Pipeline en Jenkins

Un Pipeline Jenkins es la clave para el despliegue continuo en esta herramienta open source. La ventajas favorita sobre DevOps es el hecho de realizar despliegues de forma automática (siempre y cuando supere todas las pruebas).

Y para poder realizar esos despliegues, se debe configurar un pipeline, que definirá el ciclo de vida de tu aplicación desde que descargas el código de repositorio, hasta que despliegas en el entorno que necesites.

Índice

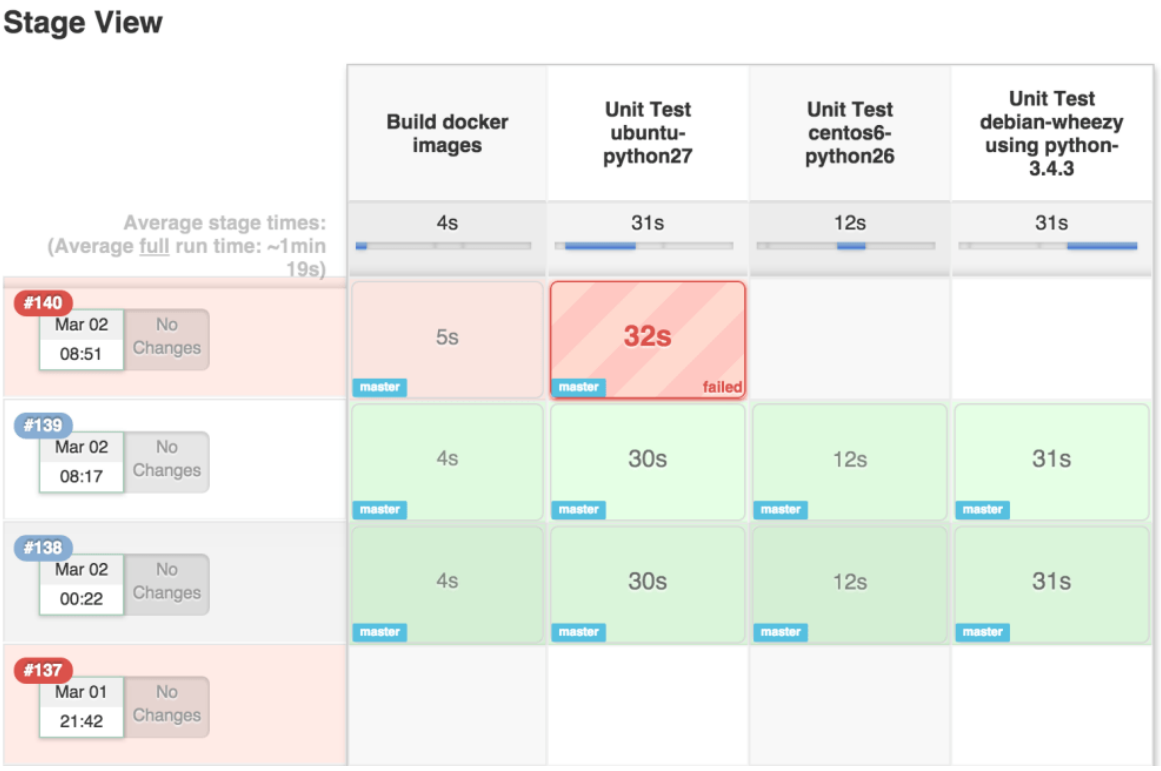
Qué es un Pipeline Jenkins ?

Si consideramos un pipeline como un conjunto de instrucciones del proceso que sigue una aplicación desde el repositorio de control de versiones hasta que llega a los usuarios, podríamos decir que un pipeline **Jenkins** es un conjunto de plugins que soporta la implementación e integración de pipelines de despliegue continuo en **Jenkins**.

Cada cambio en cualquier 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 despliegues necesarios.

Un pipeline Jenkins provee un gran conjunto de herramientas para dar forma con código a un pipeline de entrega de la aplicación.



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 el pipeline de despliegue continua como parte de la aplicación para que sea versionado y revisado como cualquier otra parte del código.

Para crear el código de ese Jenkinsfile, se hace uso de una sintaxis propia que puedes consultar aquí.

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.

Aunque la sintaxis para definir un pipeline Jenkins es igual, tanto a través de la interfaz web como con un Jenkinsfile, generalmente se considera como una mejor práctica realizarlo de la segunda forma y subirlo al repositorio.

Tipos de sintaxis para un pipeline Jenkins

Un Jenkinsfile puede ser escrito usando dos tipos de sintaxis: **Declarative** (declarativa) o **Scripted** (guionizada).

