# Git

Git: Documentation | Book | gitfaq

## Instalación

Git for All Platforms: http://git-scm.com

Git for Windows: https://gitforwindows.org/
Además de Git SCM, añade herramientas como Git BASH, Git GUI, etc.

GitHub Desktop: https://windows.github.com
https://mac.github.com

## Obtener ayuda

```
git help <verb>               // git help clone
git <verb> -h | --help        // git clone -h | --help
```

## Guía rápida

Convertir el directorio actual en un repositorio: `git init`.
Descargar un repositorio ya existente: `git clone <url>`.
Descargar de GitHub y sincronizar archivos locales: `git pull`.

Observar todos los archivos del directorio actual, hacer un commit y guardarlo:

```
git add .
git commit -m "My message"
git push

// Lo mismo, en una línea:
git add . && git commit -m "My message" && git push

// O bien hacer un commit con los archivos que se hayan modificado (obviando git add):
git commit -am "My message"
git push
```

Comprobar versión de Git: `git --version`.
Actualizar Git: `git update-git-for-windows`.

# Empezar un proyecto

Git Book: Getting a Git Repository

Podemos empezar un proyecto con `git init` o con `git clone`:

## git init

Para empezar a controlar las versiones de un directorio que todavía no está controlado por Git escribimos:

```
git init
```

Esto creará un subdirectorio `.git`.

Si vamos a guardar el repositorio en GitHub hay que accecer a https://github.com/new y crear un nuevo repositorio. Muy importante tomar nota de la url, que tendrá la siguiente forma:

```
https://github.com/<user>/<repository-name>.git
```

Después ejecutamos las siguientes líneas en nuestro proyecto:

```
git remote add origin https://github.com/<user>/<repository-name>.git
git add .
git commit -m "first commit"
git push -u origin master
```

Lo mismo, pero si queremos renombrar la rama `master` a `main`:

```
git remote add origin https://github.com/<user>/<repository-name>.git
git branch -M main
git add .
git commit -m "first commit"
git push -u origin main
```

## git clone

Otra manera de empezar un proyecto es copiando un repositorio ya existente en GitHub:

```
git clone https://github.com/djimenezweb/pruebas.git
```

# remote

Los repositorios remotos son versiones que están guardadas en otro lugar: en GitHub, GitLab, Bitbucket, en la red local o incluso en otro directorio local.
Para comprobar los repositorios remotos vinculados a este proyecto local, escribimos `git remote`. Por defecto está vacío.

```
git remote                    // Muestra los alias disponibles

git remote -v | --verbose     // Muestra alias y url

git remote show origin        // Muestra el alias origin con más detalles
```

Para vincular un repositorio local con un repositorio remoto usaremos `git remote add` seguido por dos valores: el alias y la ruta.
Por ejemplo, en `git remote add origin https://github.com/john/test.git` el valor `origin` es el alias con el que nos referimos a esa ruta.

Cuando se copia un repositorio con `git clone` el campo `origin` ya viene por defecto con la url del repositorio. Esto se puede comprobar simplemente con `git remote show origin` : si está en blanco es que no hay ningún `origin` configurado.

# add, status, commit

Git Book: Recording Changes to the Repository

## git add

Todos los archivos de un directorio pueden ser `untracked` o `tracked`. Por defecto son `untracked`, es decir, no hemos pedido a Git que detecte cambios en ellos.
Con `git add <file>` se crea una captura (snapshot) de un archivo y Git observará si sufre cambios.
Si hacemos `git add <folder>` se observarán todos los archivos de esa carpeta.
Para observar todos los archivos del directorio actual: `git add .`.
Si se vuelve a modificar alguno de esos archivos, habrá que añadirlo de nuevo con `git add`.

Los archivos observados ( `tracked` ) pueden ser a su vez `unmodified`, `modified` o `stagged`.
Cuando un archivo está `stagged` quiere decir que su último snapshot será guardado en el próximo `commit`.
Con `git reset .` se puede deshacer la acción de `git add .`.

## git status

Podemos comprobar el estado de los archivos de un directorio con `git status`.
La versión abreviada ( `git status -s` o `--short` ) usa distintos símbolos: `?` (untracked), `A` (added to the staging area) o `M` (modified). Estos símbolos se muestran en dos columnas. La primera indica el estado del staged area y la segunda el estado del directorio de trabajo. Por ejemplo:

```
 M README              Modified in the working directory but not yet staged
MM Rakefile            Modified, staged and then modified again
A  lib/git.rb          Added to the staging area
M  lib/simplegit.rb    Modified and staged
?? LICENSE.txt         Untracked
```

## git commit

Para confirmar los cambios haremos `git commit`. De esta manera todos los cambios que hubiera en el staged area se guardarán. Es necesario añadir un mensaje entre comillas después de `-m` o `--message`:

```
git commit -m "fix benchmarks for speed"
```

Si hacemos `git commit` a secas se abrirá el editor de texto por defecto, donde habrá que escribir el mensaje entre comillas. ¡Importante! Si dejamos el mensaje en blanco se cancelará el commit.

