DevOps

INTRODUCCIÓN

© JMA 2020. All rights reserved

Introducción

- 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.

Introducción

- DevOps establece una “intersección” entre Desarrollo, Operaciones y Calidad, pero no se rige por un marco estándar de prácticas, es una cultura o movimiento que permite una interpretación mucho más flexible en la medida en que cada organización quiera llevarlo a la práctica, según su estructura y circunstancias.
- El objetivo final de DevOps es minimizar el riesgo de los cambios que se producen en las entregas y dar así un mayor valor tanto a los clientes como al propio negocio.
- La característica clave de DevOps es derribar las tradicionales barreras entre los desarrolladores, testers y especialistas en operaciones (Sistemas / Infraestructura) para que trabajen en colaboración a través de herramientas compartidas, corrigiendo diferencias entre personas y entre objetivos, creando vínculos más cercanos entre los participantes con objetivos en común. Además, incorporar el feedback de los clientes/usuarios en el proceso de desarrollo para acelerar la respuesta a errores y mejoras.

Principios

- Según DASA™ (DevOps Agile Skills Association) DevOps presenta 6 principios necesarios para la entrega de servicios TIC.
- Estos principios son:
  1. Acción centrada en el cliente (valor de actuación, innovación)
  2. Crear con el fin en mente (pensamiento de producto y servicio, mentalidad de ingeniero, colaboración)
  3. Responsabilidad Extremo a Extremo (gestión de gastos, nicho, soporte de rendimiento)
  4. Equipos autónomos interfuncionales (perfiles TI, habilidades complementarias)
  5. Mejora continua (fallos rápidos, experimentos)
  6. Automatizar todo lo que se pueda (mejora de la calidad, maximizar el flujo)

Principios

- Acción centrada en el cliente, que se manifiesta por medio del coraje para exponer las disfunciones y liderar cambios; y en la cultura abierta de innovación necesaria para realizar los cambios. Es imprescindible maximizar el retorno de la inversión existente en software, a la vez que innovar y adoptar nuevas tecnologías.
- Crear teniendo en cuenta el objetivo final, lo que requiere una visión amplia de producto y servicio, y no sólo de proyecto, una colaboración entre todas las partes implicadas desde la concepción hasta la operación.
- Responsabilidad extremo a extremo (end to end), hace que todos los participantes se sientan igualmente responsables del producto en todas su etapas, en lugar de poner foco en completar su parte y olvidarse de los demás. A efectos prácticos esto se traduce en trabajo en equipo, combinar el desarrollo y las operaciones para crear un equipo unilateral que se enfoque en la entrega de objetivos comunes.

Principios

- Equipos autónomos multidisciplinares (cross funtional), no es posible tener la visión y responsabilidad extremos a extremo si no se cuenta con equipos capacitados para hacerse cargo del ciclo de vida completo del producto. Además, estos equipos necesitan tener tanto la competencia como la autonomía necesaria para hacerse realmente responsables de su trabajo.
- Mejora continua. El entorno abierto de innovación mencionado anteriormente requiere un contexto, y sobre todo una actitud que fomente la mejora continua, y el inconformismo. Mejorar requiere experimentar, no tener miedo al fracaso sino aprender de él y medir, que es la única forma de evaluar si se produce o no la mejora buscada.
- Automatizar todo lo que se pueda. La aproximación que promueve DevOps, requiere dedicar los recursos disponibles a las actividades que ofrezcan más valor, dejando de lado las que no lo aporten o automatizando todo lo posible. De esa manera, las personas se centran en lo verdaderamente importante y valioso para el objetivo de la organización.

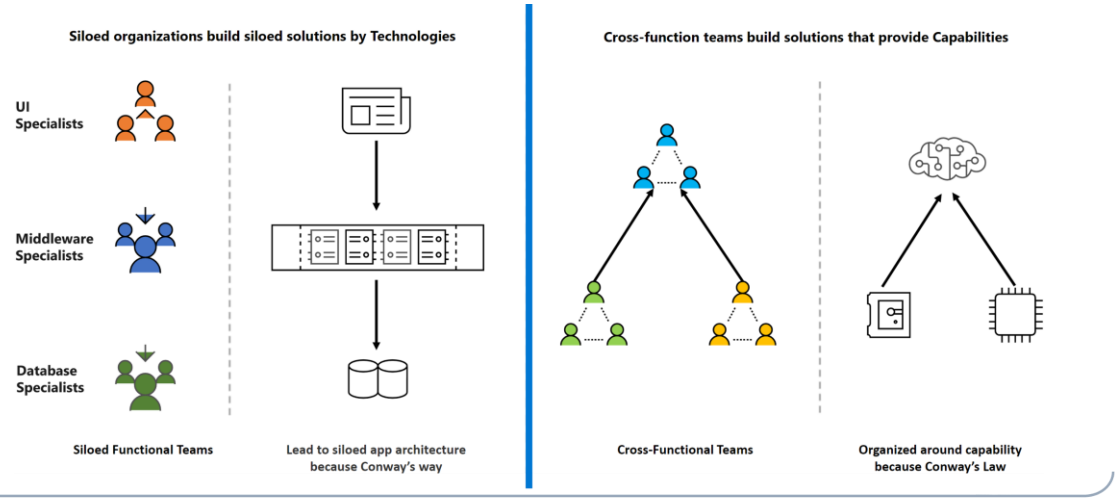
Retos culturales

- Para poder realizar el cambio cultural que supone la implementación del modelo DevOps la organización tiene que ser capaz de trabajar en los siguientes factores:
  - Aprendizaje continuo:
    - Los equipos se auto-organizan, facilitan sesiones de aprendizaje, …
  - Experimentación:
    - Ofrecer tiempo, entornos aislados (sandbox), eliminar barreras, fallo controlado
  - Calidad en cada versión del producto o servicio:
    - Responsabilidad del producto entregado, automatización
  - Cultura de ingeniero:
    - Ambiciones compartidas por el equipo, experimentación
  - Cultura de efectividad:
    - El fin en mente, retrospectivas, filosofía Lean
  - Cultura de pensamiento centrado en el producto o servicio:
    - Feedback de usuario, story boards
  - Cultura de toma de responsabilidades:
    - No explicar ‘cómo’, explicar ‘qué’, transparencia, pensamiento centrado en el cliente

Topologías de equipos

- La distribución de roles, responsabilidades y confianza entre los equipos de TI central, los equipos de las aplicaciones y el equipo de QA es primordial para la transformación operativa involucrada al adoptar DevOps.
- Los equipos de TI se esfuerzan por mantener el control. Los propietarios de aplicaciones buscan maximizar la agilidad. QA busca maximizar la calidad. El equilibrio que se establece en última instancia los diferentes objetivos influye en gran medida en el éxito del modelo DevOps.
- Ley de Conway: cualquier organización que diseña un sistema genera un diseño cuya estructura es una copia de la estructura de comunicación de la organización.
- Según la ley de Conway, los equipos generan arquitecturas basadas en su estructura de comunicación. Comprender este principio es fundamental a medida que trabaja para lograr el equilibrio necesario entre autonomía, control y calidad.

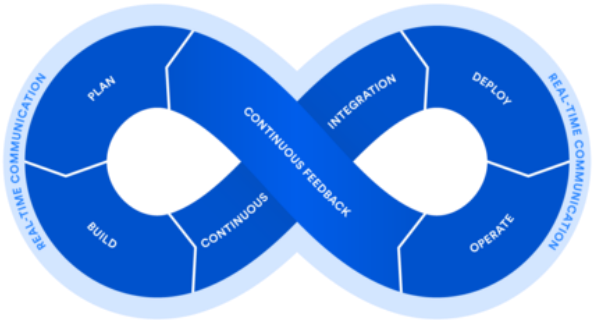
Topologías de equipos



Topologías de equipos

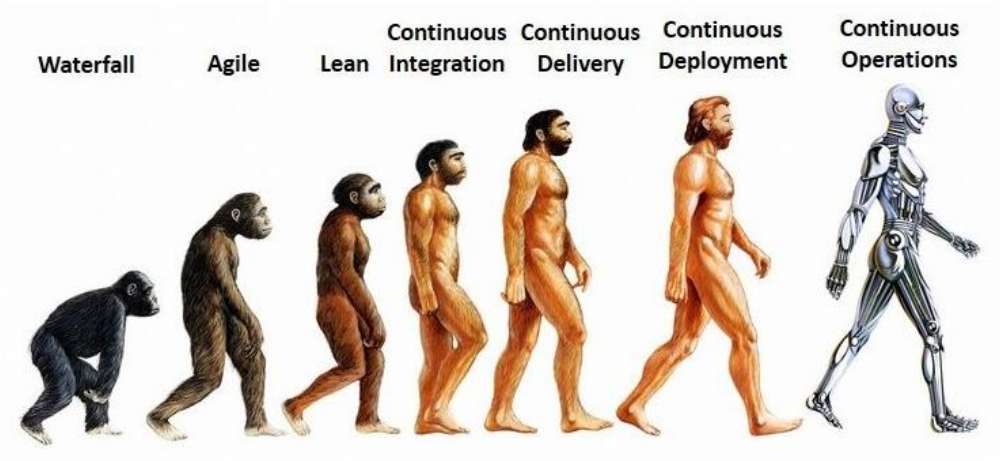
- La solución de Conway es burlar la ley de Conway. Si la organización sigue una estructura determinada para producir servicios y productos y está buscando optimizar, debe replantear la estructura organizativa: evolucionar el equipo y la estructura organizativa para lograr la arquitectura deseada.
- Este principio conduce a topologías de equipo diseñadas intencionadamente en las que los equipos son responsables del extremo a extremo de las aplicaciones, sistemas o plataformas que poseen para lograr la disciplina completa de DevOps.
- Desde una perspectiva de DevOps, las organizaciones deben optimizar la respuesta rápida a las necesidades del cliente. Los equipos que poseen, diseñan e implementan sus aplicaciones y sistemas encuentran su mayor nivel de autonomía en las arquitecturas con las siguientes características:
  - Arquitectura evolucionista que admite cambios constantes
  - Implementabilidad
  - Capacidad de prueba

Ciclo de vida



- Planificar: 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

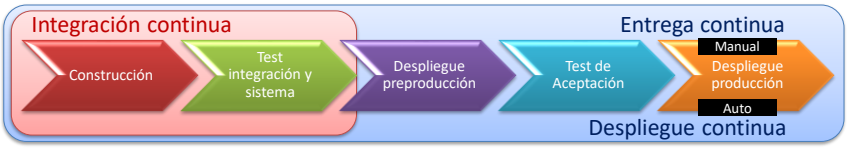


Ciclo “continuo”

- 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 ejecutan automáticamente las pruebas unitarias y, si alguna de ellas falla, la construcción 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.
- En la implementación continua, con solo presionar un botón pasa por la construcción, las pruebas automatizadas, la automatización de los entornos y la producción. Esto se vuelve importante con respecto a las operaciones continuas.



Automatización de la Prueba

- La automatización de la prueba permite ejecutar muchos casos de prueba de forma consistente y repetida en las diferentes versiones del sistema sujeto a prueba (SSP) y/o entornos. Pero la automatización de pruebas es más que un mecanismo para ejecutar un juego de pruebas sin interacción humana. Implica un proceso de diseño de productos de prueba, entre los que se incluyen:
  - Software.
  - Documentación.
  - Casos de prueba.
  - Entornos de prueba
  - Datos de prueba
- La Solución de Automatización Pruebas (SAP) debe permitir:
  - Implementar casos de prueba automatizados.
  - Monitorizar y controlar la ejecución de pruebas automatizadas.
  - Interpretar, informar y registrar los resultados de pruebas automatizadas.

