

BLOQUE 6. CONTROL DE VERSIONES

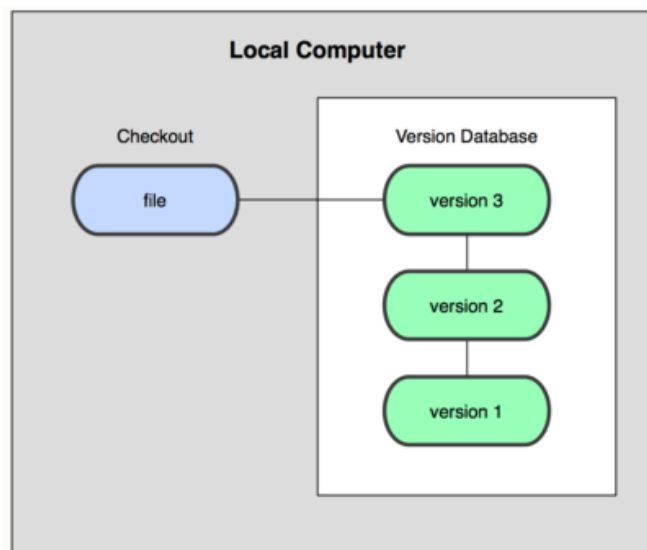
1. Introducción a los sistemas de control de versiones. Definición

¿Qué es un sistema de control de versiones? Un sistema de control de versiones es un registro de los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.

2. Tipos de sistemas de control de versiones y características

Sistemas de control de versiones local

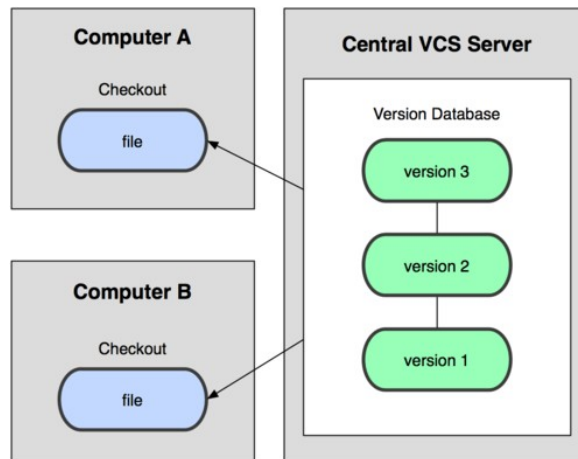
Un método de control de versiones usado por mucha gente es copiar los archivos a otro directorio (quizás indicando la fecha y hora en que lo hicieron, si son avisados). Este enfoque es muy común porque es muy simple, pero también tremendamente propenso a errores.



Sistemas de control de versiones centralizados (cvs, subversion..)

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 desde ese lugar central. Durante muchos años éste ha sido el estándar para el control de versiones



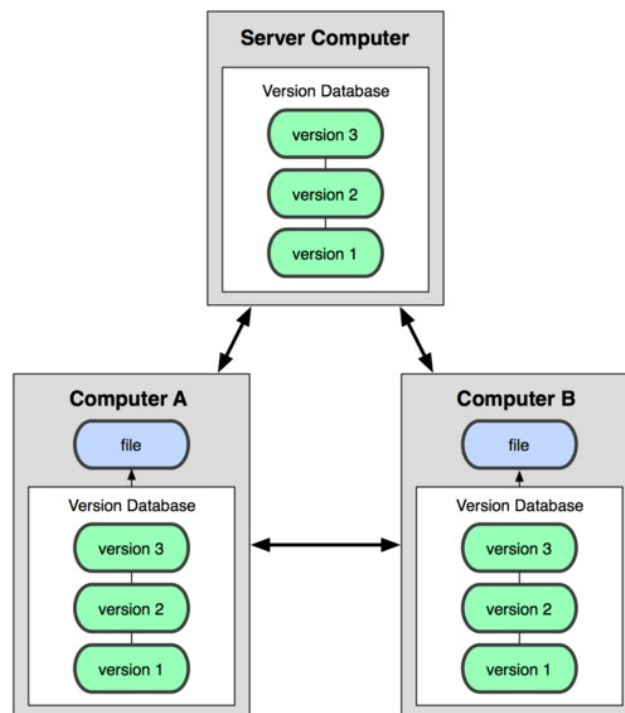
Esta configuración ofrece muchas ventajas, especialmente frente a VCSs locales. Por ejemplo, todo el mundo puede saber (hasta cierto punto) en qué están trabajando los otros colaboradores del proyecto. Los administradores tienen control detallado de qué puede hacer cada uno; y es mucho más fácil administrar 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, pierdes absolutamente todo —toda la historia del proyecto salvo aquellas instantáneas que la gente pueda tener en sus máquinas locales. Los VCSs locales sufren de este mismo problema— cuando tienes toda la historia del proyecto en un único lugar, te arriesgas a perderlo todo.

