Update Multiple Libraries from Jenkins Pipeline

Para esta POC teniamos que simular que teníamos múltiples librerías subidas a nuestro repositorio, para hacer una pipe que pueda actualizarlas todas en el orden correcto sin la necesidad de ir una por una a mano (Cada librería deberá tener una pipe propia que actualizara esa librería en concreto).

Para esto primero de todo son necesarios algunos archivos .txt en el repositorio que tengamos creado. Primero crearemos un archivo “Update Order” en el que pondremos el nombre de las pipes “hijas” o las pipes que actualizan las librerías de manera independiente.

Ejemplo:

- Library_1
- Library_3
- Library_2

De esta manera podemos establecer un orden de actualización para asegurar que no dan error con las dependencias de cada una.

Para simular esto en mi POC lo que he hecho en vez de tener 3 microservicios enteros de ejemplo, he creado tres archivos pom.xml, que simulan los archivos pom de los microservicios. De esta manera cada pipe independiente actualizara la versión de este archivo.

Entonces, además del archivo mencionado anteriormente, tambien necesitaremos otro mas, que se llamara “Check File” en este en un inicio tendremos que tener las versiones en las que esta cada librería, debería ser la misma para todas.

Ejemplo:

- 1.0.1
- 1.0.1
- 1.0.1

Este archivo se actualizara solo con la pipe. Este le sirve a la pipe para detectar posibles errores, por ejemplo que no se hayan actualizado las 2 primera pipes, y en la tercera se corta la luz. El archivo quedaría guardado por ejemplo con:
- 2.0.1, 2.01, 1.01

Entonces al volver a ejecutar la pipe, vera en el archivo que solo hace falta actualizar la 3ra librería, ya que las demás ya lo están. De todas manera existe un check llamado override, el cual si marcamos y ejecutamos la pipe, volverá a

actualizar las pipes que ya estaban actualizadas con la versión especificada. En el ejemplo anterior, se volverían a actualizar todas con las versión 2.0.1.

Ademas, al final de la ejecución, se hará un push con los cambios que hayamos hecho al repositorio donde se guardan estos “microservicios”.

Genial, ahora que ya sabemos lo básico de lo que tenemos, vamo a comentar paso por paso lo que hace la pipe. Tanto la pipe “padre”, la que ejecuta todas las demás pipes, y las pipes “hijas”, las que actualizan cada librería independientemente.

El código se mostrara al final, de todas manera podéis acceder a mi repositorio publico para leerlo con mas facilidad:

https://github.com/RubenMailloBaena/JenkinsPipelineTest/tree/main/Libraries_Update_Files

Pipe Padre:

El primer paso, es añadir los parámetros necesarios, para esta necesitamos una String que será la nueva versión a la que queremos actualizar las librerías. Este tan solo puede contener números y puntos: (ej: 2.0.1.2).

Ademas tambien tendremos un booleano, que será el check comentado anteriormente, para saber si queremos volver a actualizar las librerías que ya estaban en esa versión.

Entonces, ahora que tenemos los parámetros el primer stage es crear el workspace o directorio de Jenkins en el que clonaremos nuestro repositorio. Este se llamara “/repo-temp”.

Y en el siguiente stage clonamos el repositorio.

El siguiente stage “Execute Pipelines” es donde empieza. Lo primero que hacemos es comprobar que existe el archivo “Order_File” donde debería estar el orden de actualizacion de las librerías. Si no se encuentra, la pipe fallara y terminara. Si lo encontramos, separamos su contenido para obtener el orden de pipes a ejecutar. Seguidamente, buscamos el archivo de Check_File, donde están las versiones actuales de cada librería.

Ahora empezamos un bucle donde iremos ejecutando las pipes correspondientes, si no estamos con el check de override activado, comprobarmos en el checkfile si esta librería esta en la misma versión que queremos actualizar. Si es así, pasamos a la siguiente iteración del bucle. Si no, ejecutamos la pipe hija, pasandole por

parámetros los datos del repositorio y la versión a actualizar.