Objetivos de la automatización de pruebas

- Mejorar la eficiencia de la prueba.
- Aportar una cobertura de funciones más amplia.
- Reducir el coste total de la prueba.
- Realizar pruebas que los probadores manuales no pueden.
- Acortar el período de ejecución de la prueba.
- Aumentar la frecuencia de la prueba y reducir el tiempo necesario para los ciclos de prueba.

Ventajas de la automatización de pruebas

- Se pueden realizar más pruebas por compilación ("build").
- La posibilidad de crear pruebas que no se pueden realizar manualmente (pruebas en tiempo real, remotas, en paralelo).
- Las pruebas pueden ser más complejas.
- Las pruebas se ejecutan más rápido.
- Las pruebas están menos sujetas a errores del operador.
- Uso más eficaz y eficiente de los recursos de pruebas
- Información de retorno más rápida sobre la calidad del software.
- Mejora de la fiabilidad del sistema (por ejemplo, repetibilidad, consistencia).
- Mejora de la consistencia de las pruebas.

Desventajas de la automatización de pruebas

- Requiere tecnologías adicionales.
- Existencia de costes adicionales.
- Inversión inicial para el establecimiento de la SAP.
- Requiere un mantenimiento continuo de la SAP.
- El equipo necesita tener competencia en desarrollo y automatización.
- Puede distraer la atención respecto a los objetivos de la prueba, por ejemplo, centrándose en la automatización de casos de prueba a expensas del objetivo de las pruebas.
- Las pruebas pueden volverse más complejas.
- La automatización puede introducir errores adicionales.

Limitaciones de la automatización de pruebas

- No todas las pruebas manuales se pueden automatizar.
- La automatización sólo puede comprobar resultados interpretables por la máquina.
- La automatización sólo puede comprobar los resultados reales que pueden ser verificados por un oráculo de prueba automatizado.
- No es un sustituto de las pruebas exploratorias.

Entornos

- Un enfoque de varios entornos permite compilar, probar y liberar código con mayor velocidad y frecuencia para que la implementación sea lo más sencilla posible. Permite quitar la sobrecarga manual y el riesgo de una versión manual y, en su lugar, automatizar el desarrollo con un proceso de varias fases destinado a diferentes entorno.
  - Desarrollo: es donde se desarrollan los cambios en el software.
  - Prueba: permite que los evaluadores humanos o las pruebas automatizadas prueben el código nuevo y actualizado. Los desarrolladores deben aceptar código y configuraciones nuevos mediante la realización de pruebas unitarias en el entorno de desarrollo antes de permitir que esos elementos entren en uno o varios entornos de prueba.
  - Ensayo/preproducción: donde se realizan pruebas finales inmediatamente antes de la implementación en producción, debe reflejar un entorno de producción real con la mayor precisión posible.
  - UAT: Las pruebas de aceptación de usuario (UAT) permiten a los usuarios finales o a los clientes realizar pruebas para comprobar o aceptar el sistema de software antes de que una aplicación de software pueda pasar a su entorno de producción.
  - Producción: es el entorno con el que interactúan directamente los usuarios.

Herramientas de Automatización

- Una cadena de herramientas de DevOps es una colección de herramientas que permite a los equipos de DevOps colaborar en todo el ciclo de vida del producto y abordar los aspectos básicos clave de DevOps. Las herramientas que incluye una cadena de herramientas de DevOps funcionan como una unidad integrada para la planificación, la integración continua, la entrega continua, las operaciones, la colaboración y los comentarios.
- Se puede adoptar diferentes tipos de cadenas de herramientas de DevOps:
  - Todo en uno: proporciona una solución completa que podría no integrarse con otras herramientas de terceros. Las cadenas de herramientas todo en uno pueden ser útiles para las organizaciones que empiezan su recorrido de DevOps.
  - Personalizado: permite a los equipos traer y mezclar herramientas existentes que conocen y usan ya en la cadena de herramientas de DevOps más amplia. La integración es fundamental para evitar que estos tipos de cadenas de herramientas pasen tiempo innecesario entre pantallas, inicien sesión en varios lugares y se enfrenten al reto de compartir información entre herramientas.

Herramientas: Planificación

- Considere la posibilidad de adoptar una herramienta que admita prácticas de planeación continua:
  - Planificación de la versión
  - Identificación de características y epopeyas
  - Establecimiento de prioridades
  - Estimación
  - Definición de casos de usuario
  - Perfeccionamiento del trabajo pendiente
  - Planificación de un sprint
  - Scrum diario
  - Revisión de sprint
  - Retrospectiva

Herramientas: Integración/entrega continua

- Al implementar Integración continua (CI)/Entrega continua (CD), considere la posibilidad de adoptar una herramienta que admita:
  - Sistemas de control de versiones. Todo el contenido del proyecto debe registrarse en un único repositorio de control de versiones como Git: código, pruebas, scripts de base de datos, scripts de compilación e implementación, y cualquier otra cosa necesaria para crear, instalar, ejecutar y probar la aplicación.
  - Estrategia de ramificación.
  - Compilaciones automatizadas.
  - Tenga en cuenta que su opción de repositorio también se ve influenciada por los requisitos de soberanía y residencia de datos (que deban hospedarse localmente en determinados países o regiones).
- Para minimizar la cantidad de configuración manual necesaria para aprovisionar recursos, considere la posibilidad de adoptar Infraestructura como código (IaC).
- Adopte herramientas de exploración de código para ayudar a detectar defectos de código lo antes posible. Incluya comprobaciones previas a la implementación para validar y confirmar los cambios antes de cualquier función de implementación (ejemplo: "what-if").
- Las herramientas de CI/CD aceleran el tiempo de comercialización del producto. Las herramientas que permiten paralelizar tareas y aprovechar la escalabilidad elástica en la infraestructura hospedada en la nube mejoran el rendimiento del proceso de CI/CD.
- Considere la posibilidad de usar las características de la herramienta de CI/CD que tomen métricas del rendimiento de DevOps. Los paneles e informes pueden realizar un seguimiento de los aspectos del proceso de desarrollo, como el plazo, el tiempo de ciclo, el progreso del trabajo, etc.

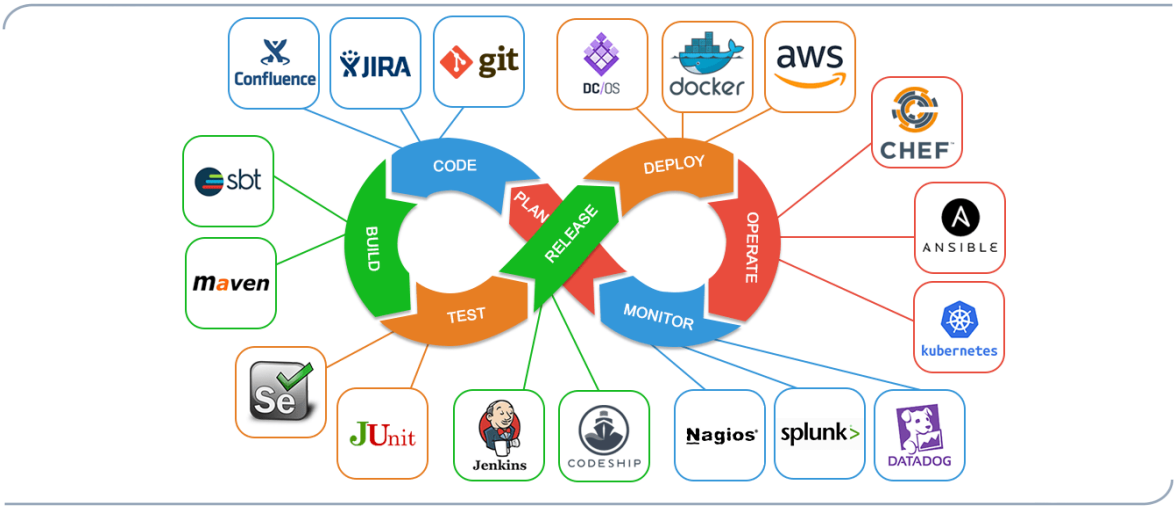
Herramientas: Operaciones continuas

- Las operaciones continuas son un enfoque que ayuda a las organizaciones a mantener la continuidad del resultado entre los clientes y los sistemas internos a través de la entrega ininterrumpida de servicios o funciones críticos. Los objetivos de las operaciones continuas son:
  - Reducir o eliminar la necesidad de tiempos de inactividad planificados o interrupciones, como el mantenimiento programado, la optimización de la capacidad y la implementación.
  - Aumentar la confiabilidad y la resistencia generales de los sistemas en tres aspectos: personas, procesos y herramientas.
- Use herramientas nativas de nube para:
  - Supervisar las métricas clave para el rendimiento y la disponibilidad del servicio.
  - Obtener experiencia digital y Customer Insights.
  - Genere respuestas basadas en inteligencia para incidentes, la recuperación del sistema o el escalado.
  - Automatice el mantenimiento proactivo y tareas como la implementación o las actualizaciones del sistema.

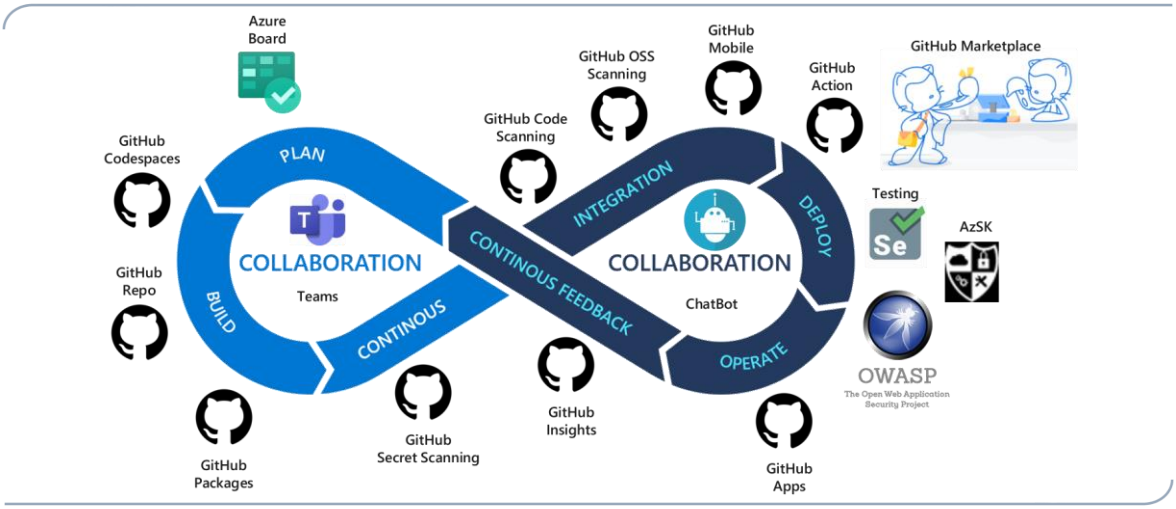
Herramientas: Colaboración y comentarios

- Los bucles de comentarios rápidos constituyen el núcleo del proceso de CI/CD. Una herramienta de CI/CD usa comentarios para resolver condiciones en la lógica de flujo de trabajo de CI/CD y muestra información a los usuarios, normalmente a través de un panel.
- La compatibilidad con las notificaciones por correo electrónico y la integración con IDE o plataformas de comunicación garantiza que pueda mantenerse informado sobre lo que sucede sin tener que comprobar un panel. Asegúrese de disponer de la opción de configurar las alertas que va a recibir, ya que obtener demasiadas alertas hace que se transformen en ruido de fondo (spam).
- Cualquier herramienta que elija para la colaboración debe admitir las siguientes prácticas de colaboración:
  - Colaboración Kanban
  - Colaboración de contenido wiki
  - Colaboración ChatOps
  - Centro de reunión

