



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

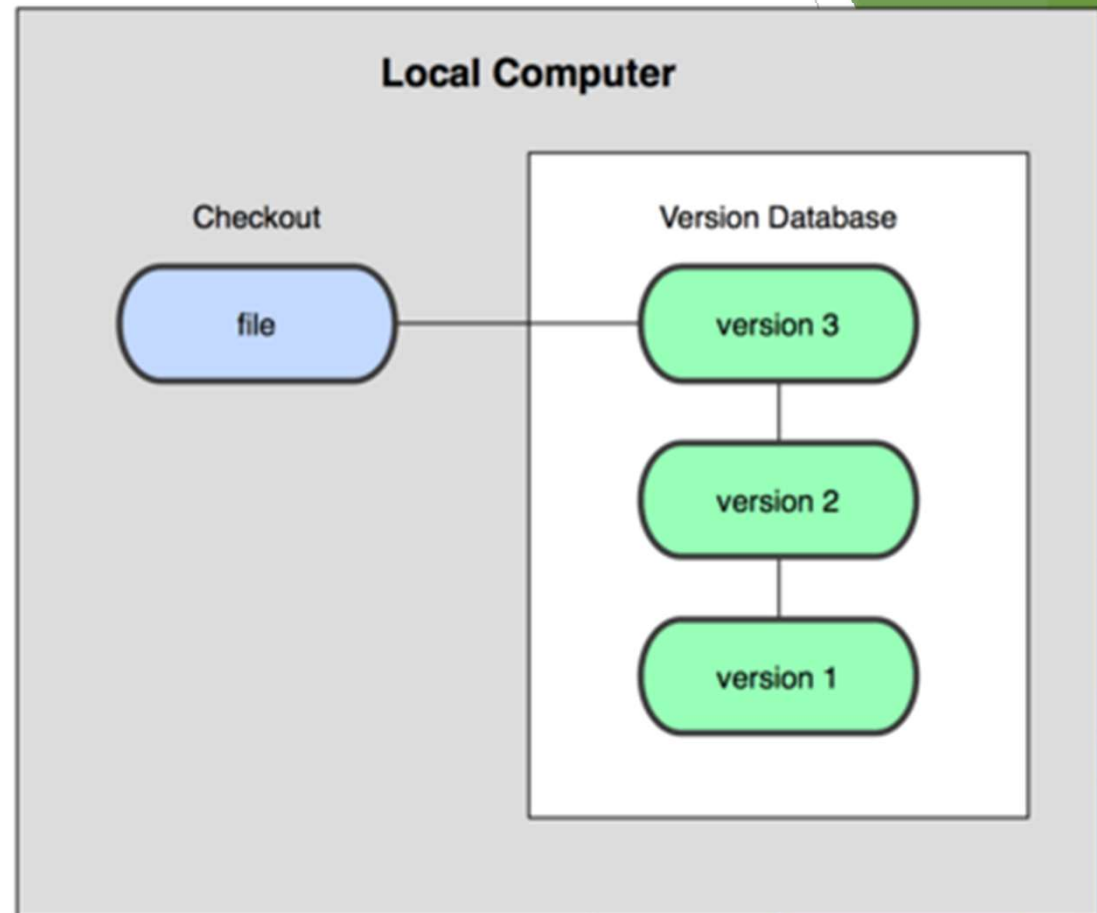


# Control de versiones

- ▶ 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

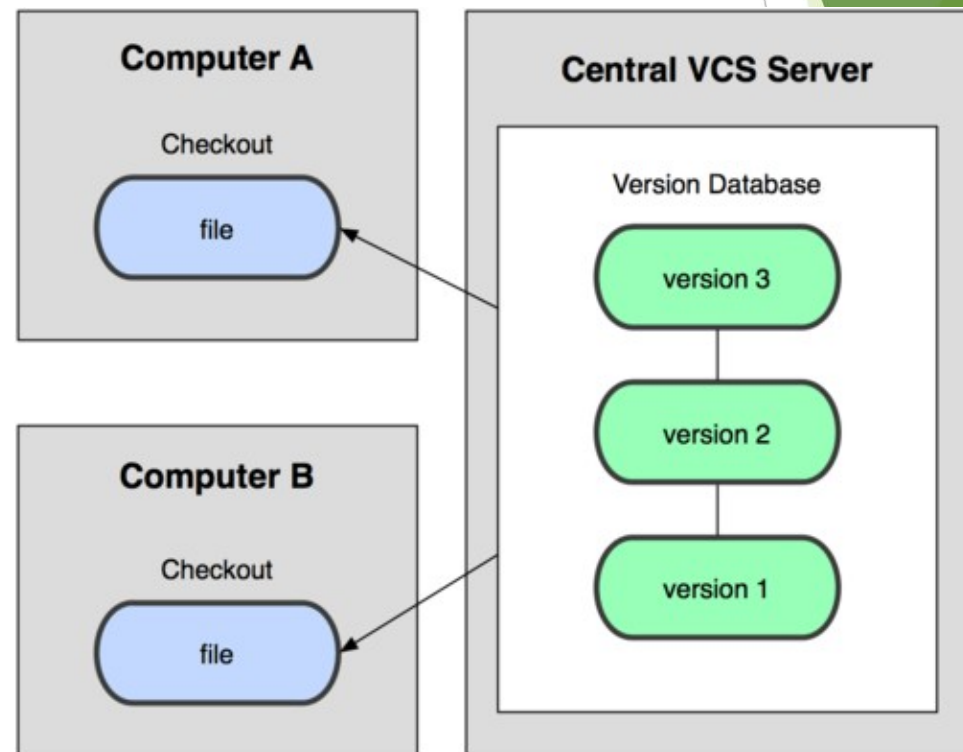


# Sistema de control de versiones local

- ▶ 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

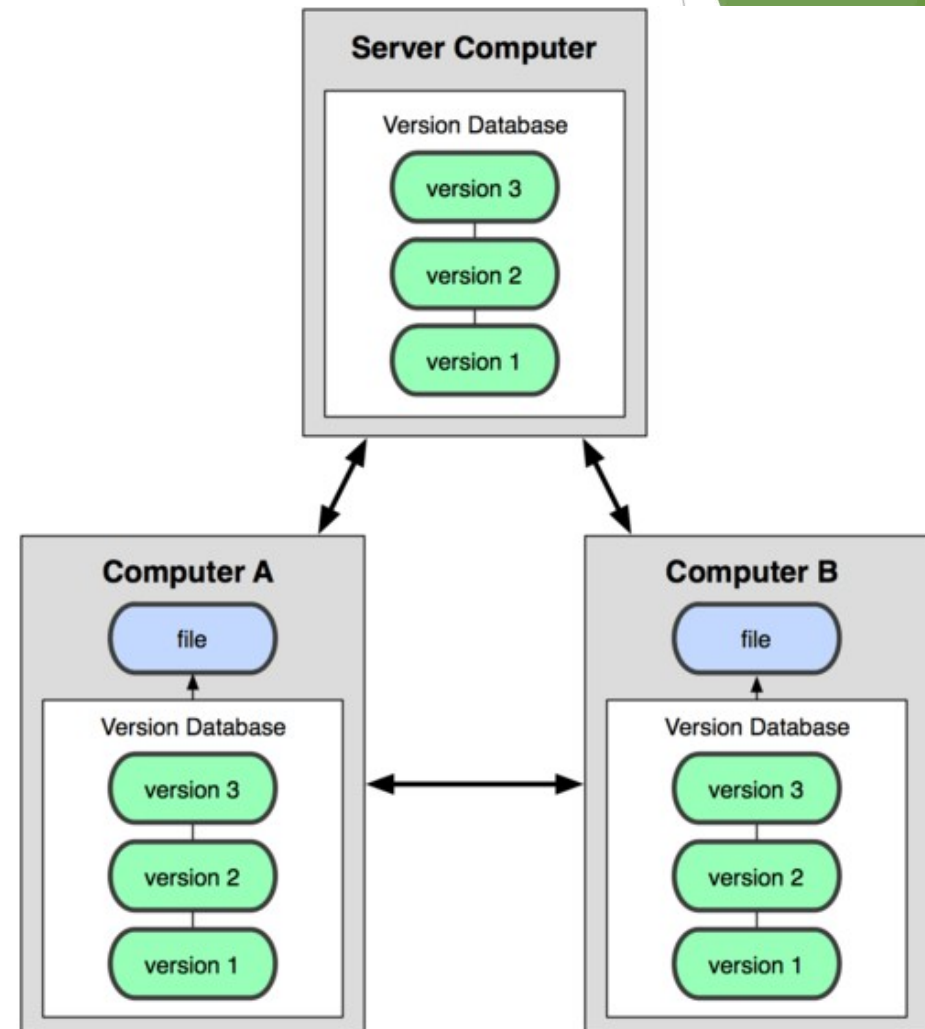


# Sistemas de control de versiones centralizados (CVCS)

- ▶ 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

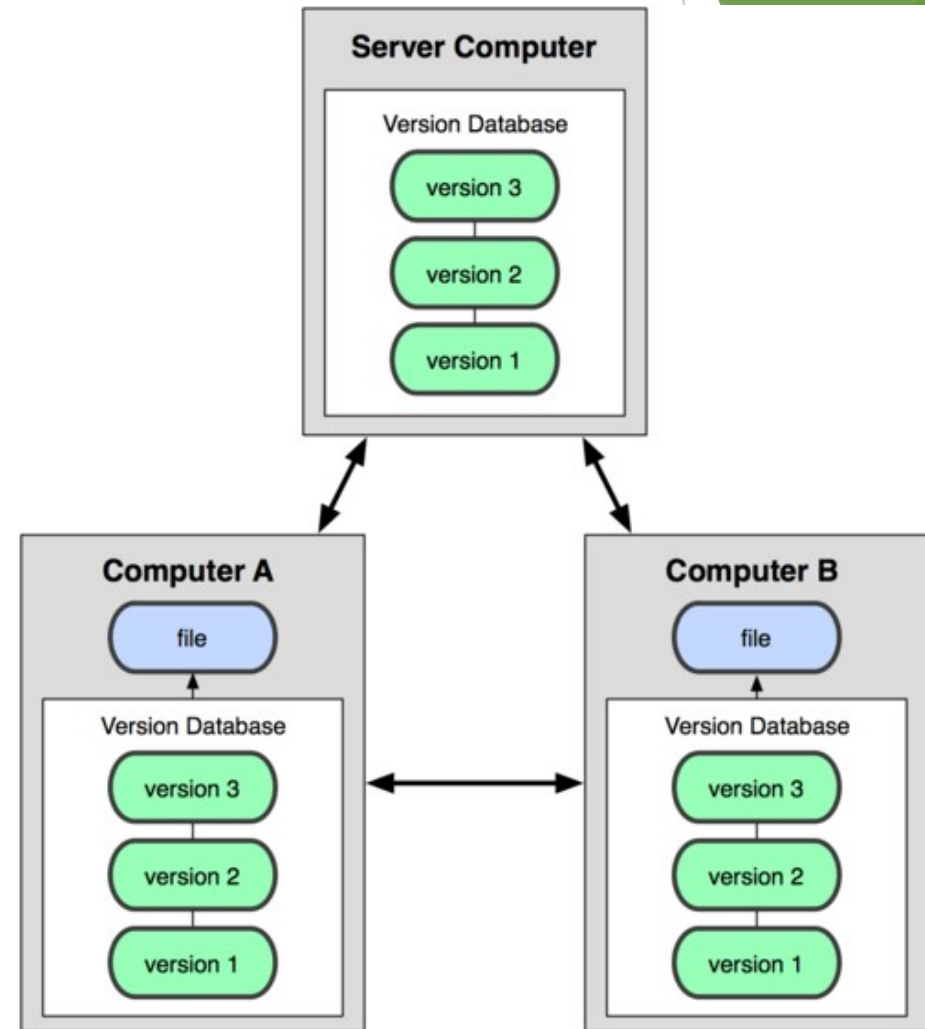
- ▶ Distributed Version Control Systems(DVCSs)
- ▶ Ejemplos de sistemas DVCS: Git, Mercurial, Bazaar o Darcs
- ▶ Los clientes no sólo descargan la última instantánea de los archivos. También replican completamente el repositorio.
- ▶ Si un servidor cae, y estos sistemas estaban colaborando a través de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

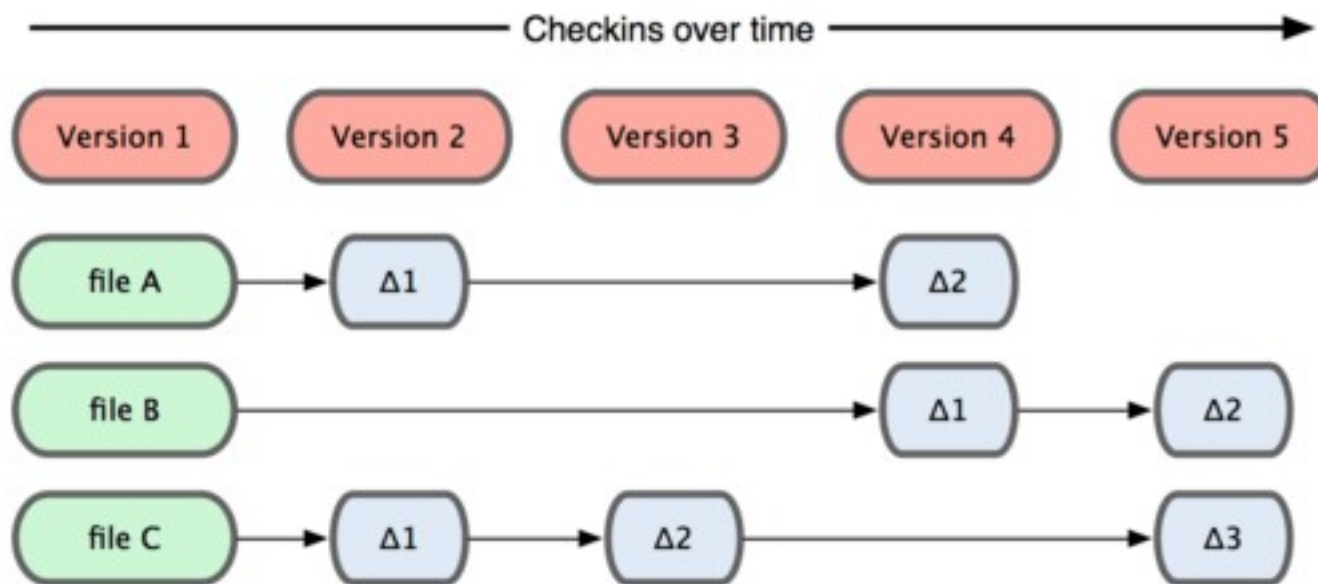
- ▶ Git es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente.
- ▶ Objetivo: llevar registro de los cambios en archivos de computadora 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

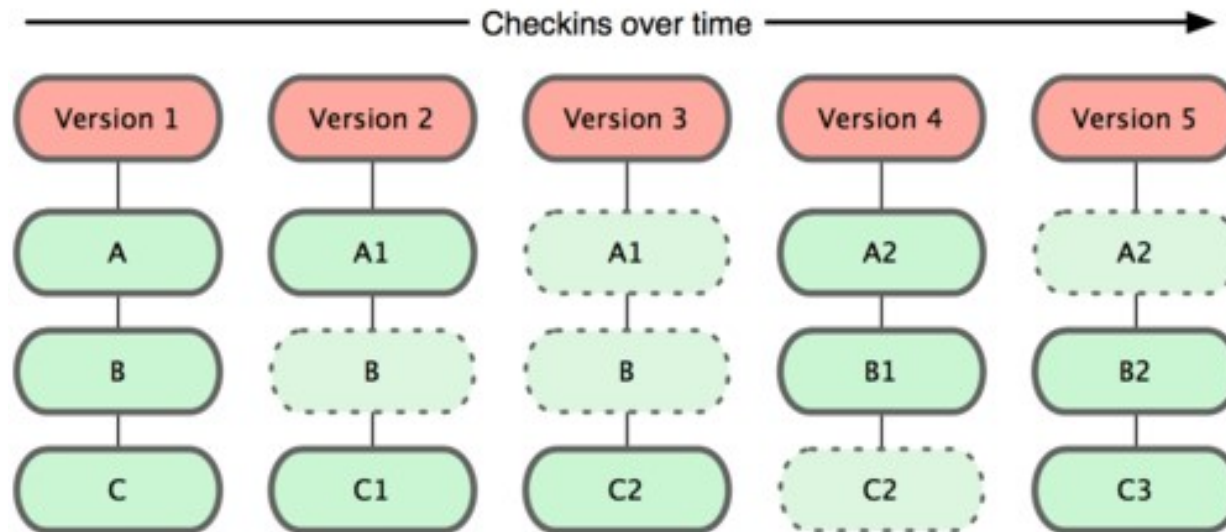
# Fundamentos de GIT

- ▶ 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 viajanado 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

