

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green. These shapes are concentrated on the left and right sides of the image, leaving a large white central area.

GIT

INTRODUCCIÓN

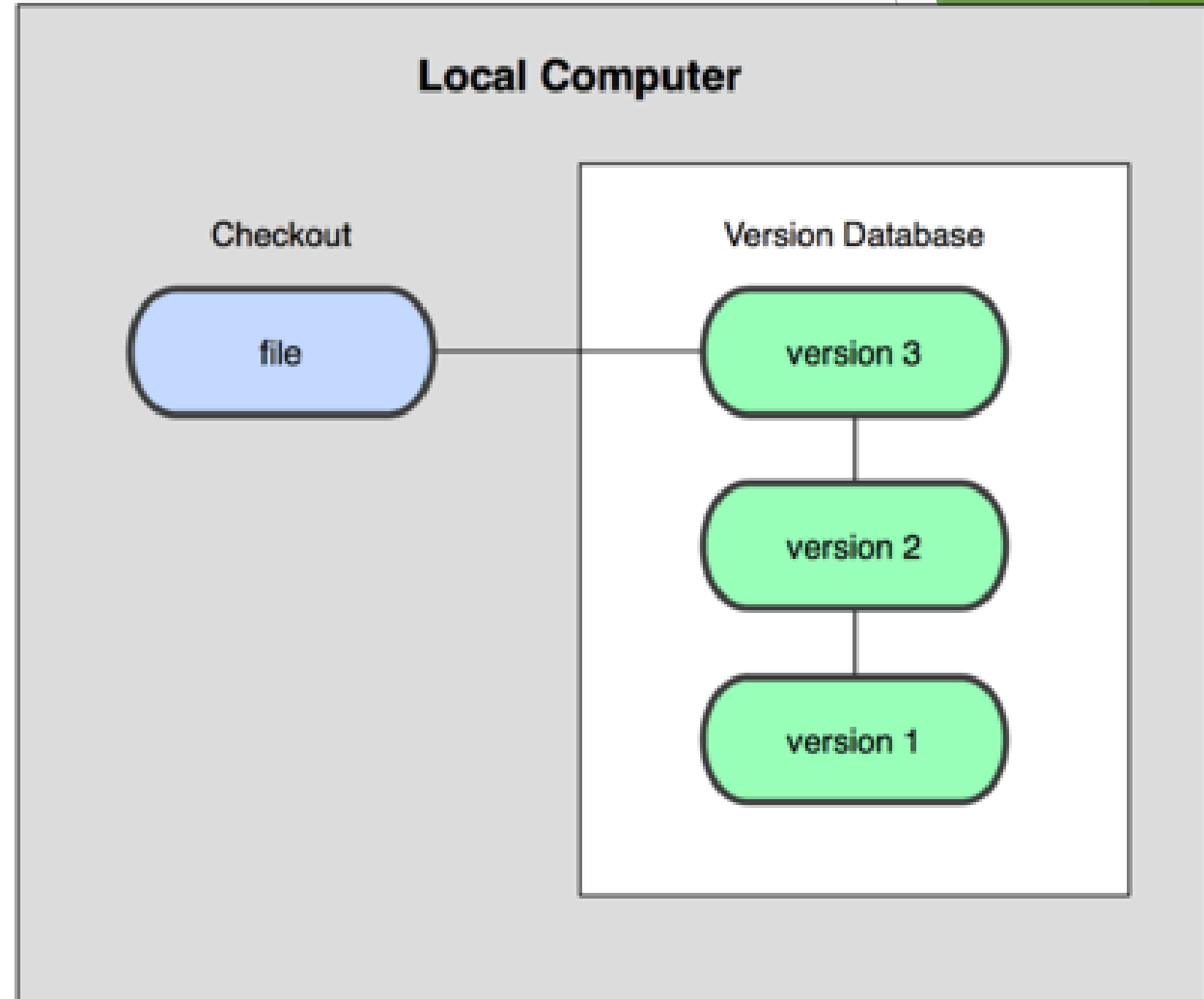
- ▶ Hablaremos de algunos conceptos relativos a las herramientas de control de versiones
- ▶ Veremos cómo tener Git funcionando en tu sistema
- ▶ Aprenderemos el por qué existe Git y por qué usarlo
- ▶ Aprenderemos a configurar GIT para empezar a trabajar con él

Control de versiones

- ▶ Sistema que registra cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, para poder recuperar versiones específicas más adelante.
- ▶ Cualquier tipo de archivo que se encuentre en un ordenador puede ponerse bajo control de versiones.
- ▶ Ejemplo: un desarrollador web quiere mantener cada versión de su página por si lo necesitara en algún momento. Con un sistema de control de versiones (Version Control System o VCS en inglés) cubrimos esta necesidad.
- ▶ Permite acciones como:
 - ▶ Revertir archivos a un estado anterior
 - ▶ Revertir un proyecto a un estado anterior
 - ▶ Comparar cambios a lo largo del tiempo
 - ▶ Ver quién hizo modificaciones por última vez
 - ▶ Ver cómo surgió un problema
 - ▶ Etc.

Sistema de control de versiones local

- ▶ Mucha gente utiliza un método de control de versiones, que es copiar los archivos a otro directorio
- ▶ Enfoque muy común porque es muy simple, pero propenso a errores.
- ▶ Es fácil olvidar en qué directorio te encuentras, y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías.
- ▶ Solución: se desarrolló el sistema de VCSs locales, que contenían una BBDD en la que se llevaba registro de todos los cambios realizados sobre los archivos

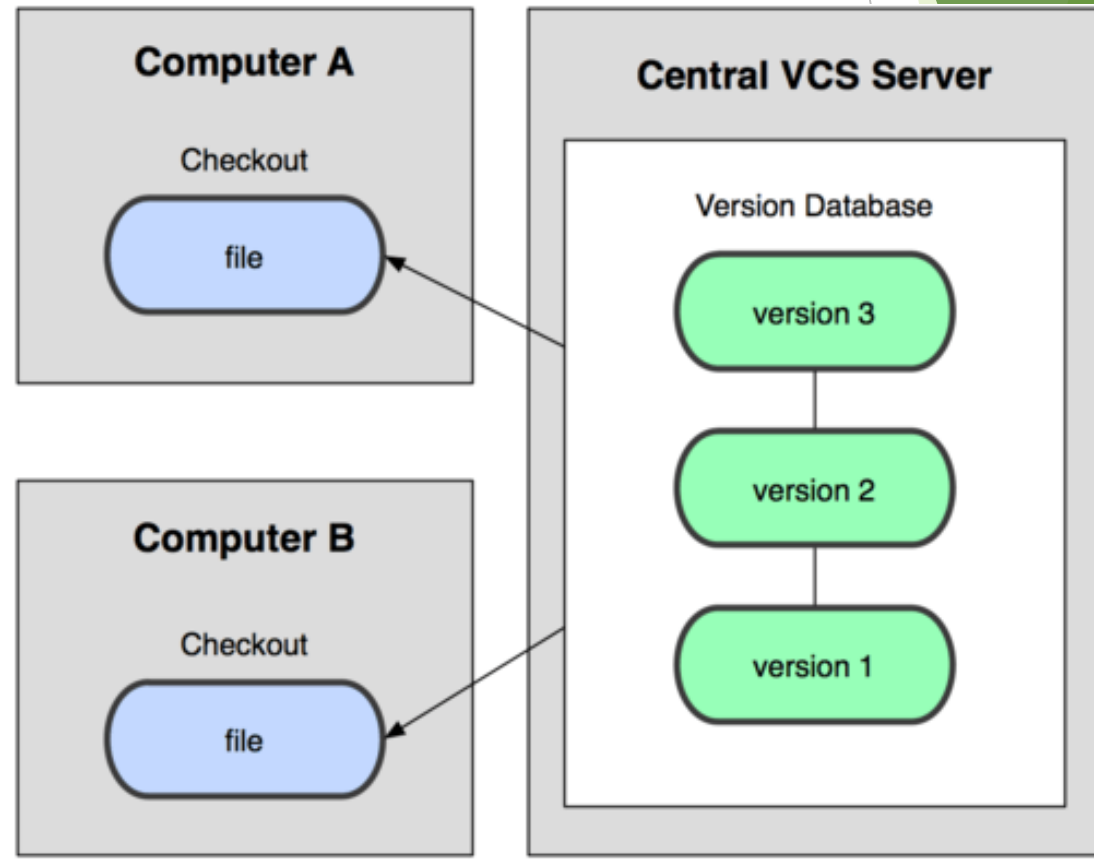


Sistema de control de versiones local

- ▶ El sistema RCS (Revision Control System) es uno de los SCV locales más populares que hubo.
- ▶ Todavía podemos encontrarlo en muchos de los ordenadores actuales.
- ▶ El famoso sistema operativo Mac OS X incluye el comando RCS cuando instalas las herramientas de desarrollo.
- ▶ Esta herramienta funciona guardando diferencias entre archivos(parches) de una versión a otra en un formato especial en disco. Así se puede entonces recrear cómo era un archivo en cualquier momento sumando los distintos parches.

