UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Desarrollo de rutinas para integración continua en la implementación de software

Autor: Jesús Martínez Alfaro

Tutor: Antonio León Fernández

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingeniería de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2020-21

Valencia, 29 de junio de 2021

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universitat Politècnica de València
Edificio 4D. Camino de Vera, s/n, 46022 Valencia
Tel. +34 96 387 71 90, ext. 77190
www.etsit.upv.es

VLC/ CAMPUS
VALENCIA, INTERNATIONAL CAMPUS OF EXCELLENCE

EMAS
Gestión ambiental verificada
REG NO. ES-CV- 000030

Resumen

Las empresas de desarrollo de software en la actualidad se enfrentan a la gestión de programas de un gran tamaño y de equipos con numerosos trabajadores dedicados a la mejora y evolución de programas interrelacionados entre sí. Ante ello, resulta casi imprescindible el empleo de herramientas tales como los repositorios compartidos o la compilación continua para permitir que el desarrollo en paralelo por parte de todos los trabajadores pueda confluir de forma segura y eficaz.

Para ello, analizaremos en base a la experiencia la utilidad de la herramienta de Jenkins, un programa que, entre otras cosas, permite automatizar la detección de errores y conflictos en el código y la generación de versiones de diferentes productos, facilitando el trabajo de todo el equipo de trabajadores y permitiendo optimizar al máximo los recursos. A su vez, integraremos toda este sistema con los servicios de Jira, un sistema que permite asignar tareas concretas sobre cada producto y cuyo desarrollo será el que motivará la generación de nuevas versiones con nuevas funcionalidades en cada producto. En definitiva, las tecnologías actuales ofrecen una gran cantidad de recursos y opciones que han de ser aprovechados en toda empresa que aspire a ser competitiva y eficiente, y en este trabajo estudiaremos algunas de las más utilizadas actualmente.

Resum

Les empreses de desenvolupament de software en l'actualitat s'enfronten a la gestió de programes d'una gran grandària i d'equips amb nombrosos treballadors dedicats a la millora i evolució de programes interrelacionats entre sí. Davant això, resulta quasi imprescindible l'ús d'eines com ara els repositoris compartits o la compilació contínua per a permetre que el desenvolupament en paral·lel per part de tots els treballadors puga confluir de manera segura i eficaç.

Per a això, analitzarem sobre la base de l'experiència la utilitat de l'eina de Jenkins, un programa que, entre altres coses, permet automatitzar la detecció d'errors i conflictes en el codi i la generació de versions de diferents productes, facilitant el treball de tot l'equip de treballadors i permetent optimitzar al màxim els recursos. Al seu torn, integrarem tota aquest sistema amb els serveis de Jira, un sistema que permet assignar tasques concretes sobre cada producte i el desenvolupament del qual serà el que motivarà la generació de noves versions amb noves funcionalitats en cada producte. En definitiva, les tecnologies actuals ofereixen una gran quantitat de recursos i opcions que han de ser aprofitats en tota empresa que aspire a ser competitiva i eficient, i en aquest treball estudiarem algunes de les més utilitzades actualment.

Abstract

Software development companies today face managing large programs and team making with numerous workers dedicated to improving and evolving interrelated programs. Given this, it is essential to use tools such as shared repositories or continuous compilation to allow parallel development by all workers to flow safely and efficiently.

To do this, we will analyze, based on experience, the usefulness of the Jenkins tool, a program that, among other things, allows automating the detection of errors and conflicts in the code and the generation of versions of different products, facilitating the work of the team of workers and

allowing the maximum optimization of resources. Besides, we will integrate this entire system with Jira services, a system that allows assigning specific tasks on each product and whose development will be the one that will motivate the generation of new versions with new functionalities in each product. In conclusion, current technologies offer a large number of resources and options that must be used in any company that aspires to be competitive and efficient, and in this work we will study some of the most used today.

Índice

Capítulo 1. Introducción y objetivos

1.1 Motivación

En la mayoría de empresas enfocadas al desarrollo de software-y no hace falta que sean de un gran tamaño- se debe enfocar de forma clara la manera en la que se va a resolver la colaboración y el trabajo entre todos los desarrolladores. Tal y como hemos podido ver a lo largo del grado, el trabajo en equipo es un punto clave en el ámbito laboral, y, así, no se suelen encontrar situaciones en las que cada trabajador desarrolle su trabajo por separado. Más bien todo lo contrario.

Cuando se trata de numerosos productos, con fuentes de código relacionadas entre sí, y que van avanzando, creciendo y corrigiéndose con el paso del tiempo, la situación acaba desembocando en un cierto número de trabajadores desarrollando los mismos programas.

Por ello, han surgido, desde hace ya unos cuantos años, diferentes herramientas que permiten facilitar esta tarea. Así, por ejemplo, hoy en día es una práctica muy habitual el uso de repositorios remotos, que serán espacios en la nube donde se almacenarán los diferentes programas y productos. De esta forma, cada desarrollador irá trabajando sobre una copia de dicho repositorio en su entorno local, e irá subiendo los cambios una vez estos hayan funcionado y estén probados.

Sin embargo, esta funcionalidad puede acabar siendo un arma de doble filo, dado que dicho trabajo en paralelo por parte de un número determinado de desarrolladores puede desembocar en que las competencias se diluyan y no quede claro quién se encarga de qué, o, incluso, a conflictos en el código, ya que, como es lógico, puede darse en caso donde las diferentes mejoras o correcciones que se realicen sean incompatibles entre sí.

Por tanto, este trabajo pretende estudiar algunas de las posibilidades que están implantándose a día de hoy en las empresas de desarrollo de software, entendiendo que el conocimiento de las mismas resulta fundamental a la hora de gestionar adecuadamente un equipo de trabajo, y que su uso-tanto de las aplicaciones que analizaremos en este trabajo, como de muchas otras que se pueden encontrar-resulta clave para que una empresa pueda optimizar sus recursos y aprovechar al máximo el trabajo realizado.

1.2 Objetivos

El objetivo principal de este trabajo será conocer en profundidad la herramienta de la compilación continua, entendiendo los beneficios que puede aportar en el desarrollo de software actual y viendo sobre la práctica la implementación que puede tener en una empresa de tamaño grande.

Para ello se empleará el programa Jenkins [1], que no es más que un programa con una interfaz intuitiva y fácil de emplear que nos permitirá automatizar una serie de acciones sobre el software desarrollado con dos objetivos fundamentales: la comprobación del funcionamiento de los productos tras los diferentes cambios que puedan haberse introducido, por un lado; y la generación de versiones automatizada y de forma periódica, permitiendo que la conexión entre la empresa y los diferentes clientes pueda realizarse de forma rápida y sencilla, y que el producto pueda estar disponible y actualizado en la última versión existente de forma sencilla y rápida.

Las funcionalidades ofrecidas por Jenkins, pese a la sencillez que ofrece en cuanto a su funcionamiento, pueden llegar a ser muy amplias. Por ello, tras una primera fase donde se

implementen los productos principales de los que se dispone en Jenkins, se abordarán las posibilidades de integración de este trabajo con Jira [2].

Jira es un programa de reparto de tareas de forma interactiva que permite la elaboración y asignación de tareas sobre todo el software en desarrollo. De esta forma, supone una herramienta clave para optimizar el reparto de tareas, y además permitirá generar un flujo de trabajo eficiente al permitir la conexión entre los diferentes departamentos de la empresa, por ejemplo, al facilitar la tarea de revisión en aquellas personas encargadas de la puesta a punto definitiva de los productos de la empresa.

Conjuntar Jira con Jenkins nos permitirá optimizar aún más la generación de versiones, facilitando la documentación de los distintos cambios aplicados a cada versión generada.

En definitiva, las herramientas a día de hoy para mejorar el desarrollo de software son muy amplias y variadas, y en este proyecto se pretende conocer, a través de la aplicación práctica en una empresa, si los beneficios [3] son tales como los podemos prever, en general, y si las aplicaciones elegidas pueden ser las óptimas para una empresa grande, en particular.

1.3 Metodología

Para realizar este trabajo, lo primero será familiarizarse con el programa que marcará todo el proyecto, Jenkins, por lo que se comenzará con una fase de instrucción y formación sobre el funcionamiento de la misma.

Una vez hecho esto, será necesario plantearse los objetivos generales perseguidos para la compilación de los productos. Es decir, dado que el trabajo de compilación automática se realizará con una serie de productos con muchas particularidades y programas resultantes diferentes, será necesario ver qué puntos en común podremos tener de cara a poder estructurar los rasgos generales de nuestro trabajo.

Tras ello, se estudiarán las opciones de integrar Jira en Jenkins, viendo cómo podríamos aprovechar la información obtenida con la resolución de tareas de Jira para documentar los cambios generados en cada versión.

Finalmente, se implementarán todas estas funciones para una serie de productos, por lo que estudiaremos sobre un grupo de los mismos como hacer la implementación concreta de cada uno de los productos elegidos. Es necesario aclarar que, al tratarse de productos reales de una empresa, el contenido no se podrá ver, si bien todos los elementos diseñados para su compilación y generación de versiones, se podrán documentar adecuadamente.

Para conseguir los objetivos marcados en este apartado, emplearemos esta calendarización:

Actividad (Semanas)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Formación sobre Jenkins y Jira																					
Preparación jobs básicos																					
Vinculación con Jira																					
Preparación jobs complejos																					
Redacción de la memoria																					

Diagrama 1: Diagrama de Gantt

En esta tabla podemos ver el reparto temporal del trabajo, en base a la metodología y las distintas fases planteadas previamente.

1.4 Estructura de la memoria

La memoria constará de 5 capítulos.

El primero servirá para introducir el contenido del proyecto y estructurar las diferentes fases del trabajo.

En el segundo abordaremos los primeros pasos de trabajo con la aplicación de Jenkins, comenzando a ver las posibilidades de trabajo que nos ofrece y pudiendo sacar una estructura general que seguiremos en los capítulos posteriores. En este capítulo se desarrollarán los primeros extractos de código que necesitaremos

En el tercero, analizaremos las opciones que nos ofrece la integración de Jira en Jenkins, centrando especialmente las utilidades de la misma en la documentación de las diferentes versiones generadas. En base de los beneficios que pueda reportarnos, desarrollaremos nuevos pequeños programas que podamos necesitar en la fase final.

En el cuarto capítulo, aplicaremos los programas obtenidos en los dos capítulos anteriores sobre una serie de productos concretos, particularizandolo para su automatización.

Por último, concluiremos con un quinto capítulo para valorar el éxito de los objetivos planteados y valorar si las utilidades que pretendíamos conseguir han sido satisfechas en el proyecto.

Capítulo 2. Primera fase de integración con Jenkins

2.1 Consideraciones previas

Una vez realizada la introducción al proyecto que vamos a desarrollar, se ha de comenzar con el trabajo con Jenkins. Para ello, nos formaremos sobre su funcionamiento a través de la propia página de Jenkins y diferentes vídeos acerca de sus funciones.

Jenkins requerirá de un servidor web a través del cual vayamos trabajando nuestro proyecto. En nuestro caso, instalaremos dicho servidor web en una máquina virtual en la que además instalaremos una copia de todo el repositorio de código de la empresa, de forma que en esa misma máquina se podrán ir compilando los diferentes proyectos, que generarán aplicaciones sobre las cuales buscaremos dicha automatización de generación de versiones, y, por tanto, la integración continua.

Aquí, podemos observar como la máquina virtual usará Windows Server 2016, 8 GB de memoria RAM y contará con los recursos equivalentes a un procesador de 2 núcleos de 2,20GHz. En este caso, la máquina virtual nos vendrá dada por la empresa, por lo que no se tendrá que realizar demasiado trabajo para su configuración y puesta a punto.

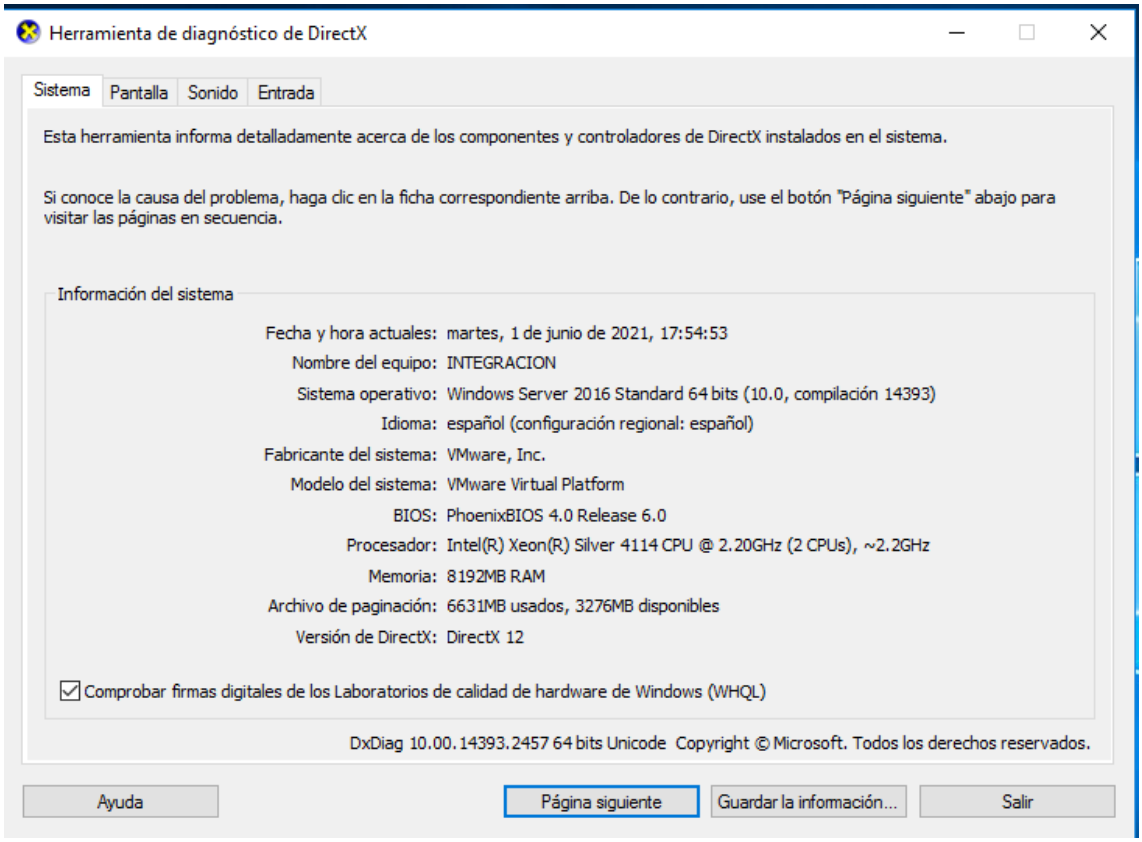


Figura 1: Especificaciones máquina virtual

Hecho esto, bastará con acceder desde cualquier navegador de un ordenador con acceso a la máquina virtual (o desde la propia máquina virtual, claro) con tu usuario registrado en Jenkins para poder acceder al panel de control y comenzar a trabajar.

