UNIVERSIDAD POLITÉCNICA DE CARTAGENA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Trabajo Fin de Grado

Herramientas de Integración y Desarrollo Continuo para Proyectos de Software Colaborativo

---

Autor: César Francisco San Nicolás Martínez
Director: Pablo Pavón Mariño
Codirector: José Juan Pedreño Manresa

17 de Julio de 2017

Autor:	César Francisco San Nicolás Martínez
E-mail del autor:	cesarfsannicolasmartinez@gmail.com
Director:	Pablo Pavón Mariño
E-mail del director:	pablo.pavon@upct.es
Co-director:	José Juan Pedreño Manresa
E-mail del codirector:	josej.pedreno@upct.es
Título del TFG:	Herramientas de integración y desarrollo continuo para proyectos de software colaborativo.

Resumen:
Desarrollo de dos sistemas de integración continua para la herramienta Net2Plan basado cada uno de ellos en un gestor de integración continua, el primero en Jenkins y el segundo en Travis.

Titulación:	Grado en Ingeniería Telemática
Departamento:	Tecnologías de la Información y las Comunicaciones
Fecha de presentación:	17 de Julio de 2017

Índice general

Capítulo 1

Introducción

Desde hace unos años, el desarrollo de software se ha convertido en una práctica cada vez más frecuente en cualquier empresa [1]TIC. A medida que la complejidad de los proyectos de software aumenta, son necesarios especialistas en diferentes lenguajes y tecnologías, siendo un requisito que éstos trabajen de forma cooperativa y en sintonía.

De ahí surge el gran problema que es integrar todos los cambios realizados en el código por las diferentes personas pertenecientes a un mismo equipo, que se acrecienta cuando varias personas trabajan sobre el mismo fragmento de código, lo cual resulta bastante tedioso de solucionar. Como solución a los problemas planteados, surge la metodología de **integración continua**.

1.1. Motivación

El desarrollo de **Net2Plan** se remonta a 2011 siendo una herramienta inicialmente creada por solo dos personas para ser utilizada principalmente en docencia e investigación. Seis años más tarde, se ha convertido en una aplicación utilizada también por empresas e investigadores relacionados con las redes de telecomunicación, y el equipo encargado del desarrollo y mantenimiento ha crecido en concordancia.

Debido a este incremento de personal asociado al proyecto, se hace necesario introducir y aplicar la metodología de integración continua para evitar problemas de conflicto de código; garantizar la integridad los cambios realizados, por pequeños que sean, mediante el uso de tests y automatizar el despliegue de nuevas versiones de Net2Plan a GitHub, para que los usuarios puedan disponer de la aplicación de forma sencilla.

1.2. Objetivos

El objetivo principal de este trabajo es crear un sistema de integración continua para el desarrollo de la herramienta Net2Plan, pudiendo desglosarse los siguientes hitos:

- Utilizar eficientemente Git y GitFlow para evitar conflictos de código.

[1]Tecnologías de la Información y la Comunicación

- Utilizar la integración continua para minimizar trabajo y esfuerzo en las tareas de detección de errores y despliegue.
- Desplegar un sistema de integración continua de Net2Plan con Jenkins y otro con Travis.
- Evaluar ambos sistemas, y elegir uno u otro en función de sus ventajas y desventajas.

1.3. Plan de trabajo

Para la consecución de los objetivos marcados, el plan de trabajo consta de diferentes fases, cada una de ellas destinada a cumplir un objetivo concreto:

- Estudio y familiariación con las distintas tecnologías necesarias para implantar los sistemas de integración continua (Git, GitFlow y Docker).
- Estudio y documentación con Jenkins y Travis.
- Desarrollo del sistema de integración continua con Jenkins.
- Desarrollo del sistema de integración continua con Travis.
- Comparativa entre ambos sistemas.
- Escritura de la memoria.

1.4. Estructura de la memoria

La memoria de este Trabajo de Fin de Grado se estructura de la siguiente manera:

- Capítulo 1: Introducción

  Breve introducción al problema actual que tienen los equipos de desarrollo de software y de como la integración continua ayuda a solucionarlos. Se detallan las motivaciones del proyecto, los objetivos y el plan de trabajo.

- Capítulo 2: Tecnología y estado del arte

  Descripción de las de las distintas herramientas que forman parte de un sistema de integración continua, como son Git, Gitflow o Docker; su estado actual y como se deben usar. Así mismo, se hace una breve introducción a **Net2Plan**, que es la herramienta destinada a la implantación de un sistema de integración continua. Por último, se reseñan Jenkins y Travis, que son los principales gestores de integración continua para proyectos *open-source*.

- Capítulo 3: Desarrollo e integración continua

  Desarrollo de los sistemas de integración continua para Net2Plan con Jenkins y Travis. Se explica detalladamente los pasos que se han seguido para desarrollar cada uno de los sistemas, y las herramientas que se han utilizado para ello.

- Capítulo 4: Conclusiones

  Reflexión final del proyecto, comparando ambos sistemas de integración continua, y escogiendo uno de ellos para implantar definitivamente en función de sus ventajas y desventajas. Por último, se hablará de posibles ampliaciones de este proyecto.

Capítulo 2

Tecnología y estado del arte

En este capítulo se hablará sobre las diferentes metodologías y herramientas que actualmente permiten agilizar el desarrollo de software, como son la integración continua, Git o Docker.

Así mismo, se hará una breve introducción a Net2Plan, que es la herramienta *open-source* en la que se han llevado a cabo las metodologías y herramientas anteriormente citadas.

Por último, se citarán los dos principales gestores de integración continua para proyectos de software *open-source*, que son Jenkins (2.5.1) y Travis (2.5.2).

2.1. Net2Plan

Net2Plan [1] es una herramienta *open-source* programada en Java dedicada a la planificación, optimización y simulación de redes de comunicaciones desarrollada por el grupo de investigación GIRTEL de la Universidad Politécnica de Cartagena. En sus inicios, fue concebida como una herramienta para docencia sobre redes de comunicaciones. Sin embargo, actualmente se ha convertido en una poderosa herramienta de optimización y planificación de redes, con un repositorio de recursos para la planificación de redes, tanto para el entorno académico como para el entorno de la industria y la empresa.

Net2Plan está basado en una representación de redes con componentes abstractos, tales como nodos, enlaces, demandas, ... Ésto está pensado para poder planificar cualquier tipo de red, sin importar la tecnología que utilice. Para poder personalizar las redes a gusto del usuario, cada componentes permite añadir atributos. Además, hay clases que permiten modelar una tecnología en concreto (redes IP, WDM o, incluso, escenarios de NFV).

Net2Plan tiene dos modos de uso: mediante interfaz gráfica (GUI) y línea de comandos (CLI). La interfaz gráfica está pensada para utilizar en sesiones de laboratorio como un recurso formativo, o para poder ver más detalladamente la red sobre la que se está trabajando. Por otro lado, el modo línea de comandos facilita los estudios de investigación, ya que permite automatizar ejecuciones de algoritmos o simulaciones. Como se ha hablado antes, ambos modos permiten utilizar Net2Plan en el entorno educativo (investigación o enseñanza) y en el entorno de la industria y la empresa.

