aunque la tecnología tenga manera de aplicar cambios en caliente debemos tener en cuenta qué se cambia, porque podría resultar en usuarios perdiendo transacciones o datos:

Ejemplo

Un cambio aparentemente sencillo e inocuo podría ser **una modificación de un campo de datos** de una petición GET de una API que solo consuma nuestra aplicación. Si desplegamos todos los cambios a la vez, no debería haber problema. Pero si por lo que sea no sincronizamos el cambio del frontend con el del backend (algo habitual cuando encuentran desacoplados), podría ocurrir que un usuario en medio de una operación crítica tenga errores de cliente que le impidan culminar con éxito la transacción, al no entender la nueva respuesta de una API que, quizá, ni siquiera sabía que existía.

Jenkins, herramienta clave para trabajar con CI/CD



Jenkins es una herramienta de automatización que nos ayudará a aprender los básicos de los sistemas de CICD.

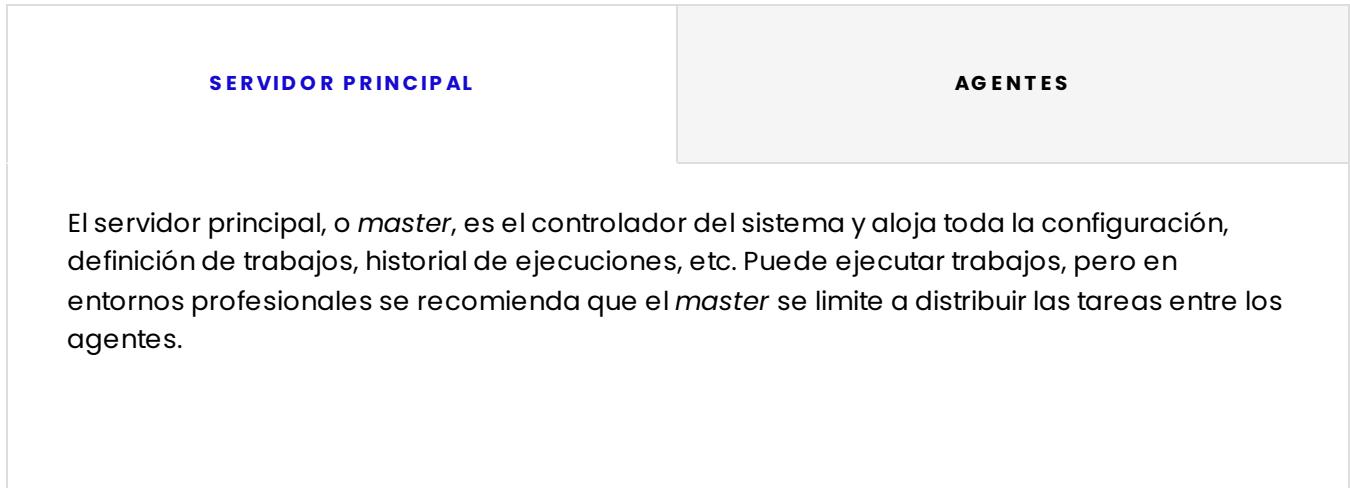
Es una herramienta de código abierto disponible para múltiples arquitecturas que puede manejar cualquier tipo de construcción, pruebas y despliegues gracias a su arquitectura de plugins. Evolucionó a partir de **Hudson, una herramienta de Sun Microsystems** que aún continúa en desarrollo, pero que pasó a ser de código propietario.

La idea central de Jenkins es la de crear trabajos (a veces denominados *jobs*) que realicen ciertas operaciones como la compilación, el empaquetado, las pruebas, el despliegue, etc. Se puede diseñar un trabajo que lleve a cabo todas las tareas del pipeline o varios trabajos simples con tareas pequeñas y encadenarlos con un trabajo principal. Esta ejecución de trabajos por parte de otros trabajos ha sido uno de los paradigmas fundamentales de Jenkins.