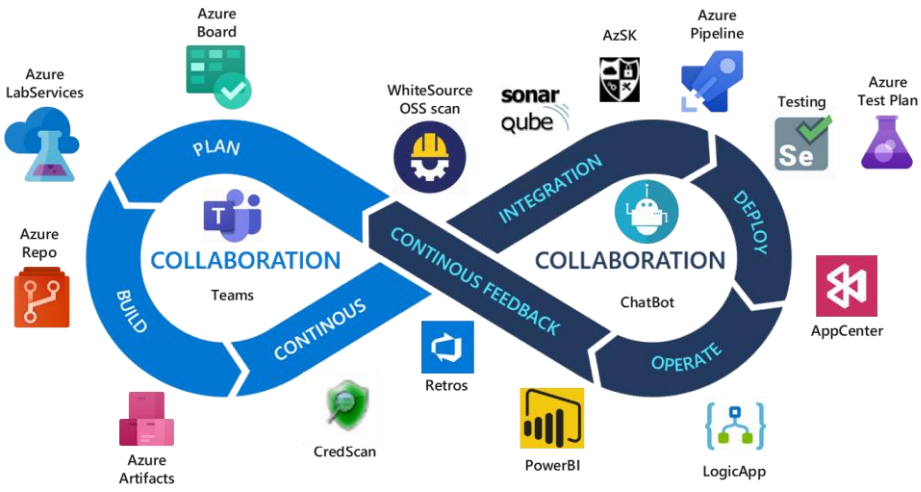
Herramientas de Automatización



GitHub



Azure DevOps



Azure DevOps

- Los puntos del esquema anterior exponen cómo sería un proceso de gestión del ciclo de vida de la una aplicación:
  1. Podemos usar nuestro Visual Studio para desarrollar nuestro código, aunque no es necesario usar esta herramienta en particular, podemos usar cualquier herramienta de desarrollo de código.
  2. De manera regular podremos ir guardando nuestro código en nuestro repositorio de Git dentro del módulo de Azure Repos.
  3. Mediante la integración continua, podemos desencadenar pruebas y la compilación de la aplicación mediante el módulo Azure Test Plans.
  4. Tras esto, con la implementación automatizada de los artefactos y la aplicación de valores de configuración específicos por entornos, podremos realizar despliegues a cada uno de los entornos que compongan nuestra infraestructura con los Azure Pipelines.
  5. En el caso de que no dispongamos de Azure Web Apps, los artefactos anteriormente mencionados también se pueden desplegar a nuestros servidores locales.
  6. Azure Application Insights es otra herramienta propia de Application Performance Management (APM) extensible para desarrolladores web en varias plataformas. Que pueden usar para supervisar y administrar la información sobre el estado, el rendimiento y el uso.
  7. Una vez analizados los resultados podemos volver al código.
  8. Mediante Azure Boards podremos hacer un seguimiento del trabajo diario y pendiente de los equipos

Prácticas de DevOps

- Más allá del establecimiento de una cultura de DevOps, los equipos ponen en práctica el método DevOps implementando determinadas prácticas a lo largo del ciclo de vida de las aplicaciones.
- Algunas de estas prácticas ayudan a agilizar, automatizar y mejorar una fase específica.
- Otras abarcan varias fases y ayudan a los equipos a crear procesos homogéneos que favorezcan la productividad.
- Estas practicas incluyen:
  - Control de versiones
  - Desarrollo ágil de software
  - Automatización de pruebas
  - Integración y entrega continuas (CI/CD)
  - Infraestructura como código
  - Administración de configuración
  - Supervisión continua

Control de versiones

- Control de versiones es la práctica de administrar el código y la documentación por versiones, haciendo un seguimiento de las revisiones y del historial de cambios para facilitar la revisión y la recuperación del código.
- Esta práctica suele implementarse con sistemas de control de versiones, como Git, que permite que varios desarrolladores colaboren para crear código. Estos sistemas proporcionan un proceso claro para fusionar mediante combinación los cambios en el código que tienen lugar en los mismos archivos, controlar los conflictos y revertir los cambios a estados anteriores.
- El uso del control de versiones es una práctica de DevOps fundamental que ayuda a los equipos de desarrollo a trabajar juntos, dividir las tareas de programación entre los miembros del equipo y almacenar todo el código para poder recuperarlo fácilmente si fuese necesario.
- El control de versiones es también un elemento necesario en otras prácticas, como la integración continua y la infraestructura como código.

SEMVER

- El sistema SEMVER (Semantic Versioning) es un conjunto de reglas para proporcionar un significado claro y definido a las versiones de los proyectos.
- El sistema SEMVER se compone de 3 números, siguiendo la estructura X.Y.Z:
  - X se denomina Major: indica cambios rupturistas
  - Y se denomina Minor: indica cambios compatibles con la versión anterior
  - Z se denomina Patch: indica resoluciones de bugs (compatibles)
- Básicamente, cuando se arregla un bug se incrementa el patch, cuando se introduce una mejora se incrementa el minor y cuando se introducen cambios que no son compatibles con la versión anterior, se incrementa el major.
- De este modo cualquier desarrollador sabe qué esperar ante una actualización de su librería favorita. Si sale una actualización donde el major se ha incrementado, sabe que tendrá que ensuciarse las manos con el código para pasar su aplicación existente a la nueva versión.

Políticas de versionado

- Dentro de la política de versionado es conveniente planificar la obsolescencia y la política de desaprobación.
- La obsolescencia programada establece el periodo máximo, como una franja temporal o un número de versiones, en que se va a dar soporte a cada versión, evitando los sobrecostes derivados de mantener versiones obsoletas indefinidamente.
- Dentro de la política de desaprobación, para ayudar a garantizar que los consumidores tengan tiempo suficiente y una ruta clara de actualización, se debe establecer el número de versiones en que se mantendrá una característica marcada como obsoleta antes de su desaparición definitiva.
- La obsolescencia programada y la política de desaprobación beneficia a los consumidores del producto porque proporcionan estabilidad y sabrán qué esperar a medida que los productos evolucionen.
- Para mejorar la calidad y avanzar las novedades, se podrán realizar lanzamientos de versiones Beta y Release Candidatos (RC) o revisiones para cada versión mayor y menor. Estas versiones provisionales desaparecerán con el lanzamiento de la versión definitiva.

Desarrollo ágil de software

- El método ágil es un enfoque de desarrollo de software que hace hincapié en la colaboración en equipo, en los comentarios de los clientes y usuarios, y en una gran capacidad de adaptación a los cambios mediante ciclos cortos de lanzamiento de versiones.
- Los equipos que practican la metodología ágil proporcionan mejoras y cambios continuos a los clientes, recopilan sus comentarios y, después, aprenden y ajustan el software en función de lo que el cliente quiere y necesita.
- El método ágil es muy diferente a otros marcos más tradicionales, como el modelo en cascada, que incluye ciclos de lanzamiento de versiones largos definidos por fases secuenciales. Kanban y Scrum son dos marcos populares asociados al método ágil.

Infraestructura como código

- La infraestructura cómo código define las topologías y los recursos del sistema de un modo descriptivo que permite a los equipos administrar esos recursos igual que lo harían con el código. Las diferentes versiones de esas definiciones se pueden almacenar en sistemas de control de versiones, donde se pueden revisar y revertir, de nuevo, igual que el código.
- La práctica de la infraestructura como código permite a los equipos implementar recursos del sistema de un modo confiable, repetible y controlado. Además, la infraestructura como código ayuda a automatizar la implementación y reduce el riesgo de errores humanos, especialmente en entornos complejos de gran tamaño.
- Esta solución repetible y confiable para la implementación de entornos permite a los equipos mantener entornos de desarrollo y pruebas que sean idénticos al entorno de producción. De igual modo, la duplicación de entornos en otros centros de datos y en plataformas en la nube es más sencilla y más eficiente.

Administración de configuración

- Administración de la configuración hace referencia a la administración del estado de los recursos de un sistema, incluidos los servidores, las máquinas virtuales y las bases de datos.
- El uso de herramientas de administración de la configuración permite a los equipos distribuir cambios de un modo controlado y sistemático, lo que reduce el riesgo de modificar la configuración del sistema. Los equipos utilizan herramientas de administración de la configuración para hacer un seguimiento del estado del sistema y evitar alteraciones en la configuración, que es como se desvía la configuración de un recurso del sistema a lo largo del tiempo del estado definido para él.
- Al usarla junto con la infraestructura como código, resulta fácil elaborar plantillas y automatizar la definición y la configuración de sistemas, lo que permite a los equipos usar entornos complejos a escala.

Supervisión continua

- Supervisión continua significa tener visibilidad total y en tiempo real del rendimiento y el estado de toda la pila de aplicaciones, desde la infraestructura subyacente donde se ejecutan las aplicaciones hasta los componentes de software de niveles superiores.
- La visibilidad consiste en la recopilación de datos de telemetría y metadatos, así como en el establecimiento de alertas para condiciones predefinidas que garanticen la atención de un operador. La telemetría incluye registros y datos de eventos recopilados de varias partes del sistema que se almacenan donde pueden analizarse y consultarse.
- Los equipos de DevOps de alto rendimiento se aseguran de establecer alertas útiles que les permitan tomar medidas y recopilan datos de telemetría muy completos para obtener conclusiones a partir de enormes cantidades de datos.
- Estas conclusiones ayudan a los equipos a mitigar los problemas en tiempo real y a ver cómo mejorar la aplicación en futuros ciclos de desarrollo.

DevSecOps

DevSecOps

- DevSecOps significa desarrollo, seguridad y operaciones. Se trata de un enfoque que aborda la cultura, la automatización y el diseño de plataformas, e integra la seguridad como una responsabilidad compartida durante todo el ciclo de vida.
- DevOps no solo concierne a los equipos de desarrollo y operaciones. Si desea aprovechar al máximo la agilidad y la capacidad de respuesta de los enfoques de DevOps, la seguridad de la TI también debe desempeñar un papel integrado en el ciclo de vida completo de sus aplicaciones.
- Antes, el papel de la seguridad estaba aislado y a cargo de un equipo específico en la etapa final del desarrollo. Cuando los ciclos de desarrollo duraban meses o incluso años, no pasaba nada. Pero eso quedó en el pasado. Una metodología efectiva de DevOps garantiza ciclos de desarrollo rápidos y frecuentes (a veces de semanas o días), pero las prácticas de seguridad obsoletas pueden revertir incluso las iniciativas de DevOps más eficientes.

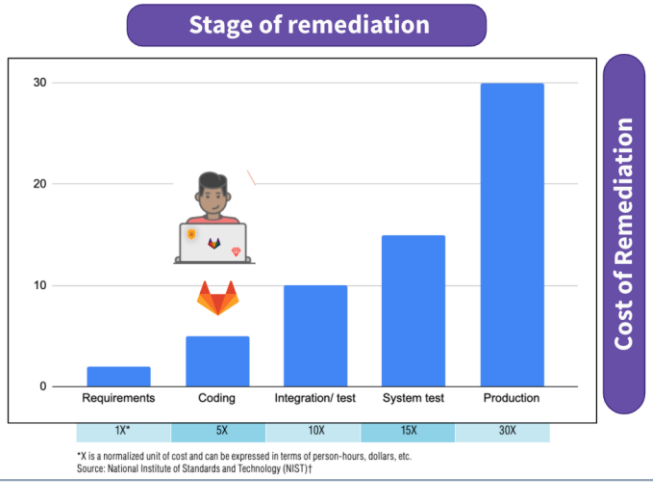
Desafíos

- **Los equipos son reacios a integrarse**: La esencia de DevSecOps es la integración de equipos para que puedan trabajar juntos en lugar de forma independiente. Sin embargo, no todo el mundo está preparado para hacer el cambio porque ya está acostumbrado a los procesos de desarrollo actuales.
- **Guerra de herramientas**: Dado que los equipos han estado trabajando por separado, han estado utilizando diferentes métricas y herramientas. En consecuencia, les resulta difícil ponerse de acuerdo sobre dónde tiene sentido integrar las herramientas y dónde no. No es fácil reunir herramientas de varios departamentos e integrarlas en una plataforma. Por lo tanto, el desafío es seleccionar las herramientas adecuadas e integrarlas adecuadamente para construir, implementar y probar el software de manera continua.
- **Implementar seguridad en IC/DC**: Generalmente, se ha pensado en la seguridad como algo que llega al final del ciclo de desarrollo. Sin embargo, con DevSecOps, la seguridad es parte de la integración y el desarrollo continuos (IC/DC). Para que DevSecOps tenga éxito, los equipos no pueden esperar que los procesos y herramientas de DevOps se adapten a los viejos métodos de seguridad. Al integrar los controles de seguridad en DevOps, las organizaciones están adoptando el nuevo modelo DevSecOps para aprovechar todo el potencial de IC/DC. Cuando las empresas implementan tecnologías de control de acceso o seguridad desde el principio, se aseguran de que esos controles estén en línea con un flujo de IC/DC.