Esperamos a que la pipe hija termine, si termina correctamente, actualizamos el checkFile, para tener correctamente todas las librerías actualizadas. Este proceso se repite hasta que se ejecutan todsa las pipes hijas correctamente, si algúna en algún momento falla, se detiene la ejecución. Pero las librerías anteriores quedaran actualizadas. Ademas, cada librería creara un archivo de logs, donde podemos ver sus prints para encontrar errores o posibles problemas rápidamente. Tambien se imprimirán por la consola de la pipe padre.

Por ultimo lo que hacer es hacer un push con el archivo de check file actualizado.

Ahora vamos a ver las pipes hijas.

Lo primero que hacen al ser ejecutadas, es recuperar su respectivo archivo pom.xml del repositorio que deberán actualizar.

Si no lo encuentran, esta fallara y se detendrá la ejecución. En caso contrario validara que la versión de entrada que le ha llegado es superior a la versión en la que está la Liberia. (Si estamos en la versión 2.0.1, no podremos actualizarla a la versión 1.9.2 ya que es una versión anterior).

SI la versión es valida, modificara el archivo con esta nueva versión y subirá el cambio al repositorio, con la información que le ha llegado por parámetros.

Podemos ver tambien que tiene una función en la que le podemos pasar Strings, y estos quedaran guardados en los logs que mas tarde podremos recuperar desde la pipe padre.
Habra un LOG general con el contenido de todas las pipes, y además un LOG independiente para cada pipe. Para poder buscar de manera mas precisa.

De todas formas el código esta lleno de comentarios explicativos sobre lo que hace cada parte del código 😊. Pero se puede ver mucho mas fácil en el github.

https://github.com/RubenMailloBaena/JenkinsPipelineTest/tree/main/Libraries_Update_Files