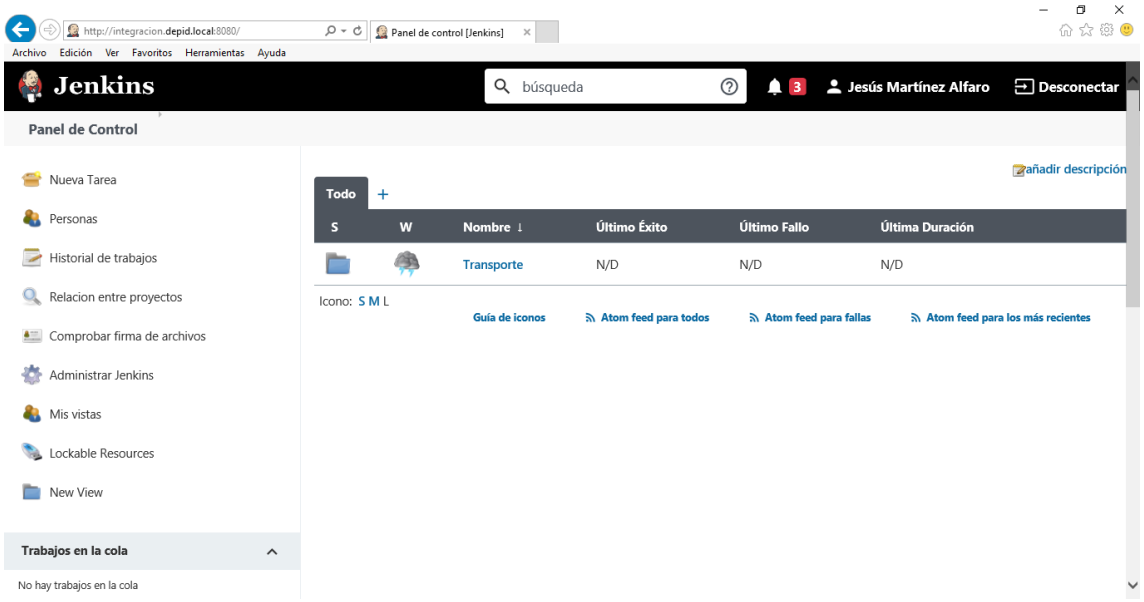


Figura 2: Interfaz general de Jenkins

Como podemos ver en la interfaz del programa, a la izquierda encontraremos una opción de "Administrar Jenkins", desde la cual podremos gestionar algunas cuestiones necesarias para el funcionamiento del programa, como la instalación de plugins, el establecimiento de credenciales para los diferentes servicios con los que conectemos Jenkins, o la configuración de esos mismos servicios, tales como Jira (que trabajaremos en capítulos posteriores) o con los servidores de código fuente, en nuestro caso, SVN[4] o Gitlab[5].

Para ello, el primer paso será la instalación de los plugins correspondientes, tales como aquellos que permiten la compilación con Visual Studio (MSBuild), programa que emplearemos para la mayoría de nuestras aplicaciones; los que nos permitirán conectar con el servidor de SVN y de Gitlab; o las de funciones como la consulta HTTP Request o la aplicación de Jira, que también emplearemos en capítulos posteriores.

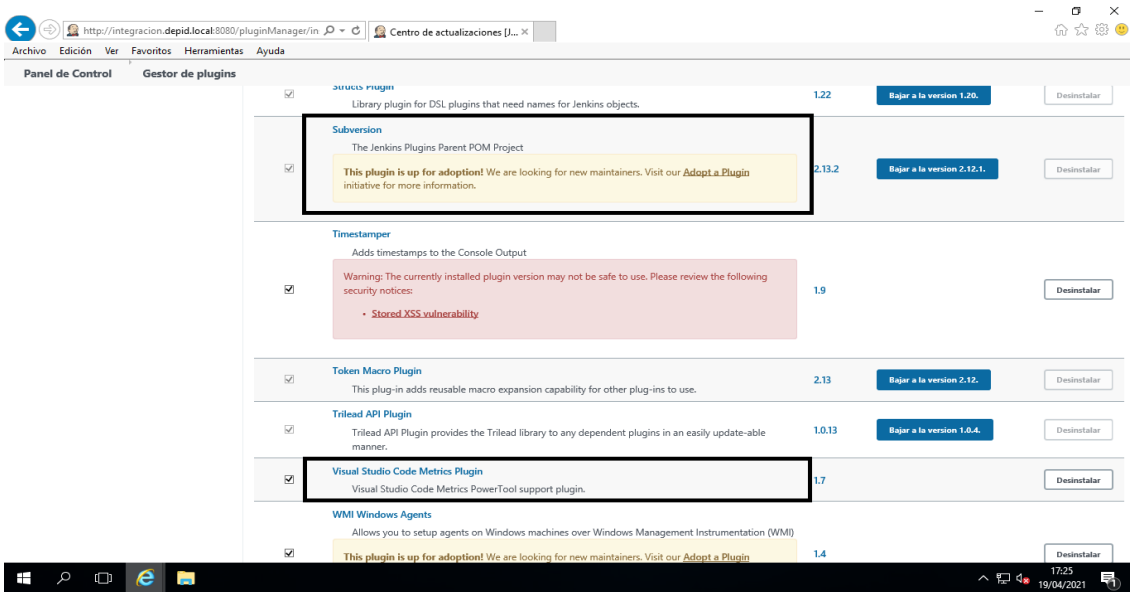


Figura 3: Lista de plugins instalables.

Una vez instalados dichos plugins, configuraremos en las opciones del programa la vinculación con los diferentes programas y aplicaciones del equipo de los que haremos uso en aquellos en los que es necesario una configuración previa.

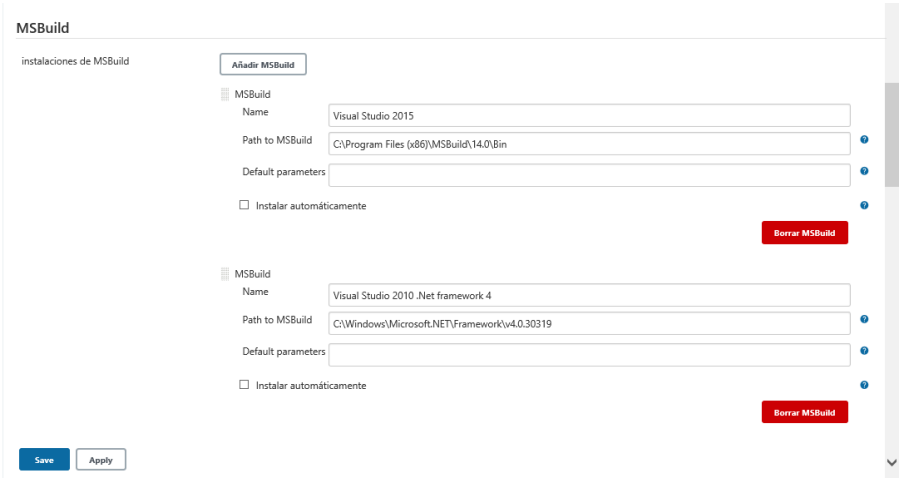


Figura 4: Asociación de Jenkins con el Visual Studio

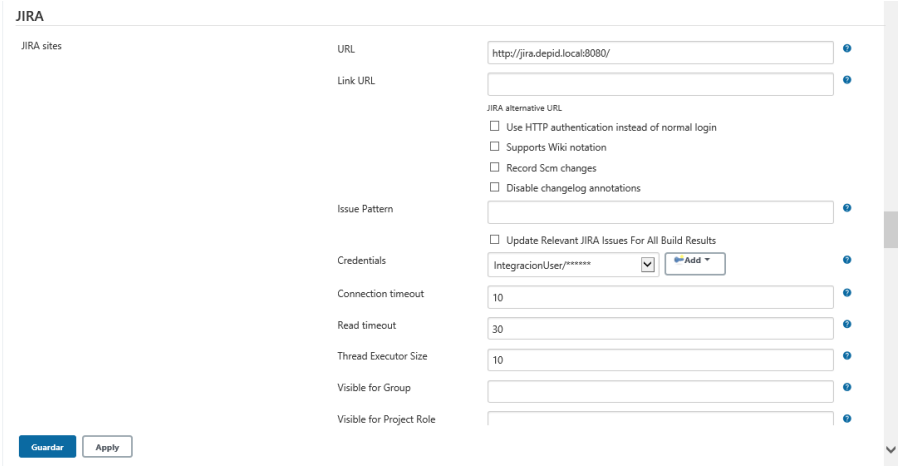


Figura 5: Vinculación de Jenkins con Jira

Por último, estableceremos unas credenciales válidas, en nuestro caso, aludiendo al usuario de integración (IntegracionUser) con las cuales se podrá acceder a los repositorios y servidores necesarios para la realización de las tareas necesarias.

2.2 Estructura del servidor de Jenkins

Una vez realizada toda la configuración previa, será necesario estructurar como trabajaremos dentro de la aplicación de Jenkins.

La jerarquía y estructura del trabajo será clave para que las funciones deseadas se cumplan adecuadamente, ya que con este servidor no solo automatizaremos determinados procesos, sino

que también podremos lanzarlos de forma puntual cuando determinado producto necesite ser revisado o necesite una versión. Eso implicará que cualquier desarrollador debe saber cómo lanzar una compilación de cualquier producto en caso de que sea necesario.

Para estructurar el servidor de Jenkins nos basaremos en la estructura de productos y aplicaciones de la propia empresa. En este sentido, serán 5 las ramas principales del repositorio del que descargar el código fuente, compilar aplicaciones y obtener los productos resultantes, las que definirán la estructura del servidor.

En lo concreto de cada proyecto, debemos recordar las dos funciones principales que nos planteábamos en un inicio: por una parte, la compilación continua, de forma que cada vez que se produzcan cambios en el repositorio, el programa compile para comprobar que todo funciona correctamente; y, por otro, la generación periódica de versiones, para la cual generaremos otro formato de jobs (tareas de Jenkins) que además de compilar el programa, generarán los instaladores necesarios para obtener el producto que se puede trasladar a los clientes.

Por tanto, cada producto tendrá dos jobs: el de compilación simple (ej: XXX_trunk) y el de generación de versión (ej: XXX_trunk_BuildNewVersion.), que se generarán, ante la existencia de cambios en el repositorio, diaria y semanalmente respectivamente.

2.3 Job básico de compilación

El primer tipo de tareas con Jenkins tienen un carácter bastante sencillo, y es que lo único que haremos será bajar el código fuente del repositorio y compilarlo haciendo una llamada al programa del Visual Studio.

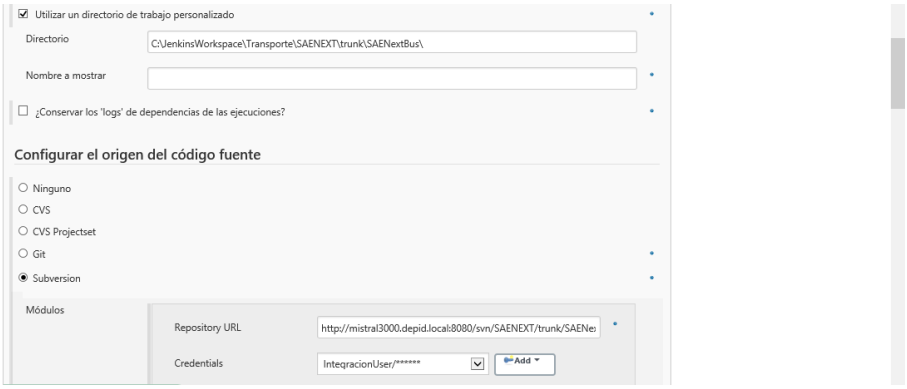


Figura 6: Configuración de un job básico (I)

En esta primera captura podemos ver como generaremos un directorio concreto para el producto trabajado, y, a su vez, configuramos el origen del código fuente con nuestro repositorio de Subversion.

Figura 7: Lanzamiento compilación de la solución de Visual Studio

Y en esta siguiente captura podemos ver como llamar al programa de Visual Studio (configurado previamente para vincular con el programa adecuado de nuestra máquina).

Además de todo esto, podremos configurar otras funciones como la comprobación de prerrequisitos para asegurar que la compilación se da cuando los proyectos previos funcionan, o el desencadenamiento de compilación cuando se lanza una compilación previa en la jerarquía definida.

Más allá de este esbozo, centraremos el resto de la memoria en el trabajo desarrollado con el segundo formato de jobs, que será el que requerirá mayor trabajo en cuanto a la configuración y las funciones buscadas.

2.4 Jobs de generación de versión (I): funciones generales

La base de este formato de jobs será la misma que para los anteriores, por lo que partiremos del último punto del 2.3 para comenzar a plantear la estructura general de estas tareas.

2.4.1 Obtención de número de versión

Un elemento fundamental para la generación automática de versiones es la obtención del número de versión. Si bien escribir este número a mano no supondría mayor problema que mirarlo en el proyecto y renombrar el fichero, al intentar automatizar este proceso será necesario algo más complejo.

Para ello, nos valdremos de los códigos ejecutables NSIS[6], que nos permitirán de una forma sencilla obtener la versión de un fichero tipo .dll (librería) o .exe (ejecutable) y escribir en un fichero de texto la versión obtenida.

```
1  !define File "..\Bin\SAEBasic10.dll"
2  !include "Version.txt"
3  !include LogicLib.nsh
4  !include FileFunc.nsh
5  !insertmacro GetTime
6
7  Name "GetVersion"
8
9  OutFile "GetVersion.exe"
10 SilentInstall silent
11
12 Section
13
14  ## Get file version
15     GetDllVersion "${File}" $R0 $R1
16     IntOp $R2 $R0 / 0x00010000
17     IntOp $R3 $R0 & 0x0000FFFF
18     IntOp $R4 $R1 / 0x00010000
19     IntOp $R5 $R1 & 0x0000FFFF
20     StrCpy $R1 "$R2.$R3.$R4.$R5"
21     StrCpy $R6 "$R2.$R3.$R4.$R5"
22
23  ## Write it to a !define for use in main script
24     FileOpen $R0 "$EXEDIR\Version.txt" w
25     FileWrite $R0 '!define CurrentVersion "$R6"'
26     FileWrite $R0 "$\r$\n" ; we write an extra line
27     FileWrite $R0 '!define CurrentRevision "$R5"'
28     FileWrite $R0 "$\r$\n" ; we write an extra line
29     FileClose $R0
```

Figura 8: Fichero NSI (I)

Este fichero NSI, tal y como se indica en el código, generará un fichero GetVersion.exe que será el que realmente realizará las funciones indicadas abajo: obtener el número de versión referido como $R6 y el número de revisión de Subversion referido como $R5, que serán escritos en el fichero Version.txt. Por tanto, tras la llamada al fichero NSI se llamará al ejecutable y podremos obtener la versión en nuestro fichero de texto.

