



DECODED

CODE

# FULL STACK DEVELOPER BOOTCAMP

Desarrollamos  
*{ talento }*

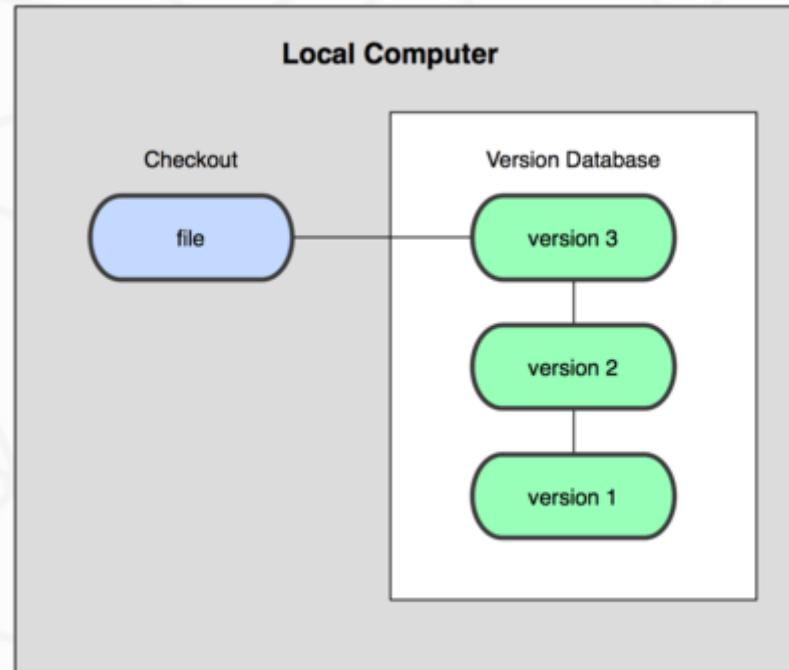
# **GIT**

## Control de versiones

El control de versiones es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedes recuperar versiones específicas más adelante.

## Sistema de control de versiones locales

Un método de control de versiones usado por mucha gente es copiar los archivos a otro directorio (quizás indicando la fecha y hora en que lo hicieron, si son avisados). Éste enfoque es muy común porque es muy simple, pero también tremadamente propenso a errores. Es fácil olvidar en qué directorio te encuentras, y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías.



# Historia de Git

## 2002

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**.

# 2005

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

## Objetivos de 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)

# Instalación

[Web oficial](#)

# Cómo funciona

## 3 estados

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

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

# Flujo de trabajo

1. Modificas una serie de archivos en tu directorio de trabajo.
2. Preparas los archivos, añadiéndolos a tu área de preparación. (git add)
3. Confímas 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. (git commit)

# Configuración

# Tu identidad

```
$ git config --global user.name "John Doe"  
$ git config --global user.email johndoe@example.com
```

# Tu editor

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

# Comprobando tu configuración

Si quieres comprobar tu configuración, puedes usar el comando git config --list para listar todas las propiedades que Git ha configurado:

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
...
```

# Ayuda

Si alguna vez necesitas ayuda usando Git, hay tres formas de ver la página del manual (manpage) para cualquier comando de Git:

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

# Uso en local

# Inicializando un directorio existente

Si estás empezando el seguimiento en Git de un proyecto existente, necesitas ir al directorio del proyecto y escribir:

```
$ git init
```

Esto crea un nuevo subdirectorio llamado `.git` que contiene todos los archivos necesarios del repositorio –un esqueleto de un repositorio Git. Todavía no hay nada en tu proyecto que esté bajo seguimiento. (Véase el Capítulo 9 para obtener más información sobre qué archivos están contenidos en el directorio `.git` que acabas de crear.)

# Añadiendo archivos

Si deseas empezar a controlar versiones de archivos existentes (a diferencia de un directorio vacío), probablemente deberías comenzar el seguimiento de esos archivos y hacer una confirmación inicial. Puedes conseguirlo con unos pocos comandos git add para especificar qué archivos quieres controlar, seguidos de un commit para confirmar los cambios:

```
$ git add *.c  
$ git add README  
$ git commit -m 'versión inicial del proyecto'
```

# Clonando un repositorio existente

Puedes clonar un repositorio con git clone [url]. Por ejemplo, si quieres clonar la librería Ruby llamada Grit, harías algo así:

```
$ git clone git://github.com/schacon/grit.git
```

Esto crea un directorio llamado "grit", 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 te metes en el nuevo directorio grit, verás que están los archivos del proyecto, listos para ser utilizados.

# Protocolos de transferencia

Git te permite usar distintos protocolos de transferencia.

El ejemplo anterior usa el protocolo git://, pero también te puedes encontrar con http(s):// o usuario@servidor:/ruta.git, que utiliza el protocolo de transferencia SSH.

# Comprobando el estado de tus archivos

Tu principal herramienta para determinar qué archivos están en qué estado es el comando git status. Si ejecutas este comando justo después de clonar un repositorio, deberías ver algo así:

```
$ git status  
# On branch master  
nothing to commit, working directory clean
```

## Viendo tus cambios

Si quieres saber exactamente lo que ha cambiado, no sólo qué archivos fueron modificados— puedes usar el comando `git diff`.

`git diff` te muestra exactamente las líneas añadidas y eliminadas —el parche, como si dijésemos.

## Confirmando tus cambios

Cuando estás listo para confirmar tus cambios. La forma más fácil de confirmar es escribiendo git commit:

```
$ git commit  
# o  
git commit -m "objetivo conseguido"
```

## Saltándote el área de preparación

Aunque puede ser extremadamente útil para elaborar confirmaciones exactamente a tu gusto, el área de preparación es en ocasiones demasiado compleja para las necesidades de tu flujo de trabajo. Si quieras saltarte el área de preparación, Git proporciona un atajo. Pasar la opción `-a` al comando `git commit` hace que Git prepare todo archivo que estuviese en seguimiento antes de la confirmación, permitiéndote obviar toda la parte de `git add`:

```
git commit -am "mensaje"
```

# Viendo el histórico de commits

# Git log

Para comprobar los cambios de nuestro repositorio utilizamos **git log**

```
$ git log
```

Cuando ejecutes git log sobre este proyecto, deberías ver una salida similar a esta:

```
$ git log  
commit ca82a6dff817ec66f44342007202690a93763949  
Author: Scott Chacon <schacon@gee-mail.com>  
Date:   Mon Mar 17 21:52:11 2008 -0700
```

changed the version [number](#)

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
Author: Scott Chacon <schacon@gee-mail.com>  
Date:   Sat Mar 15 16:40:33 2008 -0700
```

removed unnecessary test code

# La siguiente lista de opciones interesantes.

- `-p` Muestra el mensaje introducido en cada confirmación.
- `--stat` Muestra estadísticas sobre los archivos modificados en cada confirmación.
- `--shortstat` Muestra solamente la línea de resumen de la opción.
- `--name-only` Muestra la lista de archivos afectados.
- `--name-status` Muestra la lista de archivos afectados, indicando además si fueron añadidos, modificados o eliminados
- `--abbrev-commit` Muestra solamente los primeros caracteres de la suma SHA-1
- `--relative-date` Muestra la fecha en formato relativos, en lugar de formato completo.
- `--graph` Muestra una gráfico ASCII con la historia de ramificaciones.
- `--pretty` Muestra las confirmaciones usando un formato alternativo.

# La siguientes opciones sirven para limitar la salida

- -(n) Muestra el número de las últimas confirmaciones.
- --since, --after Muestra aquellas confirmaciones hechas después de la fecha especificada.
- --until, --before Muestras las confirmaciones realizadas antes de la fecha específica.
- --author Muestra sólo aquellas confirmaciones cuyo author coincide con la cadena especificada.
- --committer Muestra solo aquellas confirmaciones cuyo confirmador coincide con la cadena especificada.
- --S Muestra sólo aquellas confirmaciones que añandan o eliminan código que corresponda con la cadena especificada.

# Repositorios remotos

# Trabajando con repositorios remotos

Para poder colaborar en cualquier proyecto Git, necesitas saber cómo gestionar tus repositorios remotos. Los 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.

## Mostrando tus repositorios remotos

Para ver qué repositorios remotos tienes configurados, puedes ejecutar el comando `git remote`. Mostrará una lista con los nombres de los remotos que hayas especificado. Si has clonado tu repositorio, deberías ver por lo menos "origin" —es el 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
```

## Mostrando repos remotos

También puedes 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)
```

# 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
```

# Recibiendo de tus repositorios remotos

Para recuperar datos de tus repositorios remotos puedes ejecutar:

```
$ git pull [remote-name]
```

O:

```
$ git fetch [remote-name]
```

## 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
```

# Inspeccionando un repositorio remoto

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
```

# Eliminando y renombrando repositorios remotos

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
```

# Trabajando en ramas locales.

## Como crear y trabajar con ramas.

Supongamos que estamos trabajando en un proyecto y queremos añadir nueva funcionalidad al mismo. La forma adecuada de hacerlo con Git es crear una nueva rama con el nombre de la nueva funcionalidad donde añadiremos nuestros cambios y después cambiaremos a ella (git branch switching).

