

4.- Sistemas de control de versiones.

Caso práctico

En **BK programación**, debido al incremento de trabajo que están teniendo últimamente, han pensado en ampliar la plantilla. Su intención es ofrecer un contrato de trabajo a **Carlos**, pues sus conocimientos de diseño web serán de utilidad para trabajar como programador en la empresa.

Ada se está dando cuenta que, a medida que la empresa crece, el número de proyectos aumenta al igual que lo va a hacer la plantilla de personal. Por eso cree fundamental instalar un sistema de control de versiones para facilitar la integración del código fuente y demás documentos, de cada uno de los programadores, a los respectivos proyectos en desarrollo; de este modo la integración conjunta del trabajo individual de cada uno de los empleados será más sencilla, controlada y existirán nuevas posibilidades en cuanto a la disposición del código y documentación de los proyectos.

Cuando realizamos un proyecto software es bastante habitual que vayamos haciendo pruebas, modificando nuestros fuentes continuamente, añadiendo funcionalidades, etc. Muchas veces, antes de abordar un cambio importante que requiera tocar mucho código, nos puede interesar guardarnos una versión de los fuentes que tenemos en ese momento, de forma que guardamos una versión que sabemos que funciona y abordamos, por separado, los cambios.

Si no usamos ningún tipo de herramienta que nos ayude a hacer esto, lo más utilizado es directamente hacer una copia de los fuentes en un directorio separado. Luego empezamos a tocar. Pero esta no es la mejor forma. Hay herramientas, los sistemas de control de versiones, que nos ayudan a guardar las distintas versiones de los fuentes.

Con un sistema de control de versiones hay un directorio, controlado por esta herramienta, donde se van guardando los fuentes de nuestro proyecto con todas sus versiones. Usando esta herramienta, nosotros sacamos una copia de los fuentes en un directorio de trabajo, ahí hacemos todos los cambios que queramos y, cuando funcionen, le decimos al sistema de control de versiones que nos guarde la nueva versión. El sistema de control de versiones suele pedirnos que metamos un comentario cada vez que queremos guardar fuentes nuevos o modificados.

También, con esta herramienta, podemos obtener fácilmente cualquiera de las versiones de nuestros fuentes, ver los comentarios que pusimos en su momento e, incluso, comparar distintas versiones de un mismo fuente para ver qué líneas hemos modificado.

Aunque los sistemas de control de versiones se hacen imprescindibles en proyectos de cierta envergadura y con varios desarrolladores, de forma que puedan mantener un sitio común con las versiones de los fuentes a través de un sistema de control de versiones, también puede ser útil para un único desarrollador en su casa, de forma que siempre tendrá todas las versiones de su programa controladas.

Los sistemas de control de versiones son programas que permiten gestionar un repositorio de archivos y sus distintas versiones; utilizan una arquitectura cliente-servidor en donde el servidor guarda la(s) versión(es) actual(es) del proyecto y su historia. Sirven para mantener distintas versiones de un fichero, normalmente código fuente, documentación o ficheros de configuración.

"Hay que unirse, no para estar juntos, sino para hacer algo juntos. "

Juan Donoso Cortés (1808-1853); Ensayista español.

4.1.- Conceptos básicos de sistemas de control de versiones.

Existen una serie de conceptos necesarios para comprender el funcionamiento de los sistemas de control de versiones, entre los cuales destacamos los siguientes:

- ✓ **Revisión:** Es una visión estática en el tiempo del estado de un grupo de archivos y directorios. Posee una etiqueta que la identifica. Suele tener asociado metadatos ("datos sobre datos". o "informaciones sobre datos") como pueden ser:
 - ➔ Identidad de quién hizo las modificaciones.
 - ➔ Fecha y hora en la cual se almacenaron los cambios.
 - ➔ Razón para los cambios.
 - ➔ De qué revisión y/o rama se deriva la revisión.
 - ➔ Palabras o términos clave asociados a la revisión.
- ✓ **Copia de trabajo:** También llamado "*Árbol de trabajo*", es el conjunto de directorios y archivos controlados por el sistema de control de versiones, y que se encuentran en edición activa. Está asociado a una rama de trabajo concreta.
- ✓ **Rama de trabajo(o desarrollo):** En el más sencillo de los casos, una rama es un conjunto ordenado de revisiones. La revisión más reciente se denomina principal (main) o cabeza. Las ramas se pueden separar y juntar según como sea necesario, formando un grafo de revisión.
- ✓ **Repositorio:** Lugar en donde se almacenan las revisiones. Físicamente puede ser un archivo, colección de archivos, base de datos, etc.; y puede estar almacenado en local o en remoto (servidor).
- ✓ **Conflicto:** Ocurre cuando varias personas han hecho cambios contradictorios en un mismo documento (o grupo de documentos); los sistemas de control de versiones solamente alertan de la existencia del conflicto. El proceso de solucionar un conflicto se denomina **resolución**.
- ✓ **Cambio:** Modificación en un archivo bajo control de revisiones. Cuando se unen los cambios en un archivo (o varios), generando una revisión unificada, se dice que se ha hecho una **combinación** o integración.
- ✓ **Parche:** Lista de cambios generada al comparar revisiones, y que puede usarse para reproducir automáticamente las modificaciones hechas en el código.

