unir
LA UNIVERSIDAD DE INTERNET

FORMACIÓN PROFESIONAL

UNIDAD FORMATIVA 6

Herramientas de gestión de ciclo de vida

CICD y Jenkins

Índice

CICD y Jenkins

Esta lección va a introducir esa segunda pata fundamental que se identificó en los modelos de SDLC Ágiles, que son los sistemas de construcciones automatizados, generalizados como sistemas de Integración continua y despliegue continuo.

Veremos el porqué de estos dos términos y profundizaremos en por qué en DevOps es tan fundamental está metodología: se trata de una metodología, no de una serie de herramientas sin más, pues implica unos procesos y tareas a los que los equipos deben **adaptarse** si se desea aplicar con éxito y conseguir que asuman los objetivos de calidad del software como suyos.

De hecho, veremos que estos procesos se pueden formalizar en el concepto de ***Pipeline*****,** una serie lógica de fases que nos permiten disponibilizar nuestro software en producción desde el sistema de control de versiones de una manera automática y segura.

Para terminar, concretaremos todo esto con una herramienta específica que nos ayudará a poner en marcha todos estos conceptos en nuestros equipos: un sistema de CICD totalmente funcional, de código libre y alta presencia en el mercado, llamado Jenkins.

Objetivos

1. Repasar la teoría detrás de los sistemas de integración continua y su papel, también trascendental, para el éxito de una organización Ágil.

2. Entender el concepto de *Pipeline* en DevOps.

3. Aprender más sobre Jenkins, su arquitectura y fundamentos de uso.

Integración Continua y Despliegue Continuo

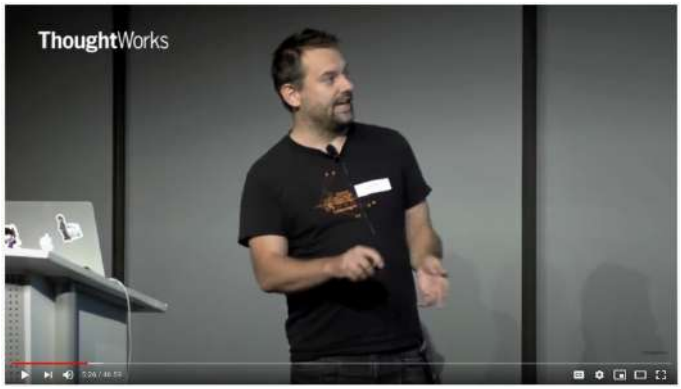
Tenemos ya casi todas las piezas en su sitio: sabemos construir el software, etiquetar el resultado y 'subirlo' a un almacén de artefactos… Nos falta desplegar y listo, ¿verdad?

La verdad es que hasta ahora todo el proceso se ha hecho de modo manual y cualquier proceso manual es tendiente a fallos: el error humano es el más difícil de evitar, por lo que cabe preguntarse: ¿Es posible automatizar este proceso?

La respuesta es obviamente afirmativa: 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…).
- Sabe enviar un correo al responsable de una nueva característica porque rompe test de integración, así como detener el proceso de release.
- Saber si debe hacer *rollback* de una release porque está fallando algo…

ThoughtWorks. *Jez Humble - Continuous Delivery*.



En este caso es Jez Humble, otro de los padres de la entrega continua, quien habla de lo que implica adoptarla, 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.
- Empaquetado.
- 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*), este capítulo va 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.

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** (o el ID del commit, o cualquier otra manera de identificar el código) y **presionar el botón de OK**. El resto de 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.

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:

- 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.

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, editará 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 en el capítulo anterior se pueden evitar si se persiguen dos objetivos: desplegar a menudo y de manera automática.

La **integración continua**, según Martin Fowler, 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.*

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**, según Jez Humble, 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, apps o sistemas embebidos, se puedan llevar a cabo de una manera predecible y rutinaria siempre que haga falta.*

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

El despliegue continuo es 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.

Aunque hay técnicas para resolver las dos primeras técnicas (*feature flags* para la primera y diversas técnicas de despliegue, como *rolling updates*, para la segunda), algunas organizaciones pueden no haber llegado a ese nivel de madurez y optan por separar la integración y la entrega del despliegue.