Ambos tipos se construyen de forma diferente.

Un pipeline declarative es una característica reciente en un pipeline Jenkins que:

- Provee unas características sintácticas más completas que sobre las de tipo scripted.
- Está hecho para que tanto su escritura como lectura sean más sencillas.

Muchos de los componentes individuales de un Jenkinsfile, llamados pasos (steps), a menudo, son iguales en ambas sintaxis.

¿Por qué utilizar un Pipeline Jenkins?

Jenkins es una herramienta que soporta patrones de automatización.

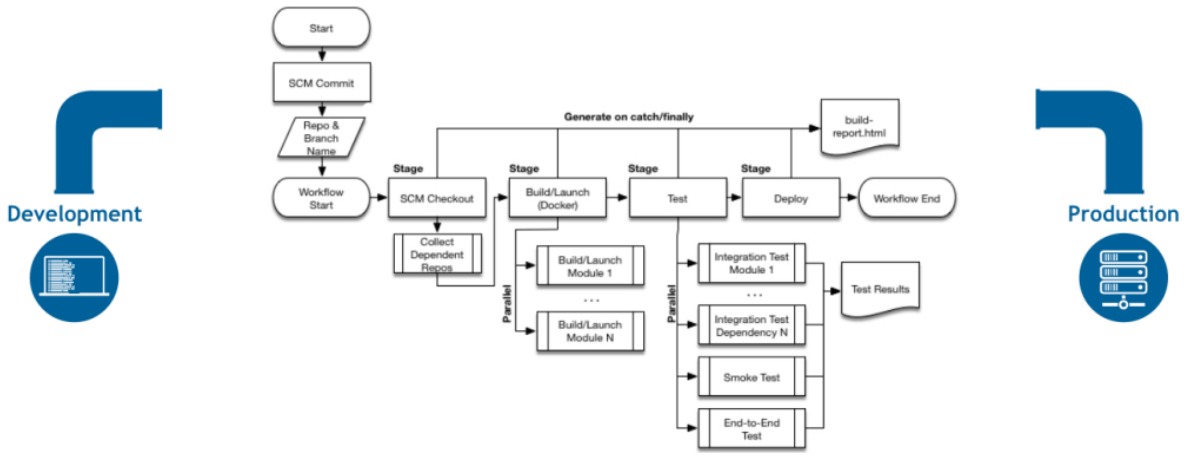
Un Pipeline Jenkins añade un potente conjunto de herramientas de automatización que, soporta casos de uso que van desde una simple integración continua (CI), hasta exhaustivos pipelines de entrega continua (CD).

Dando forma a una serie de tareas relacionadas, los usuarios pueden aprovechar muchas de las características de un pipeline, cómo por ejemplo:

- Código: los Pipelines Jenkins se implementan en el código y suelen incorporarse al proyecto a través del sistema de control de versiones, facilitando a los equipos la habilidad para editar y revisar a través de su pipeline.
- Durable: Los pipelines, perduran tanto a los reinicios planificados como a los no planificados del servidor de Jenkins.
- Pausable: Opcionalmente, pueden parar y esperar la intervención o la aprobación de una persona, antes de continuar su ciclo.
- Versátil: Soportan complejos requisitos de despliegue continuo, incluyendo la habilidad de hacer forks, loops, o realizar trabajo en paralelo.
- Extensible: Además, también soporta extensiones personalizadas de su lenguaje específico de dominio, así como múltiples opciones para la integración con otros plugins.

Mientras que Jenkins siempre ha permitido formas rudimentarias de enlazar jobs para realizar tareas secuencialmente, los Pipelines Jenkins hacen de este concepto algo mucho más destacable.

Aquí puedes ver un ejemplo de escenario de despliegue continuo configurable en un Pipeline Jenkins



Cuando estamos haciendo uso de los pipelines es importante conocer todas las palabras clave que están disponibles para idear los scripts que nos ayudarán a dicha tarea.

Estas están documentadas a profundidad en la documentación:

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

Los bloques dentro de un pipeline declarativo pueden contener únicamente 3 cosas: Secciones, Directivas y Pasos (Sections, Directives, Steps)

Secciones (*Sections)*

Las secciones en un pipeline declarativo contienen Directivas y Pasos (Directives, Steps) y definen la estructura del mismo.