2.4.2 Release notes y la documentación de versiones

Otra cuestión clave para la generación de versiones será el documento de Release notes, que nos permitirá documentar los cambios realizados por los desarrolladores en cada versión, para poder saber en qué punto se incluyen los diferentes cambios o mejoras. Para ello, aprovecharemos, entre otras cosas, el NSI del apartado anterior para poder escribir en la cabecera del documento el número de versión y la fecha en la que se ha generado.

```
30
31    ## Update RELEASE_NOTES
32       ${GetTime} "" "L" $0 $1 $2 $3 $4 $5 $6
33       FileOpen $R0 "$EXEDIR\RELEASE_NOTES.txt" a
34       ;FileSeek $R0 0-5 SET
35       ;FileSeek $R0 0-5 SET
36       ;FileWrite $R0 "$\r$\n" ; we write an extra line
37       FileWrite $R0 'Version $R6 ($0/$1/$2)'
38       FileWrite $R0 "$\r$\n" ; we write an extra line
39       FileWrite $R0 '**********************************'
40       FileWrite $R0 "$\r$\n" ; we write an extra line
41       ;FileWrite $R0 "$\r$\n" ; we write an extra line
42       FileClose $R0
43
44
45    SectionEnd
```

Figura 9: Fichero NSI (II)

En este punto, encontraríamos una limitación y es que el texto a añadir en el reléase notes se haría de forma manual porque no tenemos un lugar del que obtener dicho texto.

Sin embargo, y tal y como nos planteamos en la introducción, aprovecharemos las opciones que nos ofrece Jira para dicha documentación.

Capítulo 3. Vinculación con el sistema de Jira

3.1 Consideraciones previas

De la misma forma que la aplicación de Jenkins, Jira requerirá de un servidor web sobre el que trabajar. Como se explicaba en el primer capítulo, Jira es una aplicación que permite la asignación de tareas de forma interactiva y ágil, permitiendo tanto repartir el trabajo como documentarlo para poder ver los objetivos conseguidos o por cumplir en cada momento.

Dicho trabajo se dará mediante la creación de pequeñas tareas sobre las que los diferentes desarrolladores irán trabajando (en adelante las llamaremos "issues" por ser el nombre empleado en la propia aplicación). El funcionamiento podría entenderse de la siguiente manera: al detectar una funcionalidad sin cubrir, un error en la aplicación, o cualquier otro problema por solucionar, se abrirá un issue, que será asignado a un desarrollador. Cuando este desarrollador acabe su tarea, la dará por concluida, relatando en el propio cierre del issue los cambios introducidos, cuestión que aprovecharemos tal y como veremos más adelante. Es decir, al acabar la tarea, rellenará un apartado donde se escribirá la información que posteriormente recogeremos para nuestro objetivo.

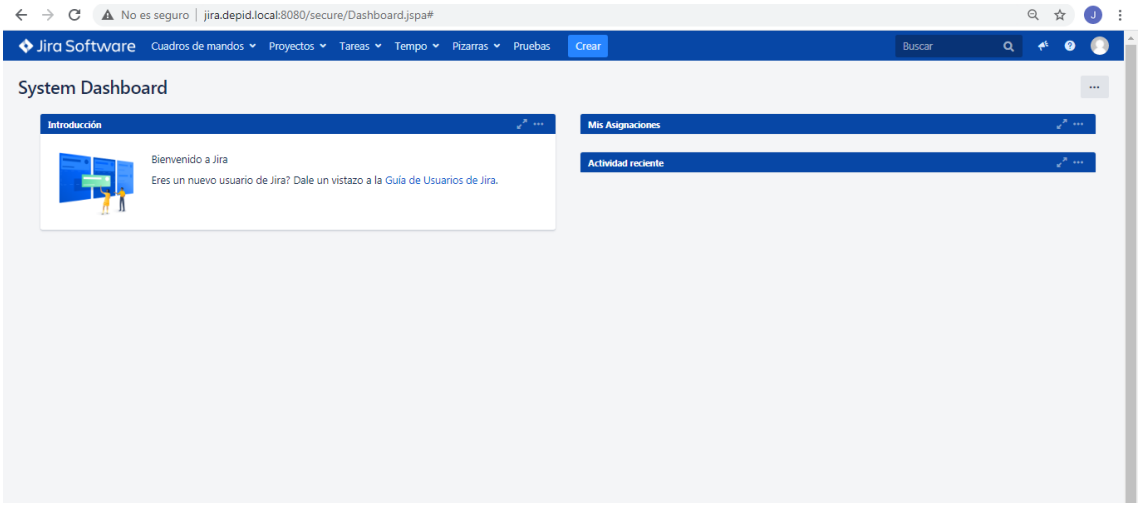


Figura 10: Interfaz principal Jira

Dado que ya en el segundo capítulo integramos Jenkins con Jira, ahora podemos comenzar a trabajar sobre el propio Jira, y una vez realizada la preparación, estudiaremos cómo conseguir desde Jenkins la información deseada.

De cara a los objetivos que perseguimos en nuestro proyecto con la integración de Jira, el objetivo será poder obtener un resumen sobre los cambios que cada issue introduce en sus respectivas aplicaciones, de forma que cuando dicho issue se complete, figure el cambio realizado en nuestro documento de RELEASE_NOTES, fundamental tanto para los desarrolladores como para los clientes que empleen la aplicación.

Teniendo en cuenta los estados en los que un issue puede estar (por iniciar, en desarrollo, por revisar, finalizado…), lo que haremos será incluir un campo en todos los issues denominado Release Notes en el que siempre que se pase al estado "Por revisar" se deberá rellenar con la información necesaria. Además, de cara a poder diferenciar qué issues incluir y qué issues no, se establecerán diferentes campos como el proyecto, subsistema o número SVN, que emplearemos a la hora de filtrar las tareas.

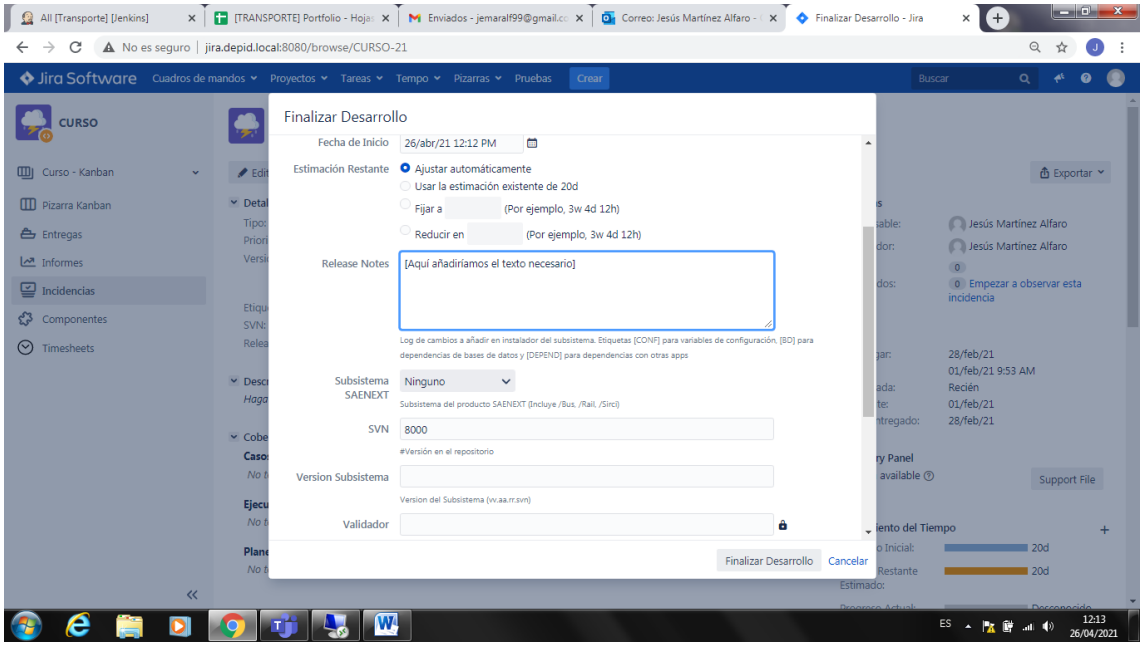


Figura 11: Campo Release Notes

Una vez realizado esto, podemos pasar a Jenkins para continuar con el trabajo.

3.2 Obtención del campo Release Notes

Como comentábamos en el apartado anterior, Jira nos da la opción de establecer una serie de filtros que nos permitirán separar los issues que necesitemos en cada caso. Dicha información, más allá de poder verla en la propia interfaz de Jira, podemos obtenerla en formato json mediante una consulta http, que nos dará todos los datos en bruto. Por tanto, lo primero que deberemos hacer es descargar el plugin de Jenkins de HTTP Request y configurar cómo se realizaría la consulta.

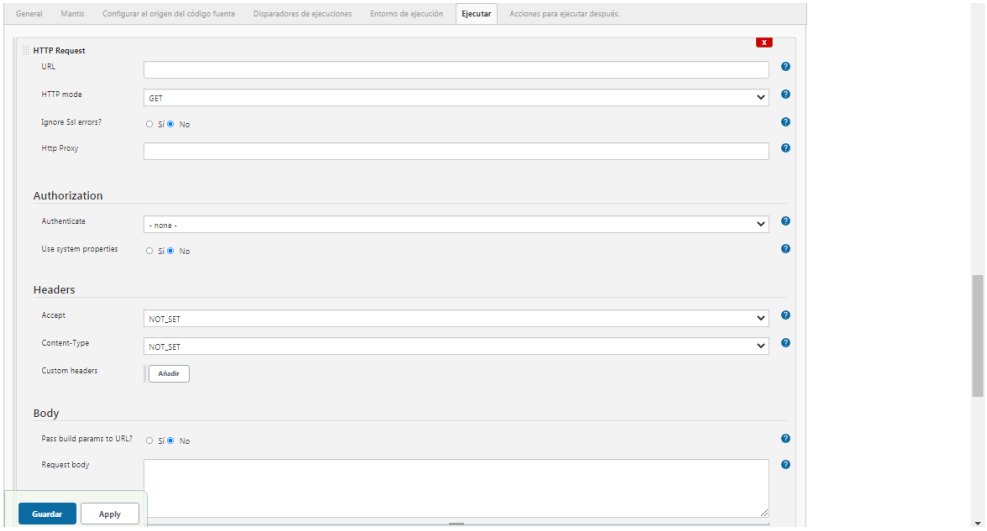


Figura 12: Consulta HTTP (I)

Como podemos ver, habremos de incluir una consulta http en la que podremos elegir el modo, en nuestro caso GET. Además, nos autenticaremos con el usuario de integración acreditado previamente, y estableceremos en las cabeceras el formato de la respuesta como un json codificado con UTF-8. Finalmente, definiremos el fichero de salida como un fichero de texto que utilizaremos posteriormente.

El resultado será un fichero con este aspecto, donde el número de issues variará en función de los filtros indicados, por supuesto.

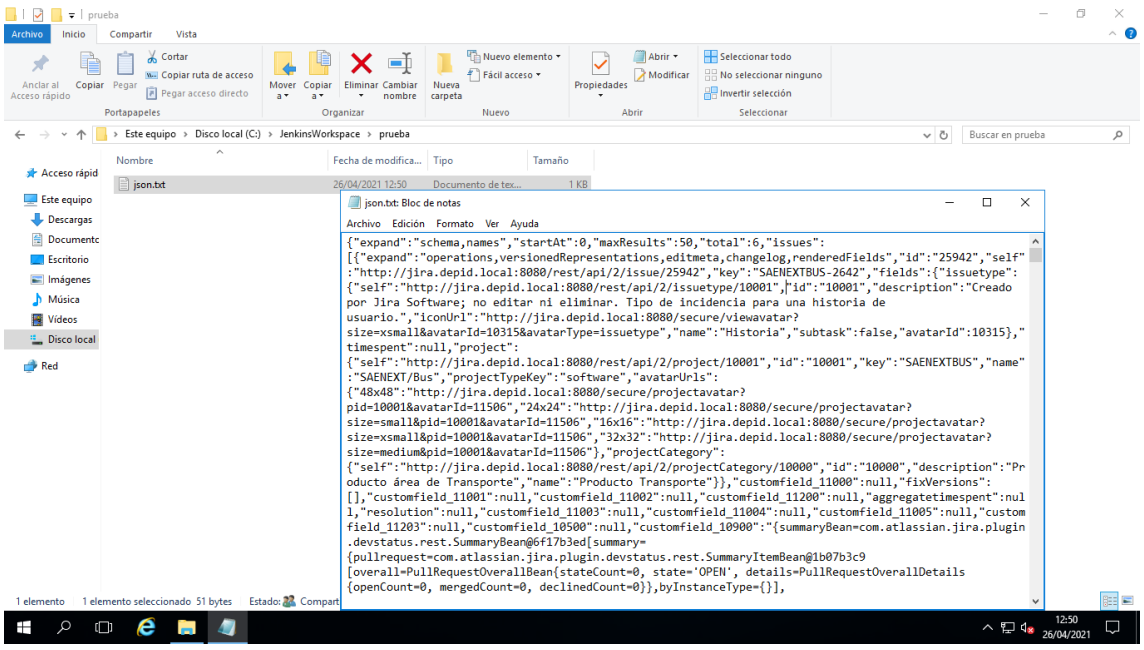


Figura 13: Resultado consulta HTTP

Una vez obtenido el fichero json.txt, podremos observar que entre los diferentes campos, el denominado "customfield_11200" obtendrá la información del campo release notes de Jira, por lo que deberemos extraer esa información para escribirla después en nuestro documento Release_Notes.

Los ficheros en formato JSON se estructuran en base a diferentes niveles, en forma de objetos (JSONObjects) y vectores (JSONArrays). Así, será necesario ir entrando en los objetos/vectores necesarios hasta acceder al dato que nos interesa.

Para ello, diseñaremos una aplicación a la que llamaremos eJIRA, en la que, introduciendo el json.txt, el Release_Notes.txt y Version.txt (este último será un fichero donde tengamos las dos versiones, última generada y la que vamos a generar en este momento), realizaremos todo el proceso para actualizar el Release Notes.