La idea de aplicar integración y entrega continuas en el desarrollo de software choca frontalmente con las metodologías tradicionales, en las que una versión no se consideraba lista para producción hasta que se habían completado muchas comprobaciones y verificaciones de calidad, en general de manera manual. Sin embargo, si la construcción, las pruebas y el despliegue se automatizan de manera agresiva, la cantidad de tiempo y dinero dedicado a las pruebas manuales antes de liberar cada versión se reducen o desaparecen completamente, ya que la calidad del producto se asegura en cada cambio durante todo el desarrollo.

De hecho, es habitual que la calidad aumente cuando se adapta un proyecto tradicional a una metodología de CICD, ya que los errores se encuentran antes y son más fáciles de corregir: si un desarrollador introduce un error en un cambio y el sistema se lo notifica a los pocos minutos, la idea de lo que acaba de programar aún está fresca y le será más fácil corregirlo. Sin embargo, si el error no aparece hasta varias semanas después, el desarrollador tiene que repasar todo lo que hizo ese día y familiarizarse con la característica en la que estaba trabajando, todo ello antes de siquiera intentar arreglar el problema; todo esto asumiendo que el desarrollador sigue trabajando en la empresa semanas o meses después de haber introducido el cambio.

Otro de los axiomas de Extreme Programming dice que si un proceso es *doloroso* para la organización, la manera de reducir el dolor es hacerlo **más a menudo**. En el caso de las integraciones de código, la manera de hacerlas más inocuas es integrar a menudo. Los cambios serán más pequeños y, por tanto, introducirán menos riesgo.

Esto puede parecer evidente, pero aún es habitual encontrar organizaciones en las que los desarrolladores reciben tareas largas y complejas que requieren semanas de trabajo antes de integrar sus cambios en la rama principal del control de cambios. La posibilidad de que ese cambio introduzca errores es más alta que si la tarea original se hubiera dividido en, por ejemplo, diez subtareas y cada una de ellas se hubiera integrado individualmente.

Los errores de cada tarea se habrían solucionado por separado (eso sin contar con la posibilidad de que dos desarrolladores modifiquen en el mismo fichero y uno de ellos se encuentre con conflictos que deberá resolver antes de poder integrar sus cambios).

La integración continua lleva este paradigma de integrar frecuentemente al extremo y altera profundamente el modelo de desarrollo:

- Se detecta cada error tan pronto como se integra en la rama principal.
- Si se detecta un error, se arregla inmediatamente.
- Si se sigue este flujo, el software siempre está en un estado funcional.
- Si la batería de pruebas (tanto unitarias como funcionales) es suficientemente amplia, es cambio es susceptible de convertirse en una versión liberada en producción.

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:

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

	Nivel de madurez de una organización					
		Construcción y CI	**Entornos y despliegues**	**Liberación de versiones**	**Pruebas**	**Gestión de DB**
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 pases 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.

Tabla 1. Nivel de madurez de una organización. Adaptación del libro citado.

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.

Un pipeline[1] es cualquier sistema que permite encadenar estas fases cumpliendo dos características fundamentales:

- El código y los artefactos generados en cada fase se mueven de una fase a otra de manera **automática**.
- El flujo se repite continuamente, en base a eventos o de forma temporizada.

[1] 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*.

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:

1. Validación y construcción.
2. Control de calidad.
3. Publicación de artefactos.
4. Despliegue.

Validación y construcción

Sin lugar a dudas 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:

1. Modificaciones en alguna rama.
2. Creación de Pull Request.
3. Tarea planificada (por ejemplo, cada noche).

El objetivo de esta fase del pipeline es el de generar un **artefacto**: 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: análisis estático de código, ejecución de tests unitarios, descarga de paquetes, lanzamiento de scripts o 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, 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.

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 sobretodo 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?

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 una 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.

Despliegue

Una vez llegados a esta fase de la 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 encontremos y del tipo de cambio que queramos desplegar. Además, en la siguiente lección profundizaremos mucho más 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:* son cambios que no implican pérdida de servicio de la aplicación y se pueden desplegar sin que los usuarios 'noten nada'.
- 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 front-end con el del back-end (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.

Mejores prácticas en un *pipeline*

En este capítulo se resumen algunas de las mejores prácticas para diseñar pipelines en entornos de integración y entrega continuas.