Sistemas de control de versiones distribuidos(Git, Mercurial)

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: 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 puedes colaborar con distintos grupos de gente simultáneamente dentro del mismo proyecto.

3. Introducción al control de versiones con GIT

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:

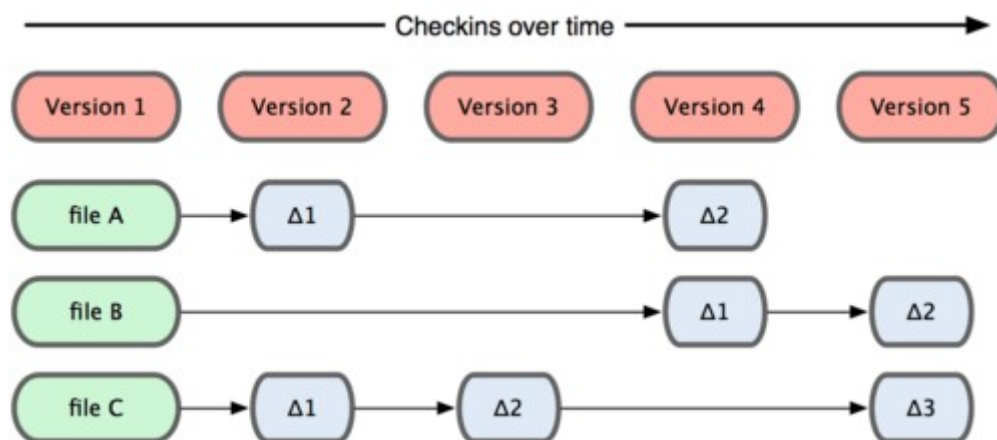
- Velocidad
- Diseño sencillo
- Fuerte apoyo al desarrollo no lineal (miles de ramas paralelas)
- Completamente distribuido
- Capaz de manejar grandes proyectos (como el núcleo de Linux) de manera eficiente (velocidad y tamaño de los datos)

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

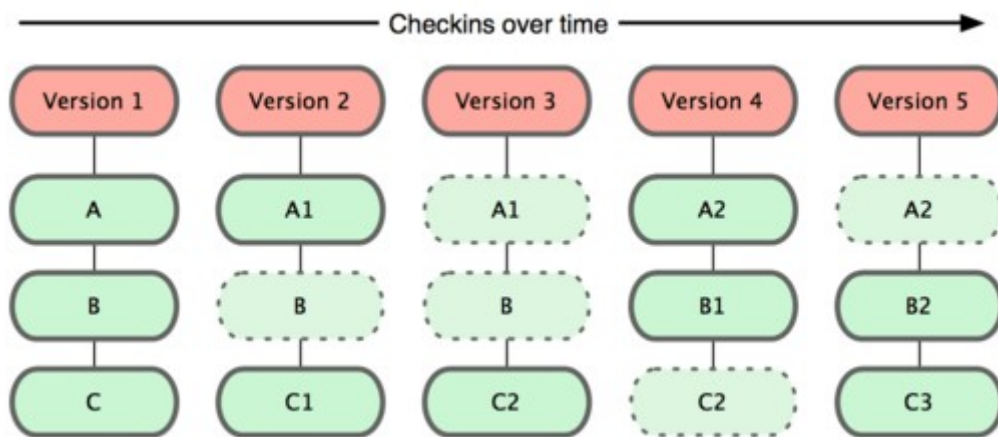
Fundamentos de GIT

1. Instantáneas, no diferencias

Otros sistemas (CVS, Subversion) tienden a almacenar los datos como cambios de cada archivo respecto a una versión base.



Git no modela ni almacena sus datos de este modo. En cambio, Git 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.



2. Casi cualquier operación es local

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para operar. Por lo general no se necesita información de ningún otro ordenador de tu red. Si estás acostumbrado a un CVCS donde la mayoría de las operaciones tienen esa sobrecarga del retardo de la red, este aspecto de Git te va a hacer pensar que los dioses de la velocidad han bendecido Git con poderes sobrenaturales. Como tienes toda la historia del proyecto ahí mismo, en tu disco local, la mayoría de las operaciones parecen prácticamente inmediatas.