Beneficios

- Desarrollar software seguro desde el diseño.
- Identificación temprana de vulnerabilidades en el código y ataques.
- Aplicar la seguridad de forma más rápida y ágil.
- Responder a los cambios y requisitos rápidamente.
- Mejorar la colaboración y comunicación entre equipos.
- Generar una mayor conciencia sobre seguridad entre todos los miembros.
- Enfocarse en generar el mayor valor de negocio.

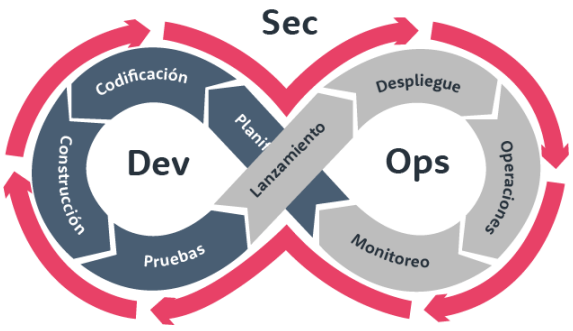
Detección temprana



Implementación

- DevSecOps supone un cambio en la filosofía de trabajo, donde la seguridad se convierte en una responsabilidad compartida (extendida a los equipos de desarrollo y operaciones) e integrada a lo largo de todo el proceso de desarrollo y despliegue. Para poner en práctica DevSecOps, la seguridad debe convertirse en una condición más en el proceso de diseño, desarrollo y entrega de software.
- DevSecOps consta de dos partes diferenciadas pero que forman parte del mismo proceso.
  - Seguridad como código (SaC): cuando se incorporan la seguridad al desarrollo y las operaciones de herramientas informáticas o aplicaciones.
  - Infraestructura como código (IaC): hace referencia al conjunto de herramientas de desarrollo de operaciones (DevOps) utilizadas para configurar y actualizar los componentes de la infraestructura para garantizar un entorno controlado. Trata las infraestructuras como un software.

Ciclo de vida



- Asesorar a los desarrolladores sobre las vulnerabilidades de seguridad para ayudar a configurar las herramientas y las opciones de diseño.
- Implementar pruebas de seguridad automatizadas para la arquitectura, diseño y despliegue en cada proyecto nuevo.
- Trabajar con el equipo de operaciones para llevar a cabo un mantenimiento de seguridad regular y actualizaciones de la infraestructura.
- Usar técnicas de monitorización continua para asegurar que los sistemas de seguridad están trabajando según lo previsto.

Buenas practicas

- **Implemente la automatización para proteger el entorno de IC/DC**: Uno de los aspectos clave del entorno IC/DC es la velocidad. Y eso significa que la automatización es necesaria para integrar la seguridad en este entorno, al igual que incorporar los controles y pruebas de seguridad esenciales a lo largo del ciclo de vida del desarrollo. También es importante implementar pruebas de seguridad automatizadas en las canalizaciones de IC/DC para permitir el análisis de vulnerabilidades en tiempo real.
- **Aborde los problemas de seguridad de la tecnología de código abierto**: Está aumentando el uso de herramientas de código abierto para el desarrollo de aplicaciones. Por lo tanto, las organizaciones deben abordar las preocupaciones de seguridad relacionadas con el uso de dichas tecnologías. Sin embargo, dado que los desarrolladores están demasiado ocupados para revisar el código fuente abierto, es importante implementar procesos automatizados para administrar el código fuente abierto, así como otras herramientas y tecnologías de terceros. Por ejemplo, utilidades como Open Web Application Security Project (OWASP) pueden verificar que no haya vulnerabilidades en el código que dependa de componentes de código abierto.
- **Integre el sistema de seguridad de la aplicación con el sistema de gestión de tareas**: Esto creará automáticamente una lista de tareas con errores que el equipo de seguridad de la información puede ejecutar. Además, proporcionará detalles procesables, incluida la naturaleza del defecto, su gravedad y la mitigación necesaria. Como tal, el equipo de seguridad puede solucionar problemas antes de que terminen en los entornos de desarrollo y producción.

Seguridad del entorno y de los datos

- **Estandarice y automatice el entorno**: los servicios deben tener la menor cantidad de privilegios posible para reducir las conexiones y los accesos no autorizados.
- **Centralice las funciones de control de acceso y de identidad de los usuarios**: el control de acceso estricto y los mecanismos de autenticación centralizados son fundamentales para proteger los microservicios, ya que la autenticación se inicia en varios puntos.
- **Aísle de la red y entre sí aquellos contenedores que ejecutan microservicios**: abarca tanto los datos en tránsito como en reposo, ya que ambos pueden ser objetivos valiosos para los atacantes.
- **Cifre los datos entre las aplicaciones y los servicios**: una plataforma de organización de contenedores con funciones de seguridad integradas disminuye las posibilidades de accesos no autorizados.
- **Incorpore puertas de enlace de API seguras**: las API seguras aumentan el control de los enrutamientos y las autorizaciones. Al disminuir la cantidad de API expuestas, las empresas pueden reducir las superficies de ataque.

Seguridad del proceso de CI/CD

- **Integre análisis de seguridad para los contenedores**: estos deberían ser parte del proceso para agregar contenedores al registro.
- **Automatice las pruebas de seguridad en el proceso de integración continua**: esto incluye ejecutar herramientas de análisis de seguridad estática como parte de las compilaciones, así como examinar las imágenes de contenedores prediseñadas para detectar puntos vulnerables de seguridad conocidos a medida que se incorporan al canal de diseño.
- **Incorpore pruebas automatizadas para las funciones de seguridad al proceso de pruebas de aceptación**: automatice las pruebas de validación de los datos de entrada, así como las funciones de verificación, autenticación y autorización.
- **Automatice las actualizaciones de seguridad, como la aplicación de parches para los puntos vulnerables conocidos**: lleve a cabo este proceso mediante el canal de DevOps. Esto elimina la necesidad de que los administradores inicien sesión en los sistemas de producción mientras crean un registro de cambios documentado y rastreable.
- **Automatice las funciones de gestión de la configuración de los sistemas y los servicios**: de esta manera, podrá cumplir con las políticas de seguridad y eliminar los errores manuales. También se recomienda automatizar los procesos de auditoría y corrección.

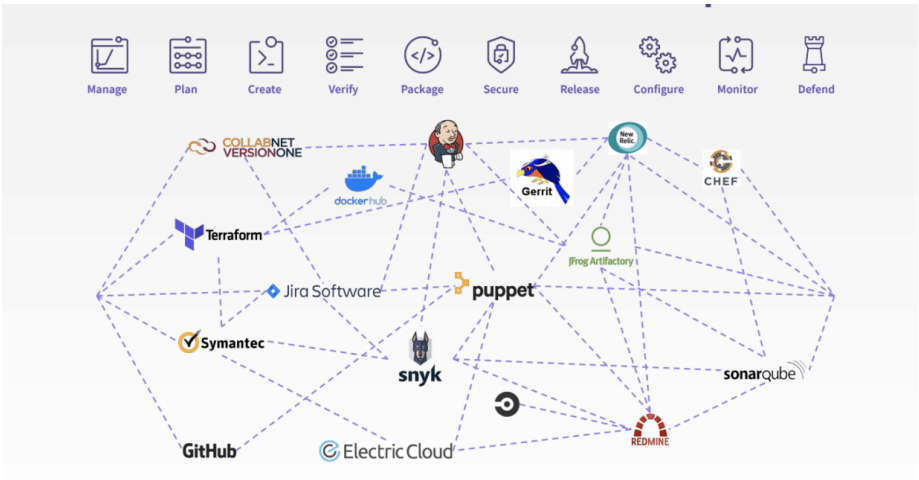
Integración en el ciclo de vida

- Pruebas estáticas de seguridad del software
  - Escanea el código fuente de la aplicación y los archivos binarios para detectar posibles vulnerabilidades.
- Pruebas dinámicas de seguridad del software
  - Escanea la aplicación en tiempo de ejecución para detectar vulnerabilidades como:
    - SQL injection
    - DoS attack
    - XSS
  - Es necesario que la aplicación se este ejecutando en un entorno de laboratorio, donde se realizan las pruebas de afuera hacia adentro sin tener acceso al código fuente. También hacen parte de los tipos de pruebas conocidas como de caja negra.
- Escaneo de dependencias
  - Analiza las dependencias externas del proyecto como npm, ruby gem, composer, entre otras para detectar vulnerabilidades en cada commit de código. Por ejemplo, verificar que no estamos utilizando librerías de código obsoletas o deprecadas.
- Cumplimiento de licencias
  - Busca automáticamente en las dependencias del proyecto las licencias aprobadas o en lista negra de su organización para evitar utilizar software pirata.

¿Es DevSecOps adecuado para tu equipo?

- Los beneficios de DevSecOps son claros: velocidad, eficiencia y colaboración. Pero, ¿cómo saber si es adecuado para tu equipo? Si tu organización se enfrenta alguno de los siguientes desafíos, un enfoque de DevSecOps podría ser una buena medida:
  - Los equipos de desarrollo, seguridad y operaciones están aislados. Si el desarrollo y las operaciones están aislados de los problemas de seguridad, no pueden crear software seguro. Y si los equipos de seguridad no forman parte del proceso de desarrollo, no pueden identificar los riesgos de manera proactiva. DevSecOps reúne a los equipos para mejorar los flujos de trabajo y compartir ideas. Las organizaciones podrían incluso ver una mejora en la moral y la retención de los empleados.
  - Los largos ciclos de desarrollo dificultan satisfacer las demandas de los clientes o de las partes interesadas. Una de las razones de la dificultad podría ser la seguridad. DevSecOps implementa la seguridad en cada paso del ciclo de vida del desarrollo, lo que significa que una seguridad sólida no requiere que todo el proceso se detenga.
  - Se está migrando a la nube (o considerando hacerlo). Mudarse a la nube a menudo significa incorporar nuevos procesos, herramientas y sistemas de desarrollo. Es el momento perfecto para hacer que los procesos sean más rápidos y seguros, y DevSecOps podría hacerlo mucho más fácil.

