

MP_0487. Entornos de desarrollo

UF4. Control de versiones

4.2. Herramientas para el control de versiones

Índice

☰	Objetivos	3
☰	Herramientas	4
☰	Introducción a GIT	7
☰	Componentes y funciones de GIT	9
☰	¿Cómo funciona?	12
☰	Instalación de GIT	14
☰	Uso de GIT por consola	16
☰	Enlaces de interés	23
☰	Resumen	24

Objetivos

Con esta lección perseguimos los siguientes objetivos:

- 1 Conocer las principales herramientas para el control del versiones.
- 2 Aplicar las técnicas para sincronizar proyectos con servidores GitHub y resolver discrepancias.
- 3 Reconocer y utilizar los principales comandos de GIT para configurar un repositorio local.

¡Ánimo y adelante!

Herramientas

Los sistemas de **gestión de versiones** de código fuente son herramientas que nos permiten trabajar en equipo en el desarrollo software y mantener un histórico de versiones del proyecto en el que se trabaja.

Existen una gran variedad de herramientas que proporcionan un sistema de control de versiones. Cada una incorpora una serie de características que habrán de ser tenidas en cuenta a la hora de seleccionar el sistema que más se acerque a nuestros requerimientos.



Se pueden enunciar las siguientes **características** básicas:

- 1 Tipo de repositorio que incorpora: central o distribuido.
- 2 Tipo de despliegue: exclusivos o colaborativos.
- 3 Soporte para “commit” atómicos.
- 4 Soporte para gestión de directorios.
- 5 Capacidad de mostrar árboles de otros sistemas.
- 6 Protocolos de red soportados (HTTP, SSH, FTP, SMTP, POP3).
- 7 Soporte de etiquetas.
- 8 Posibilidad de firmar versiones.
- 9 Soporte con sistemas operativos.
- 10 Tipo de cliente de acceso: propio o de fabricante externo.
- 11 Integración con herramientas de desarrollo.
- 12 Posibilidad de integración de sistemas de autenticación estándar.
- 13 Tipo de licencia.
- 14 Compatibilidad con nombres Unicode.
- 15 Capacidad de almacenamiento del repositorio.

A continuación se expone un cuadro que relaciona los principales sistemas de control de versiones con características básicas obtenidas de la fuente “Wikipedia-Inglés”.

**Tabla Control Version.pdf**

291.7 KB



Introducción a GIT

GIT es un sistema de control de versiones de código fuente. Fue inicialmente diseñado y desarrollado por Linus Torvalds para el desarrollo del núcleo Linux.

Es una herramienta de **software libre** distribuido bajo los términos de la Licencia Pública General GNU versión 2. Es importante conocer el ciclo de vida del software para poder manejar esta herramienta.

En GIT se trabaja con el concepto de **ramas o "branches"**, es decir, se puede estar trabajando en diferentes versiones, funcionalidades o características si tocar el código principal.

GIT Puede usarse integrado en un IDE o a través de una instalación externa. GIT tiene instrucciones que se pueden usar a través de la **línea de comandos**, aunque existen herramientas gráficas como **Egit** en Eclipse.

GIT permite la creación de una historia para una colección de archivos e incluye la funcionalidad para revertir la colección de archivos a otro estado. Otro estado puede significar otra colección diferente de archivos o contenido diferente de los archivos.

Para efectuar cambios en la colección de archivos primero se añaden al "*stage index*" y después se agregan al repositorio "*commit*".

GIT mantiene todas las versiones. Por lo tanto se puede revertir a cualquier punto en la historia del código fuente. GIT permite clonar repositorios incluyendo la historia completa, sincronizar repositorios vía "*push*" transfiriendo los cambios a un remoto o "*pull*" obteniendo los cambios de un remoto.

Las ventajas de un sistema como GIT frente a otro centralizado son:

Libre y de código abierto

GIT es un **software libre** bajo la licencia de código abierto GPL. Además de estar disponible libremente en Internet, GIT puede ser usado sin tener que pagar. El código puede ser descargado y modificado si es necesario.

