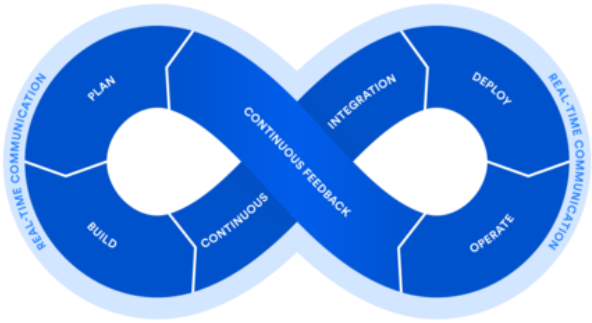
CALIDAD CONTINUA EN ENTORNOS ITERATIVOS

DevOps

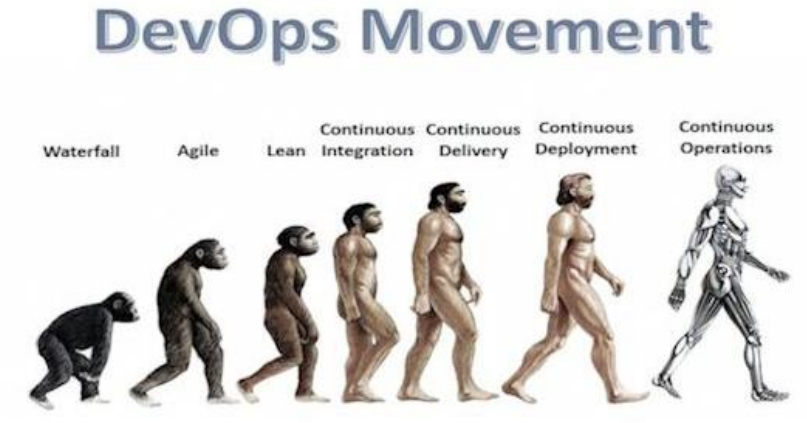
- El término DevOps como tal se popularizó en 2009, a partir de los “DevOps Days” celebrados primero en Gante (Bélgica) y está fuertemente ligado desde su origen a las metodologías ágiles de desarrollo software.
- DevOps está destinado a denotar una estrecha colaboración entre lo que antes eran funciones puramente de desarrollo, funciones puramente operativas y funciones puramente de control de calidad.
- Debido a que el software debe lanzarse a un ritmo cada vez mayor, el antiguo ciclo de desarrollo-prueba-lanzamiento de "cascada" se considera roto. Los desarrolladores también deben asumir la responsabilidad de la calidad de los entornos de prueba y lanzamiento.

Ciclo de vida continuo



- Plan: identificar qué funcionalidad se quiere resolver
- Construir: el desarrollo puro, escribir código, pruebas unitarias y documentar lo que se vea necesario
- Integración Continua: automatizar desde el código hasta el entorno de producción
- Desplegar: automatizar el paso a producción
- Operar: vigilar el correcto funcionamiento del entorno, monitorizar
- Feed back: verificar que la funcionalidad tiene valor para el usuario o el retorno esperado

Movimiento DevOps

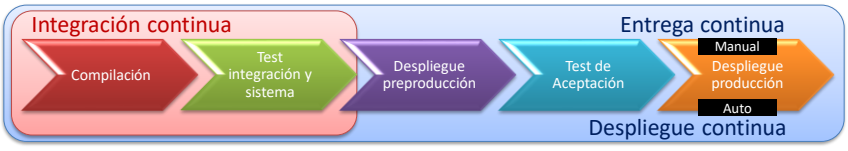


Ciclo “continuo”

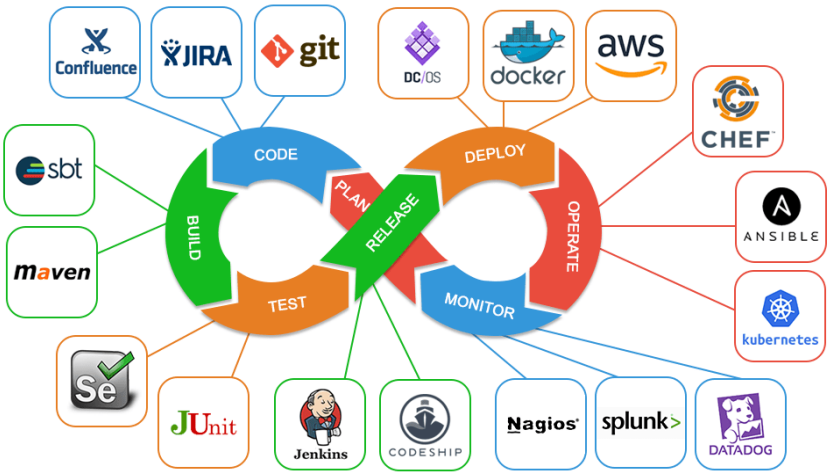
- Muchas empresas se encuentran en algún lugar entre la cascada y las metodologías ágiles. DevOps realmente comienza donde está el “Lean” en la imagen. A medida que se eliminan los cuellos de botella y se comienza a brindar coherencia, podemos comenzar a avanzar hacia la integración continua, la entrega continua y, tal vez, hasta la implementación y las operaciones continuas.
- Con la integración continua, los desarrolladores necesitan escribir pruebas unitarias y se crea un proceso de construcción automatizado. Cada vez que un desarrollador verifica el código, se ejecuta automáticamente una prueba unitaria y, si alguna de ellas falla, la compilación completa falla. Esta es una mejora con respecto al modelo anterior porque se introducen menos errores en el control de calidad y la compilación solo contiene código de trabajo que reduce la acumulación de defectos.
- Al automatizar las pruebas unitarias en los procesos de construcción, eliminamos muchos errores humanos, mejorando así la velocidad y confiabilidad.

Ciclo “continuo”

- La entrega continua CD (continuous delivery) hace referencia a entregar las actualizaciones a los usuarios según estén disponibles sobre una base sólida y constante.
- La despliegue continuo CD (continuous deployment) es la automatización del despliegue de la entrega continua, que no exista intervención humana a la hora de realizar el despliegue en producción.
- Llegar a CI y CD son el objetivo que muchas empresas se esfuerzan por cumplir y están aprovechando el DevOps para ayudarles a lograrlo. Sin embargo, algunas empresas deben ir más allá porque pueden realizar entregas muchas veces al día o tener una presencia web muy popular.
- La implementación continua con solo presionar un botón pasa por la compilación, las pruebas automatizadas, la automatización de los entornos y la producción. Esto se vuelve importante con respecto a las operaciones continuas.



Herramientas de Automatización



Calidad continua

- La popularidad de los enfoques ágiles, desde Scrum hasta DevOps y ágiles a escala, es una de las principales impulsoras del concepto de calidad continua. Existe una necesidad de mejora continua y "calidad a velocidad", que consta de tres pilares:
  - desplazamiento a la izquierda
  - pruebas continuas
  - circuito de retroalimentación continua.
- El desplazamiento a la izquierda se refiere a mover las actividades de pruebas hacia la izquierda del proceso de desarrollo (inicio-final), viendo al mismo como la línea temporal de etapas que se lee de izquierda a derecha. Las pruebas son una más de la suma de actividades que deben coordinarse perfectamente para lograr "calidad a velocidad".
- La calidad continua se apoya en los principios de Automatiza todo lo que se pueda y Todo como código que sigue el DevOps.

Pruebas continuas

- Un tema específico en una estrategia de prueba para DevOps es la automatización de las pruebas. Dado que DevOps suele coincidir con la entrega continua, la mayoría de las pruebas deben realizarse automáticamente durante el proceso. La mayoría de las tareas de ejecución de pruebas deben realizarse automáticamente durante el proceso de integración continua y entrega o despliegue continuo y a menudo se utiliza el término prueba continua.
- Las pruebas continuas son más que solo automatizar pruebas. Se trata de la integración de la automatización de pruebas en las actividades fundamentales de DevOps y en el proceso de CI/CD, enfocado en obtener retroalimentación sobre la calidad del sistema antes y durante la entrega y la implementación.
- Las pruebas continuas ponen la cantidad adecuada de automatización de pruebas (según los riesgos a cubrir) en el proceso de CI/CD, centrándose en ejecutar las pruebas adecuadas en el momento y velocidad adecuados, y estableciendo una estrategia de prueba bien pensada.