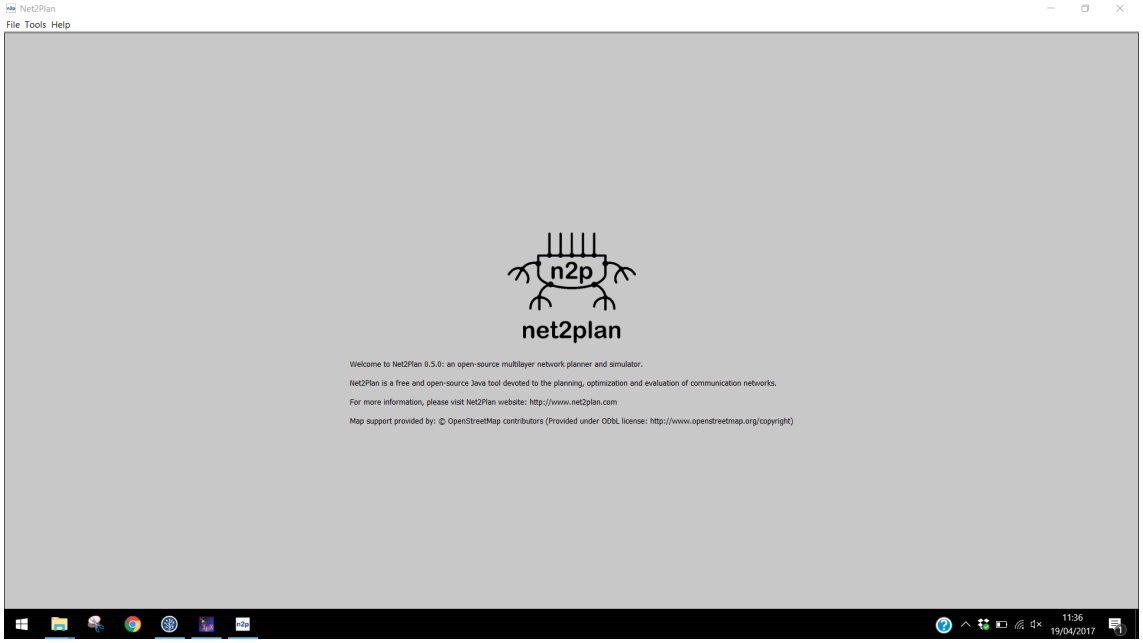

FIGURA 2.1: Ventana de inicio de Net2Plan

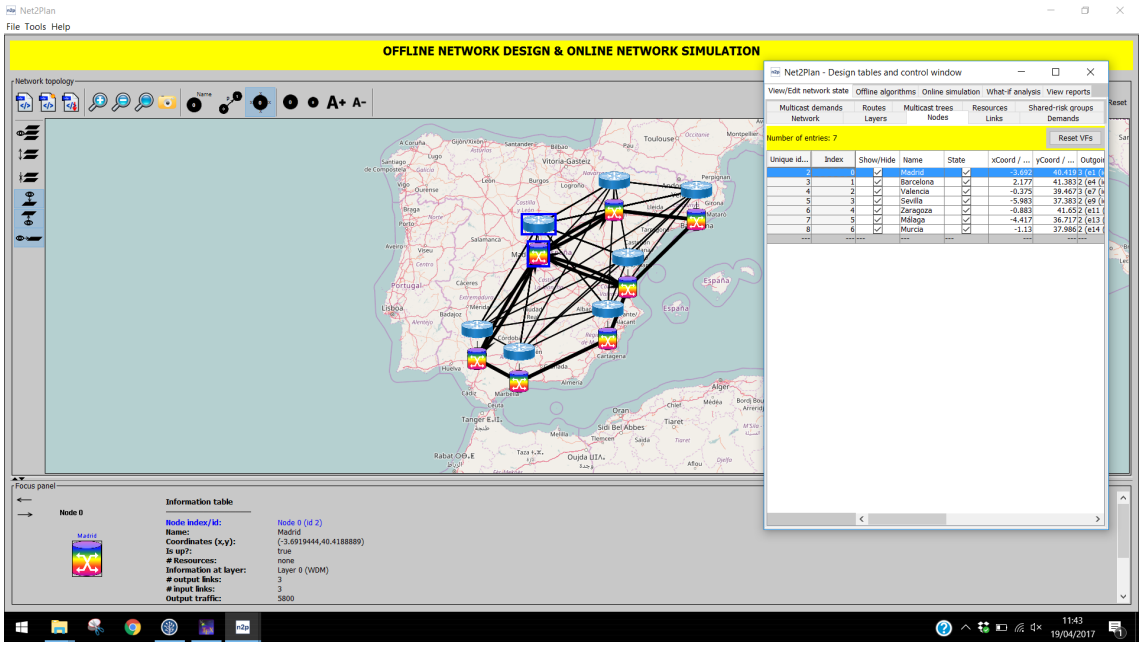

FIGURA 2.2: Ventana *Offline network desing and online network simulation*

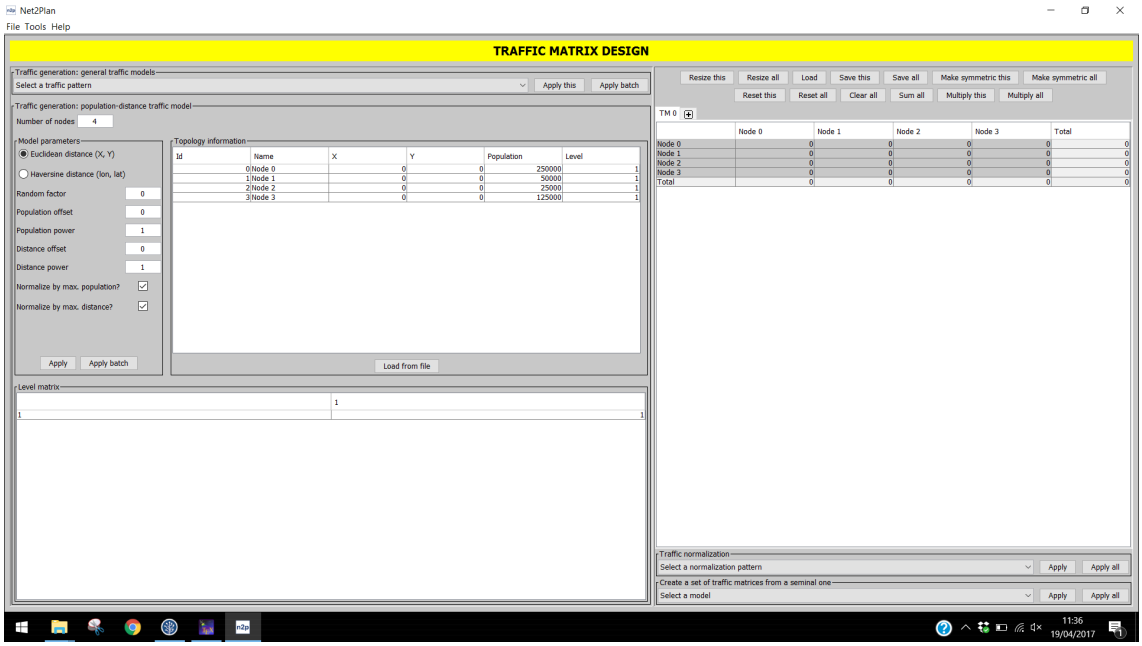

FIGURA 2.3: Ventana *Traffic matrix design*

2.2. Git

Git es un sistema de control de versiones (CVS) creado en 2005 por Linus Torvalds para gestionar el desarrollo de código del kernel de Linux. Está diseñado para trabajar de forma concurrente en diferentes partes de código, a través de ramas (**`branch`**, en inglés).

Cada **`commit`** representa una serie de cambios respecto a la versión anterior de una rama en concreto, y, además, tiene un identificador único (a través del algoritmo SHA1). Un *commit* puede tener varios padres, es decir, varias ramas. Esto es lo que se conoce como **`merge`**.

Un *merge* es la unión de dos o más ramas en una única, mezclando los cambios de todas ellas. Una de las ramas siempre será la base, ya que cuando realizamos esta acción, aplicamos todos los cambios sobre la rama base. Si git no puede decidir automáticamente como aplicar los cambios, se deberás resolver los posibles conflictos manualmente, mediante aplicaciones pensadas para ese fin, como *diff-merge*.

Para utilizar git se puede hacer mediante comandos, o utilizando algún programa para trabajar mediante interfaz gráfica (GUI), como *SourceTree* (ver foto 2.4), para Windows exclusivamente, o *GitKraken* (ver foto 2.5), para Windows y Linux.

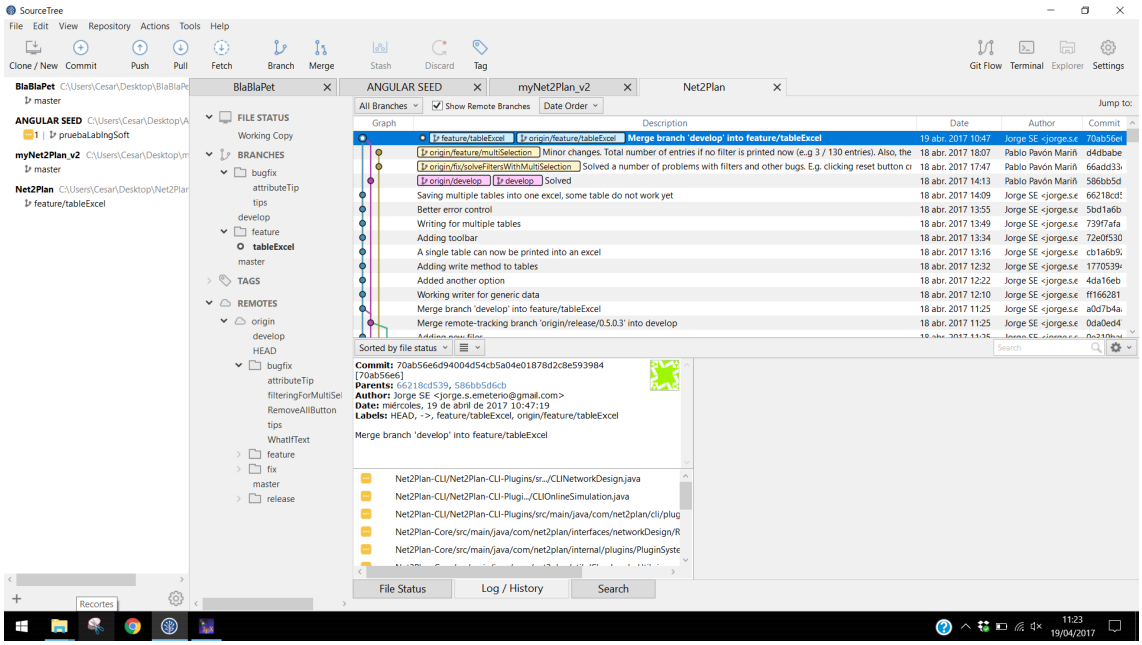

FIGURA 2.4: Interfaz gráfica de SourceTree

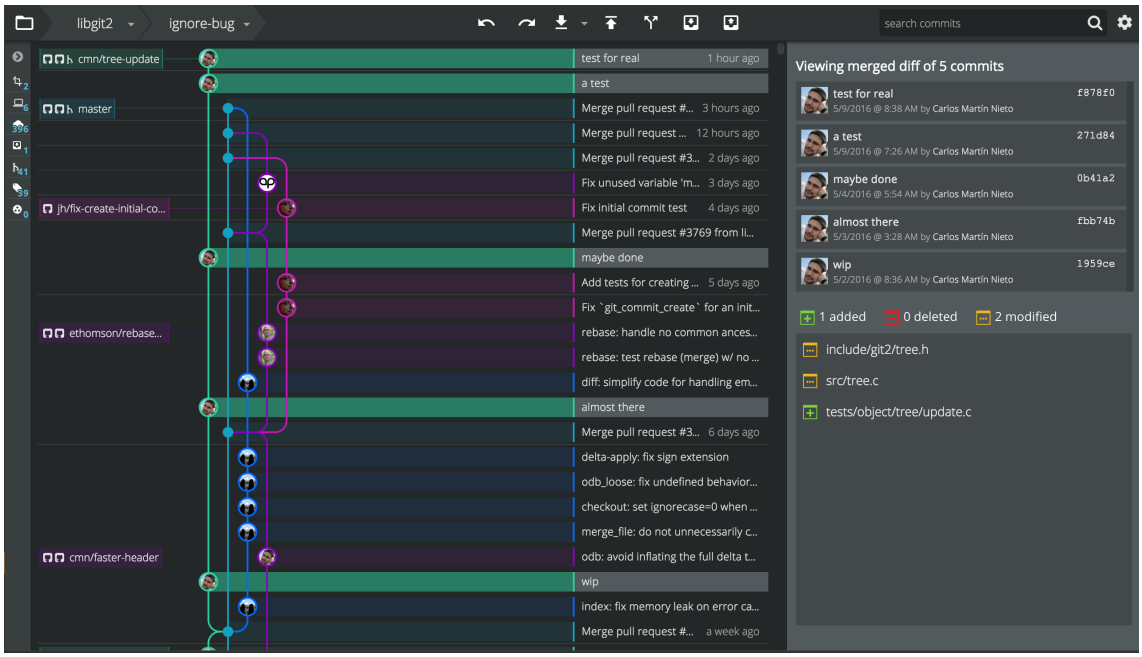

FIGURA 2.5: Interfaz gráfica de GitKraken

2.2.1. Comandos básicos

- **git init [nombre-proyecto]** Crea un nuevo repositorio llamado nombre-proyecto.
- **git add [archivo]** Añade archivo al *stash*, que es un área especial que contiene todo aquello sobre lo que queremos mantener el control de cambios.
- **git commit [descripcion]** Guarda los cambios y mueve el puntero HEAD a este commit.
- **git diff** Permite ver la diferencia entre el *stash* y el estado actual.
- **git status** Permite ver el estado del repositorio.
- **git checkout -b [nombre]** Crea una nueva rama llamada nombre tomando como base el commit actual.
- **git checkout [nombre]** Cambia el puntero a la rama nombre.
- **git merge [nombre]** Realiza un *merge* de la rama nombre sobre la rama actual. Si no se puede realizar, nos pedirá resolver el conflicto manualmente.
- **git mergetool** Abre la herramienta que tengamos como predeterminada para solucionar conflictos.

2.2.2. GitFlow

GitFlow es una metodología para organizar el trabajo en grupo consistente en separar el desarrollo de cada nueva característica en una rama, manteniendo aislada la rama *master*, para producción, de la rama *develop*, para desarrollo. Sus principales características son:

- La rama `master` hace referencia al código que actualmente se encuentra en producción, es decir, la última versión completamente funcional.
- La rama `develop` guarda el código que actualmente se está desarrollando. A ella se debe integrar el trabajo diario, al menos, una vez al día.
- Habrá una rama `feature/X` por cada nueva funcionalidad que se desarrolle, obtenida de la rama *develop*. Por ejemplo: feature/newButton. Una vez finalizado el desarrollo de la nueva funcionalidad, se realiza un *merge* de la rama *feature/X* en *develop*.
- Habrá una rama `release/X` por cada *release*. Por ejemplo: release/0.5.0.1. Ésto permite liberar la rama *develop* para continuar con el desarrollo, y a su vez permite detectar y solucionar fallos más fácilmente.
- Habrá una rama `bugfix/X` por cada fallo encontrado en `release/X`. Cuando el fallo se haya solucionado, se hará un *merge* de la rama `bugfix/Z` tanto en la rama `release/X` como en la rama `develop`.

- Habrá una rama `hotfix/X` por cada fallo encontrado en producción. Cuando el fallo se haya solucionado, se hará un *merge* de la rama `hotfix/X` tanto en la rama `master` como en la rama `develop`.

