**¿Qué es Git?**

**Git es un sistema de control de versiones,** distribuido y open source.

Git debe instalarse descargando el instalador adecuado al sistema operativo en uso, desde la página oficial.

Ver más abajo: Teoría

**Git Bash.**

Una vez instalado Git, se puede acceder a **Git Bash Here (GB), la terminal o consola de comandos de Git.**

Al acceder a esta opción, se abre la consola de comandos.

Dentro de nuestra terminal podemos: revisar nuestra estructura de archivos, añadir, eliminar y modificar archivos, y muchas cosas más. La terminal es una herramienta muy poderosa para los desarrolladores.

**Comandos básicos**

"List": Nos permite ver el contenido de la carpeta donde nos encontramos.

$ **ls**

"Change Directory": Nos permite movernos a una nueva carpeta o "directorio"

$ **cd [carpeta]**

Nota: para cambiar de unidad también se usa cd. Por ejemplo cd e:

"Make Directory": Crea una nueva carpeta en el directorio donde nos encontramos actualmente.

$ **mkdir [nombre de la carpeta]**

"Touch": creará un nuevo archivo.

$ **touch [nombre del archivo]**

"Remove": Elimina permanentemente un archivo. ¡CUIDADO! Esto traspasa cualquier "papelera de reciclaje" que exista en el sistema y elimina los archivos PERMANENTEMENTE. Como nota, esto no eliminará carpetas, necesitamos un comando especial para eso que aprenderemos más adelante.

$ **rm [nombre del archivo]**

**Comandos de vinculación con el repositorio local**

**git init**: este comando se usa para crear un nuevo repositorio. El mismo, se utiliza una sola vez: durante la configuración inicial de un repositorio nuevo. Al ejecutar este comando, se creará un nuevo sub-directorio .git en el directorio de trabajo actual.

Voy describiendo mi primer práctica:

-Utilizando el comando creo un repositorio en e:/a2/d6/t/pr/git

**git branch**: este comando permite crear, listar y eliminar **ramas**.

Las ramas (branch) son altamente importantes en el mundo de Git. Usando ramas, varios desarrolladores pueden trabajar en paralelo en el mismo proyecto simultáneamente. Git tiene una rama principal llamada **master**, de la que parten las restantes ramas que sean creadas.

-Crear una nueva rama:

**git branch <nombre-de-la-rama>**

Este comando creará una rama en local. En mi ejemplo creé dos ramas:

**$ git branch rama1**

**$ git branch rama2**

-Enviar (push) la nueva rama al repositorio remoto:

**git push <nombre-remoto> <nombre-rama>**

<nombre-remoto>: es el path del repositorio. Se puede obtener en GitHub, opción Repository, Repository settings.

En mi ejemplo:

**git push https://github.com/soygargola/git.git/ rama1**

**git push https://github.com/soygargola/git.git/ rama1**

-Visualizar ramas

**git branch**

**git branch –list**

En el ejemplo, ambos comandos exhibieron:

\* master

rama1

rama2

-Borrar una rama

**git branch -d <nombre-de-la-rama>**

Lo siguiente retomarlo cuando, más abajo, se comience a vincular Git con GigHub.

Enviadas las ramas al repositorio remoto, en GitHub, Menú: View, Branches List: se muestran las ramas existentes. Al clickear sobre una de ellas aparece ventana emergente con el mensaje que interpreto así:

Cambio de Rama

Usted tiene cambios para esta rama. Qué le gustaría hacer?:

-Dejar los cambios en master. Podrá volver más tarde a esta rama.

-Traer los cambios a la rama.

Cambiar-Cancelar

Nota: no hice pruebas y tampoco investigué como pasar de master a una rama en local, o si esto ocurre automáticamente cuando lo hago en GitHub.

**git checkout**

Es uno de los comandos más utilizados en git. Se usa para:

-Posicionarse en una rama (requisito para poder trabajar dentro de la rama).

-Chequear archivos y commits (lo veremos luego).

Para posicionarse en una rama:

**git checkout <nombre-de-la-rama>**

Hay algunos requisitos para el cambio exitoso entre ramas:

-Los cambios en la rama actual deben confirmarse o preparse (commitearse o stagearse) antes del cambio de rama.