3.3 eJIRA

```
10  public class App {
11
12    public static void main(String[] args) {
13      //OPCION LEER FICHERO TXT
14      File json_arx=null;//fichero de la consulta http
15      File release_arx=null;//release notes para lectura
16      File destino=null; //release notes para escritura
17      File svn_arx=null;//version.txt
18
19      FileReader fr_json=null;
20      FileReader fr_release=null;
21      FileReader fr_svn=null;
22
23      BufferedReader br_json=null;
24      BufferedReader br_release=null;
25      BufferedReader br_svn=null;
26
27      FileWriter fw;
28      BufferedWriter bw;
29
30      try {
31
32        json_arx=new File  (args[0]);
33        fr_json=new FileReader(json_arx);
34        br_json= new BufferedReader(fr_json);//abrimos fichero con resultado de la consulta en jenkins
35
36        release_arx=new File(args[1]);
37        fr_release=new FileReader(release_arx);
38        br_release=new BufferedReader(fr_release); //abrimos el release notes ya existente para leer su contenido
39
40        destino=new File(args[1]);//declaramos el release notes para reescribirlo
41
42        svn_arx=new File(args[2]);
43        fr_svn=new FileReader(svn_arx);
44        br_svn=new BufferedReader(fr_svn);//abrimos el fichero del que extraer las versiones de svn
```

Figura 14: eJIRA (I)

En esta primera captura podemos ver la estructura del eJIRA, que básicamente trabajará con 3 archivos de lectura (json, release y svn) y uno de escritura (destino) para los que definiremos sus respectivos BufferedReader o Writer y FileReader o Writer. Introduciremos los ficheros .txt en la propia compilación del programa. Tal y como se puede intuir, el primer fichero será la información en formato JSON obtenida en la consulta HTTP; el segundo será nuestro RELEASE_NOTES ya existente; y el tercero, el fichero Version.txt donde habremos obtenido, tal y como se relató en el segundo capítulo, el número de versión.

```
46      StringBuilder svn_builder=new StringBuilder();
47      String linea_svn;
48
49      while ((linea_svn=br_svn.readLine())!=null) {
50          svn_builder.append(linea_svn);
51        }
52
53      String svn_string=svn_builder.toString();//obtenemos todo el texto de version.txt
54      StringTokenizer st=new StringTokenizer(svn_string,"\"");//usamos comillas dobles como separador
55      String[] vec_aux=new String[st.countTokens()];
56      int a=0;
57
58      while (st.hasMoreTokens()) {
59        String str=st.nextToken();
60        vec_aux[a]=str;//almacenamos cada elemento separado en un vector de strings
61        a++;
62      }
63      String numero_version="";
64      int svn_ultimo=Integer.parseInt(vec_aux[3]);//cogemos el svn de la ultima version de la 4a linea del vector, transformandolo en int
65      int svn_actual=Integer.parseInt(vec_aux[5]);//cogemos el svn actual de la 6a linea del vector, transformandolo en int
66      if (a>6) {
67
68        numero_version=vec_aux[7];
69      }
70
```

Figura 15: eJIRA (II)

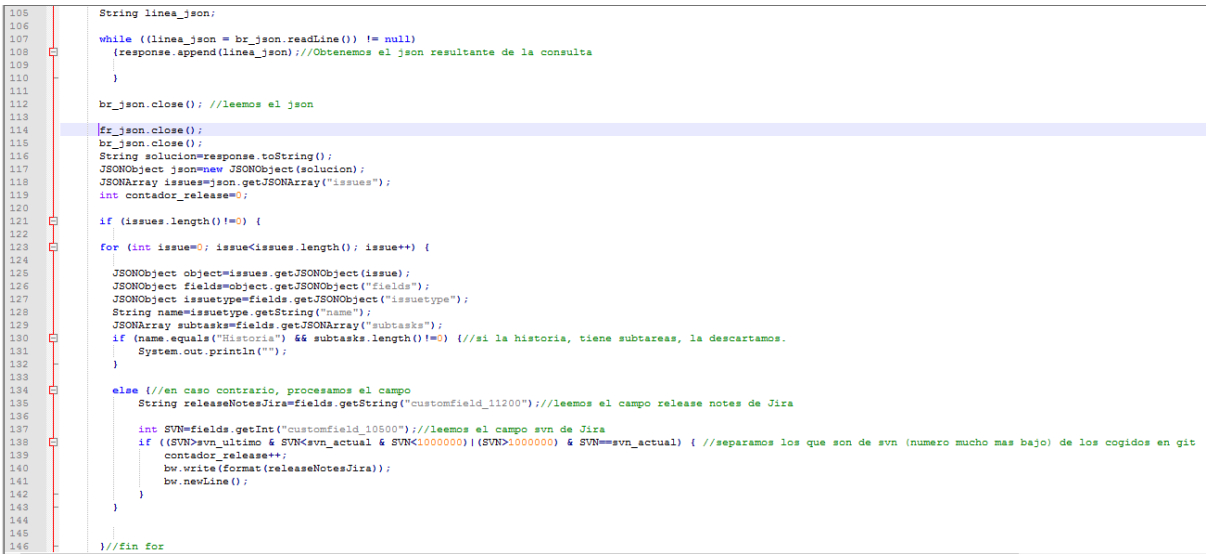
Una vez abierta la lectura de los ficheros, comenzaremos por obtener los números de SVN de la última versión generada y la versión que vamos a generar en esta compilación, lo cual nos permitirá, más adelante, seleccionar qué campos de release notes de Jira coger y cuáles no, tal y cómo podemos ver en esta captura.

```
72
73
74      String linea_release;
75
76      String[] release=new String[10000];
77      int i=0;
78      while ((linea_release=br_release.readLine())!=null) {
79        release[i]=linea_release;//Leemos el release notes ya existente. Lo almacenamos de esta forma para que al reescribirlo la mantenga
80        i++;
81      }
82
83      fr_release.close();
84      br_release.close();
85
86      destino.delete();
87      destino.createNewFile(); //vaciamos el fichero RELEASE_NOTES tras haber guardado su contenido
88
89      fw=new FileWriter(destino.getAbsoluteFile(),true);
90      bw=new BufferedWriter(fw); //el writer del release notes lo abrimos aquí porque necesitamos que no esté declarado al borrar su contenido anterior.
91        if (svn_actual<1000000) {
92        bw.write(release[0]);
93        bw.newLine();
94        bw.write(release[1]);
95        bw.newLine();
96        }
97        else {
98            bw.write("Versión "+ numero_version);
99            bw.newLine();
100           bw.write("*****************************");
101           bw.newLine();
102       }
103
```

Figura 16: eJIRA (III)

Después, obtendremos todo el texto presente en el Release_notes mediante su lectura, y, tras ello, escribiremos la cabecera para escribir debajo el texto obtenido. En este caso, se establecerá una diferencia en función del valor del campo svn_actual, que responde al distinto formato entre las aplicaciones de gitlab y Subversion, pero que en la práctica y para entender esta aplicación no nos afecta. Por último, y dado que la escritura desde java en la clase BufferedWriter la haremos de arriba abajo, guardamos todo el texto previo del Release_Notes.txt para después escribirlo cuando hayamos procesado ya el json.txt.

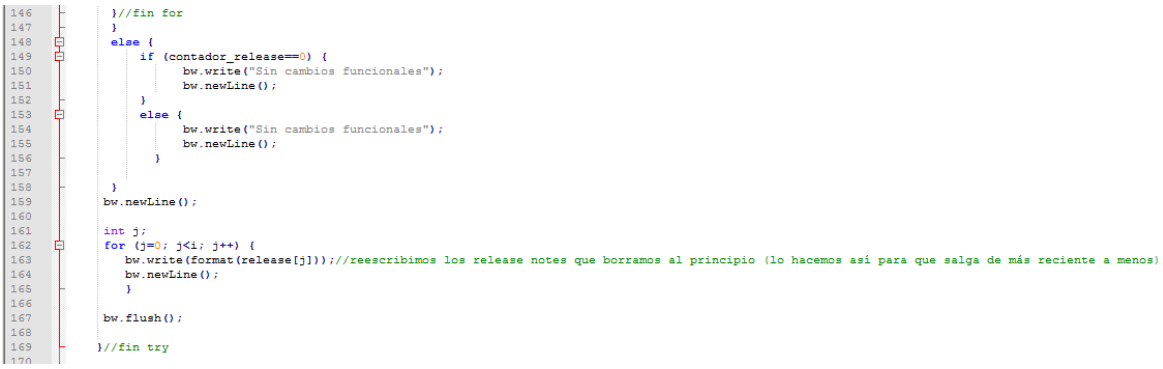
Así, el orden de trabajo sobre el documento será el siguiente: obtendremos todo el release notes previo, almacenandolo en un vector de string; y escribiremos, de arriba a abajo, lo que queremos que aparezca en el nuevo release notes: la cabecera con la versión y la fecha, el contenido de la consulta HTTP y todos los contenidos previos, que habíamos borrado anteriormente para poder ordenar los elementos en base al criterio seleccionado.



```
105     String linea_json;
106
107     while ((linea_json = br_json.readLine()) != null)
108       (response.append(linea_json);//Obtenemos el json resultante de la consulta
109
110       )
111
112     br_json.close(); //leemos el json
113
114     fr_json.close();
115     br_json.close();
116     String solucion=response.toString();
117     JSONObject json=new JSONObject(solucion);
118     JSONArray issues=json.getJSONArray("issues");
119     int contador_release=0;
120
121     if (issues.length()!=0) {
122
123     for (int issue=0; issue<issues.length(); issue++) {
124
125       JSONObject object=issues.getJSONObject(issue);
126       JSONObject fields=object.getJSONObject("fields");
127       JSONObject issuetype=fields.getJSONObject("issuetype");
128       String name=issuetype.getString("name");
129       JSONArray subtasks=fields.getJSONArray("subtasks");
130       if (name.equals("Historia") && subtasks.length()!=0) {//si la historia, tiene subtareas, la descartamos.
131           System.out.println("");
132       }
133
134       else {//en caso contrario, procesamos el campo
135           String releaseNotesJira=fields.getString("customfield_11200");//leemos el campo release notes de Jira
136
137           int SVN=fields.getInt("customfield_10500");//leemos el campo svn de Jira
138           if ((SVN>svn_ultimo & SVN<svn_actual & SVN<1000000)|(SVN>1000000) & SVN==svn_actual) { //separamos los que son de svn (numero mucho mas bajo) de los cogidos en git
139               contador_release++;
140               bw.write(format(releaseNotesJira));
141               bw.newLine();
142           }
143       }
144
145
146     }//fin for
```

Figura 17: eJIRA (IV)

Por último, leeremos el texto de nuestro tercer fichero (json.txt) y en este caso usaremos la clase Json de Java para llegar hasta el objeto "customfield_11200", que será el que nos dará la información del campo release notes de la tarea de Jira, estableciendo previamente algunos filtros en base al tipo de tarea impuestos por el formato de trabajo de la empresa (si es tipo Historia y tiene subtareas dentro de la Historia, no se cogerá el campo). Finalmente, para decidir si cogemos el campo o no, miraremos el valor del campo SVN del issue ("customfield_10500") y si está entre los valores de svn_ultimo y svn_actual, se cogerá.



```
146     }//fin for
147     }
148     else {
149         if (contador_release==0) {
150               bw.write("Sin cambios funcionales");
151               bw.newLine();
152         }
153         else {
154               bw.write("Sin cambios funcionales");
155               bw.newLine();
156           }
157
158     }
159     bw.newLine();
160
161     int j;
162     for (j=0; j<i; j++) {
163        bw.write(format(release[j]));//reescribimos los release notes que borramos al principio (lo hacemos así para que salga de más reciente a menos)
164        bw.newLine();
165        }
166
167     bw.flush();
168
169     }//fin try
```

Figura 18: eJIRA (V)

En caso de que no haya ningún issue que entre en los criterios marcados, escribiremos "Sin cambios funcionales" en el RN para no dejar el hueco en blanco, y, en todo caso, tras acabar esta escritura, se escribirá abajo todo lo almacenado en el vector release (el texto que había previamente).

Hecho todo esto, habremos actualizado el Release_Notes.txt de forma automatizada con esta aplicación, a la cual llamaremos con el programa "compilador.bat" desde la carpeta GeneracionInstalador, en la cual estarán todos los archivos necesarios para que la aplicación funcione.

Figura 19: Llamada a eJIRA

Por último en lo relativo a este capítulo de Jira, dado el formato seleccionado, necesitaremos hacer uso de un fichero que llamaremos auxiliar.nsi, que funcionará de forma prácticamente igual al _Version.nsi, generando un GetVersionAux.exe que nos permitirá escribir la versión actual debajo de la última versión generada, y así obtendremos el Version.txt que empleamos en eJira. Este paso nos permitirá obtener el rango de versiones del que seleccionar la información.

Capítulo 4. Implementación completa de la automatización de versiones

Una vez definida la estructura general de nuestro proyecto, en este capítulo abordaremos la implementación completa de una serie de jobs, empleados para la automatización de versiones de diferentes aplicaciones de la empresa, con una serie de peculiaridades que tendremos que considerar a para poder generar las versiones adecuadamente.

En este caso, debemos considerar que en el desarrollo de todo el trabajo en la empresa se han llegado a añadir al sistema de Jenkins más de 30 aplicaciones diferentes, pero que en lo referido a los conocimientos reportados por cada una de ellas, podemos encontrar estructuras muy parejas entre unas y otras que, de cara a esta memoria, analizaremos de forma conjunta. Por tanto, analizaremos un número de aplicaciones concretas, a partir de las cuales básicamente extrapolamos el trabajo realizado y el formato del job para las aplicaciones restantes.

Así, podremos diferenciar 4 tipos de aplicaciones, variables en número y estructura.

Primero, trabajaremos aquellas cuyo instalador es generado mediante un script NSI. Serán las aplicaciones más usadas (y por tanto más actualizadas) por la empresa, y por ello se comenzará por este grupo, siendo también aquellas que a nivel de automatización se ajustarán mejor al modelo planteado en el capítulo 2.

Para continuar, trabajaremos con el grupo de aplicaciones referidas a los instaladores VDPROJ[7], explicados en su respectivo capítulo. En este caso, emularemos la estructura en la mayoría de los aspectos, si bien será necesario realizar varios ajustes para las peculiaridades del mismo.

En tercer lugar, pasaremos a trabajar con las aplicaciones de Gitlab y relacionadas con el lenguaje de Node JavaScript, que, si bien en este caso cambian diametralmente su estructura, adaptaremos para intentar aprovechar las funcionalidades referidas al Release Notes.

Por último, hablaremos de un último grupo de aplicaciones, sobre las cuales habremos de modificar completamente el trabajo ya que la aplicación se generará en un entorno de Linux, del cual hablaremos en su momento.

4.1 Generación de instaladores con nsi

Para este formato de aplicaciones, el resultado final será un fichero de tipo .msi[8] (instalador) generado a raíz de ficheros NSIS de las propias aplicaciones. Para poder entender de forma más clara cómo funcionará el job en conjunto, nos basaremos en este diagrama, donde veremos una serie de fases que se verán representadas posteriormente en las figuras indicadas con el número:

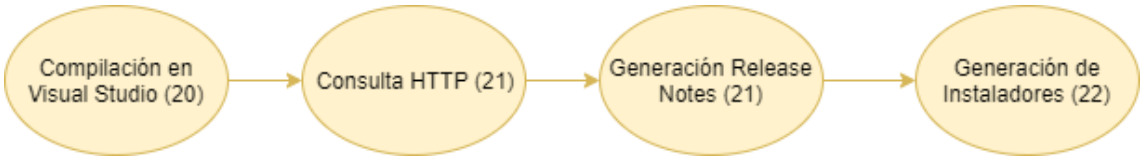


Diagrama 2: Pasos de la generación de versión.