Usualmente las palabras clave que veremos en las secciones son:

- Pipeline: Modelo definido por un usuario de un proceso de despliegue continuo
- Node (nodo): Máquina que es parte del entorno de Jenkins y es capaz de ejecutar un Pipeline Jenkins
- Stage (etapa): Subconjunto de tareas que se realizan a través de todo el ciclo del pipeline (Construir, Probar, Desplegar, etc…), que es utilizado por varios plugins para visualizar o mostrar el estado y progreso del proceso.
- Step (paso): Cada una de las tareas que componen una etapa. Básicamente, cada paso le dice a Jenkins, que hacer en cada uno de los puntos concretos del ciclo a realizar.
- Post Define una seria de una o mas instrucciones que corren al final del pipeline y estan marcados con

una condición, tal como: always, success o failure. Usualmente se usan para enviar notificaciones de el resultado final de dicho pipeline.

Directivas (directives)

Las directivas expresan la configuración del pipeline

- **Trigger** Esta sección define las formas automatizadas en las cuales un pipeline debería ejecutarse de nuevo. En nuestro ejemplo el comando cron indica una frecuencia de tiempo. Hay 3 disponibles
  - **cron** Acepta un string al estilo "cron" para definir la frecuencia con la que el pipeline se debe ejecutar
  - **upstream** Acepta un string con nombres de jobs separado por comas y un limite. Entonces cuando uno de los jobs en el string llega a su limite, el pipeline se ejecuta de nuevo.
  - **pollSCM** Acepta un string al estilo "cron" que define un intervalo sobre el cual se deben buscar nuevos cambios en el sistema de control de versiones.
- **Agent Define** que tipo de agente estaremos usando
- **Options** Hay un buen numero de opciones que hacen diferentes cosas:
  - **buildDiscarder** Persiste artefactos y salida de consola para un número especifico de ejecuciones
  - **checkoutToSubdirectory** clona el repositorio en cuestión en un sub-directorio del espacio de trabajo
  - **disableConcurrentBuilds** Prohíbe la ejecución concurrente del pipeline. Útil para cuando se acceden a recursos compartidos

- **newContainerPerStage** cuando esta especificado cada stage creara un nuevo contenedor de docker en el mismo nodo, en lugar de correr todos los stages en el mismo contenedor.
- **overrideIndexTriggers** Permite sobre escribir el valor por defecto de los disparadores que indexan las ramas (branch) en el sistema de control de versiones.
- **preserveStashes** preserva el código de los últimos n trabajos terminados. En caso de ser necesario repetir la compilación.
- **quietPeriod** Configura el periodo de inactividad en segundos para el pipeline.
- **retry** reintenta el pipeline en caso de fallo
- **skipDefaultCheckout** Saltearse clonar el código del repositorio default configurado en el agent
- **skipStagesAfterUnstable** no ejecutar stages después de que la compilación resulto en "intestable"
- **timeout** como en el ejemplo, configura un tiempo máximo para que la compilación termine. Después de este tiempo jenkins cancelará el pipeline.

- **Environment** Define un conjunto de llave valor, que se usa para definir variables de entorno
- **Parameter** Define una lista personalizada de parámetros disponibles durante la ejecución del pipeline
- **Stage** permite la agrupación lógica de steps (como ya habíamos dicho antes)
- **When** Determina si un stage debería ser ejecutado dependiendo de la condición que defina.

Pasos (*Steps)*

Los *steps* son la parte mas fundamental de un *pipeline.* Definen las operaciones que van a ser ejecutadas.

- **sh** define un comando de terminal. Abre un *shell* y lo ejecuta. Es posible definir casi cualquier operación usando este tipo de paso
- **custom** Jenkins nos ofrece muchas operaciones que se pueden usar como *steps*. (por ejemplo *echo)* muchos de ellos simplemente envuelven instrucciones *sh* para que sean mas fáciles de usar. Por conveniencia los plugins también pueden definir sus propias operaciones
- **script** Estos ejecutan un bloque de código escrito en *groovy* que pueden ser usados para escenarios no comunes en donde se necesita algún tipo de control de flujos.

Sintaxis de un Pipeline Jenkins

Éstas son las diferencias entre los dos tipos de sintaxis, la de los Declarative Pipelines, y la de los Scripted Pipelines.

Ten en cuenta que tiene elementos comunes que pueden ser fácil de confundir.