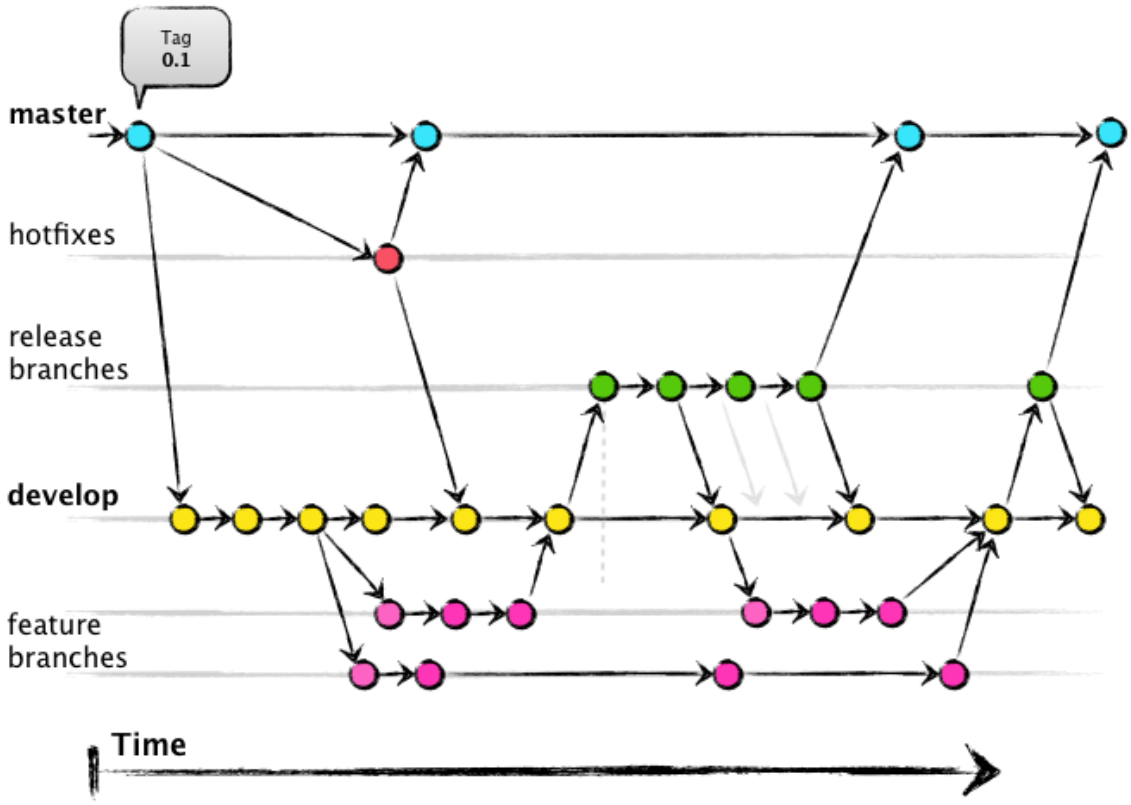

FIGURA 2.6: GitFlow

2.3. Integración continua

> La integración continua es una práctica de desarrollo de software en la cuál los miembros del equipo integran su trabajo de forma frecuente. Cada integración desencadena una ejecución de pruebas que permiten detectar errores lo mas rápido posible.
>
> *Martin Fowler*

La integración continua [2] es una metodología de trabajo propuesta a principios de los años 90, debido a que en los primeros diseños de la metodología XP (*eXtreme Programming*), se hizo popular el concepto de **integration hell**. Este se define como la dificultad de combinar el trabajo de dos o más personas que han estado trabajando y realizando diferentes cambios sobre la misma base de código. Si los cambios son sustanciales, combinar el trabajo de todos ellos puede llegar a ser realmente complicado, y acabar ocupando incluso más tiempo que el propio desarrollo.

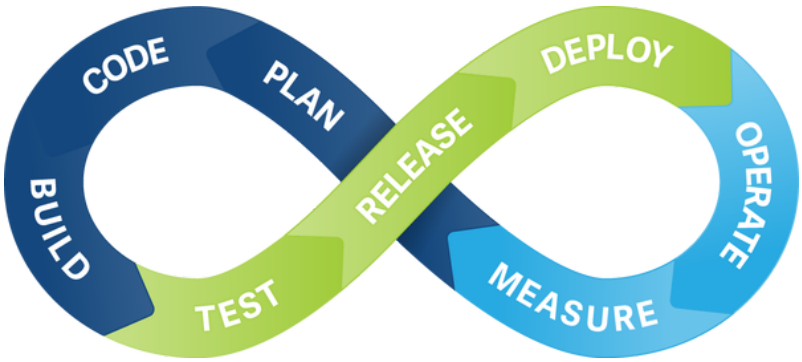

FIGURA 2.7: Modelo de integración continua

2.3.1. Sistema de integración continua

Para que un sistema de integración continua [2] [3] sea eficiente, hay determinadas prácticas que debemos llevar a cabo:

- Mantener un repositorio de una única fuente

  Ciertos proyectos de desarrollo de *software* involucran grandes cantidades de archivos que necesitan ser tratados como un conjunto. Eso es un esfuerzo muy grande, especialmente cuando dichos proyectos están integrados por un gran número de personas. Así que, con el paso de los años, se han creado diferentes herramientas para gestionar dicho esfuerzo.

  Esas herramientas se conocen de diferentes formas (herramientas de control de versiones, herramientas de control de código fuente, etc...) y, actualmente son una parte esencial de la mayoría de proyectos de desarrollo de *software*.

  Existen múltiples herramientas para este fin, de las cuáles destaca `GitHub`. Es una plataforma web que hace uso de Git, del que ya se habló en la sección 2.2. En ella podemos crear lo que se conoce como **repositorio**, que es un *almacén virtual* donde se guardan los archivos de nuestro proyecto.

  Siguiendo la metodología GitFlow, explicada en la sección 2.2.2, debemos crear una rama `master` y una rama `develop`. La segunda de ellas será lo que se conoce como la *main-line*, es decir, la rama principal donde se irán acumulando los diferentes cambios.

- Automatizar las *builds* de un proyecto

  Mover el código fuente a un sistema de ejecución puede ser a veces un proceso complicado, ya que involucra la compilación, el mover archivos, el acceso a las bases de datos, etc... Sin embargo, como la gran mayoría de tareas que componen el desarrollo de software, éstas pueden, y deben, ser automatizadas. Ejecutar comandos extensos a mano o ir navegando entre distintas ventanas haciendo *click* es una pérdida de tiempo.

Para ello, existen diferentes entornos o herramientas para automatizar la *build* de cualquier proyecto de desarrollo de software. Para Java existen tres herramientas: Gradle, Maven y Ant. En el proyecto Net2Plan se utiliza Maven, así que se comentará brevemente.

Maven es una herramienta creada para realizar la *build* de un proyecto Java. Se basa en la utilización de un archivo llamado POM (*Project Object Model*), escrito en formato XML, en el que se describe el proyecto a construir, sus dependencias con otros módulos (Maven permite crear una estructura de proyecto multi-módulo, para separar las diferentes partes del proyecto) y componentes externos, y el orden en el que se deben construir los elementos. Tiene varias tareas ya definidas, como el empaquetado del proyecto (compilar todo el proyecto y crear un archivo .exe o .zip).

- Hacer que el proyecto ejecute sus propios *tests*

  Un programa puede compilar y ejecutarse correctamente, pero eso no quiere decir que su funcionalidad sea la que debe. Actualmente, los IDE de programación tienen una herramienta para hacer *debug* del código, es decir, ir línea por línea viendo que sucede, pero a veces no es suficiente para detectar todos los fallos.

  Por esta razón surge la idea de incluir tests personalizados en la ejecución de la `build` del proyecto, y, aunque no sean perfectos para detectar el 100 % de los errores, pueden detectar la gran mayoría. Gracias a los conceptos de XP (*eXtreme Programming*) y TDD (*Test Driven Development*), la práctica de incluir test personalizados en la compilación de software se ha popularizado y los equipos de desarrollo se han dado cuenta de su utilidad.

  Para tener un proyecto que ejecute sus propios tests, necesitas un conjunto de tests que se encarguen de detectar fallos en la funcionalidad del código del proyecto. Para ello existen diferentes herramientas, aunque la más común es XUnit, que son un conjunto de herramientas *open-source* para programar tests. Por ejemplo, existe NUnit para .NET o JUnit para Java.

- Todos los miembros del proyecto hacen *commit* a la rama principal todos los días

  La integración continua permite que los desarrolladores informen a los otros desarrolladores de los cambios que han realizado. Esto se realiza mediante un *commit*.

  El funcionamiento es el siguiente, siguiendo la metodología GitFlow (2.2.2):

  Cuando un desarrollador quiere hacer cambios en la rama `develop`, crea a partir de ella una rama llamada `feature/X`, siendo X el nombre. En esta nueva rama, el desarrollador llevará a cabo su trabajo, y por cada pequeño objetivo que vaya cumpliendo, hará `commit` a la rama. Una vez al día, se deberá actualizar la rama `develop` mediante lo que se conoce en `GitHub` como `pull-request`, que consiste en solicitar al jefe del proyecto que incluya tus cambios en la rama `develop`, ya que, normalmente, únicamente el jefe o los jefes del proyecto son quienes tienen acceso directo a las ramas `master` y `develop`.

Hacer esto frecuentemente posibilita que los desarrolladores vayan haciendo copias de su trabajo diario, y que los demás desarrolladores vean los cambios y se puedan solucionar los conflictos entre distintos trabajos.

- Todos los *commit* a la rama principal ejecutan una *build*

  Realizando *commits* diarios a la rama `develop` (en el caso de `GitHub` mediante `pull-request`), podemos tener *builds* que han sido "testeadas". Esto significa que la rama principal está en un estado correcto, por lo menos en teoría. En la práctica, los usuarios no suelen ejecutar una *build* manualmente por cada *commit* que realizan.

  Para solucionar esto, existen los gestores de integración continua. Para proyectos de software de carácter *open-source* existen principalmente dos: Jenkins y Travis. Son programas de software que permiten automatizar las *build* de un proyecto para que se ejecuten automáticamente tras cada *commit* realizado. También permite ejecutar *build* manualmente. De esta forma. el desarrollador no tiene que preocuparse de ejecutar el mismo la *build*.

  En la sección 2.5 hablaremos más detalladamente de varios gestores de integración continua, entre los que se encuentran Jenkins y Travis.

- Las *builds* erróneas deben arreglarse inmediatamente

  Una parte clave de la integración continua es que cuando una `build` falle, se debe arreglar inmediatamente, ya que la integración continua se basa en el hecho de trabajar siempre sobre una base estable.

  A veces, la manera más rápida de arreglar el fallo es volver atrás y revertir el último `commit` realizado. A menos que la causa del fallo sea trivial y se detecte al momento, lo correcto es volver atrás y debuggear en la rama `feature/X` antes de volver a hacer el commit a la rama principal.

  > No hay tarea con prioridad más alta que arreglar una *build* errónea.
  >
  > *Kent Beck*

  Esto no significa que cuando haya un error todos los desarrolladores deban dejar de lado lo que están haciendo para centrarse en el error. Simplemente la persona que ha causado el fallo con su `commit` deber centrarse en solucionar ese fallo, pero sin dejar de lado sus labores.

- Realizar la *build* lo más rápido posible

  Uno de los objetivos de la integración continua es el devolver a los desarrolladores un *feedback* rápidamente. Nada es más desesperante en el entorno de la integración continua que una *build* tarde mucho tiempo (en este entorno, mucho tiempo equivale a, aproximadamente, una hora).

  El concepto XP (*eXtreme Programming*) aconseja disminuir el tiempo de ejecución de *build*, ya que cada minuto que ahorres en ello, es un minuto que tiene el desarrollador para realizar su trabajo. Muchos expertos recomiendan que la ejecución de una *build* no sobrepase los diez minutos.

Reducir el tiempo se puede lograr con un método que consiste en separar la *build* en varias mas cortas, y según tu objetivo, ejecutar unas u otras. Para que quede más claro, ahí va un ejemplo: Estás trabajando en tu proyecto, y estas realizando cambios en la parte gráfica. Ahorramos tiempo si solo ejecutamos la build que incluye los test correspondientes a la parte gráfica, y no todos ellos.

- Hacer pruebas en una copia del escenario de producción real

  El fin de realizar tests es detectar, bajo condiciones determinadas, cualquier problema que el proyecto pudiera tener en producción. Una condición que escapa a los tests en el entorno donde la aplicación va a ser ejecutada. Por ello es importante realizar pruebas en los distintos entornos donde la aplicación será ejecutada antes de llevarla a producción, aún cuando los tests no detecten fallos.

  Para aplicaciones web, que están pensadas en ser ejecutadas en un servidor remoto, hay programas como `XAMPP` que permite utilizar un servidor apache y un servidor SQL en la máquina local. También se puede utilizar un servidor privado que sea una copia exacta del servidor real para hacer pruebas.

  Net2Plan es una aplicación de escritorio para trabajar localmente. Para aplicaciones como estas, se realiza una build local y se prueba en los distintos sistemas operativos (Windows, Linux, MAC) para ver posibles fallos.