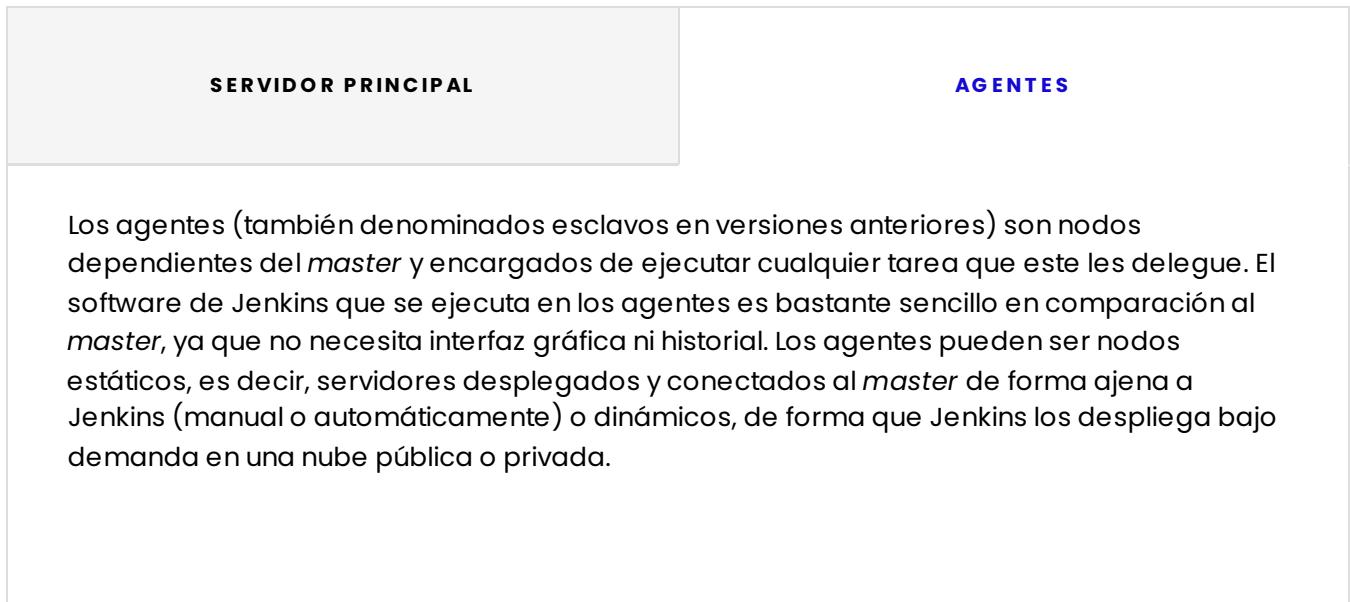
El repositorio de plugins es muy amplio y, de hecho, algunas de las principales características se ofrecen de esta manera. Es más, el primer paso del asistente de configuración de Jenkins es la instalación de los plugins recomendados.

Arquitectura

Jenkins se basa en una arquitectura de **ejecución distribuida**, con un servidor principal y varios agentes:



El servidor principal, o *master*, es el controlador del sistema y aloja toda la configuración, definición de trabajos, historial de ejecuciones, etc. Puede ejecutar trabajos, pero en entornos profesionales se recomienda que el *master* se limite a distribuir las tareas entre los agentes.



Los agentes (también denominados esclavos en versiones anteriores) son nodos dependientes del *master* y encargados de ejecutar cualquier tarea que este les delegue. El software de Jenkins que se ejecuta en los agentes es bastante sencillo en comparación al *master*, ya que no necesita interfaz gráfica ni historial. Los agentes pueden ser nodos estáticos, es decir, servidores desplegados y conectados al *master* de forma ajena a Jenkins (manual o automáticamente) o dinámicos, de forma que Jenkins los despliega bajo demanda en una nube pública o privada.