Sistemas de control de versiones centralizados (CVCS)

- ▶ Surgió el problema de la colaboración. Los equipos de trabajo, desarrolladores, etc. necesitan colaborar con desarrolladores en otros sistemas de manera colaborativa.
- ▶ Se desarrollaron sistemas de control de versiones centralizados (Centralized Version Control Systems o CVCS)
- ▶ 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 fue el estándar para el control de versiones

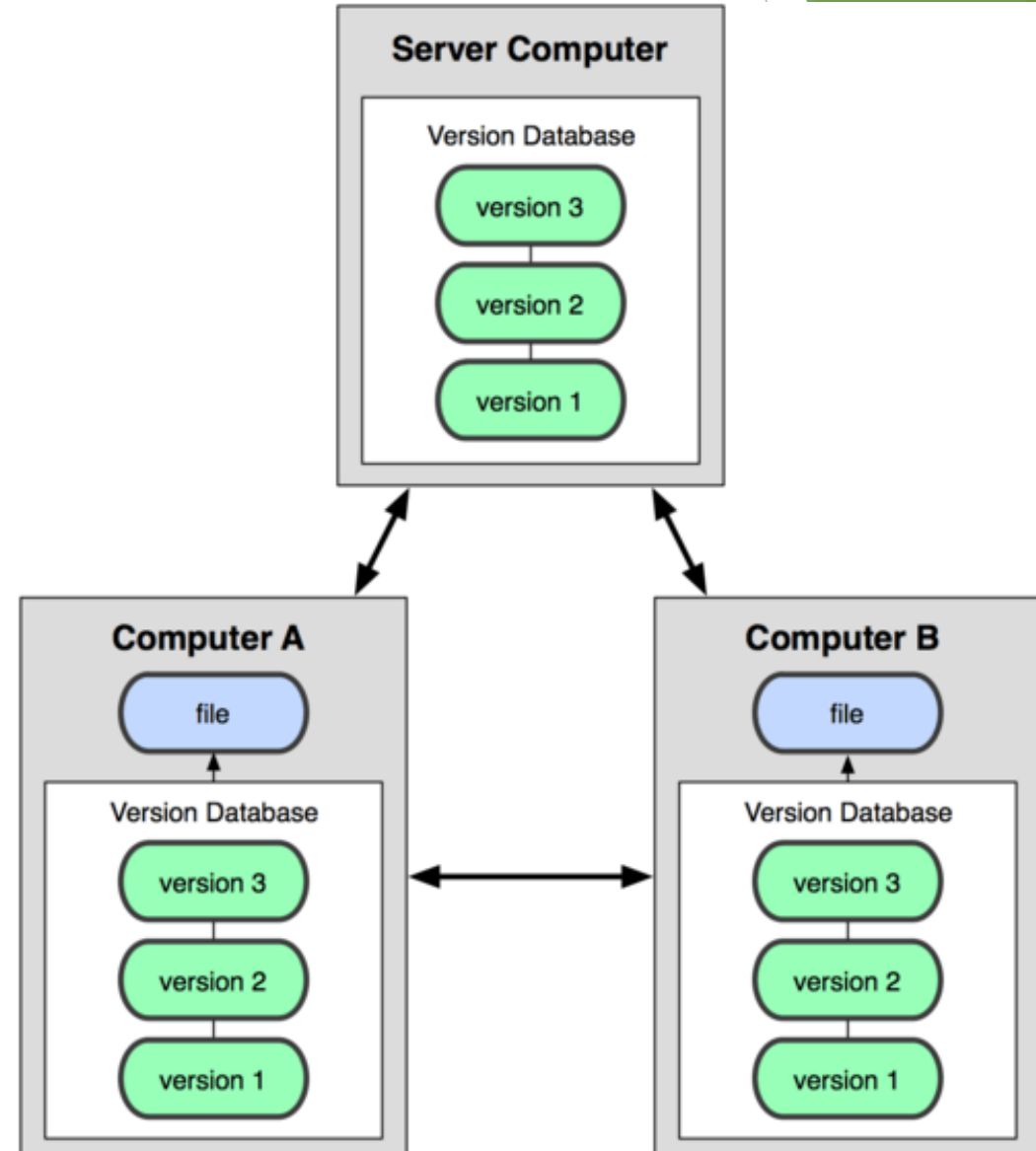


Sistemas de control de versiones centralizados (CVCS)

- ▶ Ofrece muchas ventajas frente a VCSs locales.
- ▶ Ejemplo: todo el mundo puede saber en qué están trabajando los otros colaboradores del proyecto.
- ▶ Los administradores tienen control detallado de qué puede hacer cada uno
- ▶ Es mucho más fácil administrar un CVCS que tener que lidiar con bases de datos locales en cada cliente.
- ▶ Desventajas:
 - ▶ El servidor centralizado es un riesgo. Si ese servidor se cae 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 lo que esté en remoto.
 - ▶ Mismo problema que en VCS locales: Guardar todo en un único lugar es riesgo de perderlo todo.

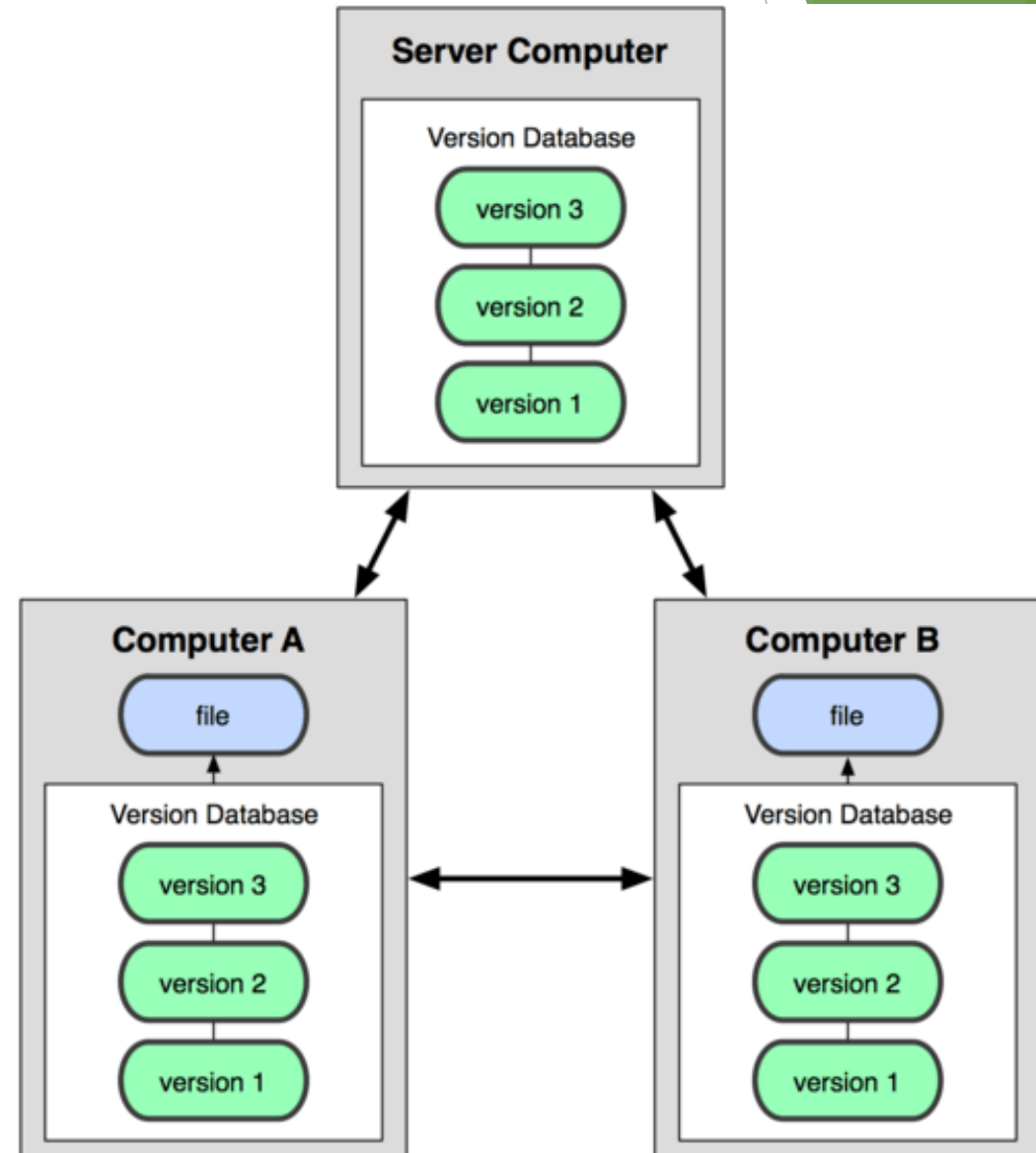
Sistemas de control de versiones distribuidos

- ▶ Distributed Version Control Systems(DVCSs)
- ▶ Ejemplos de sistemas DVCS: Git, Mercurial, Bazaar o Darcs
- ▶ Los clientes no sólo descargan la última instantánea de los archivos. También replican completamente el repositorio.
- ▶ Si un servidor cae, 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.