Por ejemplo, para navegar por la historia del proyecto, Git no necesita salir al servidor para obtener la historia y mostrártela, simplemente la lee directamente de tu base de datos local. Esto significa que ves la historia del proyecto casi al instante. Si quieres ver los cambios introducidos en un archivo entre la versión actual y la de 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 desde la red y hacerlo de manera local.

Esto también significa que hay muy poco que no puedas hacer si estás desconectado o sin VPN. Si te subes a un avión o a un tren y quieres trabajar un poco, puedes confirmar tus cambios felizmente hasta que consigas una conexión de red para subirlos. Si te vas a casa y no consigues que tu cliente VPN funcione correctamente, puedes seguir trabajando. En muchos otros sistemas, esto es imposible o muy doloroso.

3. Git tiene integridad

Todo en Git es verificado mediante una suma de comprobación (checksum en inglés) 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 sepa. Esta funcionalidad está integrada en Git al más bajo nivel y es parte integral de su filosofía. No puedes perder información durante su transmisión o sufrir corrupción de archivos sin que Git lo detecte.

El mecanismo que usa Git para generar esta suma de comprobación se conoce como hash SHA-1. Se trata de una cadena de 40 caracteres hexadecimales (0-9 y a-f), y se calcula en base a los contenidos del archivo o estructura de directorios. Un hash SHA-1 tiene esta pinta: 24b9da6552252987aa493b52f8696cd6d3b00373

Verás estos valores hash por todos lados en Git, ya que los usa con mucha frecuencia. De hecho, Git guarda todo no por nombre de archivo, sino por el valor hash de sus contenidos.

4. Git solo añade información

Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información. Como en cualquier VCS, puedes perder o estropear cambios que no has confirmado todavía; pero después de confirmar una instantánea en Git, es muy difícil de perder, especialmente si envías (push) tu base de datos a otro repositorio con regularidad.

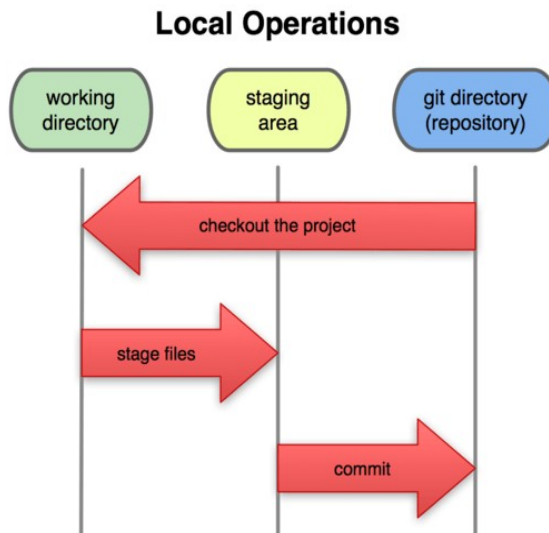
5. Los tres estados

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

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

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

- **el directorio de Git (Git directory)**
- **el directorio de trabajo (working directory)**
- **el área de preparación (staging area)**



- **El directorio de Git (repository)** es donde Git 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 clonas 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 los puedas 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 tu próxima confirmación. A veces se le denomina índice, pero se está convirtiendo en estándar el referirse a ella como el área de preparación.

El flujo de trabajo básico en Git es algo así:

1. Modificas una serie de archivos en tu directorio de trabajo.
2. Preparas los archivos, añadiéndolos a tu área de preparación.
3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esas instantáneas de manera permanente en tu 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).

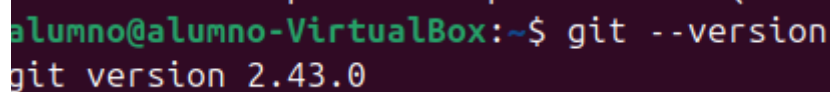
4. Instalación y configuración básica de GIT en Ubuntu en local

```
sudo apt update
```

```
sudo apt install git
```

Una vez instalado puedes comprobar la versión con el comando

```
git --version
```



```
alumno@alumno-VirtualBox:~$ git --version
git version 2.43.0
```

Tu identidad

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

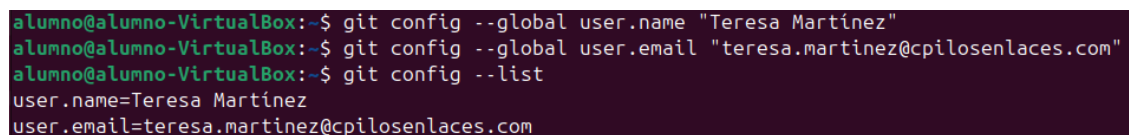
Lo primero que deberías hacer cuando instalas Git es establecer tu nombre de usuario y dirección de correo electrónico. Esto es importante porque las confirmaciones de cambios (commits) en Git usan esta información, y es introducida de manera inmutable en los commits que envías:

```
git config --global user.name "Your Name"
```

```
git config --global user.email "Your Email"
```