Requisitos previos

- La automatización debe realizarse en todas las variedades de pruebas seleccionadas, pero tenga en cuenta que la automatización del 100 % de las pruebas no es un objetivo en sí mismo.
- Las pruebas deben tener un valor comercial y un enfoque de calidad basado en el riesgo.
- Una infraestructura que admita la capacidad de gestionar datos de prueba.
- La capacidad de crear entornos rápidamente y eliminarlos si ya no son necesarios.
- La capacidad de poblar los entornos con los datos correctos.
- La capacidad de generar y mantener automáticamente casos de prueba.
- El tamaño del conjunto de prueba debe ser lo suficientemente pequeño como para ejecutarlo durante un período breve.
- Integración de soluciones de automatización de pruebas con la solución CI/CD.
- Todo el personal de DevOps debe participar en la ingeniería de calidad.
- Las herramientas elegidas cumplen con los requisitos organizativos y técnicos para ser utilizadas de manera eficiente.

Orquestación

- La orquestación de pruebas es la alineación de una gran cantidad de tareas de automatización de pruebas y otras tareas relacionadas con el control de calidad para todos los equipos involucrados en un proceso de CI/CD, para una ejecución optimizada de las pruebas. Este término se refiere tanto al proceso de orquestación como a la implementación técnica del mismo en el proceso.
- La necesidad de mejorar continuamente la velocidad, la cobertura y la calidad de las pruebas ha llevado a una situación en la que aparecen "islas de automatización" en el ciclo de vida que están encadenadas con procesos manuales. Dado que todo el sistema sólo puede moverse tan rápido como su componente más lento, todo el proceso no puede escalar a las demandas de eficiencia y velocidad.
- La orquestación de pruebas elimina las "islas de automatización" al combinar tareas manuales y automatizadas de una manera integral.
- Las políticas de calidad y prueba facilitan la orquestación de pruebas, ya que alinean las actividades de automatización de pruebas en los equipos, lo cual es una condición previa para recopilar información para paneles de control en tiempo real, de extremo a extremo, que informan sobre la confianza para establecer el valor comercial buscado.

Implementar la Orquestación

- Crea visibilidad de los procesos de calidad mediante la implementación de paneles de control de calidad personalizados en todos los procesos de CI/CD .
- Asegúrate de que los equipos de DevOps cuenten con el apoyo adecuado de personas especializadas para tareas para las que los equipos no tienen el conocimiento y/o la capacidad o que se encuentran en otro nivel organizacional.
- Mueve hacia la izquierda las pruebas para identificar fallos en las primeras etapas del ciclo de vida y evitar islas de automatización en la entrega.
- Optimiza las herramientas de automatización de pruebas y las operaciones de prueba en DevOps.
- Automatiza el autoaprovisionamiento de datos de prueba y el aprovisionamiento de entornos de prueba con servicios virtuales y datos de prueba.
- Aprovecha la inteligencia artificial y las tecnologías de aprendizaje automático para optimizar los ciclos de prueba.

Todo como código (EaC)

- Todo como código (Everything as Code) es un enfoque de automatización de las operaciones de TI y DevOps que utiliza el código para definir y gestionar recursos de todo tipo, incluida la infraestructura.
- Todo como código permite a los equipos utilizar archivos de configuración basados en código (script) para gestionar los procesos de CI/CD en la cadena de entrega de software, definir configuraciones o requisitos de seguridad, … Esto no significa que el código gestione todos los recursos o procesos. Siempre será necesario realizar algunas tareas manualmente. Lo que sí significa es que un equipo se compromete a utilizar el código para gestionar los recursos y procesos siempre que sea posible.
- A utilizar código se puede utilizar el Control de versiones para administrarlo, haciendo un seguimiento de las revisiones y del historial de cambios, facilitando puntos de restauración para las reversiones, auditorias, portabilidad, consistencia …
- Así mismo, el código permite a automatización de las:
  - **Pruebas de configuración**: consisten en comprobar todos y cada uno de los dispositivos, en sus propiedades mínimo y máximo posibles.
  - **Pruebas de operación**: consisten en comprobar la correcta implementación de los procedimientos de operación, incluyendo la planificación y control de trabajos, arranque y re-arranque del sistema, etc.
  - **Pruebas de entorno**: consisten en verificar las interacciones del sistema con otros sistemas dentro del mismo entorno.

Todo como código

- **Infraestructura como código** (IaaC): Cada pieza de infraestructura, como máquinas virtuales, aplicaciones web, registros de contenedores, almacenes de claves, etc., debe definirse con scripts (Terraform, Ansible, Kubernetes, …) o plantillas (ARM de Azure, CloudFormation de AWS, …). Esto permitirá controlar las versiones de toda la infraestructura de forma efectiva. Cuando se requiere un cambio de infraestructura, se debe cambiar el script o la plantilla de origen, no la plantilla de destino, que a su vez se implementará mediante implementación continua.
- **Configuración como código** (CaC): A medida que se implementa la infraestructura, también es necesario realizar configuraciones e instalaciones de software encima de la infraestructura, como certificados ZooKeeper, Consul, DNS o SSL. Al configurarlos, todo el proceso se automatiza y no se requieren pasos manuales. Esto garantiza que el estado deseado de los servicios siempre esté definido, cumplido y controlado en origen.
- **Pipelines como código**: Al definir también toda la canalización, señalando la infraestructura, las configuraciones y, en última instancia, el código de software y cómo se implementa, verificado en el control de versiones como todo lo demás, se dedica la máxima cantidad de tiempo a desarrollar funcionalidades en lugar de administrar implementaciones. La configuración de la canalización también define procedimientos de prueba, umbrales de calidad, …
- **Documentación como código**: Toda la documentación debe utilizar un lenguaje de marcado común, como Markdown o Doxygen (HTML y formatos propietarios son conflictivos), y estar versionada en el control de versiones como el resto del código. Markdown o Doxygen se pueden representar en múltiples formatos (HTML, PDF, …), según las necesidades de los usuarios.

Integración y Entrega continua

CI INSTALACIÓN (JENKINS Y SONAR)

Contenedores

- Crear versión de Jenkins con Maven, Node y Docker, creando el fichero llamado Dockerfile:

```
FROM jenkins/jenkins:lts
USER root
RUN apt-get update && apt-get install -y lsb-release maven
RUN curl -fsSLo /usr/share/keyrings/docker-archive-keyring.asc \
  https://download.docker.com/linux/debian/gpg
RUN curl -fsSL https://deb.nodesource.com/setup_18.x | bash - && apt-get install -y nodejs
RUN echo "deb [arch=$(dpkg --print-architecture) \
  signed-by=/usr/share/keyrings/docker-archive-keyring.asc] \
  https://download.docker.com/linux/debian \
  $(lsb_release -cs) stable" > /etc/apt/sources.list.d/docker.list
RUN apt-get update && apt-get install -y docker-ce-cli
USER jenkins
RUN jenkins-plugin-cli --plugins "blueocean docker-workflow jacoco htmlpublisher sonar job-dsl
    maven-plugin pipeline-maven"
```


Contenedores

- Generar imagen Docker:
  - docker build -t jenkins-maven-docker .
- Configurar red docker
  - docker network create jenkins
- Crear contenedores Docker
  - docker run -d --name sonarQube --publish 9000:9000 --network jenkins sonarqube:latest
  - docker run -d --name jenkins-docker-in-docker --detach --privileged --network jenkins --network-alias docker --env DOCKER_TLS_CERTDIR=/certs --volume jenkins-docker-certs:/certs/client --volume jenkins-data:/var/jenkins_home --publish 2376:2376 docker:dind --storage-driver overlay2
  - docker run --name jenkins --network jenkins --env DOCKER_HOST=tcp://docker:2376 --env DOCKER_CERT_PATH=/certs/client --env DOCKER_TLS_VERIFY=1 --publish 50080:8080 --publish 50000:50000 --volume **$(pwd)/**jenkins_home:/home --volume jenkins-data:/var/jenkins_home --volume jenkins-docker-certs:/certs/client:ro --env JAVA_OPTS="-Dhudson.plugins.git.GitSCM.ALLOW_LOCAL_CHECKOUT=true" jenkins-maven-docker
    - Copiar clave generada para proceder a la instalación (/var/jenkins_home/secrets/initialAdminPassword)
    - Sustituir $(pwd)/ por %cd%\ en Windows

MailHog

- MailHog es una herramienta de pruebas de correo electrónico de código abierto dirigida principalmente a los desarrolladores.
- MailHog sirve para emular un servidor de SMTP en local y permite atrapar el correo SMTP saliente, de modo que se puedan ver el contenido de los email sin que estos se envíen realmente.
- Instalación:
  - docker run -d --name mailhog -p 1025:1025 -p 8025:8025 mailhog/mailhog
- Para acceder a la cache del correo saliente:
  - http://localhost:8025/

Red e Instalación de Jenkins

- Configurar red docker
  - docker network create jenkins
  - docker network connect jenkins jenkins
  - docker network connect --alias docker jenkins jenkins-docker-in-docker
  - docker network connect jenkins sonarQube
  - docker network connect jenkins mailhog
- En el navegador: http://localhost:50080/
  - Desbloquear Jenkins con la clave copiada (docker logs jenkins)
  - Install suggested plugins
  - Create First Admin User
  - Instance Configuration Jenkins URL: http://localhost:50080
  - Save and Finish
  - Start using Jenkins