Herramientas



Referencias

- OWASP Top Ten
  - https://owasp.org/www-project-top-ten/
- OWASP Top 10 CI/CD Security Risks
  - https://owasp.org/www-project-top-10-ci-cd-security-risks/
- GitLab: Global DevSecOps Survey
  - https://about.gitlab.com/developer-survey/
- https://www.devsecops.org/

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 --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

GESTIÓN DEL SOFTWARE, COLABORACIÓN Y EL CONTROL DE VERSIONES

Evolución del software

- Durante el desarrollo
  - El desarrollo del software siempre es progresivo, incluso en el ciclo de vida en cascada
  - El desarrollo evolutivo consiste, precisamente, en una evolución controlada (ciclo de vida espiral, prototipos evolutivos)
- Durante la explotación
  - Durante la fase de mantenimiento se realizan modificaciones sucesivas del producto
- Además de ello, el desarrollo de software se realiza normalmente en equipo, por lo que es necesario integrar varios desarrollos en un solo producto

Control de versiones

- Un control de versiones es un sistema que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.
- Los sistemas de control de versiones son herramientas de software que ayudan a los equipos de software a gestionar los cambios en el código fuente a lo largo del tiempo.
- A medida que los entornos de desarrollo se aceleran, los sistemas de control de versiones ayudan a los equipos de software a trabajar de forma más rápida e inteligente. Son especialmente útiles para los equipos de DevOps, ya que les ayudan a reducir el tiempo de desarrollo y a aumentar las implementaciones exitosas.

Características

- Mecanismo de almacenamiento de los elementos que deba gestionar (ej. archivos de texto, imágenes, documentación...).
- Posibilidad de realizar cambios sobre los elementos almacenados (ej. modificaciones parciales, añadir, borrar, renombrar o mover elementos).
- Registro histórico de las acciones realizadas con cada elemento o conjunto de elementos (normalmente pudiendo volver o extraer un estado anterior del producto).

Ventajas

- Un completo historial de cambios a largo plazo de todos los archivos.
  - Esto quiere decir todos los cambios realizados por muchas personas a lo largo de los años. Los cambios incluyen la creación y la eliminación de los archivos, así como los cambios de sus contenidos. Que, quien y cuando se realizaron los cambios. Facilita volver a una versión anterior.
- Creación de ramas y fusiones.
  - La creación de una "rama" permite mantener múltiples flujos de trabajo independientes los unos de los otros al tiempo que ofrece la facilidad de volver a fusionar ese trabajo, lo que permite que los desarrolladores verifiquen que los cambios de cada rama no entran en conflicto. Facilita el trabajo colaborativo.
- Trazabilidad.
  - Ser capaz de trazar cada cambio que se hace en el software y conectarlo con un software de gestión de proyectos y seguimiento de errores, además de ser capaz de anotar cada cambio con un mensaje que describa el propósito y el objetivo del cambio, no solo ayuda con el análisis de la causa raíz y la recopilación de información.

Conceptos

- Versión
  - "Versión" es la forma particular que adopta un objeto en un contexto dado.
- Revisión
  - Desde el punto de vista de evolución, es la forma particular de un objeto en un instante dado. Se suele denominar "revisión".
- Configuración
  - Una "configuración" es una combinación de versiones particulares de los componentes (que pueden evolucionar individualmente) que forman un sistema consistente.
  - Desde el punto de vista de evolución, es el conjunto de las versiones de los objetos componentes en un instante dado

Conceptos

- Línea base
  - Llamaremos "línea base" a una configuración operativa del sistema software a partir del cual se pueden realizar cambios subsiguientes.
  - La evolución del sistema puede verse como evolución de la línea base
- Cambio
  - Un cambio representa una modificación específica a un archivo bajo control de versiones. La granularidad de la modificación considerada un cambio varía entre diferentes sistemas de control de versiones.
  - Un "cambio" es el paso de una versión de la línea base a la siguiente
  - Puede incluir modificaciones del contenido de algún componente, y/o modificaciones de la estructura del sistema, añadiendo o eliminando componentes

Conceptos

- Branch:
  - Una “ramificación”, bifurcación o variante es una bifurcación de la línea base u otra ramificación que a partir de ese momento evoluciona y se versiona por separado.
  - Las ramificaciones representan una variación espacial, mientras que las revisiones representan una variación temporal.
- Repositorio
  - El repositorio es el lugar en el que se almacenan los datos actualizados e históricos de cambios, a menudo en un servidor.
  - El repositorio permite ahorrar espacio de almacenamiento, evitando guardar por duplicado elementos comunes a varias versiones o configuraciones

Sistemas de Control de Versiones Manuales

- Un método de control de versiones, usado por muchas personas, es copiar los archivos a otro directorio (quizás indicando la fecha y hora en que lo hicieron, si son ingeniosos).
- Este método es muy común porque es muy sencillo, pero también es tremendamente propenso a errores.
- Es fácil olvidar en qué directorio te encuentras y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías.
- Consume mucho espacio al duplicar elementos que no se han modificado entre versiones.

Sistemas de Control de Versiones Centralizados

- El siguiente gran problema con el que se encuentran las personas es que necesitan colaborar con desarrolladores en otros sistemas. Los sistemas de Control de Versiones Centralizados (CVCS por sus siglas en inglés) fueron desarrollados para solucionar varias personas que necesitan colaborar.
- Estos sistemas, como CVS, Subversion y Perforce, tienen un único servidor que contiene todos los archivos versionados y varios clientes que descargan los archivos desde ese lugar central. Este ha sido el estándar para el control de versiones por muchos años.
- Esta configuración ofrece muchas ventajas, especialmente frente a VCS locales, permite trabajar de forma exclusiva: para poder realizar un cambio es necesario comunicar al repositorio el elemento que se desea modificar y el sistema se encargará de impedir que otro usuario pueda modificar dicho elemento, hasta que sea liberado.
- Sin embargo, esta configuración también tiene serias desventajas:
  - Es el punto único de fallo que representa el servidor centralizado.
  - Sin conexión al servidor nadie podrá colaborar o guardar cambios en archivos en los que hayan estado trabajando.

Sistemas de Control de Versiones Distribuidos

- Los sistemas de Control de Versiones Distribuidos o DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes no solo descargan la última copia instantánea de los archivos, sino que se replica completamente el repositorio, existe un repositorio local y otro central.
- De esta manera, si un servidor deja de funcionar y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios disponibles en los clientes puede ser copiado al servidor con el fin de restaurarlo. Cada clon es realmente una copia completa de todos los datos.
- Se puede trabajar de forma local. Permite operaciones más rápidas.
- Cada usuario modifica la copia local y cuando el usuario decide compartir los cambios el sistema automáticamente intenta combinar las diversas modificaciones. El principal problema es la posible aparición de conflictos que deban ser solucionados manualmente o las posibles inconsistencias que surjan al modificar el mismo fichero por varias personas no coordinadas.

https://git-scm.com/
https://git-scm.com/book/es/v2

GIT

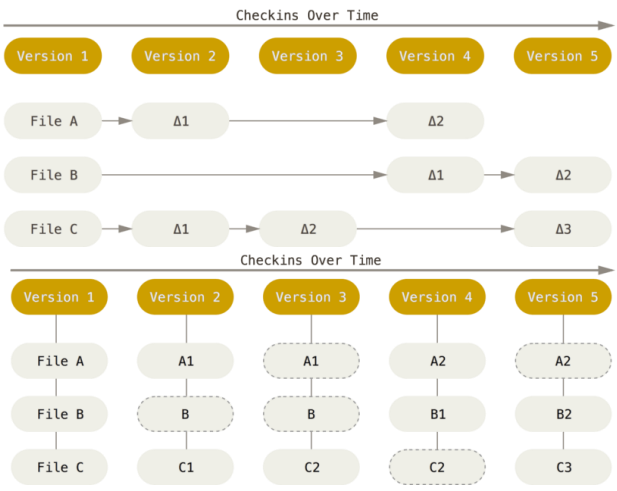
GIT

- Git (pronunciado "guit") es un sistema de control de versiones distribuido de código abierto y gratuito diseñado para manejar todo, desde proyectos pequeños a muy grandes, con velocidad y eficiencia.
- Git es, con diferencia, el sistema de control de versiones moderno más utilizado del mundo. Git es un proyecto de código abierto maduro y con un mantenimiento activo que desarrolló originalmente Linus Torvalds, el famoso creador del kernel del sistema operativo Linux, en 2005.
- Git es fácil de aprender y ocupa poco espacio con un rendimiento increíblemente rápido. Supera a las herramientas SCM como Subversion, CVS, Perforce y ClearCase con características como bifurcaciones locales económicas, áreas de preparación convenientes y múltiples flujos de trabajo. Funciona a la perfección en una amplia variedad de sistemas operativos e IDE (entornos de desarrollo integrados).
- No confundir con GitHub, que es una forja (plataforma de desarrollo colaborativo) para alojar proyectos utilizando el sistema de control de versiones Git, siendo la plataforma más importante de colaboración para proyectos Open Source.
- Instalación:
  - https://git-scm.com/

Fundamentos de Git

- La principal diferencia entre Git y cualquier otro VCS es la forma en la que manejan sus datos. Conceptualmente, la mayoría de los otros sistemas almacenan la información como una lista de cambios en los archivos, manejan la información que almacenan como un conjunto de archivos y las modificaciones hechas a cada uno de ellos a través del tiempo.
- Git maneja los datos como un conjunto de copias instantáneas de un sistema de archivos miniatura. Cada vez que se confirma un cambio o guarda el estado del proyecto en Git, él básicamente toma una foto del aspecto de todos los archivos en ese momento y guarda una referencia a esa copia instantánea. Para ser eficiente, si los archivos no se han modificado Git no almacena el archivo de nuevo, sino un enlace al archivo anterior idéntico que ya tiene almacenado. Git maneja los datos como una secuencia de copias instantáneas.
- La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para funcionar. Por lo general no se necesita información de ningún otro equipo de tu red.
- Todo en Git es verificado mediante una suma de comprobación (checksum) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo detecte.
- Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil hacer algo que no se pueda enmendar.

Instantáneas, no diferencias



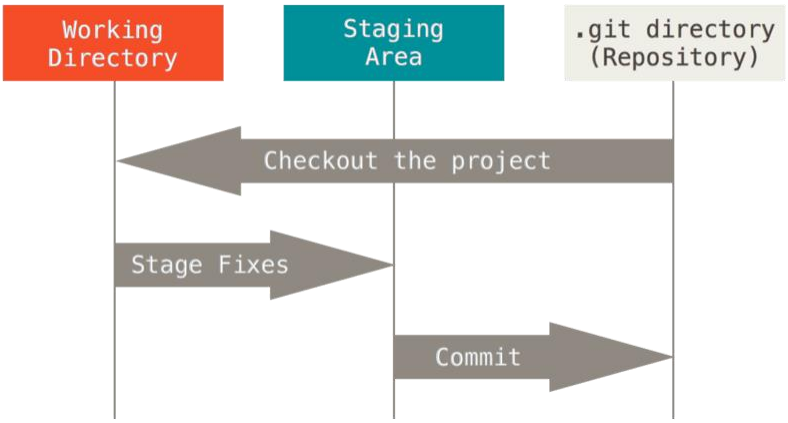
Estados

- Git tiene tres estados principales en los que se pueden encontrar tus archivos:
  - Modificado (modified): significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos.
  - Preparado (staged): significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.
  - Confirmado (committed): significa que los datos están almacenados de manera segura en tu base de datos local.
- Esto nos lleva a las tres secciones principales de un proyecto de Git:
  - El directorio de Git (git directory) es donde se almacenan los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde un repositorio central.
  - El área de preparación (staging area) es un archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina índice ("index"), pero se está convirtiendo en estándar el referirse a ella como el área de preparación.
  - El directorio de trabajo (working directory) es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar.

Flujo de trabajo

- Git plantea una gran libertad en la forma de trabajar en torno a un proyecto. Sin embargo, para coordinar el trabajo de un grupo de personas en torno a un proyecto es necesario acordar como se va a trabajar con Git. A estos acuerdos se les llama flujo de trabajo. Un flujo de trabajo de Git es una fórmula o una recomendación acerca del uso de Git para realizar trabajo de forma uniforme y productiva. Los flujos de trabajo más populares son git-flow, GitHub-flow y One Flow.
- El flujo de trabajo básico en Git es algo así:
  1. Modificas una serie de archivos en tu directorio de trabajo.
  2. Preparas los archivos, añadiéndolos a tu área de preparación.
  3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación y almacena esa copia instantánea de manera permanente en tu directorio de Git.
  4. Periódicamente, se sincronizan el repositorio local con el repositorio central para que los cambios estén disponibles para el resto de los colaboradores, actuando también como copia de seguridad del repositorio local.
- Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (committed). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada (staged). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (modified).