Compilación única

La compilación del código fuente a partir de una versión o de un commit debería ocurrir **una única vez**. El paquete ya construido (por generalizar a lenguajes no compilados) debería ser el mismo a lo largo del todo el pipeline, es decir, debería ser el mismo sobre el que se ejecutan las pruebas unitarias, las pruebas de validación, de aceptación de usuario e incluso el mismo que se despliega en producción. Si se compila el mismo código múltiples veces puede ocurrir lo siguiente:

- Puede haber cambiado la versión o la configuración del compilador (en caso de que el paquete se construya en equipos diferentes, por ejemplo).
- Puede haber cambiado la versión de una dependencia que debe empaquetarse con el binario.

Por tanto, incluso aunque el código sea el mismo, el binario resultante puede ser diferente y, por tanto, las pruebas dejan de tener valor. Además, incluso aunque el resultado sea idéntico, el proceso de construcción lleva tiempo y uno de los objetivos de los pipelines es ofrecer información al desarrollador lo antes posible.

El problema es similar en lenguajes interpretados ya que, aunque estos no se compilen, es habitual que pasen por un proceso de empaquetado en el que se incluyen dependencias o en el que se aplican técnicas de minimización o de ofuscación. Pequeñas diferencias en la configuración de la herramienta de minimización pueden tener el mismo efecto que el uso de versiones de compilador diferentes en un lenguaje compilado. El problema se repite si el resultado de la primera fase es una imagen de contenedor o una plantilla de máquina virtual: cualquier proceso de construcción de paquetes a partir del código fuente debería llevarse a cabo una única vez.

Por tanto, es recomendable compilar o empaquetar el código una única vez, almacenarlo en un repositorio y obtenerlo de este en todas las etapas del pipeline.

Usar el mismo proceso de despliegue en todos los entornos

Una manera adicional de reducir el riesgo de los despliegues en producción es usar el mismo proceso en todos los entornos, ya sea el portátil de un desarrollador, un entorno efímero para pruebas automáticas o el entorno de producción. La frecuencia de despliegue de cada entorno decrece a medida que aumenta el riesgo:

- Los desarrolladores pueden desplegar sus entornos locales múltiples veces al día, pero el riesgo es cero.
- Lo mismo ocurre con los entornos de pruebas, aunque se despliegan menos a menudo.
- Los despliegues en producción, incluso aun cuando son habituales, son menos numerosos que los despliegues de desarrollo y pruebas. El riesgo es el más alto, ya que es el entorno más importante de todos.

Por tanto, si el proceso de despliegue de producción se prueba múltiples veces antes de desplegarlo realmente, **el riesgo se habrá reducido considerablemente**. Siempre habrá configuraciones específicas, pero si los scripts (o cualquier otra automatización) se han probado, se habrán eliminado muchos errores durante las ejecuciones de bajo riesgo.

Como se acaba de mencionar, cada entorno tendrá configuraciones específicas. Como mínimo, la dirección IP y los nombres de equipo serán diferentes, pero es habitual que haya muchas otras diferencias:

- La configuración del sistema operativo.
- La ubicación de las bases de datos.
- Las credenciales para conectarse a APIs externas.
- Cualquier configuración que no puede fijarse en código.

Aun cuando existan diferencias, el script de despliegue debería ser el mismo. Las configuraciones específicas pueden almacenarse en ubicaciones diferentes, separadas del script. Por ejemplo, se puede preparar un fichero por entorno con las variables específicas de cada uno y versionar todos los ficheros en el sistema de control de cambios. También se pueden almacenar las configuraciones en un sistema externo, específico para gestionar configuraciones. En cada despliegue, el script comprueba qué tipo de entorno está desplegando, bien por el nombre del equipo o, más habitualmente, a partir de una variable de entorno del pipeline (por ejemplo, env=DEV o env=PROD).

Esta separación entre el script de despliegue y la configuración obliga a la colaboración entre equipos, especialmente en grandes organizaciones en las que los entornos de desarrollo, prueba y producción son administrados por equipos diferentes (esto puede ocurrir aun cuando la empresa haya adoptado un modelo DevOps).

Incluso si el proceso de despliegue es aún manual, seguir esta recomendación puede ayudar a reducir el riesgo. El primer paso será establecer un proceso homogéneo de despliegue, seguido de una automatización paulatina, idéntica en todos los entornos.