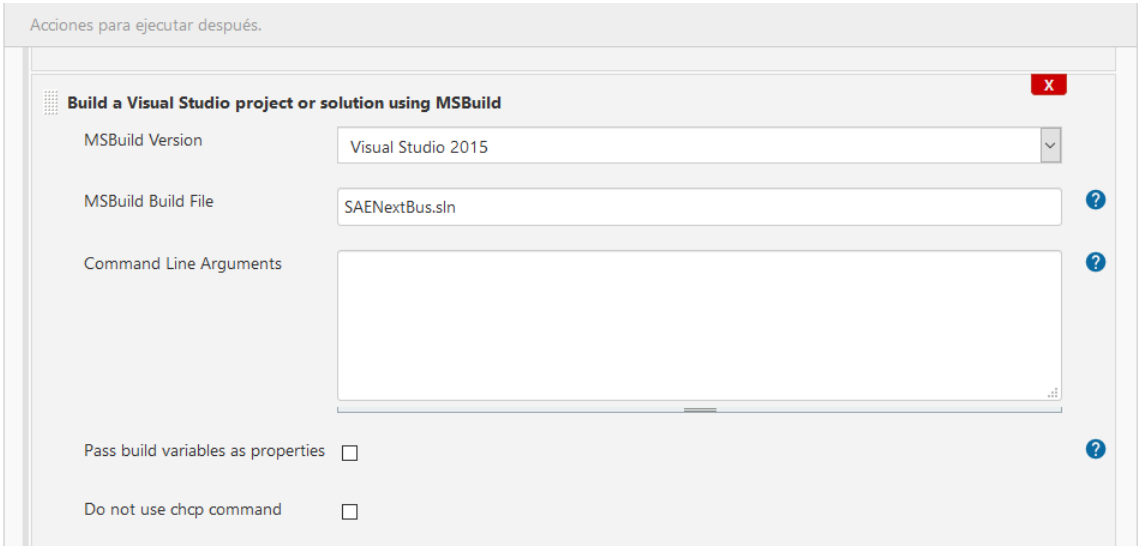


Figura 20: Compilación de la solución de Visual Studio

Lo primero de todo será compilar la propia solución de VisualStudio, que nos permitirá obtener las librerías y ficheros necesarios para los instaladores que queramos generar.

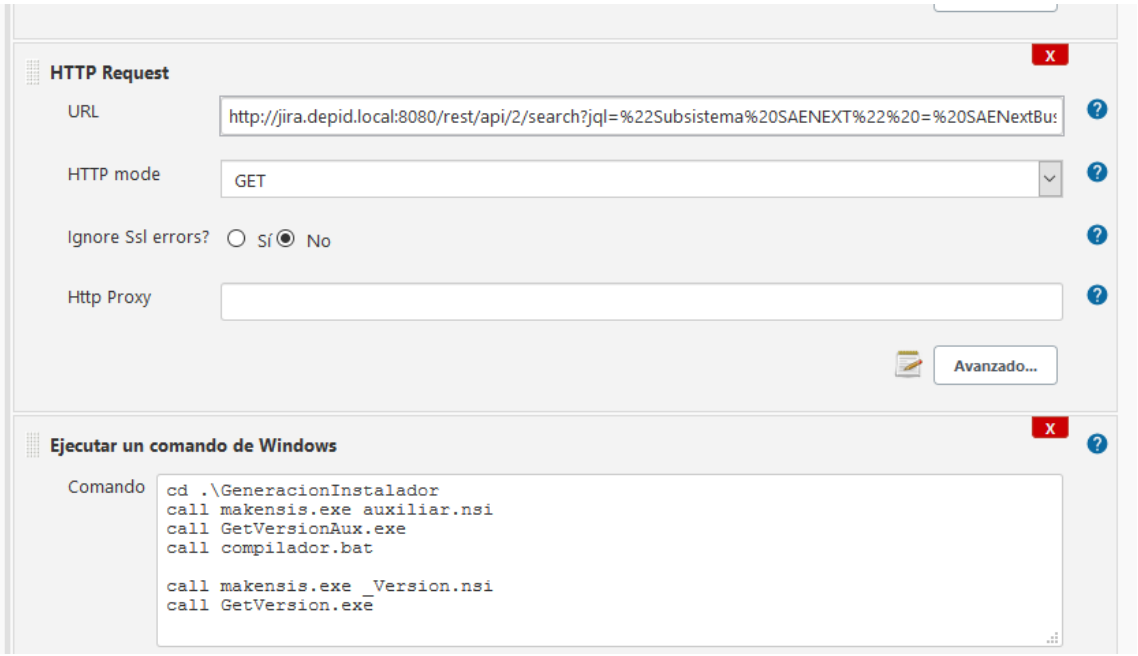


Figura 21: Consulta HTTP y utilización de eJIRA

Tras ello, realizaremos la consulta HTTP con los parámetros necesarios para obtener la información correspondiente a esta aplicación, y, después, llamaremos al fichero .bat que llama a eJIRA, precedido del nsi auxiliar para obtener el rango de valores de SVN sobre los que filtrar la consulta, y seguido del nsi general de versión, para volver a dejar el fichero Version.txt con la versión actualizada. En este paso, el documento de RELEASE_NOTES ya estaría listo.

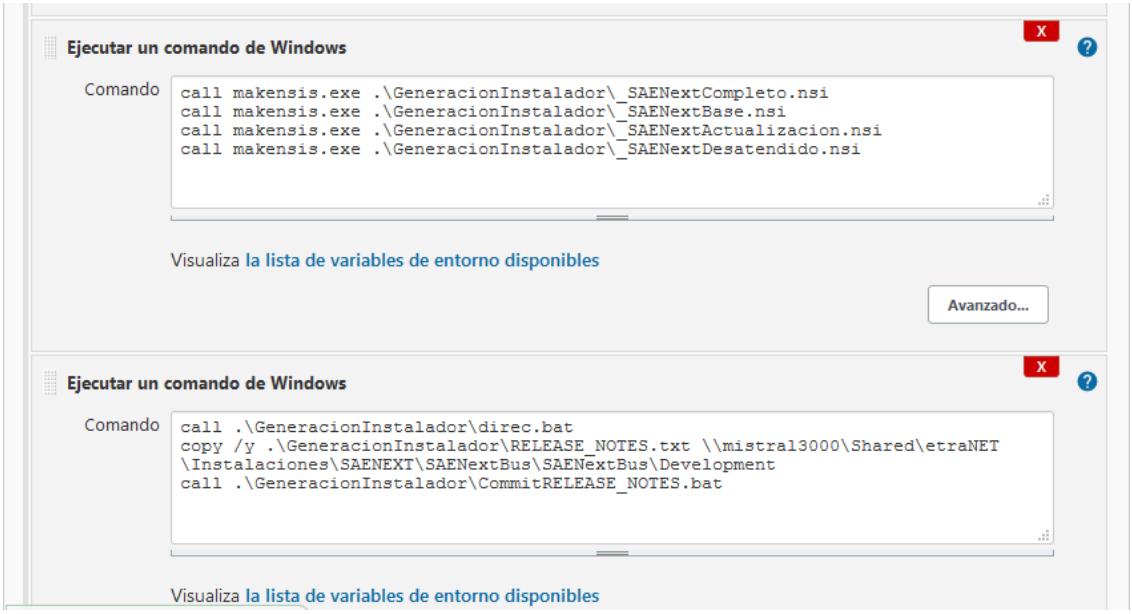


Figura 22: Generación instaladores y subida al repositorio

Para finalizar, llamaremos una serie de scripts NSI, que generarán instaladores diferentes de tipo .exe. En este caso, podemos ver como los instaladores podrán generarse de forma básica, completa, para actualizaciones o para servicios desatendidos, en función de si el cliente ya tiene la aplicación operativa, o no, o si solo quiere unos servicios básicos, por ejemplo. El contenido de estos ficheros NSI nos venía dado y no se han elaborado como el resto de scripts, por lo que no se trabajará su contenido, si bien su función queda clara y en resultado dejará 4 ficheros ejecutables.

Estos cuatro ficheros ejecutables serán los que traslademos al repositorio, en este caso a través de un batch llamado “direc.bat”, que simplemente creará una carpeta con el número de versión donde meter los cuatro instaladores, y ya de ahí trasladados al repositorio, junto con el RELEASE_NOTES. Por último, con el script de commitRELEASE_NOTES.bat actualizaremos el RELEASE_NOTES.txt y el Version.txt en nuestro repositorio svn para que en la siguiente generación de versión se parta desde el punto correcto.

Figura 23: Contenido direc.bat

Con este formato trabajaremos todas aquellas soluciones cuyos instaladores puedan generarse adecuadamente mediante scripts NSI.

4.2 Generación de instaladores con .vdproj

4.2.1 Incompatibilidad con Jenkins

Entrando en un segundo grupo de aplicaciones, nos encontramos aquellas cuyos instaladores se generan mediante proyectos de tipo vdproj (Visual Studio Setup And Deployment Project). Estos proyectos están integrados en las soluciones, y su función principal es generar archivos ejecutables que permitirán la instalación de la aplicación en la máquina del cliente.

A priori, en este caso podríamos seguir el mismo proceso que en las soluciones anteriores, pero ahorrándonos el paso en que llamábamos a los scripts NSI para generar los instaladores.

Sin embargo, al compilar las soluciones desde Jenkins, observaremos un problema que nos obligará a replantearnos la estructura del job.

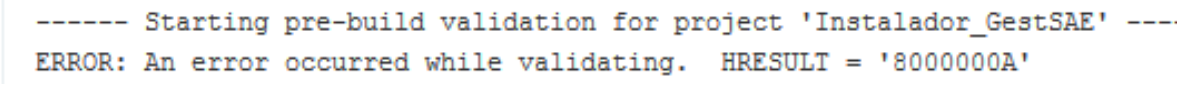


Figura 24: Error en compilación en Jenkins

Sin motivos aparentes, al compilar las soluciones en las que están presentes este formato de proyectos (vdproj), nos encontraremos con este error de validación. Tras investigar sobre ello[9], vemos dos opciones: por una parte, intentaremos añadir una clave en el registro regedit que permita los procesos aparte del build de la solución (EnableOutOfProcBuild) para permitir que estos compilen, ya que los vdproj se compilan siempre al final de la solución y a nivel interno del visual studio parecen funcionar de forma diferente.

Sin embargo, esta solución no permitirá que la compilación desde Jenkins funcione. Sobre esto, ha sido muy complejo encontrar información sobre las causas del error, si bien se puede imaginar que se deba a una cuestión de permisos de la propia aplicación sobre la máquina virtual, aunque no hemos conseguido solucionarlo.

Centrándonos en el objetivo de conseguir automatizar la compilación, recurriremos a un método que nos permita que sea Jenkins quien ejecute la compilación sino conseguir que se lance desde la propia máquina, aunque sea un proceso iniciado desde Jenkins, para así poder evitar ese potencial problema de permisos que podría estar ocurriendo. Para ello, haremos uso del programador de tareas.

El programador de tareas nos permitirá llamar un fichero .bat en el que estaremos lanzando la compilación de la solución desde la ventana de comandos. A su vez, desde Jenkins haremos la llamada a esta tarea, que iniciará dicho proceso y que nos permitirá compilar la solución adecuadamente.

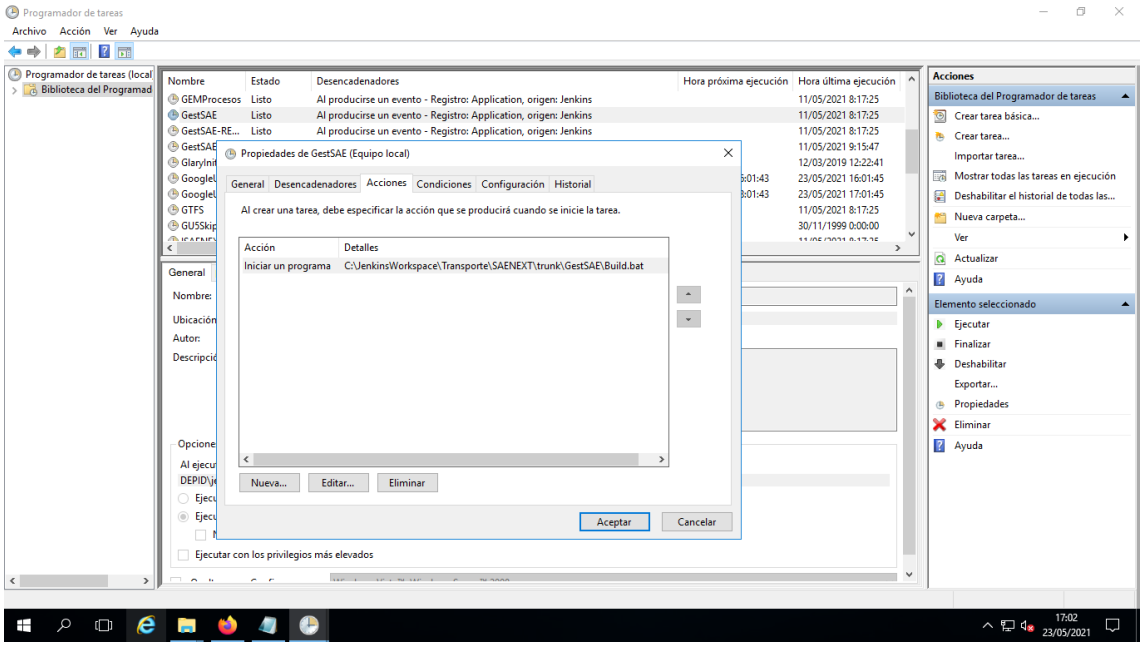


Figura 25: Ejemplo tarea

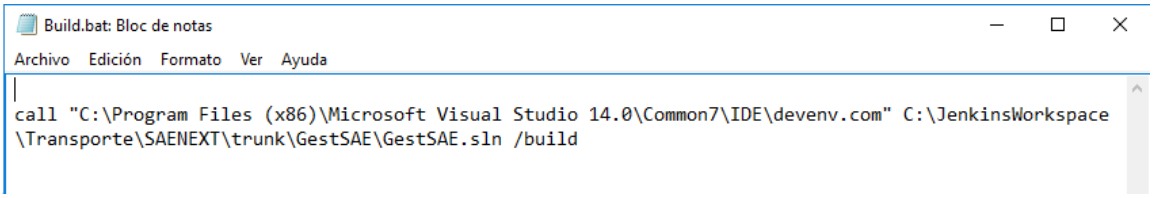


Figura 26: Lanzamiento de compilación desde el programador de tareas.

Una vez comprobado que este método nos funciona, habremos encontrado una manera de solventar la limitación que nos planteaba Jenkins.

4.2.2 Versión del ejecutable

Una vez solucionada la propia compilación, nos encontramos con otro problema derivado del uso de estos archivos .vdproj, y es que, hasta ahora, los desarrolladores de cada aplicación modificaban manualmente un campo del proyecto referido a la versión, que sería después el número que aparece en la interfaz de la aplicación cuando el cliente va a hacer uso de ella.

De esta forma, será necesario buscar una forma en la que cambiar este número antes de compilar la solución (en este caso mediante el programador de tareas) para que la versión se indique correctamente. Para ello, generaremos una nueva aplicación denominada CambiaVersion.java, que sencillamente modificará el fichero vdproj de la misma manera que podríamos modificar un documento de texto, para cambiar la versión por el valor obtenido en Version.txt previamente.