Con el empleo de los sistemas de control de versiones se consigue mantener un repositorio con la información actualizada. La forma habitual de trabajar consiste en mantener una copia en local y modificarla. Después actualizarla en el repositorio. Como ventaja tenemos que no es necesario el acceso continuo al repositorio.

Algunos sistemas de control de versiones permiten trabajar directamente contra el repositorio; en este caso tenemos como ventaja un aumento de la transparencia, a pesar de que como desventaja existe el bloqueo de ficheros.



Supongamos que estamos 3 programadores aportando código al mismo proyecto dentro de la empresa **BK programación**; para la integración del código se emplea un sistema de control de versiones; ¿en qué caso se pueden producir conflictos?, ¿cómo soluciona el sistema de control de versiones el conflicto?

El conflicto se va a producir cuando más de un programador intente integrar cambios en el mismo código. La herramienta de control de versiones no soluciona el conflicto, sólo informa de su existencia.

4.2.- Procedimiento de uso habitual de un sistema de control de versiones.

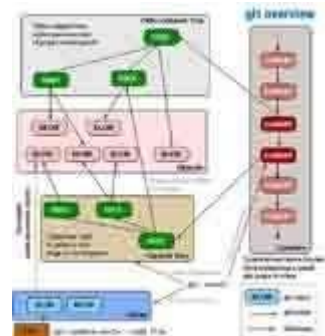
El funcionamiento general de un sistema de control de versiones sigue el siguiente ciclo de operaciones:

- ✓ Descarga de ficheros inicial (Checkout):
 - ➔ El primer paso es bajarse los ficheros del repositorio.
 - ➔ El "checkout" sólo se hace la primera vez que se usan esos ficheros.
- ✓ Ciclo de trabajo habitual:

- Modificación de los ficheros, para aplicar los cambios oportunos como resultado de la aportación de cada una de las personas encargadas de manipular el código de los ficheros.
- Actualización de ficheros en local (Update): Los ficheros son modificados en local y, posteriormente, se sincronizan con los ficheros existentes en el repositorio.
- Resolución de conflictos (si los hay): Como resultado de la operación anterior es cuando el sistema de control de versiones detectará si existen conflictos, en cuyo caso informa de ello y siendo los usuarios implicados en la manipulación del código afectado por el conflicto, los encargados de solucionarlo.
- Actualización de ficheros en repositorio (Commit): Consiste en la modificación de los ficheros en el repositorio; el sistema de control de versiones comprueba que las versiones que se suben estén actualizadas.

Como conclusión, vemos que los sistemas de control de versiones permiten las siguientes funciones:

- ✓ Varios clientes pueden sacar copias del proyecto al mismo tiempo.
- ✓ Realizar cambios a los ficheros manteniendo un histórico de los cambios:
 - Deshacer los cambios hechos en un momento dado.
 - Recuperar versiones pasadas.
 - Ver históricos de cambios y comentarios.
- ✓ Los clientes pueden también comparar diferentes versiones de archivos.
- ✓ Unir cambios realizados por diferentes usuarios sobre los mismos ficheros.
- ✓ Sacar una "foto" histórica del proyecto tal como se encontraba en un momento determinado.
- ✓ Actualizar una copia local con la última versión que se encuentra en el servidor. Esto elimina la necesidad de repetir las descargas del proyecto completo.
- ✓ Mantener distintas ramas de un proyecto.



Debido a la amplitud de operaciones que el sistema de control de versiones permite aplicar sobre los proyectos que administra, muchos de los sistemas de control de versiones ofrecen establecer algún método de autorización, es decir, la posibilidad por la cual a ciertas personas se les permite, o no, realizar cambios en áreas específicas del repositorio.

Algunos proyectos utilizan un sistema basado en el honor: cuando a una persona se le permite la posibilidad de realizar cambios, aunque sea a una pequeña área del repositorio, lo que reciben es una contraseña que le permite realizar cambios en cualquier otro sitio del repositorio y sólo se les pide que mantenga sus cambios en su área. Hay que recordar que no existe ningún peligro aquí; de todas formas, en un proyecto activo, todos los cambios son revisados. Si alguien hace un cambio donde no debía, alguien más se dará cuenta y dirá algo. Es muy sencillo, si un cambio debe ser rectificado todo está bajo el control de versiones de todas formas, así que sólo hay que volver atrás.

Entre las funciones que permite realizar un sistema de control de versiones cabe destacar:

- ✓ Sacar copias del proyecto al mismo tiempo.
- ✓ Realizar cambios a los ficheros manteniendo un histórico de los cambios.
- ✓ Los clientes pueden también comparar diferentes versiones de archivos.
- ✓ Unir cambios realizados por diferentes usuarios sobre los mismos ficheros.
- ✓ Sacar una "foto" histórica del proyecto tal como se encontraba en un momento determinado.
- ✓ Actualizar una copia local con la última versión que se encuentra en el servidor.
- ✓ Mantener distintas ramas de un proyecto.