-La rama a la que se desea cambiar debe existir en local.

Hay un atributo que permite crear y cambiarte a la rama creada al mismo tiempo:

**git checkout -b <nombre-de-tu-rama>**

Para volver a la rama principal:

**git checkout master**

En lo que sigue continué trabajando en master.

**git status**: con este comando se consulta el estado de las ramas (incluida master) del repositorio local.

Básicamente, este comando nos dice si:

-La rama actual está actualizada

-Hay algo para confirmar, enviar o recibir (pull).

-Hay archivos en preparación (staged), sin preparación(unstaged) o que no están recibiendo seguimiento (untracked)

-Hay archivos creados, modificados o eliminados

Siguiendo con la práctica, al ejecutar este comando tras haber creado el repositorio, en consola vemos:

**On branch master** (En la rama maestra)

**No commits yet** (Sin commits aún)

**Untracked files:** (Archivos sin seguimiento:)

**(use "git add <file>..." to include in what will be committed)** (use “git add para incluir archivos a coomitear)

**GitHub.docx**

**git.docx**

**~$GitHub.docx**

**~$git.docx**

**nothing added to commit but untracked files present (use "git add" to track)** (No hay nada agregado para confirmar, sin embargo hay archivos sin seguimiento. Use git add para incluir archivos a commitear).

El listado anterior, muestra que todos los archivos de la carpeta están sin seguimiento, es decir, Ningún archivo de la carpeta está preparado (staged) ni confirmado (commit).

**git add:** este comando pasa un archivo de modificado a preparado, es decir, lo pasa del área de trabajo a la la staged área. Este paso previo es imprescindible para poder confirmar (commitear) los archivos.

Añadir un único archivo:

**git add <archivo>**

Añadir todo de una vez:

**git add -A**

Importante: El comando git add **no cambia el repositorio**. Los cambios no se guardan hasta que no utilicemos el comando de confirmación **git commit**.

Con git add, preparé los archivos git.docx y GitHub.docx

Consultando ahora el estado con git status, en pantalla de consola se exhibe:

**On branch master**

**No commits yet**

**Changes to be committed:** (Cambios a ser comitteados)

**(use "git rm --cached <file>..." to unstage)**

**new file: GitHub.docx**

**new file: git.docx**

**Untracked files:**

**(use "git add <file>..." to include in what will be committed)**

**~$GitHub.docx**

**~$git.docx**

Vemos que se sigue informando sin archivos commiteados, dos archivos preparados para commitear, que son los archivos stageados mediante git add y dos archivos sin seguimiento).

**git commit -m [mensaje de referencia]:** este comando toma todos los cambios del área preparación (stagin area) y los guarda en el historial de git. Para futuras referencias, puede añadirse mensajes acerca de los cambios hechos, esto facilitará ir atrás en el historial y encontrar los cambios que se fueron realizando. Una forma de añadir un mensaje, es usar la bandera (flag) "-m"en el comando y, a continuación, escribir el mensaje entre comillas.

Este quizás sea el comando más utilizado de Git. Una vez que se llega a cierto punto en el desarrollo de un proyecto, se desea guardar los cambios (quizás, en coincidencia con la finalización de alguna etapa).

git commit es como establecer un punto de control en el proceso de desarrollo al cual se puede volver más tarde si es necesario. El comando:

**git commit -m "mensaje de confirmación"**

Importante: **git commit guarda los cambios únicamente en local**.

Siguiendo con el ejemplo, hice:

**$ git commit -m 'Archivos git.doxc y GitHub.docx al 23/07/22'**

En consola se exhibe:

[master (root-commit) 2ec0e5a] Archivos git.doxc y GitHub.docx al 23/07/22

2 files changed, 0 insertions(+), 0 deletions(-)

create mode 100644 GitHub.docx

create mode 100644 git.docx

Aparentemente indica que se incorporaron dos nuevos archivos y que no hubo archivos eliminados ni archivos ya existentes que en este commit se actualizaran.

Consultado el estado mediante git status, solo se exhiben los archivos sin seguimiento.