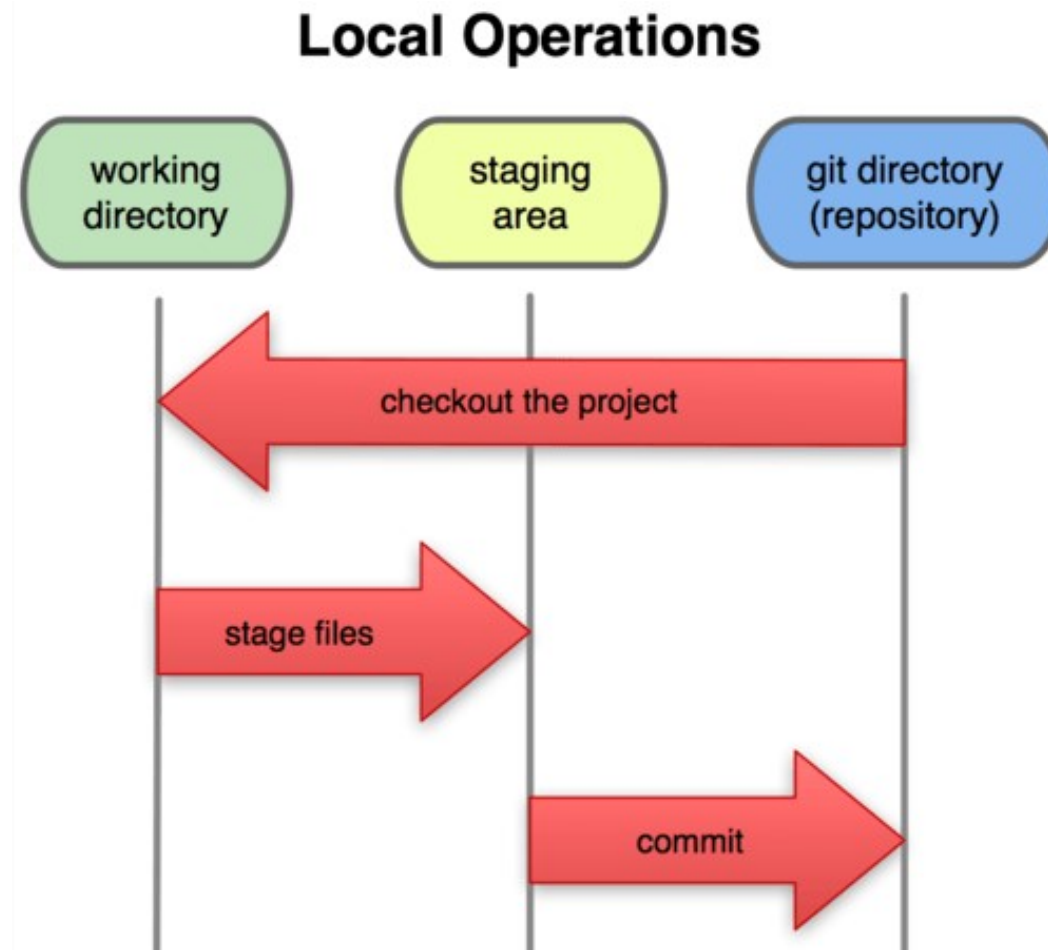
## LOS TRES ESTADOS PRINCIPALES:

- ▶ 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

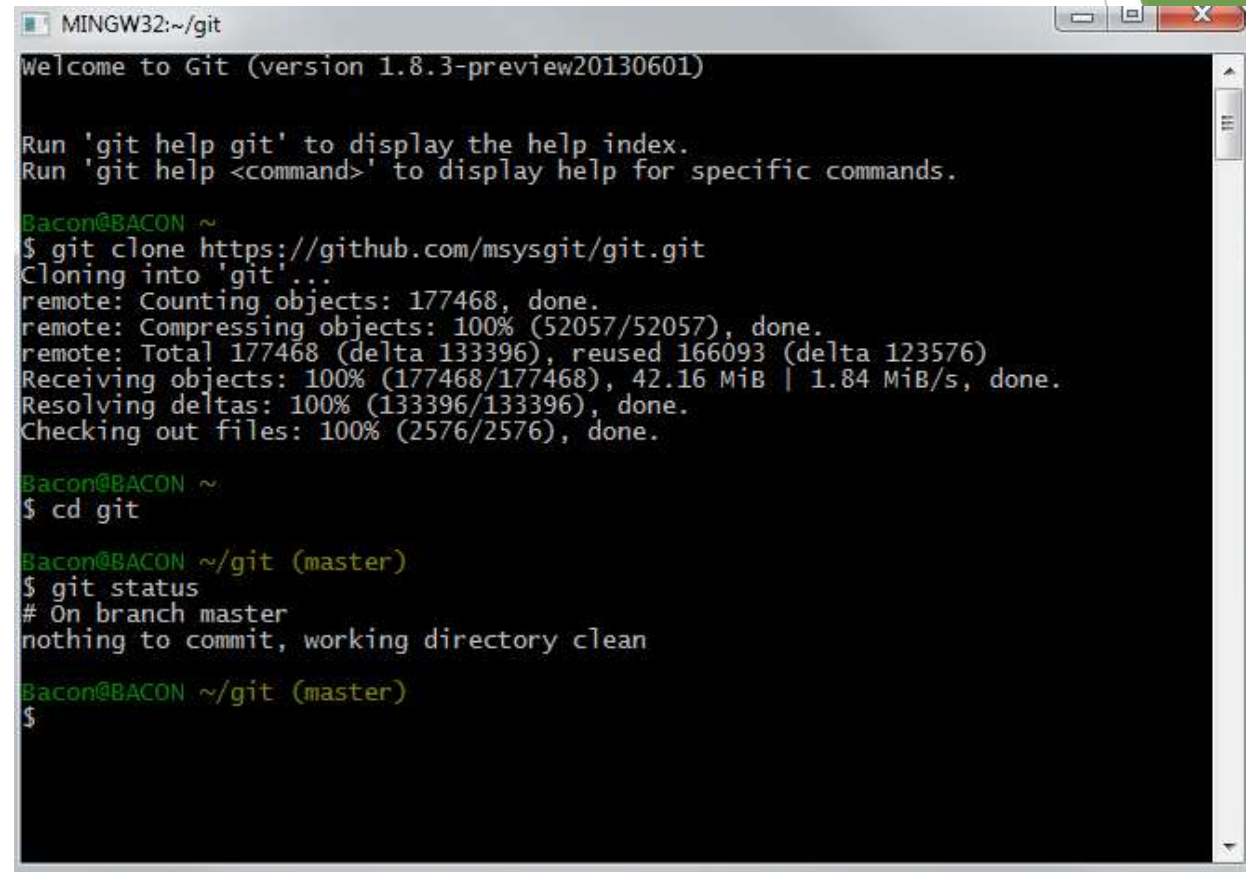
1. Modificar archivos en tu directorio de trabajo
  2. Preparar los archivos, añadiéndolos a tu “área de preparación”
  3. Confirmar los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esas instantáneas de manera permanente en tu directorio de Git
- ▶ Si una versión 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

A screenshot of a MINGW32 terminal window titled 'MINGW32:~/git'. The window shows the output of the 'git clone' command, which successfully clones the repository from GitHub. The terminal text is as follows:

```
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

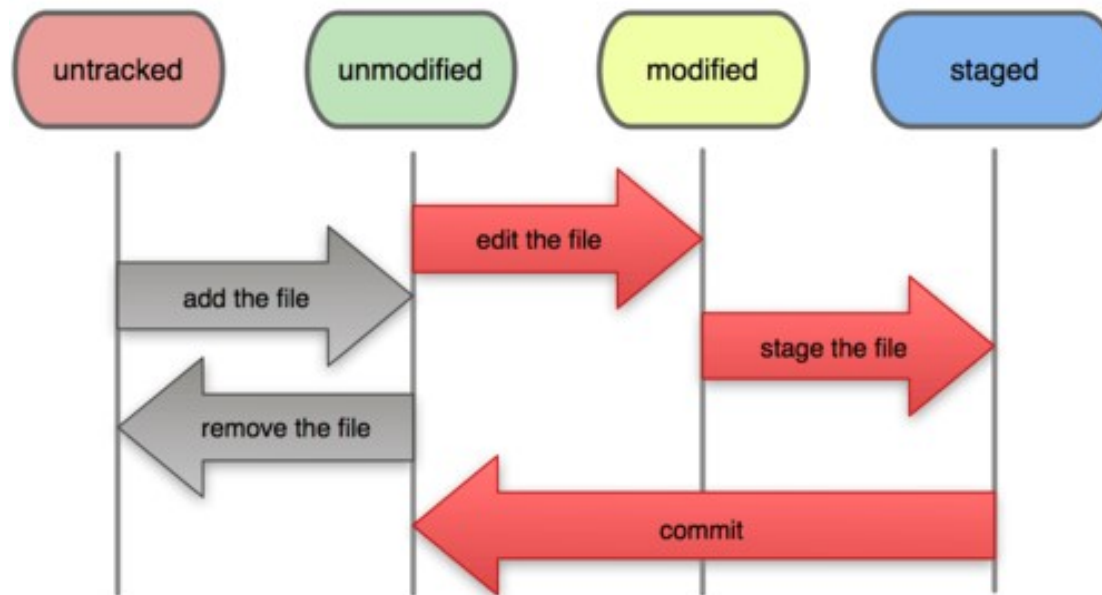
- ▶ Tenemos un repositorio Git completo, y una copia de trabajo de los archivos de ese proyecto.
- ▶ Necesitamos hacer algunos cambios, y confirmar instantáneas de esos cambios en el repositorio cada vez que el proyecto alcance un estado que deseemos guardar.
- ▶ Cada archivo en nuestro directorio de 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 puedes 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 he 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
```

# Ejercicio: Trabajando con repositorios GIT

- ▶ Probar en clase los comandos vistos hasta ahora. Hacer los ejercicios del 1-10 del siguiente enlace:

[https://www.jesusamieiro.com/wp-content/uploads/2014/03/20141009\\_Git\\_Ejemplo\\_GPUL\\_UDC.pdf](https://www.jesusamieiro.com/wp-content/uploads/2014/03/20141009_Git_Ejemplo_GPUL_UDC.pdf)

- ▶ Ejemplos:

[https://www.jesusamieiro.com/wp-content/uploads/2014/03/20141009\\_Git\\_Ejemplo\\_GPUL\\_UDC.pdf](https://www.jesusamieiro.com/wp-content/uploads/2014/03/20141009_Git_Ejemplo_GPUL_UDC.pdf)

[https://www.tutorialspoint.com/git/git\\_review\\_changes.htm](https://www.tutorialspoint.com/git/git_review_changes.htm)

<http://rogerdudler.github.io/git-guide/index.es.html>

<https://www.uco.es/aulasoftwarelibre/curso-de-git/>



The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern and dynamic visual effect.

# GIT - Repositorios remotos

# 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
```

# Repositorios remotos - GitHub

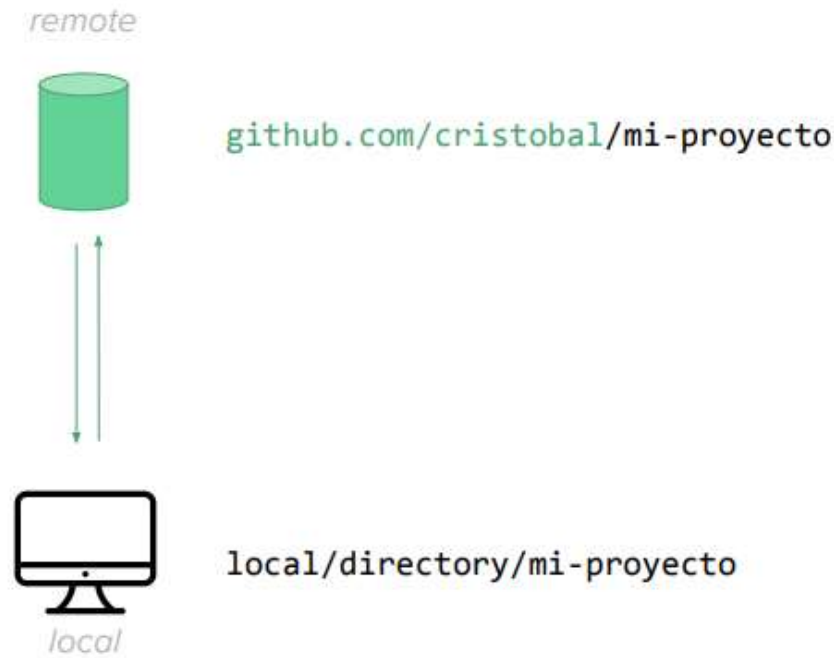
- ▶ GitHub es una plataforma de desarrollo colaborativo de software para alojar proyectos utilizando el sistema de control de versiones (VCS) Git
- ▶ Brinda herramientas muy útiles para el trabajo en equipo dentro un proyecto
- ▶ Es necesario crear una cuenta en GitHub para trabajar con repositorios remotos

<https://github.com/>

# Repositorios remotos - GitHub

Recordando:

- ▶ Un repositorio es la unidad básica. La forma más fácil de imaginarlo es como la carpeta donde se encuentra nuestro proyecto.
- ▶ En él tendremos nuestro código, podremos hacer una copia local (repositorio local) desde la cual podremos actualizar y subir cambios al repositorio remoto, y un largo etc



# Repositorios remotos - GitHub

## TRABAJANDO CON “COMMIT”

- ▶ Conjunto de cambios que se hacen al repositorio.
- ▶ Imaginemos que modificamos una línea de código de un archivo de nuestro repositorio. La forma de actualizarlo es hacer un “commit” de ese cambio.
- ▶ Antes de hacerlo, hemos de ver qué cambios han surgido en nuestro repositorio local comparándolo con el repo remoto. Después añadimos los cambios y finalmente hacemos el commit (empaquetamos los cambios).

# Repositorios remotos - GitHub

## COMPROBAR DIFERENCIAS ENTRE REPOSITORIO “LOCAL” Y “REMOTO”

► \$ git status

```
λ git status
On branch ocp-3.6
Your branch is up-to-date with 'origin/ocp-3.6'.
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:   cicd-template.yaml

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

# Repositorios remotos - GitHub

## COMPLETANDO UN “COMMIT”

- ▶ `$ git add .`
- ▶ `$ git status`

```
λ git status
On branch ocp-3.6
Your branch is up-to-date with 'origin/ocp-3.6'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   cicd-template.yaml
```

# Repositorios remotos - GitHub

## COMPLETANDO UN “COMMIT”

- ▶ Finalmente empaquetamos los cambios añadidos en un commit.
- ▶ `$ git commit -m “mensaje sobre los cambios”`

```
λ git status
On branch ocp-3.6
Your branch is up-to-date with 'origin/ocp-3.6'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   cicd-template.yaml
```



# Repositorios remotos - GitHub

## HACER UN “PUSH” AL REPOSITORIO REMOTO

- ▶ Cuando hacemos un PUSH significa que estamos “empujando” el paquete de cambios (el commit) al repositorio remoto.
- ▶ En el caso de que estemos en la rama ejemplo del repo local, los cambios serán actualizados en la rama ejemplo del repo remoto.
- ▶ `$ git push`

# Repositorios remotos - GitHub

## ACTUALIZAR REPOSITORIO LOCAL - USO DE “PULL”

- ▶ Para actualizar tu repositorio local al commit más nuevo se ejecuta el pull en el directorio de trabajo para “bajar y fusionar” los cambios remotos
- ▶ Si el repositorio local está actualizado, la consola mostrará un “Already up-to-date”
- ▶ `$ git pull`

# Repositorios remotos - GitHub

## CONFLICTOS PUSH/PULL

- ▶ Se puede dar el caso en el que existan conflictos
- ▶ Esto ocurre cuando se intenta hacer un PUSH a un repositorio que no tenemos actualizado en local
- ▶ Para ello, deberíamos hacer un PULL del cambio que no tenemos en local, generar un commit de nuevo y hacer un push de él (habiendo solucionado los conflictos)

# Repositorios remotos - GitHub

## TRABAJANDO CON PUSH/PULL

\$ git push

**Caso 1:** Todo ha ido bien. El repo se actualiza.

**Caso 2:**  $\nexists$  conflictos. Pero otrx compañerx ha hecho *push* antes que tú. Esto implica:

1. `$ git pull`
2. `$ git commit -m "merge with latest changes"`
3. `$ git push`

**Caso 3:**  $\exists$  conflictos. El código que pretendes “empujar” el repo entra en contradicción con el del último *commit*.