Rápido y pequeño

GIT no se basa en un servidor central, por eso no hay necesidad de interactuar con el servidor remoto para cada operación. Además está construido en lenguaje C y es muy liviano. El tamaño de los datos que reflejan la estructura del servidor es muy pequeño. Por lo tanto GIT es un programa liviano y muy eficiente.

Respaldo implícito

Al ser un sistema descentralizado cada cliente es una copia del servidor, por lo que es muy difícil perder todos los datos.

Seguridad

GIT utiliza una función *hash* criptográfica (SHA1) para nombrar e identificar objetos dentro de la base de datos. Cada *id* es de 160 bits de largo y son representados en hexadecimal.

Pocos recursos

En caso de controladores centralizados el servidor debe ser muy potente y tener bastante memoria. En el caso de GIT no es tan necesario, pues el proceso se realiza en los clientes y solo se interactúa con el servidor para actualizar datos.

Ramificación sencilla

La gestión de ramas en un servidor GIT es muy rápida y sencilla, pues no copia todo, si no solo los cambios.

Tutorial de GIT

En este enlace encontrarás un tutorial en inglés con todo lo que necesitas saber sobre GIT y su uso.

Componentes y funciones de GIT

En un sistema de control de versiones GIT podemos identificar los siguientes elementos y acciones:

Local repository

Todas las herramientas de versionado de código proporcionan un lugar privado de trabajo. En ese lugar los programadores trabajan en su parte de código y cuando el trabajo está validado se puede efectuar un “commit” hacia el repositorio central. Esos cambios se reflejarán desde ese momento en dicho repositorio.

En GIT la copia privada es de todo el repositorio, no de solo una parte, por lo tanto el programador podrá cambiar, añadir y eliminar cualquier fichero. Contiene la historia, las diferentes versiones en el tiempo y los distintos “branch” y etiquetas. Cada copia del repositorio es un repositorio completo. El repositorio permite obtener revisiones en la copia actual local.

Working directory and staging area or index

El directorio de trabajo es el lugar donde los archivos están desprotegidos y es en ese lugar donde el programador puede realizar cambios. Los cambios realizados se reflejarán directamente en el repositorio. GIT usa otra estrategia diferente, ya que no hace seguimiento a todos y cada uno de los archivos modificados, sino que cuando hacemos “commit” se buscan los archivos presentes en el área de preparación o “stage index” y solo se consideran los archivos presentes en dicha área (y no todos los archivos modificados) para la confirmación “commit”.

Blobs

“Blob” abreviatura de “Binary Large Object”. Cada versión de un archivo es un “blob” que contiene solo los datos de los archivos y no los metadatos sobre el archivo. Es un archivo binario y la base de datos de GIT nombra con el *id hash* SHA1 ese archivo. Por lo tanto, los archivos en GIT no tienen nombre.

Trees

Representan un directorio y contienen “*blobs*” y subdirectorios. Almacenarán ficheros “*blob*” y otros “*trees*” nombrados con el *id hash* SHA1.

Commits

Sirven para **mantener el estado del repositorio**. Se nombran con el *id hash* SHA1. Cada uno es un elemento de la lista y está enlazado con un “*commit*” padre. Si un “*commit*” tiene varios padres es que se ha creado juntando dos “*branches*”. Se ejecuta por lo tanto un “*commit*” de los cambios a un repositorio. Esto crea una revisión nueva, la cual puede ser obtenida después, por ejemplo si se desea ver el código fuente de una versión anterior. Cada “*commit*” contiene el autor y el que realizó dicho “*commit*”, siendo así posible identificar el origen del cambio.