Estos agentes se encargan, por tanto, de **ejecutar cualquier tarea del pipeline**, por lo que deben disponer de las herramientas al respecto. Por ejemplo, para descargar un repositorio de Git, compilar un proyecto en Java, una interfaz web en NodeJS, ejecutar las pruebas de ambos proyectos y generar una imagen de Docker con ambos, estos nodos deben tener instalado:

- El cliente de Git.
 - El Java SDK y el constructor (probablemente, maven, ant o gradle).
 - NodeJS y npm.
 - El cliente de Docker.
-

**Jenkins se puede encargar de instalar este software si
se configura previamente en la Global Tool
Configuration.**

De lo contrario, el administrador deberá encargarse de instalar los paquetes en los agentes si estos se despliegan manualmente o bien preparar las imágenes si Jenkins los provisiona bajo demanda. Esta opción puede ser la más cómoda si es necesario disponer de agentes con software diferente para proyectos con necesidades diferentes.

En cualquier caso, hay que **tener en cuenta si hacen falta agentes de diferentes sistemas operativos**. Jenkins es **multiplataforma**, por lo que incluso el sistema operativo puede ser diferente de un agente a otro: se pueden compilar proyectos para Linux, Windows y MacOS desde un único *master*.

Cada agente tiene uno o varios ejecutores. Estos ejecutores son procesos iniciados por el agente para un trabajo de Jenkins concreto. Así, es posible ejecutar varios trabajos en paralelo en cada agente.

Estructura de un trabajo

El elemento central de Jenkins es el **trabajo**. Un trabajo está compuesto, a grandes rasgos, de:

- Uno o varios *triggers* (o disparadores) que inician la ejecución.
- Una lista de agentes en los que se permite ejecutar el trabajo.
- Parámetros de entrada, si aplican.
- Tareas que se ejecutarán (que pueden arrancar otros trabajos).
- Tareas de finalización (o *postprocessing*).
- Artefactos archivados.

Los *triggers* arrancan la ejecución de un trabajo. Pueden ser **periódicos** (al estilo de las tareas de cron), **manuales** (es decir, iniciados por un usuario), desde otro trabajo o a partir de un *webhook*.

Un *webhook* no es más que una **llamada a una API REST**. Jenkins acepta llamadas de [GitHub](#) en la ruta. Evaluará el contenido de la llamada para ejecutar un trabajo u otro.

Los triggers periódicos pueden estar condicionados a que existan **cambios en un repositorio**. Por ejemplo, Jenkins puede comprobar periódicamente un repositorio e iniciar el trabajo sólo si han aparecido commits nuevos en una rama concreta.

Este modelo síncrono añade demasiada carga tanto a Jenkins como al sistema de control de cambios, por lo que se pueden aprovechar los *webhooks* para arrancar los trabajos asíncronamente. Por ejemplo, GitHub puede lanzar un *webhook* cuando se abre una *pull request* o cuando aparecen *commits* nuevos en una rama; Jenkins puede arrancar el trabajo al recibir el *webhook*, evitando así las comprobaciones periódicas.

El modelo síncrono se conoce como *polling*; aunque es sencillo de implementar, implica múltiples llamadas de manera continuada.

Los agentes en los que se puede ejecutar el trabajo se indican con etiquetas, por lo que no es necesario especificar nombres de servidores concretos. Por ejemplo, un proyecto de Java basado en *maven* que genere paquetes de instalación para Linux y Windows puede usar un trabajo con las etiquetas ['linux', 'maven'] y otro trabajo con ['windows', 'maven']. El administrador de Jenkins debe haber etiquetado previamente los agentes en función de las funcionalidades que ofrecen. Si el *master* no encuentra un agente que cumpla los requisitos, el trabajo fallará.

Los parámetros de entrada **funcionan como en cualquier lenguaje de programación**. Estos parámetros pueden tener valores por defecto, se pueden recibir desde otros trabajos o pueden ser especificados por el usuario en un arranque manual. Hay que tener en cuenta que Jenkins es totalmente agnóstico en cuanto al contenido de los trabajos, por lo que podría usarse para tareas ajenas al desarrollo de software, como tareas de copia de seguridad, de facturación, ETLs en bases de datos, etc. En este tipo de trabajos, **el usuario podría especificar unos parámetros concretos** para cada ejecución.

¿Sabías que? Un ETL, de extract/transform/load, es una tarea que lee, transforma y escribe datos entre diversas fuentes de datos o entre diferentes tablas de una misma base de datos.