**$ git log**: este comando muestra el histórico de commits. No muestra los archivos, sino las commiteadas realizadas. Ejecutado el comando en nuestro ejemplo, consola exhibe:

**commit 2ec0e5ae104fda2a70bb3e2a5b20cea20b6cb968 (HEAD -> master)**

**Author: soygargola <rebertoni@hotmail.com>**

**Date: Sat Jul 23 15:21:27 2022 -0300**

**Archivos git.doxc y GitHub.docx al 23/07/22**

Donde se informa rama (en este caso master), el hash del commit, autor, fecha y mensaje descriptivo.

Nota: **git log -p nnArchivo**: muestra el historial de commits de nnArchivo. De su pantalla se sale con “**q+Enter**”.

-Continuando con la práctica, hago cambios en git.docx, los guardo. En estas condiciones, ante git status, la consola exhibe:

**On branch master**

**Changes not staged for commit:**

**(use "git add <file>..." to update what will be committed)**

**(use "git restore <file>..." to discard changes in working directory)**

**modified: git.docx**

**Untracked files:**

**(use "git add <file>..." to include in what will be committed)**

**~$GitHub.docx**

**~$git.docx**

**~WRL4034.tmp**

**no changes added to commit (use "git add" and/or "git commit -a")**

Observar:

-Informa el archivo modificado (git.docx) como no stageado y sugiere usar git add para stagearlo.

-Los archivos no modificados (hayan sido commiteados anteriormente , como GuiHub.docx, o no, son informados sin seguimiento).

A continuación stageo y commiteo el archivo git.docx. En estas condiciones, consola responde al comando git log con:

**commit e04f8e00855a29fac5ddd45b4ad7208f837733b4 (HEAD -> master)**

**Author: soygargola <rebertoni@hotmail.com>**

**Date: Sat Jul 23 15:49:25 2022 -0300**

**Archivo git.docx Version 2**

**commit 2ec0e5ae104fda2a70bb3e2a5b20cea20b6cb968**

**Author: soygargola <rebertoni@hotmail.com>**

**Date: Sat Jul 23 15:21:27 2022 -0300**

**Archivos git.doxc y GitHub.docx al 23/07/22**

Donde se observan los dos commit realizados.

**git log –oneline**: La opción oneline se usa para mostrar una commiteada por línea. También muestra los primeros siete caracteres del SHA de confirmación y el mensaje de confirmación. A estos siete caracteres se los suele referir como “código de comprobación” y pueden utilizarse para identificar la commiteada en comandos como git revert.

-Continuando con el ejemplo, hago un cambio no deseado en GitHub.docx, lo stageo y commiteo, con lo que git log exhibe en pantalla:

**commit a4efd127283d9c814c2d34428e1efcc3958816cb (HEAD -> master)**

**Author: soygargola <rebertoni@hotmail.com>**

**Date: Sat Jul 23 16:16:06 2022 -0300**

**Archivo GitHub.docx Version 2**

**commit e04f8e00855a29fac5ddd45b4ad7208f837733b4**

**Author: soygargola <rebertoni@hotmail.com>**

**Date: Sat Jul 23 15:49:25 2022 -0300**

**Archivo git.docx Version 2**

**commit 2ec0e5ae104fda2a70bb3e2a5b20cea20b6cb968**

**Author: soygargola <rebertoni@hotmail.com>**

**Date: Sat Jul 23 15:21:27 2022 -0300**

**Archivos git.doxc y GitHub.docx al 23/07/22**

-**git checkout**: Cuando detecto que el cambio realizado en GitHub.docx no era deseado, debo restaurar el archivo a la versión anteriormente commiteada (la identificada con el mensaje: Archivos git.doxc y GitHub.docx al 23/07/22). Para ello, se utiliza el comando git checkout así:

**$ git checkout 2ec0e5ae104fda2a70bb3e2a5b20cea20b6cb968 e:/a2/d6/t/pr/git/GitHub.docx**

Es, decir, se le adjuntan como parámetros el hash del commit y el path del archivo a restaurar.

El archivo es restaurado a la versión anterior y consola muestra:

**Updated 1 path from cca701f**

Importante: esta restauración NO ELIMINA el commit que contiene la versión no deseada.

**git revert**

A veces, se requiere deshacer los cambios. Hay varias maneras para deshacer cambios en local y/o en remoto (dependiendo de lo que necesitemos). Deben utilizarse cuidadosamente estos comandos para evitar borrados no deseados.