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.FileReader;
import java.util.StringTokenizer;


public class CambiaVersion{
    public static void main(String[] args) throws Exception {
        File arx=null;
        File destino=null;
        File version=null;

        FileReader fr_arx=null;
        BufferedReader br_arx=null;
        FileReader fr_version=null;
        BufferedReader br_version=null;

        FileWriter fw;
        BufferedWriter bw;

        try {
            arx=new File(args[0]);
            destino=new File(args[0]);
            version=new File(args[1]);

            fr_arx=new FileReader(arx);
            br_arx= new BufferedReader(fr_arx);//abrimos fichero con resultado de la consulta en jenkins

            fr_version= new FileReader(version);
            br_version= new BufferedReader(fr_version);
```

Figura 27: CambiaVersion.java (I)

Aquí podemos ver como el programa trabajará en base a dos archivos que le insertemos al compilar: el vdproj en cuestión y Version.txt

```
        StringBuilder svn_builder=new StringBuilder();
        String linea_svn;

        while ((linea_svn=br_version.readLine())!=null) {
            svn_builder.append(linea_svn);
        }

        String svn_string=svn_builder.toString();//obtenemos todo el texto de version.txt
        StringTokenizer st=new StringTokenizer(svn_string,"\"");//usamos comillas dobles como separador
        String[] vec_aux=new String[st.countTokens()];
        int a=0;

        while (st.hasMoreTokens()) {
            String str=st.nextToken();
            vec_aux[a]=str;//almacenamos cada elemento separado en un vector de strings
            a++;
        }

        String svn_actual=vec_aux[7];//cogemos el svn actual de la 8a linea del vector
        System.out.println(svn_actual);

        StringBuilder arxi=new StringBuilder();
        String linea;

        String[] vdproj=new String[10000];
        int i=0;
        while ((linea=br_arx.readLine())!=null) {
            vdproj[i]=linea;//Leemos el release notes ya existente. Lo almacenamos de esta forma para que al reescribirlo la mantenga
            i++;
        }
```

Figura 28: CambiaVersion.java (II)

Tras ello, leeremos del Version.txt el valor de svn a introducir en el vdproj, y leeremos todas las líneas de código que componen el vdproj, para posteriormente reescribirlo, modificando específicamente la línea que hace referencia a "Product Version".

```
            fr_arx.close();
            br_arx.close();
            destino.delete();
            destino.createNewFile();

            fw=new FileWriter(destino.getAbsoluteFile(),true);
            bw=new BufferedWriter(fw);

            for (int j=0; j<i;j++) {
                if (vdproj[j].contains("\"ProductVersion\"")) {
                    String linea_mod="\"ProductVersion\" = \"8:" + svn_actual + "\"";
                    bw.write("       "+linea_mod);
                    bw.newLine();
                    System.out.println(linea_mod);
                }
                else {
                    bw.write(vdproj[j]);
                    bw.newLine();
                }
            }
            bw.flush();
        }
        finally {
            System.out.println("ok");
        }