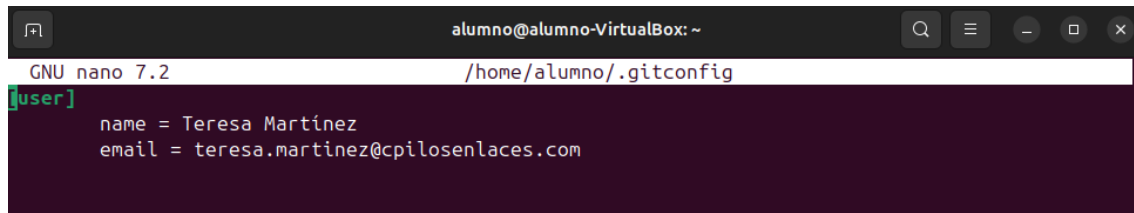
Si quieres comprobar tu configuración y listar todas las propiedades que Git ha configurado puedes usar el comando:

```
git config --list
```



```
alumno@alumno-VirtualBox:~$ git config --global user.name "Teresa Martinez"
alumno@alumno-VirtualBox:~$ git config --global user.email "teresa.martinez@cpilosenlaces.com"
alumno@alumno-VirtualBox:~$ git config --list
user.name=Teresa Martinez
user.email=teresa.martinez@cpilosenlaces.com
```

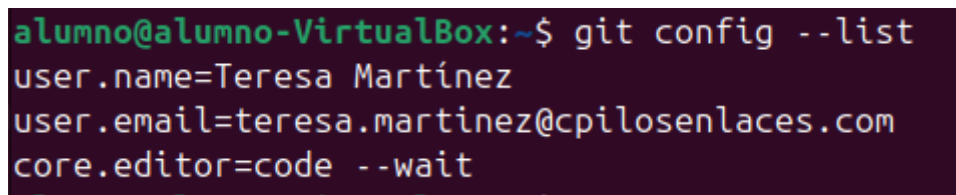

Esta configuración se almacena en el archivo ~/.gitconfig. Este fichero es específico a tu usuario. Puedes hacer que Git lea y escriba específicamente en este archivo pasando la opción --global. En nuestro caso el fichero contendrá...



```
alumno@alumno-VirtualBox: ~  
GNU nano 7.2 /home/alumno/.gitconfig  
[user]  
  name = Teresa Martinez  
  email = teresa.martinez@cpilosenlaces.com
```

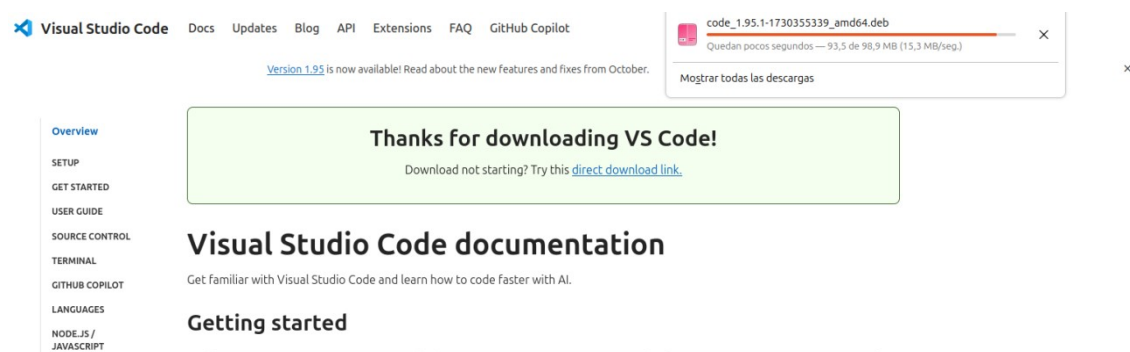
También se puede cambiar la configuración del editor por defecto (vi)

```
git config --global core.editor "nano"  
  
git config --global core.editor "gedit"  
  
git config --global core.editor "code --wait" (Visual studio code)
```



```
alumno@alumno-VirtualBox:~$ git config --list  
user.name=Teresa Martínez  
user.email=teresa.martinez@cpilosenlaces.com  
core.editor=code --wait
```

Instalación VScode...