Instalación de Plugins en Jenkins

- Administrar Jenkins > Instalar Plugins
  - JaCoCo
  - Html Publisher
  - SonarQube Scanner
  - Maven Integration
  - Pipeline Maven Integration
  - Docker Pipeline
  - Job DSL
  - Blue Ocean
- Re arrancar al terminar

Configuración de SonarQube

- Entrar SonarQube: http://localhost:9000/
  - Usuario: admin
  - Contraseña: admin
- Configuración
  - Pasar a castellano: Administration > Marketplace > Plugins: Spanish Pack
  - Los webhooks se utilizan para notificar a servicios externos cuando el análisis de un proyecto ha finalizado. Setting > Webhooks
    - Name: jenkins-webhook
    - URL: http://host.docker.internal:50080/sonarqube-webhook
- Generar un user token en Usuario > My Account > Security
    - Generate Tokens: Jenkins,
    - Type: User Token
    - Generar y copiar

Configuración de Plugins en Jenkins

- Administrar Jenkins > Manage Credentials
  - Click link (global) in System table Global credentials (unrestricted).
  - Click Add credentials
    - Kind: Secret Text
    - Scope: Global
    - Secret: Pegar el token generado en SonarQube
    - Description: Sonar Token
- Administrar Jenkins > Configurar el Sistema
  - En SonarQube servers:
    - Add SonarQube
    - Name: SonarQubeDockerServer (estrictamente el mismo usado en Jenkinsfile)
    - URL del servidor: http://sonarQube:9000 (el mismo de la instalación docker)
    - Server authentication token: Sonar Token
    - Guardar

Configuración de Plugins en Jenkins

- Administrar Jenkins > Configurar el Sistema
  - En Notificación por correo electrónico
    - Servidor de correo saliente (SMTP): host.docker.internal
    - Puerto de SMTP: 1025
- Administrar Jenkins > Global Tool Configuration
  - Instalaciones de Maven → Añadir Maven
    - Nombre: maven-plugin
    - Instalar automáticamente > Instalar desde Apache
  - Instalaciones de SonarQube Scanner → Añadir SonarQube Scanner
    - Nombre: SonarQube Scanner
    - Instalar automáticamente > Install from Maven Central

Personalizar el contenedor

- Para entrar en modo administrativo
  - docker container exec -u 0 -it jenkins /bin/bash
- Para actualizar la versión de Jenckins
  - mv ./jenkins.war /usr/share/jenkins/
  - chown jenkins:jenkins /usr/share/jenkins/jenkins.war
- Para instalar componentes adicionales
  - apt-get update && apt-get install -y musl-dev
  - ln -s /usr/lib/x86_64-linux-musl/libc.so /lib/libc.musl-x86_64.so.1

JENKINS

Introducción

- Jenkins es un servidor open source para la integración continua. Es una herramienta que se utiliza para compilar y probar proyectos de software de forma continua, lo que facilita a los desarrolladores integrar cambios en un proyecto y entregar nuevas versiones a los usuarios. Escrito en Java, es multiplataforma y accesible mediante interfaz web. Es el software más utilizado en la actualidad para este propósito.
- Con Jenkins, las organizaciones aceleran el proceso de desarrollo y entrega de software a través de la automatización. Mediante sus centenares de plugins, se puede implementar en diferentes etapas del ciclo de vida del desarrollo, como la compilación, la documentación, el testeo o el despliegue.
- La primera versión de Jenkins surgió en 2011, pero su desarrollo se inició en 2004 como parte del proyecto Hudson. Kohsuke Kawaguchi, un desarrollador de Java que trabajaba en Sun Microsystems, creó un servidor de automatización para facilitar las tareas de compilación y de realización de pruebas.
- En 2010, surgieron discrepancias relativas a la gestión del proyecto entre la comunidad y Oracle y, finalmente, se decidió el cambio de denominación a “Jenkins”. Desde entonces los dos proyectos continuaron desarrollándose independientemente. Hasta que finalmente Jenkins se impuso al ser utilizado en muchos más proyectos y contar con más contribuyentes.

Por qué usarlo

- Antes de disponer de herramientas como Jenkins para poder aplicar integración continua nos encontrábamos con un escenario en el que:
  - Todo el código fuente era desarrollado y luego testeado, con lo que los despliegues y las pruebas eran muy poco habituales y localizar y corregir errores era muy laborioso. El tiempo de entrega del software se prolongaba.
  - Los desarrolladores tenían que esperar al desarrollo de todo el código para poner a prueba sus mejoras.
  - El proceso de desarrollo y testeo eran manuales, por lo que era más probable que se produjeran fallos.
- Sin embargo, con Jenkins (y la integración continua que facilita) la situación es bien distinta:
  - Cada commit es desarrollado y verificado. Con lo que en lugar de comprobar todo el código, los desarrolladores sólo necesitan centrarse en un commit concreto para corregir bugs.
  - Los desarrolladores conocen los resultados de las pruebas de sus cambios durante la ejecución.
  - Jenkins automatiza las pruebas y el despliegue, lo que ahorra mucho tiempo y evita errores.
  - El ciclo de desarrollo es más rápido. Se entregan más funcionalidades y más frecuentemente a los usuarios, con lo que los beneficios son mayores.

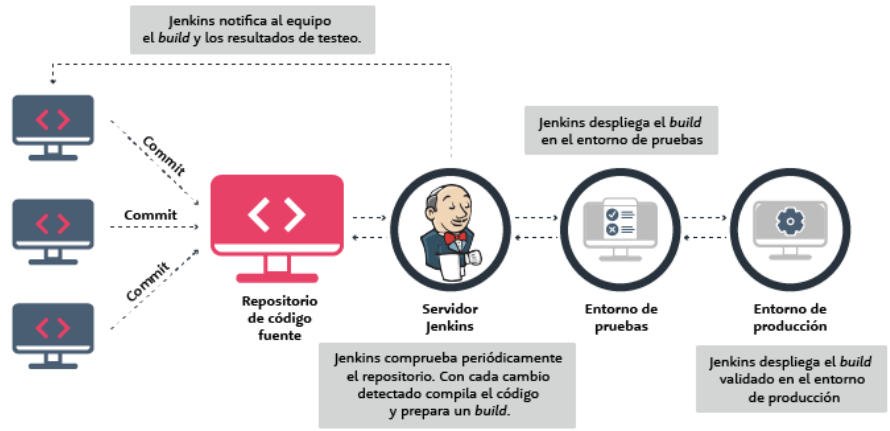
Qué se puede hacer

- Con Jenkins podemos automatizar multitud de tareas que nos ayudarán a reducir el time to market de nuestros productos digitales o de nuevas versiones de ellos. Concretamente, con esta herramienta podemos:
  - Automatizar la compilación y testeo de software.
  - Notificar a los equipos correspondientes la detección de errores.
  - Desplegar los cambios en el código que hayan sido validados.
  - Hacer un seguimiento de la calidad del código y de la cobertura de las pruebas.
  - Generar la documentación de un proyecto.
  - Podemos ampliar las funcionalidades de Jenkins a través de múltiples plugins creados por la comunidad, diseñados para ayudarnos en centenares de tareas, a lo largo de las diferentes etapas del proceso de desarrollo.

Cómo funciona

- Para entender cómo funciona Jenkins vamos a ver un ejemplo de cómo sería el flujo de integración continua utilizando esta herramienta:
  1. Un desarrollador hace un commit de código en el repositorio del código fuente.
  2. El servidor de Jenkins hace comprobaciones periódicas para detectar cambios en el repositorio.
  3. Poco después del commit, Jenkins detecta los cambios que se han producido en el código fuente. Compila el código y prepara un build. Si el build falla, envía una notificación al equipo en cuestión. Si resulta exitoso, lo despliega en el servidor de testeo.
  4. Después de la prueba, Jenkins genera un feedback y notifica al equipo el build y los resultados del testeo.
  5. Jenkins continúa revisando el repositorio frecuentemente y todo el proceso se repite.

Cómo funciona



Pros y Contras

- Ventajas
  - Es sencilla de instalar.
  - Es una herramienta opensource respaldada por una gran comunidad.
  - Es gratuita.
  - Es muy versátil, gracias a sus centenares de plugins.
  - Está desarrollada en Java, por lo que funciona en las principales plataformas.
- Desventajas
  - Su interfaz de usuario es anticuada y poco intuitiva, aunque puede mejorarse con plugins como Blue Ocean.
  - Sus pipelines son complejas y pueden requerir mucho tiempo de dedicación a las mismas.
  - Algunos de sus plugins están desfasados.
  - Necesita de un servidor de alojamiento, que puede conllevar configuraciones tediosas y requerir ciertos conocimientos técnicos.
  - Necesita ampliar su documentación en algunas áreas.