- Despliegue continuo

  El proceso de despliegue continuo es la etapa final de la integración continua. Una vez que nuestro software de integración continua ha ejecutado la *build* del proyecto, ha superado los tests con éxito y hemos obtenido un archivo ejecutable con nuestra aplicación, llega el momento de mandarlo a producción.

  En función del software de integración continua que tengamos, podemos realizar el despliegue de una forma u otra, aunque la forma más común es mediante un script. En concreto, para realizar despliegue en GitHub, hay que utilizar un script para hacer una llamada a la `REST-API` de GitHub pasándole varios parámetros, como el nombre, la descripción o la ruta del archivo ejecutable. Esto nos permite hacer una nueva `release`.

2.4. Docker

Docker es un proyecto *open-source* que permite desplegar aplicaciones dentro de contenedores, proveyendo a cada una de ellas un gran nivel de abstracción e independencia respecto de las otras. Está únicamente disponible para utilizar en el sistema operativo Linux, ya que utiliza gran parte de las funciones de su Kernel para permitir que cada contenedor se ejecuta en una sola instancia de Linux, evitando la sobrecarga de mantener máquinas virtuales.

Por otro lado, también permite exportar cada contenedor con su aplicación de una máquina a otra, ahorrando tiempo en instalaciones.

Para desplegar las aplicaciones en los contenedores, existe *DockerHub*, que es un repositorio centralizado donde se encuentran todas las imágenes de las distintas aplicaciones.

2.4.1. Comandos básicos

- **docker pull jenkins** Descarga la imagen oficial de Jenkins que se encuentra en *DockerHub*.
- **docker run** Crea un nuevo contenedor. Este comando tiene diferentes parámetros de entrada:
  - –rm *Borra el contenedor cuando terminemos de usarlo.
  - -it *Conecta el contenedor a la terminal de Linux.
  - –name cont *Pone el nombre cont al contenedor.
  - -p 24601:80 *Mapea el puerto 24601 de la máquina al puerto 80 del contenedor.
  - -v /dev:/code *Monta el directorio /dev como el directorio /code dentro del contenedor.
  - jenkins:1.2 *Instancia la versión 1.2 de Jenkins (si ponemos *latest* como versión, descarga la más reciente).
  - ./script.sh *Ejecuta este comando al iniciar el contenedor.
- **docker start mycont** Inicia el contenedor mycont.
- **docker stop mycont** Para el contenedor mycont.
- **docker kill mycont** Mata el contenedor mycont.
- **docker images** Muestra las imágenes disponibles en local.
- **docker rmi myimag** Borra la imagen myimag en local.
- **docker ps -a** Muestra todos los contenedores creados en local.
- **docker rm mycont** Borra el contenedor mycont en local.
- **docker login** Hace login en *DockerHub* y guarda nuestras credenciales.
- **docker push myuser/myapp:1.4** Sube la imagen a *DockerHub* con el tag "myuser/myapp:1.4".

2.5. Gestores de integración continua

2.5.1. Jenkins

Jenkins es un gestor de integración continua *open-source* escrito en Java. Está basado en el proyecto *Hudson*, de hecho, éste último cambió en 2011 su nombre a Jenkins, debido a problemas con Oracle.

FIGURA 2.8: Ventana de inicio de Jenkins

Para funcionar, Jenkins necesita ser corrido en un servidor. Soporta herramientas de control de versiones como Git o Subversion y puede ejecutar proyectos basados en Ant y Maven, así como scripts programados en *Bash*.

Jenkins está pensado para proyectos grandes, en los cuáles están involucrados un elevado número de personas. Por ello, permite configurar una matriz de gestión de permisos mediante *LDAP*. Así, cada usuario con acceso a Jenkins puede únicamente modificar configuración según esté configurada dicha matriz de permisos (ver foto 2.9).

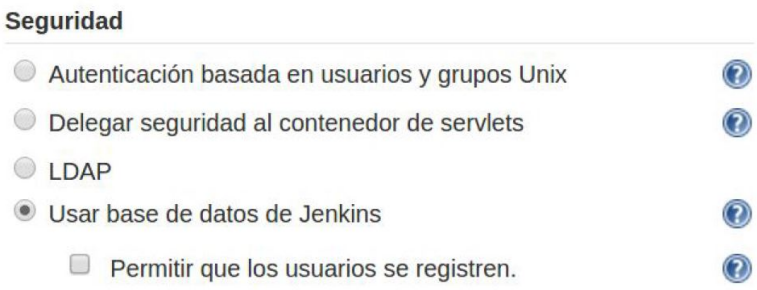

FIGURA 2.9: Configuración de permisos en Jenkins

Para trabajar con las distintas herramientas de desarrollo de software (GitHub, BitBucket, Maven, Ant, Gradle,...), así como añadir funcionalidades como obtener reportes de los tests o enviar notificaciones de correo cuando una build es errónea, utilizamos diferentes **`Plugins`**.

Algunos de los Plugins más utilizados en Jenkins son los siguientes:

- Mailer Plugin

  Permite enviar notificaciones de correo electrónico cuando una *build* es errónea o cuando después de varias *builds* erróneas, vuelve a haber una correcta.

- Maven Project Plugin

  Permite integración con Maven. Ejecuta los módulos del proyecto en paralelo, accede directamente a los plugins de testeo de Maven para ver resultados.

  Así mismo, tiene varias variables de entorno que permiten obtener datos del proyecto Maven fácilmente, por ejemplo, la variable *POM VERSION* permite obtener directamente la versión del proyecto almacenada en el *POM*.

- Credentials Plugin

  Permite almacenar información de usuario en Jenkins. Se utiliza para almacenar distintos usuarios para las distintas plataformas que integran el proyecto que Jenkins se encarga de automatizar. Puedes tener almacenadas tus credenciales de Github o de BitBucket, por poner un ejemplo.

Además de estos Plugins, existen otros más específicos para cada tipo de proyecto.

Como curiosidad, existe un plugin llamado Chuck Norris Plugin, que muestra fotos de Chuck Norris en función de si la *build* de tus proyectos ha sido errónea o exitosa (ver foto 2.10).

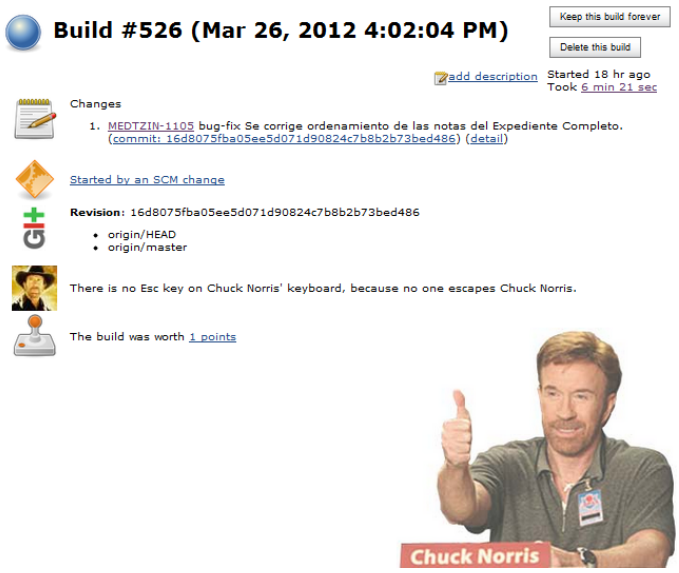

FIGURA 2.10: Chuck Norris Plugin

Jenkins se encarga de gestionar los distintos repositorios mediante tareas.

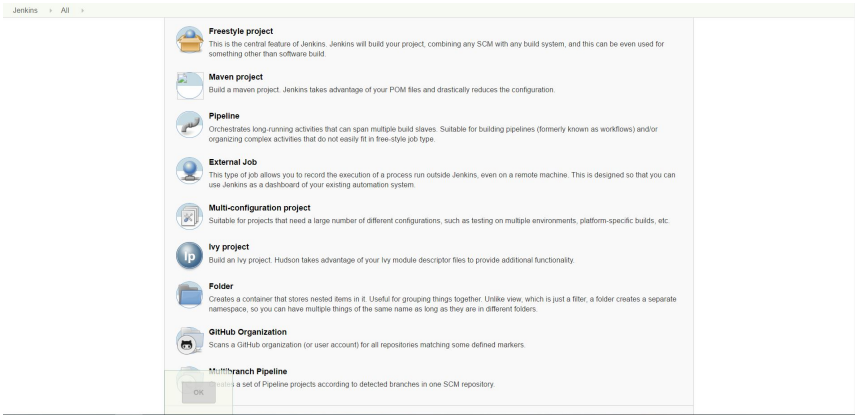

FIGURA 2.11: Tipos de tareas en Jenkins

Existen varios tipos de tareas, como se puede ver en la foto 2.11.

2.5.1.1. Tarea Maven

Puesto que Net2Plan es una herramienta Java gestionada por Maven, se hará una breve explicación de las tareas para proyectos Maven.

Éstas se separan en diferentes pasos. A continuación vemos una breve explicación de cada uno de ellos:

1. **General**

   Se establece el nombre y una descripción de la tarea, así como opciones auxiliares.

2. **Source Code Management**

   Se establece el sistema de control de versiones del repositorio (Git, Subversion o ninguno).

3. **Build Triggers**

   Se establece el método mediante el cuál se activara la *build* del proyecto (mediante un *push* a la rama, mediante un *pullrequest*, etc...).

4. **Build Environment**

   Se establece el entorno de trabajo de la *build* del proyecto. Se puede configurar el despliegue a un entorno de producción, poner marcas de tiempo, parar el despliegue si la build es errónea, entre otros.

5. **Pre Steps**

   Se establecen los pasos previos a la ejecución de la *build*.

6. **Build**

   Se establece el POM que se debe utilizar y las tareas de Maven que se deben ejecutar.

7. **Post Steps**

   Se establecen los pasos posteriores a la ejecución de la *build*.

8. **Build Settings**

   Se establecen configuración de la ejecución de la *build*, como, por ejemplo, avisar mediante correo cuando la *build* sea exitosa o errónea.

9. **PostBuild Actions**

   Se establecen las acciones a ejecutar después que la *build* haya sido exitosa.

2.5.2. Travis

Travis es un gestor de integración continua utilizado principalmente para proyectos de software alojados en GitHub. Cuenta con dos versiones, la versión *open-source* (http:www.travis-ci.org) y la versión profesional de pago (http:www.travis-ci.com).

A diferencia de Jenkins, Travis es corrido en un servidor en la nube privado, por lo que no es necesario el tener un servidor propio.

Para acceder a Travis, debemos acceder con nuestra cuenta de GitHub, y automáticamente, Travis detectará los repositorios de dicho usuario. A continuación, para que Travis actúe correctamente, debemos añadir a nuestro repositorio en GitHub un archivo llamado .travis.yml.

Travis también permite declarar nuestras propias variables de entorno, para que sean accesibles durante la *build*, y lo más importante, permite la opción de que vayan cifradas, por si tenemos que pasar una contraseña o un nombre de usuario.

2.5.2.1. Ciclo de vida

Travis tiene su propio ciclo de vida, que es la secuencia de pasos que va siguiendo en la *build* del proyecto.

Los pasos vistos en la figura 2.12 son los pasos obligatorios que todo proyecto ejecutado por Travis realiza. Para ello, debemos especificar en el archivo .travis.yml el comando o script que queremos ejecutar en cada uno de los pasos.

Además, existen tres pasos opcionales que permiten el despliegue de nuestro proyecto al entorno de producción (en caso de Travis, a GitHub). Esos pasos son: before_deploy, deploy y after_deploy. Éstos ocurren entre el paso after_success/_failure y after_script.

Ahora se va a explicar que ocurre en los principales pasos del ciclo de vida de Travis:

- **install:** Instala las dependencias requeridas.
- **script:** Ejecuta un script para realizar la *build* del proyecto.
- **after-success:** Acciones a ejecutar cuando la *build* ha sido exitosa.