Flujo de trabajo



Flujo de trabajo

- Git-Flow (Creado en 2010 por Vincent Driessen)
  - Es el flujo de trabajo más conocido. Está pensado para aquellos proyectos que tienen entregables y ciclos de desarrollo bien definidos. Está basado en dos grandes ramas con infinito tiempo de vida (ramas master y develop) y varias ramas de apoyo, unas orientadas al desarrollo de nuevas funcionalidades (ramas feature-*), otras al arreglo de errores (hotfix-*) y otras orientadas a la preparación de nuevas versiones de producción (ramas release-*).
- GitHub-Flow (Creado en 2011 por el equipo de GitHub)
  - Es la forma de trabajo sugerida por las funcionalidades propias de GitHub. Está centrado en un modelo de desarrollo iterativo y de despliegue constante. Está basado en cuatro principios:
    - Todo lo que está en la rama master está listo para ser puesto en producción
    - Para trabajar en algo nuevo, debes crear una nueva rama a partir de la rama master con un nombre descriptivo. El trabajo se irá integrando sobre esa rama en local y regularmente también a esa rama en el servidor
    - Cuando se necesite ayuda o información o cuando creemos que la rama está lista para integrarla en la rama master, se debe abrir una pull request (solicitud de integración de cambios).
    - Alguien debe revisar y visar los cambios para fusionarlos con la rama master
    - Los cambios integrados se pueden poner en producción.
  - GitHub intenta simplificar la gestión de ramas, trabajando directamente sobre la rama master y generando e integrando las distintas features directamente a esta rama.
- One Flow (Creado en 2015 por Adam Ruka)
  - En él cada nueva versión de producción está basada en la versión previa de producción. La mayor diferencia con Git Flow es que no tiene rama de desarrollo.

Terminología

- commit: un objeto Git, una instantánea de todo su repositorio comprimido en un SHA
- branch (rama): un puntero móvil ligero a una confirmación
- clone: una versión local de un repositorio, incluidas todas las confirmaciones y ramas
- remote: un repositorio común en un servidor central (como GitHub) que todos los miembros del equipo usan para intercambiar sus cambios
- fork: una copia de un repositorio del servidor central propiedad de un usuario diferente
- pull request (solicitud de extracción): un lugar para comparar y discutir las diferencias introducidas en una rama con revisiones, comentarios, pruebas integradas y más
- HEAD: que representa el directorio de trabajo actual, el puntero HEAD se puede mover a diferentes ramas, etiquetas o confirmaciones cuando se usa git checkout

Crear un repositorio Git

- Se puede obtener un proyecto Git de dos maneras: La primera es tomar un proyecto o directorio existente e importarlo en Git. La segunda es clonar un repositorio existente en Git desde otro servidor.
- Inicializando un repositorio en un directorio existente (esto crea un subdirectorio oculto llamado .git que contiene todos los archivos necesarios del repositorio local):
  - git init
- Añadiendo los ficheros al repositorio local:
  - git add .
- Para confirmar los cambios:
  - git commit -m 'initial project version'
- Para vincular el repositorio local con el repositorio central y sincronizar los cambios:
  - git branch -M main
  - git remote add origin https://github.com/myuser/myrepository.git
  - git push -u origin main
- Clonando un repositorio existente:
  - git clone https://github.com/myuser/myrepository localdir

Trabajar con un repositorio Git

- Enumera todos los archivos nuevos o modificados de los cuales se van a guardar cambios
  - git status
- Muestra las diferencias entre archivos que no se han enviado aún al área de espera
  - git diff
- Registra los cambios del archivo permanentemente en el historial de versiones
  - git commit -m"[descriptive message]"
- Sube todos los commits de la rama local al repositorio central
  - git push
- Descarga los cambios del repositorio central a la rama local
  - git pull

Ramificaciones

- Git permite dividir la rama principal de desarrollo (master) y a partir de ahí continuar trabajando de forma independiente: solución de problemas, nueva versión, experimentación … Los cambios en una rama no afectan a las otras ramas.
- Crear una Rama Nueva:
  - git branch hotfix
- Para saltar de una rama a otra se utiliza el comando git checkout.
  - git checkout hotfix
- Los cambios, confirmaciones y otras se realizan sobre la rama actual. Una vez terminados los trabajos en la rama se pueden fusionar con la rama principal (si hay cambios en la principal se pueden producir conflictos que habrá que solucionar):
  - git checkout master
  - git merge hotfix
- Es importante borrar la rama fusionada que ya vamos a necesitar
  - git branch -d hotfix

Solicitud de incorporación de cambios

- Las pull requests, solicitud de incorporación de cambios, son una funcionalidad que facilita la colaboración entre desarrolladores.
- En su forma más sencilla, las solicitudes de incorporación de cambios son un mecanismo para que los desarrolladores notifiquen a los miembros de su equipo que han terminado una función. Una vez la rama de función está lista, el desarrollador realiza la solicitud de incorporación de cambios mediante el repositorio central. Así, todas las personas involucradas saben que deben revisar el código y fusionarlo con la rama principal.
- Pero la solicitud de incorporación de cambios es mucho más que una notificación: es un foro especializado para debatir sobre una función propuesta. Si hay algún problema con los cambios, los miembros del equipo pueden publicar feedback en las solicitudes de incorporación de cambios e incluso modificar la función al enviar confirmaciones de seguimiento. El seguimiento de toda esta actividad se realiza directamente desde la solicitud de incorporación de cambios.
- Cuando realizas una pull request, lo que haces es solicitar que otro desarrollador (el mantenedor del proyecto) incorpore (o haga un pull) una rama de tu repositorio (fork) al suyo.

Merge Conflicts

- Un merge conflict se produce cuando hay discrepancias al sincronizar versiones que entran en conflicto y Git no puede determinar automáticamente qué es correcto:
  - La versión original modificada es distinta de la versión actual.
  - Difieren las versiones en diferentes ramas.
  - Se va a sobre escribir un cambio local pendiente.
- Podemos sacar más información ejecutando el comando git status.
- Git agrega algunas líneas adicionales en el archivo para marcar el conflicto:

```
<<<<<<< HEAD
=======
>>>>>>> commit_name
```

- La línea ======= es el "centro" del conflicto. Todo el contenido entre el centro y la línea <<<<<<< HEAD es contenido que existe en la rama principal actual a la que apunta la referencia HEAD. Por el contrario, todo el contenido entre el centro y >>>>>>> commit_name es contenido que está presente en nuestra rama de fusión.
- La forma más directa de resolver un conflicto de fusión manualmente es editar el archivo conflictivo, eliminar las marcas, dejando una de las dos secciones, las dos o ninguna. Las utilidades de los entornos pueden ayudar a resolver los conflictos.
- Una vez resuelto el conflicto o conflictos, con git add se preparar el nuevo contenido fusionado y con git commit se confirma la solución.

Ignorar Archivos

- A veces, tendrás algún tipo de archivo que no quieres que Git añada automáticamente o más aun, que ni siquiera quieras que aparezca como no rastreado. Este suele ser el caso de archivos generados automáticamente como trazas o archivos creados por el sistema de compilación. En estos casos, se puede crear un archivo de texto llamado .gitignore que liste patrones de los archivos a excluir.

  reame.md
  /node_modules
  doc/**/*.md
- Las reglas usan patrones globales y puedes incluir en el archivo .gitignore las siguientes:
  - Ignorar las líneas en blanco y aquellas que comiencen con #.
  - Emplear patrones glob estándar que se aplicarán recursivamente a todo el directorio del repositorio local.
  - Los patrones pueden comenzar en barra (/) para evitar recursividad.
  - Los patrones pueden terminar en barra (/) para especificar un directorio.
  - Los patrones pueden negarse al añadir al principio el signo de exclamación (!).

Comandos

- Configurar: Configura la información del usuario para todos los respositorios locales
  $ git config --global user.name "[name]"
  Establece el nombre que estará asociado a tus commits
  $ git config --global user.email "[email address]"
  Establece el e-mail que estará asociado a sus commits
- Crear repositorios: Inicializa un nuevo repositorio u obtiene uno de una URL existente
  $ git init [project-name]
  Crea un nuevo repositorio local con el nombre especificado
  $ git remote add origin [url]
  Especifica el repositorio remoto para su repositorio local
  $ git clone [url]
  Descarga un proyecto y toda su historial de versiones
- Suprimir el seguimiento de cambios: Un archivo de texto llamado .gitignore suprime la creación accidental de versiones para archivos y rutas que concuerdan con los patrones especificados
  $ git ls-files --others --ignored --exclude-standard
  Enumera todos los archivos ignorados en este proyecto

Comandos

- Efectuar cambios
  - $ git status
    - Enumera todos los archivos nuevos o modificados de los cuales se van a guardar cambios
  - $ git diff
    - Muestra las diferencias entre archivos que no se han enviado aún al área de espera
  - $ git add [file]
    - Guarda el estado del archivo en preparación para realizar un commit
  - $ git diff --staged
    - Muestra las diferencias del archivo entre el área de espera y la última versión del archivo
  - $ git reset [file]
    - Mueve el archivo del área de espera, pero preserva su contenido
  - $ git commit -m"[descriptive message]"
    - Registra los cambios del archivo permanentemente en el historial de versiones
  - $ git rm [file]
    - Borra el archivo del directorio activo y lo pone en el área de espera en un estado de eliminación
  - $ git rm --cached [file]
    - Retira el archivo del historial de control de versiones, pero preserva el archivo a nivel local
  - $ git mv [file-original] [file-renamed]
    - Cambia el nombre del archivo y lo prepara para ser guardado

Comandos

- Sincronizar cambios: Registrar un marcador para un repositorio e intercambiar historial de versiones
  - $ git fetch [bookmark]
    - Descarga todo el historial del marcador del repositorio
  - $ git merge [bookmark]/[branch]
    - Combina la rama del marcador con la rama local actual
  - $ git push [alias] [branch]
    - Sube todos los commits de la rama local al repositorio central
  - $ git pull
    - Descarga el historial del marcador e incorpora cambios
- Branches (cambios grupales): Nombra una serie de commits y combina esfuerzos ya completados
  - $ git branch
    - Enumera todas las ramas en el repositorio actual
  - $ git branch [branch-name]
    - Crea una nueva rama
  - $ git checkout [branch-name]
    - Cambia a la rama especificada y actualiza el directorio activo
  - $ git merge [branch-name]
    - Combina el historial de la rama especificada con la rama actual
  - $ git branch -d [branch-name]
    - Borra la rama especificada

Comandos

- Repasar historial: Navega e inspecciona la evolución de los archivos de proyecto
  - $ git log
    - Enumera el historial de versiones para la rama actual
  - $ git log --follow [file]
    - Enumera el historial de versiones para el archivo, incluidos los cambios de nombre
  - $ git diff [first-branch]...[second-branch]
    - Muestra las diferencias de contenido entre dos ramas
  - $ git show [commit]
    - Produce metadatos y cambios de contenido del commit especificado
- Rehacer commits: Borra errores y elabora un historial de reemplazo
  - $ git reset [commit]
    - Deshace todos los commits después de [commit], preservando los cambios localmente
  - $ git reset --hard [commit]
    - Desecha todo el historial y regresa al commit especificado