Proyectos

- Un proyecto de construcción de Jenkins contiene la configuración para automatizar una tarea o paso específico en el proceso de creación de aplicaciones. Estas tareas incluyen recopilar dependencias, compilar, archivar o transformar código y probar e implementar código en diferentes entornos.
- Jenkins admite varios tipos de trabajos de compilación
  - Proyecto de estilo libre (freestyle): Esta es la característica principal de Jenkins, la de ejecutar el proyecto combinando cualquier tipo de repositorio de software (SCM) con cualquier modo de construcción o ejecución (make, ant, mvn, rake, script ...). Por tanto se podrá tanto compilar y empaquetar software, como ejecutar cualquier proceso que requiera monitorización.
  - Proyecto Maven: Ejecuta un proyecto maven. Jenkins es capaz de aprovechar la configuración presente en los ficheros POM, reduciendo drásticamente la configuración.
  - Pipeline: Gestiona actividades de larga duración que pueden abarcar varios agentes de construcción. Apropiado para construir pipelines (conocidas anteriormente como workflows) y/o para la organización de actividades complejas que no se pueden articular fácilmente con tareas de tipo freestyle.
  - Proyecto multi-configuración: Adecuado para proyectos que requieran un gran número de configuraciones diferentes, como testear en múltiples entornos, ejecutar sobre plataformas concretas, etc.
  - Carpeta: Crea un contenedor que almacena elementos anidados en él..
  - Multibranch Pipeline: Crea un conjunto de proyectos Pipeline según las ramas detectadas en un repositorio de SCM.

Freestyle Project

- Jenkins Freestyle Project es un trabajo de construcción, secuencia de comandos o canalización repetible que contiene pasos y acciones posteriores a la construcción. Es un trabajo o tarea mejorado que puede abarcar múltiples operaciones. Permite configurar activadores de compilación y ofrece seguridad basada en proyectos para un proyecto de Jenkins. También ofrece complementos para ayudarlo a construir pasos y acciones posteriores a la construcción.
- Aunque los trabajos de estilo libre son muy flexibles, admiten un número limitado de acciones generales de compilación y posteriores a la compilación. Cualquier acción especializada o no típica que un usuario quiera agregar a un proyecto de estilo libre requiere complementos adicionales.
- Los elementos de este trabajo son:
  - SCM opcional, como CVS o Subversion donde reside el código fuente.
  - Disparadores opcionales para controlar cuándo Jenkins realizará compilaciones.
  - Algún tipo de script de construcción que realiza la construcción (ant, maven, script de shell, archivo por lotes, etc.) donde ocurre en trabajo real.
  - Pasos opcionales para recopilar información de la construcción, como archivar los artefactos y/o registrar javadoc y resultados de pruebas.
  - Pasos opcionales para notificar a otras personas/sistemas con el resultado de la compilación, como enviar correos electrónicos, mensajes instantáneos, actualizar el rastreador de problemas, etc.

Configurar un trabajo

- Desechar ejecuciones antiguas determina cuándo se deben descartar los registros de compilación para este proyecto. Los registros de compilación incluyen la salida de la consola, los artefactos archivados y cualquier otro metadato relacionado con una compilación en particular. Mantener menos compilaciones significa que se usará menos espacio en disco en el directorio raíz del registro.
- Los parámetros permiten solicitar a los usuarios una o más entradas que se pasarán a una compilación. Cada parámetro tiene un Nombre y algún tipo de Valor, dependiendo del tipo de parámetro. Estos pares de nombre y valor se exportarán como variables de entorno cuando comience la compilación, lo que permitirá que las partes posteriores de la configuración de la compilación (como los pasos de la compilación) accedan a esos valores, usando la sintaxis ${PARAMETER_NAME}.
- El plugins de git proporciona operaciones fundamentales de git para los proyectos de Jenkins. Puede sondear, buscar, pagar y fusionar contenidos de repositorios de git.
- Los disparadores de ejecuciones automatizan la ejecución del proyecto: Lanzar ejecuciones remotas (ejem: desde 'scripts'), Construir tras otros proyectos, Ejecutar periódicamente, GitHub hook trigger for GITScm polling, Consultar repositorio (SCM)

Configurar un trabajo

- Desechar ejecuciones antiguas determina cuándo se deben descartar los registros de compilación para este proyecto. Los registros de compilación incluyen la salida de la consola, los artefactos archivados y cualquier otro metadato relacionado con una compilación en particular. Mantener menos compilaciones significa que se usará menos espacio en disco en el directorio raíz del registro.
- Los parámetros permiten solicitar a los usuarios una o más entradas que se pasarán a una compilación. Cada parámetro tiene un Nombre y algún tipo de Valor, dependiendo del tipo de parámetro. Estos pares de nombre y valor se exportarán como variables de entorno cuando comience la compilación, lo que permitirá que las partes posteriores de la configuración de la compilación (como los pasos de la compilación) accedan a esos valores, usando la sintaxis ${PARAMETER_NAME}.
- El plugins de git proporciona operaciones fundamentales de git para los proyectos de Jenkins. Puede sondear, buscar, pagar y fusionar contenidos de repositorios de git.
- Los disparadores de ejecuciones automatizan la ejecución del proyecto: Lanzar ejecuciones remotas (ejem: desde 'scripts'), Construir tras otros proyectos, Ejecutar periódicamente, GitHub hook trigger for GITScm polling, Consultar repositorio (SCM)

© JMA 2020. All rights reserved

Configurar un trabajo

- En la ejecución se indica el paso o conjunto de pasos a dar por Jenkins para realizar la construcción de la aplicación. Los pasos pueden ser: ejecutar en línea de comandos, ejecutar Ant o Gradle, ejecutar tareas 'maven' de nivel superior, procesar trabajos DSLs, … Los pasos disponibles dependen de los plugins instalados.
- Se pueden estableces acciones para ejecutar después de la ejecución de los pasos: Notificación por correo, Editable Email Notification, Ejecutar otros proyectos, Guardar los archivos generados, Agregar los resultados de los tests de los proyectos padre, Almacenar firma de ficheros para poder hacer seguimiento, Publicar Javadoc, Publicar los resultados de tests JUnit, Publish HTML reports, Git Publisher, Set GitHub commit status (universal), Delete workspace when build is done.

Notificación por correo

- Una de las acciones para ejecutar después mas típica es enviar un correo electrónico para cada compilación inestable. De esta forma se notifican rápidamente a los desarrolladores de los problemas ocurridos.
- Si está configurada la acción de Notificación por correo en el trabajo, Jenkins enviará un correo electrónico a los destinatarios especificados cuando ocurra un determinado evento importante:
  - Cada compilación fallida desencadena un nuevo correo electrónico.
  - Una compilación exitosa después de una compilación fallida (o inestable) activa un nuevo correo electrónico, lo que indica que la crisis ha terminado.
  - Una compilación inestable después de una compilación exitosa desencadena un nuevo correo electrónico, lo que indica que hay una regresión.
  - A menos que se configure, cada compilación inestable desencadena un nuevo correo electrónico, lo que indica que la regresión todavía está allí.
- La Notificación por correo requiere tener instalado el plugins Mailer y configurarlo en Manage Jenkins > Configure System > E-mail Notification.