Comando para ayuda sobre un comando:

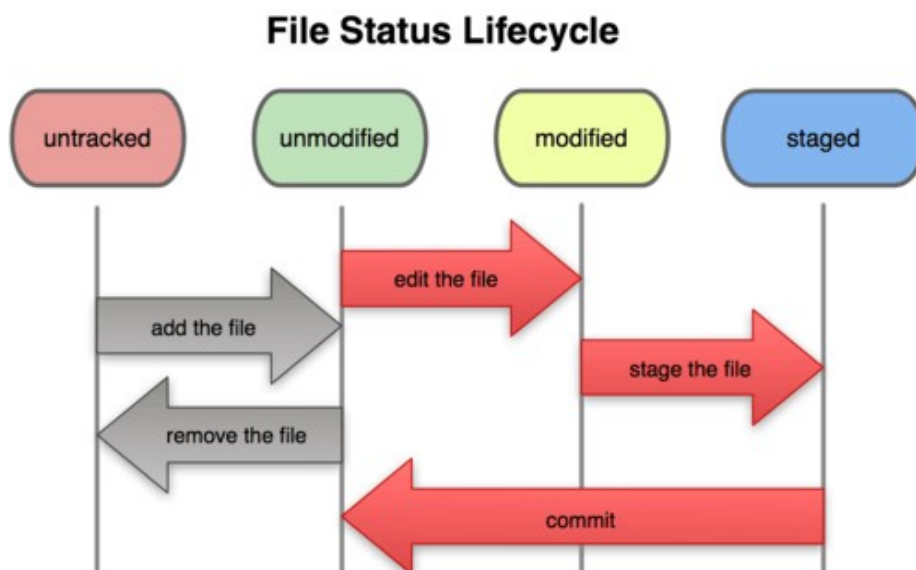
```
git help <comando>
```

5. Trabajando con GIT en local, comandos básicos.

Ciclo de vida

Los archivos en un proyecto Git pueden existir en uno de los siguientes estados:

- **Sin seguimiento (untracked)**: archivos que son nuevos o que no se han agregado al control de versiones de Git.
- **Seguimiento (tracked)**: archivos que se han agregado al control de versiones y se están monitoreando para detectar cambios.
- **Modificado (modified)**: archivos rastreados que se han modificado pero que aún no se han preparado para su confirmación.
- **En preparación (staged)**: archivos modificados que se han seleccionado y preparado para la siguiente confirmación.
- **Confirmado (committed)**: archivos preparados que se han registrado oficialmente en el historial del repositorio después de una confirmación.



Vamos a ver algunas de las acciones más típicas:

1. Inicializar un repositorio GIT

Si estás empezando el seguimiento en Git de un proyecto existente, necesitas ir al directorio del proyecto que deseas empezar a seguir y escribir.

```
git init
```

Esto crea un nuevo subdirectorio llamado `.git` (repositorio de git) dentro del directorio del proyecto que contiene todos los archivos necesarios del repositorio un esqueleto de un repositorio Git.



2. Agregar archivos a Git (sin seguimiento a seguimiento)

Cuando se crea un nuevo archivo en un repositorio Git, comienza su vida como *sin seguimiento*.

Para comenzar a rastrear el archivo, debe agregarlo al índice (staged area) usando el comando `git add`. Una vez agregado, el archivo pasa al estado *staged*, lo que significa que Git ahora lo está monitoreando para futuros cambios.

```
lenovo@LAPTOP-K87IDOM2 MINGW64 ~/Mis documentos/prueba
$ git init
Initialized empty Git repository in C:/Users/lenovo/Documents/prueba/.git/

lenovo@LAPTOP-K87IDOM2 MINGW64 ~/Mis documentos/prueba (main)
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        archivo.txt

nothing added to commit but untracked files present (use "git add" to track)

lenovo@LAPTOP-K87IDOM2 MINGW64 ~/Mis documentos/prueba (main)
$ git add archivo.txt

lenovo@LAPTOP-K87IDOM2 MINGW64 ~/Mis documentos/prueba (main)
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   archivo.txt
```

3. Ver el estado del archivo

Para ver el estado actual de los archivos en su repositorio, puede usar el comando

```
git status
```

Este comando proporciona una descripción general de los archivos sin seguimiento, modificados y preparados, lo que le ayuda a comprender qué se incluirá en la próxima confirmación.