Branches

Cada **rama** o “**branch**” se crea para ejecutar una nueva línea de desarrollo o versión. GIT tiene una rama genérica. Cada rama que se crea, una vez finalizada se fusiona con la principal. Cada rama se referencia con un “HEAD”, que enlaza con el último “*commit*” en la rama. Por lo tanto, cada vez que hacemos “*commit*”, “HEAD” se actualizará. Un “*branch*” es una línea separada de código con su propia historia. Es posible crear un nuevo “*branch*” a partir de uno existente y cambiar el código independientemente de los otros “*branches*”. Uno de los “*branches*” es el original (generalmente llamado *master*). El usuario selecciona un “*branch*” y trabaja en él, que será llamado copia actual (*working copy*).

Tags

Cada “tag” es un nombre para una versión específica del repositorio. Son parecidas a las “*branches*”, pero son inmutables. Con un *tag* es posible tener un punto con un tiempo al cual siempre sea posible revertir el código.

Clone

Genera instancias en el repositorio. Trabaja con el área de trabajo y refleja el repositorio completo. El repositorio local es trabajado por el programador y se sincroniza con el general del servidor de vez en cuando.

Pull

Permite copiar los cambios de una instancia del repositorio remoto a uno local. Esta operación se utiliza para sincronizar dos instancias de repositorio.

Push

Permite copiar los cambios de una instancia del repositorio local a uno remoto. Sirve para almacenar los cambios de forma definitiva en el repositorio GIT.

HEAD

Es el enlace o puntero al último "commit" del último "branch". Siempre que se realice un "commit" este puntero se actualiza. Los punteros "HEAD" de los "branches" se almacenan en el directorio `/refs/heads/.git`.

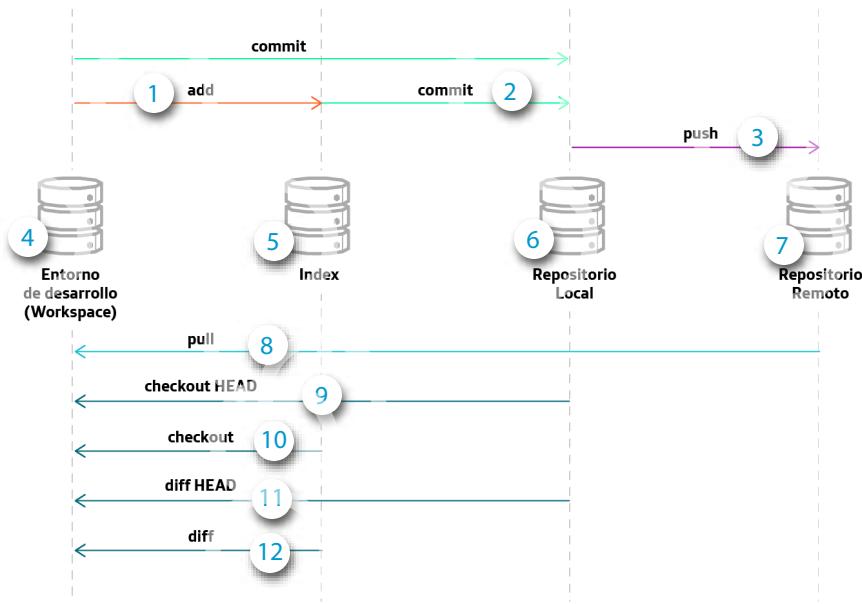
Revision

Representa una versión del código fuente. Las revisiones en GIT están representadas por "commit". Estos "commit" son identificados por su *id hash SHA1*.

URL

Representa la ubicación del repositorio GIT. Está escrita en el fichero de configuración de GIT.

¿Cómo funciona?



1 Add

Añade archivos al índice sin producir el *commit*. Podemos ir añadiendo diferentes documentos mientras realizamos nuestro desarrollo, que se subirán cuando realicemos el *commit*.

2 Commit

Realiza el guardado de la versión en el repositorio. Una de las opciones del *commit* es poner un nombre a la versión almacenada, así como una descripción del trabajo realizado.