1. `$ git pull`
2. `-- resolver conflictos (visibles en tu IDE) --`
3. `$ git add .`
4. `$ git commit -m "solved merge conflicts"`
5. `$ git push`

# Repositorios remotos - GitHub

- ▶ Ejercicio: Trabajar con GitHub. Crear nuestro primer repositorio remoto y subir nuestro proyecto local
  - ▶ Crear una cuenta
  - ▶ Sincronizarlo con tu repositorio local. Es decir. Hacer un “PUSH” desde el repositorio local
  - ▶ Ojo: en nuestro primer uso, tenemos que autenticarnos en GitHub con claves generas HTTP/SSH desde GIT:
    - ▶ <https://help.github.com/articles/set-up-git/>
    - ▶ Tutorial para subir la clave SSH generada a nuestro repositorio remoto:  
<https://www.adictosaltrabajo.com/tutoriales/github-first-steps-upload-project/>
  - ▶ Tutorial para configurar Github:
  - ▶ <https://conociendogithub.readthedocs.io/en/latest/data/dinamica-de-uso/>

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern and dynamic visual effect. The shapes are layered, with some appearing more prominent than others, and they extend towards the edges of the frame.

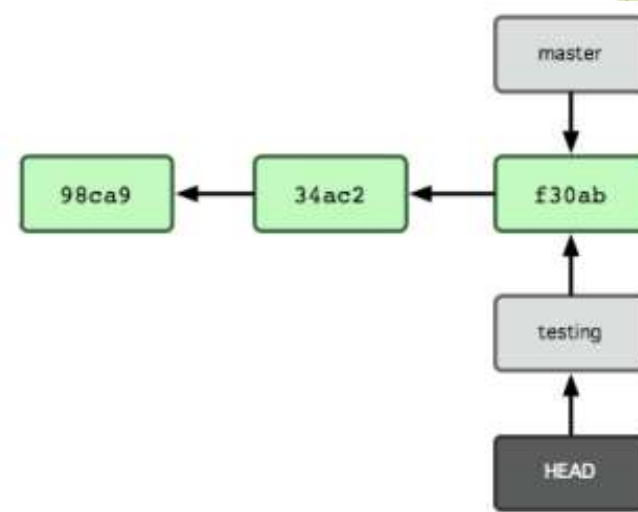
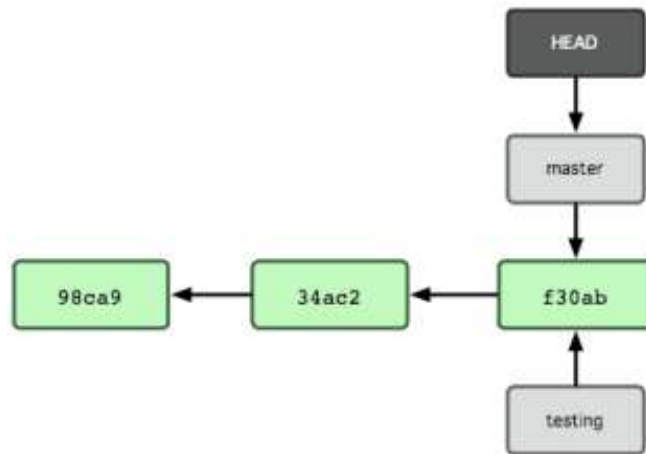
GIT - RAMAS

# Ramas en GIT

- ▶ Por defecto existe la rama(branch) master
- ▶ Similar al trunk de SVN desde el punto de vista estructural.
- ▶ NO ES IGUAL SEMÁNTICAMENTE AL trunk
  - ▶ El branch master se considera que contiene el código que se puede poner en producción.
  - ▶ El branch master es una referencia
- ▶ Listar branches / Averiguar branch actual:
  - ▶ `$ git branch [-v] -a`
  - ▶ La referencia HEAD apunta al branch actual

# Ramas en GIT

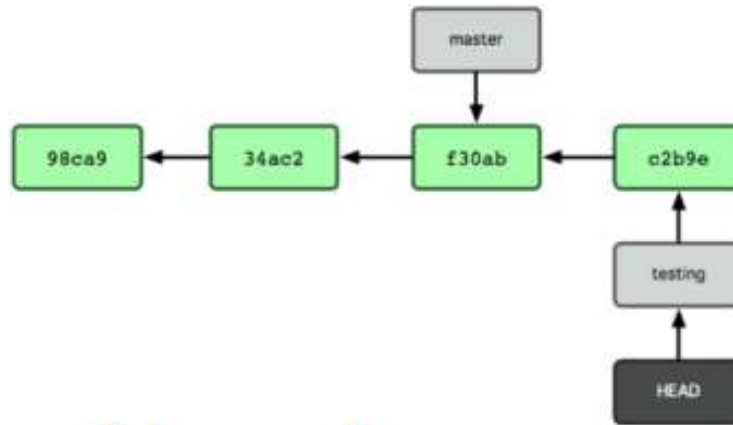
- ▶ Crear un branch (local)
  - ▶ `git branch <nombre branch>`
  - ▶ Crea un branch a partir del branch actual
- ▶ Pasar a trabajar a otro Branch
  - ▶ `git checkout <nombre branch>`
- ▶ Los dos comandos anteriores a la vez:
  - ▶ `git checkout -b <nombre branch>`



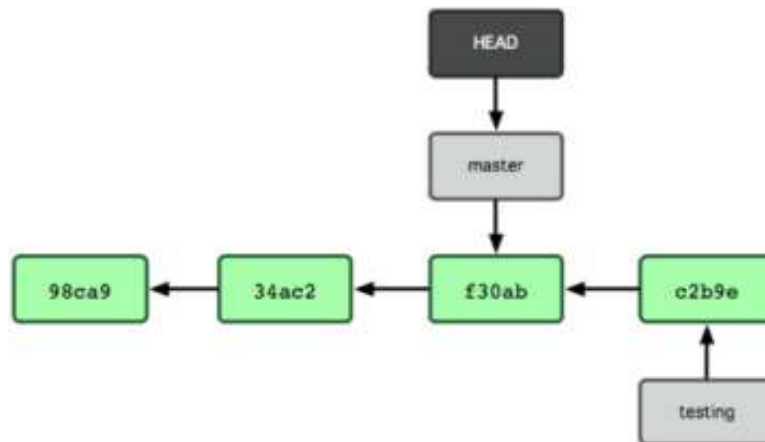


# Ramas en GIT

- Al hacer commit se realizarán sobre el branch activo

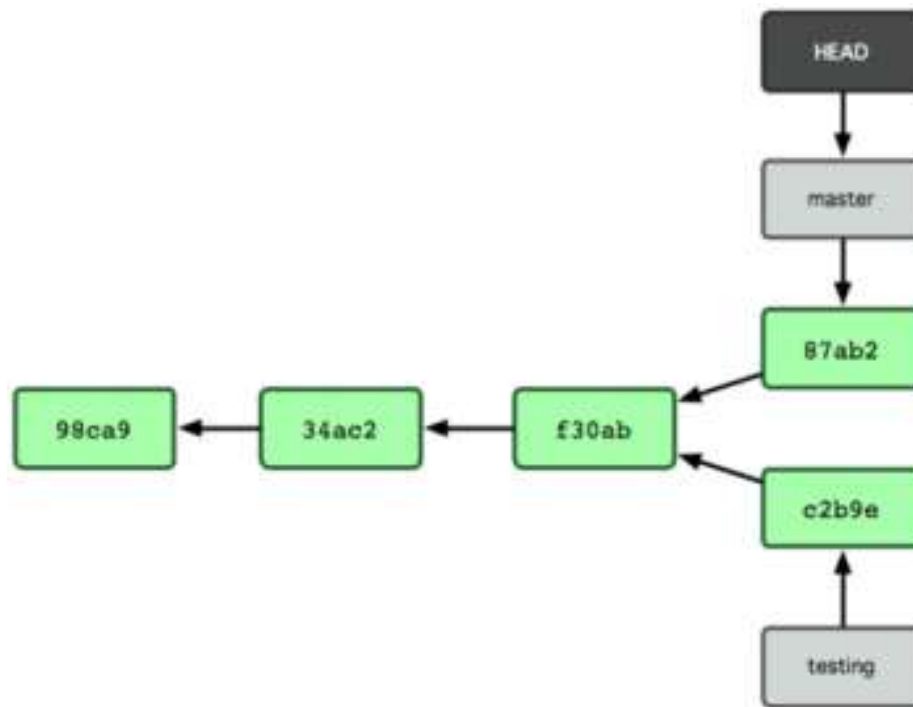


- Podemos volver al branch master cuando queramos



# Ramas en GIT

- ▶ Al modificar el branch master la estructura del repositorio cambia:
  - ▶ La historia de los branches diverge
  - ▶ Es necesario hacer un “merge” (reconciliar) los cambios



# Ramas en GIT

## FLUJO DE TRABAJO CON BRANCHES

1. Crear un branch cuando tengo que hacer una tarea o quiero experimentar algo.
2. Trabajar sobre el branch (desarrollar, hacer pruebas)
3. Nos aseguramos que la copia de trabajo está limpia. Es decir, que no hay ningún cambio pendiente
4. Actualizamos nuestro branch de trabajo con los cambios que haya habido en master
5. Cuando estamos contentos con el trabajo hacemos un “merge” del trabajo en el branch master

# Ramas en GIT

- ▶ ¿Cómo hacemos el merge?
  - ▶ Checkout del branch donde vamos a integrar los cambios
    - ▶ `$ git checkout master`
  - ▶ Integramos los cambios
    - ▶ `$ git merge tarea`
- ▶ Cuando se realizan los merges es posible que haya que resolver conflictos
- ▶ Conflictos: modificaciones sobre un mismo archivo que git no sabe resolver

# Ramas en GIT

- ▶ ¿Cómo averiguo los branches que hay?
  - ▶ `$ git branch [-a -v]`
  - ▶ El branch activo aparece con un '\*'
- ▶ ¿Cómo averiguo que branches NO están integrados con el branch activo?
  - ▶ `$ git branch --no-merged`
- ▶ ¿Cómo averiguo que branches SÍ están integrados con el branch activo?
  - ▶ `$ git branch --merged`
- ▶ Una vez que un branch está integrado puedo eliminarlo si lo deseo
  - ▶ `$ git branch -d <branch>`

# Publicar ramas remotas

- ▶ Cuando deseamos subir una rama al repositorio remoto con el resto del mundo, debemos “empujar” la rama (push)
- ▶ Las ramas locales no se sincronizan automáticamente con los remotos en los que escribes.
- ▶ Por ello, podemos trabajar con ramas privadas para el trabajo que no desees compartir, llevando al remoto sólo las ramas necesarias.
- ▶ Ejemplo. Para la rama “pruebas” en local, necesitamos guardar nuestros cambios en remoto. Basta con seguir la sintaxis: `$git push (remoto) (rama)`

```
$ git push origin pruebas
```

```
Counting objects: 20, done.
```

```
Compressing objects: 100% (14/14), done.
```

```
Writing objects: 100% (15/15), 1.74 KiB, done.
```

```
Total 15 (delta 5), reused 0 (delta 0)
```

```
To git@github.com:schacon/simplegit.git
```

```
* [new branch] pruebas -> pruebas
```

# Borrar ramas remotas

- ▶ En el caso de que ya has terminado con una rama remota.
- ▶ Es decir, hemos terminado cierta funcionalidad e incorporado (merge) a la rama master en el remoto(por ejemplo).
- ▶ Entonces ya podemos borrar la rama remota utilizando:

```
$ git push [nombreremoto] :[rama]
```

- ▶ Por ejemplo, borrando cierta rama del servidor remoto, puedes utilizar:

```
$ git push origin :pruebas
```

```
To git@github.com:schacon/simplegit.git
```

```
- [deleted]          pruebas
```

- ▶ Ahora nos faltaría por eliminar en local también la rama

# Ejercicios - Ramas en GIT

- ▶ 1. Probar en clase el ejercicio sobre ramas que aparece en el siguiente enlace:

[https://www.jesusamieiro.com/wp-content/uploads/2014/03/20141009\\_Git\\_Ejemplo\\_GPUL\\_UDC.pdf](https://www.jesusamieiro.com/wp-content/uploads/2014/03/20141009_Git_Ejemplo_GPUL_UDC.pdf)

- ▶ 2. Ejercicio para ramificar y fusionar:

<https://git-scm.com/book/es/v1/Ramificaciones-en-Git-Procedimientos-b%C3%A1sicos-para-ramificar-y-fusionar>

- ▶ 3. Ejercicio con ramas remotas

<https://git-scm.com/book/es/v1/Ramificaciones-en-Git-Ramas-Remotas>



The background features abstract green geometric shapes. On the left, a solid green trapezoid points towards the center. On the right, a complex arrangement of overlapping, semi-transparent green triangles and polygons creates a layered, dynamic effect. The central text is positioned between these two main graphic elements.

# GIT - ETIQUETAS

# Creando tags

- ▶ Tipos de tags en Git
  - ▶ Anotados. Genera un objeto en el repositorio
  - ▶ Ligeros. Similar a los branches
  - ▶ En ambos casos los tags se crean en el repositorio local
- ▶ Crear un tag ligero a partir del último commit
  - ▶ `$ git tag <nombre tag>`
- ▶ Crear tags anotados a partir del último commit
  - ▶ `$ git tag -a <nombre tag> [-m <mensaje>]`
- ▶ Crear tag anotado y firmado (con GPG)
  - ▶ `$ git tag -s <nombre tag> [-m <mensaje>]`

# ETIQUETAS - TAGS

- ▶ Git tiene la habilidad de etiquetar (tag) puntos específicos en la historia como importantes
- ▶ Se suele usar para marcar puntos donde se ha lanzado alguna versión (v1.0, y así sucesivamente)
- ▶ Veremos cómo listar etiquetas disponibles, crear nuevas etiquetas y tipos diferentes de etiquetas
- ▶ Git usa dos tipos principales de etiquetas:
  - ▶ Ligeras. Muy parecida a una rama que no cambia (puntero a una confirmación específica)
  - ▶ Anotadas. Almacenadas como objetos en la bbdd de Git. Tienen: suma de comprobación, nombre del etiquetador, correo electrónico y fecha, mensaje de etiquetado, pueden estar firmadas y verificadas con GNU Privacy Guard (GPG)
- ▶ Se recomienda crear etiquetas anotadas para disponer de toda esta información. Si necesitas una etiqueta temporal y no quieres almacenar el resto de información, mejor usar etiquetas ligeras

# Creando etiquetas

- ▶ Crear tag de un commit pasado
  - ▶ Utilizar git log para averiguar el SHA1 del commit
  - ▶ `$ git tag -a <nombre tag> [-m <mensaje>] <SHA1>`
  - ▶ Ejemplo:  
`$ git tag -a v1.4 -m 'my version 1.4'`
  - ▶ Ver resultado:  
`$ git tag`
- ▶ ¿Cómo compartir un tag?
  - ▶ `$ git push <remote> <nombre tag>`
- ▶ ¿Cómo compartir todos los tags?
  - ▶ `$ git push <remote> --tags`

# Creando etiquetas anotadas

- Crear una etiqueta anotada en Git es simple. La forma más fácil es especificar `-a` al ejecutar el comando `tag`:

```
$ git tag -a v1.4 -m 'my version 1.4'
```

```
$ git tag
```

```
v0.1
```

```
v1.3
```

```
v1.4
```

- El parámetro `-m` especifica el mensaje, el cual se almacena con la etiqueta. Si no se especifica un mensaje para la etiqueta anotada, Git lanza tu editor para poder escribirlo.