#### Versión rápida

Podemos usar `git commit` con la opción `-a` o `--all` para añadir al stage area todos los archivos observados que se hayan modificado, sin necesidad de haber hecho `git add`. Por ejemplo:

```
git commit -a -m "Add new benchmarks"
```

Las flags `-a` y `-m` se pueden abreviar en una sola: `-am`.

# push, fetch, pull

Git: git push | Working with Remotes

## git push

Para subir todos los commits pendientes escribimos:

```
git push <url> <branch>
```

La url puede ser una ruta o un alias, como `origin`: `git push origin <branch>`.
Pero si ya tenemos `origin` y `branch` configurados bastará con escribir `git push` para subir los datos.

Con la flag `-u` o `--set-upstream` hacemos que los futuros `git pull` no necesiten más argumentos adicionales, sino que ya estarán configurados.

## git fetch y git pull

Con `git fetch origin` se descargan los datos del proyecto remoto que aún no tenemos. Pero no los fusiona (merge), es decir, no modifica el proyecto local hasta que el usuario lo haga manualmente.

En cambio `git pull` descarga los datos y los incorpora (merge) a la ramificación (branch) actual.
Es una combinación de `git fetch` y `git merge`.

(Para situaciones sencillas con una sola ramificación nos bastará con hacer `git pull`).

# Branches

Git Book: [Branches in a Nutshell]

Un repositorio tiene una rama, que por defecto es `master` (a veces se renombra a `main`).
Todos los commits que se hagan se harán en esa rama.
Pero podemos añadir ramas adicionales de tal manera que los commits se hagan sólo en esa rama.

## git branch y checkout

Con `git branch -a` o `--all` obtenemos una lista de todas las ramas. La rama en la que estemos estará destacada con algún color o símbolo especial.

### Crear rama

Con `git branch <name>` se crea una nueva rama y con `git checkout <name>` cambiamos a ella:

```
git branch my-first-branch
git checkout my-first-branch
```

También podemos crear una rama y cambiar a ella en una sola línea:

```
git checkout -b my-second-branch
```

Una vez hayamos cambiado a una rama, todos los commits se harán a esa rama.
Y para volver de nuevo a la rama principal:

```
git checkout master
```

### Renombrar rama

Para renombrar la rama actual se usa `git branch -M` seguido del nombre al que la queremos cambiar.
`-M` es una abreviación de `--move --force`.
Por ejemplo, `git branch -M main` renombra la rama `master` a `main`.

### Borrar rama

Borramos una rama con `-d` o `-D` (funcionan de manera distinta si se ha hecho un merge o no):

```
git branch -D my-first-branch
```

## git merge

Podemos fusionar dos ramas con `merge`. Como normalmente insertaremos el contenido de la nueva rama en `master`, deberemos estar en la rama principal: `git checkout master` y desde ahí hacemos `merge`:

```
git merge my-first-branch
```

Ahora todos los cambios que se hicieron en la rama `my-first-branch` también estarán presentes en `master`. La rama puede borrarse, puesto que está integrada en el contenido principal:

```
git branch -d my-first-branch
```

# Deshacer cambios

Podemos volver al estado de un commit anterior.
Hay varias maneras distintas: `checkout`, `revert` y `reset`.

Primero, para ver una lista de commits usamos `git log`:

```
git log
git log --oneline
```

Saldrá una lista de commits con un código (como `b7b2d3d`) y el mensaje que se escribió al hacer el commit.
Usaremos los comandos `git checkout`, `git revert` o `git reset` con el código del commit. Por ejemplo:

```
git checkout b7b2d3d

git revert b7b2d3d

git reset b7b2d3d
```

## git checkout

`git checkout` carga en el editor el estado del commit anterior pero sin guardar nada. Es un método seguro que permite leer distintos commits sin alterar nada.
Se pueden hacer cambios y guardar en un commit nuevo.
También se puede hacer una nueva rama a partir de este commit con `git checkout -b <name>`.
Para volver a la versión principal: `git checkout master`.

## git revert

`git revert` anula los cambios de un commit en concreto, como si ese commit no hubiera existido nunca.
En realidad el commit sigue existiendo en la lista de commits, pero se añade un commit nuevo que representa los cambios deshechos.