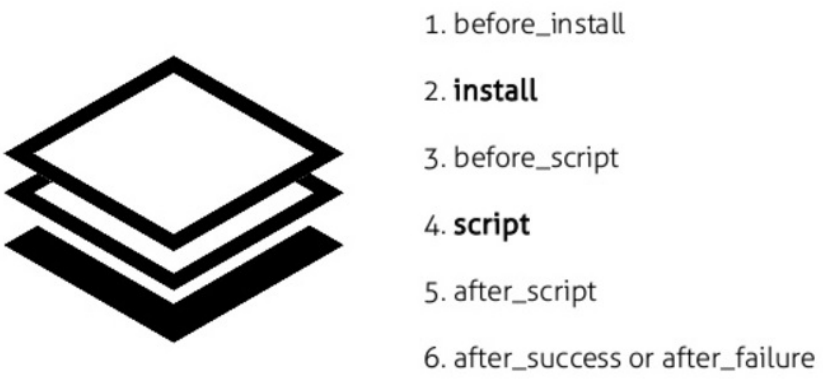

FIGURA 2.12: Ciclo de vida de Travis

- **after-failure:** Acciones a ejecutar cuando la *build* ha sido errónea.
- **deploy:** Realiza el despliegue de nuestra aplicación. Este paso necesita de parámetros para funcionar correctamente, tal como el proveedor (GitHub, Heroku, ...) o la rama desde la cuál se realizará el despliegue (siguiendo la metodología GitFlow (ver sección 2.2.2), la rama desde la que se hace el despliegue es `master`).

Capítulo 3

Desarrollo e integración continua

Para el desarollo del proyecto, lo primero que se ha hecho ha sido crear una copia del repositorio Net2Plan en GitHub para que las pruebas no afecten al código original.

A partir de ese punto, se ha integrado cada gestor de integración continua por separado, para hacer una visión general de cada uno de ellos aplicados al proyecto Net2Plan, y ver cuál se adapta mejor a los objetivos y necesidades.

3.1. Integración continua con Jenkins

Para empezar a trabjar con Jenkins [4], lo primero de todo es disponer de un servidor propio. En este caso, se ha utilizado el servidor Hannibal, perteneciente al grupo GIRTEL de la Universidad Politécnica de Cartagena. Dicho servidor funciona con sistema operativo Ubuntu Server LTS.

Ahora se instala Docker (ver 2.4) en Hannibal, ya que se necesita un contenedor Docker exclusivo para instalar Jenkins.

A continuación, debemos ejecutar el siguiente comando en la consola de Hannibal:

```
sudo docker run --name jenkins_girtel
-p 24601:8080 -p 49998:50000
-v /srv/docker/jenkins:/var/jenkins_home -d jenkins
```

Gracias a este comando, se ha creado un nuevo contenedor llamado jenkins_girtel, cuyo puerto 8080 está redireccionado al puerto 24601 del servidor Hannibal. Dicho comando nos enviará una respuesta con un código alfa-numérico, que hay que guardar, ya que nos servirá para crear la cuenta de usuario de Jenkins.

Para acceder al contenedor jenkins_girtel, simplemente hay que abrir un navegador web e introducir la siguiente URL:

```
hannibal.upct.es:24601
```

La primera vez que se accede a Jenkins, pide que se introduzca el código mencionado anteriormente. Una vez introducido, se crea la cuenta de administrador, para poder acceder a la gestión de tareas dentro del propio Jenkins.

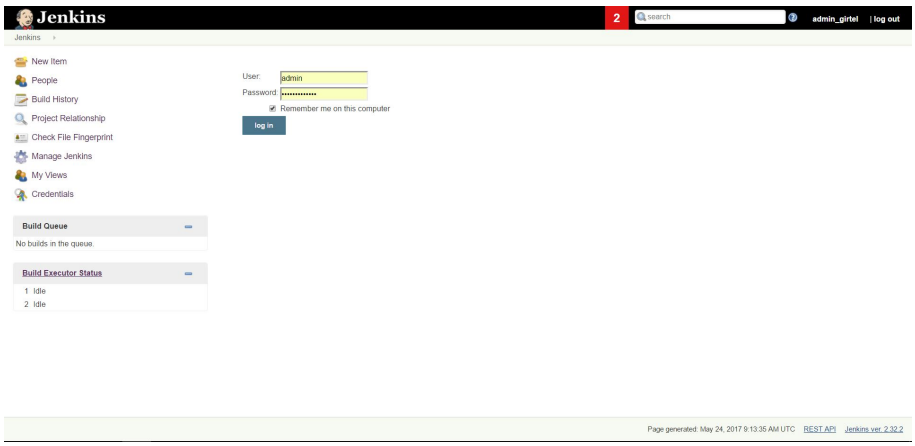

FIGURA 3.1: Ventana de inicio de Jenkins

En la foto 3.1 podemos ver la ventana de inicio de Jenkins. Pide autenticarse mediante usuario y contraseña, que se han creado previamente.

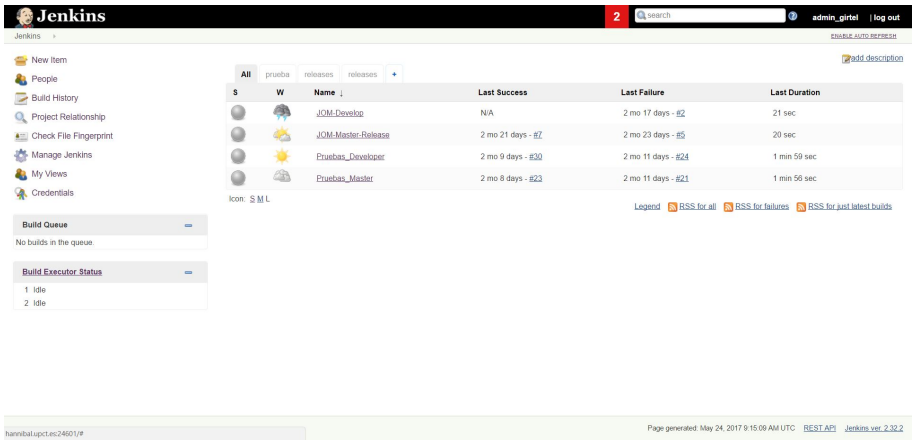

FIGURA 3.2: Ventana *DashBoard* de Jenkins

Si la autenticación es correcta, se mostrará la ventana de inicio de Jenkins (ver foto 3.2).

Aquí se pueden ver todas las tareas creadas, así como el menú de usuario donde se puede crear una nueva tarea, editar las actuales, cambiar la configuración, etc...

El primer paso que se debe hacer es dar permisos en GitHub para que Jenkins pueda obtener datos de los repositorios alojados. Para ello, el propietario del repositorio en GitHub debe acceder a los ajustes del propio repositorio para darle permisos a Jenkins (lo que se conoce como un *web-hook*).

Una vez hecho esto, para poder trabajar con un repositorio en Jenkins, se debe crear una nueva tarea.

En la sección 2.5.1 se vieron los distintos tipos de tareas que ofrece Jenkins, y concretamente, las tareas basadas en proyectos gestionados por Maven, que permiten obtener variables e información del POM fácilmente.

En concreto, se han creado dos tareas independientes: una para controlar la rama *master* y otra para controlar la rama *develop*.

3.1.1. Tarea Net2Plan-Master

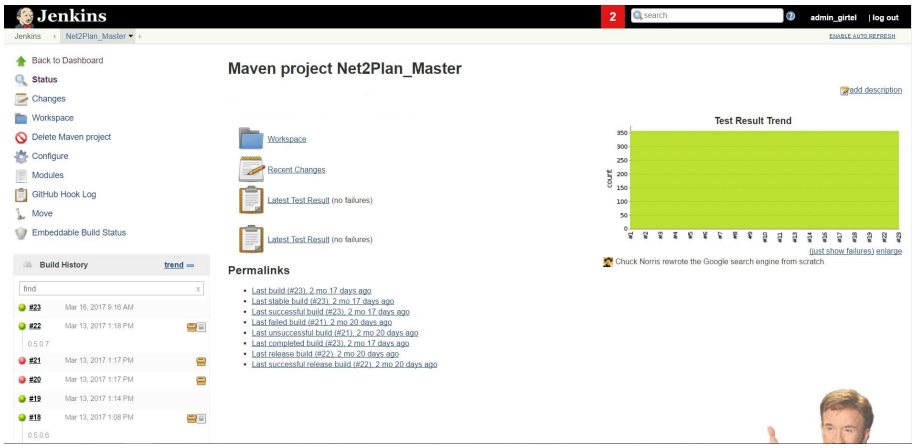

FIGURA 3.3: *Dashboard* de la tarea Net2Plan-master

En la foto 3.3 se puede ver la ventana principal de la tarea que controla la rama *master* de Net2Plan.

A continuación se explica el contenido de cada paso de la tarea:

1. **General:**

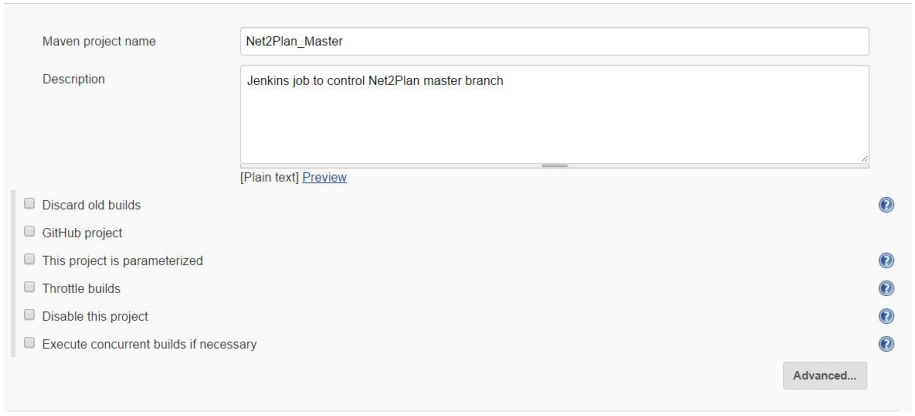

FIGURA 3.4: Paso *General* de la tarea Net2Plan-Master

En este paso se establecen el nombre y la descripción de la tarea.

2. **Source Code Management:**

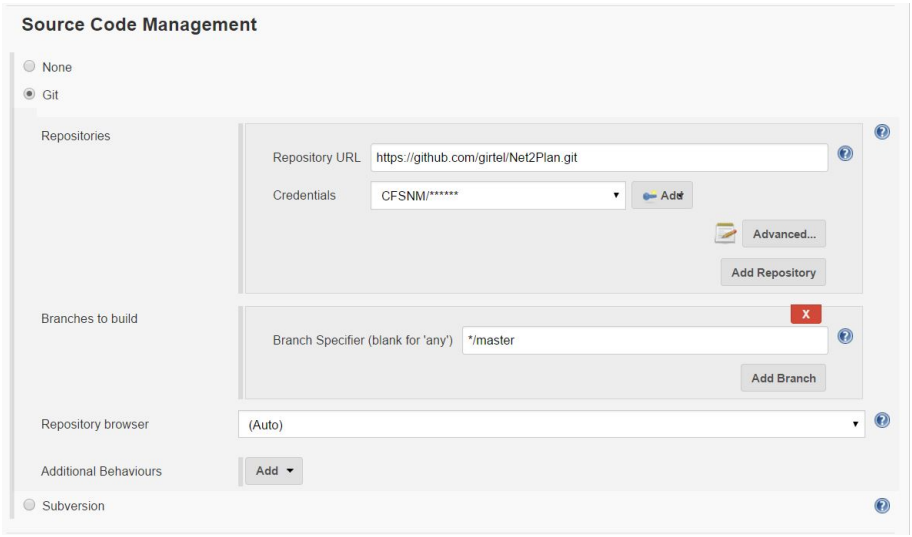

FIGURA 3.5: Paso *Source Code Management* de la tarea Net2Plan-Master

En este paso se establece Git como sistema de control de versiones. Dentro de esa opción, se establece girtel/Net2Plan como repositorio a controlar y sus credenciales. Así mismo, se indica que esta tarea controlará la rama *master*.