Fundamentos de los Declarative Pipelines

Éste sería un ejemplo:

```
pipeline {
    agent any
    stages {
        stage('Construir') {
            steps {
                //
            }
        }
        stage('Probar') {
            steps {
                //
            }
        }
        stage('Desplegar') {
            steps {
                //
            }
        }
    }
```

En este Pipeline podemos destacar los siguientes elementos:

1. ***agent any*** : Esto indica que el pipeline se ejecutará en cualquiera de los nodos.
2. ***stage***`('Construir')`
   Define la etapa con el nombre “Construir” (utilizado para el seguimiento en el reporte y logs)
3. ***steps***: Esta parte contiene cada uno de los pasos a realizar en la etapa de “Construir”

Fundamentos de los Scripted Pipelines

En los Pipelines Scripted, uno o más de los bloques del nodo hacen el trabajo principal a través de todo el ciclo. Aunque no es un requisito imprescindible, al encerrar las tareas en el bloque del nodo se consiguen dos cosas:

1. Programar los pasos del bloque para ejecutarse añadiendo un elemento a la cola de trabajo de Jenkins. Tan pronto como haya una ejecución disponible en un nodo, los pasos se ejecutarán.

2. Crear un workspace (un directorio específico para un Pipeline particular) donde las tareas se pueden realizar sobre los ficheros descargados del repositorio. Ten en cuenta que dependiendo de la configuración de Jenkins, algunos workspaces no se limpian automáticamente después de un periodo de inactividad.

Ejemplo de un Scripted Pipeline:

```
node {
    stage('Construir') {
          //
    }
    stage('Probar') {
          //
    }
    stage('Deploy') {
          //
    }
}
```

En este caso, lo más destacable sería:

1. ***node:*** Esto indica que el pipeline se ejecutará en cualquiera de los nodos.
2. ***stage***('Construir'): Define la etapa de 'Construir'. Este bloque es opcional en los Scripted Pipelines, pero, facilitará la lectura de cada una de las fases y tareas a seguir en la interfaz de Jenkins.
3. ***step:*** Esta parte contiene cada uno de los pasos a realizar en la etapa de "Construir"

Ejemplo de un Pipeline Jenkins

Aquí puedes ver un ejemplo más completo de un Pipeline Jenkins, utilizando la sintaxis declarativa.

```
pipeline {
    agent any
    options {
        skipStagesAfterUnstable()
    }
    stages {
      stage('Construir') {
        steps {
                sh 'make'
        }
      }
      stage('Probar'){
        steps {
                sh 'make check'
                junit 'reports/**/*.xml'
        }
      }
       stage('Desplegar') {
            steps {
                sh 'make publish'
            }
      }
   }
}
```

Ahora veamos un *pipeline* un poco mas complejo para después explicarlo:

```
pipeline {
    agent any
    triggers { cron('* * * * *') }
    options { timeout(time: 5) }
    parameters {
        booleanParam(name: 'DEBUG_BUILD', defaultValue: true,
        description: 'Es esto un build para depurar?')
    }
    stages {
        stage('Example') {
            environment { NAME = 'Debug' }
            when { expression { return params.DEBUG_BUILD } }
            steps {
                echo "Hola desde $NAME"
                script {
                    def browsers = ['chrome', 'firefox']
                    for (int i = 0; i < browsers.size(); ++i) {
                        echo "probando el browser ${browsers[i]}."
                    }
                }
            }
        }
    }
    post { always { echo 'Siempre es bueno decir hola!' } }
}
```

Ahora veamos paso a paso que es lo que hace el anterior pipeline:

1. Utiliza cualquier *agent* disponible
2. Se ejecuta automáticamente cada minuto
3. Se detiene si la ejecución tarda mas de 5 minutos

4. Pregunta si el parámetro DEBUG_BUILD es verdadero
5. Configura la variable NAME para que su valor sea “Debug”
6. Si el parámetro es verdadero:
    1. imprime “Hola desde Debug”
    2. imprime “Probando el navegador Chrome”
    3. imprime “Probando el navegador Firefox”
7. Imprime “Siempre es bueno decir hola!” Sin importar si la ejecución termina con o sin errores

Ahora bien, ya basta de ejemplos “hola mundo”!. Vamos a definir un *pipeline* funcional que clona, compila y prueba un repositorio.

Definimos nuestro *pipeline* asi:

```
pipeline {
    agent any
    stages {
        stage("Clonar") {
            steps {
                git url: 'https://github.com/ejemplo/gradle.git'
            }
        }
        stage("Compilar") {
            steps {
                sh "./gradlew compileJava"
            }
        }
        stage("Probar") {
            steps {
                sh "./gradlew test"
            }
        }
    }
}
```

Lo guardamos como ya hemos dicho y ya tenemos nuestro artefacto creado y listo para ser usado. Noten que en el paso de Clonar, el repositorio no existe. Deben ustedes crear su propio repositorio y modificar los comandos para compilarlo y probarlo.

En el caso particular de este ejemplo, es un proyecto de java que se compila y se prueba usando gradle en lugar de maven. Pero esto es cuestión de que lo ajusten a su gusto y a su proyecto.