3 Push

Copiar los cambios de una instancia del repositorio local a uno remoto. Sirve para almacenar los cambios de forma definitiva en el repositorio GIT.

4 Workspace

El directorio de trabajo es el lugar donde los archivos están desprotegidos, y es en ese lugar donde el programador puede realizar cambios.

5 Staging index

Área donde se controlan los cambios. Aquí se almacenan los cambios en el árbol de trabajo antes de la confirmación.

6 Repositorio local

El repositorio local contiene el historial de las diferentes versiones que se han subido a lo largo del tiempo y las diferentes ramas y etiquetas.

Cada copia del repositorio es un repositorio completo. El repositorio permite obtener revisiones en la copia actual local.

7 Repositorio remoto

Siendo un sistema distribuido, este es otro repositorio. Aquí podemos guardar cualquiera de las ramas o la última. Un ejemplo de uso podría ser GitHub.

8 Pull

Copiar los cambios de una instancia del repositorio remoto a uno local. Esta operación se utiliza para sincronizar dos instancias de repositorio.

9 Checkout HEAD

HEAD

Es el enlace o puntero al último "commit" de la última "branch". Siempre que se realice un "commit" este puntero se actualiza. Los punteros "HEAD" de las "branches" se almacenan en el directorio "/refs/heads/.git".

Checkout

Copia de trabajo local en su equipo a partir de una versión procedente del repositorio.

10 Checkout

Copia de trabajo local en su equipo a partir de una versión procedente del repositorio.

11 diff HEAD

Muestra los cambios entre una versión indicada y la versión en la que nos encontramos actualmente.

12 Diff

Muestra los cambios entre las diferentes versiones según el índice actual y el almacenado en el repositorio.

Instalación de GIT

El control de versiones con GIT está disponible para las tres principales plataformas: Windows, Linux y OSX. A continuación vamos a ver el procedimiento de instalación en cada una de ellas.

Windows

En Windows vamos a instalar un terminal mejorado llamado [Cmder](#), que incorpora **GIT** de serie, así como otros comandos básicos de Linux.

La instalación es desatendida. Al terminar tendremos nuestro control de versiones en marcha con solo ejecutar Cmder.

Linux

GIT se encuentra en el repositorio de paquetes de los sistemas Linux, por lo que solo tendremos que invocar su instalación desde la línea de comandos. Dependiendo de tu distribución (tu gestor de paquetes):

Debian/Ubuntu

```
sudo apt-get install git-core
```

RedHat/CentOS

```
yum install git-core
```

OSX

Para realizar la instalación en ordenadores con sistema OSX de Apple vamos a emplear **Homebrew**, que es un gestor muy útil para tener acceso a paquetes de distribuciones Linux.

Si no tenemos instalado aún Homebrew lo instalaremos con la siguiente línea de comandos:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Una vez instalado Homebrew podemos instalar GIT ejecutando:

```
brew install git
```

Ayuda para la instalación de GIT

Tutorial de instalación de GIT para cualquier plataforma: Windows, OSX y Linux.

INSTALACIÓN DE GIT

Uso de GIT por consola

Para usar **GIT** en el terminal primero tenemos que conocer algunos **comandos básicos del CLI** (línea de comandos o Command Line Interface) de **Linux**:

Pwd

Saber la carpeta actual.

Cd

Cambiar de carpeta.

Cd ..

Retroceder una carpeta.

Mkdir

Crear carpeta. P. ej.: *mkdir carpeta*.

Cat

Ver el contenido de un archivo de texto. P. ej.: *cat archivo.cat*.

Vi

Editor de texto por consola.

Rm

Borrar archivo. P. ej.: *rm nombreDeArchivo*.