4.3.- Sistemas de control de versiones centralizados y distribuidos.

El método de control de versiones usado por mucha gente es copiar los archivos a otro directorio controlando la fecha y hora en que lo hicieron. Este enfoque es muy común porque es muy simple, pero también tremendamente propenso a errores. Es fácil olvidar en qué directorio nos encontramos, y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no queríamos. Para hacer frente a este problema, los programadores desarrollaron hace tiempo **sistemas de control de versiones locales** que contenían una simple base de datos en la que se llevaba registro de todos los cambios realizados sobre los archivos.

Una de las herramientas de control de versiones más popular fue un sistema llamado rcs. Esta herramienta funciona básicamente guardando conjuntos de parches (es decir, las diferencias entre archivos) de una versión a otra en un formato especial en disco; puede entonces recrear cómo era un archivo en cualquier momento sumando los distintos parches.



El siguiente gran problema que se encuentra la gente es que necesitan colaborar con desarrolladores en otros sistemas. Para solventar este problema, se desarrollaron los **sistemas de control de versiones centralizados** (Centralized Version Control Systems o CVCSs en inglés). Estos sistemas, como CVS, Subversion y Perforce, tienen un único servidor que contiene todos los archivos versionados, y varios clientes que descargan los archivos de ese lugar central. Durante muchos años, éste ha sido el estándar para el control de versiones.

Esta configuración ofrece muchas ventajas, especialmente frente a VCSs locales. Por ejemplo, todo el mundo sabe hasta cierto punto en qué está trabajando el resto de gente en el proyecto. Los administradores tienen control detallado de qué puede hacer cada uno; y es mucho más fácil administrar un CVCS que tener que lidiar con bases de datos locales en cada cliente.

Sin embargo, esta configuración también tiene serias desventajas. La más obvia es el punto único de fallo que representa el servidor centralizado. Si ese servidor se cae durante una hora, entonces durante esa hora nadie puede colaborar o guardar cambios versionados de aquello en que están trabajando. Si el disco duro en el que se encuentra la base de datos central se corrompe, y no se han llevado copias de seguridad adecuadamente, se pierde absolutamente todo, toda la historia del proyecto salvo aquellas instantáneas que la gente pueda tener en sus máquinas locales.

Es aquí donde entran los **sistemas de control de versiones distribuidos** (Distributed Version Control Systems o DVCSs en inglés). En un DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes no sólo descargan la última instantánea de los archivos sino que replican completamente el repositorio. Así, si un servidor muere, y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios de los clientes puede copiarse en el servidor para restaurarlo. Cada vez que se descarga una instantánea, en realidad se hace una copia de seguridad completa de todos los datos.

Es más, muchos de estos sistemas se las arreglan bastante bien teniendo varios repositorios con los que trabajar, por lo que se puede colaborar con distintos grupos de gente de maneras distintas simultáneamente dentro del mismo proyecto. Esto permite establecer varios tipos de flujos de trabajo que no son posibles en sistemas centralizados, como pueden ser los modelos jerárquicos.

4.4.- Git como sistema de control de versiones.

El núcleo de Linux es un proyecto de software de código abierto con un alcance bastante grande. Durante la mayor parte del mantenimiento del núcleo de Linux (1991-2002), los cambios en el software se pasaron en forma de parches y archivos. En 2002, el proyecto del núcleo de Linux empezó a usar un DVCS propietario llamado BitKeeper.

En 2005, la relación entre la comunidad que desarrollaba el núcleo de Linux y la compañía que desarrollaba BitKeeper se vino abajo, y la herramienta dejó de ser ofrecida gratuitamente. Esto impulsó a la comunidad de desarrollo de Linux (y en particular a Linus Torvalds, el creador de Linux) a desarrollar su propia herramienta basada en algunas de las lecciones que aprendieron durante el uso de BitKeeper. Algunos de los objetivos del nuevo sistema fueron los siguientes:

- ✓ Velocidad.
- ✓ Diseño sencillo.
- ✓ Fuerte apoyo al desarrollo no lineal (miles de ramas paralelas).
- ✓ Completamente distribuido.
- ✓ Capaz de manejar grandes proyectos como el núcleo de Linux de manera eficiente (velocidad y tamaño de los datos).

Git es un sistema rápido de control de versiones, está escrito en C y se ha hecho popular sobre todo a raíz de ser el elegido para el kernel de Linux.

Desde su nacimiento en 2005, **Git** ha evolucionado y madurado para ser fácil de usar y, aún así, conservar estas cualidades iniciales. Es tremendamente rápido, muy eficiente con grandes proyectos, y tiene un increíble sistema de ramificación (branching) para desarrollo no lineal.

La principal diferencia entre **Git** y cualquier otro VCS (Subversion y compañía incluidos) es cómo Git modela sus datos. Conceptualmente, la mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) modelan la información que almacenan como un conjunto de archivos y las modificaciones hechas sobre cada uno de ellos a lo largo del tiempo.