3. **Build Triggers:**

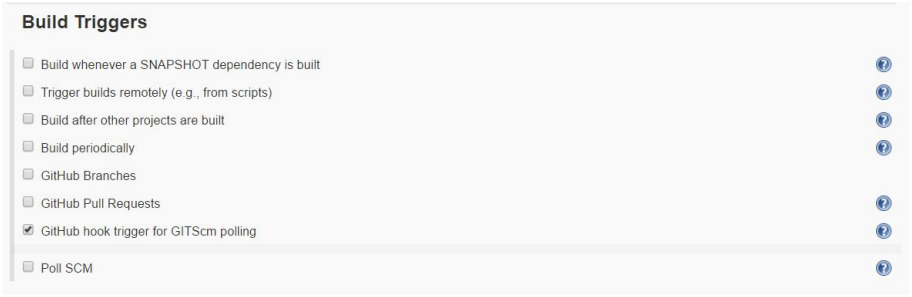

FIGURA 3.6: Paso *Build Triggers* de la tarea Net2Plan-Master

En este paso se establece que la ejecución de la *build* empezará cada vez que se haga un *push* desde Git a la rama *master*.

4. **Build Environment:**

FIGURA 3.7: Parte 1 del paso *Build Environment* de la tarea Net2Plan-Master

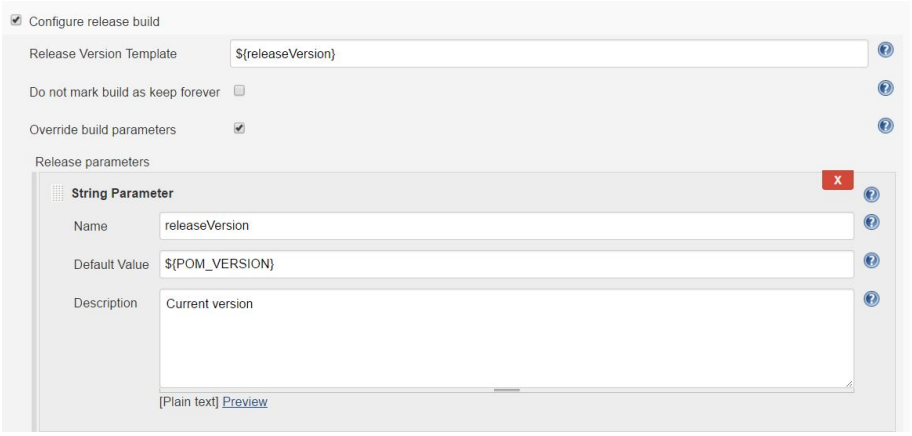

FIGURA 3.8: Parte 2 del paso *Build Environment* de la tarea Net2Plan-Master

En este paso se establecen marcas de tiempo en la consola para ver el tiempo que pasa en la ejecución de la *build.* y también se establece un registro de *releases*, para que Jenkins guarde un registro interno de la versión que corresponde a cada *build*.

5. **Build:**

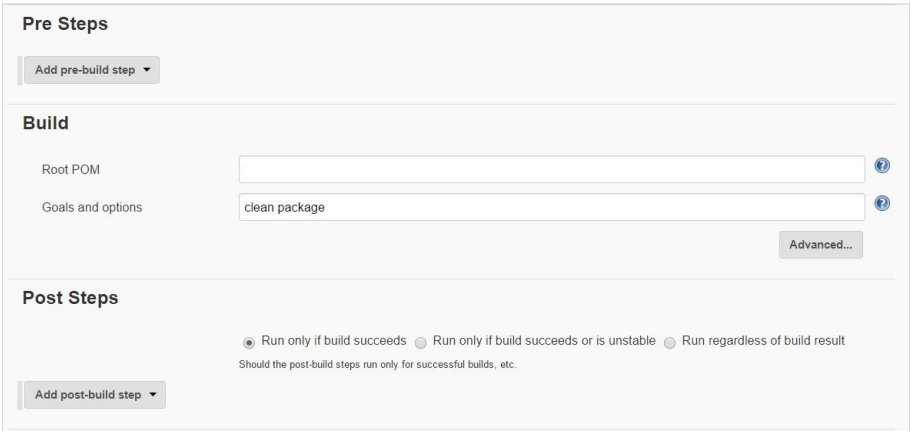

FIGURA 3.9: Paso *Build* de la tarea Net2Plan-Master

En este paso se establecen el archivo POM del repositorio (se deja en blanco, ya que Net2Plan tiene el nombre del POM por defecto) y los pasos a ejecutar de Maven: clean, para que se limpie el *workspace*, y package, para que se cree el ejecutable de Net2Plan.

6. **Build Settings:**

FIGURA 3.10: Paso *Build Settings* de la tarea Net2Plan-Master

En este paso se configuran las notificaciones por correo electrónico, para avisar en el caso de una *build* errónea.

7. **Post Build Actions:**

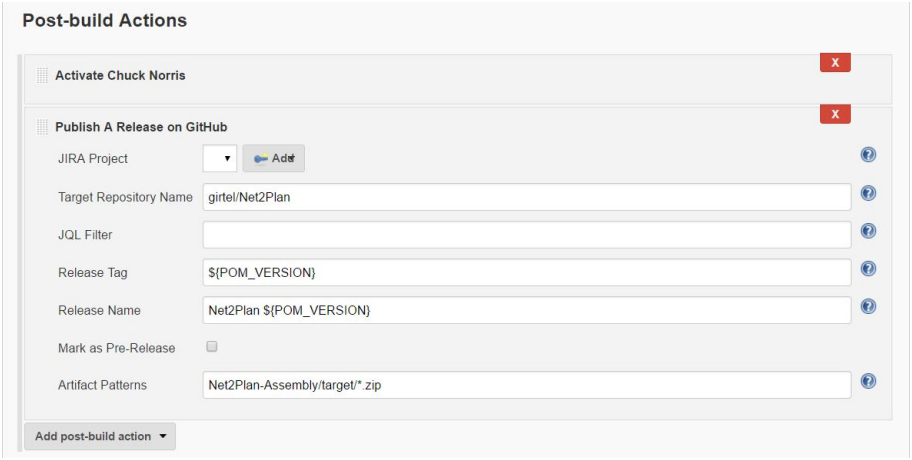

FIGURA 3.11: Paso *Post Build Actions* de la tarea Net2Plan-Master

En este paso se establece el Plugin de Chuck Norris (ver sección 2.5.1) y también se configura el despliegue a GitHub, estableciendo diferentes parámetros como el nombre de la nueva versión, o la ruta del ejecutable a subir.

3.1.2. Tarea Net2Plan-Develop

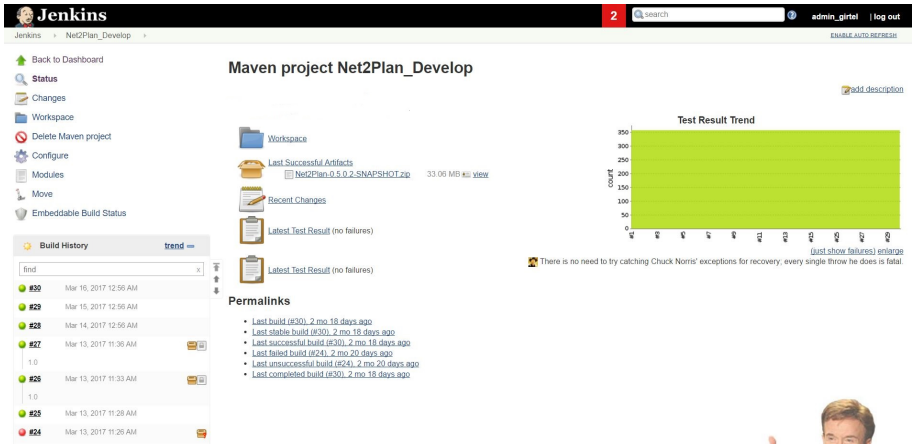

FIGURA 3.12: *Dashboard* de la tarea Net2Plan-develop

En la foto 3.12 se puede ver la ventana principal de la tarea que controla la rama *develop* de Net2Plan.

A continuación se explica el contenido de cada paso de la tarea:

1. **General:**

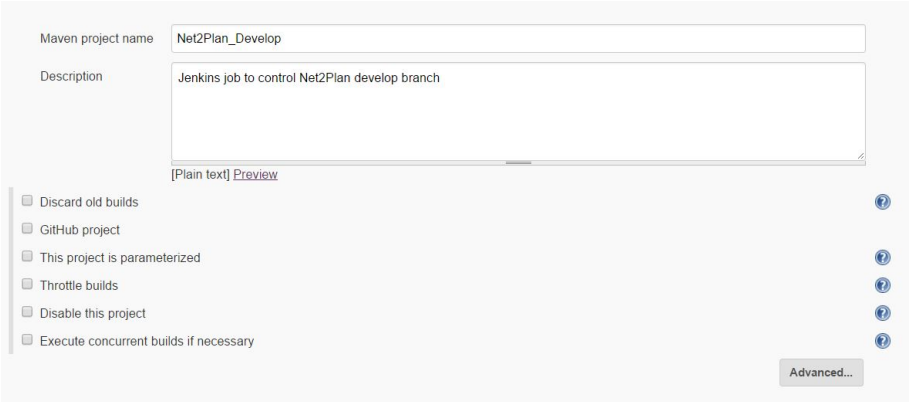

FIGURA 3.13: Paso *General* de la tarea Net2Plan-Develop

En este paso se establecen el nombre y la descripción de la tarea.

2. **Source Code Management:**

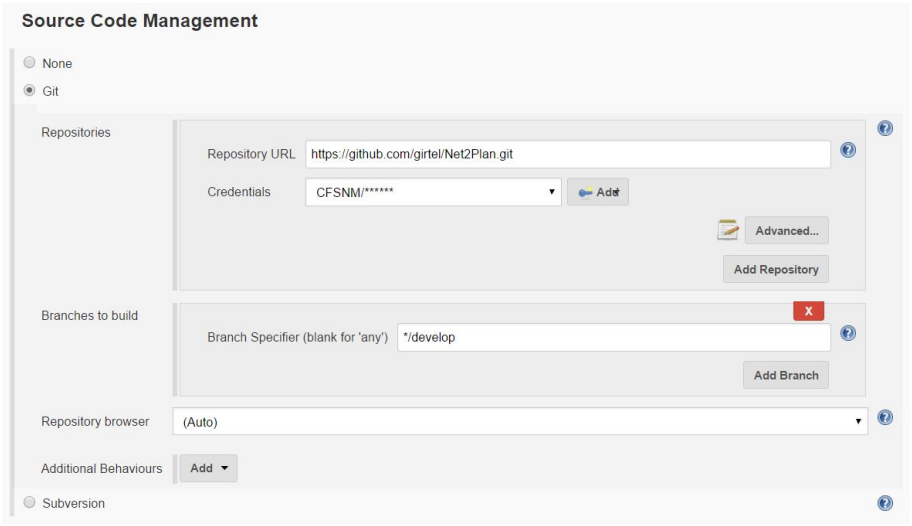

FIGURA 3.14: Paso *Source Code Management* de la tarea Net2Plan-Develop

En este paso se establece Git como sistema de control de versiones. Dentro de esa opción, se establece girtel/Net2Plan como repositorio a controlar y sus credenciales. Así mismo, se indica que esta tarea controlará la rama *develop*.

3. **Build Triggers:**

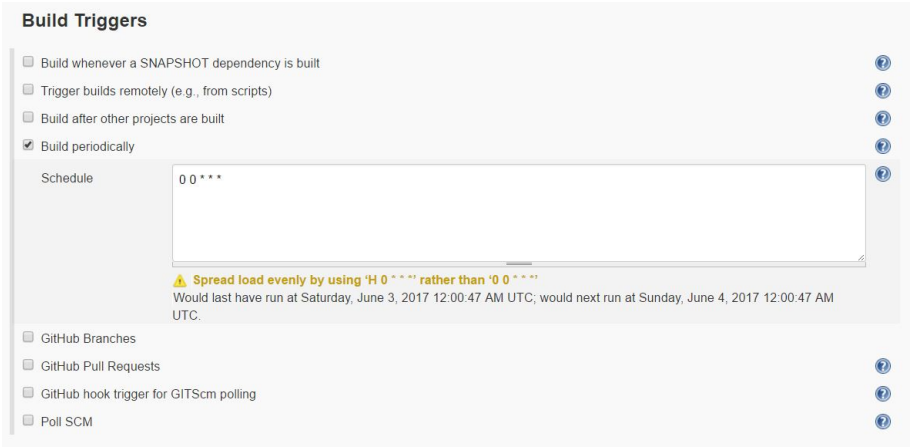

FIGURA 3.15: Paso *Build Triggers* de la tarea Net2Plan-Develop

En este paso se establece que la ejecución de la *build* se realizará una vez por día a las 12 de la noche. Dicha ejecución se programa mediante CRON, que permite ejecutar procesos de forma periódica.