Tanto las tareas principales como las de finalización soportan flujos condicionales (no ejecutar esta tarea si ocurre 'X'), errores (detener la ejecución en caso de error), etc.

Los artefactos son archivos que se almacenan tras la ejecución del trabajo.

Pueden ser **informes de pruebas unitarias en formato JUnit**, archivos de log o el resultado de la construcción de un paquete (aunque estos paquetes se suelen archivar en un sistema específico, separado de Jenkins, para su consumo por otros sistemas).

Flujo de ejecución de un pipeline

Cada trabajo de Jenkins se puede entender como un pipeline que se ejecuta múltiples veces: cada ejecución se denomina *build*, incluso aunque la tarea del trabajo ejecute comandos que no tengan nada que ver con la construcción de un paquete.

Cada ejecución corresponde a un *trigger* concreto. En un pipeline de integración continua típico, este podría ser un *webhook* desde GitHub. Jenkins iniciará una nueva ejecución del trabajo y seleccionará un agente al que enviar los detalles de la tarea.

El agente ejecutará las tareas y reportará el estado y los logs al *master*. También reportará el código de salida para detener la progresión del trabajo en caso de fallo o continuar hasta que termine satisfactoriamente. Al finalizar, enviará los artefactos al *master*.

El historial mantiene todos los datos de cada ejecución, los *logs* y los artefactos que se han almacenado.

Los plugins pueden añadir información adicional: por ejemplo, el plugin de Git añade detalles como la rama que ha descargado el trabajo y los *commits* nuevos que han aparecido desde la ejecución anterior. Además, Jenkins puede generar informes con la evolución de cada trabajo.

Jenkinsfiles

Un Jenkinsfile es un archivo escrito en Groovy con los detalles del trabajo, sus fases, los scripts, etc. Son la alternativa a la definición de trabajos a base de formularios web en la interfaz de Jenkins. Además, tal como ya sabemos, se pueden versionar junto al resto del código de la aplicación. Su nombre habitual es, precisamente, Jenkinsfile, pero pueden tener otros nombres sin que por ello pierdan su funcionalidad.

Hay **dos maneras de definir un trabajo** en Jenkins a partir de un Jenkinsfile:

Escribiendo el código del Jenkinsfile directamente en el formulario web

En este caso, el código sigue separado del repositorio de la aplicación, pero ofrece la flexibilidad del lenguaje específico y se puede usar para tareas que no dependan de un repositorio. El trabajo contiene la definición de *triggers*, metadatos, etc.

Indicando la ruta al repositorio y el nombre del Jenkinsfile

En este caso, Jenkins descarga primero el Jenkinsfile, lo interpreta y, entonces, inicia la ejecución del trabajo. Hay que tener en cuenta que el Jenkinsfile contiene la especificación completa del trabajo y puede incluir selectores de agentes, *triggers*, tareas de finalización, etc. Si se han definido *triggers* en el formulario web del trabajo, tendrán prioridad sobre los del Jenkinsfile.

1

Sintaxis declarativa y en script

Los Jenkinsfiles admiten **dos tipos de sintaxis**: declarativa y en script.

EN SCRIPT

DECLARATIVA

La sintaxis en script **es imperativa**, es decir, el usuario debe definir el control de flujo y de errores y depende de expresiones propias de Groovy.

EN SCRIPT**DECLARATIVA**

La [sintaxis declarativa](#) fue **introducida en Jenkins 2**, utiliza un lenguaje específico de dominio (DSL, de Domain-Specific Language), propio de Jenkins, pero basado en Groovy, y delega el control de flujo y de errores en el propio motor de Jenkins. La sintaxis en script es más versátil, pero suele requerir más código. La sintaxis declarativa es más fácil de leer y, por norma general, permite la misma funcionalidad que la sintaxis en script.