Git no modela ni almacena sus datos de este modo, modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente hace una "foto" del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Para ser eficiente, si los archivos no se han modificado, Git no almacena el archivo de nuevo, sólo un enlace al archivo anterior idéntico que ya tiene almacenado.



Casi cualquier operación es local, la mayoría de las operaciones en **Git** sólo necesitan archivos y recursos locales para operar; por ejemplo, para navegar por la historia del proyecto, no se necesita salir al servidor para obtener la historia y mostrarla, simplemente se lee directamente de la base de datos local. Esto significa que se ve la historia del proyecto casi al instante. Si es necesario ver los cambios introducidos entre la versión actual de un archivo y ese archivo hace un mes, **Git** puede buscar el archivo hace un mes y hacer un cálculo de diferencias localmente, en lugar de tener que pedirle a un servidor remoto que lo haga, u obtener una versión antigua del archivo del servidor remoto y hacerlo de manera local.

Aparte de todo lo anterior, **Git** posee integridad debido a que todo es verificado mediante una suma de comprobación antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo detecte. Como consecuencia de ello es imposible perder información durante su transmisión o sufrir corrupción de archivos sin que **Git** sea capaz de detectarlo.

En la siguiente tabla se resume el concepto de las herramientas de control de versiones, entre ellas GIT.

Sistema de Control de Versiones	
¿QUÉ SON?	<p>Se llama control de versiones a la <u>gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo</u></p> <p>En informática los Sistemas de Control de Versiones se encargan de controlar las distintas versiones del código fuente sobre un proyecto de desarrollo de software determinado</p>
TIPOS DE SISTEMAS DE CONTROL DE VERSIONES	<ul style="list-style-type: none"> ✓ Centralizados: <ul style="list-style-type: none"> ➔ Único repositorio ➔ Almacena todo el código del proyecto ➔ Lo administra un único usuario o grupo de ellos ➔ Ejemplos: CVS, Subversion ✓ Distribuidos: <ul style="list-style-type: none"> ➔ Cada usuario tiene su repositorio ➔ Los distintos repositorios pueden intercambiar y mezclar revisiones ➔ Ejemplos: Git, Mercurial
CONCEPTOS RELACIONADOS	<ul style="list-style-type: none"> ✓ Repositorio: Lugar en el que se almacenan los datos actualizados e históricos del proyecto ✓ Revisión: Es una visión estática en el tiempo del estado de un grupo de archivos y directorios ✓ Rama (branch): Un módulo o proyecto puede ser bifurcado en un determinado momento (ramificado), y a partir de ahí las dos copias del proyecto o módulo pueden sufrir distintas transformaciones de modo independiente ✓ Integración, unión o merge: Une 2 conjuntos de cambios sobre uno o varios ficheros en una revisión unificada
ORIGEN Y DESARROLLO DE GIT	<ul style="list-style-type: none"> ✓ Durante el desarrollo del <u>núcleo de Linux se empezó utilizando</u> un Sistema de Control de Versiones Distribuido llamado <u>Bitkeeper</u>. ✓ En <u>2005 Bitkeeper dejó de ser gratuito</u> y Linus Torvalds, junto con su equipo, decidieron desarrollar su propia herramienta de control de versiones, de donde <u>surge GIT</u> ✓ <u>Git es un sistema rápido de control de versiones, escrito en C y se ha hecho popular sobre todo a raíz de ser elegido para el kernel de Linux.</u> ✓ <u>Git es rápido, muy eficiente</u> con grandes proyectos, y tiene un <u>increíble sistema de ramificación</u>. Posee <u>integridad</u> debido a que todo es verificado mediante una suma de comprobación antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma.
IMPORTANCIA DE GIT	<p>La autoridad y poder que ha adquirido Git se observa simplemente con ver diversos proyectos que lo utilizan, entre ellos:</p> <ul style="list-style-type: none"> ✓ Android, Debian, Fedora, Eclipse, CakePHP, GNOME, OpenSUSE, PostgreSQL, Ruby on Rails, Samba, VLS <p>A parte de lo anterior GitHub:</p> <ul style="list-style-type: none"> ✓ Es un servicio de hospedaje web para proyectos que utilizan el sistema de control de versiones Git. GitHub ofrece tanto planes comerciales como planes gratuitos para proyectos de código abierto

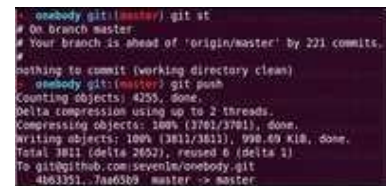
4.5.- Funcionamiento de Git.

Git tiene tres estados principales en los que se pueden encontrar los archivos: confirmado (`committed`), modificado (`modified`) y preparado (`staged`).