El uso del editor Vi por consola puede resultar algo complicado si no tenemos experiencia con su funcionamiento. A continuación puedes ver un esquema con sus funciones básicas y los atajos de teclado utilizados.

vi resumen	Al iniciar vi comienzas en Modo Comando Al pulsar ESC regresas de nuevo al Modo Comando
SALIR Salir si no ha habido cambios :q Salir si no ha habido cambios :q! Guardar cambios y salir :wq	MODO COMANDO-MOVIMIENTO Muchos comandos pueden ser precedidos por un número de repeticiones  Moverse 5 palabras hacia adelante 5w Moverse 8 líneas hacia arriba 8k Ir a la línea 1 1G Ir a la línea 15 15G Ir al final del fichero G Preguntar dónde estoy ctrl+G
BUSCAR Realizar una búsqueda /consulta Repetir búsqueda :n	MODO COMANDO-EDICIÓN Pueden incluir cualquier Comando-Movimiento Cambiar al final de la línea c\$ Borrar 5 palabras d5w borrar hasta el final de la línea D Eliminar la línea actual dd Cortar la línea actual cc Copiar la línea actual yy Pegar antes P o después p Deshacer el último cambio u Repetir el último comando .
	MODO INserCIÓN Añadir aquí a Añadir al final de la línea A Insertar aquí i Insertar al comienzo de la línea I Reemplazar uno o varios caracteres R

Atajos de teclado del editor Vi, integrado en todos los sistemas Linux.

Primeros pasos

Una vez instalado GIT, lo primero es crear un repositorio en la carpeta del “proyecto”.

Para crear la carpeta ponemos:

md proyecto

Entramos en la carpeta con:

cd proyecto

El siguiente comando nos imprimirá en pantalla la ubicación actual a nivel de carpetas:

`pwd`

Ahora, ya situados en la carpeta, para iniciar el repositorio ponemos:

`git init`

Luego configuramos el nombre y email para que quede esa información en el repositorio:

`Git config --global user.email "tumail"`

`Git config --global user.name "tunombre"`

Antes de continuar vamos a crear algunos archivos de texto dentro de nuestra carpeta para luego añadirlos al repositorio. Lo podemos hacer con el comando `touch` de Linux, que nos crea archivos de texto plano vacíos:

`touch archivo.txt`

`touch archivo2.txt`

Creando nuestro primer `commit`

Ahora que ya tenemos el repositorio, podemos añadir archivos al estado intermedio (listos pero no subidos):

`Git add nombreArchivo`

Si queremos subir todo podemos usar los comodines (con el `"."` indicamos que queremos todo):

`Git add .`

Aún no hemos subido nada (lo hemos colocado en el `index`), pero podemos ver su estado con:

`Git status`

Para crear una versión y subirla al repositorio solo tenemos que lanzar el `commit`:

`Git commit`

Esto subirá los cambios al repositorio (seguimos en nuestro PC o servidor):

- Se abre un archivo de texto donde la primera línea es el título y a partir de la tercera es la descripción.
- El editor es Vi; recordamos que con `:a` pasamos al modo edición, con `ESC` vamos al modo comando y con `:wq` guardamos y salimos.

Aunque lo recomendable es poner la descripción, podemos poner el atributo `-m` en el comando, esto no abrirá el editor subiendo directamente el repositorio:

Git commit -m "titulo del repositorio"

Para ver los *commit* que hemos realizado usamos:

Git log

```
ciberkaos@MSTW10pro MINGW64 ~/eclipse-workspace2/Calculadora/src (master)
$ git log
commit a08da5e319713bba87fcb4136f536eeea0b50555 (HEAD -> master)
Author: Antonio <antonio.oter@telefonica.net>
Date:   Thu Nov 16 18:53:20 2017 +0100
        primera version
```

Ejemplo de visualización de *commits* con GIT Log

Si queremos una visión mas compacta:

Git log --oneline

```
$ git log --oneline
a08da5e (HEAD -> master) primera version
```

Cambiando de versión con *checkout*

Con GIT podemos trabajar o ver el código en cualquiera de los estados guardados. Para cambiar entre ellos tenemos el atributo *checkout*.