        }
    }
```

Figura 29: CambiaVersion.java (III)

En esta última captura veremos cómo se realiza la modificación del campo de versión, y de esta forma habremos conseguido modificar la versión del vdproj de forma automática.

4.2.3 Estructura del job

Para acabar con este grupo de aplicaciones, simplemente generaremos los Jobs con una estructura muy similar a la del anterior grupo añadiendo los cambios introducidos previamente. En este caso, el procedimiento será el siguiente:

Diagrama 3: Pasos de la generación de versión

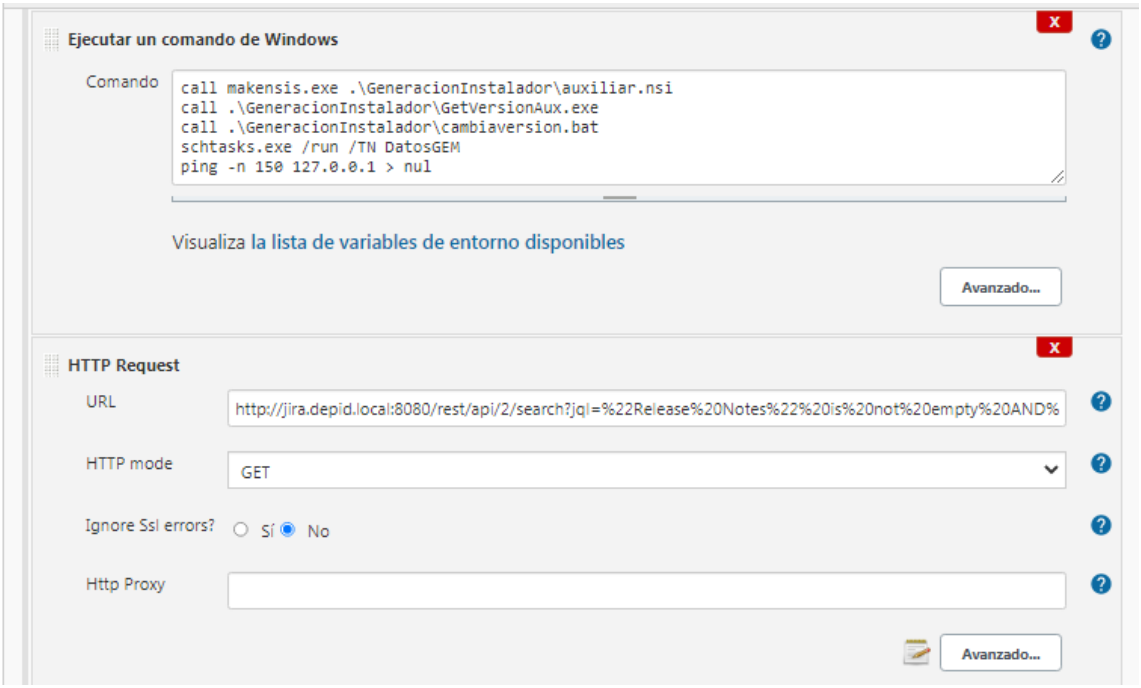


Figura 30: Job (I)

Así, empezaremos obteniendo la versión SVN con el auxiliar.nsi, y, tras ello, llamaremos a cambiaversion.bat, un script que simplemente llamara el CambiaVersion.java explicado en el apartado anterior, introduciéndole el fichero .vdproj y el Version.txt.

Tras ello, llamaremos a la tarea generada (en este caso llamada DatosGEM), y nos encontraremos con una nueva cuestión a solucionar, ya que, al llamar la tarea, el compilador de Jenkins continuaría de forma paralela mientras se genera la solución, y, por tanto, nos encontraríamos un problema para que el instalador esté generado cuando es necesario.

Por ello, realizaremos un ping a una dirección cualquiera atrasando durante el tiempo que consideremos (en este caso 150 segundos) la compilación, para que de tiempo a que la tarea se resuelve y el job pueda continuar.

Tras ello, seguiremos los pasos anteriores, realizando la consulta HTTP, llamaremos de nuevo a eJIRA para generar el Release Notes, y haremos uso de un nuevo script (version.bat) antes de obtener el instalador definitivo.

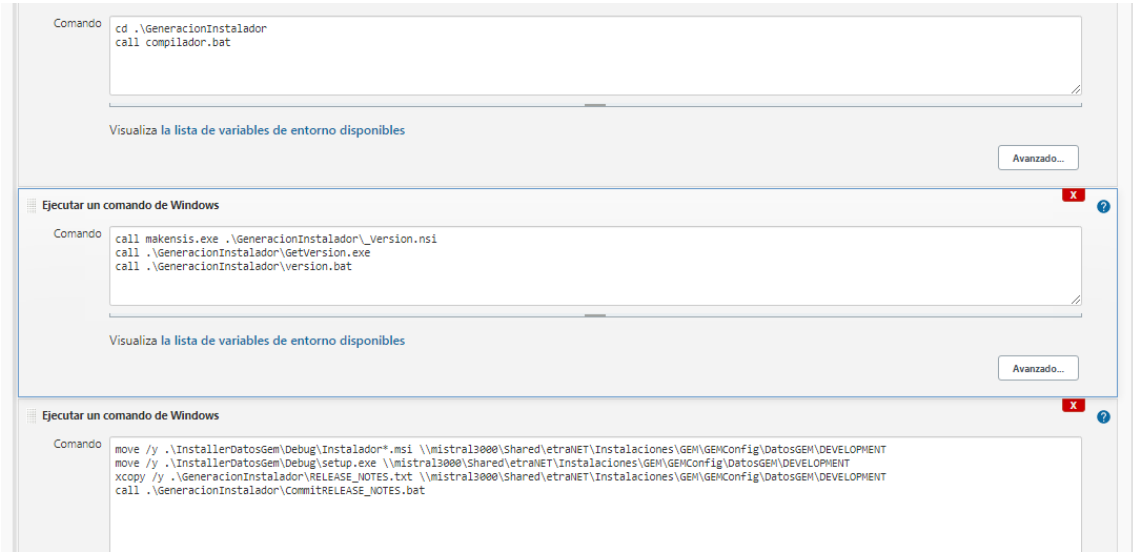


Figura 31: Job (II)

Este script version.bat nos servirá para sustituir una de las funciones que cumplían los NSIs del formato de aplicaciones anterior, ya que en estos se insertaba el número de versión en el nombre del instalador, y, a través de este script realizaremos esta labor.

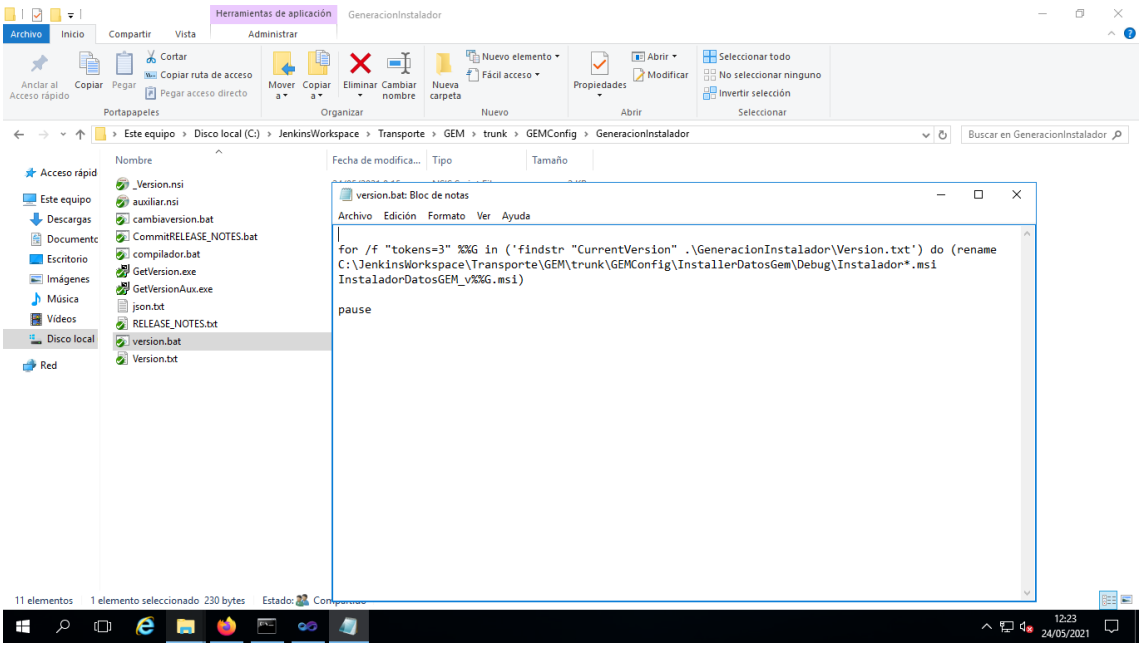


Figura 32: version.bat

Tras ello, simplemente trasladaremos el instalador, el RELEASE_NOTES y otro archivo denominado setup.exe a nuestro repositorio de aplicaciones, y haremos un commit para dejar todo listo para la siguiente compilación.

Con este segundo formato podremos integrar en Jenkins el segundo grupo de aplicaciones, que dejará listo casi todo el servidor. Sin embargo, el desarrollo de la empresa hacia otras líneas de producción ha hecho que, además de emplear subversion como repositorio, se lleve trabajando durante un tiempo con otras aplicaciones en node javascript repositadas en gitlab, por lo que deberemos realizar una serie de cambios para poder integrar dichas aplicaciones en Jenkins.

4.3 Aplicaciones en Gitlab

4.3.1 Integración de Gitlab con Jenkins

Antes de comenzar a trabajar sobre las aplicaciones, será necesario integrar, tal y como hicimos con el repositorio de subversion o con el servidor de Jira, el servidor de Gitlab[10] con el de Jenkins.

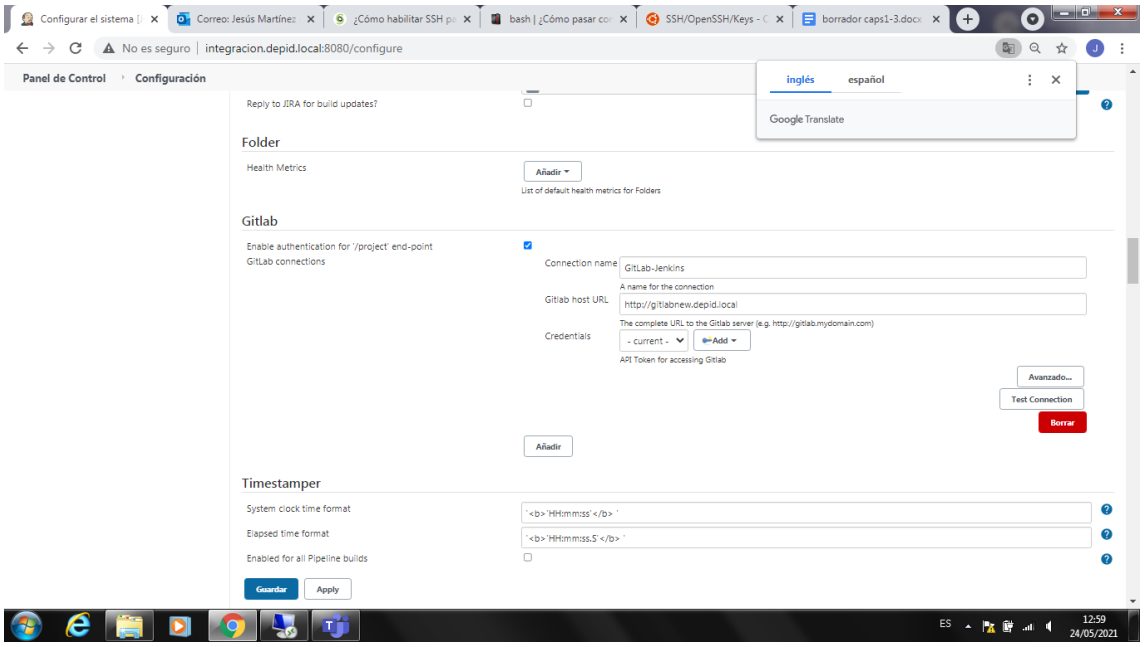


Figura 33: Vinculación Gitlab-Jenkins

Para ello, simplemente instalaremos el plugin de Gitlab en Jenkins, conectaremos con el servidor de gitlab correspondiente, y, a su vez, desde el propio Gitlab autorizaremos la conexión generando una clave mediante la cual después accederemos desde Jenkins a Gitlab.

4.3.2 Aplicaciones

Dentro de las aplicaciones de Gitlab, será necesario distinguir dos tipos de aplicaciones: aquellas trabajadas dentro del propio sistema operativo de la máquina virtual, que serán las de lenguaje Node JavaScript[11], y aquellas que se trabajen en el entorno de Linux[12]. El factor común será que están repositadas en Gitlab, pero el trabajo entre ellas no tendrá apenas factores en común.

4.3.2.1 Aplicaciones NodeJS

Una vez hecha la vinculación de Gitlab con Jenkins, podremos descargar el repositorio en nuestra máquina y comenzar a trabajar. En este caso, hemos de tener en cuenta que el formato de las aplicaciones se aleja totalmente de lo trabajado hasta ahora, ya que ni lenguaje de programación, ni compilador, ni formato de instalador, coincidirán.

Sin embargo, y con el objetivo de poder reutilizar la estructura planificada y la generación del RELEASE_NOTES mediante eJIRA, adaptaremos la programación a este formato. Para ello, hemos de tener en cuenta que las aplicaciones de Node JS ya vienen con una serie de scripts elaborados previamente por los desarrolladores de estas aplicaciones que generan el instalador de forma manual, por lo que nuestro trabajo se centrará en automatizar este script, adaptándolo a lo que hemos producido previamente para ahorrar trabajo y optimizar la estructura del conjunto de aplicaciones.

Para ello, generaremos un script por cada aplicación denominado generaversion_x.bat, siendo x la aplicación correspondiente.

La principal cuestión, de nuevo, y dado que el script release.bat que nos viene dado ya nos generará la carpeta comprimida que buscamos, será la generación correcta del Release Notes.

En este caso, deberemos tener en cuenta dos cambios: por una parte, la numeración por versiones de este conjunto de aplicaciones cambia de formato, ya que, en lugar de seguir un formato secuencial en el último número como hacían las de Subversion (1.YY/MM/SVN) se harán en formato secuencial mes a mes, reiniciándose la cuenta mensualmente (1.YY/MM/Anterior+1), y teniéndose en cuenta que se generan versiones, de forma habitual, cada fin de semana.

Por otra parte, esto hará que a la hora de vincular el sistema con Jira, necesitemos replantearnos el establecimiento del campo SVN, para que podamos seguir estableciendo un rango del que seleccionar los campos Release Notes que después aparecerán en nuestro archivo.

Tal y como hemos visto en los apartados anteriores, este formato de aplicaciones seguirá los siguientes pasos:

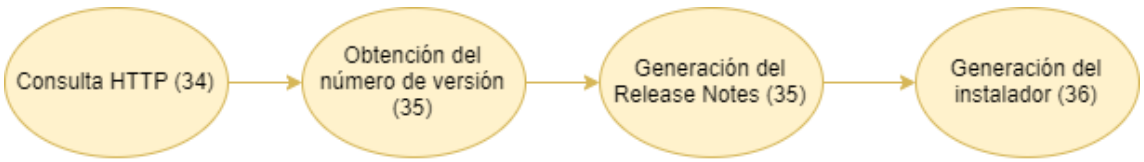


Diagrama 4: Estructura de los jobs de gitlab.

Siguiendo con lo indicado en el diagrama, comenzaremos haciendo una consulta HTTP para obtener la información necesaria para el release notes, y, tras ello, llamaremos a la tarea, que a su vez llamará al script que definiremos a continuación y que realizará el resto de tareas necesarias.

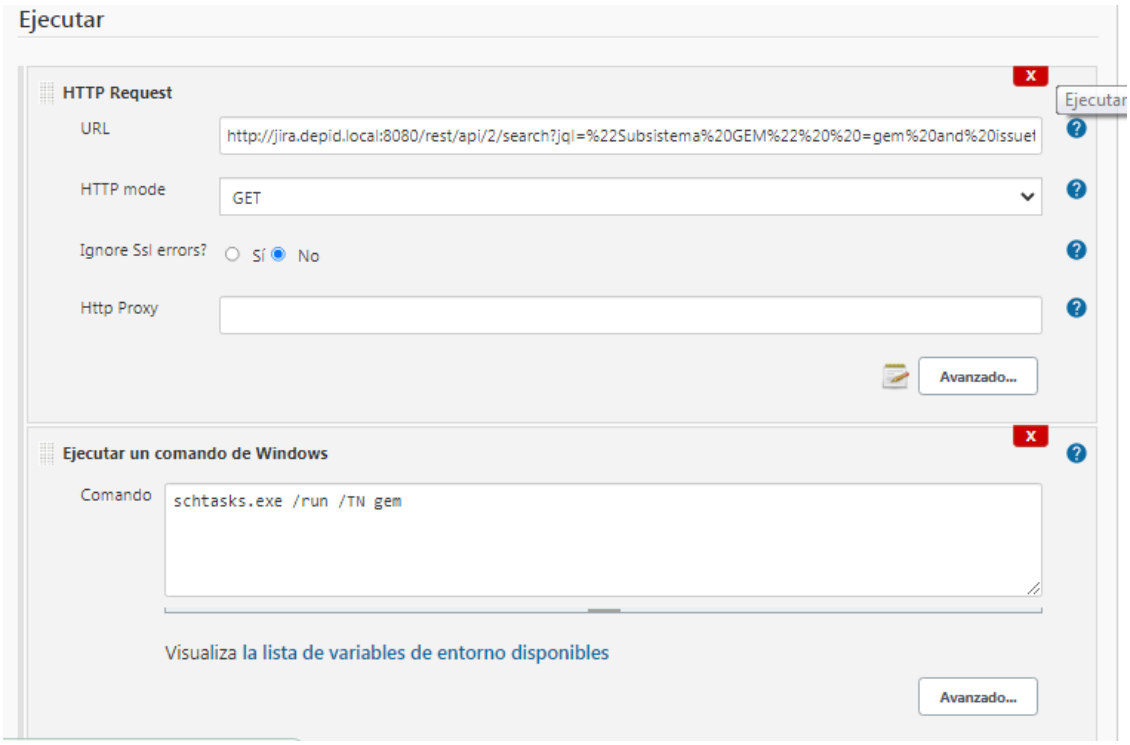


Figura 34: Job app gitlab.

Teniendo en cuenta las consideraciones realizadas al inicio del apartado, nuestro generaversion.bat será tal que así:

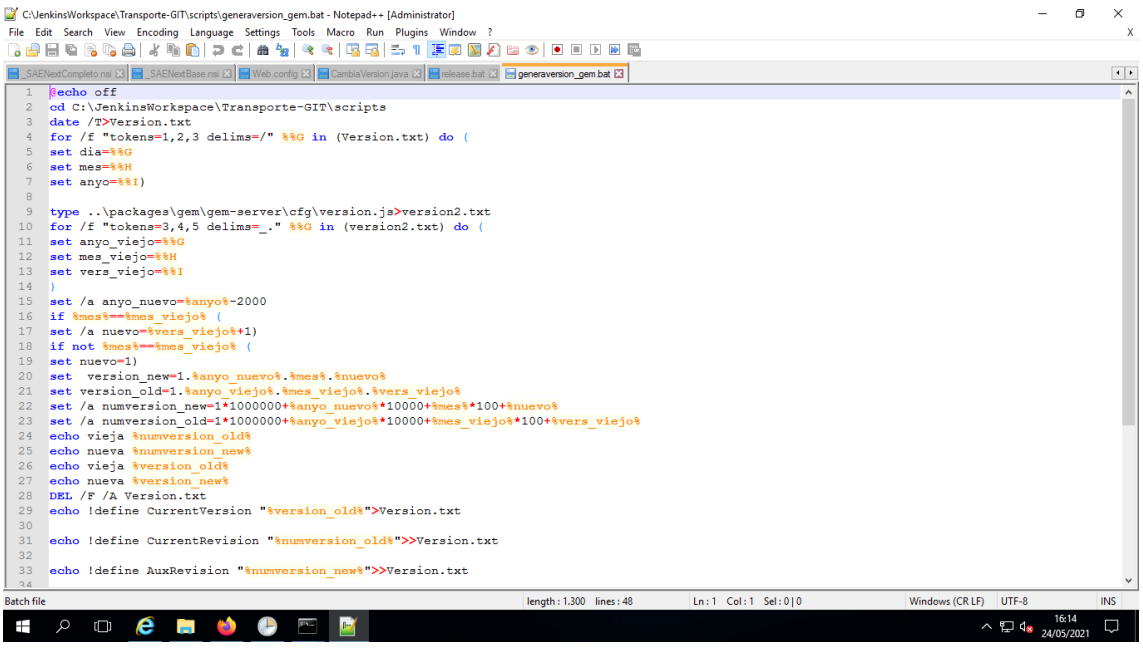


Figura 35: GeneraVersion.bat (I)

En esta primera parte, obtendremos la fecha actual y la fecha reflejada en la última versión generada (almacenada en version.js), y, comparandolas para averiguar si se ha producido un cambio de mes (en cuyo caso reiniciaríamos la cuenta) o sí la última versión se generó el mismo mes, en cuyo caso el último dígito del número de versión se obtendría simplemente sumando 1.

Además, obtendremos un número de versión que será el que luego comparemos en Jira, pudiendo así filtrar adecuadamente.

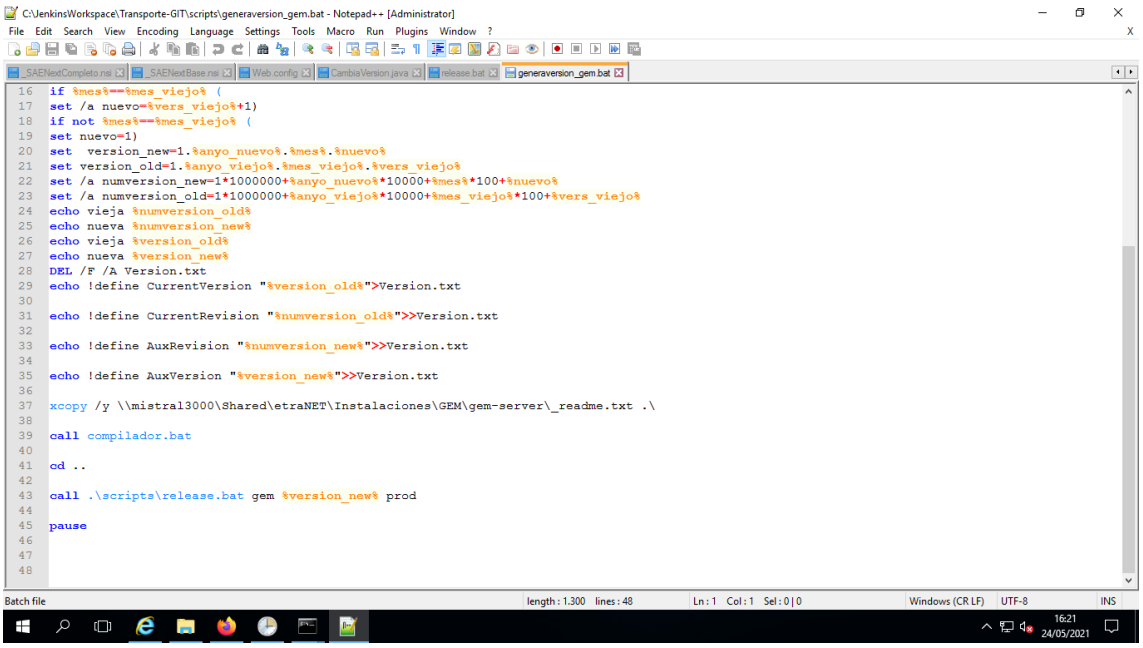


Figura 36: GeneraVersion.bat (II)

Tras ello, borraremos el contenido de Version.txt (que habíamos usado previamente como archivo auxiliar) y escribiremos en él los valores asemejándonos al formato empleado en las anteriores aplicaciones, para que posteriormente pueda usarse el Version.txt en eJIRA de la misma manera que en el resto de apps. Seguidamente, llamaremos a eJIRA, y, para acabar, con el release notes (en este caso _readme.txt) listo, llamaremos el release.bat para que se genere la solución: una carpeta comprimida que será el instalador a exportar en este caso.

En este caso, el propio release.bat al que hacemos referencia en nuestro código se encargará de trasladar el resultado de la compilación al lugar correspondiente, por lo que nuestro job estaría finalizado.

4.3.2.2 Aplicaciones de Linux

Para concluir con el último grupo de aplicaciones integradas en Jenkins, abordaremos aquellas dependientes de Linux. En este caso, nos encontramos aplicaciones que, repositadas en Gitlab tal y como las anteriores, requieren una compilación dentro del entorno de Linux debido a que es donde fueron desarrolladas. Para ello, emplearemos una máquina virtual de Linux existente en la empresa, sobre la que algunos desarrolladores trabajan, para conectarnos a la misma y trabajar ahí.

Siendo conscientes de que nuestra máquina de integración funciona totalmente en el sistema operativo de Windows, será necesario buscar un método en el que conectar con la máquina virtual de Linux, poder compilar allí la aplicación, obtener el instalador resultante, y poder trasladarlo a la máquina de integración sobre la que trabajamos para poder trasladarla a los clientes posteriormente. Además, no sólo necesitaremos acceder a la máquina de Linux manualmente, sino automatizar dicho proceso de acceso y trabajo en la máquina virtual desde Jenkins, ya que el motivo fundamental de este trabajo pasa por evitar un trabajo manual de los desarrolladores para que nuestro programa lo realice de forma automática.

En este apartado emplearemos, fundamentalmente, dos nuevos protocolos: por un lado, aprovecharemos SSH[13] para realizar la conexión con la máquina de Linux, y, por otro lado, usaremos SCP[14] , protocolo que nos permitirá la transferencia segura de archivos entre un host remoto y un host local. Estos pasos darán lugar a la siguiente estructura:

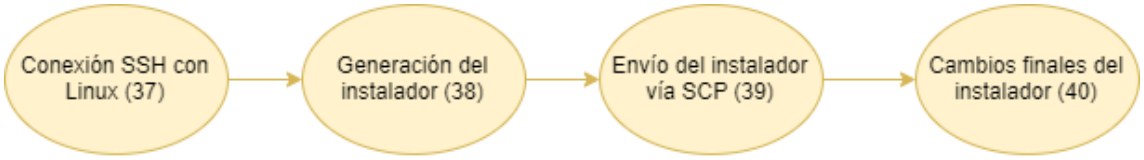


Diagrama 5: Estructura de los jobs de apps de Linux.

Una vez aclarado cómo se daría el proceso de forma manual, necesitaremos aplicarlo a Jenkins. Para ello, primero, instalaremos el plugin relativo a SSH, que nos permitirá realizar una conexión permanente entre la máquina de integración y la máquina de Linux, de forma que podremos realizar los comandos necesarios dentro de la máquina de Linux desde Jenkins.