- ✓ **Confirmado** significa que los datos están almacenados de manera segura en la base de datos local.
- ✓ **Modificado** estado en el que se ha modificado el archivo pero todavía no se ha confirmado a la base de datos.
- ✓ **Preparado** significa que se ha marcado un archivo modificado en su versión actual para que vaya en la próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de **Git**:

- ✓ El **directorio de Git** (`Git directory`): Almacena los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando se clona un repositorio desde otro ordenador.
- ✓ El **directorio de trabajo** (`working directory`): Es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que se pueda usar o modificar.
- ✓ El **área de preparación** (`staging area`): es un sencillo archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en la próxima confirmación. A veces se denomina **índice**, pero se está convirtiendo en estándar el referirse a ello como el **área de preparación**.



```
oeebody@git:~$ git st
# On branch master
# Your branch is ahead of 'origin/master' by 221 commits.
#
nothing to commit (working directory clean)
oeebody@git:~$ git push
Counting objects: 4255, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3701/3701), done.
Writing objects: 100% (3812/3812), 948.09 KiB, done.
Total: 3812 (delta 2452), reused 0 (delta 0)
To git@github.com:sevenm/oeebody.git
4b63351..7aa03b9 master -> master
```

El flujo de trabajo básico en Git consiste en:

1. Modificar una serie de archivos en el directorio de trabajo.
2. Preparar los archivos, añadiendo instantáneas de ellos al área de preparación.
3. Confirmar los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esa instantánea de manera permanente en el directorio de Git.

Si una versión concreta de un archivo está en el directorio de Git, se considera **confirmada** (`committed`). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está **preparada** (`staged`). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está **modificada** (`modified`).

Observa el siguiente vídeo:

http://www.youtube.com/watch?feature=player_embedded&v=IltAgXMQMnY

Se trata de un impresionante vídeo que forma parte de una serie de vídeos de **Git** y **GitHub** realizados por un excelente profesional que es Jesús Conde, que lo demuestra en la calidad de sus vídeo-tutoriales. Este vídeo demuestra cómo realizar el trabajo básico con **Git**, mediante sus comandos fundamentales que permitirán:

- ✓ Configurar e inicializar un repositorio.
- git init - git clone
- ✓ Empezar y detener el rastreo de archivos.
- ✓ Guardar cambios en `stage` y `commit`.
- ✓ Ignorar ciertos archivos y patrones.
- ✓ Deshacer errores.
- ✓ Navegar por el historial del proyecto.
- ✓ Hacer Push y Pull en repositorios remotos.

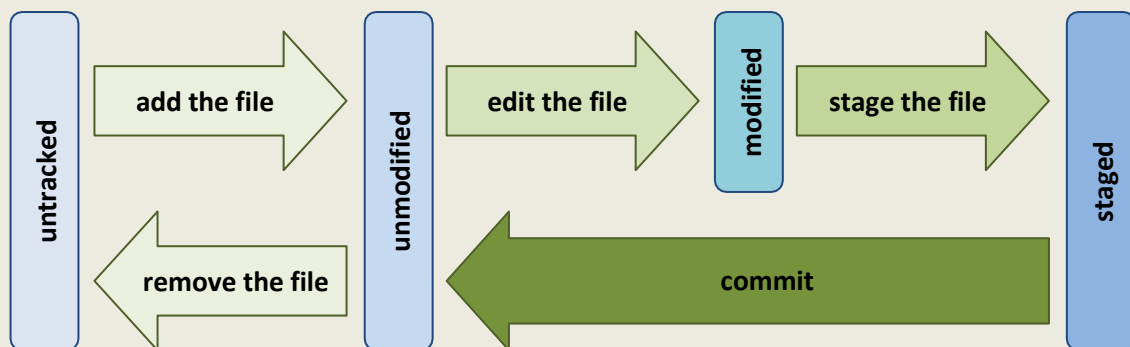
Empieza mostrando como obtener un repositorio Git, empleando 2 métodos:

1. Proyecto creado en un directorio existente e importarlo dentro de Git; comando `git init`
 - a. Crear carpeta para proyecto.
 - b. Dentro del directorio creado ejecutar `git init` automáticamente se crea un directorio `.git` con una serie de archivos y carpetas encargadas de llevar a cabo el control de versiones del proyecto.
 - c. Para añadir archivos al proyecto se emplea el comando `git add` seguido del archivo a añadir y luego ejecutamos el comando `git commit`. Tener en cuenta lo que ocurre si no se ha hecho el `git init` cómo se demuestra en el vídeo.
2. Clonar un repositorio existente desde un servidor; comando `git clone` con lo que se obtiene una copia exacta del proyecto del servidor.
 - a. La sintaxis es: `git clone "protocolo://url del proyecto a clonar"` ; tener en cuenta la carpeta en la que estamos situados.

Los archivos en estado tracked (**rastreado**) en este caso los archivos pueden estar en estado:

- ✓ **Unmodified** en este caso los archivos se encuentran tal cual como están en el repositorio.
- ✓ **Modified** en este caso hemos realizado alguna modificación de los archivos que hemos bajado del repositorio.
- ✓ **Staged** cuando hemos realizado el `git add` sobre el archivo para que se añada al repositorio en el próximo `commit`.