4. Modificación de archivos (seguimiento de modificación)

Si modificas un fichero y ejecutas `git status` verás que aparece bajo la cabecera “Modificados pero no actualizados” (“Changes not staged for commit”) esto significa que un archivo bajo seguimiento ha sido modificado en el directorio de trabajo, pero no ha sido preparado todavía. En este punto, el archivo no está listo para incluirse en una confirmación; es necesario prepararlo y moverlo al estado *staged*

5. Preparando archivos para la confirmación (modificados a preparados)

Para prepararlo, habría que ejecutar el comando git add (es un comando multiuso puedes utilizarlo para empezar el seguimiento de archivos nuevos, para preparar archivos, y para otras cosas como marcar como resueltos archivos con conflictos de unión)

```
git add nombreFichero
```

Para añadir todo:

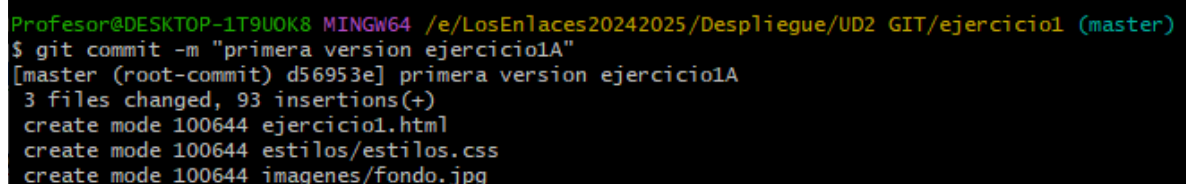
```
git add -all  
git add .
```

6. Confirmación de archivos (de preparado a confirmado)

Una vez que los archivos estén en el estado preparado, puede enviar los cambios al repositorio con el comando git commit. Esto captura una instantánea de los archivos preparados y guarda ese estado en el historial del repositorio como una nueva confirmación. Cada confirmación tiene un identificador único (hash SHA-1) y un mensaje que describe los cambios realizados.

Recuerda que cualquier cosa que todavía esté sin preparar —cualquier archivo que hayas creado o modificado, y sobre el que no hayas ejecutado git add desde su última edición— no se incluirá en esta confirmación. Se mantendrán como modificados en tu disco.

```
git commit  
git commit -m "mensaje"
```



```
Profesor@DESKTOP-1T9UOK8 MINGW64 /e/LosEnlaces20242025/Despliegue/UD2 GIT/ejercicio1 (master)  
$ git commit -m "primera version ejercicio1A"  
[master (root-commit) d56953e] primera version ejercicio1A  
3 files changed, 93 insertions(+)  
create mode 100644 ejercicio1.html  
create mode 100644 estilos/estilos.css  
create mode 100644 imagenes/fondo.jpg
```

Si quieres saltarte el área de preparación, Git proporciona un atajo. Pasar la opción -a al comando git commit hace que Git prepare todo archivo que estuviese en seguimiento antes de la confirmación, permitiéndote obviar toda la parte de git add.

```
git commit -a -m "mensaje"
```

```
Profesor@DESKTOP-1T9UOK8 MINGW64 /e/LosEnlaces20242025/Despliegue/UD2 GIT/ejercicio1 (master)
$ git commit -a -m "segunda version ejercicio1B"
warning: in the working copy of 'ejercicio1.html', LF will be replaced by CRLF the next time Git touches it
[master 53238a9] segunda version ejercicio1B
 2 files changed, 57 insertions(+), 59 deletions(-)
```

7. Saltar archivos (ignorar archivos)

En algunos casos, es posible que desee que Git ignore ciertos archivos o directorios, como archivos de registro, directorios de compilación o archivos de configuración confidenciales. Para hacer esto, puede crear un archivo llamado `.gitignore` en el directorio raíz de su proyecto y enumerar los patrones de archivos que deben ignorarse.

He aquí un archivo `.gitignore` de ejemplo:

```
$ cat .gitignore
*.loa
*~
```

La primera línea le dice a Git que ignore cualquier archivo cuyo nombre termine en `.lo` o `.la` —archivos objeto que suelen ser producto de la compilación de código—. La segunda línea le dice a Git que ignore todos los archivos que terminan en tilde (`~`), usada por muchos editores de texto, como Emacs, para marcar archivos temporales. También puedes incluir directorios de log, temporales, documentación generada automáticamente, etc. Configurar un archivo `.gitignore` antes de empezar a trabajar suele ser una buena idea, para así no confirmar archivos que no quieres en tu repositorio Git.