Una manera de justificar el uso de un mismo proceso en todos los despliegues es que, en caso de fallo, es posible reducir la causa a una de estas tres:

- Una o varias de las opciones de configuración del entorno en el que ha ocurrido el fallo son erróneas.
- Una dependencia de ese entorno (la base de datos, o una API externa) tiene un problema.
- La configuración original del entorno.

Por supuesto, el riesgo no desaparece. Identificar cuál de estas causas es el origen del error no es fácil, pero sería aún más difícil si cada proceso de despliegue añadiera un poco de caos adicional.

Probar los despliegues con un smoke test

El script de despliegue debería incluir un smoke test automático para comprobar que el sistema está funcionando satisfactoriamente. Esta prueba se limita a comprobar que:

- El sistema ha arrancado.
- Los sistemas de los que depende son accesibles y funcionan correctamente.

Por ejemplo, en un servidor web, no es suficiente con comprobar que el proceso esté en ejecución. Una prueba válida sería lanzar una petición HTTP al servidor con un comando externo (por ejemplo, con curl o wget) a una ruta[2] específicamente preparada para escribir y leer en la base de datos y, al menos, leer del sistema de archivos donde se alojan los recursos estáticos. Si el comando curl expira por un timeout o recibe un código de error, el smoke test se considera fallido.

Esta prueba no sustituye a las pruebas de aceptación, ya que no está pensado para comprobar funcionalidades, pero aporta fiabilidad al proceso de despliegue. Idealmente, la información de esta prueba incluirá unos detalles mínimos del fallo para facilitar el análisis por parte del equipo. Siguiendo con el ejemplo anterior, la prueba incluirá el mensaje de error de curl (si falla por un timeout, por un error de red o DNS o por un certificado expirado o inválido) y el código de error, las cabeceras y el cuerpo de la respuesta.

Una ruta bien escrita incluirá suficientes detalles en la respuesta: si han fallado las credenciales de la base de datos, si hubo un error al leer un fichero de disco, etc.

Desplegar una copia de producción

Un problema muy habitual en la industria es que los entornos de producción son muy diferentes de los entornos de desarrollo y de prueba. Esto se puede deber al coste, a la complejidad del entorno o a una combinación de factores. Gracias a la virtualización, estas limitaciones se han visto reducidas, pero sigue siendo común que los desarrolladores dispongan de una versión reducida o simulada del entorno de producción. Para aumentar la confianza en que el despliegue de producción va a funcionar, las pruebas de integración continua deberían llevarse a cabo en entornos tan parecidos como sea posible a producción.

Idealmente, las pruebas se podrían ejecutar en copias idénticas a producción. Esto puede parecer más fácil de lo que es, ya que hay que tener en cuenta lo siguiente:

- La infraestructura (red, firewall, VPNs, etc.) es la misma.
- El sistema operativo y el nivel de parches es el mismo.
- La pila de la aplicación usa las mismas versiones (por ejemplo, en caso de usar un middleware o un servidor de aplicación, como un servidor Tomcat para ejecutar una aplicación Java).
- Los datos de la aplicación están en un estado conocido. Migrar datos durante una actualización puede ser difícil, por lo que es fundamental probarlo antes de actualizar los datos en producción.

[2] El comando sería, por ejemplo, `curl https://servidor.dominio/healthcheck`.

El problema de la infraestructura se puede solventar con despliegues basados en plantillas como los ofrecidos por AWS CloudFormation. La configuración del sistema operativo y del middleware se pueden replicar usando herramientas de automatización de despliegues como Ansible o Puppet, o asegurando que se usa una misma imagen de Docker o de máquina virtual en todos los despliegues.

En caso de fallo, el pipeline ha de detenerse

El equipo de desarrollo debe asumir que los cambios introducidos en el control de versiones deben pasar satisfactoriamente todas las fases del pipeline, incluyendo la construcción y las pruebas. Cualquier fallo debe detener la ejecución del pipeline, impidiendo que se arranquen las fases posteriores y ofreciendo información sobre el fallo a los desarrolladores. El equipo al completo debe hacerse cargo del fallo y arreglarlo para permitir que los nuevos cambios no se vean afectados por un error existente.

El pipeline se puede adaptar a diferentes metodologías de desarrollo

Hasta el momento, se ha asumido que los desarrolladores suben sus cambios directamente a la rama principal del sistema de control de versiones y que las fases de construcción y pruebas se ejecutan con el código de esta rama. Hay otras metodologías de desarrollo y el diseño del pipeline se puede adaptar a todas ellas.