Jenkins DSL

- El plugin DSL permite crear varios jobs de forma automática a partir de un job principal llamado seed (semilla o plantilla). De esta forma evitamos tener que volver a crear los mismos jobs para cada nuevo proyecto que precise de tareas similares, pudiendo montar todas las tareas para cada proyecto simplemente ejecutando la tarea semilla. Si modificamos la tarea semilla y volvemos a ejecutarla Jenkins, actualizará los jobs ya creados.
- La tarea semilla es un proyecto de estilo libre que consta básicamente un paso Process Job DSLs de ejecución con un script de Groovy dónde se definen la configuración de las tareas a crear de forma automática: mediante instrucciones se configuran las opciones del proyecto que anteriormente se configuraban manualmente a través del interfaz gráfico.
- Es necesario habilitar los script cada vez que se modifican:
  - Administrar Jenkins > In-process Script Approval

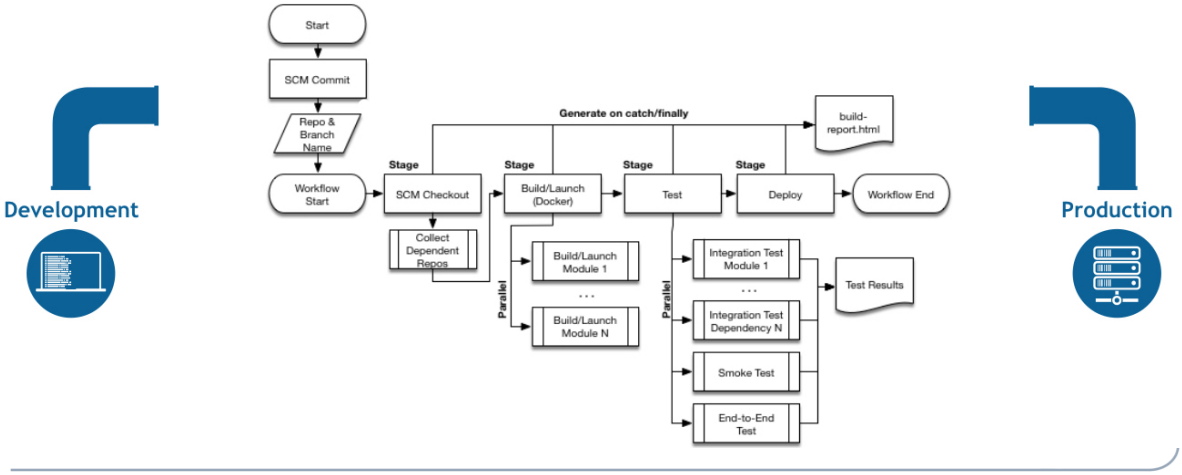
Jenkins DSL

```
job('generado-por-dsl') {
  description('La tarea se ha generado desde otro job')
    parameters {
      booleanParam('FLAG', true)
      choiceParam('OPTION', ['option 1 (default)', 'option 2', 'option 3'])
    }
    scm {
      github('jenkins-docs/simple-java-maven-app', 'master')
    }
    steps {
      shell("echo 'Empiezo el proceso'")
      maven('clean verify')
    }
    publishers {
      mailer('me@example.com', true, true)
    }
}
```

Pipeline

- Un pipeline es una forma de trabajar en el mundo DevOps, bajo la Integración Continua, que nos permite definir el ciclo de vida completo de un desarrollo de software.
- Un pipeline consiste en un flujo comprendido en varias fases que van en forma secuencial, siendo la entrada de cada una la salida de la anterior. Es un conjunto de instrucciones del proceso que sigue una aplicación desde el repositorio de control de versiones hasta que llega a los usuarios.
- El pipeline estaría formado por un conjunto de procesos o herramientas automatizadas que permiten que tanto los desarrolladores como otros roles, trabajen de forma coherente para crear e implementar código en un entorno de producción.
- Cada cambio en el software, lleva a un complejo proceso hasta que es desplegado. Este proceso incluye desde construir software de forma fiable y repetible (conocido como "build"), hasta realizar todos los pasos de testing y los despliegues necesarios.
- Un pipeline Jenkins es un conjunto de plugins que soporta la implementación e integración de pipelines (canalizaciones) de despliegue continuo en Jenkins. Jenkins provee un gran conjunto de herramientas para dar forma con código a un flujo de entrega de la aplicación.

Pipeline



Definición

- La definición de un pipeline Jenkins se escribe en un fichero de texto (llamado Jenkinsfile) que se puede subir al repositorio junto con el resto del proyecto de software.
- Ésta es la base del “Pipeline como código”: tratar la canalización de despliegue continuo como parte de la aplicación para que sea versionado y revisado como cualquier otra parte del código.
- La creación de un Jenkinsfile y su subida al repositorio de control de versiones, otorga una serie de inmediatos beneficios:
  - Crear automáticamente un pipeline de todo el proceso de construcción para todas las ramas y todos los pull request.
  - Revisión del código en el ciclo del pipeline.
  - Auditar todos los pasos a seguir en el pipeline
  - Una sencilla y fiable fuente única, que puede ser vista y editada por varios miembros del proyecto.

Sintaxis de canalización

- Para definir una canalización se puede utilizar la sintaxis de canalización declarativa y la sintaxis de canalización con secuencias de comandos.
- En la sintaxis de canalización declarativa, el bloque pipeline define todo el trabajo realizado a lo largo de todo el Pipeline.

```
pipeline {
    // …
}
```

- En la sintaxis de canalización con secuencias de comandos, uno o más bloques node hacen el trabajo principal en todo el Pipeline. Aunque no es un requisito obligatorio de la sintaxis con secuencias de comandos, confinar el trabajo del Pipeline dentro de un bloque node hace dos cosas:
  - Programa los pasos contenidos dentro del bloque para que se ejecuten agregando un elemento a la cola de Jenkins. Tan pronto como un ejecutor esté libre en un nodo, los pasos se ejecutarán.
  - Crea un espacio de trabajo (un directorio específico para ese canal en particular) donde se puede trabajar en los archivos extraídos del control de código fuente.

```
node {
    // …
}
```

Sintaxis Básica Declarativa

https://www.jenkins.io/doc/book/pipeline/syntax/

- Pipeline {} Identificamos dónde empieza y termina el pipeline así como los pasos que tiene
- Agent. Especificamos cuando se ejecuta el pipeline. Uno de los comandos más utilizados es any, para ejecutar el pipeline siempre y cuando haya un ejecutor libre en Jenkins.
- Node (nodo): Máquina que es parte del entorno de Jenkins y es capaz de ejecutar un Pipeline Jenkins. Los nodos son agrupaciones de tareas o steps que comparten un workspace.
- Stages (etapas). Bloque donde se definen una serie de etapas a realizar dentro del pipeline.
- Stage. Son las etapas lógicas en las que se dividen los flujos de trabajo de Jenkins. Bloque que define una serie de tareas realizadas dentro del pipeline, por ejemplo: build, test, deploy, etc. Podemos utilizar varios plugins en Jenkins para visualizar el estado o el progreso de estos estados.
- Steps (pasos). Son todos los pasos a realizar dentro de un stage. Podemos definir uno o varios pasos.
- Step. Son los pasos lógicos en las que se dividen los flujos de trabajo de Jenkins. Es una tarea simple dentro del pipeline. Fundamentalmente es un paso donde se le dice a Jenkins qué hacer en un momento específico o paso del proceso. Por ejemplo, para ejecutar un comando en shell podemos tener un paso en el que tengamos la línea sh ls para mostrar el listado de ficheros de una carpeta.

Jenkinsfile

```
pipeline {
  agent any
  triggers { // Sondear repositorio a intervalos regulares
    pollSCM('* * * * *')
  }
  stages {
    stage("Compile") {
      steps {
        sh "mvn compile"
      }
    }
    stage("Unit test") {
      steps {
        sh "mvn test"
      }
      post {
        always {
          junit 'target/surefire-reports/*.xml'
        }
      }
    }
    stage("SonarQube Analysis") {
      steps {
        withSonarQubeEnv('SonarQubeDockerServer') {
          sh 'mvn clean verify sonar:sonar'
        }
        timeout(2) { // time: 2 unit: 'MINUTES'
          // In case of SonarQube failure or direct timeout exceed, stop Pipeline
          waitForQualityGate abortPipeline: waitForQualityGate().status != 'OK'
        }
      }
    }
    stage("Build") {
      steps {
        sh "mvn package -DskipTests"
      }
    }
    stage("Deploy") {
      steps {
        sh "mvn install -DskipTests"
      }
    }
  }
}
```

Definición de entornos de ejecución

- La arquitectura de Jenkins está diseñada para entornos de construcción distribuidos. Nos permite usar diferentes entornos para cada proyecto de compilación, equilibrando la carga de trabajo entre múltiples agentes que ejecutan trabajos en paralelo.
- El controlador de Jenkins es el nodo original de la instalación de Jenkins. El controlador de Jenkins administra los agentes de Jenkins y organiza su trabajo, incluida la programación de trabajos en agentes y la supervisión de agentes. Los agentes se pueden conectar al controlador Jenkins mediante equipos locales o en la nube.
- Hay varias formas de definir agentes para usar en Pipeline, como máquinas físicas, máquinas virtuales, clústeres de Kubernetes y con imágenes de Docker.
- Los agentes requieren una instalación de Java y una conexión de red al controlador Jenkins.