Comandos

- Guardar fragmentos: Almacena y restaura cambios incompletos
  - $ git stash
    - Almacena temporalmente todos los archivos modificados de los cuales se tiene al menos una versión guardada
  - $ git stash pop
    - Restaura los archivos guardados más recientemente
  - $ git stash list
    - Enumera todos los grupos de cambios que están guardados temporalmente
  - $ git stash drop
    - Elimina el grupo de cambios más reciente que se encuentra guardado temporalmente

GitHub

- echo "# Mi repositorio" >> README.md
- git init
- git add .
- git commit -m "initial commit"
- git remote add origin https://github.com/myuser/myrepository.git
- git branch -M main
- git push -u origin main

- https://github.com/github/gitignore

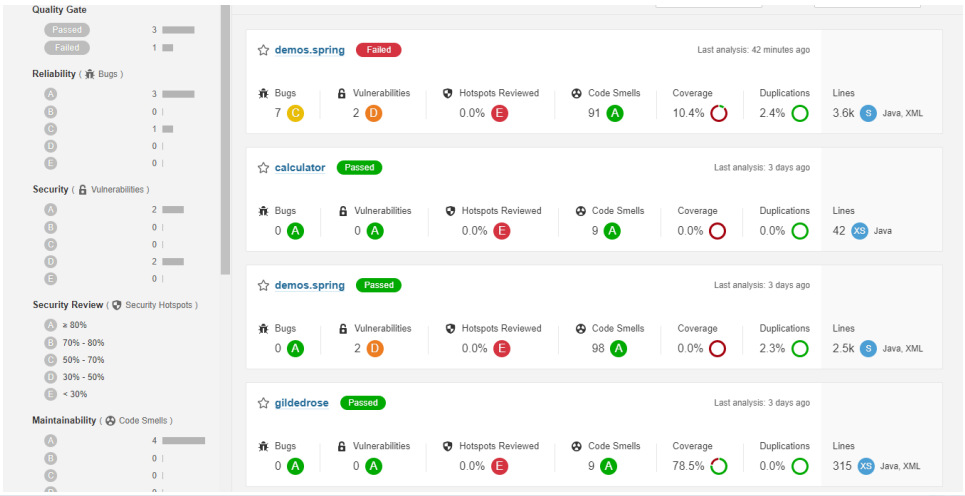
http://www.sonarqube.org/

SONARQUBE

Introducción

- SonarQube (conocido anteriormente como Sonar) es una herramienta de revisión automática de código para detectar errores, vulnerabilidades y malos olores en su código. Puede integrarse con su flujo de trabajo existente para permitir la inspección continua de código en las ramas de su proyecto y las solicitudes de incorporación de cambios.
- SonarQube es una plataforma para la revisión y evaluación del código fuente. Es una herramienta esencial para la fase de pruebas y auditoría de código dentro del ciclo de desarrollo de una aplicación y se considera perfecta para guiar a los equipos de desarrollo durante las revisiones de código.
- Es open source y realiza el análisis estático de código fuente integrando las mejores herramientas de medición de la calidad de código como Checkstyle, PMD o FindBugs, para obtener métricas que pueden ayudar a mejorar la calidad del código de un programa.
- Informa sobre código duplicado, estándares de codificación, pruebas unitarias, cobertura de código, complejidad ciclomática, posible errores, comentarios y diseño del software.
- Aunque pensado para Java, acepta mas de 20 lenguajes mediante extensiones. Se integra con Maven, Ant y herramientas de integración continua como Atlassian, Jenkins y Hudson.

Sonar



Proceso

1. Los desarrolladores desarrollan y combinan código en un IDE (preferiblemente usando SonarLint para recibir comentarios inmediatos en el editor) y registran su código en su plataforma DevOps.
2. La herramienta de integración continua (CI) de una organización verifica, crea y ejecuta pruebas unitarias, y un escáner SonarQube integrado analiza los resultados.
3. El escáner publica los resultados en el servidor de SonarQube, que proporciona comentarios a los desarrolladores a través de la interfaz de SonarQube, correo electrónico, notificaciones en el IDE (a través de SonarLint) y decoración en las solicitudes de extracción o combinación (cuando se usa Developer Edition y superior).

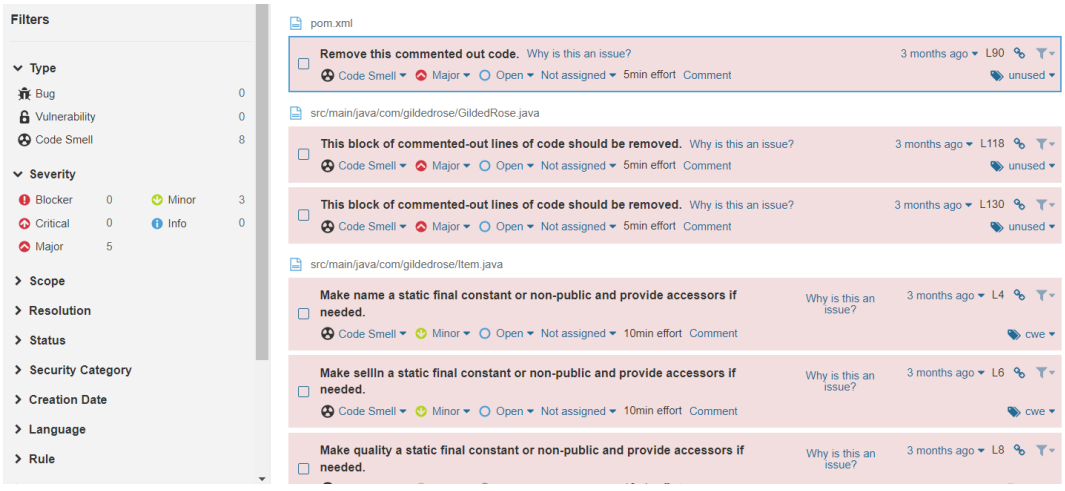
Beneficios

- Alerta de manera automática a los desarrolladores de los errores de código para corregirlos previamente a la implementación en producción.
- No sólo muestra los errores, también las reglas de codificación, la cobertura de las pruebas, las duplicaciones, la complejidad y la arquitectura, plasmando todos estos datos en paneles de control detallados.
- Ayuda al equipo a mejorar en sus habilidades como programadores al facilitar un seguimiento de los problemas de calidad.
- Permite la creación de paneles y filtros personalizables para centrarse en áreas clave y entregar productos de calidad a tiempo.
- Favorece la productividad al reducir la complejidad del código acortando tiempos y costes adicionales al evitar cambiar el código constantemente.

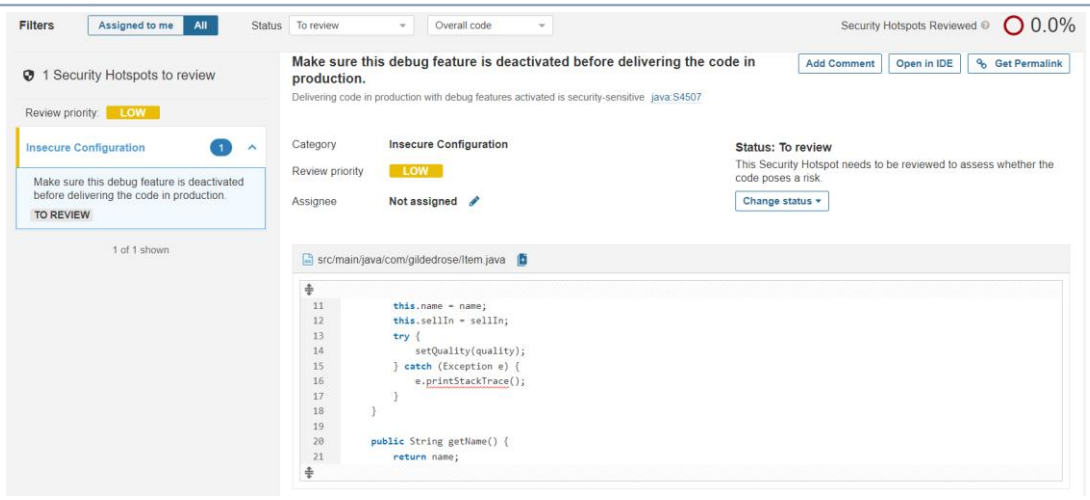
Herramientas

- SonarLint: es un producto complementario que funciona en su editor y brinda comentarios inmediatos para que pueda detectar y solucionar problemas antes de que lleguen al repositorio.
- Quality Gate: permite saber si su proyecto está listo para la producción.
- Clean as You Code: es un enfoque de la calidad del código que elimina muchos de los desafíos que conllevan los enfoques tradicionales. Como desarrollador, se enfoca en mantener altos estándares y asumir la responsabilidad específicamente en el Nuevo Código en el que está trabajando.
- Issues: SonarQube plantea problemas cada vez que una parte de su código infringe una regla de codificación, ya sea un error que romperá su código (bug), un punto en su código abierto a ataques (vulnerabilidad) o un problema de mantenimiento (código apestoso).
- Security Hotspots: Puntos de acceso de seguridad destaca piezas de código sensibles a la seguridad que deben revisarse. Tras la revisión, descubrirá que no hay ninguna amenaza o que debe aplicar una solución para proteger el código.

Problemas



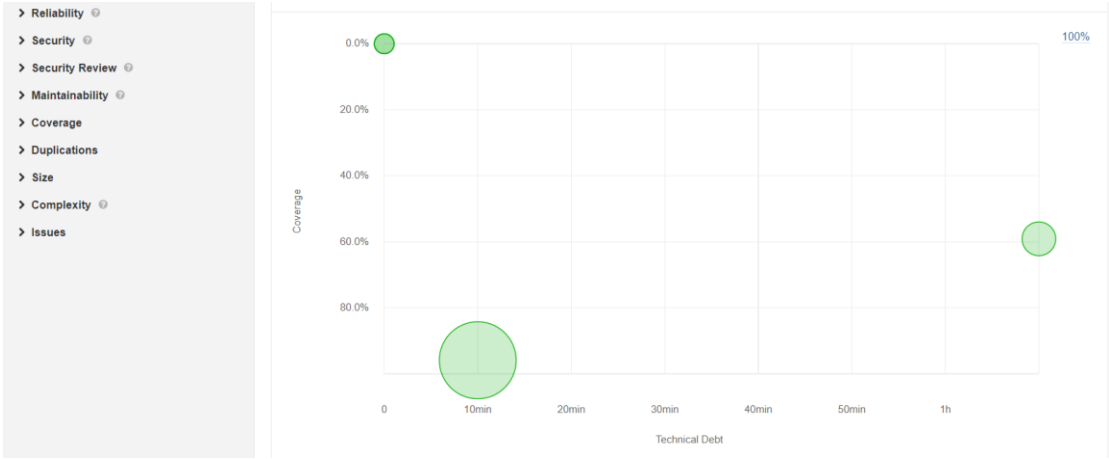
Puntos de acceso de seguridad



Métricas principales

- **Complejidad**: Refleja la Complejidad Ciclomática calculada en base al número de caminos a través del código normalmente observado a nivel de métodos o funciones individuales.
- **Duplicados**: Nos indica el número de bloques de líneas duplicados. Ayuda a evitar resultados distintos en operaciones iguales.
- **Evidencias**: Son los fragmentos nuevos de código de un proyecto que detecta que incumplen con alguna de las reglas establecidas.
- **Mantenibilidad**: Se refiere al recuento total de problemas de Code Smell.
- **Umbrales de calidad**: Define los requisitos del proyecto antes de ser lanzado a producción, como, por ejemplo, que no deben haber evidencias bloqueantes o la cobertura de código sobre el código nuevo debe ser mayor que el 80%.
- **Tamaño**: Permiten hacerse una idea del volumen del proyecto en términos generales.
- **Pruebas**: Son una forma de comprobar el correcto funcionamiento de una unidad de código y de su integración.