Los archivos en estado **untracked (no rastreado)**, archivos nuevos en la carpeta del proyecto sobre los que no se ha realizado ninguna operación "git" para añadirlos al repositorio. Los estados anteriores se explican siguiendo un ciclo de vida para los archivos, mostrando cómo van cambiando.



El comando `git status` muestra cómo se encuentran los archivos de nuestro proyecto en relación al repositorio del servidor. Si existe la necesidad de excluir archivos del directorio del proyecto para que estos no sean rastreados por GIT, se emplea el archivo `.gitignore` que se demuestra en el vídeo cómo hacerlo empleando los patrones:

<http://www.jedit.org/users-guide/globs.html>

4.6.- Instalación de Git.

Procederemos a instalar Git en una máquina con Debian 6.0.1Squeeze.

Suele ser útil instalar Git desde código fuente, porque obtendremos la versión más reciente. Cada versión de Git tiende a incluir útiles mejoras en la interfaz de usuario, por lo que utilizar la última versión es a menudo el camino más adecuado, realizando para ello la compilación de software desde código fuente.

Para instalar Git, es necesario disponer de las siguientes librerías de las que Git depende: *curl*, *zlib*, *openssl*, *expat* y *libiconv*; para instalarlas podemos ejecutar:

```
#apt-get install libcurl4-gnutls-dev libexpat1-dev gettext libz-dev
```

Una vez instaladas las librerías anteriores procederemos a realizar la descarga de Git desde su página, pudiendo emplear para ello:

```
#wget http://kernel.org/pub/software/scm/git/git-1.7.6.tar.bz2
```

siendo la versión 1.7.6 la más reciente en este momento.

También podríamos dirigirnos a su web y realizar manualmente la descarga:

<http://git-scm.com>

Una vez haya finalizado la descarga procederemos a la compilación e instalación del paquete, podemos seguir los siguientes pasos:

```
#tar -xjvf git-1.7.6.tar.bz2
#cd git-1.7.6
#apt-get build-dep git-core
#apt-get install libssl-dev
#make prefix=/usr/local all doc
#make prefix=/usr/local install install-doc
```

Una vez hecho esto, también es posible obtener Git a través del propio Git para futuras actualizaciones, empleando para ello el siguiente comando de manera que descargaría automáticamente el código fuente desde su repositorio:

```
# git clone git://git.kernel.org/pub/scm/git/git.git
```

Si queremos instalar Git en Linux a través de un instalador binario, en general puedes hacerlo a través de la herramienta básica de gestión de paquetes que trae Debian en nuestro caso, aunque también podríamos realizar la instalación mediante el comando:

```
# apt-get install git-core
```

Sea una forma u otra la que hayamos decidido para realizar la instalación de Git, para comprobar si se ha realizado correctamente comprobamos con el siguiente comando:

```
# git --version
```

que en nuestro caso debería devolver "git version 1.7.6".

4.7.- Configuración de Git (I).

Las opciones de configuración reconocidas por **Git** pueden distribuirse en dos grandes categorías: las del lado cliente y las del lado servidor. La mayoría de las opciones que permiten configurar las preferencias personales de trabajo están en el lado cliente. Aunque hay multitud de ellas, aquí vamos a ver solamente unas pocas, nos centraremos en las más comúnmente utilizadas y en las que afectan significativamente a la forma personal de trabajar. Para consultar una lista completa con todas las opciones contempladas en la versión instalada de Git, se puede emplear el siguiente comando:

```
$ git config --help
```

Git trae una herramienta llamada `git config` que permite obtener y establecer variables de configuración que controlan el aspecto y funcionamiento de **Git**.

Lo primero que se debe hacer cuando se instala **Git** es establecer el nombre de usuario y dirección de correo electrónico. Esto es importante porque las confirmaciones de cambios (commits) en Git usan esta información, y es introducida de manera inmutable en los commits que el usuario va a enviar:

```
$ git config --global user.name "alumno"
$ git config --global user.email alumno@example.com
```



Solamente se necesita hacer esto una vez si se especifica la opción `--global`, ya que Git siempre usará esta información para todo lo que se haga en ese sistema. En el caso de querer sobrescribir esta información con otro nombre o dirección de correo para proyectos específicos, puedes ejecutar el mismo comando sin la opción `--global` cuando estemos en el proyecto concreto.

Una vez que la identidad está configurada, podemos elegir el editor de texto por defecto que se utilizará cuando Git necesite que introduzcamos un mensaje. Si no se indica nada, Git usa el editor por defecto del sistema que, generalmente, es **Vi** o **Vim**. En el caso de querer usar otro editor de texto, como **emacs**, podemos hacer lo siguiente:

```
$ git config --global core.editor emacs
```



Otra opción útil que puede ser interesante configurar es la herramienta de diferencias por defecto, usada para resolver conflictos de unión (merge). Supongamos que quisiéramos usar **vimdiff**:

```
$ git config --global merge.tool vimdiff
```

Git acepta *kdifff3*, *tkdiff*, *meld*, *xxdiff*, *emerge*, *vimdiff*, *gvimdiff*, *ecmerge* y *opendiff* como herramientas válidas. En cualquier momento podremos comprobar la configuración que tenemos mediante el comando:

```
$ git config --list
```

Cuando necesitemos ayuda utilizando Git tenemos tres modos de conseguir ver su página del manual (manpage) para cualquier comando:

```
$ git help <comando>
$ git <comando> --help
$ man git-<comando>
```

4.7.1.- Configuración de Git (II).