Definición de entornos de ejecución

- Al ejecutar una canalización:
  - Jenkins pone en cola todos los pasos contenidos en el bloque para que los ejecute. Tan pronto como haya un ejecutor disponible, los pasos comenzarán a ejecutarse.
  - Se asigna un espacio de trabajo que contendrá archivos extraídos del control de versiones, así como cualquier archivo de trabajo adicional para Pipeline.
- La directiva agent, obligatoria para todas las canalizaciones, le dice a Jenkins dónde y cómo ejecutar todo el Pipeline o un subconjunto del mismo (una etapa específica).

```
pipeline {
    agent any
```

Definición de entornos de ejecución

- La directiva agent permite ejecutar el Pipeline o una etapa:
  - none: Cuando se aplica en el bloque de nivel superior del pipeline, no se asignará ningún agente global para toda la ejecución de Pipeline y cada sección stage deberá contener su propia sección agent. Por ejemplo: agent none
  - any: en cualquier agente disponible. Por ejemplo: agent any
  - label: en un agente definido en el entorno de Jenkins con la etiqueta proporcionada. Se pueden utilizar condiciones de etiqueta. Por ejemplo:agent { label 'my-defined-label' }
  - node: se comporta igual que agent { label 'labelName' }, pero permite opciones adicionales: agent { node { label 'labelName' } }
  - docker: con el contenedor dado que se aprovisionará dinámicamente en un nodo preconfigurado para aceptar Pipelines basados en Docker: agent { docker 'maven:3.8.1-adoptopenjdk-11' }
  - dockerfile: con un contenedor creado a partir de un Dockerfile contenido en el repositorio de origen: agent { dockerfile true }.
  - kubernetes: dentro de un pod implementado en un clúster de Kubernetes.

Disparadores

- La directiva triggers, opcional pero única dentro del bloque pipeline, permite definir localmente las formas automatizadas en las que se debe volver a activar Pipeline. Si están disponibles integraciones basada en webhooks (como GitHub o BitBucket) serán innecesarios. Los activadores actualmente disponibles son:
  - cron: Acepta una cadena de estilo cron para definir un intervalo regular en el que se debe volver a activar:
    ```
    triggers { cron('H */4 * * 1-5') }
    ```
  - pollSCM: Acepta una cadena de estilo cron para definir un intervalo regular en el que Jenkins debe verificar si hay nuevos cambios en el control se versione, en cuyo caso la canalización se volverá a activar:
    ```
    triggers { pollSCM('H */4 * * 1-5') }
    ```
  - upstream: Acepta una lista de trabajos separados por comas y un umbral. Cuando cualquier trabajo en la lista finaliza con el umbral mínimo, la canalización se volverá a activar:
    ```
    triggers { upstream(upstreamProjects: 'job1,job2', threshold: hudson.model.Result.SUCCESS) }
    ```

Variables de entorno

- Las variables de entorno se pueden configurar globalmente o por etapa (solo se aplicarán a la etapa en la que están definidas). Este enfoque para definir variables desde dentro Jenkinsfile puede ser muy útil para compartir valores e instruir scripts, como un Makefile, para configurar la compilación o las pruebas de manera diferente para ejecutarlas dentro de Jenkins.
- La configuración de una variable de entorno dentro de una canalización depende de si se utiliza una canalización declarativa (directiva environment) o con secuencias de comandos (paso withEnv).

```
pipeline {
    agent any
    environment {
        DOCKERHUB_CREDENTIALS = credentials('DockerHub')
    }
    stages {
        stage("SonarQube Analysis") {
            environment {
                scannerHome = tool 'SonarQube Scanner'
            }
            steps {
```


Variables de entorno

- Por convención, los nombres de las variables de entorno suelen especificarse en mayúsculas, con palabras individuales separadas por guiones bajos.
- Las variables de entorno se pueden configurar en tiempo de ejecución y se pueden usar con scripts de shell (sh), por lotes de Windows (bat) y de PowerShell (powershell). Cada script puede devolver el estado (returnStatus) o la salida estándar (returnStdout).

```
environment {
    CC = """${sh(
        returnStdout: true, script: 'echo "clang"'
    )}"""
    EXIT_STATUS = """${sh(
        returnStatus: true, script: 'exit 1'
    )}"""
}
```

- Jenkins Pipeline admite declarar una cadena con comillas simples o comillas dobles, pero la interpolación de cadenas con la notación ${exp} solo está disponible con comillas dobles:

```
sh("curl https://example.com/doc/${EXIT_STATUS}")
sh 'echo $DOCKERHUB_CREDENTIALS_PSW' // sin interpolación
```

Variables de entorno

- Jenkins Pipeline expone las variables de entorno a través de la variable global env, disponible desde cualquier lugar dentro de un archivo Jenkinsfile.
- Los valores mas comúnmente usados son: JOB_NAME, BUILD_ID, BUILD_NUMBER, BUILD_TAG, BUILD_URL, EXECUTOR_NUMBER, JAVA_HOME, JENKINS_URL, NODE_NAME, WORKSPACE.

```
echo "Running ${env.BUILD_ID} on ${env.JENKINS_URL}"
```

- La variable global currentBuild se puede usar para hacer referencia a la compilación que se está ejecutando actualmente: id, number, displayName, projectName, description, result (SUCCESS, UNSTABLE, FAILURE, ...), …

```
mail to: 'team@example.com',
        subject: "Status of pipeline: ${currentBuild.fullDisplayName}",
        body: "${env.BUILD_URL} has result ${currentBuild.currentResult}"
```

- La lista completa está disponible en http://localhost:50080/pipeline-syntax/globals.
- Los complementos de Jenkins también pueden suministrar y establecer variables de entorno, los valores disponibles son específicos de cada complemento .

Credenciales

- Otro uso común para las variables de entorno es establecer o anular credenciales "ficticias" en scripts de compilación o prueba. Debido a que es una mala practica colocar las credenciales directamente en un Jenkinsfile, Jenkins Pipeline permite a los usuarios acceder de forma rápida y segura a las credenciales predefinidas en el Jenkinsfile sin necesidad de conocer sus valores.
- La sintaxis declarativa de Pipeline, dentro de environment, tiene el método de ayuda credentials() que admite credenciales de texto secreto, usuario con contraseña y archivos secretos.

```
environment {
    DOCKERHUB_CREDENTIALS = credentials('DockerHub')
}
```

- Si es una credencial usuario con contraseña, establece las siguientes tres variables de entorno:
  - DOCKERHUB_CREDENTIALS: contiene un nombre de usuario y una contraseña separados por dos puntos en el formato username:password.
  - DOCKERHUB_CREDENTIALS _USR: una variable adicional que contiene solo el nombre de usuario.
  - DOCKERHUB_CREDENTIALS_PSW: una variable adicional que contiene solo la contraseña.
- Para mantener la seguridad y el anonimato de estas credenciales, si el trabajo muestra el valor de estas variables de credenciales desde dentro del Pipeline, Jenkins solo devuelve el valor "****" para reducir el riesgo de que se divulgue información secreta.

Parámetros

- Las canalizaciones admiten su personalización aceptando parámetros especificados por el usuario en tiempo de ejecución. La configuración de parámetros se realza a través de la directiva parameters. Si se configura la canalización para aceptar parámetros mediante la opción Generar con parámetros, se puede acceder a esos parámetros como miembros de la variable params.

```
pipeline {
    agent any
    parameters {
        string(name: 'Greeting', defaultValue: 'Hello', description: 'How should I greet the world?')
    }
    stages {
        stage('Example') {
            steps {
                echo "${params.Greeting} World!"
            }
        }
    }
}
```


Opciones

- La directiva options permite configurar opciones específicas de la canalización desde dentro del propio Pipeline.
- Pipeline proporciona varias de estas opciones, como buildDiscarder, pero también pueden ser proporcionadas por complementos, como timestamps.
- La directiva options es opcional pero solo puede aparecer una vez en la raíz del Pipeline.

```
pipeline {
    agent any
    options {
        retry(3)
        timeout(time: 1, unit: 'HOURS')
    }
    stages {
```

- Cada etapa puede contar con una directiva options pero solo puede contener pasos como retry, timeout, timestamps u opciones declarativas que sean relevantes para un stage, como skipDefaultCheckout.

Etapas y Pasos

- Una canalización de entrega continua es una expresión automatizada del proceso para obtener software desde el control de versiones hasta el despliegue.
- Las canalizaciones se dividen en etapas que se componen de varios pasos que nos permiten crear, probar e implementar aplicaciones. Jenkins Pipeline nos permite componer múltiples pasos de una manera fácil que pueden ayudar a modelar cualquier tipo de proceso de automatización.
- Hay que pensar en un "paso" como un solo comando que realiza una sola acción. Cuando un paso tiene éxito, pasa al siguiente paso. Cuando un paso no se ejecuta correctamente, la canalización fallará. Cuando todos los pasos de la canalización se han completado con éxito, se considera que la canalización se ha ejecutado con éxito.
- La sección stages contine una o más directivas stage. Cada stage debe tener un nombre, una sección steps (con los pasos de la etapa) o stages (etapas anidadas) y, opcionalmente, secciones agent, post,, …

Generador de fragmentos

- La utilidad integrada "Snippet Generator" es útil para crear fragmentos de código para pasos individuales, descubrir nuevos pasos proporcionados por complementos o experimentar con diferentes parámetros para un paso en particular y acceder a la documentación.
- El generador de fragmentos se completa dinámicamente con una lista de los pasos disponibles para la instancia de Jenkins. La cantidad de pasos disponibles depende de los complementos instalados que exponen explícitamente los pasos para su uso en Pipeline. La mayoría de los parámetros son opcionales y se pueden omitir en su secuencia de comandos, dejándolos en los valores predeterminados.
- Para generar un fragmento de paso con el Generador de fragmentos:
  1. Navegar hasta el vínculo Sintaxis de Pipeline (Pipeline Syntax) desde la configuración del proyecto Pipeline o en http://localhost:50080/pipeline-syntax/.
  2. Seleccionar el paso deseado en el menú desplegable Paso de muestra
  3. Usar el área generada dinámicamente debajo del menú desplegable Paso de muestra para configurar el paso seleccionado.
  4. Hacer clic en Generar script de canalización para crear un fragmento de canalización que se puede copiar y pegar en una canalización.