# Creando etiquetas anotadas

- Se pueden ver los datos de la etiqueta junto con la confirmación que fue etiquetada usando el comando git show:

```
$ git show v1.4
```

```
tag v1.4
```

```
Tagger: Scott Chacon <schacon@gee-mail.com>
```

```
Date: Mon Feb 9 14:45:11 2009 -0800
```

```
my version 1.4
```

```
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
```

```
Merge: 4a447f7... a6b4c97...
```

```
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date: Sun Feb 8 19:02:46 2009 -0800
```

```
Merge branch 'experiment'
```

- Muestra información del autor de la etiqueta, fecha en la que la confirmación fue etiquetada, y mensaje de anotación antes de mostrar la información de la confirmación

# Creando etiquetas anotadas

- Si ejecutamos `git show` sobre una etiqueta, se puede ver la firma GPG adjunta a ella:

```
► $ git show v1.5
tag v1.5
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:22:20 2009 -0800
my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.8 (Darwin)
iEYEABECAAYFAkmQurIACgkQ0N3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBw
gySN
Ki0An2JeAVUCAiJ70x6ZEtK+NvZAJ82/
=WryJ
-----END PGP SIGNATURE-----
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

# Creando etiquetas ligeras

- ▶ La otra forma de etiquetar confirmaciones es con una etiqueta ligera. Es la suma de comprobación de la confirmación almacenada en un archivo
- ▶ Ninguna otra información es guardada
- ▶ Para crear una etiqueta ligera no añadir las opciones -a, -s o -m:

```
$ git tag v1.4-lw
```

```
$ git tag
```

```
v0.1
```

```
v1.3
```

```
v1.4
```

```
v1.4-lw
```

```
v1.5
```



# Creando etiquetas ligeras

- ▶ Ejecutando el comando `git show` en la etiqueta, no veremos ninguna información extra como con las etiquetas anotadas. El comando sólo muestra la confirmación.

```
$ git show v1.4-lw
```

```
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
```

```
Merge: 4a447f7... a6b4c97...
```

```
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date: Sun Feb 8 19:02:46 2009 -0800
```

```
Merge branch 'experiment'
```

# Verificando etiquetas

- ▶ Para verificar una etiqueta firmada:

```
$ git tag -v [tag-name]
```

- ▶ Este comando utiliza GPG para verificar la firma. Necesitas la clave pública del autor de la firma en tu llavero para que funcione correctamente

# Compartiendo etiquetas

- ▶ Debemos enviarlas explícitamente a un servidor compartido después de haberlas creado
- ▶ Este proceso es igual a compartir ramas remotas
- ▶ `$ git push origin [tagname]`

```
$ git push origin v1.5
```

```
Counting objects: 50, done.
```

```
Compressing objects: 100% (38/38), done.
```

```
Writing objects: 100% (44/44), 4.56 KiB, done.
```

```
Total 44 (delta 18), reused 8 (delta 1)
```

```
To git@github.com:schacon/simplegit.git
```

```
* [new tag]          v1.5 -> v1.5
```

# Compartiendo etiquetas

- ▶ Si tienes un montón de etiquetas que quieres enviar a la vez, se puede usar la opción `--tags` en el comando `git push`.
- ▶ Transfiere todas tus etiquetas que no estén ya en el servidor remoto.