Una manera segura para deshacer nuestras commits es utilizar git revert.

Para deshacer un commit debe adjuntarse al comando el hash de ese commit. En nuestro ejemplo, deseamos deshacer el último commit ya que tiene un cambio no deseado. El comando es:

**$ git revert e04f8e00855a29fac5ddd45b4ad7208f837733b4**

Otra posibilidad es, consultar el historial de commits mediante  git log -- oneline y nos valemos del código de comprobación del commit que se desea deshacer, por ejemplo:

**git revert e04f8e0**

Después de esto, aparecerá una pantalla de la que se sale con shift + q (logré salir con q+Enter).

El comando git revert deshará el commit que le hemos indicado, pero creará un nuevo commit deshaciendo el anterior.

La ventaja de utilizar git revert es que no afecta al commit histórico. Esto significa que se siguen viendo todos los commits en el histórico, incluso los revertidos.

Otra medida de seguridad es que todo sucede en local a no ser que los enviemos al repositorio remoto. Por esto es que git revert es más seguro de usar y es la manera preferida para deshacer los commits.

(Este comando no lo entendí muy bien)

**git merge**

Cuando se haya completado el desarrollo en la rama propia, será necesario fusionar la rama con su rama padre (dev o master). Esto se hace con el comando git merge.

git merge básicamente integra las características de la rama con todos los commits realizados a las ramas dev (o master). Los pasos para la fusión son:

-Posicionarse en la rama padre. Si esta es dev:

**git checkout dev**

-Actualizar la rama dev local:

**git fetch**

Fusionar:

**git merge <nombre-de-la-rama>**

Importante: **Asegurarse que la rama padre tenga la última versión antes de fusionar** otras ramas, caso contrario pueden surgir conflictos.

**Vinculación a GitHub.**

Una vez creado y trabajados algunos comandos en local, incorporé el repositorio a GitHub desde GitHub que tiene una opción destinada a tal fin (Menú Flie, Add local repository)

Nota: intenté subir el repositorio directamente desde Git Bash pero no lo logré. No se si es imposible hacerlo.

Como están las cosas, en estos momentos, tengo el mismo repositorio en local y en GitHub.

**git push**

Cuando el repositorio remoto ya existe, tras confirmar cambios en local, estos cambios deben enviarse al repositorio remoto. Esto se hace mediante git push.

**git push <nombre-remoto> <nombre-de-tu-rama>**

Si la rama no ha sido subida, debe hacerse esto previamente mediante:

**git push --set-upstream <nombre-remoto> <nombre-de-tu-rama>**

o

**git push -u origin <nombre-de-tu-rama>**

Importante: **git push solamente carga los cambios que han sido confirmados.**

En el ejemplo, uilicé el comando:

**git push** [**https://github.com/soygargola/git.git/**](https://github.com/soygargola/git.git/)

La repuesta en consola fue:

Everything up-to-date(Todo actualizado)

(No me queda clara la utilidad, ya que aparentemente cuando quedan vinculado el repositorio local con el de GitHub, los cambios se reflejan automáticamente. Tal vez sea para vincularlos por primera vez. Todo a verificar.)

**git pull**:

El comando git pull se utiliza para **recibir actualizaciones del repositorio remoto**. Este comando es una combinación del git fetch y del git merge lo cual significa que cundo usemos el git pull recogeremos actualizaciones del repositorio remoto (git fetch) e inmediatamente aplicamos estos últimos cambios en local (git merge).

**git pull <nombre-remoto>**

Esta operación puede generar conflictos que tengamos que resolver manualmente.

**git clone <nombre-remoto>:** Este comando permite clonar un repositorio desde GitHub a la PC local. Para ello, ingresando a la consola de git:

-Crear una carpeta donde se alojará el repositorio con mkdir. En mi caso cree /git/prueba

-Posicionado en /prueba tipear

**$ git clone** [**https://github.com/soygargola/git.git/**](https://github.com/soygargola/git.git/)

El comando descarga el repositorio permitiendo tenerlo disponible como una copia local.