PIPE HIJA (actualiza la librería):

```
pipeline{
  agent any

  parameters{
    //Parametros que recibiremos a traves de la pipe padre.
    string(name: 'PARAM_1', defaultValue: '0.0.0', description: 'New Version')
    string(name: 'USERNAME', defaultValue: 'rubenDefaultValue', description: 'github username')
    string(name: 'EMAIL', defaultValue: 'rubenDefaultValue@gmail.com', description: 'github email')
    string(name: 'BRANCH', defaultValue: 'mainDefaultValue', description: 'github push branch')
    string(name: 'PARENT_PATH', defaultValue: 'defaultPath', description: 'parent pipeline path')
    string(name: 'PUSH_URL', defaultValue: 'url', description: 'URL to push new changes')
    booleanParam(name: 'OVERRIDE_VERSION', defaultValue: false, description: 'overrides version')
  }

  environment{
    POM_FILE = '_pom.xml'
    MESSAGE = ''
  }

  stages{
    stage('Get Respective Pom.xml'){
      steps{
        //Entramos al repositorio donde tenemos clonado el repositorio de github, donde en este caso tenemos los poms.xml de las librerias.
        dir('../Update_Versions_Pipeline/repo-temp/Libraries_Update_Files'){
          script{
            logMessage("----- Starting Pipeline... -----")

            //PARA SIMULAR UN ERROR AL ACTUALIZAR Y QUE NO SE GUARDE SU VERSION EN LA CHECKLIST. (descomentar el error).
            //error "TEST ERROR"

            //Guardamos el nombre del pom.xml que tenemos que actualizar y comprobamos que exista el archivo.
            def pomName = "${env.JOB_BASE_NAME}${env.POM_FILE}"
            if(fileExists("${pomName}")){
```

```
                //Si existe, guardamos su contenido. Obtenemos la version actual escrita en el POM.XML
                logMessage("POM.xml FOUND\n")
                def file = readFile(pomName)

                def currentVersion = (file =~ /<version>(.*?)<\/version>/)[0][1]
                logMessage("Current Version: ${currentVersion}")
                logMessage("New Version: ${env.PARAM_1}")
                logMessage("Validating new Version...\n")

                //Comprobamos que es una version valida, pasandole la version actual y la nueva.
                if(!isValidVersion("${currentVersion}", "${env.PARAM_1}")){
                  logMessage("[ERROR]: The new version can't be de same, or a latter number!")
                  error "[ERROR]: The new version can't be de same, or a latter number!"
                }

                if(params.OVERRIDE_VERSION){
                  logMessage("Overriding new version!\n")
                }
                else{
                 logMessage("New Version Validated!\n")
                }

                logMessage("Updating Library...")

                //Si llegamos aqui significa que la version es valida, por lo que actualizamos el pom.xml de esta libreria con la nueva version.
                def fileUpdated = file.replaceFirst(/<version>.*?<\/version>/, "<version>${env.PARAM_1}</version>")
                writeFile(file: "${pomName}", text: fileUpdated)

                logMessage("Library Updated to Version: ${env.PARAM_1}")
              }
              else{
                error "POM.xml not found"
              }
            }
          }
        }
```

```
    }
    stage('Push Updated Pom.xml'){
      steps{
        //Entramos al directiorio donde se encuentra el repositorio de git clonado por la pipe padre.
        dir('../Update_Versions_Pipeline/repo-temp'){
          //Hacemos un push con el nuevo archivo pom.xml actualizado.
          withCredentials([string(credentialsId: 'github_push', variable: 'GITHUB_TOKEN')]){
            script{
              logMessage("Pushing Changes to Repo")

              powershell '''
                git config core.autocrlf true

                git config user.name "$env:USERNAME"
                git config user.email "$env:EMAIL"

                git add Libraries_Update_Files/*.xml

