Vicente Julio Casanova Pozo

ACTIVIDAD FINAL

Que es Git?

Es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente. Su propósito es llevar registro de los cambios en archivos de computadora y coordinar el trabajo que varias personas realizan sobre archivos compartidos.

Al principio, Git se pensó como un motor de bajo nivel sobre el cual otros pudieran escribir la interfaz de usuario o front end como Cogito o StGIT. 3​Sin embargo, Git se ha convertido desde entonces en un sistema de control de versiones con funcionalidad plena. 4​Hay algunos proyectos de mucha relevancia que ya usan Git, en particular, el grupo de programación del núcleo Linux.

El mantenimiento del software Git está actualmente (2009) supervisado por Junio Hamano, quien recibe contribuciones al código de alrededor de 280 programadores. En cuanto a derechos de autor Git es un software libre distribuible bajo los términos de la versión 2 de la Licencia Pública General de GNU.

GitHub

Es una forja (plataforma de desarrollo colaborativo) para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de código fuente de programas de ordenador. El software que opera GitHub fue escrito en Ruby on Rails. Desde enero de 2010, GitHub opera bajo el nombre de GitHub, Inc. Anteriormente era conocida como Logical Awesome LLC. El código de los proyectos alojados en GitHub se almacena típicamente de forma pública.

El 4 de junio de 2018 Microsoft compró GitHub por la cantidad de 7500 millones de dólares1​2​, al inicio el cambio de propietario generó preocupaciones y la salida de algunos proyectos de este repositorio3​, sin embargo no fueron representativos. GitHub continua siendo la plataforma más importante de colaboración para proyectos Open Source.

GitLab

Es un servicio web de control de versiones y desarrollo de software colaborativo basado en Git. Además de gestor de repositorios, el servicio ofrece también alojamiento de wikis y un sistema de seguimiento de errores, todo ello publicado bajo una Licencia de código abierto.

GitLab es una suite completa que permite gestionar, administrar, crear y conectar los repositorios con diferentes aplicaciones y hacer todo tipo de integraciones con ellas, ofreciendo un ambiente y una plataforma en cual se puede realizar las varias etapas de su SDLC/ADLC y DevOps.

Fue escrito por los programadores ucranianos Dmitriy Zaporozhets y Valery Sizov en el lenguaje de programación Ruby1​ con algunas partes reescritas posteriormente en Go, inicialmente como una solución de gestión de código fuente para colaborar con su equipo en el desarrollo de software. Luego evolucionó a una solución integrada que cubre el ciclo de vida del desarrollo de software, y luego a todo el ciclo de vida de DevOps. La arquitectura tecnológica actual incluye Go, Ruby on Rails y Vue.js.

Diferencias

Tanto GitHub como GitLab permiten mantener repositorios de código, tanto públicos como privados. La diferencia estriba en que, mientras que GitLab es siempre gratuito, GitHub solo te permite repositorios privados bajo suscripción. Pero no solo esto, también presentan otras diferencias:

Niveles de autentificación. Mientras que GitHub solo te da la opción de lectura o escritura en un repositorio, con GitLab puedes proporcionar acceso a determinadas partes del proyecto.

GitLab permite incluir adjuntos en un issue o problema.

GitLab permite marcar un proyecto como que está en desarrollo, lo que avisará a otros usuarios de la situación del mismo. Esto permite evitar que otros usuarios integren tu proyecto antes de tiempo…

GitHub tiene una comunidad de usuarios y una cantidad de proyectos muy superior a GitLab. Mientras que en GitLab hay poco mas de 100.000 proyectos, en GitHub superan los 35.000.000.

COMANDOS

Configurar Nombre que salen en los commits

git config --global user.name "dasdo"

Configurar Email

git config --global user.email dasdo1@gmail.com

Marco de colores para los comando

git config --global color.ui true

**Iniciando repositorio**

Iniciamos GIT en la carpeta donde esta el proyecto

git init

Clonamos el repositorio de github o bitbucket

git clone <url>

Añadimos todos los archivos para el commit

git add .

Hacemos el primer commit

git commit -m "Texto que identifique por que se hizo el commit"

subimos al repositorio

git push origin master

**GIT CLONE**

Clonamos el repositorio de github o bitbucket

git clone <url>

Clonamos el repositorio de github o bitbucket ?????

git clone <url> git-demo

**GIT ADD**

Añadimos todos los archivos para el commit

git add .