En el ejemplo quedó en /prueba el mismo repositorio que en /git. No tiene sentido clonar un repositorio que ya era local y de allí fue subido a GitHub. La clonación, tiene sentido cuando queremos traer a local un repositorio nuevo de GitHub. Puede ser también, cuando el repositorio local fue dañado o borrado accidentalmente.

**Teoría**

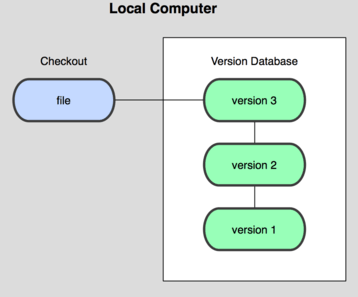
**Version Control System**

¿Qué es un control de versiones, y por qué debería importarte? **Un control de versiones es un sistema que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante**.

Si eres diseñador gráfico o web, y quieres mantener cada versión de una imagen o diseño (algo que sin duda quieres), un sistema de control de versiones (Version Control System o VCS en inglés) es una elección muy sabia. Te permite revertir archivos a un estado anterior, revertir el proyecto entero a un estado anterior, comparar cambios a lo largo del tiempo, ver quién modificó por última vez algo que puede estar causando un problema, quién introdujo un error y cuándo, y mucho más. Usar un VCS también significa generalmente que, si rompes o pierdes archivos, puedes recuperarlos fácilmente.

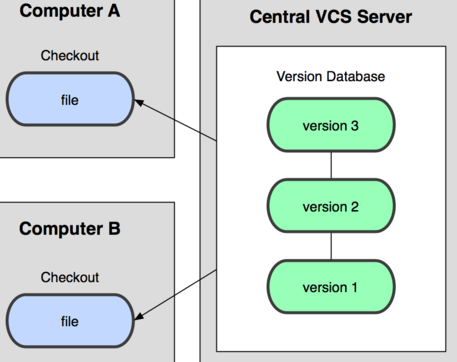
Hay varios tipos de sistemas de versionado, estos pueden ser:

**Locales**



Un método de control de versiones, usado por muchas personas, es copiar los archivos a otro directorio (quizás indicando la fecha y hora en que lo hicieron, si son ingeniosos). Este método es muy común porque es muy sencillo, pero también es tremendamente propenso a errores. Es fácil olvidar en qué directorio te encuentras y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías. Como se pueden imaginar, este sistema funciona bien para trabajar solos, pero si queremos incorporar otra gente al equipo van a empezar a surgir problemas.

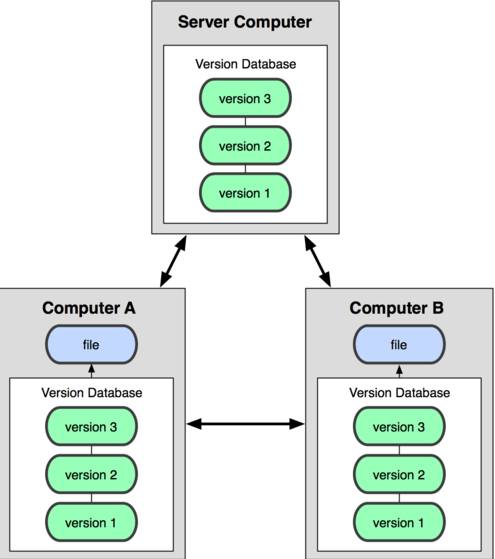
**Centralizados**



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

Este sistema ofrece varias ventajas, por ejemplo: Todo el mundo puede saber en qué están trabajando los demás colaboradores y los administradores tienen control sobre qué archivos pueden ver/modificar cada colaborador. Pero también presenta un problema importante: que hay un punto único de fallo. ¿Si este server se cae? Nadie puede seguir trabajando ni trackeando sus cambios. ¿O si se rompe y no hay backups? Se pierde absolutamente todo el trabajo realizado.

**Distribuido**



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 replican completamente el repositorio en sus PC. 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.