Sistemas de control de versiones distribuidos

- ▶ Muchos de estos sistemas funcionan muy bien teniendo varios repositorios con los que trabajar
- ▶ Se puede colaborar con distintos grupos de gente simultáneamente dentro del mismo proyecto.
- ▶ Esto permite establecer varios flujos de trabajo que no son posibles en sistemas centralizados, como pueden ser los modelos jerárquicos.



GIT

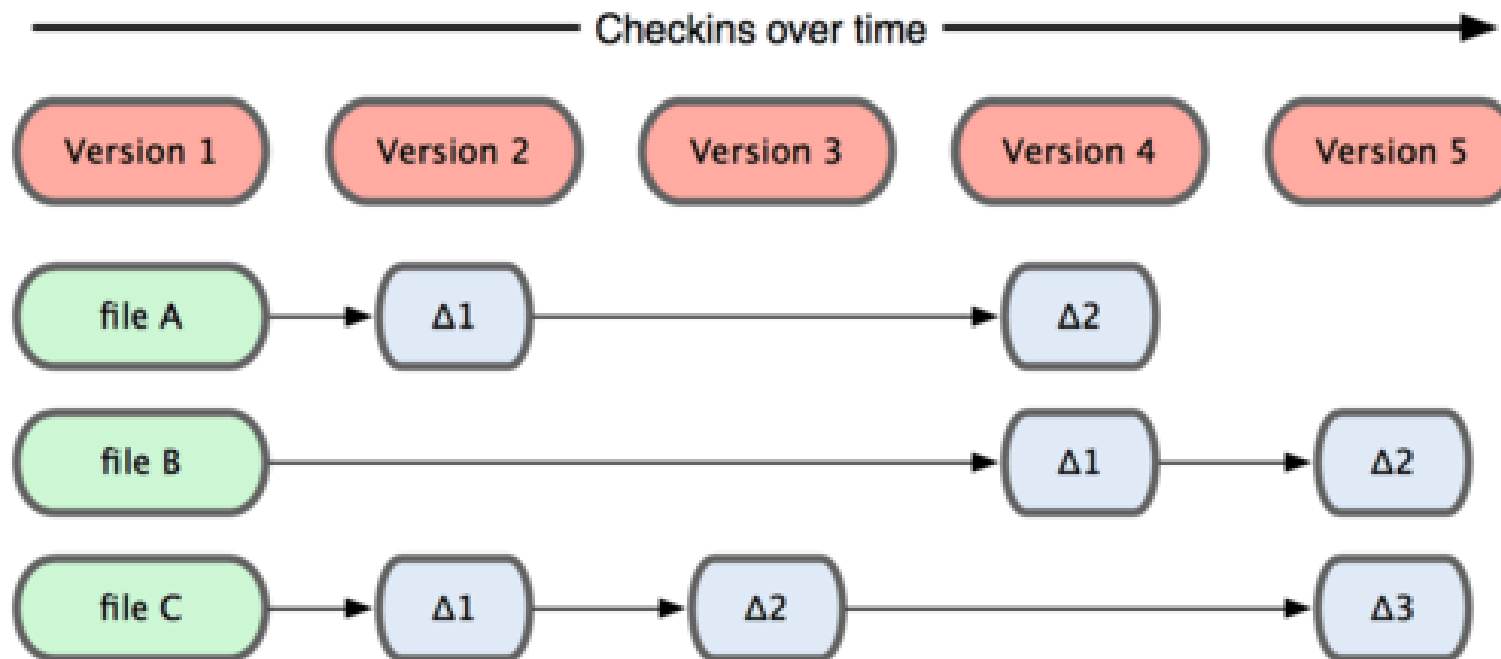
- ▶ Git es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente.
- ▶ Objetivo: llevar registro de los cambios en archivos de computadora y coordinar el trabajo que varias personas realizan sobre archivos compartidos.
- ▶ Historia: 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 desarrolladores del núcleo de Linux y la compañía que desarrollaba BitKeeper se rompió, y la herramienta dejó de ser ofrecida gratuitamente.
- ▶ Esto impulsó a la comunidad de desarrollo de Linux (encabezada por Linus Torvalds, creador de Linux) a desarrollar su propia herramienta basada en algunas de las lecciones que aprendieron durante el uso de BitKeeper.

GIT

- ▶ Objetivos del nuevo sistema GIT:
 - ▶ 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 2005, Git ha evolucionado y madurado para ser fácil de usar y aún conservar estas cualidades iniciales.
- ▶ Muy rápido, eficiente con grandes proyectos, y con un gran sistema de ramificación (branching) para desarrollo no lineal

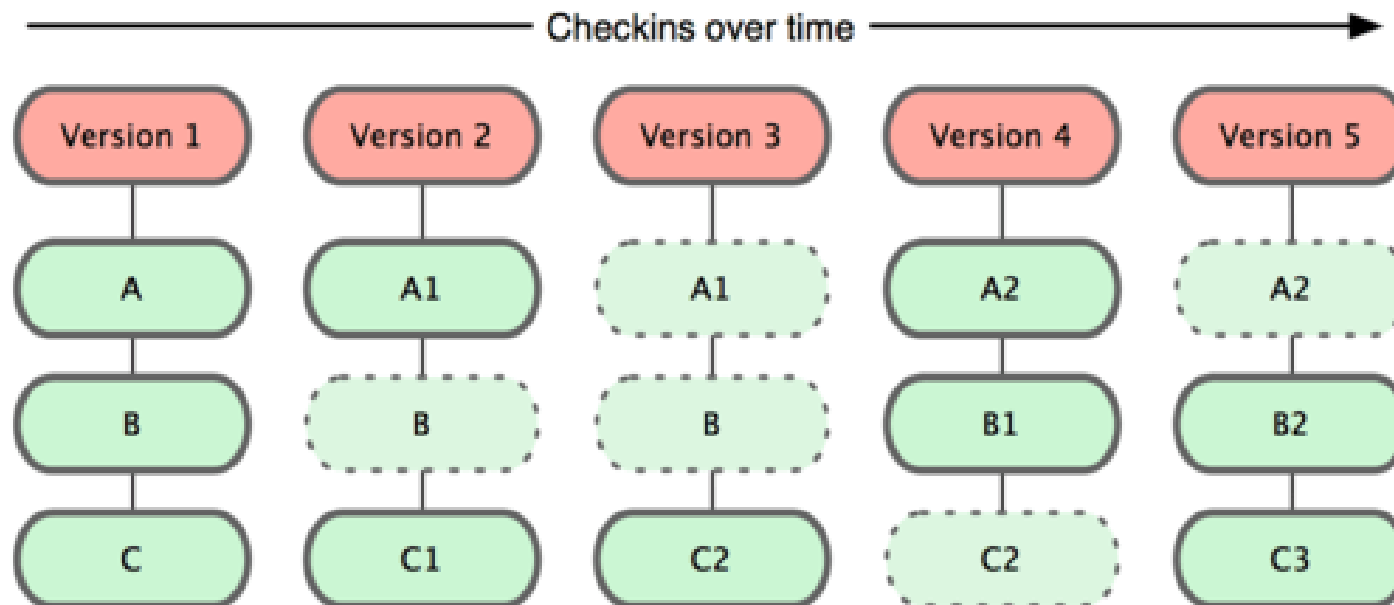
Fundamentos de GIT

- ▶ La principal diferencia entre Git y cualquier otro VCS es cómo Git modela sus datos.
- ▶ 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:



Fundamentos de GIT

- ▶ En cambio, Git modela sus datos 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, se hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea.
- ▶ 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.