# Como generar una nueva rama

Podemos utilizar **git branch**

```
//Creamos una nueva rama.  
$ git branch "nombre de la rama"  
$ git checkout "nombre de la rama"  
//O simplemente.  
$ git checkout -b "nombre de la nueva rama"
```

## Como comprobar la rama en que estamos.

Para poder comprobar las ramas en la que nos encontramos utilizamos  
**git branch**

```
$ git branch //Nos muestra las ramas existentes y en la que nos encontramos.
```

# Podemos comprobar la diferencia entre ramas con diff

```
$ git diff --start[rama1] [rama2]
```

# Mezclando ramas con git merge

Nos debemos posicionar sobre la rama que va ha recibir el merge y posteriormente ejecutar el merge

```
$ git checkout master  
$ git merge [rama a mergear]
```

# Comprobando conflictos después de un merge

¿Qué pasa si editamos el mismo archivo en ambas ramas y sobre las mismas líneas?. Pues que se generará un conflicto que tendremos que resolver a mano por que Git aunque es útil y potente, no tiene la capacidad de decidir que versión del código en conflicto es la correcta.

# Vemos la situación después de un problema de merge

```
<<<<< HEAD:index.html
  [div id="footer"]contact : email.support@github.com[/div]
=====
  [div id="footer"]
    please contact us at support@github.com
  [/div]
>>>>> iss53:index.html
```

## Podemos ver el contenido

Donde nos dice que la versión en HEAD (la rama master, la que habías activado antes de lanzar el comando de fusión) contiene lo indicado en la parte superior del bloque (todo lo que está encima de =====) y que la versión en iss53 contiene el resto, lo indicado en la parte inferior del bloque. Para resolver el conflicto, has de elegir manualmente el contenido de uno o de otro lado.

# Como podemos deshacer un Merge

En caso de que hayamos realizado un merge por error podemos deshacer el mismo mediante una **git reset**

```
$ git reset --hard HEAD
```

El parámetro **--hard** se asegura que tanto nuestro índice de archivos como el directorio de trabajo cambien para que coincidan con lo que era antes del merge.

## Como eliminar ramas

Podemos eliminar las ramas existentes saliendo de la misma y utilizando la siguiente instrucción.

```
$ git branch -d [rama a eliminar]
```

# Deshaciendo cambios

# Quitar archivos del stage

Para sacar del stage los archivos que hemos añadido (git add) para hacerles commit, podemos utilizar git reset, así cambiaremos el estado del archivo a modified

```
$ git reset HEAD nombreArchivo.js
```

## Modificando tu último commit

Uno de los casos más comunes en el que quieras deshacer cambios es cuando confímas demasiado pronto y te olvidas de añadir algún archivo, o te confundes al introducir el mensaje de confirmación. Si quieres volver a hacer la confirmación, puedes ejecutar un commit con la opción `--amend`:

```
git add archivoOlvidado.js  
git commit --amend
```

# Deshaciendo la modificación de un archivo

Para eliminar todos las modificaciones en un archivo:

```
$ git checkout -- nombreArchivo.js
```

# Deshacer todas las modificaciones

Para eliminar todos las modificaciones y dejar todo como el último commit

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

# Guardar cambios para mas tarde

Eliminar los cambios realizados desde el ultimo commit y guardarlos para aplicarlos más tarde:

```
git stash
```

# Aplicar cambios guardados

```
git stash apply
```

# Revertir un commit antiguo

```
git revert <commitid>
```

# Github

Recordad que github es la carta de presentación mas potente que tiene un programador, y por ello debemos cuidarlo con mucho mimo.

## Respecto a los *commits*

Un buen log de commits se distingue por tener muchos commits pequeños, mensajes claros y muy específicos (nada general del tipo, "*lo que he hecho hoy*").

## Respecto al *readme*:

El *readme* es la cara visible del repo, y por ello también debemos cuidarlo tanto como podamos.

Debería abarcar como mínimo:

1. Para que sirve el repo.
2. como se usa.
3. Además siempre que se pueda debería tener un link a una demo. (en los proyectos de front es muy fácil hacer con gh-pages)

## Respecto a *vuestro perfil*:

Los repos más visibles son los *Pinned repositories*. Podéis y debéis elegirlos. Lo que más orgullosos estéis deberían estar pinneados.

## Respecto al código:

¿En qué nos fijamos cuando ojeamos el código de un programador? ¿En que funcione? No. Eso se da por supuesto. Lo que miramos es **el orden, la claridad, la legitimidad, un buen naming**... Si un código transmite paz es que el programador es bueno. Lo más complicado de la programación es **dominar el caos y evitar la complejidad**. Resulta fácil decirlo pero difícil hacerlo. Tened este punto SIEMPRE en mente durante vuestra carrera. Éste es el objetivo más alto al que podéis aspirar.

¿Y como podemos conseguirlo? Si estás haciendo algo y os resulta demasiado complicado de hacer quizás (casi seguro) haya una manera más fácil.