```
$ git push origin --tags
```

```
Counting objects: 50, done.
```

```
Compressing objects: 100% (38/38), done.
```

```
Writing objects: 100% (44/44), 4.56 KiB, done.
```

```
Total 44 (delta 18), reused 8 (delta 1)
```

```
To git@github.com:schacon/simplegit.git
```

```
* [new tag]          v0.1 -> v0.1
* [new tag]          v1.2 -> v1.2
* [new tag]          v1.4 -> v1.4
* [new tag]          v1.4-lw -> v1.4-lw
* [new tag]          v1.5 -> v1.5
```

# Ejercicio: probando etiquetas

- Ver ejercicio 22 para probar uso de etiquetas

[https://www.jesusamieiro.com/wp-content/uploads/2014/03/20141009\\_Git\\_Ejemplo\\_GPUL\\_UDC.pdf](https://www.jesusamieiro.com/wp-content/uploads/2014/03/20141009_Git_Ejemplo_GPUL_UDC.pdf)

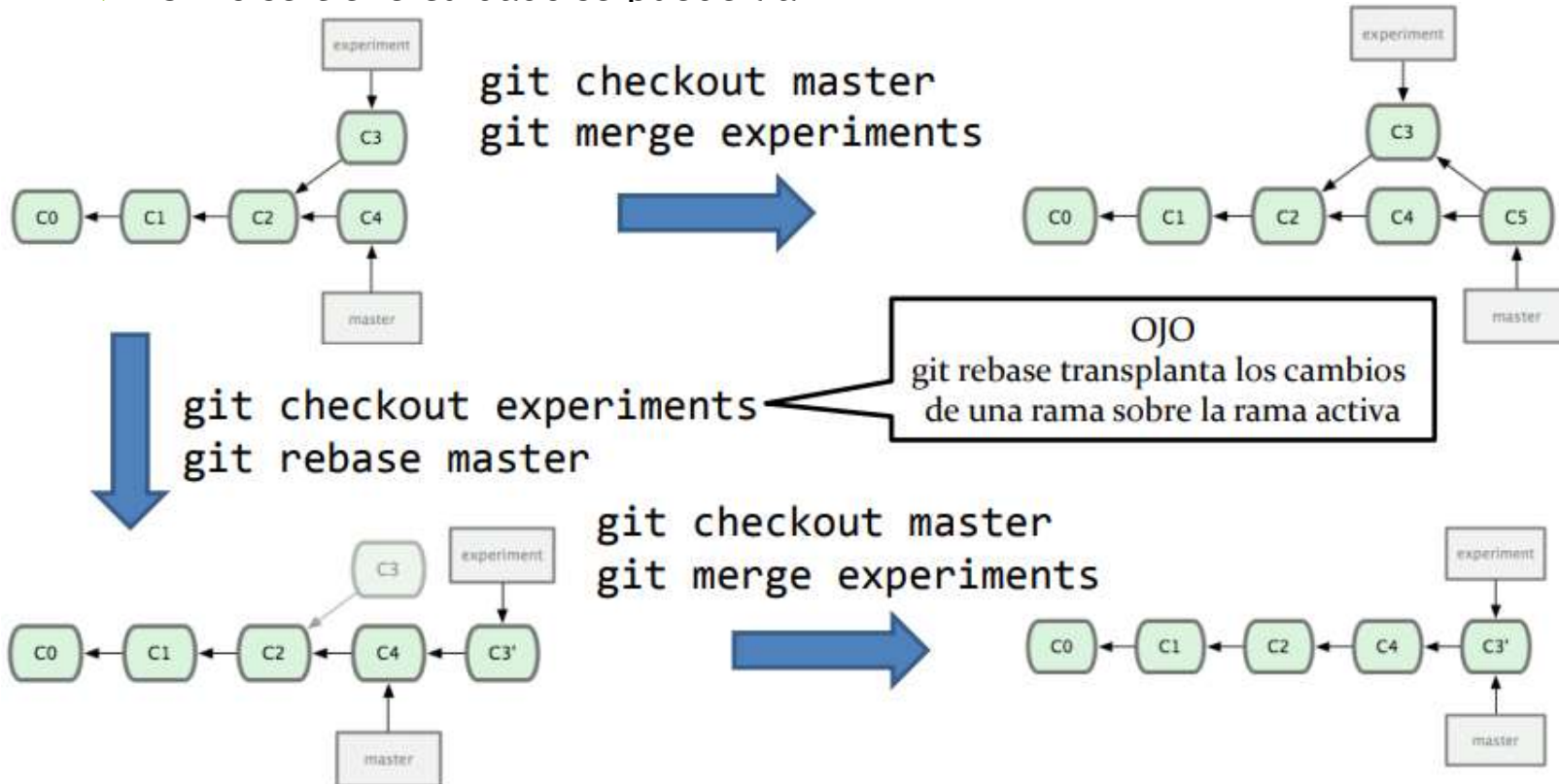


The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern and dynamic visual effect.

# GIT REBASE

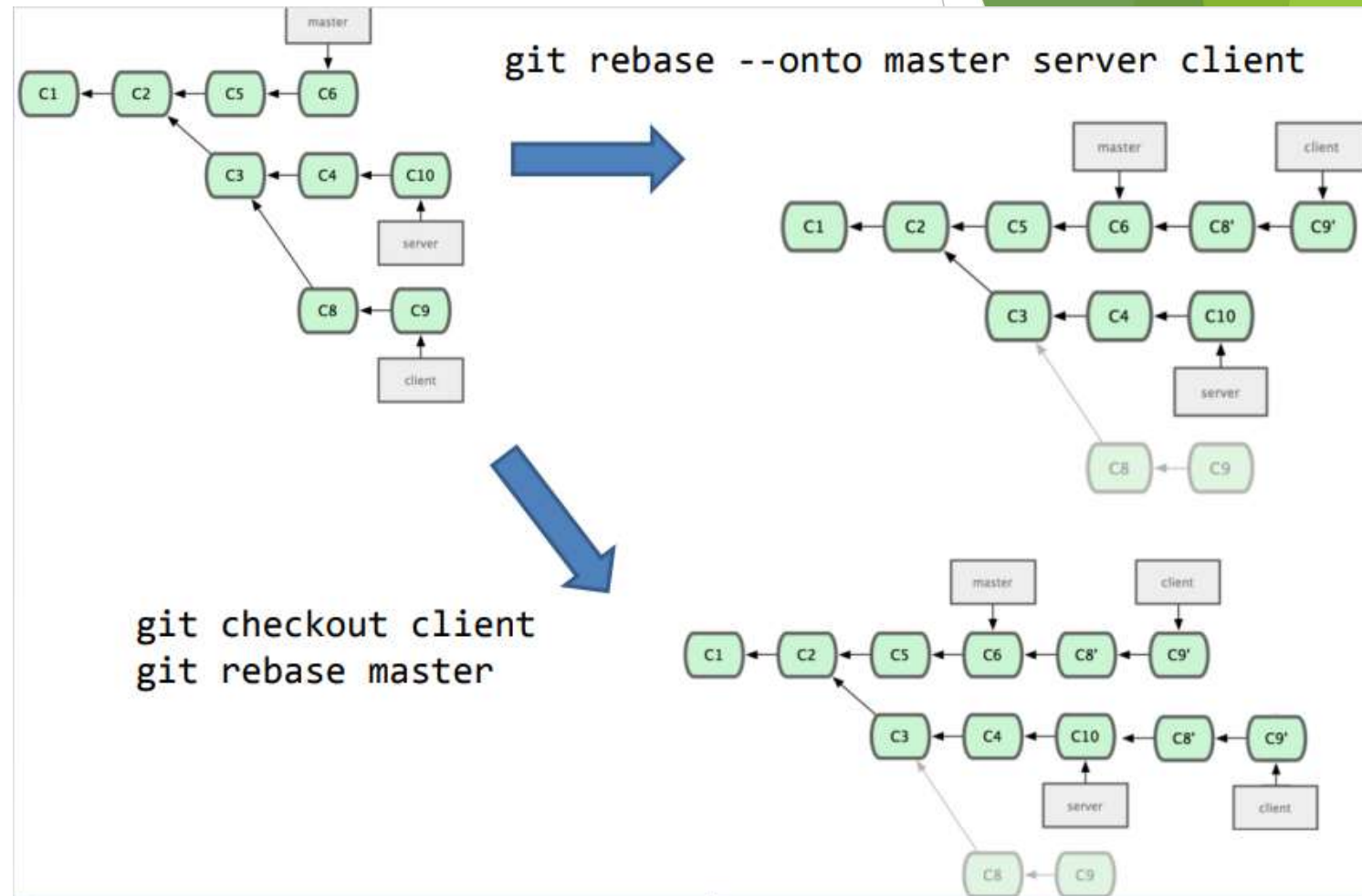
# GIT rebase

- ▶ Es otra manera de integrar cambios de un branch en otro
- ▶ ADVERTENCIA: Reescribe la historia del repositorio
  - ▶ Si no se tiene cuidado se puede liar



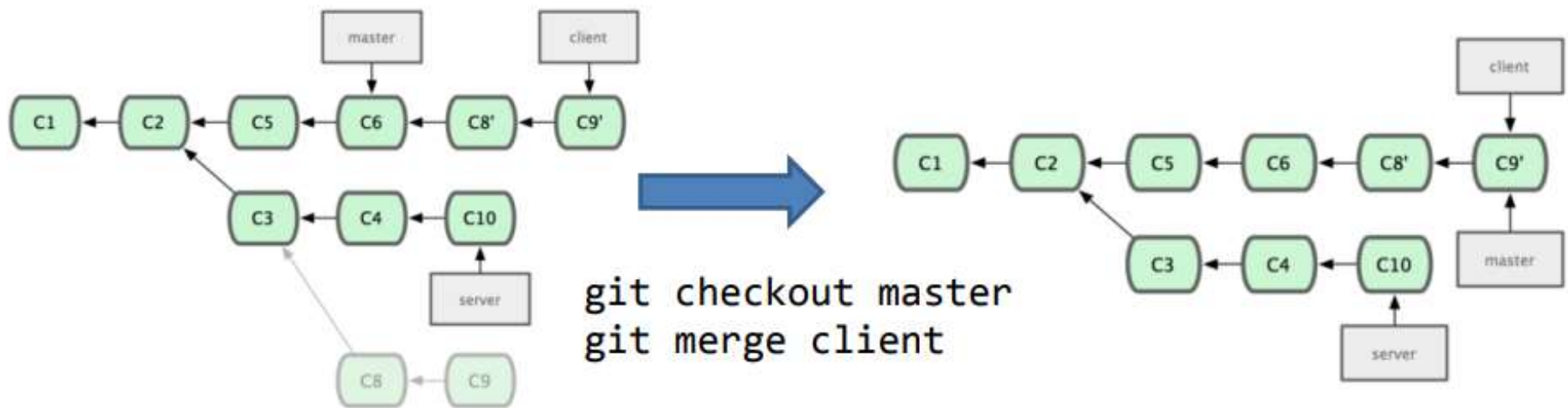
# GIT rebase - caso avanzado

- pull --rebase
- Permite traer los cambios de un remote y aplicarlos pero utilizando un rebase en vez de un merge
- Útil para no complicar la historia del repositorio y para abordar poco a poco la reconciliación con el branch del que hemos partido

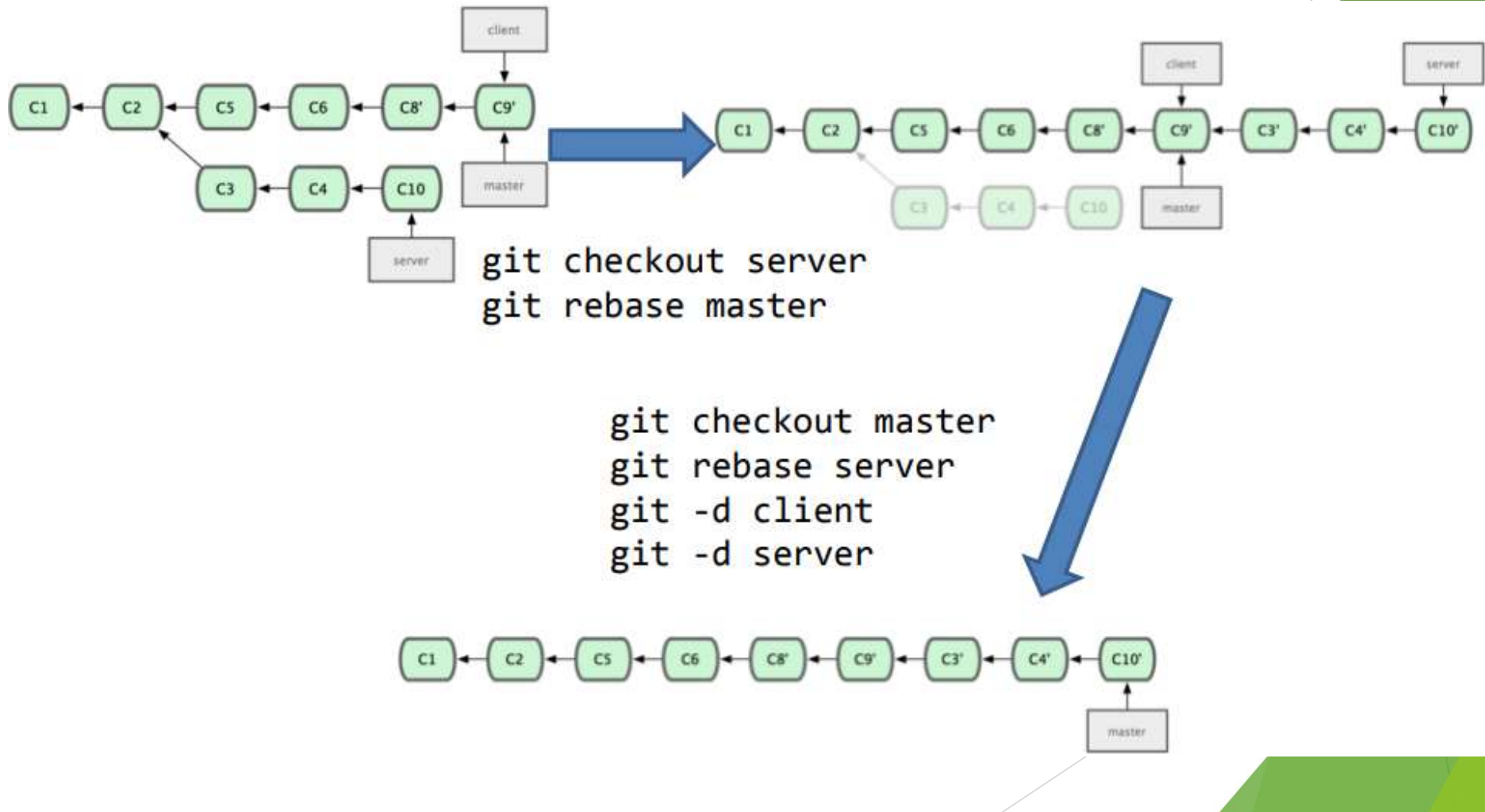




# GIT rebase - caso avanzado



# GIT rebase - caso avanzado



# GIT rebase

- ▶ Seguir el tutorial del libro para probar git rebase

<https://git-scm.com/book/es/v1/Ramificaciones-en-Git-Reorganizando-el-trabajo-realizado>

- ▶ Probar git rebase siguiendo el tutorial de ejemplo:

<https://code.tutsplus.com/es/tutorials/rewriting-history-with-git-rebase--cms-23191>

- ▶ Otro tutorial:

<https://www.solucionex.com/blog/git-merge-o-git-rebase>

# Ejercicio:

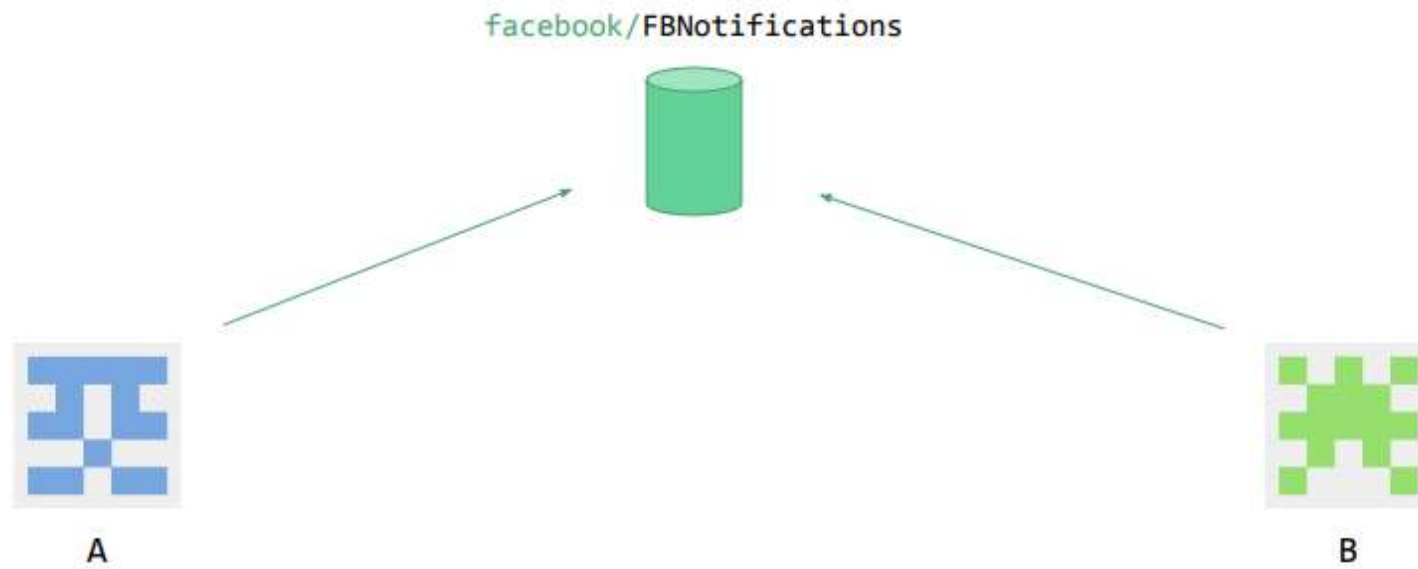
- ▶ Colaborando en un servidor remoto usando Github. En grupos de dos, vamos a probar cómo trabajar de manera colaborativa.
- ▶ Es necesario añadir un nuevo colaborador en un repositorio de GitHub
- ▶ Ver ejercicio 23
- ▶ [https://www.jesusamieiro.com/wp-content/uploads/2014/03/20141009\\_Git\\_Ejemplo\\_GPUL\\_UDC.pdf](https://www.jesusamieiro.com/wp-content/uploads/2014/03/20141009_Git_Ejemplo_GPUL_UDC.pdf)

The background features abstract green geometric shapes. On the left, a solid green trapezoid points upwards. On the right, a complex arrangement of overlapping translucent green triangles and polygons creates a layered, crystalline effect. The central text is positioned between these two main graphic elements.

# WORKFLOW EN GIT

# Problemas en nuestro workflow

- Ejemplo: Dos desarrolladores (A y B) están colaborando en un proyecto open-source de Facebook llamado FBNotifications.



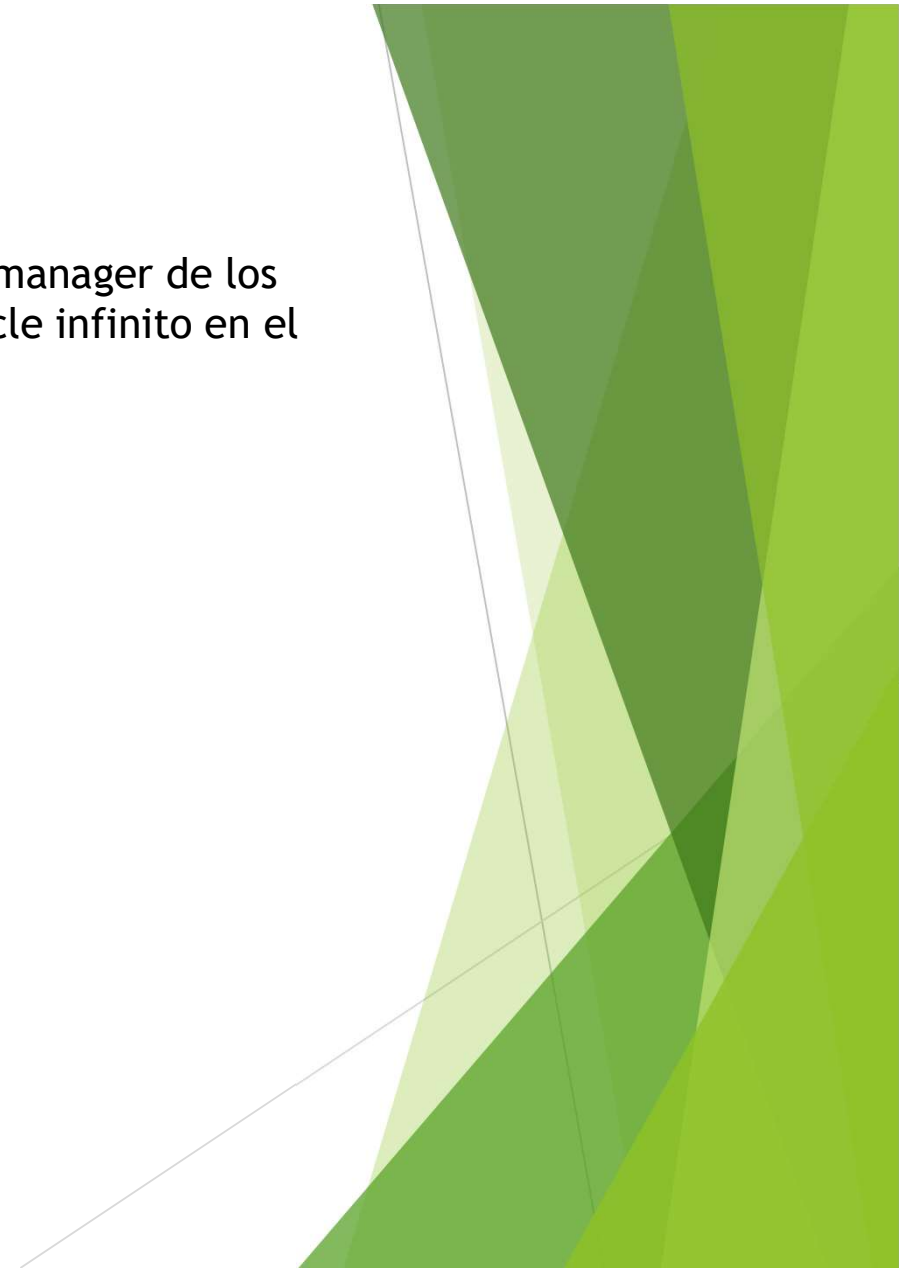
# Problemas en nuestro workflow

- A está muy seguro de que sus últimos cambios en local son correctos por lo que procede a añadirlos a un commit y hacer un push directamente al repositorio



# Problemas en nuestro workflow

- PROBLEMA: Al pasar los tests del commit del repositorio, el manager de los desarrolladores descubre que hay un bug causado por un bucle infinito en el código de A, por lo que le pide a B que lo solucione.





# Problemas en nuestro workflow

## ► POSIBLE SOLUCIÓN:



# Problemas en nuestro workflow

- ▶ SOLUCIÓN IMPUESTA POR EL MANAGER:
- ▶ A partir de ahora habrá una nueva forma de trabajar: Siempre y cuando se deba hacer cambios en el repositorio, se hará mediante **Pull Request**

# Trabajando con Workflow en Git

## ► Pull Request:

- Conjunto de cambios propuestos a un repositorio por un usuario y aceptado o rechazado por otro usuario colaborador de ese repositorio.
- Cada Pull Request tiene su propio hilo de discusión

## ► Fork

- Una copia personal en tu cuenta del repositorio de otro usuario
- Permite tener dos repositorios git idénticos pero con distinta URL
- Los forks nos permiten hacer cambios libremente de un proyecto sin afectar el original
- Esta copia permanecerá adjunta al original permitiendo remitir los cambios a éste mediante un mecanismo llamado pull-request
- También es posible mantener tu fork up-to-date actualizándose con los últimos cambios del original
- Tendremos dos repositorios independientes que pueden cada uno evolucionar de forma totalmente autónoma.

# Diferencia entre Fork y Clone

## ► CLONE

- Cuando hacemos un clon de un repositorio, bajas una copia del mismo a tu máquina
- Empiezas a trabajar, haces modificaciones y haces un push
- Cuando haces el push estás modificando el repositorio que has clonado

## ► FORK

- Cuando haces un fork de un repositorio, se crea un nuevo repositorio en tu cuenta de Github o Bitbucket, con una URL diferente (fork)
- Acto seguido tienes que hacer un clon de esa copia sobre la que empiezas a trabajar de forma que cuando haces push, estás modificando TU COPIA (fork)
- El repositorio original sigue intacto

# Fork - Uso

- ▶ Permitir a los desarrolladores contribuir a un proyecto de forma segura.
- ▶ Ejemplo de cosas que podemos hacer en un proyecto:
  1. Usuario1 hace un fork de mi repositorio, para lo que sólo necesito darle permiso de lectura.
  2. Usuario1 trabaja en SU COPIA (fork) del repositorio. Como es suya, puede hacer lo que quiera, la puede borrar, corromper, reescribir la historia del proyecto..., es su copia(fork) y no nos influye
  3. Cuando Usuario1 termina de programar y testear el parche, me avisa de que ya lo tiene y me dice “En la rama parche\_de\_usuario1 de MI COPIA (fork), tienes el parche que corrige el Bug XXXX”
  4. Voy a su repositorio, miro lo que ha hecho y si está bien lo incorporo (merge) a mi repositorio, que es el original

# Fork - Uso

► Las ventajas que tiene utilizar Fork:

1. Usuario1 trabaja con SU COPIA. En ningún momento le tengo que dar acceso al repositorio central.
2. El proceso de incorporación de los cambios de Usuario1 es muy sencillo. Si no hay conflictos en los ficheros puede que sea tan simple como ejecutar un par de comandos git.
3. Usuario1 tiene muy fácil contribuir, no le cuesta esfuerzo.
4. Puedo gestionar muy fácilmente las contribuciones de muchas personas

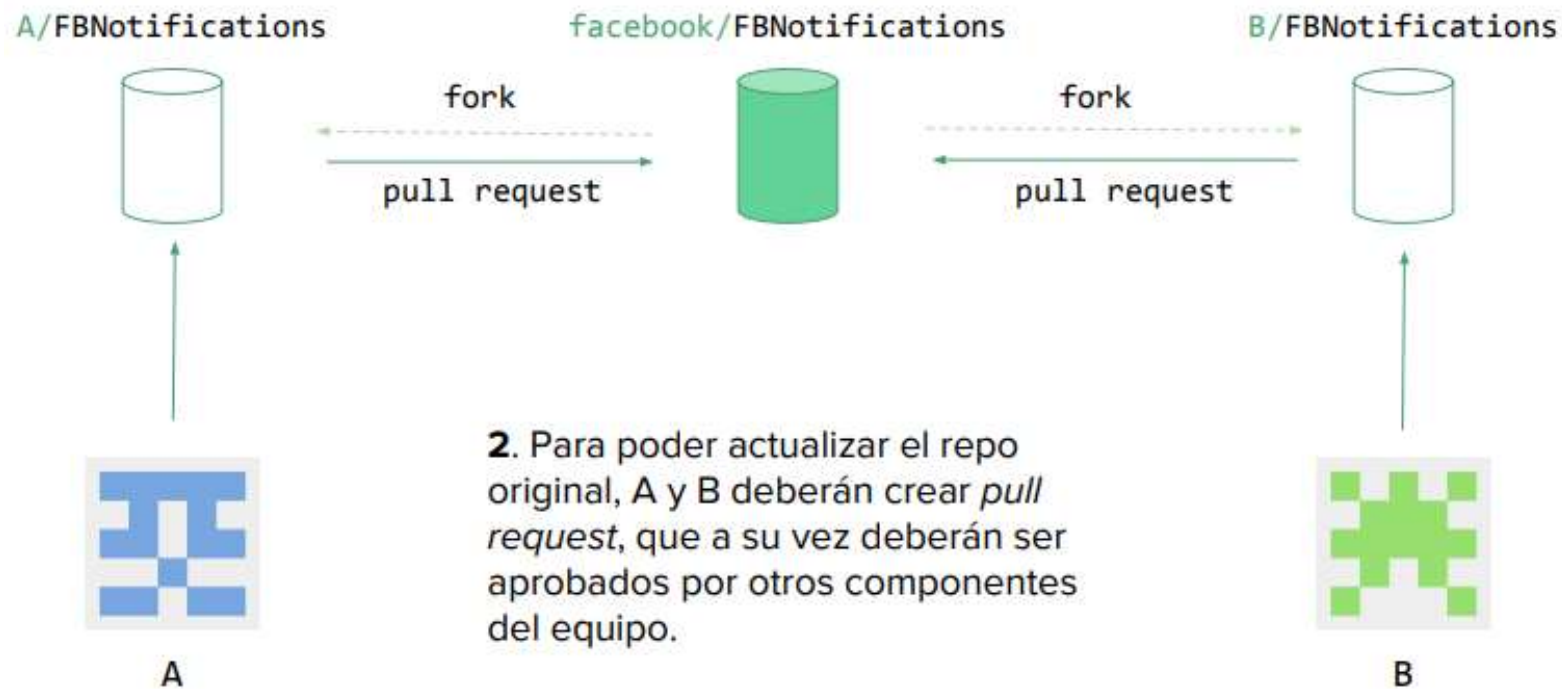
# Trabajando con Workflow en Git

- Para el ejemplo anterior, el manager podría haber usado una combinación entre Fork y Pull-Request para evitar problemas de código erróneo.
- Primero, los usuarios hacen un “fork” del repositorio



# Trabajando con Workflow en Git

- Los usuarios realizan sus modificaciones en local. Cuando han terminado crean un “pull request” y lo envían al repositorio. Sus cambios deben ser aprobados por el administrador





# Trabajando con Workflow en Git

- ▶ Ejemplo: trabajando con Fork y Git Pull Request

1. Crear forks

<http://aprendegit.com/fork-de-repositorios-para-que-sirve/>

2. Trabajar con Pull Request

<http://aprendegit.com/que-es-un-pull-request/>

- ▶ Trabaja con Forks - múltiples repositorios remotos

<http://aprendegit.com/mantener-tu-fork-al-dia/>

Otro ejercicio sencillo:

- ▶ <https://github.com/hcs/bootcamp-git/wiki/Exercise-Making-a-Pull-Request>

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern and dynamic visual effect.

# EJERCICIO GIT - WORKFLOW

# Ejercicio - Workflow en Git

- ▶ Trabajando en equipo. Formar equipos de 3 personas (A,B,C)
  1. A crea un repositorio en GitHub (por ejemplo, “pildora-git”)
  2. A añade como colaboradores a B y C
  3. Todos hacen un clone del repositorio remoto (\$ git clone ...)
  4. B crea un archivo index.html en el repositorio y hace add/commit/push al repositorio remoto. Usar el contenido del archivo alojado en github y hacer un push:  
  
<https://raw.githubusercontent.com/cbeldacap/píldora-git/master/ejemplo/index.html>
  5. C edita la línea 5 del archivo (<h1>) cambiando el contenido.
  6. A edita la misma línea con contenido distinto al de C
  7. A hace git add . / git commit commit -m “...”/ git push al repo.

## Ejercicio - Workflow en Git

8. C hace add/commit/push al repo y encuentra que otro usuario (A) ha actualizado el conflicto en el push
9. Cuando C hace un pull encuentra que hay conflicto en una de las líneas de código (Ayuda). ¿Qué ocurre en nuestro código en estos casos?