Fundamentos de GIT

OPERACIONES “CASI” LOCALES

- ▶ 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.
- ▶ En comparación con otros CVCS con sobrecarga del retardo de la red, Git da sensación de velocidad y dinamismo.
- ▶ La historia de tu proyecto estará en tu disco local. La mayoría de las operaciones parecen prácticamente inmediatas.
- ▶ Ejemplo: para navegar por la historia del proyecto, Git la lee directamente de tu base de datos local. No necesita acceder al servidor para obtener la historia y mostrarla.
- ▶ Se puede ver la historia del proyecto casi al instante de manera local. Para 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 pedir al servidor remoto que lo haga, u obtener una versión antigua desde la red y hacerlo de manera local.

Fundamentos de GIT

OPERACIONES “CASI” LOCALES

- ▶ Esto es una gran ventaja para poder trabajar con o sin conexión.
- ▶ Pocas cosas que no puedas hacer si estás desconectado o sin VPN.
- ▶ Si estás viajando 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.
- ▶ Esto es muy difícil en muchos otros sistemas
 - ▶ En Perforce, por ejemplo, no puedes hacer mucho cuando no estás conectado al servidor
 - ▶ En Subversion y CVS, puedes editar archivos, pero no puedes confirmar los cambios a tu base de datos (porque tu base de datos no tiene conexión).
- ▶ **Git te da ese dinamismo de trabajar con tu repositorio local con o sin conexión**

Fundamentos de GIT

INTEGRIDAD EN GIT

- ▶ Cualquier cambio es verificado mediante una suma de comprobación (checksum) antes de guardarse, y es identificado a partir de ese momento mediante dicha suma.
- ▶ Hace 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 se puede 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. Cadena de 40 caracteres hexadecimales (0-9 y a-f). Se calcula en base a los contenidos del archivo o estructura de directorios.
Ejemplo de Hash:

d921970abaf03b3cf0e71ddcdaab3147ba72cdef

- ▶ Veremos con frecuencia estos valores hash en Git. De hecho, Git guarda por el valor hash de sus contenidos.

Fundamentos de GIT

GENERALMENTE, GIT SÓLO AÑADE INFORMACIÓN

- ▶ Casi cualquier acción en Git sólo añade información a la base de datos de Git.
- ▶ 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.
- ▶ Esto convierte a Git en una gran herramienta. Sabemos que podemos experimentar sin peligro de “cargarnos” nuestro proyecto al 100%. Podremos volver a un estado recuperable.

Fundamentos de GIT

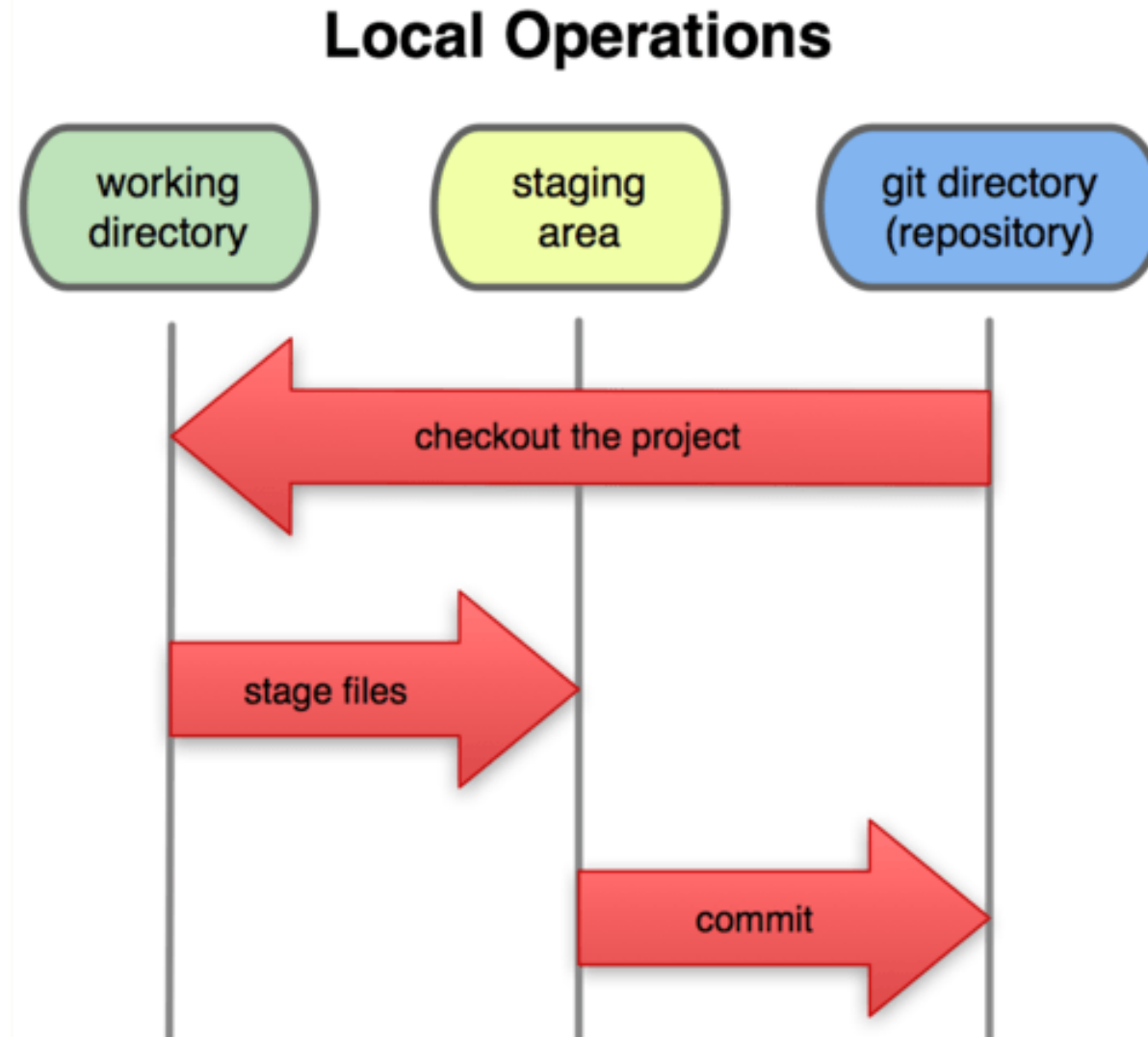
LOS TRES ESTADOS PRINCIPALES:

- ▶ Git tiene tres estados principales para los archivos:
 - ▶ Confirmado (committed). Los datos están almacenados de manera segura en la base de datos local
 - ▶ Modificado (modified). Archivo modificado pero todavía no confirmado a la base de datos
 - ▶ Preparado (staged). Se ha marcado un archivo modificado en su versión actual para que vaya en la próxima confirmación.

Fundamentos de GIT

TRES SECCIONES PRINCIPALES DE UN PROYECTO EN GIT

- ▶ Esto nos lleva a las tres secciones principales de un proyecto de Git:
 - ▶ Directorio de Git (Git directory)
 - ▶ Directorio de trabajo (working directory)
 - ▶ Área de preparación (staging area)



Fundamentos de GIT

TRES SECCIONES PRINCIPALES DE UN PROYECTO EN GIT

- ▶ **Directorio de Git (Git directory)**
 - ▶ 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
- ▶ **Directorio de trabajo (working directory)**
 - ▶ Copia de una versión del proyecto
 - ▶ Los 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
- ▶ **Área de preparación(staging area)**
 - ▶ 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”