Añadimos el archivo para el commit

git add <archivo>

Añadimos todos los archivos para el commit omitiendo los nuevos

git add --all

Añadimos todos los archivos con la extensión especificada

git add \*.txt

Añadimos todos los archivos dentro de un directorio y de una extensión especifica

git add docs/\*.txt

Añadimos todos los archivos dentro de un directorios

git add docs/

**GIT COMMIT**

Cargar en el HEAD los cambios realizados

git commit -m "Texto que identifique por que se hizo el commit"

Agregar y Cargar en el HEAD los cambios realizados

git commit -a -m "Texto que identifique por que se hizo el commit"

De haber conflictos los muestra

git commit -a

Agregar al ultimo commit, este no se muestra como un nuevo commit en los logs. Se puede especificar un nuevo mensaje

git commit --amend -m "Texto que identifique por que se hizo el commit"

**GIT PUSH**

Subimos al repositorio

git push <origien> <branch>

Subimos un tag

git push --tags

**GIT LOG**

Muestra los logs de los commits

git log

Muestras los cambios en los commits

git log --oneline --stat

Muestra graficos de los commits

git log --oneline --graph

**GIT DIFF**

Muestra los cambios realizados a un archivo

git diff

git diff --staged

**GIT HEAD**

Saca un archivo del commit

git reset HEAD <archivo>

Devuelve el ultimo commit que se hizo y pone los cambios en staging

git reset --soft HEAD^

Devuelve el ultimo commit y todos los cambios

git reset --hard HEAD^

Devuelve los 2 ultimo commit y todos los cambios

git reset --hard HEAD^^

Rollback merge/commit

git log

git reset --hard <commit\_sha>

**GIT REMOTE**

Agregar repositorio remoto

git remote add origin <url>

Cambiar de remote

git remote set-url origin <url>

Remover repositorio

git remote rm <name/origin>

Muestra lista repositorios

git remote -v

Muestra los branches remotos

git remote show origin

Limpiar todos los branches eliminados

git remote prune origin

**GIT BRANCH**

Crea un branch

git branch <nameBranch>

Lista los branches

git branch

Comando -d elimina el branch y lo une al master

git branch -d <nameBranch>

Elimina sin preguntar

git branch -D <nameBranch>

**GIT TAG**

Muestra una lista de todos los tags

git tag

Crea un nuevo tags

git tag -a <verison> - m "esta es la versión x"

Que son las ramas?

Para entender realmente cómo ramifica Git, previamente hemos de examinar la forma en que almacena sus datos.

Recordando lo citado en [ch01-introduction], Git no los almacena de forma incremental (guardando solo diferencias), sino que los almacena como una serie de instantáneas (copias puntuales de los archivos completos, tal y como se encuentran en ese momento).

En cada confirmación de cambios (commit), Git almacena una instantánea de tu trabajo preparado. Dicha instantánea contiene además unos metadatos con el autor y el mensaje explicativo, y uno o varios apuntadores a las confirmaciones (commit) que sean padres directos de esta (un padre en los casos de confirmación normal, y múltiples padres en los casos de estar confirmando una fusión (merge) de dos o más ramas).

Para ilustrar esto, vamos a suponer, por ejemplo, que tienes una carpeta con tres archivos, que preparas (stage) todos ellos y los confirmas (commit). Al preparar los archivos, Git realiza una suma de control de cada uno de ellos (un resumen SHA-1, tal y como se mencionaba en [ch01-introduction]), almacena una copia de cada uno en el repositorio (estas copias se denominan "blobs"), y guarda cada suma de control en el área de preparación (staging area):

TIPOS DE RAMA

La principal característica de las ramas principales es que solo existe una de cada tipo. El objetivo es que no se instancien y que no reciban código de forma directa a través de commit, siempre tienen que recibir código a través de ramas de tipo Feature, Release y Hotfix, siempre a través de ramas auxiliares.

Es un riesgo recibir código directamente en la rama Master, porque puede generar defectos en el repositorio en las subidas a producción, que no contemplemos o que no preveamos, por lo que siempre es mejor integrar código en otras ramas antes de integrar con las ramas Master y Develop.