En este tema, solo se van a mostrar ejemplos con sintaxis declarativa, ya que es más fácil de entender y está cada vez más extendida.

i **Importante:** en cualquier momento dentro de un bloque declarativo se puede pasar al modo *script*, sin más que usar un bloque *script* dentro de un *step*.

```
stage('Build')

{
steps

{
script
{
// Comandos modo script
}
}
}
```

2

Estructura de un Jenkinsfile

El código de un Jenkinsfile con **sintaxis declarativa** está formado por un bloque pipeline inicial, unas directivas iniciales de configuración, un bloque stage para fase, y unas directivas post al terminar. El siguiente ejemplo contiene algunas de estas opciones.

```
pipeline
  agent
    label
  }
{
{
  'python'
  trigger
  {
    cron('0
      10
      *       *       *')
  }
  environment
  {
    IMAGE_LABEL
    =
      'calculator-app'
  }
```

```
stages

{
  stage('Build')

  {
    steps
    {
      echo
      'Building      stage!'
      sh
      'make  build'
    }
  }

  stage('Push')

  {
    when
    {
      branch
      'master'
    }
  }

  steps
  {
    sh
    'make
    push'
  }
}
```

```
}

stage('Unit
tests')

{
steps
{
sh
'make
test-unit'

archiveArtifacts
artifacts:
'results/unit-result.xml'

}
}

}
}

}
```

Un **trabajo que procese este código** se comportará de esta manera:

- Ejecutará el trabajo en un agente con la etiqueta *python*. Si no hay ninguno disponible, fallará.
- Creará un **cron job** que ejecutará este *job* todos los días a las 10:00.
- La variable de entorno *IMAGE_LABEL* estará disponible en todas las etapas.
- Hay tres etapas (**stages**) que se ejecutarán en orden: *build*, *push* y *unit tests*. Si una de las etapas termina en fallo, las siguientes no se ejecutan.

- Cada etapa contiene una serie de pasos (**steps**).

Los pasos que empiezan por 'sh' son comandos de shell/Powershell (depende del SO del agente). El paso 'echo' imprime un mensaje en el log de Jenkins. El paso 'archiveArchifacts' guardará el fichero indicado como un artefacto generado en el *build*, accesible desde la interfaz gráfica. No se debe usar para almacenar artefactos, pero puede ser útil para persistir logs o resultados de tests.

- El condicional *when* permite elegir cuándo se ejecutará un paso determinado. La etapa *push* solo se ejecutará si se ha lanzado el trabajo sobre la rama *master*. Si se ejecuta para la rama *develop*, por ejemplo, este paso no se ejecutará.

Manejo de secretos

Jenkins permite almacenar, a nivel de *master*, valores secretos que no queremos exponer en los repositorios o que hacen falta para poder ejecutar los *steps* de los pipelines: credenciales de acceso a repositorios o almacenes de artefactos, credenciales de AWS, claves SSH...

Estas credenciales se pueden dar de alta **a nivel global** (accesibles por todos los *jobs*) o restringidas a ciertos proyectos: de este modo, podemos perfilar qué secretos pueden usar los usuarios, evitando que existan *leaks* de datos no intencionados (o, al menos, no poniéndolo fácil).

Jenkins puede **almacenar**:

- Texto en claro, como un token de una API.

- Pares de usuario/contraseña.
- Ficheros (lo mismo que el texto, pero asociado a un fichero), como un *kubeconfig* preconfigurado.
- Claves SSH para acceso a servidores.
- Certificados.



Lo cierto es que el uso avanzado de credenciales es un tanto críptico, por lo que se recomienda repasar la [documentación](#) oficial al respecto.

Un ejemplo, tomado de la web de Jenkins, del **cómo inyectar credenciales AWS** como variables de entorno:

```
environment {  
  
    BITBUCKET_CREDENTIALS = credentials('jenkins-bitbucket-creds')  
  
}
```

Este bloque va a inyectar **tres variables de entorno nuevas**:

BITBUCKET_CREDENTIALS	BITBUCKET_CREDENTIALS_USR	BITBUCKET_CREDENTIALS_PSW
Contiene el usuario y contraseña en el formato '<user>:<password>'.		

BITBUCKET_CREDENTIALS	BITBUCKET_CREDENTIALS_USR	BITBUCKET_CREDENTIALS_PSW
Solo contiene el usuario.		