**Historia de Git**

Como muchas de las grandes cosas en esta vida, Git comenzó con un poco de destrucción creativa y encendida polémica. 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:

Velocidad

Diseño sencillo

Fuerte apoyo al desarrollo no lineal (miles de ramas paralelas)

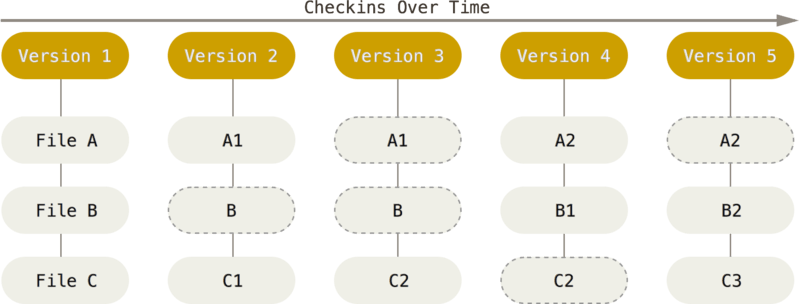
Completamente distribuido

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 a gran escala, y tiene un increíble sistema de ramificación (branching) para desarrollo no lineal.

**Conceptos de Git**

Git modela sus datos como **un conjunto de instantáneas de los archivos de tu proyecto**. **Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él 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**.



**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. Como tienes toda la historia del proyecto ahí mismo, en tu disco local, la mayoría de las operaciones parecen prácticamente inmediatas (con otros sistemas el proceso involucra llamados por red que generan retardos importantes).

Observación mía: de esto entiendo que puedo trabajar con Git en mi PC sin necesidad de GitHub.

**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.

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, con lo cuál, es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa.** Un hash SHA-1 tiene esta pinta:

24b9da6552252987aa493b52f8696cd6d3b00373

Observación: cuando ocurre un conflicto git lo avisa (por ejemplo, dos usuarios trabajando simultáneamente sobre el mismo archivo).

Vamos a distinguir dos directorios:

-El directorio de trabajo, que, a su vez, constituye una copia de una versión del proyecto.

-El directorio de git, donde git almacena los metadatos y la base de datos de tu proyecto. Observación mía: entiendo que es el directorio .git que se genera dentro del directorio de trabajo.

Puedo sacar archivos de la base de datos comprimida en el directorio de Git, y colocarlos en disco para usar o modificar.

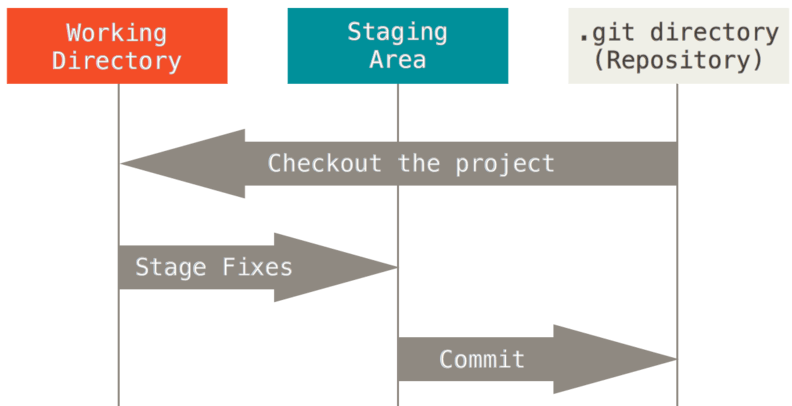
**Estados**

Para Git, los archivos dentro del directorio de trabajo pueden estar en unos de los siguientes estados:

**committed**: significa que los datos están almacenados de manera segura en tu base de datos local.

**modified**: significa que has modificado el archivo, pero todavía no lo has commiteado a tu base de datos.

**staged**: significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima commiteada.



Hay un archivo simple, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación (commiteada).

Al contenido de este archivo o al archivo mismo se lo conoce como **staging area**.

Sabiendo esto, el flujo de trabajo básico en Git sería algo así:

-Modificas una serie de archivos en tu directorio de trabajo.

-Stageas los archivos, añadiendolos a tu staging area o área de preparación.

-Commiteas o 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, y 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 (no se incluyó en el área de preparación), está modificada (modified).

En lo sucesivo, utilizaré indistintamente los siguientes términos como sinónimos:

-Commiteado = confirmado

-Stageado = preparado

-Modificado sin estar preparado ni confirmado: modificado = sin seguimiento.