```
<<<<<< HEAD --Tu commit-- ===== --Último  
commit del repo-- >>>>>>  
bf454eff1b2ea242ea0570389bc75c1ade6b7fa0
```

10. C hace add/commit/push de nuevo generando una versión definitiva
11. C crea la nueva rama “add-button”. (\$ git checkout -b add-button)
12. A y B hacen pull de los últimos cambios y, después de hacer check-out a la nueva rama, B añade un botón al <body> del archivo y sube los cambios al repo.
13. C hace pull de los cambios y en la misma rama add-button añade un segundo botón por debajo del que había añadido B.

## Ejercicio - Workflow en Git

14. Finalmente, A (después de tener totalmente actualizado su repositorio local), hace un pull request desde la rama add-button a la rama master (éste deberá ser aprobado). Ahora las dos ramas del repositorio deberían ser iguales.
15. Todos hacen un pull con los últimos cambios.
16. B crea un issue en GitHub señalando que, después de todo el lío, el cliente ha decidido que sólo quiere un botón. Asigna como “encargado” a C.
17. C elimina el último botón creado por B y hace un commit (y un push) nombrando el issue que está solucionando: `$ git commit -m “issue #1 solved”`
18. A y B hacen un fork del repositorio (hacerlo desde GitHub)
19. Probar a hacer cambios (da igual el collaborator) en un fork y crear un pull request desde GitHub
20. Aceptar el pull request (da igual el collaborator)

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern and dynamic visual effect.

# GIT FLOW

# ¿Qué es Git-flow?

- ▶ Gitflow es un diseño de workflow de Git que fue publicado y popularizado por Vincent Driessen
- ▶ El workflow de Gitflow define un modelo de ramas estricto, diseñado en torno a la versiones del proyecto
- ▶ Esto proporciona un marco robusto para la gestión de proyectos más grandes
- ▶ Ideal para proyectos que tienen un ciclo de release planificado
- ▶ Este workflow no agrega ningún concepto o comando nuevo más allá de lo que se requiere para el workflow de la rama de características
- ▶ En cambio, asigna funciones muy específicas a diferentes ramas y define cómo y cuándo deberían interactuar.
- ▶ Además de las ramas de “features”, se usan ramas individuales para preparar, mantener y registrar releases.
- ▶ Por supuesto, también puede aprovechar todos los beneficios del workflow de la rama Feature: pull requests, experimentos aislados y colaboración más eficiente

# ¿Qué es Git-flow?

- ▶ Es una idea abstracta de un workflow de Git.
- ▶ Esto significa que dicta qué tipo de ramas configurar y cómo combinarlas
- ▶ git-flow es una herramienta de línea de comando real que tiene un proceso sencillo de instalación
- ▶ Disponible en múltiples sistemas operativos: OSX, Windows, Linux
- ▶ Git-flow es un “envoltorio” alrededor de Git
- ▶ El comando `git flow init` es una extensión del comando `git init` predeterminado y no cambia nada en su repositorio que no sea crear ramas por el usuario



# ¿Cómo funciona Git-flow?

- ▶ En lugar de trabajar con una única rama maestra, nuestro workflow utiliza dos ramas para registrar el historial del proyecto.
  - ▶ La rama master almacena el historial de versiones oficial
  - ▶ La rama develop sirve como rama de integración de las “features” añadidas
- ▶ Es conveniente etiquetar (tags) todas las confirmaciones en la rama master con un número de versión
- ▶ El primer paso es completar la rama master por defecto con una rama de desarrollo.

```
$ git branch develop
```

```
$ git push -u origin develop
```
- ▶ Esta rama contendrá el historial completo del proyecto, mientras que la master contendrá una versión abreviada. Otros desarrolladores deberían ahora clonar el repositorio central y crear una rama de seguimiento de develop

# ¿Cómo funciona Git-flow?

- La ejecución de `git flow init` en un repositorio existente creará la rama de `develop`:

```
$ git flow init
```

```
Initialized empty Git repository in ~/project/.git/
```

```
No branches exist yet. Base branches must be created now.
```

```
Branch name for production releases: [master]
```

```
Branch name for "next release" development: [develop]
```

```
How to name your supporting branch prefixes?
```

```
Feature branches? [feature/]
```

```
Release branches? [release/]
```

```
Hotfix branches? [hotfix/]
```

```
Support branches? [support/]
```

```
Version tag prefix? []
```

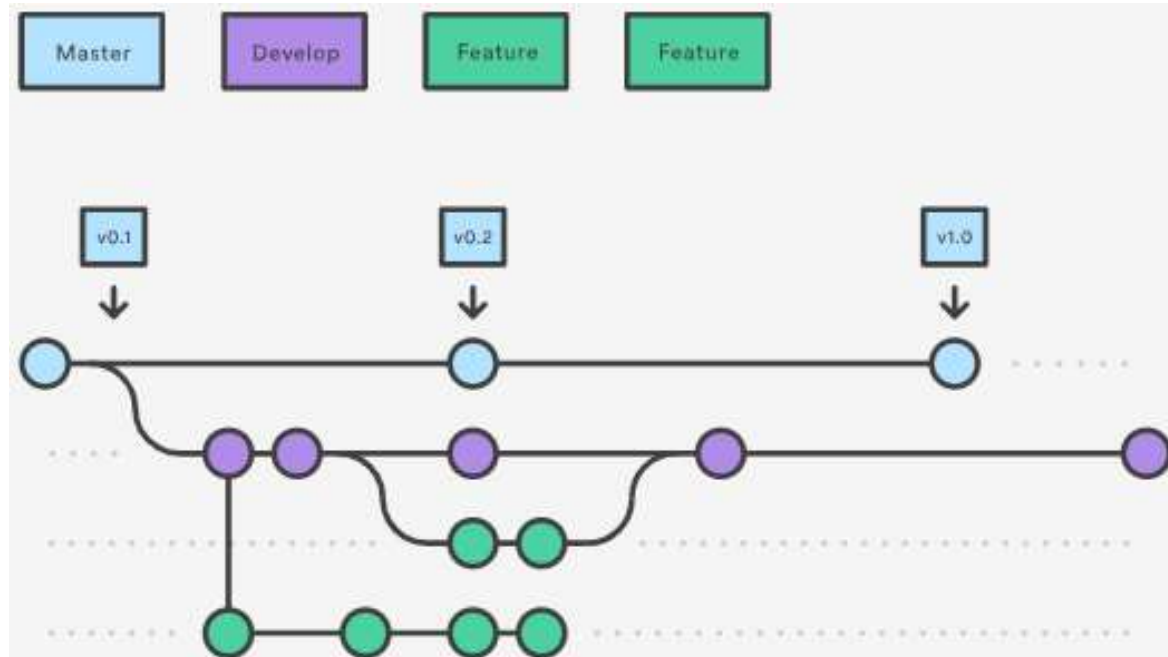
```
$ git branch
```

```
* develop
```

```
master
```

# Ramas Feature

- ▶ Cada nueva “feature” debe residir en su propia rama, que pueda ser enviada al repositorio central para que se pueda hacer backup/colaboración
- ▶ En lugar de ramificarse desde la rama master, las ramas Feature utilizan develop como su rama padre.
- ▶ Cuando se completa una feature, se mergea de nuevo con la rama develop. Las características nunca deben interactuar directamente con la rama master



# Ramas Feature

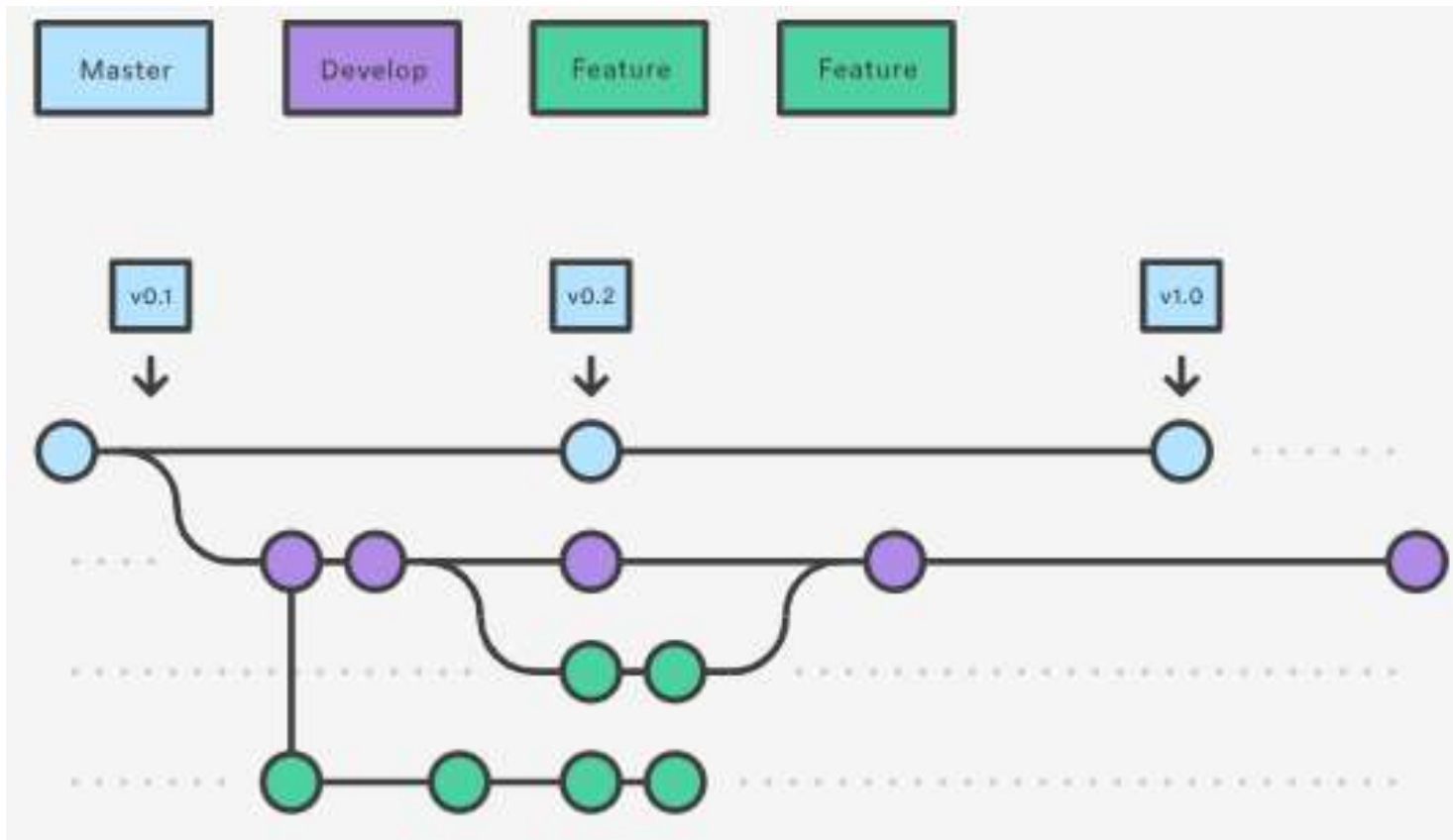
- ▶ Tener en cuenta que las ramas Feature combinadas con la rama develop, son a todos los efectos, el workflow de la rama Feature
- ▶ Las ramas Feature generalmente se crean en la última rama de Develop



# Ramas Feature

- ▶ Tener en cuenta que las ramas Feature combinadas con la rama develop, son a todos los efectos, el workflow de la rama Feature
- ▶ Las ramas Feature generalmente se crean en la última rama de Develop
- ▶ Para iniciar una rama Feature:
  - \$ git flow feature start nombre\_rama\_feature
  - ▶ Tras esto, se puede trabajar con Git como haríamos normalmente
- ▶ Para terminar una rama Feature:
  - \$ git flow feature finish nombre\_rama\_feature
- ▶ Sin Git Flow
  - ▶ Crear rama para la feature nueva
  - \$ git checkout develop
  - \$ git checkout -b nombre\_rama\_feature
  - ▶ Terminar con la rama
  - \$ git checkout develop
  - \$ git merge nombre\_rama\_feature

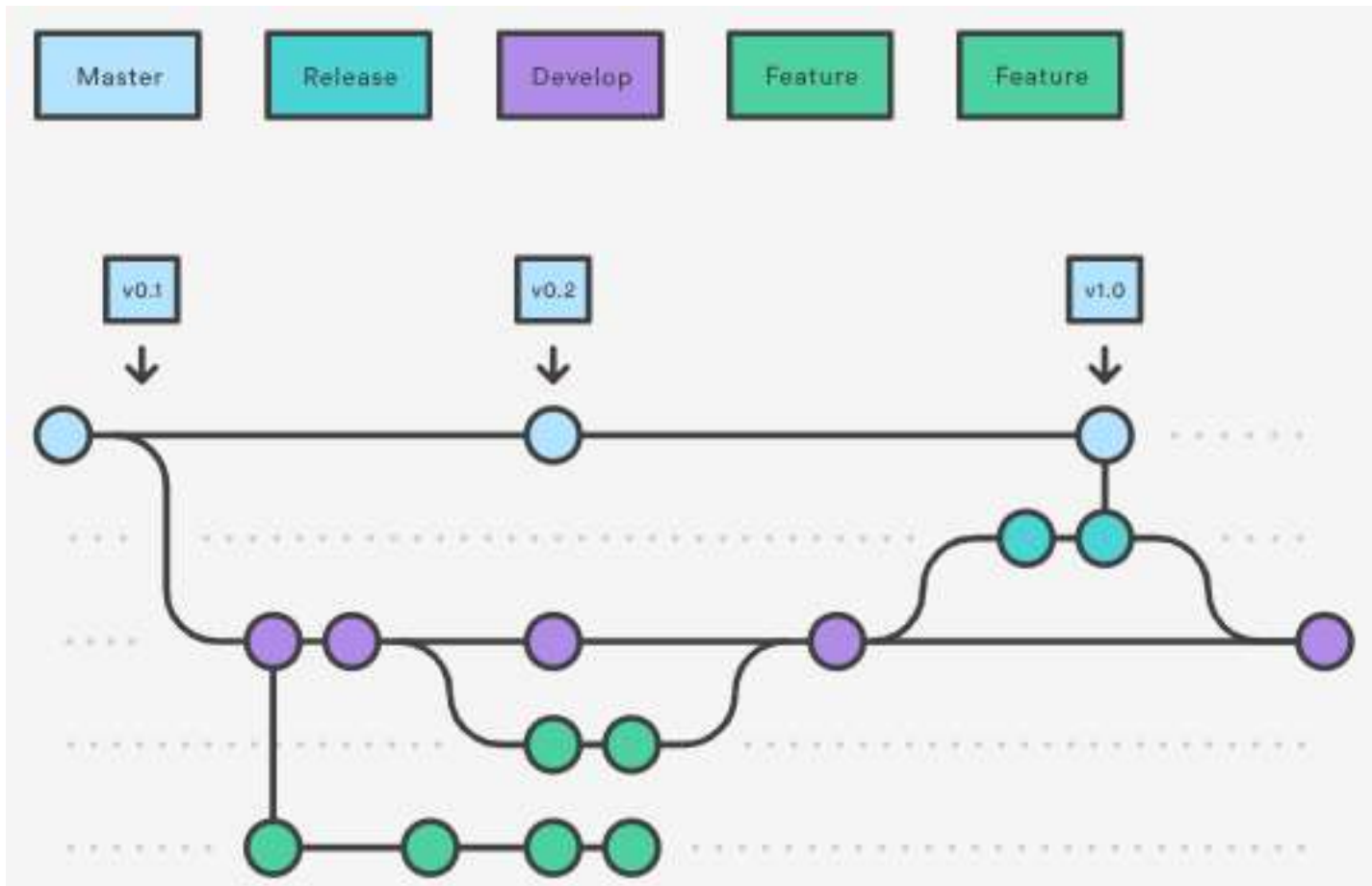
# Ramas Feature



# Ramas Release

- ▶ Una vez que la rama develop ha completado suficientes “features” para una reléase(se acerca la fecha de release), se bifurca una rama Release desde la rama develop
- ▶ La creación de esta rama inicia el siguiente ciclo de Release, por lo que no se pueden agregar nuevas features después de este punto
- ▶ Sólo se admite introducir en esta rama correcciones de errores, generación de documentación y otras tareas orientadas a la versión
- ▶ Una vez que está lista la rama Release, se fusiona con la master y se etiqueta con un número de versión.
- ▶ Además, debería fusionarse nuevamente con develop, que puede haber progresado desde que se inició la release

# Ramas Release





# Ramas Release

- ▶ El uso de una rama dedicada para preparar releases permite que un equipo pueda pulir la release actual mientras que otro equipo continúa trabajando en las features para la próxima versión
- ▶ También crea fases de desarrollo bien definidas (por ejemplo, es fácil decir: "Esta semana nos estamos preparando para la release 4.0" y verla realmente en la estructura del repositorio).
- ▶ Hacer ramas Release es otra operación directa de ramificación. Al igual que las ramas Features, las ramas Release se basan en la rama Develop
- ▶ Crear rama Release con Git Flow:  