BITBUCKET_CREDENTIALS	BITBUCKET_CREDENTIALS_USR	BITBUCKET_CREDENTIALS_PSW
Solo contiene la contraseña.		

Este mecanismo de gestión de credenciales, aunque lioso al principio, debemos dominarlo si tenemos que **manejar diferentes perfiles de usuarios o grupos en una única instancia** de Jenkins y queremos hacerlo de forma segura.

Jenkins y contenedores

Una alternativa al uso de agentes físicos preconfigurados es el de emplear Docker para proveer los entornos de trabajo. Esto es altamente flexible, pues nos permite tener identificadas una colección de imágenes que nos servirán para poder ejecutar nuestros *jobs*, imágenes que en muchos casos pueden ser las mismas que se emplean durante el desarrollo, optimizadas para nuestra aplicación y subidas a un *registry* privado.

La **sintaxis** no puede ser más sencilla:

```
pipeline { agent any stages { stage('Build') {  
    agent { docker {  
        image 'gradle:6.7-jdk11'  
    }  
    steps {  
        sh 'gradle --version'  
    }  
    }  
} } }
```

Además, es posible indicar a Jenkins que **construya** un fichero Dockerfile que exista en la raíz del repositorio y lo utilice, usando la keyword siguiente:

```
agent {  
  docker { dockerfile true  
  }  
}
```

Otras herramientas para trabajar con CI/CD: Maven y XRay



[Apache Maven](#) es un proyecto que estandariza la configuración de un proyecto a lo largo de su ciclo de vida al completo.

Por ejemplo, contempla cada una de las fases: compilación, empaquetado, instalación de mecanismos de distribución de librerías..., permitiendo también que puedan ser utilizadas por otros equipos de desarrollo.

Este stack también **contempla el método de trabajo** enfocado a la integración continua, para poder realizar la ejecución de test unitarios y pruebas automatizadas, test de integración, etc.

¿Qué **funcionalidades** podemos destacar de Maven?

- Junto con Eclipse e IntelliJ, es un gran compilador.
- Sistema de gestión dependencias. Maven es el gestor de dependencias por excelencia a la hora de desarrollar aplicaciones con Java y SpringBoot. Su configuración se hace a través del archivo 'pom.xml' y lo que hará será especificar la configuración para construir el proyecto (.jar o .war), descargarás las dependencias desde maven-central (que es un repositorio de código que está en la web y almacena distintas librerías y versiones).

- Mecanismo distribuido de librerías.
 - Posibilidad de creación de plugins customizables.
 - Es multiplataforma ya que es válido para Linux y Windows al estar desarrollado en Java.
 - Es software libre.
 - Fomenta la reutilización de código y librerías.
 - Resulta compatible con múltiples IDEs.
-

Maven fue desarrollada en Java y creada en 2002, posee una funcionalidad similar a Apache Ant, pero tiene un modelo de configuración en formato XML.

En esta imagen te mostramos su árbol genealógico:



Las **partes del ciclo de vida principal** del proyecto Maven son las siguientes:

Compile

Genera los ficheros '.class' compilando las fuentes '.java'.

Test

Ejecuta los tests automáticos Junit.

Package

Genera el fichero '.jar' con los '.class' compilados.

Install

Copia el fichero '.jar' a un directorio de nuestro ordenador donde Maven deja todos los '.jar'.

Deploy

Copia el fichero '.jar' a un servidor remoto.

Existen **funcionalidades adicionales** que Maven ofrece y, aunque no se consideran dentro del ciclo de vida, merece la pena que las conozcas:

- 1 **Clean**: elimina todos los '.class' y '.jar' generados.
- 2 **Assembly**: genera un fichero '.zip' con todo lo necesario para instalar nuestro programa Java.
- 3 **Site**: genera un sitio web, aunque habría que configurarlo dentro de nuestro proyecto.
- 4 **Site-deploy**: sube el sitio web al servidor que se haya configurado.

Como ya hemos indicado, la configuración de Maven se realiza a través de un fichero denominado 'pom.xml'.

POM es un archivo XML del modelo de objetos del proyecto que contiene información sobre el proyecto y los detalles de configuración que Maven necesita para desarrollar el proyecto.