Debido a la importancia que actualmente poseen las interfaces web, pasaremos a instalar y configurar el entorno web de Git, éste integra un aspecto más intuitivo y cómodo para el usuario.

Partimos de que en nuestra máquina Debian ya está instalado el servidor web **Apache**, de manera que pasamos a instalar el entorno web de Git mediante el comando:

```
# apt-get install gitweb
```



A continuación vamos a crear los siguientes directorios para estructurar nuestro modo de trabajo con Git:

```
# mkdir /home/usuario/git
# mkdir /home/usuario/www_git
```

Lo siguiente que debemos realizar es editar el archivo de configuración de `gitweb` en el directorio de configuración de Apache:

```
# nano /etc/apache2/conf.d/gitweb
```

Debemos escribir las siguientes líneas en este archivo y comentar las existentes:

```
Alias /git /home/usuario/www_git
<Directory /home/usuario/www_git >
  Allow from all
  AllowOverride all
  Order allow,deny
  Options +ExecCGI
  DirectoryIndex gitweb.cgi
```

```
<files gitweb.cgi >
  SetHandler cgi-script
</files>
</directory>
SetEnv GITWEB_CONFIG /etc/gitweb.conf
```

A continuación, mover los archivos básicos del gitweb al directorio de trabajo Apache creado anteriormente:

```
# mv -v /usr/share/gitweb/* /home/usuario/www_git
# mv -v /usr/lib/cgi-bin/gitweb.cgi /home/usuario/www_git
```

Y hacemos los siguientes cambios en el archivo de configuración del **gitweb**:

```
#nano /etc/gitweb.conf
$projectroot = '/home/usuario/git/';
$git_temp = "/tmp";
#$home_link = $my_uri || "/";
$home_text = "indextext.html";
$projects_list = $projectroot;
$stylesheet = "/git/gitweb.css";
$logo = "/git/git-logo.png";
$favicon = "/git/git-favicon.png";
```

Por último, recargamos el apache:

```
# /etc/init.d/apache2 reload
```

4.8.- Trabajando con Git (I).

Lo siguiente que debemos hacer es crear la carpeta del proyecto:

```
# cd /var/cache/git/
# mkdir proyecto.git
# cd proyecto.git
```

Luego, iniciamos un repositorio para nuestro nuevo proyecto y lo configuramos de acuerdo a nuestras necesidades:

```
# git init
```

al ejecutar el comando anterior nos devuelve un mensaje similar a "Initialized empty Git repository in /var/cache/git/proyecto.git/.git/", podemos comprobar que en el directorio `.git` hay una serie de archivos asociados al proyecto que se ha creado automáticamente.

```
# echo "Una breve descripcion del proyecto" > .git/description
# git config -global user.name "Tu nombre"
# git config -global user.email "tu@correo.com"
```

si hemos creado documentos nuevos en nuestro proyecto antes de hacer el "commit", ejecutamos el siguiente comando para que los archivos presentes en la carpeta del proyecto sean añadidos al repositorio:

```
#git add .
```

Una vez incorporados los ficheros nuevos realizamos el "commit":

```
# git commit -a
```

Para marcar un repositorio como exportado se usa el archivo `git-daemon-export-ok`:

```
# cd /var/cache/git/proyecto.git
# touch .git/git-daemon-export-ok
```

Para iniciar el servicio de Git que ejecuta un servidor para hacer público nuestro repositorio, ejecutamos el siguiente comando:

```
# git daemon -base-path=/var/cache/git -detach -syslog -export-all
```



Ahora el repositorio se encuentra corriendo en el puerto 9418 de nuestro computador; podemos comprobarlo ejecutando:

```
#netstat -nautp | grep git
```

Por último, le daremos permisos de escritura a un usuario que no sea root, de tal manera que con dicho usuario se puedan hacer cambios remotos en el repositorio:

```
# adduser usuariogit
# passwd usuariogit
# chown -Rv usuariogit:usuariogit /var/cache/git/proyecto.git
```

Para acceder al repositorio podemos hacerlo de la manera convencional, basta con ejecutar el comando:

```
#git clone git://servidor/proyecto.git proyecto
```

O, también, podemos acceder vía web; mediante la <http://localhost/git/>

El comando `git commit` sólo seguirá la pista de los archivos que estaban presentes la primera vez que se ejecutó `git add`. Si son añadidos nuevos archivos o subdirectorios, debe indicarse al Git mediante el siguiente comando:

```
$ git add ARCHIVOSNUEVOS
```

De manera similar, si queremos que Git se olvide de determinados archivos, porque (por ejemplo) han sido borrados:

```
$ git rm ARCHIVOSVIEJOS
```

Renombrar un archivo es lo mismo que eliminar el nombre anterior y agregar el nuevo. También puedes usar `git mv` que tiene la misma sintaxis que el comando `mv`. Por ejemplo:

```
$ git mv ARCHIVOVIEJO ARCHIVONUEVO
```

4.8.1.- Trabajando con Git (II).