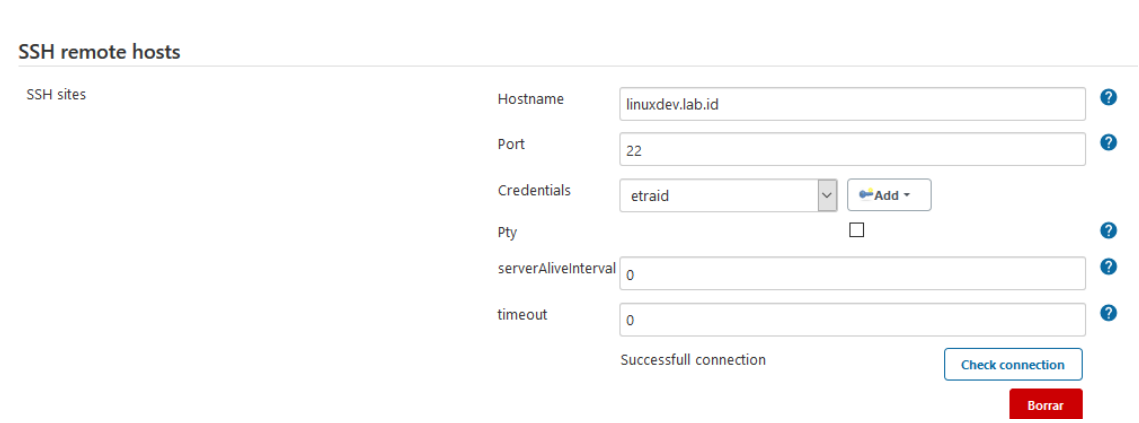


Figura 37: Conexión SSH desde Jenkins

Una vez hecho esto, habremos resuelto la generación del instalador, ya que bastará con ejecutar un comando, del cual no detallamos nada más puesto que no ha formado parte del trabajo sino que nos venía dado de los desarrolladores de la aplicación. Esto nos servirá para generar una carpeta comprimida, que será el resultado de la compilación y lo que querremos trasladar a la máquina de Jenkins después.

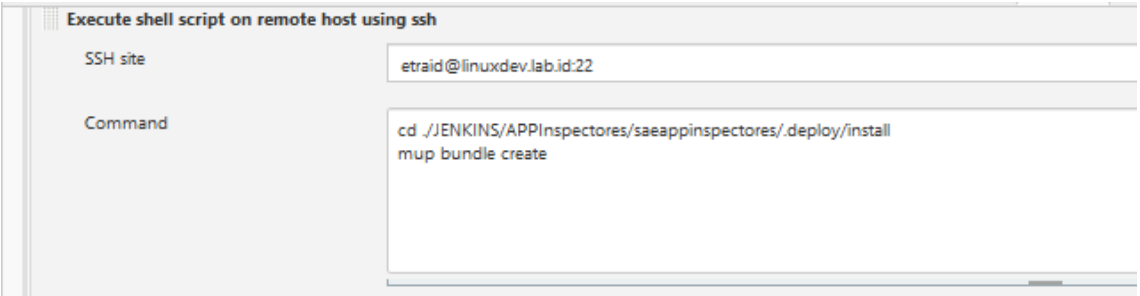


Figura 38: Creación del instalador en linux

Una vez generado el bundle, necesitaremos transferirlo a nuestra máquina de trabajo, para lo que emplearemos SCP: un protocolo de transferencia segura de archivos entre sistemas remotos.

En cuanto al comando SCP, encontraremos un problema en cuanto a la automatización, y es que el propio comando requiere de una clave a la hora de ejecutarlo, que tendría que introducirse manualmente tras ejecutar el comando SCP y que por tanto impediría el proceso tal y como lo queremos.

Para solucionar este problema, estableceremos una clave SSH asimétrica[15] entre las dos máquinas, para evitar ese proceso de autenticación y que por tanto al ejecutar el propio comando SCP ya haya autorización para transferir el archivo.

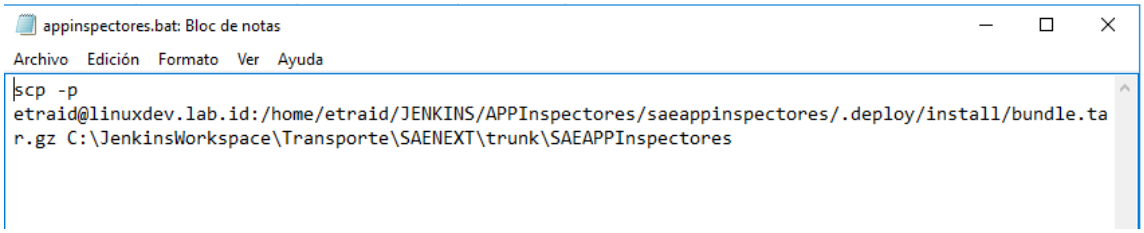


Figura 39: Comando SCP para transferir la app.

Una vez realizado esto, y tras emplear simplemente el comando de transferencia SCP, obtendremos un fichero comprimido denominado bundle.tar.gz, el cual, necesariamente, deberemos descomprimir, para poder acceder al fichero de versión interno, del que extraer la versión, y poder comprimir de nuevo la solución, cambiando su nombre para incluir este número de versión. Todo esto lo realizaremos en otro script, que denominaremos rename.bat.

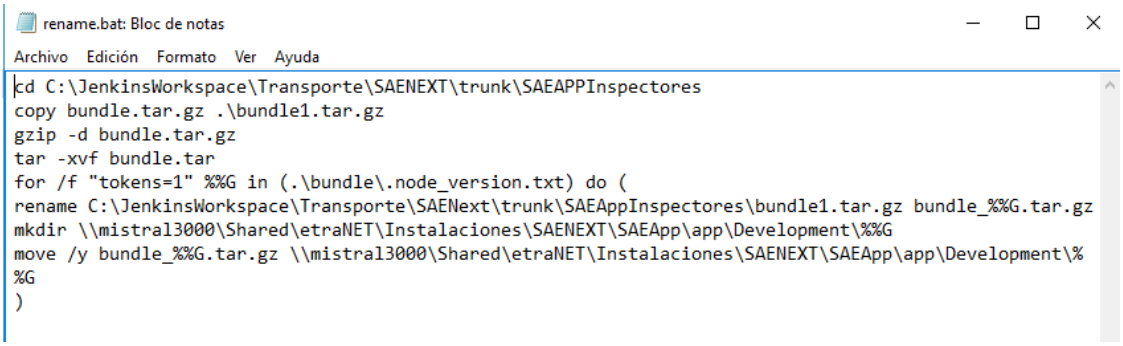


Figura 40: Rename.bat

Tal y como podemos ver en el script, realizaremos una copia del bundle, la cual descomprimiremos para acceder al archivo .node_version.txt, del cual extraeremos el número de versión, que añadiremos al nombre del bundle que dejamos sin comprimir, y, tras ello, trasladaremos al repositorio de aplicaciones.

Una vez hecho esto, bastará con llamar ambos scripts desde el programador de tareas, ya que será necesario el lanzamiento desde ahí para que la compilación funcione adecuadamente, y la compilación estará lista.

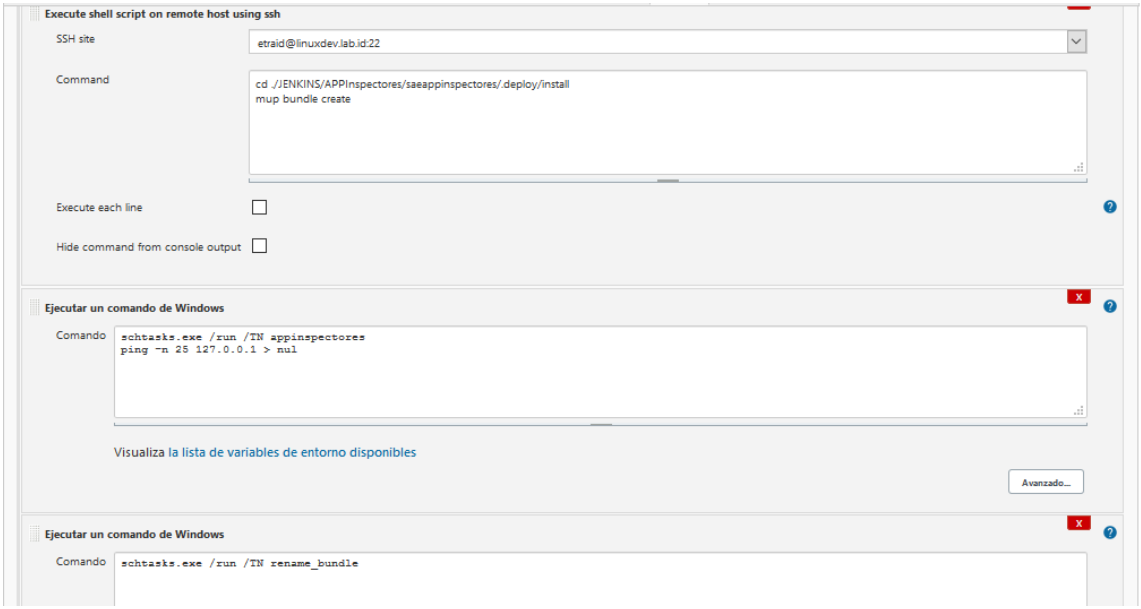


Figura 41: Job de SAEAppInspectores

Tal y como vemos en esta última captura, aprovecharemos la conexión SSH establecida en Jenkins para generar el bundle, y, tras ello, llamaremos a appinspectores.bat, transfiriendo el bundle a nuestra máquina de trabajo. Finalmente, llamaremos al rename.bat con el que ultimar la solución y cerrar el job.

Capítulo 5. Conclusiones y líneas futuras.

Tal y como se ha podido ver a lo largo de la memoria, nuestro trabajo ha permitido optimizar y automatizar la generación de versiones de hasta una treintena de aplicaciones, representadas bajo los formatos de soluciones planteados en el anterior capítulo.

Con ello, podemos observar un gran beneficio en la optimización de los recursos de la empresa, ya que el número de operaciones y acciones a realizar en la generación de las versiones de cada una de las aplicaciones, podría suponer una pérdida muy notoria en la eficacia de la empresa, al tener que destinar muchas horas de los distintos desarrolladores para un trabajo que, con esta implementación, apenas requiere un click.

En cuanto a la aplicación elegida, se considera que Jenkins es una herramienta ideal para desarrollar este proyecto, dado que consta de una interfaz verdaderamente sencilla e intuitiva, pero con un espectro de funcionalidades realmente amplio.

En todo caso, hemos podido observar algunas limitaciones (como la encontrada con los ficheros vdproj y el error derivado de su compilación en Jenkins), que más bien podemos atribuir a una desactualización de las soluciones de la empresa que a un problema del propio Jenkins, ya que este tipo de proyectos han desaparecido de las últimas versiones de Visual Studio.

Sin embargo, Jenkins nos ha permitido trabajar con diferentes tipos de repositorios (SVN, Gitlab…), integrar otro tipo de aplicaciones como ha podido ser el caso de Jira, e incluso trabajar en entornos de Linux a través de la conexión SSH, pasando por el trabajo en distintos lenguajes (C++, NodeJS, NSI…) de lo que podemos deducir un nivel de posibilidades altísimo que hemos aprovechado en nuestro trabajo.

Otro aspecto muy destacable ha sido la capacidad de adaptación del programa desarrollado en Jenkins respecto a las necesidades de la empresa. Esto se ha podido ver no solo en la flexibilidad mostrada para generar versiones de aplicaciones muy diferentes entre sí, sino también a la hora de poder adaptar la estructura de Jenkins al funcionamiento en sí de la empresa, estructurado en gran medida mediante tareas en Jira, y que ha podido integrarse totalmente en la estructura de Jenkins para que ambas patas del trabajo estén completamente unidas.

Los beneficios de la compilación continua y la integración se han podido ver de forma práctica a lo largo de todo el proyecto: el aumento de eficiencia, la optimización de los recursos humanos y técnicos de la empresa, y, además, la ordenación y estructuración de una serie de procesos que pueden llegar a ser enrevesados realizados de forma manual pero que de esta forma pasan a tener una estructura sencilla y eficaz. En definitiva, la relación entre el esfuerzo realizado para este proyecto y los beneficios obtenidos tras ello es claramente positiva.

Si bien podemos valorar que la elaboración del proyecto ha tenido un carácter muy técnico y específico, debemos entender que la realización de determinadas tareas que puedan resultar simples de forma aparente (cambiar de nombre un fichero, leer un trozo de un determinado fichero, etc), son tareas que pueden entrañar una mayor dificultad a la hora de ser automatizadas. De esta forma, el foco de este trabajo es conseguir que una serie de pequeñas tareas, sencillas de forma independiente, puedan combinarse y encajar de forma adecuada en su automatización.

Por último, en relación a posibles líneas futuras de trabajo, se considera que uno de los principales avances que podrían darse sería la introducción de herramientas de mejora y análisis de código, existentes dentro de Jenkins y que pueden darnos información interesante sobre posibles redundancias, fallos de optimitización…de los códigos existentes; y, por supuesto, se debe tener en cuenta que el trabajo desarrollado en Jenkins no puede quedar cerrado como tal, en tanto que ante el desarrollo de nuevas aplicaciones, será necesario integrarlas en el sistema de Jenkins y aprovechar todo el trabajo desarrollado aquí para seguir automatizando estos procesos.

En definitiva, la compilación continua es una práctica en crecimiento a día de hoy, extendida entre las mayores empresas de desarrollo de software, y totalmente recomendable para cualquier empresa que aspire a ser competitiva en este mundo del desarrollo.

Capítulo 6. Bibliografía

[1] 1&1 IONOS España S.L.U. (2021, mayo 28). *Jenkins tutorial*. IONOS Digitalguide. https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/jenkins-tutorial/

[2]Atlassian. (2019). *Primeros pasos con Jira Software |Tutorial gratuito*. https://www.atlassian.com/es/software/jira/guides/getting-started/basics#step-3-set-up-your-columns

[3]1&1 IONOS España S.L.U. (2021, 28 mayo). *Integración continua*. IONOS Digitalguide. https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/integracion-continua/

[4]García, C. S. (2012, 9 mayo). Subversion: Introducción y comandos básicos. Coding Something. https://codingsomething.wordpress.com/2011/02/21/subversion-introduccion-basicos/

[5] Introducción a GitLab. (2017, 12 octubre). Desarrollo Web. https://desarrolloweb.com/articulos/introduccion-gitlab.html

[6]Crespo, C. (2004, 15 febrero). *Realizar Instaladores con NSIS 2.0*. Adictos al trabajo. https://www.adictosaltrabajo.com/2004/02/15/nsis/

[7] A. (2012, 1 noviembre). Visual Studio Installer Deployment. Microsoft Docs. https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2010/2kt85ked(v%3dvs.100)

[8] J. (2018, 21 septiembre). Instaladores en Windows y qué diferencia existe entre un instalador .exe y uno .msi. campusMVP.es. https://www.campusmvp.es/recursos/post/instaladores-en-windows-y-que-diferencia-existe-entre-un-instalador-exe-y-uno-msi.aspx

[9]*An error occurred while validating. HRESULT = «8000000A»*. (2011, 27 diciembre). Stack Overflow. https://stackoverflow.com/questions/8648428/an-error-occurred-while-validating-hresult-8000000a

[10]*Jenkins CI service | GitLab*. (2019). Gitlab. https://docs.gitlab.com/ee/integration/jenkins.html

[11] Lucas, J. (2020, 1 junio). Qué es NodeJS y para qué sirve. OpenWebinars.net. https://openwebinars.net/blog/que-es-nodejs/

[12] I. (2017, 12 agosto). Razones por las que usar Linux para desarrollo. Linux Adictos. https://www.linuxadictos.com/razones-las-usar-linux-desarrollo.html

[13] C., D. (2021, 21 mayo). ¿Cómo funciona el SSH? Tutoriales Hostinger. https://www.hostinger.es/tutoriales/que-es-ssh

[14] ¿Qué es el SCP protocol? (2020, 12 agosto). IONOS Digitalguide. https://www.ionos.es/digitalguide/servidores/know-how/scp-secure-copy/

[15] Adventures, H. C.-I. T. (2020). Windows 10 OpenSSH Equivalent of. ssh-copy-id. https://www.chrisjhart.com/Windows-10-ssh-copy-id/