Esta es una metodología estricta pero que da lugar a diferentes interpretaciones o diferentes formas de llevarla en cada equipo, por lo que en algunos casos, algún experto puede permitirse no seguir esa norma, pero son casos muy específicos y siempre de personas de confianza.

En las ramas auxiliares tenemos la rama Feature, la rama Release y la Rama Hotfix, que puede instanciarse todas las veces que se consideren necesarias:

La rama Feature, para nuevas características, nuevos requisitos o nuevas historias de usuario.

La rama Release, para estandarizar o cortar una serie de código que ha estado desarrollándose en la rama Develop, se saca una rama de este tipo, se mergea y ahí se depura.

La rama Hotfix, que habitualmente se utiliza para código para depurar el código que venga de producción, por haberse detectado un defecto crítico en producción que deba resolverse, al que se le va a hacer una Release puntual para corregirlo.

Resolviendo conflictos

Algunas veces la unión de dos ramas no resulta tan bien, sino que ocurre un conflicto, esto cuando los commits de la rama a fusionar y la rama actual modifican la misma parte en un archivo en particular y git no puede decidir cuál versión elegir, y te avisa que tu debes resolverlo. Por ejemplo:

Supongamos que un directorio tenemos un archivo index.html con el siguiente contenido:

<!DOCTYPE HTML>

<html>

<head>

<title>Titulo</title>

</head>

<body>

<p>Contenido de la web</p>

</body>

</html>

Inicializaremos en el un repositorio en el mismo haciendo git init y luego haremos nuestro primer commit con git add index.html y luego git commit -m “commit inicial”

Vamos a crear una nueva rama para añadir algo de contenido:

git checkout -b contenido

Con ello ya estamos en la nueva rama y ahora vamos a cambiar el título.

Guardamos los cambios y hacemos commit en esa rama

git commit -a -m “cambios en el titulo”

Nos movemos de nuevo a la rama master

git checkout master

Hacemos otros cambios en el archivo incluyendo el título y luego commit de los cambios

git commit -a -m “se añade más contenido”

Ahora intentamos hacer merge con la rama creada anteriormente

git merge contenido

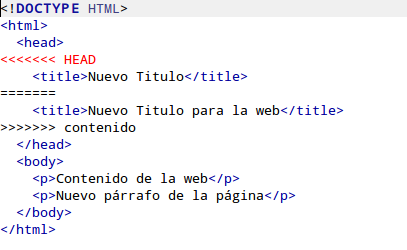
En este caso no podrá hacer el merge y nos mostrará que hay un conflicto que no nos permitirá continuar hasta que se resuelva:

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

Automatic merge failed; fix conflicts and then commit the result.

Git no proporciona una ayuda diciéndonos que archivo tiene el conflicto, el cual al abrirlo nos muestra cuáles son los cambios tanto de una rama como de la otra:



donde tenemos que elegir entre lo que está entre <<<<<<< HEAD y ======= que es contenido que tenemos en la rama donde estamos haciendo el merge (master) o entre ======= y >>>>>>> contenido donde están los cambios hechos en la rama que queremos unir (contenido).

Para ello arreglamos el archivo con los cambios elegidos, guardamos, agregamos y hacemos commit de los cambios

git commit -a

y de esta manera logramos hacer merge con éxito.

Otra manera para resolver los conflictos es que podemos indicarle de antemano a git que estrategia tomar cuando tiene que decidir un conflicto, esto con las opciones ours y theirs, de esta manera:

git merge -s recursive -X theirs rama-a-fusionar

Esto cuando queremos que git resuelva el conflicto usando los cambios de la rama a fusionar (theirs o suyos) y cuando queremos que tome los cambios de la rama donde se está fusionando (ours o nuestros):

git merge -s recursive -X ours rama-a-fusionar

### Deshaciendo merges

Si hemos realizado un merge con una rama con la que no queríamos, puedes hacer:

git reset --merge ORIG\_HEAD

### Unir automáticamente múltiples commits en uno durante el merge

Se puede unir todos los commits de una rama y fusionarlos en la rama actual especificando la opción --squash, es decir,

git merge --squash rama-a-fusionar

Merge es una de las alternativas que nos da git para unir las ramas de nuestro repositorio pero no es la única; en la próxima entrega conoceremos a rebase y su diferencia con merge.