                $env:GIT_URL = "https://${env:GITHUB_TOKEN}$env:PUSH_URL"

                git commit -m "${env:JOB_BASE_NAME} updated (v${env:PARAM_1})"
                git push --set-upstream $env:GIT_URL $env:BRANCH
              '''

              logMessage("Succesfully Pushed Changes to Repo")
            }
          }
        }
      }
    }
}
post {
    always {
      //Por ultimo, limpiamos el workspace de esta pipe, para que no queden archivos innecesarios.
      //Podremos ver los logs desde el workspace de la pipe padre.
      logMessage("Pipeline Finished!\n")
```

```
        cleanWs()
      }
    }
}

//Esta funcion nos permite determinar si la version que nos ha llegado por parametros es valida
//Sera valida si la nueva version es mas grande que la actual.
//Si por parametros nos llega el OVERRIDE_VERSION en true, si las versiones son iguales tambien sera una version valida.
//En caso de que la nueva version sea una mas pequeña, devolveremos false, no sera una version valida.
def isValidVersion(currentVersion, newVersion) {
    //Separamos los numeros de cada version en listas, separandolos por los puntos.
    def currentVersionNums = currentVersion.split("\\.")
    def newVersionNums = newVersion.split("\\.")

    // Miramos que version es mas larga (ex: 1.0.0 / 1.1 --> recorreremos 3 veces (1.0.0))
    def repetition = Math.max(currentVersionNums.length, newVersionNums.length)

    for (int i = 0; i < repetition; i++) {
        //Si una version tiene mas numero que otra, dara error indexOutOfBounds al intentar acceder al los numeros, es por eso que si nos salimos lo asignaremos como un 0.
        //Ejemplo: (recorremos la version 1.1.1, y la actual es la 1.1, al comparar el ultimo 1 (1.1.1), la segunda version al no tener mas numeros (1.1), equivaldra a un 0 (1.1.0)).
        //De esta manera evitaremos errores. Ademas pasamos de string a numeros para su comparacion.
        def currentNum = i < currentVersionNums.length ? Integer.parseInt(currentVersionNums[i]) : 0
        def newNum = i < newVersionNums.length ? Integer.parseInt(newVersionNums[i]) : 0

        //Si un numero de la nueva version es mas pequeño, es mas antigua. No es una version valida.
        if(newNum < currentNum){
            return false
        }
        //Si un numero de la nueva versio es mas grande, es una version mas nueva. Es una version valida.
        else if(newNum > currentNum){
            return true
        }
    }
    //Por ultimo, si llegamos a esta parte significa que que las versiones son iguales.
    //Si tenemos que hacer un OVERRIDE, las version sera valida. En caso contrario no lo sera.
```

```
    return params.OVERRIDE_VERSION
}


//esta funcion nos permite imprimir mensajes tanto por la consola como por archivos de texto.
//Escribe el mesaje en un archivo propio (Library_1.log), despues en otro archivo donde se encuentran los logs de todas las pipes (All_Libraries.log)
//Y finalmente tambien lo escribe por la consolas de la propia pipe.
//Estos archivos son los que se recuperan mas tarde desde la pipe padre para imprimir los logs.
def logMessage(message) {
    script {
        withEnv(["MESSAGE=${message}"]){
            powershell '''
                cd $env:PARENT_PATH

                if (Test-Path -Path 'repo-temp') {
                    # Añadimos el mensaje al archivo de salida. Y lo imprimimios en la consola
                    "[$env:JOB_BASE_NAME #$env:BUILD_ID]: $env:MESSAGE" | Out-File -Append -FilePath (Join-Path -Path 'repo-temp' -ChildPath "$env:JOB_BASE_NAME.log")
                    "[$env:JOB_BASE_NAME #$env:BUILD_ID]: $env:MESSAGE" | Out-File -Append -FilePath (Join-Path -Path 'repo-temp' -ChildPath "All_Libraries.log")
                    Write-Host $env:MESSAGE
                } else {
                    Write-Host 'repo-temp directory does not exist.'
                }
            '''
        }
    }
}
```