Pasos básicos

- sh: Ejecutar Shell Script
- bat: Ejecutar Windows Batch Script
- powershell: Ejecutar Windows PowerShell Script
- pwsh: Ejecutar PowerShell Core Script

- echo: Escribir en la salida de la consola
- mail: Enviar un correo electrónico
- tool: Utilizar una herramienta de una instalación de herramienta predefinida
- error: Lanzar error
- catchError: Capturar el error y establecer el resultado de compilación en FAILURE
- warnError: Detectar el error y configurar el resultado de compilación y etapa como UNSTABLE
- unstable: Establecer el resultado de la etapa en UNSTABLE

- git: Realizar una clonación desde el repositorio especificado.
- checkout scm: extraer el código del control de versiones vinculado

Pasos básicos

- isUnix: Comprueba si se ejecuta en un nodo similar a Unix
- pwd: Determinar el directorio actual
- dir: Cambiar el directorio actual
- deleteDir: Eliminar recursivamente el directorio actual del espacio de trabajo
- fileExists: Verificar si el archivo existe en el espacio de trabajo
- readFile: Leer archivo desde el espacio de trabajo
- writeFile: Escribir archivo en el espacio de trabajo
- stash: Guardar algunos archivos para usarlos más adelante en la compilación
- unstash: Restaurar archivos previamente escondidos
- archive: Archivar artefactos
- unarchive: Copiar artefactos archivados en el espacio de trabajo

- withEnv: Establecer variables de entorno
- withContext: use un objeto contextual de las API internas dentro de un bloque
- getContext: obtener objeto contextual de las API internas

Tiempos de espera y reintentos

- Hay algunos pasos especiales que "envuelven" a otros pasos y que pueden resolver fácilmente problemas como reintentar (retry) los pasos hasta que tengan éxito o salir si un paso dura demasiado tiempo (timeout). Cuando no se puede completar un paso, los tiempos de espera ayudan al controlador a evitar el desperdicio de recursos. Podemos anidar un retry en un timeout para que los reintentos no excedan un tiempo máximo. Si vence el timeout falla la etapa.

```
steps {
    retry(3) {
        sh './remoteload.sh'
    }
    timeout(time: 3, unit: 'MINUTES') {
        sh './health-check.sh'
    }
}
```

- Con el paso sleep se pausa la construcción hasta que haya expirado la cantidad de tiempo dada.
- El paso waitUntil recorre su cuerpo repetidamente hasta que obtiene true. Si obtiene false, espera un rato y vuelve a intentarlo. No hay límite para el número de reintentos, pero si el cuerpo arroja un error, se propaga inmediatamente.

Terminando

- Cuando la canalización haya terminado de ejecutarse, es posible que deba ejecutar pasos de limpieza o realizar algunas acciones según el resultado del Pipeline. Estas acciones se pueden realizar en la sección post del pipeline o de la etapa.

```
post {
    always {
        echo 'Esto siempre se ejecutará'
    }
    success {
        echo 'Esto se ejecutará solo si tiene éxito'
    }
    failure {
        echo 'Esto se ejecutará solo si falla'
    }
    unstable {
        echo 'Esto se ejecutará solo si la ejecución se marcó como inestable'
    }
}
```

Terminando

- La sección post define uno o más pasos adicionales que se ejecutan al finalizar la ejecución de una canalización o etapa. Admite bloques de condiciones que permiten la ejecución de pasos dentro de cada condición dependiendo del estado de finalización del Pipeline o etapa. Los bloques de condición se ejecutan en el siguiente orden:
  - always: siempre, independientemente del estado de finalización de la ejecución de Pipeline o etapa.
  - changed: cuando tiene un estado de finalización diferente al de su ejecución anterior.
  - fixed: cuando es exitosa y la ejecución anterior falló o era inestable.
  - regression: cuando falla, es inestable o se cancela y la ejecución anterior fue exitosa.
  - aborted: cuando se cancela, generalmente debido a que la canalización se anuló manualmente.
  - failure: cuando falla.
  - success: cuando es exitosa.
  - unstable: cuando es "inestable", generalmente causado por pruebas fallidas, errores de código, …
  - unsuccessful: cuando es no es exitosa.
  - cleanup: cuando termine con las anteriores, independientemente del estado.

Notificaciones

- Dado que se garantiza que la sección post de un Pipeline se ejecutará al final de la ejecución de una canalización, podemos agregar alguna notificación u otras tareas de finales en función de como termine el Pipeline.
- Hay muchas formas de enviar notificaciones sobre un Pipeline cómo enviar notificaciones a un correo electrónico, una sala de Hipchat o un canal de Slack.

```
post {
    failure {
        mail to: 'team@example.com',
            subject: "Failed Pipeline: ${currentBuild.fullDisplayName}",
            body: "Something is wrong with ${env.BUILD_URL}"
    }
}
```

- El paso mail acepta: subject, body, *to, cc, bcc, replyTo, from, charset, mimeType*.

Grabación de pruebas

- Las pruebas son una parte fundamental de una buena canalización de entrega continua.
- Jenkins puede registrar y agregar los resultados de las pruebas siempre que el test runner pueda generar archivos de resultados de pruebas.

```
stage("Unit test") {
    steps {
        sh "mvn test"
    }
    post {
        always {
            junit 'target/surefire-reports/*.xml'
        }
    }
}
```

- Jenkins generalmente incluye el paso junit, pero si el test runner no puede generar informes XML de estilo JUnit, existen complementos adicionales que procesan prácticamente cualquier formato de informe de prueba ampliamente utilizado.

Etapas opcionales

- La directiva when permite que Pipeline determine si la etapa debe ejecutarse según la condición dada. Debe contener al menos una condición, si contiene más de una condición, todas las condiciones deben devolver verdadero para que se ejecute la etapa (allOf). Si se usa una condición anyOf, se omiten las pruebas restantes tan pronto como se encuentra la primera condición "verdadera". Se pueden construir estructuras condicionales más complejas utilizando las condiciones de anidamiento: not, allOf o anyOf, y se pueden anidar a cualquier profundidad arbitraria.
- En la condición pueden participar: branch, buildingTag, changelog, changeset, changeRequest, environment, equals, expression, tag, triggeredBy.

```
stage('Deploy') {
    when {
        branch 'production'
        expression { currentBuild.result == null || currentBuild.result == 'SUCCESS' }
        anyOf {
            environment name: 'DEPLOY_TO', value: 'production'
            environment name: 'DEPLOY_TO', value: 'staging'
        }
    } steps {
```

Paralelismo

- Las etapas en la canalización pueden tener una sección parallel que contenga una lista de etapas anidadas que se ejecutarán en paralelo. Una etapa solo debe una sección steps, stages, parallel o matrix, que son excluyentes.
- Se puede forzar la cancelación de todas las etapas paralelas cuando cualquiera de ellas falla, agregando failFast true al stage que contiene la sección parallel.

```
stage('Parallel Stage') {
    failFast true
    parallel {
        stage('Branch A') {
            agent { label "for-branch-a" }
            steps {
                // ...
            }
        }
        stage('Branch B') {
            agent { label "for-branch-b" }
```


Matrices: múltiples combinaciones

- Las etapas pueden tener una sección matrix que defina una matriz multidimensional de combinaciones de nombre y valor para ejecutarse en paralelo. Nos referiremos a estas combinaciones como "celdas" en una matriz. Cada celda en una matriz puede incluir una o más etapas para ejecutarse secuencialmente usando la configuración para esa celda. Se puede forzar la cancelación del resto de celdas cuando cualquiera de ellas falla, agregando failFast true.
- La sección matrix debe incluir una sección axes y una sección stages. La sección axes define los valores para cada uno ejes en la matriz. La sección stages define las etapas a ejecutar secuencialmente en cada celda. La matriz puede tener una sección excludes para eliminar celdas no válidas de la matriz.

```
matrix {
    axes {
        axis { name 'PLATFORM' values 'linux', 'mac', 'windows' }
        axis { name 'BROWSER' values 'chrome', 'edge', 'firefox', 'safari' }
        axis { name 'ARCHITECTURE' values '32-bit', '64-bit' }
    }
    excludes {
        exclude {
            axis { name 'PLATFORM' values 'mac' }
            axis { name 'ARCHITECTURE' values '32-bit' }
        }
        exclude {
            axis { name 'PLATFORM' values 'linux' }
            axis { name 'BROWSER' values 'safari' }
        }
    }
    failFast true
    stages {
    // ...
    }
}
```