4. **Build Environment:**

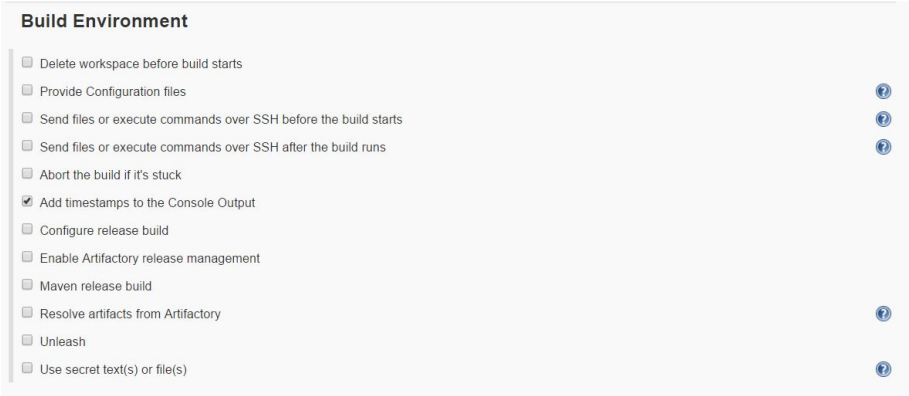

FIGURA 3.16: Paso *Build Environment* de la tarea Net2Plan-Develop

En este paso se establecen marcas de tiempo en la consola para ver el tiempo que pasa en la ejecución de la *build*.

5. **Build:**

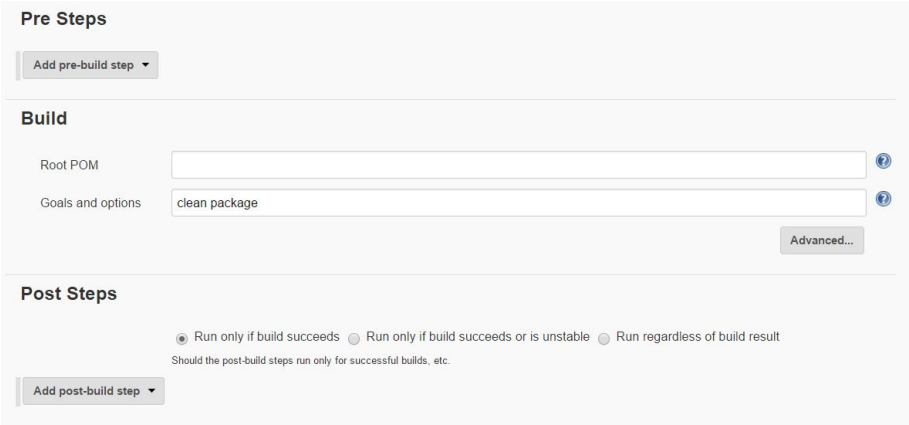

FIGURA 3.17: Paso *Build* de la tarea Net2Plan-Develop

En este paso se establecen el archivo POM del repositorio (se deja en blanco, ya que Net2Plan tiene el nombre del POM por defecto) y los pasos a ejecutar de Maven: clean, para que se limpie el *workspace*, y package, para que se cree el ejecutable de Net2Plan.

6. **Build Settings:**

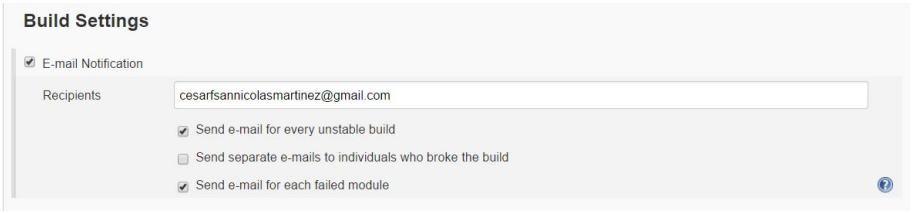

FIGURA 3.18: Paso *Build Settings* de la tarea Net2Plan-Develop

En este paso se configuran las notificaciones por correo electrónico, para avisar en el caso de una *build* errónea.

7. **Post Build Actions:**

FIGURA 3.19: Paso *Post Build Actions* de la tarea Net2Plan-Develop

En este paso se establece el Plugin de Chuck Norris (ver sección 2.5.1) y también se configura el almacenamiento del ejecutable resultante al realizar la *build* en el propio servidor Hannibal, para tener un registro de las *builds* diarias.

3.2. Integración continua con Travis

Para empezar a trabajar con Travis, lo primero de todo es acceder con la cuenta de GitHub a él.

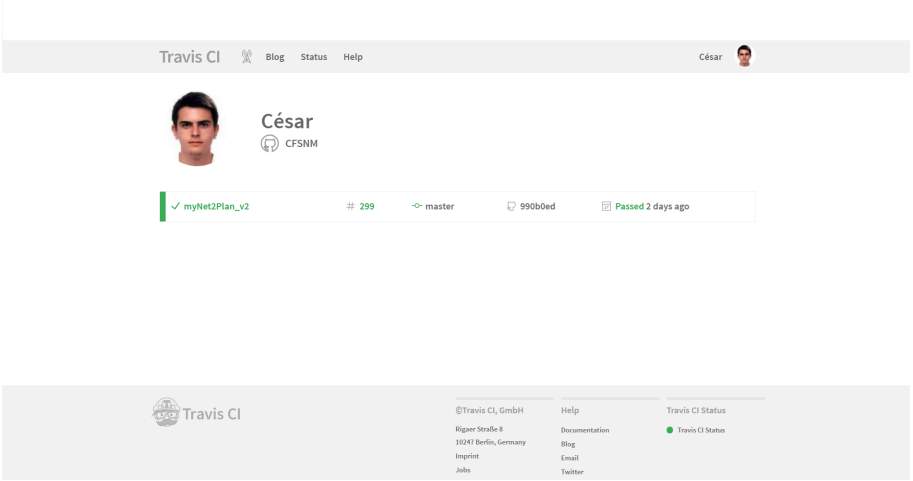

FIGURA 3.20: Ventana de usuario de Travis

En la foto 3.20 se pueden ver los repositorios asociados a la cuenta de GitHub e información acerca de su última *build*.

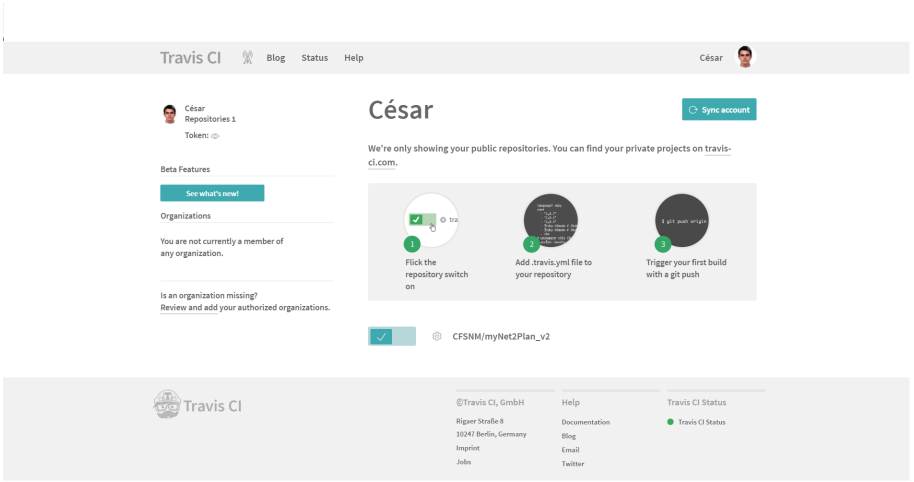

FIGURA 3.21: Ventana de configuración de usuario de Travis

En la foto 3.21 se puede ver la opción de elegir a cuales repositorios tendrá acceso Travis. Primero se pulsa el botón de sincronización, y a continuación, se marca el *tick* azul de los repositorios deseados.

En la figura 3.22 se puede ver información mas detallada de la última *build* del proyecto. Se puede observar el *commit* que ejecutó la build, el mensaje de dicho *commit*, el

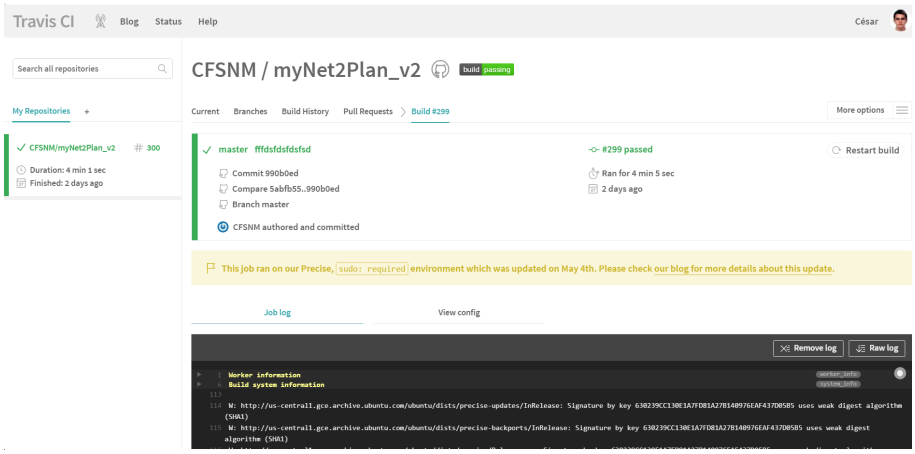

FIGURA 3.22: Proyecto Net2Plan en Travis

tiempo que ha durado, la fecha en la que se hizo, etc... Abajo, en la ventana negra, se ve el resultado de ejecutar el archivo .travis.yml.

A continuación, hay que acceder a los ajustes del repositorio en Travis para ajustar varios parámetros.

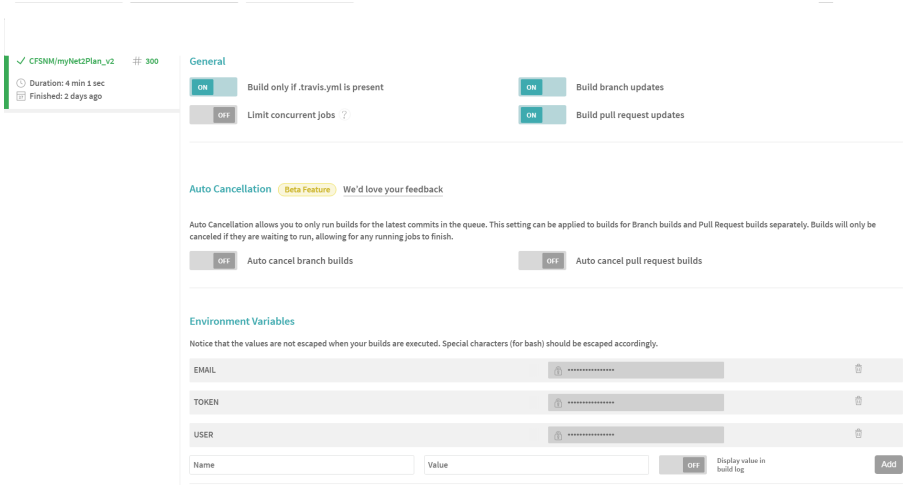

FIGURA 3.23: Ventana de ajustes de Net2Plan

En la foto 3.23 se ven distintas opciones para el proyecto. Se permite decidir si queremos que la *build* se ejecute cuando haya un *push*, un *pull-request* o ambos.