```
$ git flow release start 0.1.0
```

  
Switched to a new branch 'release/0.1.0'
- ▶ Crear rama Release sin Git Flow:  

```
$ git checkout develop
```

```
$ git checkout -b release/0.1.0
```

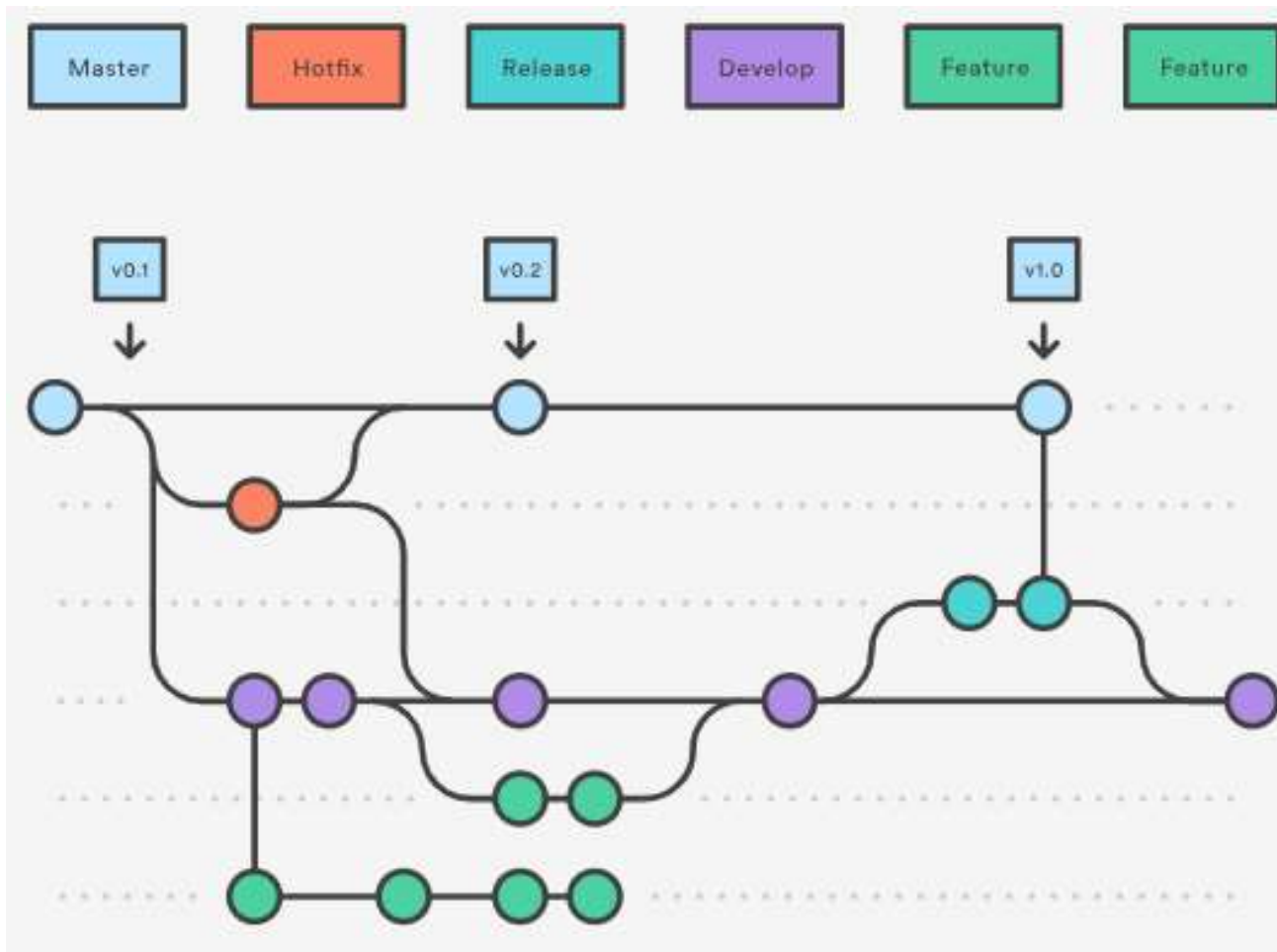
# Ramas Release

- ▶ Una vez que la reléase esté listo para lanzarse, se hará un merge en las ramas master y Develop, luego se eliminará la rama de Release
- ▶ Es importante mergerar nuevamente en rama Develop porque las actualizaciones críticas se pueden haber agregado a la rama Release y deben ser accesibles a las nuevas features
- ▶ Si su organización hace hincapié en la revisión del código, este sería un lugar ideal para un pull request
- ▶ Para finalizar una rama Release con Git Flow:
  - \$ git checkout master
  - \$ git checkout merge release/0.1.0
  - \$ git flow release finish '0.1.0'
- ▶ Para finalizar una rama Release sin Git Flow:
  - \$ git checkout develop
  - \$ git merge release/0.1.0

# Ramas Hotfix

- ▶ Las ramas de mantenimiento o "hotfix" se utilizan para parchear rápidamente releases de producción.
- ▶ Las ramas Hotfix se parecen mucho a las ramas Release y ramas Feature, excepto que se basan en la rama Master en lugar de Develop.
- ▶ Esta es la única rama que debe partir directamente de Master.
- ▶ Tan pronto como se arregle el problema, se debe mergear en rama Master Desarrollo (o la rama de reléase actual), y la rama master debe etiquetarse con un número de versión actualizado
- ▶ Tener una línea dedicada de desarrollo para corregir bugs permite al equipo solucionar problemas sin interrumpir el resto del workflow o esperar el siguiente ciclo de reléase
- ▶ Se puede pensar en ramas de mantenimiento como ramas Release ad hoc que trabajan directamente con rama Master.

# Ramas Hotfix



# Ramas Hotfix

## ► Trabajar con Hotfix con Git Flow:

```
$ git flow hotfix start hotfix_branch  
...  
$ git checkout master  
$ git merge hotfix_branch  
$ git checkout develop  
$ git merge hotfix_branch  
$ git branch -D hotfix_branch  
$ $ git flow hotfix finish hotfix_branch
```

## ► Trabajar con Hotfix sin Git Flow:

```
$ git checkout master  
$ git checkout -b hotfix_branch
```



# Workflow con GitFlow

1. Se crea una rama Develop desde Master
2. Se crea una rama Release desde Develop
3. Las ramas Features se crean a partir de Develop
4. Cuando se completa una Feature, se mergea en la rama de Develop
5. Cuando se completa la rama de Release, se mergea en Develop
6. Si se detecta un problema en Master, se crea una rama Hotfix(revisión) desde la rama Master
7. Una vez que se completa el Hotfix, se mergea con Master y Develop



# Ejemplo con Git Flow

- ▶ Ejemplo que demuestra workflow con rama Feature. Suponiendo que tenemos un repositorio configurado con la rama Master:
- ▶ Trabajado sobre una nueva feature:

```
git checkout master
git checkout -b develop
git checkout -b feature_branch
# Aquí va el trabajo sobre la rama feature
git checkout develop
git merge feature_branch
git checkout master
git merge develop
git branch -d feature_branch
```

# Ejemplo con Git Flow

- ▶ Ejemplo que demuestra workflow con rama Feature. Suponiendo que tenemos un repositorio configurado con la rama Master:
- ▶ Trabajado con Hotfix:

```
git checkout master
```

```
git checkout -b hotfix_branch
```

```
# work is done commits are added to the hotfix_branch
```

```
git checkout develop
```

```
git merge hotfix_branch
```

```
git checkout master
```

```
git merge hotfix_branch
```



# Ejercicio con Git Flow

- ▶ Probar los siguientes ejemplos referentes a Git Flow:
- ▶ <https://blog.axosoft.com/gitflow/>
- ▶ <http://aprendegit.com/git-flow-release-branches/>



# GITKRAKEN

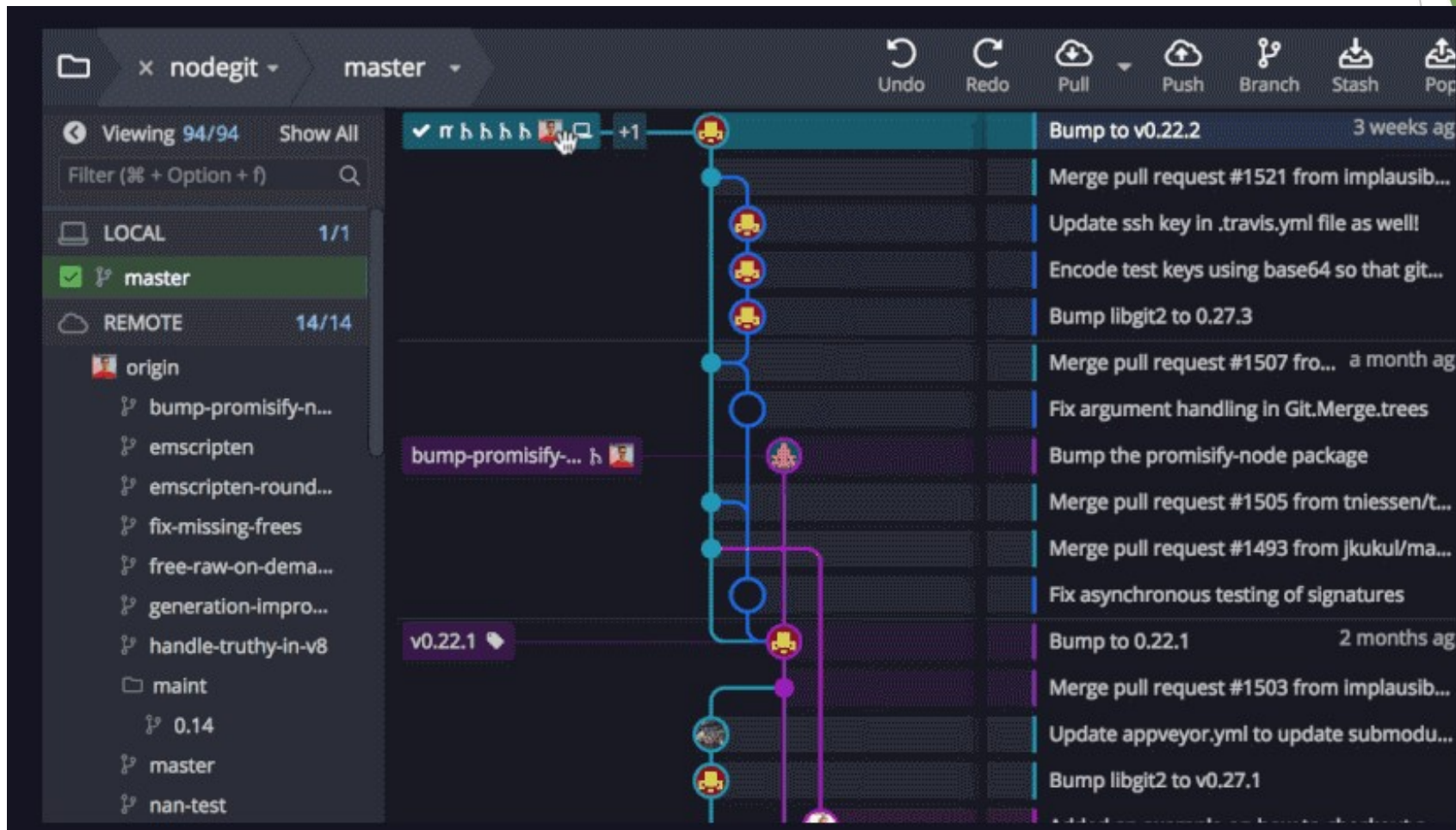


# Gikraken - intro

- ▶ Famosa GUI para trabajar con GIT
- ▶ Potente y elegante interfaz gráfica para git desarrollada con Electron.
- ▶ De forma muy sencilla podemos llevar el completo seguimiento de nuestros repositorios, ver ramas, tags, crear nuevos, todo el historial de nuestro trabajo, commits etcétera.
- ▶ GitKraken es multiplataforma, por lo tanto podemos utilizarlo en windows, linux y mac os.
- ▶ Este sistema para el control de versiones con git tienes dos planes, uno gratuito para siempre y otro de pago el cual ofrece mucha más funcionalidad, y no es excesivamente caro si te es de ayuda
- ▶ Con GitKraken puedes tener controlados todos tus proyectos con una interfaz muy elegante y sencilla
- ▶ Integración con Glthub, GitLab, Bitbucket



# Gikraken - intro



# Instalar GitKraken

- ▶ Vamos a acceder a la web de GitKraken para descargarlo e instalarlo.
- ▶ A partir de ahora podremos gestionar nuestros proyectos de manera sencilla con su interfaz gráfica
- ▶ <https://www.gitkraken.com/>



The background features abstract green geometric shapes. On the left, a solid green trapezoid points towards the center. On the right, a complex arrangement of overlapping, semi-transparent green triangles and polygons creates a layered, dynamic effect. The central text is positioned between these two main graphic elements.

# TRABAJANDO CON REPOSITORIOS GIT

# Repositorios - intro

- ▶ Los repositorios de código son herramienta de gran ayuda para los desarrolladores, especialmente cuando los equipos de trabajo son grandes.
- ▶ Cuando se trata de elegir un repositorio de código adecuado, hay una gran diversidad de donde elegir.
- ▶ Vamos a revisar tres de los más populares:
  - ▶ GitHub
  - ▶ GitLab
  - ▶ BitBucket

# Tipos de repositorios

- ▶ Como ya sabemos, un sistema de control de versiones puede ser de tres tipos:
  - ▶ Local: Los desarrolladores se encuentran en el mismo sistema de archivos.
  - ▶ Centralizado, Hay una copia del proyecto en un servidor central y los integrantes del equipo realizan una actualización de estos archivos de acuerdo a los cambios que realizan
  - ▶ Distribuido: Se trabaja en un repositorio local y los cambios se actualizan entre repositorios

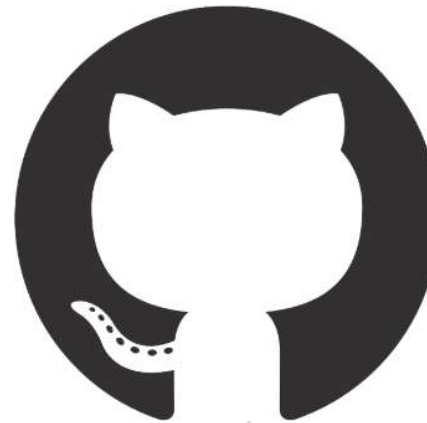


# Características de un buen repositorio

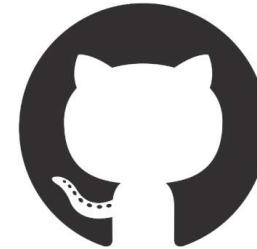
- ▶ **Pull request:** Cuando los usuarios realizan cambios en el código fuente y lo envían al repositorio, se notifica a sus colaboradores para que lo revisen
- ▶ **Revisión de código web**
- ▶ **Edición:** Si un repositorio tiene la posibilidad de sugerir una edición colaborativa en tiempo real, agrega mucho a la calidad del repositorio
- ▶ **Seguimiento de errores:** Todos los proyectos tienen errores. Mejor si un repositorio permite rastrear y resolver errores de forma colaborativa
- ▶ **Autenticación two-factor** para garantizar protección de cuentas de usuario
- ▶ La capacidad de crear **bifurcaciones o clones** del repositorio
- ▶ **Compartir segmentos de código** o archivos con cualquier persona
- ▶ Integración con **servicios de terceros**
- ▶ **Importación de repositorios:** Si los usuarios cambian de un servicio a otro, es una buena idea que los repositorios permitan importar proyectos existentes.
- ▶ **Licencia de código abierto.** En ocasiones, las organizaciones necesitan organizar un repositorio interno en su propio servidor en lugar de utilizar los recursos web públicos existentes. El único de código abierto: GitLab.

# GitHub

- ▶ Opción web más popular para almacenar repositorios git.
- ▶ Diseñado para permitir a los usuarios crear fácilmente sistemas de control de versiones basados en Git.
- ▶ ¿Por que es tan popular?
  - ▶ Admite fusiones y divisiones de versiones uniformes con la ayuda de herramientas de visualización y herramientas para la navegación a través del historial de desarrollo no lineal.
- ▶ GitHub alberga más de 50 millones de proyectos de código abierto
- ▶ <https://github.com/>
- ▶ <https://www.youtube.com/watch?v=uYiAserqw2M>
- ▶ <https://www.youtube.com/watch?v=SCNWWfXpCw0>



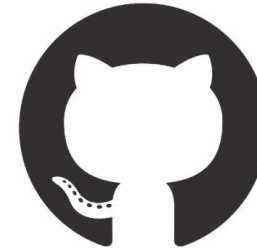
# GitHub



## VENTAJAS:

- ▶ **Seguimiento de errores:** Permite de forma colaborativa mejorar la calidad del código al mantener registro de los errores de software detectados en el proyecto.
- ▶ **Búsqueda rápida:** El repositorio proporciona una estructuración conveniente de proyectos que permite una búsqueda y clasificación eficiente. La indexación en la red permite encontrar cualquier cadena de código pública
- ▶ **Comunidad:** más de 20 millones de usuarios. Esto es una gran fuente de experiencia y habilidades compartidas.
- ▶ **Compartir:** El código fuente del proyecto no sólo se puede copiar con Git, también se puede descargar como archivo.
- ▶ **Trabajo conjunto:** GitHub brinda funciones eficientes para la administración de equipos.
- ▶ **Compatibilidad:** Los proyectos con el código en GitHub se pueden personalizar fácilmente a cualquier servicio host en la nube.
- ▶ Admite importación con **Git, SVN, TFS.**

# GitHub



## DESVENTAJAS:

- ▶ No es completamente gratuito. Para acceder a todas las funciones de GitHub, se debe actualizar a un usuario Premium.
- ▶ Limitación de tamaño. Los archivos no pueden ser mayores a 100 MB mientras que el repositorio puede alojar 1 GB de información.

# GitLab

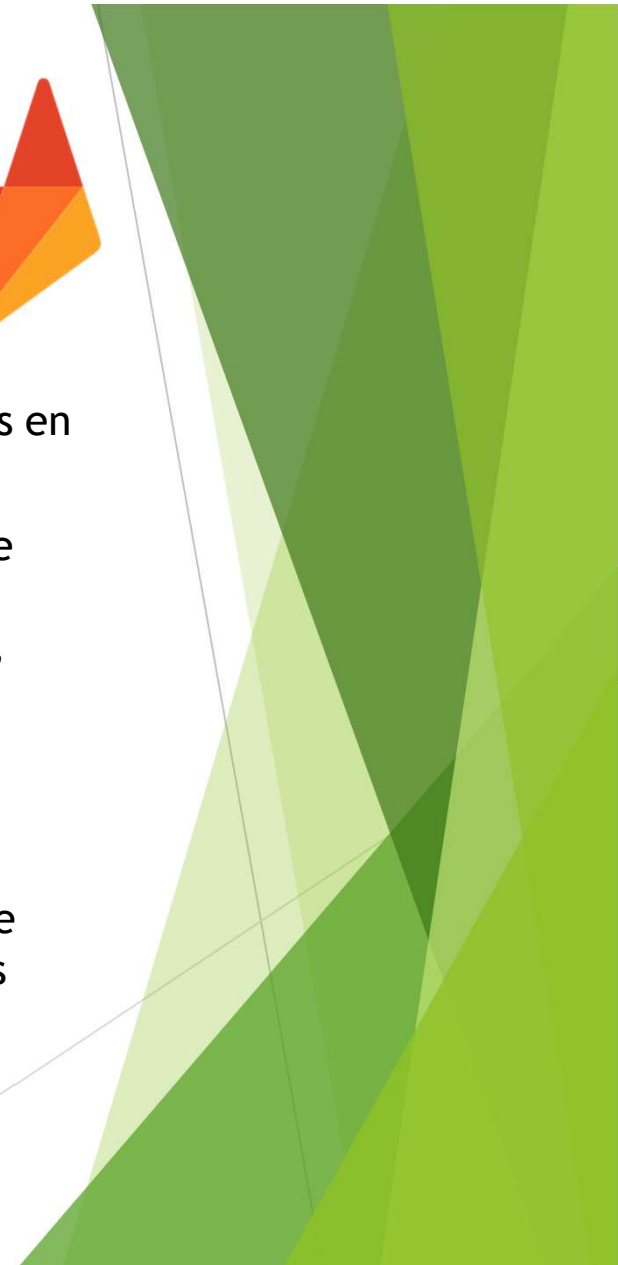
- ▶ También está desarrollado en la base del control de versiones de Git
- ▶ Funcionalidad de GitLab similar a su principal competidor, GitHub, aunque existen algunas peculiaridades importantes
- ▶ GitLab tiene versiones diferentes:
  - ▶ GitLab SAAS, adecuada para empresas
  - ▶ GitLab Community Edition, solución individual para usuarios
- ▶ <https://about.gitlab.com/>
- ▶ <https://www.youtube.com/watch?v=LqLHtTfKhUk>



# GitLab

## VENTAJAS:

- ▶ **Gratuito.**
  - ▶ Los usuarios pueden tener un número ilimitado de repositorios privados en la versión comunitaria.
  - ▶ Los usuarios tendrán que pagar si requieren la versión empresarial, que añade funcionalidades especiales como: mejora de la interacción con herramientas en línea, flujo de trabajo y administración de servidores, entre otras
- ▶ GitLab opera bajo **licencia de código abierto**
- ▶ Seguimiento de errores y edición de código basado en la web
- ▶ Integración con LDAP que permite localizar y acceder a diversos recursos de internet. GitLab EE soporta varios servicios LDAP y sincronización de grupos
- ▶ Soporta la importación de Git



# GitLab

## DESVENTAJAS:

- ▶ Interfaz algo lenta
- ▶ Frecuentes problemas técnicos con los repositorios



# BitBucket

- ▶ El servicio muy similar a GitHub y refleja la mayoría de sus características con ligeras diferencias.
- ▶ Mejor orientado a equipos de desarrollo profesional. Proporciona grandes beneficios como:
- ▶ Repositorios privados gratuitos
- ▶ Integración con Jira
- ▶ Revisión de código avanzado
- ▶ CI/CD (Integración Continua/Despliegue continuo) integrado
- ▶ Con el crecimiento del equipo, Bitbucket ofrece condiciones de precios más adecuadas comparadas con GitHub y GitLab
- ▶ Proporciona un modelo de implementación flexible para equipos
- ▶ <https://bitbucket.org/>
- ▶ <https://www.youtube.com/watch?v=BD8xfCILcBs>





# BitBucket



## VENTAJAS:

- ▶ **Repositorios privados para equipos pequeños.** Equipos pequeños, hasta 5 integrantes, pueden obtener un número ilimitado de repositorios y 500 minutos de compilación. Los precios son más bajos que con GitHub
- ▶ **Búsqueda consistente de código.** Bitbucket utiliza búsqueda semántica que analiza la sintaxis del código, asegurando que las definiciones que coincidan con su término de búsqueda tengan prioridad sobre usos y nombres de las variables
- ▶ Bitbucket encontrará funciones como lista blanca de IPs, verificación en dos pasos para dar más control a los administradores sobre quién puede ver/enviar/clonar un repo de código privado.
- ▶ Al ser propiedad de Atlassian, **Bitbucket viene con Trello(lista de problemas) y Jira(gestión de proyectos).** Se integran en cada etapa del desarrollo. Jira actualiza automáticamente la información sobre el problema detectado
- ▶ Importación de proyectos Git existentes desde Excel, Github, entre otros
- ▶ Condiciones especiales para estudiantes y profesores
- ▶ Compatible con importaciones de Git, CodePlex, Google Code, SVN

# BitBucket

## DESVENTAJAS:

- ▶ No es de código abierto, pero admite proyectos de código abierto



# ¿Qué repositorio escoger?

- ▶ Depende de las necesidades del proyecto en el que vayamos a trabajar
- ▶ Cada repositorio tiene sus fortalezas y debilidades. Los tres: GitHub, GitLab y Bitbucket tiene una gran cantidad de usuarios por las necesidades que cubren
- ▶ De los tres servicios de administración de repositorios, solo GitLab es de código abierto
- ▶ El código fuente de GitLab Community Edition está disponible en su sitio web, la versión corporativa es cerrada.
- ▶ GitHub, que alberga el mayor número de proyectos de código abierto, no es de código abierto
- ▶ Bitbucket no es de código abierto, pero cuando compra una versión independiente, se proporciona el código fuente completo con las opciones de configuración del producto

# ¿GitLab, la alternativa a GitHub?



- ▶ En verano de 2018, Microsoft compró GitHub. Esto trajo mucha polémica sobre los usuarios por posibles cambios que esto conlleve
- ▶ A raíz de esto, el número de usuarios en GitLab está creciendo
- ▶ Google está invitiendo para potenciar GitLab y sacar beneficio de esta situación
- ▶ El motivo de fondo: el rechazo a Microsoft o el temor a que se pierda el espíritu de código abierto
- ▶ Microsoft intenta calmar a los usuarios prometiendo que GitHub se mantendrá independiente y abierto como hasta ahora
- ▶ El núcleo de GitHub no cambiará, pero Microsoft sí extenderá los servicios empresariales de la plataforma
- ▶ <https://goo.gl/n54eWd>
- ▶ <https://goo.gl/NJr8w2>
- ▶ <https://goo.gl/DQrxCA>

The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern and dynamic visual effect.

# INTEGRACIÓN CONTINUA GIT

# Integración Continua

- ▶ Los procesos de desarrollo software están en constante evolución y la integración continua se ha convertido en un elemento clave a la hora de mejorar y optimizar el ciclo de desarrollo

- ▶ Integración continua:

*“Práctica de desarrollo software donde los miembros del equipo integran su trabajo frecuentemente, al menos una vez al día. Cada integración se verifica con un “build” automático (que incluye la ejecución de pruebas) para detectar errores de integración tan pronto como sea posible.”*



# Integración Continua

- La integración continua se refiere a la fase de creación y pruebas de unidad del proceso de publicación de software. Cada revisión enviada activa automáticamente la creación y las pruebas.



# Integración Continua

## Beneficios de la integración continua



### Mejore la productividad de desarrollo

La integración continua mejora la productividad del equipo al liberar a los desarrolladores de las tareas manuales y fomentar comportamientos que ayudan a reducir la cantidad de errores y bugs enviados a los clientes.



### Encuentre y arregle los errores con mayor rapidez

Gracias a la realización de pruebas más frecuentes, el equipo puede descubrir y arreglar los errores antes de que se conviertan en problemas más graves.



### Entregue las actualizaciones con mayor rapidez

La integración continua le permite a su equipo entregar actualizaciones a los clientes con mayor rapidez y frecuencia.



# Integración Continua. Git + Jenkins

- ▶ Jenkins es un servidor diseñado para la integración continua, gratuito y de código abierto (open source), también se ha convertido en el software mas utilizado para esta tarea. Hecho en Java
- ▶ Características que encontraremos en Jenkins son:
  - ▶ Configurar la herramienta para que ejecute las políticas o reglas de negocios de calidad y ver sus resultados
  - ▶ Visualizar resultado de todas la pruebas
  - ▶ Visualizar o generar la documentación del proyecto
  - ▶ Pasar versiones del código de QA a un ambiente de pre-producción para sus ultimas pruebas o bien a producción directamente
- ▶ Importante tener un repositorio de control de versiones, pueden ser git, svn, mercurial,... Ahí se encuentra lo necesario para realizar un build. Actúa a través de cualquier cambio de los repositorios.
- ▶ Empresas que usan Jenkins: la NASA, Netflix, Facebook, Yahoo!, LinkedIn
- ▶ Podemos integrarlo con: GitHub, GitLab, BitBucket
- ▶ [https://www.youtube.com/watch?v=5YEgJuEUL\\_k](https://www.youtube.com/watch?v=5YEgJuEUL_k)



## Jenkins