## git reset

`git reset` deja el proyecto en un estado anterior y borra los commits que se hubieran hecho después de ese estado.
Si tenemos el proyecto abierto en un editor se seguirá mostrando el proyecto como estaba antes de hacer el borrado. Esto puede ser útil para borrar varios commits de la lista y fusionarlos en un commit nuevo.

Con `git reset <commit> --hard` el editor sí que se actualiza.

# Configuración inicial

Git Book: First-Time Git Setup
Greg Foster: How to configure your Git Repository

Hay tres niveles de configuración: `--local`, `--global` y `--system`:

**--local**
La configuración `--local` se aplica a un repositorio específico. En caso de contradicción con los ajustes globales, los locales tienen prevalencia. Se usa para ajustes que sólo se apliquen a un proyecto en concreto. Se guardan en el archivo `.git/config` en el directorio del repositorio.

**--global**
La configuración `--global` afecta a todos los repositorios del usuario actual del ordenador.
Se usa para ajustes que deberían ser los mismos para todos los proyectos del mismo usuario, como nombre de usuario y email, patrones de ignore, etc.
Se guardan en el archivo `.gitconfig` del directorio del usuario. En mi caso: `C:\Users\Daniel\.gitconfig`.
En Mac y Linux: `~/.gitconfig`.

**--system**
La configuración `--system` afecta a todos los usuarios del ordenador actual. Se usa para ajustes que deberían ser universales para todos los proyectos y usuarios de ese ordenador, por ejemplo, para configuración de seguridad o de red.
Se guardan en el archivo `C:/Program Files/Git/etc/gitconfig`. En Mac y Linux: `/etc/gitconfig`.

## Comprobar configuración

Para comprobar la configuración existente (muestra local, global y del sistema):

```
git config --list

git config --list --show-origin             Muestra la ruta de los archivos
```

Y para comprobar la configuración de una clave en particular escribimos `git config <key>`:

```
git config user.name                        djimenezweb
```

## Editar configuración

Para abrir cada uno de los archivos de configuración ejecutamos:

```
git config --system --edit                  System
git config --global --edit                  Global
git config --edit                           Local
```

Se abrirá con el editor de texto especificado en el ajuste `core.editor`.

## Usuario y email

Hay que configurar el nombre de usuario y el email para identificar a la persona que hace los commits.

Esto no tiene que ver necesariamente con la cuenta de GitHub.
Añadimos `--global` para que se aplique a todos los proyectos del mismo usuario.
Si hiciera falta otro nombre de usuario u otro email, ejecutamos el mismo comando pero sin `--global`.

```
git config --global user.name "djimenezweb"

git config --global user.email "djimenezweb@gmail.com"
```

## Saltos de línea

Git: Git Configuration

Windows usa una secuencia de dos caracteres de control ( `\r\n` ) para indicar un salto de línea: Carriage Return (CR, `\r` ) y Line Feed (LF, `\n` ), mientras que Mac (desde OS X) y Linux sólo usan LF ( `\n` ).

Para evitar problemas de compatibilidad podemos configurar Git para normalizar los saltos de línea:

```
git config --global core.autocrlf true      En Windows

git config --global core.autocrlf input     En Mac y Linux
```

**Nota:** También hay que configurar **Prettier** y **VS Code** porque sus ajustes de saltos de línea pueden entrar en conflicto.

This setup should leave you with CRLF endings in Windows checkouts, but LF endings on macOS and Linux systems and in the repository.
Si es un proyecto de Windows y sólo para Windows entonces se puede dejar en `false`.

When CRLF conversion is enabled, Git will convert CRLF to LF during commit and LF to CRLF during checkout. A file that contains a mixture of LF and CRLF before the commit cannot be recreated by Git. For text files this is the right thing to do: it corrects line endings such that we have only LF line endings in the repository.

# Submodules

Git: git submodule | Submodules

Cómo integrar un repositorio completo o parcial dentro de otro, según ChatGPT:

Git submodules allow you to include one Git repository as a subdirectory in another.
Navigate to the Repository where you want the files to exist.

```
cd <target-repo>
```

Add the Other Repository as a Submodule:

```
git submodule add <URL-of-source-repo> <path-to-submodule>
```

Replace `<path-to-submodule>` with the directory where you want the files from the other repository to appear.