8. Comparar diferencias

Si el comando `git status` es demasiado impreciso para ti —quieres saber exactamente lo que ha cambiado, no sólo qué archivos fueron modificados— puedes usar el comando `git diff`. Ese comando compara lo que hay en tu directorio de trabajo con lo

que hay en tu área de preparación. El resultado te indica los cambios que has hecho y que todavía no has preparado.

```
git diff
```

Si quieres comparar los cambios preparados con tu última confirmación

```
git diff --staged
```

```
git diff --cached
```

```
Profesor@DESKTOP-1T9UOK8 MINGW64 /e/LosEnlaces20242025/Despliegue/UD2 GIT/ejerci
cio1 ((53238a9...))
```

```
$ git diff
```

```
diff --git a/ejercicio1.html b/ejercicio1.html
```

```
index 71ac207..9ef8bf8 100644
```

```
--- a/ejercicio1.html
```

```
+++ b/ejercicio1.html
```

```
@@ -6,7 +6,8 @@
```

```
</head>
```

```
<link rel="stylesheet" href="estilos/estilos.css">
```

```
<body>
```

```
-         <div>
```

```
+         <div>
```

```
+             <h1> ejemplo</h1> [REDACTED]
```

```
             <h1 class="fondoBlanco">HTML &amp; CSS: Curso práctico a
vanzado</h1>
```

```
             <h2 class="fondoAzul">Datos del libro</h2>
```

```
             <ul id="l1">
```

```
Profesor@DESKTOP-1T9UOK8 MINGW64 /e/LosEnlaces20242025/Despliegue/UD2 GIT/ejerci
cio1 ((53238a9...))
```

```
$ git add ejercicio1.html
```

```
Profesor@DESKTOP-1T9UOK8 MINGW64 /e/LosEnlaces20242025/Despliegue/UD2 GIT/ejerci
cio1 ((53238a9...))
```

```
$ git diff
```

```
Profesor@DESKTOP-1T9UOK8 MINGW64 /e/LosEnlaces20242025/Despliegue/UD2 GIT/ejerci
cio1 ((53238a9...))
```

```
$ git diff --staged
```

```
diff --git a/ejercicio1.html b/ejercicio1.html
```

```
index 71ac207..9ef8bf8 100644
```

```
--- a/ejercicio1.html
```

```
+++ b/ejercicio1.html
```

```
@@ -6,7 +6,8 @@
```

```
</head>
```

```
<link rel="stylesheet" href="estilos/estilos.css">
```

```
<body>
```

```
-         <div>
```

```
+         <div>
```

```
+             <h1> ejemplo</h1> [REDACTED]
```

```
             <h1 class="fondoBlanco">HTML &amp; CSS: Curso práctico a
vanzado</h1>
```

```
             <h2 class="fondoAzul">Datos del libro</h2>
```

```
             <ul id="l1">
```

```
Profesor@DESKTOP-1T9UOK8 MINGW64 /e/LosEnlaces20242025/Despliegue/UD2 GIT/ejerci
cio1 ((53238a9...))
```

```
$ git diff --cached
```

```
diff --git a/ejercicio1.html b/ejercicio1.html
```

```
index 71ac207..9ef8bf8 100644
```

```
--- a/ejercicio1.html
```

```
+++ b/ejercicio1.html
```

```
@@ -6,7 +6,8 @@
```

```
</head>
```

```
<link rel="stylesheet" href="estilos/estilos.css">
```

```
<body>
```

```
-         <div>
```

```
+         <div>
```

```
+             <h1> ejemplo</h1> [REDACTED]
```

```
             <h1 class="fondoBlanco">HTML &amp; CSS: Curso práctico a
vanzado</h1>
```

```
             <h2 class="fondoAzul">Datos del libro</h2>
```

```
             <ul id="l1">
```


9. Eliminar archivos del control de versiones

Si decide que Git ya no debe rastrear un archivo, puede eliminarlo del índice (staged area) con el comando `git rm`. Esto no sólo elimina el archivo del control de versiones, sino que también lo elimina del directorio de trabajo.

Es decir, para eliminar un archivo de Git, debes eliminarlo de tus archivos bajo seguimiento (más concretamente, debes eliminarlo de tu área de preparación), y después confirmar. El comando `git rm` se encarga de eso, y también elimina el archivo de tu directorio de trabajo, para que no lo veas entre los archivos sin seguimiento.

```
git rm nombreFichero
```

Si desea conservar el archivo localmente pero eliminarlo de Git, puede usar la opción `--cached`.

```
git rm --cached nombreFichero
```