Una metodología habitual es trabajar con Pull Requests (PR) o con una rama nueva para cada funcionalidad o arreglo. Cuando sus cambios están listos, el desarrollador abre una PR con sus cambios. Los sistemas de CI pueden detectar las nuevas pull requests y ejecutar las fases del pipeline sobre ellas, en vez de sobre el código de la rama principal. Esto tiene la ventaja de que no se añade ningún cambio a la rama principal que no haya pasado al menos una cierta batería de pruebas. Entre las desventajas se encuentran:

- El paquete construido en una pull request no se puede desplegar una vez se han fusionado los cambios. Esta metodología facilita el trabajo distribuido, por lo que pueden haber varias pull requests en paralelo. Si se fusionan los cambios de varias de ellas en un mismo momento, el paquete de una no contiene los cambios de otra.

- Del punto anterior se deduce que hay que ejecutar las mismas pruebas también en la rama principal. Esto aumenta el número de pruebas y, por tanto, **el tiempo que tarda en detectarse un error.**

Aun así, las ventajas que aporta el sistema de pull requests al trabajo colaborativo superan a las desventajas, por lo que los sistemas de CICD soportan perfectamente esta metodología.

Adaptar las fases del pipeline a los requisitos del proyecto

Como corolario del apartado anterior, se puede mencionar que las fases de un pipeline no están escritas en piedra: en algunos proyectos se ejecutarán las pruebas de rendimiento diariamente en vez de tras cada cambio (por una limitación de costes, por ejemplo); en otros puede que la compilación o empaquetado de cada módulo sea tan costosa que se hará incrementalmente, empaquetando el módulo B solamente si el módulo A ha pasado las pruebas unitarias correctamente; equipos con fechas límite muy estrictas y con una muy alta rotación de personal pueden decidir desactivar el análisis estático de estilo puntualmente (siempre y cuando hayan asumido la deuda técnica en la que incurrirán).

Las necesidades son tan dispares que hay que reconocer cuándo dividir una fase en dos, cuándo agrupar tareas en una única fase y cuándo un proyecto necesita una fase nueva que no encaja en un pipeline estándar.

Jenkins

Jenkins es una herramienta de automatización que usaremos para 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 post-processing).
- 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[3]. 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[4] 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.

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

[3] Un *webhook* no es más que una llamada a una API REST. Jenkins acepta llamadas de GitHub en la ruta http://<jenkins-host>/github-webhook/. Evaluará el contenido de la llamada para ejecutar un trabajo u otro.

[4] Este modelo se conoce como *polling*; aunque es sencillo de implementar, implica múltiples llamadas de manera continuada.

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[5], etc. En este tipo de trabajos, el usuario podría especificar unos parámetros concretos para cada ejecución.

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. El siguiente capítulo tratará en detalle este aspecto.

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 se indicaba en el capítulo anterior, 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.

[5] 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.

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.

Sintaxis declarativa y en script

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

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.

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 regla 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.

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
    }
  }
}
```

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 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 documentacion 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* - contiene el usuario y contraseña en el formato “<user>:<password>”
- *BITBUCKET_CREDENTIALS_USR* - solo contiene el usuario.
- *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
  }
}
```

Se recomienda revisar la sección correspondiente en la documentación oficial para ver todas las capacidades de Docker en Jenkins, capacidades que añaden una **nueva dimensión** a las posibilidades de la herramienta.

Del mismo modo, se puede instalar un plugin de Kubernetes que nos permitirá lanzar los agentes como pods en un cluster de Kubernetes que tengamos disponible: esta es una opción todavía más potente si cabe que usar Docker, pero es bastante compleja de configurar y, como cualquier cosa relacionada con Kubernetes, aún más compleja de mantener.

Recursos adicionales

jenkins.io

La web oficial de Jenkins contiene grandes cantidades de documentación y es una fuente de información muy fiable y recomendable. Observa la sección dedicada a Pipeline.

What is Jenkins?

What is Jenkins? *Simplilearn*.

Este vídeo resume no solo lo que es Jenkins, sino también los problemas a los que se enfrentaban los desarrolladores antes de usar herramientas de integración continua.

LA UNIVERSIDAD EN INTERNET | FORMACIÓN PROFESIONAL

PROEDUCA