Fundamentos de GIT

FLUJO BÁSICO DE GIT

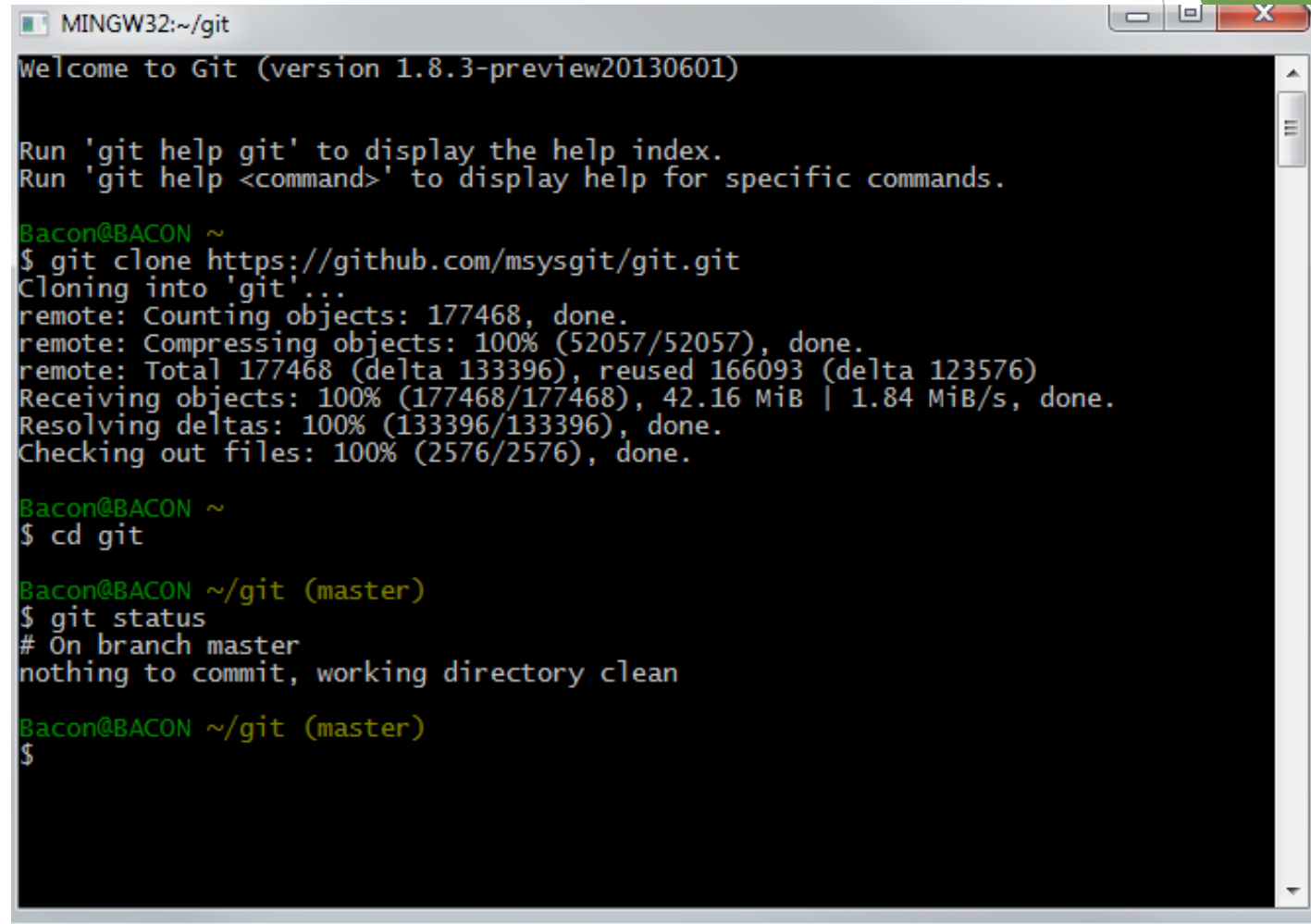
1. Modificar archivos en tu directorio de trabajo
 2. Preparar los archivos, añadiéndolos a tu “área de preparación”
 3. Confirmar los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esas instantáneas de manera permanente en tu directorio de Git
- ▶ Si una versión 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)
 - ▶ Si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (modified)

Instalación de GIT

- ▶ Trabajaremos con la versión GIT para Windows:

<https://gitforwindows.org/>

- ▶ Tras instalarlo, trabajaremos con la “consola que ofrece Git. Teclear “git bash” en el buscador de windows



```
MINGW32:~/git
Welcome to Git (version 1.8.3-preview20130601)

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

Bacon@BACON ~
$ git clone https://github.com/msysgit/git.git
Cloning into 'git'...
remote: Counting objects: 177468, done.
remote: Compressing objects: 100% (52057/52057), done.
remote: Total 177468 (delta 133396), reused 166093 (delta 123576)
Receiving objects: 100% (177468/177468), 42.16 MiB | 1.84 MiB/s, done.
Resolving deltas: 100% (133396/133396), done.
Checking out files: 100% (2576/2576), done.

Bacon@BACON ~
$ cd git

Bacon@BACON ~/git (master)
$ git status
# On branch master
nothing to commit, working directory clean

Bacon@BACON ~/git (master)
$
```

Configuración inicial

ESTABLECER TU IDENTIDAD EN GIT

- ▶ Lo primero que debemos hacer cuando se instala Git es establecer 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 "Pepe Pérez"
```

```
$ git config --global user.email mi_email@dominio.com
```

- ▶ Para comprobar tu configuración, introduce:

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

Configuración inicial

AYUDA EN GIT

- Para recurrir a ayuda usando Git, podemos consultar el manual para cualquier comando usando alguna de las siguientes sentencias en la consola de Git:

```
$ git help <comando>
```

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

- Esto nos redirige al navegador y podemos ver la ayuda en el manual online. Ejemplos:

```
$ git help status
```

```
$ git push--help
```

- Herramienta muy útil. Si nos fijamos en la barra del navegador, vemos que aparece una ruta local al buscar ayuda. Esto quiere decir que con/sin conexión a internet podremos hacer uso de la ayuda cuando lo necesitemos.

Trabajando con repositorios GIT

- ▶ A continuación aprenderemos a trabajar con repositorios GIT:
 - ▶ Cómo configurar e inicializar un repositorio
 - ▶ Comenzar y detener el seguimiento de archivos
 - ▶ Preparar (stage) y confirmar (commit) cambios
- ▶ También veremos otras acciones sobre el repositorio:
 - ▶ Configurar Git para que ignore ciertos archivos y patrones
 - ▶ Deshacer errores rápida y fácilmente
 - ▶ Navegar por la historia de tu proyecto y ver cambios entre confirmaciones
 - ▶ Cómo enviar (push) y recibir (pull) de repositorios remotos

Trabajando con repositorios GIT

INICIALIZAR UN REPOSITORIO GIT EN UN DIRECTORIO EXISTENTE

- Para crear un directorio nuevo donde alojarás tu proyecto, ábrelo y ejecuta lo siguiente para crear un nuevo repositorio de git.

```
$ git init
```

- Esto crea un nuevo subdirectorio llamado `.git` que contiene todos los archivos necesarios del repositorio —un esqueleto de un repositorio Git.

Trabajando con repositorios GIT

CLONANDO UN REPOSITORIO GIT EXISTENTE

- ▶ Se puede copiar un proyecto de un repositorio Git existente.

```
$ git clone <URL>[directory]
```

- ▶ Por ejemplo, si quisiéramos clonar el repositorio remoto de JQuery en nuestro directorio local:

```
$ git clone https://github.com/jquery/jquery
```

- ▶ Te crea un directorio llamado “jquery”, inicializa un directorio .git en su interior, descarga toda la información de ese repositorio, y saca una copia de trabajo de la última versión.
- ▶ Si accedemos al directorio “jquery”, se ven los archivos del proyecto.
- ▶ Se puede clonar el repositorio a un directorio con nombre distinto a “jquery”:

```
$ git clone https://github.com/jquery/jquery mi_directorio_jquery
```
- ▶ Ese comando hace lo mismo que el anterior, pero el directorio de destino se llamará “mi_directorio_jquery”.

Trabajando con repositorios GIT

GUARDANDO CAMBIOS EN EL REPOSITORIO

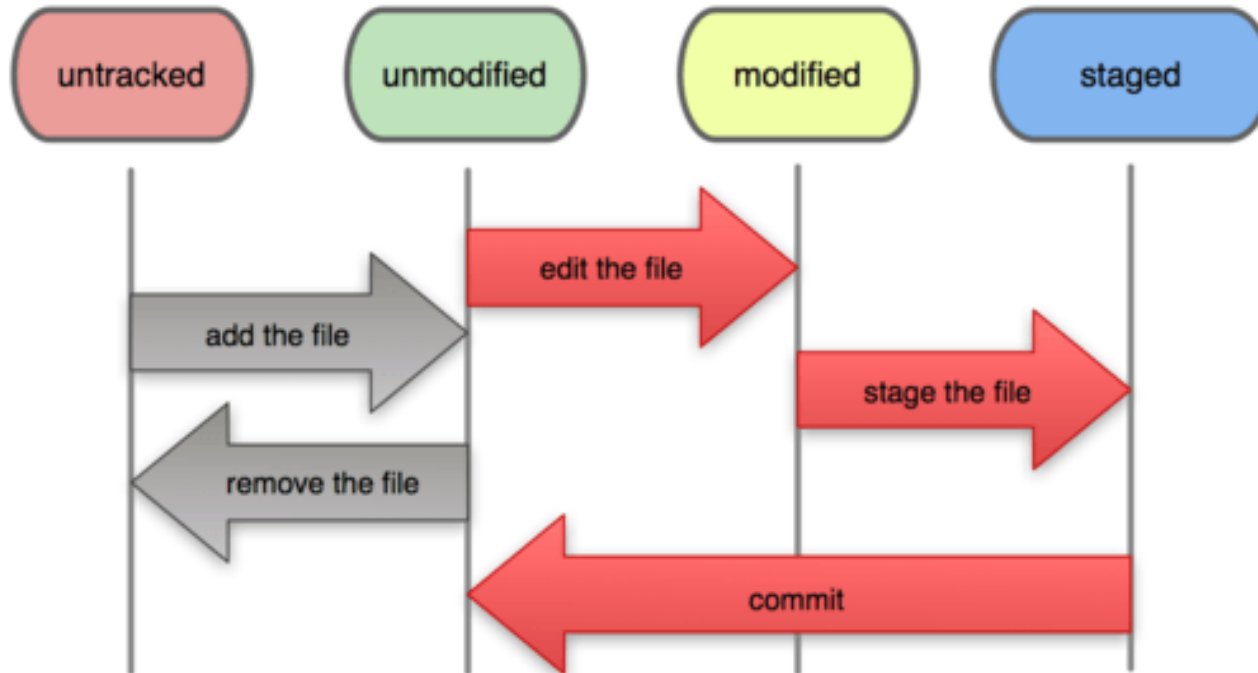
- ▶ Tenemos un repositorio Git completo, y una copia de trabajo de los archivos de ese proyecto.
- ▶ Necesitamos hacer algunos cambios, y confirmar instantáneas de esos cambios en el repositorio cada vez que el proyecto alcance un estado que deseemos guardar.
- ▶ Cada archivo en nuestro directorio de trabajo puede estar en uno de estos dos estados:
 - ▶ Bajo seguimiento (tracked). Aquellos que existían en la última instantánea; pueden estar sin modificaciones, modificados, o preparados.
 - ▶ Sin seguimiento (untracked). Los archivos bajo seguimiento son cualquier archivo de nuestro directorio que no estuviese en tu última instantánea ni está en tu área de preparación
- ▶ La primera vez que clonamos un repositorio, todos los archivos estarán bajo seguimiento y sin modificaciones, ya que los acabamos de copiar y no hemos realizado modificaciones

Trabajando con repositorios GIT

GUARDANDO CAMBIOS EN EL REPOSITORIO

- ▶ A medida que editamos archivos, Git los ve como modificados, porque los has cambiado desde tu última confirmación.
- ▶ Debemos preparar estos archivos modificados y luego confirmar los cambios preparados, y así sucesivamente durante todo el ciclo de proyecto.

File Status Lifecycle



Trabajando con repositorios GIT

COMPROBANDO EL ESTADO DE TUS ARCHIVOS

- ▶ Nuestra herramienta principal para determinar el estado de nuestros qué archivos es el comando “status”.
- ▶ Ejemplo: Si ejecutamos este comando después de clonar un repositorio, veremos algo así:

```
$ git status
```

```
# On branch master
```

```
nothing to commit, working directory clean
```

- ▶ Esto significa que tenemos un directorio de trabajo limpio, es decir, sin archivos bajo seguimiento y modificados
- ▶ Git tampoco ve ningún archivo que no esté bajo seguimiento, o estaría listado ahí.
- ▶ Además, podemos ver que nos habla de la rama en la que estamos (master). Hablaremos de ello más adelante

Trabajando con repositorios GIT

COMPROBANDO EL ESTADO DE TUS ARCHIVOS

- ▶ Ejemplo: Añadimos un nuevo archivo al proyecto. Archivo README. Si el archivo no existía y ejecutamos `git status`, veremos:

```
$ git status
```

```
# On branch master
```

```
# Untracked files:
```

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

```
#   README
```

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

- ▶ Vemos el archivo README bajo la cabecera “Archivos sin seguimiento” (“Untracked files”).
- ▶ Git ve un archivo que no estaba en la instantánea anterior;
- ▶ Git no empezará a incluirlo en las confirmaciones de tus instantáneas hasta que se lo indiques explícitamente. Se hace para evitar errores al añadir

Trabajando con repositorios GIT

SEGUIMIENTO DE NUEVOS ARCHIVOS

- ▶ Para empezar el seguimiento de un nuevo archivo se usa el comando “git add”.
- ▶ Ejemplo: para iniciar seguimiento del archivo README:

```
$ git add README
```

- ▶ Nos daría este resultado si volvemos a hacer “git status”:

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
# (use "git reset HEAD <file>..." to unstage)
```

```
#
```

```
# new file: README
```

- ▶ Ahora aparece en “Changes to be committed”. Sería la nueva instantánea
- ▶ El comando git add recibe ruta de un archivo o directorio; si es un directorio, añade todos los archivos que contenga de manera recursiva.

Trabajando con repositorios GIT

PREPARANDO ARCHIVOS MODIFICADOS

- ▶ Supongamos que modificamos un archivo que estuviese bajo seguimiento.
- ▶ Ejemplo: Si modificamos el archivo index.html que esta bajo seguimiento, y ejecutamos el comando status de nuevo, vemos algo así:

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
#   (use "git reset HEAD <file>..." to unstage)
```

```
#   new file:   README
```

```
# Changes not staged for commit:
```

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

```
#   modified:   index.html
```

Trabajando con repositorios GIT

PREPARANDO ARCHIVOS MODIFICADOS

- ▶ El archivo index.html 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.
- ▶ Para prepararlo, ejecuta el comando `git add`
- ▶ “`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

Trabajando con repositorios GIT

PREPARANDO ARCHIVOS MODIFICADOS

- ▶ Ejecutamos `git add` para preparar el archivo `index.html`, y volvemos a ejecutar `git status`

```
$ git add index.html
```

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
#   (use "git reset HEAD <file>..." to unstage)
```

```
#   new file:   README
```

```
#   modified:   index.html
```

Trabajando con repositorios GIT

PREPARANDO ARCHIVOS MODIFICADOS

- ▶ Ambos archivos están ahora preparados y se incluirán en tu próxima confirmación.
- ▶ Supongamos que en este momento recuerdas que tenías que hacer una pequeña modificación en index.html antes de confirmarlo.
- ▶ Lo vuelves abrir, haces ese pequeño cambio, y ya estás listo para confirmar. Sin embargo, si vuelves a ejecutar “git status” ves lo siguiente:

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
#   (use "git reset HEAD <file>..." to unstage)
```

```
#   new file:   README
```

```
#   modified:   index.html
```

```
# Changes not staged for commit:
```

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

```
#   modified:   index.html
```

Trabajando con repositorios GIT

PREPARANDO ARCHIVOS MODIFICADOS

- ▶ index.html aparece listado como preparado y como no preparado.
- ▶ ¿Porque? Git prepara un archivo de cuando ejecutamos el comando “git add”
- ▶ Si haces “git commit” ahora, la versión de “index.html” que se incluirá en la confirmación es la de cuando se ejecutó el comando “git add”, no la versión que vemos ahora en el directorio de trabajo.
- ▶ Si modificamos un archivo después de haber ejecutado “git add”, debemos ejecutar de nuevo “git add” para preparar la última versión del archivo:

```
$ git add index.html
```

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
#   (use "git reset HEAD <file>..." to unstage)
```

```
#   new file:   README
```

```
#   modified:   index.html
```

Trabajando con repositorios GIT

IGNORAR ARCHIVOS

- ▶ A veces tendremos archivos que no queremos que Git añada automáticamente o muestre como no versionado.
- ▶ Suelen ser archivos generados automáticamente, como logs, o archivos generados por compilador.
- ▶ Para estos casos se puede crear un archivo llamado `.gitignore`, donde se listan los patrones de nombres que deseamos ignorar.
- ▶ Ejemplo de archivo `.gitignore`:

```
$ cat .gitignore
```

```
*.[oa]
```

```
*~
```

Trabajando con repositorios GIT

IGNORAR ARCHIVOS

- ▶ Primera línea: ignorar cualquier archivo cuyo nombre termine en “.o” o “.a”, para archivos resultado de una compilación de código.
- ▶ Segunda línea: ignorar archivos que terminan en tilde (~), usada por muchos editores de texto, como Emacs, para marcar archivos temporales.
- ▶ Se pueden incluir directorios de log, temporales, documentación generada automáticamente, etc.
- ▶ Es una buena idea configurar un archivo “.gitignore” antes de empezar a trabajar, para así no confirmar archivos indeseados en el repositorio