Manually Include Only Specific Files: After the submodule is added, you can use sparse checkout to include only specific files:

```
git config -f .gitmodules submodule.<path-to-submodule>.shallow true
git config core.sparseCheckout true
echo "path/to/file1" >> .git/info/sparse-checkout
echo "path/to/file2" >> .git/info/sparse-checkout
git submodule update --init --checkout
```

Synchronize Updates: When changes are made to the files in the source repository, update the submodule:

```
git submodule update --remote
```

If you make changes in the submodule, you can commit them back to the original source repository. Here's how the process works: Go to the submodule directory within the parent repository:

```
cd <path-to-submodule>
```

Edit the files you want within the submodule.
Stage and commit the changes within the submodule:

```
git add <files>
git commit -m "Your commit message"
```

Push the changes to the original source repository:

```
git push
```

You will need appropriate permissions to push to the source repository.
Updating the Parent Repository
Once you've pushed changes to the source repository, the submodule will be at a newer commit than what the parent repository tracks. To update the parent repository:
Navigate back to the root of the parent repository:

```
cd <parent-repo>
```

Stage the Updated Submodule Reference: The parent repository tracks the submodule as a specific commit. Update the submodule reference:

```
git add <path-to-submodule>
```

Commit the change in the parent repository:

```
git commit -m "Update submodule to latest commit"
```

Push the changes in the parent repository to its remote:

```
git push
```

# Múltiples remotes

Stack Overflow: Git - Pushing code to two remotes
Jigarius: Working with Git remotes and pushing to multiple Git repositories

In recent versions of Git you can add multiple `pushurl`s for a given remote. Use the following to add two `pushurl`s to your `origin`:

```
git remote set-url --add --push origin git://original/repo.git
git remote set-url --add --push origin git://another/repo.git
```

So when you push to `origin`, it will push to both repositories.
Junio C. Hamano, the Git maintainer, explained it's how it was designed. Doing `git remote set-url --add --push <remote_name> <url>` adds a `pushurl` for a given remote, which overrides the default URL for pushes. However, you may add multiple `pushurl`s for a given remote, which then allows you to push to multiple remotes using a single `git push`. You can verify this behavior below:

```
$ git clone git://original/repo.git

$ git remote -v
origin  git://original/repo.git (fetch)
origin  git://original/repo.git (push)

$ git config -l | grep '^remote\.'
remote.origin.url=git://original/repo.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
```

Now, if you want to push to two or more repositories using a single command, you may create a new remote named `all`, or keep using the `origin`, though the latter name is less descriptive for this purpose. If you still want to use `origin`, skip the following step, and use `origin` instead of all in `all` other steps.

So let's add a new remote called `all` that we'll reference later when pushing to multiple repositories:

```
$ git remote add all git://original/repo.git

$ git remote -v
all git://original/repo.git (fetch)               <-- ADDED
all git://original/repo.git (push)                <-- ADDED
origin  git://original/repo.git (fetch)
origin  git://original/repo.git (push)

$ git config -l | grep '^remote\.all'
remote.all.url=git://original/repo.git            <-- ADDED
remote.all.fetch=+refs/heads/*:refs/remotes/all/* <-- ADDED
```

Then let's add a `pushurl` to the `all` remote, pointing to another repository:

```
$ git remote set-url --add --push all git://another/repo.git

$ git remote -v
all git://original/repo.git (fetch)
all git://another/repo.git (push)                 <-- CHANGED
origin  git://original/repo.git (fetch)
origin  git://original/repo.git (push)

$ git config -l | grep '^remote\.all'
remote.all.url=git://original/repo.git
```

```
    remote.all.fetch=+refs/heads/*:refs/remotes/all/*
    remote.all.pushurl=git://another/repo.git         <-- ADDED
```

Here `git remote -v` shows the new `pushurl` for push, so if you do git `push all master`, it will push the `master` branch to `git://another/repo.git only`. This shows how `pushurl` overrides the default url (remote.all.url).

Now let's add another `pushurl` pointing to the original repository:

```
    $ git remote set-url --add --push all git://original/repo.git

    $ git remote -v
    all git://original/repo.git (fetch)
    all git://another/repo.git (push)
    all git://original/repo.git (push)                <-- ADDED
    origin  git://original/repo.git (fetch)
    origin  git://original/repo.git (push)

    $ git config -l | grep '^remote\.all'
    remote.all.url=git://original/repo.git
    remote.all.fetch=+refs/heads/*:refs/remotes/all/*
    remote.all.pushurl=git://another/repo.git
    remote.all.pushurl=git://original/repo.git        <-- ADDED
```

You see both `pushurl`s we added are kept. Now a single `git push all master` will push the `master` branch to both `git://another/repo.git` and `git://original/repo.git`.

IMPORTANT NOTE: If your remotes have distinct rules (hooks) to accept/reject a push, one remote may accept it while the other doesn't. Therefore, if you want them to have the exact same history, you'll need to fix your commits locally to make them acceptable by both remotes and push again, or you might end up in a situation where you can only fix it by rewriting history (using `push -f`), and that could cause problems for people that have already pulled your previous changes from the repo.