Como ya hemos visto en el punto anterior, para ver un listado de los *commit* que tenemos en el repositorio tenemos:

git log --oneline

Este comando nos retorna el listado de *commits* y podemos ver que el sistema ha etiquetado cada cambio con una referencia. Esta referencia nos servirá para identificar la versión al realizar el *checkout*:

Git checkout a08da5e

Podemos comprobar que al realizar el *checkout* la cabecera (HEAD) vuelve a estar en la versión anterior de nuestro código.

```
ciberkaos@mstw10pro MINGW64 ~/eclipse-workspace2/Calculadora/src (master)
$ git checkout a08da5e
Note: checking out 'a08da5e'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at a08da5e... primera version
I
ciberkaos@mstw10pro MINGW64 ~/eclipse-workspace2/Calculadora/src ((a08da5e...))
$ git log --oneline
a08da5e (HEAD) primera version
```

Usando tags

Poner la referencia al realizar el *checkout* puede resultar un poco lioso. Para facilitarnos el trabajo tenemos los *tag*.

Si estoy en la versión que quiero etiquetar:

Git tag v01 (v01 sería la etiqueta, puedo poner lo que quiera).

Si no lo estoy se la puedo asignar dando la referencia del *commit*:

Git tag v02 o8ae584

Creando una rama

Para crear una nueva rama tenemos el comando *branch*:

Git branch nuevaRama

Para ver la rama en la que nos encontramos podemos usar:

Git branch



```
$ git branch
* (HEAD detached at a08da5e)
  master
  nuevarama
```

El * indica la rama en la que nos encontramos.

Para ir a la rama determinada usamos el *checkout*:

Git checkout nuevarama

Si quiero crear la rama e ir a ella puedo atajar:

Git checkout -b nuevarama2

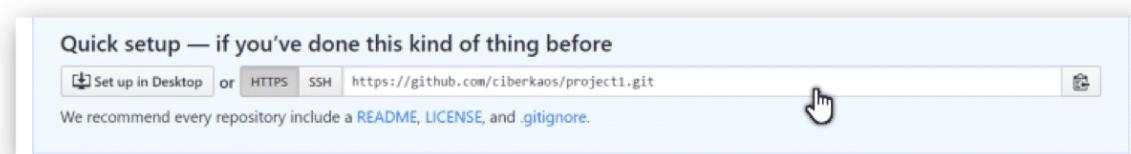
Subiendo un repositorio a remoto

Para realizar una subida a un repositorio remoto vamos a usar GitHub.

Lo primero que tenemos que hacer es una cuenta de GitHub y luego le damos a crear nuevo repositorio (público o privado).

Finalmente, teniendo la dirección de repositorio, podemos indicar la conexión remota y realizar el *push*:

```
git remote add origin https://github.com/ciberkaos/project1.git  
git push -u origin master
```



Resultado del repositorio subido a GitHub.

Enlaces de interés

A continuación tienes unos enlaces que te pueden servir de ayuda para profundizar en el uso de GIT.

Instalación de GIT

Instalación de GIT para cualquier plataforma: Windows, OSX y Linux.

[INSTALACIÓN DE GIT](#)

Vídeo de uso de GIT

Vídeo con instrucciones básicas por consola.

[GIT POR COMANDOS](#)

Manual de GIT

Manual completo de GIT.

[MANUAL DE GIT](#)

Comandos GIT

Hoja de referencia de los comandos básicos GitHub.

[COMANDOS GIT](#)

Resumen

Has finalizado esta lección.

GIT surgió como solución a los desarrolladores del *kernel* de Linux en 2005. Necesitaban una herramienta en la que multitud de desarrolladores pudieran colaborar con la misma base de código. Anteriormente se trabajaba con parches de código, que se pasaban en una lista de distribución y que hacía todo el proceso muy complejo y con una alta probabilidad de error.

En esta unidad hemos aprendido cómo instalar este sistema de control de versiones y su uso por la terminal de comandos.



PROEDUCA