Trabajando con repositorios GIT

IGNORAR ARCHIVOS - otro ejemplo

```
# a comment - this is ignored
```

```
# no .a files
```

```
*.a
```

```
# but do track lib.a, even though you're ignoring .a files above
```

```
!lib.a
```

```
# only ignore the root TODO file, not subdir/TODO
```

```
/TODO
```

```
# ignore all files in the build/ directory
```

```
build/
```

```
# ignore doc/notes.txt, but not doc/server/arch.txt
```

```
doc/*.txt
```

```
# ignore all .txt files in the doc/ directory
```

```
doc/**/*.txt
```


Trabajando con repositorios GIT

IGNORAR ARCHIVOS

- ▶ Las reglas para los patrones que pueden ser incluidos en el archivo `.gitignore` son:
 - ▶ Líneas en blanco, o comienzo por `#`, son ignoradas
 - ▶ Puedes usar patrones glob estándar
 - ▶ Se indican un directorios añadiendo una barra hacia delante (`/`) al final
 - ▶ Se puede negar un patrón añadiendo una exclamación (`!`) al principio
- ▶ Patrones glob: expresiones regulares simplificadas que pueden ser usadas por las shells.
 - ▶ Asterisco (`*`) reconoce cero o más caracteres
 - ▶ `[abc]` reconoce cualquier carácter de los especificados entre corchetes (en este caso, a, b o c)
 - ▶ Interrogación (`?`) reconoce un único carácter
 - ▶ Caracteres entre corchetes separados por un guión (`[0-9]`) reconoce cualquier carácter entre ellos (en este caso, de 0 a 9)

Trabajando con repositorios GIT

VIENDO LOS CAMBIOS PREPARADOS Y NO PREPARADOS

- ▶ Si queremos saber exactamente lo que ha cambiado y no sólo qué archivos fueron modificados se pueden usar el comando “git diff”.
- ▶ “git diff” te muestra exactamente las líneas añadidas y eliminadas
- ▶ Ejemplo: supongamos que queremos editar y preparar el archivo README otra vez, y luego editar el archivo index.html sin prepararlo. Si ejecutas el comando status, se verá algo así:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#   new file:   README
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   modified:   index.html
```

Trabajando con repositorios GIT

VIENDO LOS CAMBIOS PREPARADOS Y NO PREPARADOS

- ▶ Para ver lo que hemos modificado pero aún no has preparado, escribimos git diff:
- ▶ 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
diff --git a/index.html b/index.html
index 4afab7c..02eb7e4 100644
--- a/index.html
+++ b/index.html
@@ -7,7 +7,7 @@
<div id="info">
-     <a
href="wdfdflink_deejemplo1.com">LinkEjemplo1</a>
+     <a
href="www.link_deejemplo1.com">LinkEjemplo1</a>
      <a
href="www.link_deejemplo2.com">LinkEjemplo2</a>
      <button id="boton1">Cambiar Estilo</button>
      <button id="boton2">Cambiar Estilo2</button>
```

Trabajando con repositorios GIT

CONFIRMANDO CAMBIOS

- ▶ Tras dejar el área preparación como queremos, podemos confirmar los cambios. Antes de confirmar los cambios conviene ejecutar “git status” para cerciorarnos al 100%
- ▶ Usamos el comando “git commit”. Veremos algo como:

```
# Please enter the commit message for your changes. Lines
starting
```

```
# with '#' will be ignored, and an empty message aborts the
commit.
```

```
# On branch master
```

```
# Changes to be committed:
```

```
#   (use "git reset HEAD <file>..." to unstage)
```

```
#       new file:   README
```

```
#       modified:   index.html
```

```
".git/COMMIT_EDITMSG" 10L, 283C
```

Trabajando con repositorios GIT

CONFIRMANDO CAMBIOS

- ▶ Puedes escribir tu propio mensaje de confirmación. Así ayudará a que sepas el motivo de ese cambio.
- ▶ Se puede escribir el mensaje de confirmación desde la propia línea de comandos mediante la opción -m:

```
$ git commit -m "Arreglado bug de login de usuario"  
[master]: created 333dc4f: "Fix benchmarks for speed"  
2 files changed, 3 insertions(+), 0 deletions(-)  
create mode 100644 README
```

Trabajando con repositorios GIT

CONFIRMANDO CAMBIOS

- ▶ Tras esto, ya habrás hecha tu primera confirmación.
- ▶ Puedes ver que el comando commit ha dado cierta información sobre la confirmación:
 - ▶ A qué rama has confirmado (master)
 - ▶ Cuál es su suma de comprobación SHA-1 de la confirmación (333dc4f)
 - ▶ Archivos se modificados
 - ▶ Estadísticas acerca de cuántas líneas se han añadido y cuántas se han eliminado.
- ▶ La confirmación registra la instantánea de tu área de preparación
- ▶ Cualquier cosa que no preparases sigue estando modificada. Puedes hacer otra confirmación para añadirla a la historia del proyecto
- ▶ Cada vez que confirmas, estás registrando una instantánea de tu proyecto, a la que puedes volver o con la que puedes comparar más adelante

Trabajando con repositorios GIT

SALTANDO EL ÁREA DE PREPARACIÓN

- ▶ En ocasiones, el área de preparación es demasiado compleja para las necesidades de tu flujo de trabajo. 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. Permite omitir la parte de “git add”:

```
$ git status
```

```
# On branch master
```

```
# Changes not staged for commit:
```

```
#
```

```
#   modified:   index.html
```

```
#
```

```
$ git commit -a -m 'bug de login arreglado'
```

```
[master 83e38c7] added new benchmarks
```

```
1 files changed, 5 insertions(+), 0 deletions(-)
```

Trabajando con repositorios GIT

ELIMINAR ARCHIVOS

- ▶ Se debe eliminar de los archivos bajo seguimiento (concretamente del área de preparación), y después confirmar
- ▶ “git rm” se encarga de eso, y también elimina el archivo del directorio de trabajo, para que no lo veas entre los archivos sin seguimiento
- ▶ Si sólo eliminas el archivo de tu directorio de trabajo, aparecerá bajo la cabecera “Modificados pero no actualizados” (“Changes not staged for commit”) (es decir, sin preparar) de la salida del comando “git status”:

```
$ rm index2.html
```

```
$ git status
```

```
# On branch master
```

```
# Changes not staged for commit:
```

```
#   (use "git add/rm <file>..." to update what will be  
#   committed)
```

```
#       deleted:    index2.html
```


Trabajando con repositorios GIT

ELIMINAR ARCHIVOS

- ▶ La próxima vez que confirmemos, el archivo desaparecerá y dejará de estar bajo seguimiento.
- ▶ Si ya habías modificado el archivo y lo tenías en el área de preparación, 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.
- ▶ Puede que quieras mantener el archivo en tu equipo, pero interrumpir su seguimiento por parte de Git. Útil cuando olvidaste añadir algo a tu archivo .gitignore y lo añadiste accidentalmente. Para hacer esto, usa la opción “--cached”:
- ▶ `$ git rm --cached readme.txt`

Trabajando con repositorios GIT

HISTÓRICO DE CONFIRMACIONES

- Después de haber hecho varias confirmaciones, o si hemos clonado un repositorio que ya tenía un histórico de confirmaciones, queremos ver qué modificaciones se han llevado a cabo
- La herramienta más básica y potente para hacer esto es el comando “git log”

```
$ git log
commit ca82a6dff817ec66f44333307202690a93763949
Author: Pepe perez<pprez@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700
    changed the version number
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Pepe perez<pperez@gee-mail.com>
Date: Sat Mar 15 16:40:33 2008 -0700
    removed unnecessary test code
commit a11bef06a3f659402fe756333399ad00de2209e6
Author: Pepe Perez<peperez@gee-mail.com> Date: Sat Mar 15 10:31:28
2008 -0700 first commit
```

Trabajando con repositorios GIT

DEHACER CAMBIOS - MODIFICANDO LA ÚLTIMA CONFIRMACIÓN

- ▶ Es muy común que queramos deshacer cambios cuando confirmamos demasiado pronto y nos olvidamos de añadir algún archivo, o nos confundimos al introducir el mensaje de confirmación.
- ▶ Para volver a hacer la confirmación, ejecutar un “commit” con la opción “—amend”
- ▶ Ejemplo: si confirmas y luego te das cuenta de que se te olvidó preparar los cambios en uno de los archivos que querías añadir:

```
$ git commit -m 'initial commit'
```

```
$ git add forgotten_file
```

```
$ git commit --amend
```

Trabajando con repositorios GIT

DEHACER CAMBIOS - DESHACIENDO LA PREPARACIÓN DE UN ARCHIVO

- Supongamos que has modificado dos archivos, y quieres confirmarlos como cambios separados, pero tecleas accidentalmente "git add *" y preparas ambos.
- ¿Cómo sacar uno de ellos del área de preparación? El comando "git status" te lo recuerda:

```
$ git add .
```