Contiene valores predeterminados para la mayoría de los proyectos. Algunas de las estructuras que se pueden definir en el POM son las dependencias del [proyecto](#), los complementos que se pueden ejecutar y, por supuesto, los perfiles de compilación.

¿Qué elementos se utilizan para la creación del archivo 'pom.xml'?

Proyecto

El elemento raíz del archivo 'pom.xml'.

modelVersion

La versión del modelo POM con el que está trabajando.

groupId

El id del grupo del proyecto. Es único y, con mayor frecuencia, aplicará un ID de grupo relacionado con el nombre del paquete raíz de Java.

artifactId

Para proporcionar el nombre del proyecto que está creando.

Versión

Este elemento consta del número de versión del proyecto. Si tu proyecto ha sido lanzado en varias versiones, entonces, es conveniente presentar la versión de tu proyecto.

Ahora que ya hemos visto por encima las bases de Maven, revisemos las oportunidades que nos ofrece otra plataforma muy útil para trabajar con proyectos de desarrollo, XRay.

El objetivo de XRay es facilitarnos la planificación, el diseño, la ejecución de pruebas y la generación de informes tras dichos test.

Para este proceso, en las pruebas de cada fase, emplea tipos de incidencias específicas de Jira. Veamos cada una de las incidencias frente a la fase del proceso.

FASE DE PLANIFICACIÓN	FASE DE DISEÑO	FASE DE EJECUCIÓN	FASE DE GENERACIÓN DE INFORMES
Incidencias de planificación de pruebas.			

FASE DE PLANIFICACIÓN	FASE DE DISEÑO	FASE DE EJECUCIÓN	FASE DE GENERACIÓN DE INFORMES
La especificación se define mediante los tipos de incidencia de condición previa y de prueba. La organización de las pruebas se puede definir mediante conjuntos de pruebas.			

FASE DE PLANIFICACIÓN	FASE DE DISEÑO	FASE DE EJECUCIÓN	FASE DE GENERACIÓN DE INFORMES
Incidencias de ejecución de la prueba.			

FASE DE PLANIFICACIÓN	FASE DE DISEÑO	FASE DE EJECUCIÓN	FASE DE GENERACIÓN DE INFORMES
Incidencias de ejecución de pruebas, incluidos los informes de cumplimiento de requisitos incorporados y las incidencias personalizadas mediante las herramientas de Jira Software.			

Como podemos intuir, resulta útil añadir pruebas a un proyecto pequeño. Puedes **usar las incidencias de las pruebas para crear pruebas** según tus requisitos y realizar ejecuciones de estas *ad hoc* y no planificadas.



Para ampliar información sobre XRay te recomendamos este post publicado por [Atlassian.com](#).

Conclusión



- A lo largo de estas páginas hemos querido **acercarte al proceso de integración continua** y del despliegue continuo que actualmente se está implementando en la mayoría de las compañías, por ello, hemos definido los términos específicos de este proceso y, posteriormente, te hemos presentado la herramienta 'padre' de todo CI/CD: Jenkins.
- Además, hemos querido ofrecerte **otras alternativas**: la herramienta Maven, para la gestión principalmente de las dependencias, y de XRay, para la realización de pruebas que, toma nota, podrían ser integradas con nuestro Jenkins.
- Cabe destacar que Jenkins no es solo una tecnología para automatizar pruebas de funcionalidad, sino que normalmente se integra con otras herramientas. Un ejemplo de esto sería Kiuwan, plataforma que permite la realización de análisis de código en términos de seguridad lo que mejora a su vez la seguridad de nuestras aplicaciones.
- Por último, como lecturas complementarias para aprender más sobre el tema y, en concreto, sobre Jenkins, te recomiendo que navegues por su web oficial, ya que contiene grandes cantidades de documentación y es una fuente de información muy fiable y recomendable. Sobre todo, estudia con atención la sección dedicada a Pipeline.



Clica en estos enlaces si quieres saber más de [Kiuwan](#) y [Pipeline](#).

¡Enhорabuena! Fastbook superado



Qualentum.com