PIPE PADRE (ejecuta las pipes hijas):

```
pipeline{
  agent any

  parameters {
    //PARAMETRO CON EL NUEVO NUMERO DE VERSION
    validatingString(
      name: 'NEW_VERSION',
      defaultValue: '-1',
      regex: /^\s*[0-9.]+\s*$/, //SOLO PERMITE NUMEROS Y PUNTOS
      failedValidationMessage: 'Only can use numbers and dots (ex: 1.0.1)',
      description: 'Add the new version for the libraries.' )

    //PARAMETRO PARA DETERMINAR SI SE SOBREESCRIBEN LAS VERSIONES
    booleanParam(
      defaultValue: false,
      description: 'Update Libraries with the same version',
      name: 'OVERRIDE_VERSION'
      )
  }

   environment{
    //NOMBRE DEL ARCHIVO CON EL ORDEN DE PIPES A EJECUTAR.
    ORDER_FILE = 'UpdateOrder.txt'
    //ARCHIVO CON LAS VERSIONES DE LA LIBRERIS.
    CHECK_FILE = 'CheckList.txt'

    //GIT ATTRIBUTES
```

```
    REPO_URL = 'https://github.com/RubenMailloBaena/JenkinsPipelineTest.git'
    PUSH_URL = '@github.com/RubenMailloBaena/JenkinsPipelineTest.git'
    BRANCH = 'main'
    USERNAME = 'RubenMailloBaena'
    EMAIL = 'rubenmaillo2003@gmail.com'
    CURRENT_LIBRARY = ''
  }

  stages {
    stage('Prepare Workspace') {
      steps {
        //LIMPIAMOS EL DIRECTORIO PREVIO Y CREAMOS UN WORKSPACE NUEVO
        cleanWs()
        powershell '''
        if (Test-Path -Path repo-temp){
          Remove-Item -Path repo-temp -Recurse -Force
        }
        mkdir repo-temp
        '''
      }
    }

    stage('Clonar Repositorio') {
      steps {
        //CLONAMOS EL REPOSITORIO DE GIT EN EL WORKSPACE
        dir('repo-temp') {
          git branch: "${env.BRANCH}", url: "${env.REPO_URL}"
```

```
        }
      }
    }

    stage('Execute pipelines') {
      steps {
        //Entramos al directorio del WS donde tenemos los diferentes archivos necesarios (el orden de ejecucion y lista de checks).
        dir('repo-temp/Libraries_Update_Files') {
          script {
            //Verificamos que se encuentra el archivo con el orden de ejecucion.
            if (fileExists("${env.ORDER_FILE}")) {
              //Obtenemos su contenido y lo separamos con los \n, para tener los nombres en una lista (libraries).
              def content = readFile("${env.ORDER_FILE}")
              echo "Executing pipes in the next order:\n${content}"

              def libraries = content.split('\n').collect { it.trim() }.findAll { it }

              //Hacemos los mismo con el archivo que contiene las versiones (Checks), y lo guardamos en versions.
              def versions = []
              if(fileExists("${env.CHECK_FILE}")){
                def checkContent = readFile("${env.CHECK_FILE}")
                echo "Current Versions: \n${checkContent}"
                versions = checkContent.split('\n').collect{it.trim()}.findAll{it}
              } else{
                powershell "exit 1"
              }
```

```
//Añadimos un contador para saber por que libreria vamos en la ejecucion dentro del bucle (1,2,3, etc.).
int count = 0
for (library in libraries) { //Recorremos todas las librerias de la lista.

//Obtenemos la nueva version que tenemos en parametros, y la que tenemos en la lista de checks.
def newVersion = params.NEW_VERSION.trim()
def currentVersion = versions[count].trim()

//Si las versiones son iguales, pasaremos a la siguiente ejecucion, nos saltaremos esta pipe. (En caso de que ya este actualizada).
//Si tenemos marcado la checkBox de override, aunque las versiones sean iguales volveremos a ejecutar la pipe para actualizarlas.
if (newVersion != currentVersion || params.OVERRIDE_VERSION) {
  echo "Triggering ${library}..."
  //Añadimos todos los parametros necesarios a la pipie que vamos a ejecutar.
  def result = build job: "${library}",
  parameters: [string(name: 'PARAM_1', value: "${params.NEW_VERSION}"),
      string(name: 'USERNAME', value: "${env.USERNAME}"),
      string(name: 'EMAIL', value: "${env.EMAIL}"),
      string(name: 'BRANCH', value: "${env.BRANCH}"),
      string(name: 'PARENT_PATH', value: "${env.WORKSPACE}"),
      string(name: 'PUSH_URL', value: "${env.PUSH_URL}"),
      booleanParam(name: 'OVERRIDE_VERSION', value: params.OVERRIDE_VERSION)],
  propagate: true,
```

```
                    wait: true


                    //Actualizamos la version que teniamos en la lista de Checks.
                    versions[count] = "${params.NEW_VERSION}"


                    //Si se ha ejecutado con exito, sobreescribimos en el archivo de checks esta nueva version.
                    def contenidoArchivo = ''
                    for (int i = 0; i < versions.size(); i++) {
                       contenidoArchivo += versions[i]
                       if (i < versions.size() - 1) {
                          contenidoArchivo += '\n'
                       }
                    }
                    writeFile file: "${env.CHECK_FILE}", text: contenidoArchivo
                    echo "Updated ${env.CHECK_FILE} with new versions."


                    //Por ultimo, vamos al directorio padre, donde se encuentran los archivos de logs generados por las pipes.
                    //E intentamos recuperar el log correspondiente a la pipe ejecutada. Si lo encontramos mostramos su contenido por consola.
                    dir('..'){
                       script{
                          withEnv(["CURRENT_LIBRARY=${library}.log"]){
                             echo "Printing ${library} execution logs:"
                             powershell '''
                                if (Test-Path $env:CURRENT_LIBRARY) {
                                   Get-Content $env:CURRENT_LIBRARY | ForEach-Object { Write-Host $_ }
```

```
                    } else {
                        Write-Host "pipeline logs not found."
                    }
                '''
            }
        }
    }
    }
    count++
  }
} else {
    echo "File '${env.ORDER_FILE}' not found. Exiting."
    powershell "exit 1"
}
  }
 }
}

post {
  always {
    //Cuando toda la pipe termine, sea con errores o con exito siempre llegaremos a esta parte.
    //Aqui subimos al repositorio el archivo de Checks con las nuevas versiones. Para poder tenerlas en cuenta en la siguiente ejecucion.
    dir('repo-temp') {
      withCredentials([string(credentialsId: 'github_push', variable: 'GITHUB_TOKEN')]) {
```

```
                script {
                    powershell '''
                        git config core.autocrlf true

                        git config user.name "$env:USERNAME"
                        git config user.email "$env:EMAIL"

                        git add Libraries_Update_Files/*.txt
                        $env:GIT_URL = "https://${env:GITHUB_TOKEN}$env:PUSH_URL"

                        git commit -m "CheckFile Updated!"
                        git push --set-upstream $env:GIT_URL $env:BRANCH
                    '''

                    echo "Successfully pushed new Checks To Repo"
                }
            }
        }
    }
  }
}
```