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
#   (use "git reset HEAD <file>..." to unstage)
```

```
#
```

```
#       modified:   README.txt
```

```
#       modified:   index.html
```

Trabajando con repositorios GIT

DEHACER CAMBIOS - DESHACIENDO LA PREPARACIÓN DE UN ARCHIVO

- ▶ Aplicamos el comando `git reset HEAD <file>`
- ▶ El archivo `index.html` ahora está modificado, no preparado

```
$ git reset HEAD index.html
index.html: locally modified

$ git status
# On branch master

# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt

# Changes not staged for commit:
#   (use "git add <file>..." to update what
#   will be committed)
#
#   (use "git checkout -- <file>..." to discard
#   changes in working directory)
#
#       modified:   index.html
```

Trabajando con repositorios GIT

DEHACER CAMBIOS - DESHACIENDO MODIFICACIÓN DE UN ARCHIVO

- ▶ ¿Y si no queremos mantener las modificaciones que hecho sobre el index.html?
- ▶ ¿Cómo podemos deshacer cambios? Es decir, revertir el archivo al mismo estado en el que estaba cuando la confirmación
- ▶ Afortunadamente, “git status” también te dice como hacer esto. En la salida del último ejemplo, Git nos daba pistas con “checkout”

```
# Changes not staged for commit:
```

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

```
#   (use "git checkout -- <file>..." to discard changes in working directory)
```

```
#       modified:   index.html
```

Trabajando con repositorios GIT

DEHACER CAMBIOS - DESHACIENDO MODIFICACIÓN DE UN ARCHIVO

- ▶ Aplicando git checkout -- <file>
- ▶ Vemos que se han revertido los cambios
- ▶ Debemos ser conscientes del peligro de este comando: cualquier modificación hecha sobre este archivo ha desaparecido (lo has sobrescrito)
- ▶ Nunca usar este comando a no ser que estés absolutamente seguro de que no quieres el archivo

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

```
#   (use "git reset HEAD <file>..." to unstage)
```

```
#       modified:   README.txt
```

Repositorios remotos en GIT

- ▶ Para colaborar en cualquier proyecto Git, necesitas saber cómo gestionar tus repositorios remotos.
- ▶ Son versiones de tu proyecto que se encuentran alojados en Internet o en algún punto de la red
- ▶ Puedes tener varios, cada uno de los cuales puede ser de sólo lectura, o de lectura/escritura, según los permisos que tengas
- ▶ Colaborar con otros implica gestionar estos repositorios remotos, y mandar (push) y recibir (pull) datos de ellos cuando necesites compartir cosas
- ▶ Gestionar repositorios remotos implica:
 - ▶ Conocer cómo añadir repositorios nuevos
 - ▶ Eliminar aquellos que ya no son válidos
 - ▶ Gestionar ramas remotas e indicar si están bajo seguimiento o no
 - ▶ Etc.

Repositorios remotos en GIT

MOSTRANDO SUS REPOSITORIOS REMOTOS

- ▶ Para ver qué repositorios remotos tenemos configurados, se puede ejecutar el comando “git remote”
- ▶ Mostrará una lista con los nombres de los remotos que se hayan especificado
- ▶ Si has clonado tu repositorio, deberías ver por lo menos "origin" (nombre predeterminado que le da Git al servidor del que clonaste)

```
$ git clone git://github.com/schacon/ticgit.git Initialized empty  
Git repository in /private/tmp/ticgit/.git/ remote: Counting  
objects: 595, done. remote: Compressing objects: 100% (269/269),  
done. remote: Total 595 (delta 255), reused 589 (delta 253)  
Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.  
Resolving deltas: 100% (255/255), done.  
$ cd ticgit  
$ git remote  
origin
```

Repositorios remotos en GIT

MOSTRANDO SUS REPOSITORIOS REMOTOS

- Se puede añadir la opción -v, que muestra la URL asociada a cada repositorio remoto:

```
$ git remote -v origin git://github.com/schacon/ticgit.git  
(fetch) origin git://github.com/schacon/ticgit.git (push)
```

- Si tienes más de un remoto, te lista todos los repositorios. Ejemplo:

```
$ cd grit
```

```
$ git remote -v
```

```
bakkdoor git://github.com/bakkdoor/grit.git
```

```
cho45 git://github.com/cho45/grit.git
```

```
defunkt git://github.com/defunkt/grit.git
```

```
koke git://github.com/koke/grit.git
```

```
origin git@github.com:mojombo/grit.git
```

Repositorios remotos en GIT

AÑADIENDO REPOSITORIOS REMOTOS

- Para añadir un nuevo repositorio Git remoto, asignándole un nombre con el que referenciarlo fácilmente, ejecuta “git remote add [nombre] [url]”:

```
$ git remote origin
```

```
$ git remote add pb git://github.com/paulboone/ticgit.git
```

```
$ git remote -v
```

```
origin git://github.com/schacon/ticgit.git pb  
git://github.com/paulboone/ticgit.git
```

- Ahora la cadena "pb" se puede usar en lugar de la URL

Repositorios remotos en GIT

AÑADIENDO REPOSITORIOS REMOTOS

- Ejemplo: si quieres recuperar cierta información que todavía no tienes en tu repositorio, puedes ejecutar:

```
$ git fetch pb
```

```
remote: Counting objects: 58, done.
```

```
remote: Compressing objects: 100% (41/41), done.
```

```
remote: Total 44 (delta 24), reused 1 (delta 0)
```

```
Unpacking objects: 100% (44/44), done.
```

```
From git://github.com/paulboone/ticgit
```

```
* [new branch]      master      -> pb/master
```

```
* [new branch]      ticgit      -> pb/ticgit
```

Repositorios remotos en GIT

AÑADIENDO REPOSITORIOS REMOTOS

- ▶ Para recuperar datos de tus repositorios remotos puedes ejecutar: `$ git fetch [remote-name]`
- ▶ Este comando recupera todos los datos del proyecto remoto que no tengas todavía. Después de hacer esto, deberías tener referencias a todas las ramas del repositorio remoto, que puedes unir o inspeccionar en cualquier momento

Repositorios remotos en GIT

ENVIANDO A TUS REPOSITORIOS REMOTOS

- ▶ Cuando tu proyecto se encuentra en un estado que quieres compartir, tienes que enviarlo a un repositorio remoto.
- ▶ El comando que te permite hacer esto es sencillo: “git push [nombre-remoto][nombre-rama]”.
- ▶ Si quieres enviar tu rama maestra (master) a tu servidor origen (origin), ejecutarías esto para enviar tu trabajo al servidor:
- ▶ `$ git push origin master`
- ▶ Este comando funciona únicamente si has clonado de un servidor en el que tienes permiso de escritura, y nadie ha enviado información mientras tanto. En caso de que haya cambios, deberías bajarte antes la última versión del servidor.

Repositorios remotos en GIT

INSPECCIONANDO REPOSITORIOS REMOTOS

- ▶ Si quieres ver más información acerca de un repositorio remoto en particular, puedes usar el comando `git remote show [nombre]`.
- ▶ Si ejecutas este comando pasándole el nombre de un repositorio, como origin, obtienes algo así:

```
$ git remote show origin * remote origin URL:  
git://github.com/schacon/ticgit.git Remote branch merged with 'git  
pull' while on branch master master Tracked remote branches master  
ticgit
```

- ▶ Esto lista la URL del repositorio remoto, así como información sobre las ramas bajo seguimiento. Este comando te recuerda que si estás en la rama maestra y ejecutas "git pull", automáticamente unirá los cambios a la rama maestra del remoto después de haber recuperado todas las referencias remotas. También lista todas las referencias remotas que ha recibido

Repositorios remotos en GIT

ELIMINANDO Y RENOMBRANDO REPOSITORIOS

- ▶ Si quieres renombrar una referencia a un repositorio remoto, en versiones recientes de Git puedes ejecutar `git remote rename`. Por ejemplo, si quieres renombrar pb a paul, puedes hacerlo de la siguiente manera:

```
$ git remote rename pb paul
```

```
$ git remote
```

```
origin
```

```
paul
```

- ▶ Esto también cambia el nombre de tus ramas remotas. Lo que antes era referenciado en pb/master ahora está en paul/master.
- ▶ Si por algún motivo quieres eliminar una referencia(has movido el servidor o ya no estás usando un determinado mirror, o quizás un contribuidor ha dejado de contribuir) puedes usar el comando `git remote rm`:

```
$ git remote rm paul
```

```
$ git remote
```

```
origin
```