Algunas veces es interesante ir hacia atrás y borrar todos los cambios a partir de cierto punto porque, a lo mejor, estaban todos mal. Entonces, utilizaremos para ello el comando:

```
$ git log
```

Ese comando muestra una lista de commits recientes, y sus hashes SHA1. A continuación, debemos escribir:

```
$ git reset --hard SHA1_HASH
```

con él recuperamos el estado de un commit dado y se borran para siempre cualquier recuerdo de commits más nuevos. Otras veces es necesario saltar a un estado anterior temporalmente. En ese caso hay que escribir:

```
$ git checkout SHA1_HASH
```

este comando nos lleva atrás en el tiempo, sin tocar los commits más nuevos. Y con el comando:

```
$ git checkout master
```

podemos volver al presente. También existe la posibilidad de restaurar sólo archivos o directorios en particular, para ello los agregamos al final del comando:

```
$ git checkout SHA1_HASH algun.archivo otro.archivo
```

Esta forma de **checkout** puede sobrescribir archivos sin avisar. Para prevenir accidentes, es recomendable hacer commit antes de ejecutar cualquier comando de checkout.

Podemos obtener una copia de un proyecto administrado por git escribiendo:

```
$ git clone git://servidor/ruta/a/los/archivos
```

en el caso de ya tener una copia de un proyecto usando **git clone**, podemos actualizar a la última versión con:

```
$ git pull
```

Supongamos que estamos en un grupo de desarrollo y hemos realizado un script que nos gustaría compartir con otros. Para hacer esto con Git, en el directorio donde guardamos el script ejecutamos:

```
$ git init
$ git add .
$ git commit -m "Primer envío"
```



con lo cual, si los demás miembros del grupo de desarrollo ejecutan:

```
$ git clone tu.maquina:/ruta/al/script
```

podrán descargar el script. Esto asume que tienen acceso por **ssh**. Si no es así, ejecutamos **git daemon** y los demás desarrolladores utilizarían para obtener una copia del script:

```
$ git clone git://tu.maquina/ruta/al/script
```

De aquí en adelante, cada vez que modifiquemos el script y creemos que está listo para el lanzamiento, ejecutaremos:

```
$ git commit -a -m "Siguiendo envío"
```

y los demás desarrolladores pueden actualizar su versión yendo al directorio que tiene el script y ejecutando:

```
$ git pull
```

En el caso de que queramos averiguar qué cambios hicimos desde el último commit ejecutaremos:

```
$ git diff
```

Otra de las ventajas de Git reside en la facilidad de crear nuevas ramas de trabajo, llamadas **branch**, donde probar nuevas características y hacer cambios complejos sin que afecte a la rama de trabajo principal. Luego, el proceso de fusión (**merge**), o la vuelta a un estado anterior es igual de fácil.

Github.com es una web donde alojar proyectos utilizando el sistema de control de versiones Git. Esta web ofrece, aparte del almacenaje de proyectos, herramientas para "socializarlos", como pueden ser feeds rss, wikis o gráficos de cómo los desarrolladores trabajan en sus repositorios.

<https://www.github.com/>

Para empezar a usar Git, Github.com pone a disposición de quien los necesite, varios manuales en inglés tanto para configurar Git como crear nuestros primeros repositorios en Github.com

4.9.- Seguridad documentación en Git.

Git se encuadra en la categoría de los sistemas de gestión de código fuente distribuida. Con Git, cada directorio de trabajo local es un repositorio completo no dependiente, de acceso a un servidor o a la red. El acceso a cada repositorio se realiza a través del protocolo de Git, montado sobre ssh, o usando HTTP, aunque no es necesario ningún servidor web para poder publicar el repositorio en la red.



En el flujo de trabajo que hemos dibujado hasta el momento, los desarrolladores no subían directamente cambios al repositorio público, sino que era el responsable del mismo quien aceptaba los cambios y los incorporaba después de revisarlos. Sin embargo, Git también soporta que los desarrolladores puedan subir sus modificaciones directamente a un repositorio centralizado al más puro estilo CVS o Subversion (eliminando el papel de responsable del repositorio público).

Para que un repositorio público pueda ser utilizado de esta forma, es necesario permitir la ejecución del comando `push`. En primer lugar deberemos crear el repositorio público (tal y como hemos visto en secciones anteriores) y habilitar alguno de los siguientes accesos:

- ✓ Acceso directo por sistema de ficheros con permisos de escritura para el usuario sobre el directorio del repositorio.
- ✓ Acceso remoto vía SSH (consultar `man git-shell`) y permisos de escritura para el usuario sobre el directorio del repositorio.
- ✓ Acceso HTTP con WebDav debidamente configurado.
- ✓ Acceso mediante protocolo Git con el servicio "receive-pack" activado en el git daemon.