Despliegue

- La canalización de entrega continua más básica tendrá, como mínimo, tres etapas que deben definirse en un Jenkinsfile: construcción, prueba e implementación. Las etapas estables de construcción y prueba son un precursor importante de cualquier actividad de despliegue.
- Un patrón común es ampliar la cantidad de etapas para capturar entornos de implementación adicionales, como "pre producción" o "producción":

```
stage('Deploy - Staging') {
    steps {
        sh './deploy staging'
        sh './run-smoke-tests'
        sh './run-acceptance-tests'    }
}
stage('Deploy - Production') {
    steps {
        sh './deploy production'
    }
}
```


Intervención humana

- La canalización que implementa automáticamente el código hasta la producción puede considerarse una implementación de "despliegue continuo". Si bien este es un ideal noble, para muchos hay buenas razones por las que el despliegue continuo puede no ser práctico, pero aún pueden disfrutar de los beneficios de la entrega continua. Jenkins Pipeline es compatible con ambos.
- A menudo, al pasar entre etapas, especialmente cuando cambia el entorno, es posible que se desee la participación humana antes de continuar. Por ejemplo, para juzgar si la aplicación está en un estado lo suficientemente bueno como para "promocionar" al entorno de producción. Esto se puede lograr con el paso input: bloquea la entrada y no continuará sin que una persona confirme el progreso.

```
stage('Deliver') {
    steps {
        sh './jenkins/scripts/deliver.sh'
        input message: 'Finished using the web site? (Click "Proceed" to continue)'
    }
}
```

Docker con Pipeline

- Muchas organizaciones usan Docker para unificar sus entornos de compilación y prueba en todas las máquinas y para proporcionar un mecanismo eficiente para implementar aplicaciones. Las versiones 2.5 y posteriores de Pipeline tienen soporte integrado para interactuar con Docker desde dentro del Jenkinsfile.
- Pipeline está diseñado para usar fácilmente imágenes de Docker como entorno de ejecución para una sola etapa o toda la canalización. Lo que significa que un usuario puede definir las herramientas requeridas para su Pipeline, sin tener que configurar agentes manualmente. Prácticamente cualquier herramienta que se pueda empaquetar en un contenedor Docker se puede usar con facilidad haciendo solo modificaciones menores en un archivo Jenkinsfile.

```
pipeline {
    agent {
        docker { image 'node:16.13.1-alpine' }
    }
```


multiple-containers

```
pipeline {
    agent none
    stages {
        stage('Back-end') {
            agent {
                docker { image 'maven:3.8.1-adoptopenjdk-11' }
            }
            steps {
                sh 'mvn --version'
            }
        }
        stage('Front-end') {
            agent {
                docker { image 'node:16.13.1-alpine' }
            }
            steps {
                sh 'node --version'
            }
        }
    }
}
```

node-react

```
pipeline {
    agent {
        docker {
            image 'node:lts-buster-slim'
            args '-p 3000:3000'
        }
    }
    environment {
        CI = 'true'
    }
    stages {
        stage ('Clone') {
            steps {
                git url: 'https://github.com/jenkins-docs/simple-node-js-react-npm-app'
            }
        }
        stage('Build') {
            steps {
                sh 'npm install'
            }
        }
        stage('Test') {
            steps {
                sh './jenkins/scripts/test.sh'
            }
        }
        stage('Deliver') {
            steps {
                sh './jenkins/scripts/deliver.sh'
                input message: 'Finished using the web site? (Click "Proceed" to continue)'
                sh './jenkins/scripts/kill.sh'
            }
        }
    }
}
```


Almacenamiento en caché de datos para contenedores

- Muchas herramientas de compilación descargarán dependencias externas y las almacenarán en caché localmente para reutilizarlas en el futuro. Dado que los contenedores se crean inicialmente con sistemas de archivos "limpios", esto puede generar Pipelines más lentos, ya que es posible que no aprovechen las cachés en disco entre ejecuciones posteriores de Pipeline.
- Pipeline admite la adición de argumentos personalizados que se pasan a Docker, lo que permite a los usuarios especificar volúmenes de Docker personalizados para montar, que se pueden usar para almacenar datos en caché en el agente entre ejecuciones de Pipeline.

```
pipeline {
    agent {
        docker {
            image 'maven:3.8.1-adoptopenjdk-11'
            args '-v $HOME/.m2:/root/.m2'
        }
    }
```

Usando un Dockerfile

- Para proyectos que requieren un entorno de ejecución más personalizado, Pipeline también admite la creación y ejecución de un contenedor desde un Dockerfile del repositorio de origen. En contraste con el enfoque anterior de usar un contenedor "listo para usar", usar la sintaxis agent { dockerfile true } creará una nueva imagen a partir de un Dockerfile en lugar de extraer una de Docker Hub .

```
FROM node:16.13.1-alpine
RUN apk add -U subversion
```

- Al enviar esto a la raíz del repositorio de origen, Jenkinsfile se puede cambiar para crear un contenedor basado en este Dockerfile y luego ejecutar los pasos definidos usando ese contenedor:

```
pipeline {
    agent { dockerfile true }
```


Continuous Deployment: NodeJS

```
pipeline {
  agent any
  environment {
   REGISTRY = 'jamarton/mock-web-server'
  }

  stages {
    stage('Checkout') {
      steps {
        // Get Github repo using Github credentials (previously added to Jenkins credentials)
        git url: 'https://github.com/jmagit/MOCKWebServer'
      }
    }
    stage('Install dependencies') {
      steps {
       sh 'npm --version'
       sh "npm install"
      }
    }
    stage('Unit tests') {
      steps {
       // echo 'Run unit tests'
       sh "npm run test"
      }
    }
    stage("SonarQube Analysis") {
      environment {
        // Previously defined in the Jenkins "Global Tool Configuration"
        scannerHome = tool 'SonarQube Scanner'
      }
      steps {
        withSonarQubeEnv('SonarQubeDockerServer') {
          sh "${scannerHome}/bin/sonar-scanner -Dsonar.projectKey=MOCKWebServer \
                -Dsonar.sources=./src \
                -Dsonar.tests=./spec \
                -Dsonar.javascript.lcov.reportPaths=./coverage/lcov.info"
        }
        timeout(5) { // time: 5 unit: 'MINUTES'
          // In case of SonarQube failure or direct timeout exceed, stop Pipeline
          waitForQualityGate abortPipeline: waitForQualityGate().status != 'OK'
        }
      }
    }
    stage('Build docker-image') {
      steps {
        sh "docker build -t ${REGISTRY}:${BUILD_NUMBER} . "
      }
    }
    stage('Deploy docker-image') {
      steps {
       // If the Dockerhub authentication stopped, do it again
       // sh 'docker login'
       // sh "docker push ${REGISTRY}:${BUILD_NUMBER}"
       echo 'docker push'
      }
    }
  }
}
```

Blue Ocean (obsoleto)

- Blue Ocean en su forma actual proporciona una visualización de Pipeline fácil de usar. Estaba destinado a ser un replanteamiento de la experiencia del usuario de Jenkins, diseñado desde cero para Jenkins Pipeline. Blue Ocean estaba destinado a reducir el desorden y aumentar la claridad para todos los usuarios.
- Las principales características de Blue Ocean incluyen:
  - Visualización sofisticada de Pipelines de entrega continua (CD), lo que permite una comprensión rápida e intuitiva del estado de su Pipeline.
  - El editor de Pipeline hace que la creación de Pipelines sea más accesible al guiar al usuario a través de un proceso visual para crear un Pipeline.
  - Personalización para adaptarse a las necesidades basadas en roles de cada miembro del equipo.
  - Determinar con precisión para cuando se necesita intervención o surgen problemas. Blue Ocean muestra dónde se necesita atención, lo que facilita el manejo de excepciones y aumenta la productividad.
  - Integración nativa para sucursales y solicitudes de incorporación de cambios, lo que permite la máxima productividad del desarrollador al colaborar en código en GitHub y Bitbucket.
- *Blue Ocean no recibirá más funciones ni actualizaciones de mejoras.*

Blue Ocean (obsoleto)