Si ya habías modificado el archivo y lo tenías en el área de preparación, y quieres eliminarlo completamente deberás forzar su eliminación con la opción `-f`. Ésta es una medida de seguridad para evitar la eliminación accidental de información que no ha sido registrada en una instantánea, y que por tanto no podría ser recuperada.

```

Profesor@DESKTOP-1T9UOK8 MINGW64 /e/LosEnlaces20242025/Despliegue/UD2 GIT/ejerci
cio1 ((53238a9...))
$ touch fichero.borrar

Profesor@DESKTOP-1T9UOK8 MINGW64 /e/LosEnlaces20242025/Despliegue/UD2 GIT/ejercicio1 ((53238a9...))
$ git add fichero.borrar

Profesor@DESKTOP-1T9UOK8 MINGW64 /e/LosEnlaces20242025/Despliegue/UD2 GIT/ejercicio1 ((53238a9...))
$ git status
HEAD detached from 894e139
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   ejercicio1.html
    new file:   fichero.borrar

Profesor@DESKTOP-1T9UOK8 MINGW64 /e/LosEnlaces20242025/Despliegue/UD2 GIT/ejercicio1 ((53238a9...))
$ git rm fichero.borrar
error: the following file has changes staged in the index:
    fichero.borrar
(use --cached to keep the file, or -f to force removal)

Profesor@DESKTOP-1T9UOK8 MINGW64 /e/LosEnlaces20242025/Despliegue/UD2 GIT/ejercicio1 ((53238a9...))
$ git rm --cached fichero.borrar
rm 'fichero.borrar'

Profesor@DESKTOP-1T9UOK8 MINGW64 /e/LosEnlaces20242025/Despliegue/UD2 GIT/ejercicio1 ((53238a9...))
$ git status
HEAD detached from 894e139
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   ejercicio1.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    fichero.borrar

Profesor@DESKTOP-1T9UOK8 MINGW64 /e/LosEnlaces20242025/Despliegue/UD2 GIT/ejercicio1 ((53238a9...))
$ git add fichero.borrar

Profesor@DESKTOP-1T9UOK8 MINGW64 /e/LosEnlaces20242025/Despliegue/UD2 GIT/ejercicio1 ((53238a9...))
$ git status
HEAD detached from 894e139
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   ejercicio1.html
    new file:   fichero.borrar

Profesor@DESKTOP-1T9UOK8 MINGW64 /e/LosEnlaces20242025/Despliegue/UD2 GIT/ejercicio1 ((53238a9...))
$ git rm -f fichero.borrar
rm 'fichero.borrar'

Profesor@DESKTOP-1T9UOK8 MINGW64 /e/LosEnlaces20242025/Despliegue/UD2 GIT/ejercicio1 ((53238a9...))
$ git status
HEAD detached from 894e139
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   ejercicio1.html

```

10. Ver el histórico de confirmaciones

Después de haber hecho varias confirmaciones, o si has clonado un repositorio que ya tenía un histórico de confirmaciones, probablemente quieras mirar atrás para ver qué modificaciones se han llevado a cabo. La herramienta más básica y potente para hacer esto es el comando

```
git log
```

Por defecto, si no pasas ningún argumento, git log lista las confirmaciones hechas sobre ese repositorio en orden cronológico inverso. Es decir, las confirmaciones más recientes se muestran al principio. Como puedes ver, este comando lista cada confirmación con su suma de comprobación SHA-1, el nombre y dirección de correo del autor, la fecha y el mensaje de confirmación.

```
git log --oneline
```

11. Revertir cambios

Si realiza cambios en un archivo y decide que desea descartarlos, Git ofrece varias herramientas para revertir esos cambios.

Por ejemplo, el comando git checkout se puede usar para restaurar archivos a otro estado confirmado.

```
git checkout SHA1
```

El comando git restore se puede usar para sacar del área de preparación archivos

```
git restore --staged archivo
```

El comando git reset lo usaremos para deshacer commits (borra todos los commits posteriores)

```
git reset HEAD~1 (anterior commit sin borrar cambios en el directorio de trabajo)
```

```
git reset HEAD~2 (dos commits anteriores sin borrar cambios en el directorio de trabajo)
```

```
git reset --hard HEAD~1 (resetea al commit anterior borrando todos los cambios realizados)
```

12. Modificar tu última confirmación

Uno de los casos más comunes en el que quieres deshacer cambios es cuando confirmas demasiado pronto y te olvidas de añadir algún archivo, o te confundes al

introducir el mensaje de confirmación. Si quieres volver a hacer la confirmación, puedes ejecutar un commit con la opción --amend.

Por ejemplo

```
git commit -m 'initial commit'  
git add forgotten_file  
git commit --amend
```

Estos tres comandos acabarán convirtiéndose en una única confirmación, la segunda confirmación reemplazará los resultados de la primera.

6. Práctica1