También permite configurar variables de entorno para utilizar en la *build* del proyecto. En el caso de Net2Plan, se han configurado tres:

- **USER:** El nombre de usuario de GitHub.
- **EMAIL:** El correo electrónico del usuario de GitHub.
- **TOKEN:** El Token de autenticación del repositorio.

Todas ellas irán cifradas, ya que son variables importantes que no deben ir en texto plano. Para eso existe la opción mediante un *tick* para que aparezcan en texto plano o no.

Ahora que ya está todo configurado según las necesidades, se añade el archivo .travis.yml al repositorio en GitHub y se realiza la primera *build* de prueba.

Ahora ya está todo listo para empezar a desarrollar el sistema de integración continua con Travis.

Lo primero que se debe hacer ahora es configurar el archivo .travis.yml para que realice nuestros objetivos.

3.2.1. Archivo .travis.yml

El archivo .travis.yml [5] se ha utilizado para Net2Plan es el siguiente, siguiendo el ciclo de vida de Travis explicado en la sección 2.5.2. A continuación se ve un desglose de los distintos pasos, así como una explicación de lo que ocurre en cada paso:

1. Primeros pasos

```
language: java
jdk: oraclejdk8
sudo: required
```

   Se indica el lenguaje del proyecto, el `JDK` (*Java Development Kit*) que se va a utilizar, ya que el lenguaje es Java, y que se necesitan permisos de root para llevar a cabo la *build*.

2. addons:

```
addons:
    apt:
        packages:
            - glpk
```

   En este paso se especifican los paquetes que hay que descargar para realizar correctamente la *build* y el despliegue. En el caso de Net2Plan, se necesita el *solver* GLPK para realizar los tests de algoritmos.

3. env:

```
env:
    - DISPLAY=:99.0
```

   En este paso se crea una variable de entorno llamada `DISPLAY` con valor 99. Esta variable es necesaria para ejecutar determinados tests.

4. before_install:

```
before_install:
    - chmod +x CHANGELOG.md &&
      chmod +x Net2Plan-CI/src/scripts/**
```

```
    - sh -e /etc/init.d/xvfb start
```

En este paso se dan permisos completos a los distintos archivos y *scripts bash* que se necesitan para llevar a cabo la *build* y el despliegue. También se arranca un driver para que Travis pueda llevar a cabo los tests referentes a la GUI de Net2Plan.

5. install:

```
install:
    true
```

En este paso se instalan dependencias necesarias para la ejecución de la *build*. Si no se pone nada, ejecuta unas dependencias por defecto que son innecesarias, así que para que salte este paso, se debe escribir *true*.

6. script:

```
script:
    mvn clean package
```

En este paso se ejecutan las tareas *clean* y *package*, que realizan la *build* del proyecto Net2Plan mediante Maven. Tras este paso, estará disponible el archivo zip listo para desplegarlo a producción.

7. after_success:

```
after_success:
    echo "Successful build"
```

En este paso se muestra un mensaje para indicar que la *build* ha sido exitosa.

8. after_failure:

```
after_failure:
    echo "Errored build"
```

En este paso se muestra un mensaje para indicar que la *build* ha sido errónea. En el caso que haya fallos y se ejecute este último paso, el despliegue a GitHub se detendrá.

9. before_deploy:

```
before_deploy:
  - VERSION=$(./Net2Plan-CI/src/scripts/get_version.sh)
  - BODY=$(./Net2Plan-CI/src/scripts/changelog_script.sh)
```

En este paso se ejecutan dos *scripts bash* [6]:

- El primero es el script get_version.sh (ver A.2), que se encarga de leer la versión del proyecto del archivo POM.. La versión se guarda en la variable `VERSION`.
- El segundo es el script changelog_script.sh (ver A.1), que se encarga de leer la descripción asociada a la nueva release del archivo `CHANGELOG.md`. Dicha descripción se guarda en la variable `BODY`.

Ambas variables se utilizarán en el despliegue.

10. deploy:

```
deploy:
        provider: releases
        api-key: $TOKEN
        body: "$BODY"
        tag-name: $VERSION
        name: "Net2Plan $VERSION"
        file_glob: true
        file: Net2Plan-Assembly/target/*.zip
        skip_cleanup: true
        on:
                branch: master
                repo: girtel/Net2Plan
                tags: true
```

En este paso se realiza el despliegue de Net2Plan a GitHub. Para ello, se necesitan varios parámetros de entrada para realizar la llamada remota a la `REST-API` de GitHub [7]:

- **provider: releases** ->indica que se trata de una nueva *release* en GitHub.
- **api-key: $TOKEN** ->en la variable `TOKEN` está almacenado el token de autenticación de la cuenta de GitHub. Es necesario para poder realizar la llamada remota.
- **body: $BODY** ->como se hablo anteriormente, en la variable `BODY` está almacenada la descripción de la nueva *release*.
- **tag-name: $VERSION** ->indica el nombre de la *tag* que se asociará a la nueva *release*.
- **name: "Net2Plan $VERSION"** ->indica el nombre que tendrá la nueva *release*.
- **file_glob: true** ->poniendo este parámetro a true se hace posible que puede haber más de un archivo que se deba subir a GitHub como ejecutable de la *release*.
- **file: Net2Plan-Assembly/target/*.zip** ->indica el directorio donde se encuentra el ejecutable (o los ejecutables) que se deben subir a GitHub. Al poner *.zip, se indica que se subirán todos los archivos con formato zip dentro del directorio.
- **skip_cleanup: true** ->poniendo este parámetro a true se consigue que el ejecutable creado en el servidor de Travis tras la *build* no se borre.
- **on:** ->esta opción indica las condiciones que deberán darse para el despliegue se lleve a cabo. En concreto, se han utilizado dos condiciones:
  - **branch: master** ->la rama debe ser master para que el despliegue a GitHub se realice.

- **repo: girtel/Net2Plan** ->el repositorio debe ser ese en concreto para que el despliegue a GitHub se realice.
- **tags: true** ->poniendo este parámetro a true se consigue que únicamente se realice el despliegue cuando se haya creado una *tag* previamente de forma manual.

11. notifications:

```
notifications:
    email:
        on_success: never
        on_failure: always
```

Como último paso se configuran los avisos por correo. Cuando la *build* sea errónea avisará por correo de dicho fallo, para detectar y solucionar los fallos lo más rápido posible.

Capítulo 4

Conclusiones

El objetivo inicialmente propuesto fue el de desarrollar un sistema de integración continua para la herramienta Net2Plan que permitiera detectar y solucionar errores de forma eficiente, así como realizar posteriormente un despliegue a GitHub para que cualquier persona interesada fuera capaz de descargar la última versión de Net2Plan fácilmente.

Para llevarlo a cabo, se propuso evaluar dos gestores de integración continua, Jenkins y Travis, para ver cuál se adaptaba mejor a las necesidades requeridas.

Una vez realizados dos sistemas de integración continua, cada uno de ellos basado en un gestor, se realiza una comparación entre ambos en función de las necesidades para decidir cuál de ellos será implantado.

4.1. Comparación entre Jenkins y Travis

Se va a realizar una comparación entre ambos gestores en función de diferentes características, como la instalación o la configuración, entre otras:

- Instalación

  Travis tiene la ventaja de que, al estar alojado en la nube, no necesita un servidor privado para poder ser utilizado. Simplemente se necesita una cuenta de GitHub y conexión a Internet.

  Jenkins, en cambio, necesita de un servidor privado corriendo Docker (ver sección 2.4) para funcionar.

- Configuración

  Para configurar Travis se necesita crear en la raíz del repositorio un archivo llamado .travis.yml, en el que se van indicando los pasos a realizar (ver sección 2.5.2.1).

  Para Jenkins, en cambio, se necesita crear una tarea para gestionar el repositorio. Dicha tarea tiene unos pasos específicos que hay que configurar mediante distintos *plugins* (ver sección 2.5.1). En función de las necesidades, hay que instalar unos *plugins* u otros.

- Detección de errores

  Ambos gestores utilizan la dependencia `Surefire` de *Maven* para realizar los tests.

  La ventaja que tiene Jenkins respecto a Travis aquí es que permite crear *reports* de los propios tests, y exportarlos al escritorio para leerlos detenidamente y ver más claramente donde se han producido los fallos.

- Despliegue a GitHub

  Para realizar el despliegue en Travis, es suficiente con incluir en el archivo .travis.yml el paso `deploy` (ver sección 3.2.1).

  Para realizar el despliegue en Jenkins, se necesita un plugin llamado *Jenkins Release Plugin*.

  La desventaja de Jenkins respecto a Travis en cuanto al despliegue es que se necesita de una cuenta en JIRA, plataforma privada creada por Attlasian para el seguimiento de proyectos de software, que es de pago.

A continuación se ve una tabla comparativa entre Jenkins y Travis que resume lo expuesto arriba:

Características	Jenkins	Travis
Instalación		x
Configuración	x	x
Detección de errores	x	
Despliegue a GitHub		x

TABLA 4.1: Comparación entre Jenkins y Travis

Finalmente, teniendo en cuenta ventajas y desventajas de cada gestor, se ha elegido el sistema de integración continua basado en Travis para implantar definitivamente en Net2Plan.

4.2. Trabajo futuro

Como propuesta de trabajo futuro, se proponen las siguientes mejoras:

- Realizar un mayor número de tests para la herramienta Net2Plan mediante JUnit, para que la integración continua realice su cometido de forma eficiente.
- Evaluar otros gestores de integración continua, como CodeShip o CircleCI, y ver posibles mejoras respecto a Jenkins y Travis.
- Seccionar la *build* en otras más pequeñas (una por módulo de Net2Plan). Únicamente se ejecutará la *build* de un módulo cuando dicho módulo haya sufrido cambios.

Anexo A

Scripts bash

A.1. Script changelog_script.sh

```
#! /bin/bash
LINEA=''
ID='##'
ID2='###'
ID3='-'
FIN='####'
READ=false
FINAL=false
while read -r i;
do
        LINEA=$i;
        if  [[ $i == *$ID* ]];
        then
                READ=true
        fi
        if [[ $i == *$FIN* ]];
        then
                echo -n '</ul>'
                break
        fi
        if [[ $READ == true ]];
        then
                if  [[ $i == *$ID2* ]];
                then
                        if  [[ $FINAL == true ]];
                        then
                                echo -n '</li>'
                                echo -n '</ul>'
                        fi
                        echo -n '<li>'
                        echo -n ${LINEA:3}
                        echo -n '<ul>'
```

```
                FINAL=true
            elif [[ $i == *$ID* ]];
            then
                echo -n $LINEA
                echo -n '</h2>'
                echo -n '<br/>'
                echo -n '<ul>'
            elif [[ $i == *$ID3* ]];
            then
                echo -n '<li>'
                echo -n ${LINEA:1}
                echo -n '</li>'
            fi
        fi
done < CHANGELOG.md
```

A.2. Script get_version.sh

```
#! /bin/bash

MVN_VERSION=$(mvn -q \
-Dexec.executable="echo" \
-Dexec.args='${project.version}' \
--non-recursive \
org.codehaus.mojo:exec-maven-plugin:1.3.1:exec)

echo "$MVN_VERSION"
```

Bibliografía

[1] "Net2plan: The open-source network planner," [Último acceso: Junio 2017]. [Online]. Available: http://www.net2plan.com

[2] "Martin Fowler - Continuous Integration," [Último acceso: Junio 2017]. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html

[3] "Phases of Continuous Integration," [Último acceso: Junio 2017]. [Online]. Available: http://www.bogotobogo.com/DevOps/Continuous_Integration_Phases.php

[4] "Jenkins Documentation," [Último acceso: Junio 2017]. [Online]. Available: https://jenkins.io/doc/

[5] "Travis CI User Documentation," [Último acceso: Junio 2017]. [Online]. Available: https://docs.travis-ci.com

[6] "Bash Scripting Tutorial," [Último acceso: Junio 2017]. [Online]. Available: https://linuxconfig.org/bash-scripting-tutorial

[7] "Releases - GitHub Developer Guide," [Último acceso: Junio 2017]. [Online]. Available: https://developer.github.com/v3/repos/releases/