Métricas



Análisis

- SonarQube puede analizar mas de 20 lenguajes diferentes según la edición. El resultado de este análisis serán métricas y problemas de calidad (casos en los que se rompieron las reglas de codificación). Sin embargo, lo que se analiza variará según el lenguaje:
  - En todos los lenguajes, los datos de "culpa" se importarán automáticamente de los proveedores de SCM admitidos. Git y SVN son compatibles automáticamente.
  - En todos los lenguajes se realiza un análisis estático del código fuente (archivos Java, programas COBOL, etc.)
  - Se puede realizar un análisis estático del código compilado para ciertos lenguajes (archivos .class en Java, archivos .dll en C#, etc.)
- Durante el análisis, se solicitan datos del servidor, se analizan los archivos proporcionados para el análisis enfrentándolos a las reglas y los datos resultantes se envían de vuelta al servidor al final en forma de informe, que luego se analiza de forma asíncrona en el lado del servidor.

Reglas

- Una regla es un estándar o práctica de codificación que debe seguirse. El incumplimiento de las reglas de codificación conduce a errores, vulnerabilidades, puntos críticos de seguridad y código apestoso. Las reglas pueden comprobar la calidad de los archivos de código o las pruebas unitarias.
- Cada lenguaje de programación dispone de sus propias reglas, siendo diferentes en cantidad y tipo.
- Las reglas se clasifican en:
  - Bug: un punto de fallo real o potencial en su software
  - Code Smell: un problema relacionado con la mantenibilidad en el código.
  - Vulnerability: un punto débil que puede convertirse en un agujero de seguridad que puede usarse para atacar su software.
  - Security Hotspot: un problema que es un agujero de seguridad.
- Para Bugs y Code Smells and Bugs, no se esperan falsos positivos. Al menos este es el objetivo para que los desarrolladores no tengan que preguntarse si se requiere una solución. Para las vulnerabilidades, el objetivo es que más del 80 % de los problemas sean verdaderos positivos. Las reglas de Security Hotspot llaman la atención sobre el código que es sensible a la seguridad. Se espera que más del 80% de los problemas se resuelvan rápidamente como "Revisados" después de la revisión por parte de un desarrollador.

Control de calidad

- Los perfiles de calidad (Quality Profile) son un componente central de SonarQube donde se definen conjuntos de reglas que, cuando se violan, generan problemas en la base de código (ejemplo: los métodos no deben tener una complejidad cognitiva superior a 15). Cada lenguaje individual tiene su propio perfil de calidad predeterminado y los proyectos que no están asignados explícitamente a perfiles de calidad específicos se analizan utilizando los perfiles de calidad predeterminados.
- Un umbral de calidad (Quality Gate) es un conjunto de condiciones booleanas basadas en métricas. Ayudan a saber inmediatamente si los proyectos están listos para la producción. Idealmente, todos los proyectos utilizarán la misma barrera de calidad. El estado de Quality Gate de cada proyecto se muestra de forma destacada en la página de inicio.

Scanner

- Una vez que se haya instalado la plataforma SonarQube, estará listo para instalar un escáner y comenzar a crear proyectos. Para ello, se debe instalar y configurar el escáner más adecuado pare el proyecto:
  - Gradle - SonarScanner para Gradle
  - Ant - SonarScanner para Ant
  - Maven: use el SonarScanner para Maven
  - .NET - SonarScanner para .NET
  - Jenkins - SonarScanner para Jenkins
  - Azure DevOps - Extensión de SonarQube para Azure DevOps
  - Cualquier otra cosa (CLI) - SonarScanner

SonarScanner para Maven

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.10.0</version>
        </plugin>
        <plugin>
            <groupId>org.sonarsource.scanner.maven</groupId>
            <artifactId>sonar-maven-plugin</artifactId>
            <version>3.9.1.2184</version>
        </plugin>
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>0.8.7</version>
        </plugin>
    </plugins>
</build>
```

SonarScanner para Maven

```
<profiles>
    <profile>
        <id>coverage</id><activation><activeByDefault>true</activeByDefault></activation>
        <build>
            <plugins>
                <plugin>
                    <groupId>org.jacoco</groupId>
                    <artifactId>jacoco-maven-plugin</artifactId>
                    <executions>
                        <execution>
                            <id>prepare-agent</id><goals><goal>prepare-agent</goal></goals>
                        </execution>
                        <execution>
                            <id>report</id><goals><goal>report</goal></goals>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </build>
    </profile>
</profiles>
```


Analizando

- Analizar un proyecto Maven consiste en ejecutar un objetivo Maven: sonar:sonar desde el directorio que contiene el proyecto principal pom.xml.
- Se debe pasar un token de autenticación usando la propiedad sonar.login en la línea de comando.
  - mvn sonar:sonar -Dsonar.projectKey=Microservicios -Dsonar.host.url=http://localhost:9000 -Dsonar.login=1eeaaf436541...
- Para obtener información de cobertura, deberá generar el informe de cobertura antes del análisis.
  - mvn clean verify sonar:sonar

JavaScript

- El SonarScanner es el escáner que se debe usar cuando no hay un escáner específico para su sistema de compilación. Para descargar:
  - https://docs.sonarqube.org/8.9/analysis/scan/sonarscanner/
- Crear un archivo de configuración en el directorio raíz del proyecto llamado sonar-project.properties

```
sonar.projectKey=<projectKey>
sonar.projectName=<projectName>
sonar.projectVersion=1.0
sonar.language=js
sonar.sources=src
sonar.sourceEncoding=UTF-8
#sonar.host.url=http://localhost:9000
```

- Para ejecutar el scaner:
  - sonar-scanner.bat -D"sonar.projectKey=Web4Testing" -D"sonar.sources=." -D"sonar.host.url=http://localhost:9000" -D"sonar.login=9ea…51a3a"

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.

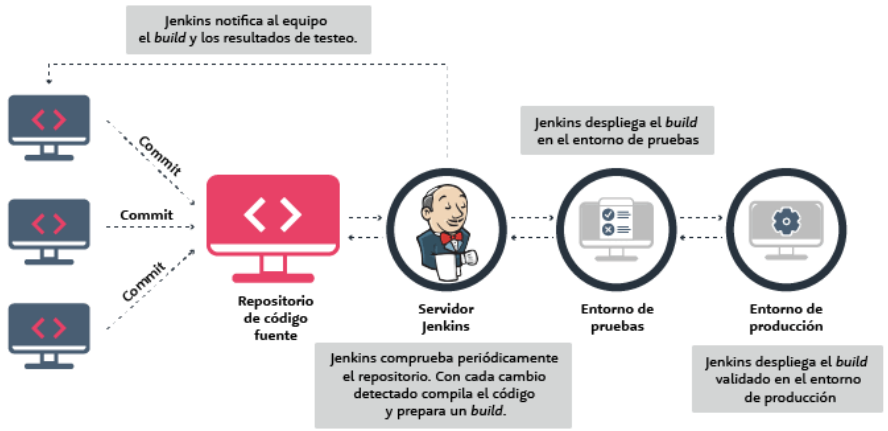
Qué se puede hacer

- 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

- 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

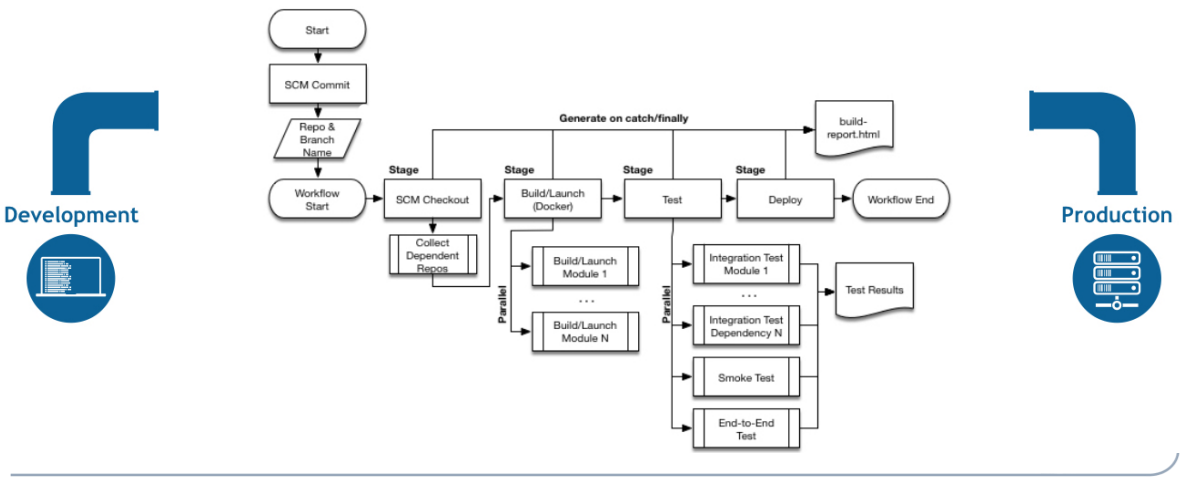
Jenkins DSL

```
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

https://www.jenkins.io/doc/pipeline/steps/

- 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.

Fingerprints

- Cuando se tienen proyectos interdependientes en Jenkins que generan artefactos (archivos), suele resultar difícil hacer un seguimiento de qué versión del artefacto utiliza cada versión de las dependencias de los proyectos. Jenkins admite la identificación de archivos mediante huellas digitales (fingerprinting) para realizar un seguimiento de las dependencias.
- La huella digital de un archivo es simplemente una hash, una suma de comprobación MD5. Jenkins mantiene una base de datos de sumas de comprobación MD5 y, para cada hash, Jenkins registra qué compilaciones de qué proyectos la usaron. Esta base de datos se actualiza cada vez que se ejecuta una compilación y se toman huellas digitales de los nuevos archivos.
- Para que esto funcione, todos los proyectos relevantes deben configurarse para registrar huellas digitales de los archivos jar. Dado que registrar huellas dactilares es una operación barata, lo más sencillo es registrar a ciegas todas las huellas dactilares de los archivos jar:
  - que produce un proyecto
  - de los que depende un proyecto

Fingerprints

- Para registrar huellas dactilares de un archivo o conjunto de archivos, en la configuración del proyecto, en la sección **Acciones para ejecutar después** se añade la acción *Almacenar firma de ficheros para poder hacer seguimiento*.
- Los campos de configuración de la acción posterior a la compilación le brindan una opción de patrón para que coincida con los archivos que desea identificar, así como opciones adicionales.

  target/**/*.jar
- Para registrar huellas dactilares en un pipeline:

  fingerprint 'target/*.jar'
- El uso del disco se ve afectado más por la cantidad de archivos a los que se les ha asignado una huella digital, que por el tamaño de los archivos o la cantidad de compilaciones que se utilizan. Por lo tanto, a menos que se tenga mucho espacio en el disco, no conviene asignar una huella digital **/*.

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 {
```


Groovy Script

- Groovy Script es un lenguaje de programación y scripts orientado a objetos basado en Java que se ejecuta en la máquina virtual Java. Groovy tiene una sintaxis muy similar a la del lenguaje Java. Groovy Script permite utilizar cierres, scripts multilínea, DSL y expresiones incrustadas en cadenas. No reemplaza a Java, sino que lo mejora con algunas funciones dinámicas.
- La definición de un pipeline Jenkins en realidad utiliza un DSL de Groovy.
- El paso script Groovy se puede utilizar en el archivo Jenkins para crear pipelines más complejos.

```
pipeline {
  agent any
  parameters {
    booleanParam(name: 'skip_test', defaultValue: false, description: 'Set to true to skip the test stage')
  }
  stages {
    stage('Testing') {
      steps {
        script {
          def test=params.skip_test
          if (test == null || !test) {
            echo 'starting test  ...'
          } else {
            echo 'skipping test ...'
          }
        }
      } } }}
```

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)