



## Git. Nivel I

Mayo, 2018



## Objetivos del nivel

- Entender los conceptos implicados en el control de Versiones.
- Aprender a usar los comandos de Git.
- Crear repositorios locales y remotos.
- Crear y usar un repositorio en GitHub.

## Prerrequisitos del nivel

Este nivel del curso no tiene pre requisitos

## Acerca de este manual

Este manual pertenece al Centro de Asesoramiento y Desarrollo Informático C.A. (CADI F1). Para obtener más información sobre este u otros cursos visite nuestra sitio Web [www.cadif1.com](http://www.cadif1.com), escribanos a la dirección de correo [cadi@cadif1.com](mailto:cadi@cadif1.com) o visítenos en nuestra sede ubicada en la Av. Pedro León Torres con calle 59, Centro Comercial Sotavento, piso 2 oficina 27, Barquisimeto estado Lara, Venezuela. Tlf. 0251-7179247, 0251-4410268.

Las marcas mencionadas en este manual son propiedad de sus respectivos dueños. Copyright 2018. Todos los derechos reservados.



## Contenido del nivel

### Capítulo 1. Introducción a Git

- 1.1.- Control de Versiones.
- 1.2.- Objetivos del Control de Versiones.
- 1.3.- Git.

### Capítulo 2. Instalación y Configuración

- 2.1.- Instalación de Git.
- 2.2.- Git Bash.
- 2.3.- Primera Configuración.

### Capítulo 3. Repositorios Locales

- 3.1.- Repositorios Git.
- 3.2.- Creando un Repositorio.
- 3.3.- Estado Del Repositorio.

### Capítulo 4. Ciclo de Vida Dentro Del Repositorio

- 4.1.- Áreas Del Repositorio.
- 4.2.- Preparando y Confirmando Cambios.

### Capítulo 5. Viajando en el Tiempo

- 5.1.- Historial de Confirmaciones.
- 5.2.- Comando Checkout.
- 5.3.- Comando Reset.

### Capítulo 6. Ramas. Parte 1

- 6.1.- Definición.
- 6.2.- Creando Ramas.
- 6.3.- Eliminando Ramas.

## Capítulo 7. Ramas. Parte 2

- 7.1.- Union de Ramas.
- 7.2.- Resolviendo Conflictos.

## Capítulo 8. Github. Parte 1

- 8.1.- Plataformas Para el Trabajo en Equipo.
- 8.2.- Github.
- 8.3.- Usando Github.

## Capítulo 9. Github. Parte 2

- 9.1.- Creando Repositorios Remotos.
- 9.2.- Repositorio Remoto& Repositorio Local.
- 9.3.- Clonando Repositorios.

## Capítulo 10. Github. Parte 3

- 10.1.- Contribuyendo en Proyectos de Terceros.
- 10.2.- Pull Request.

## Capítulo 11. Github. Parte 4

- 11.1.- Organizaciones.
- 11.2.- Equipos.

## Capítulo 12. Github Pages

- 12.1.- Página Personal.
- 12.2.- Páginas Para Repositorios.

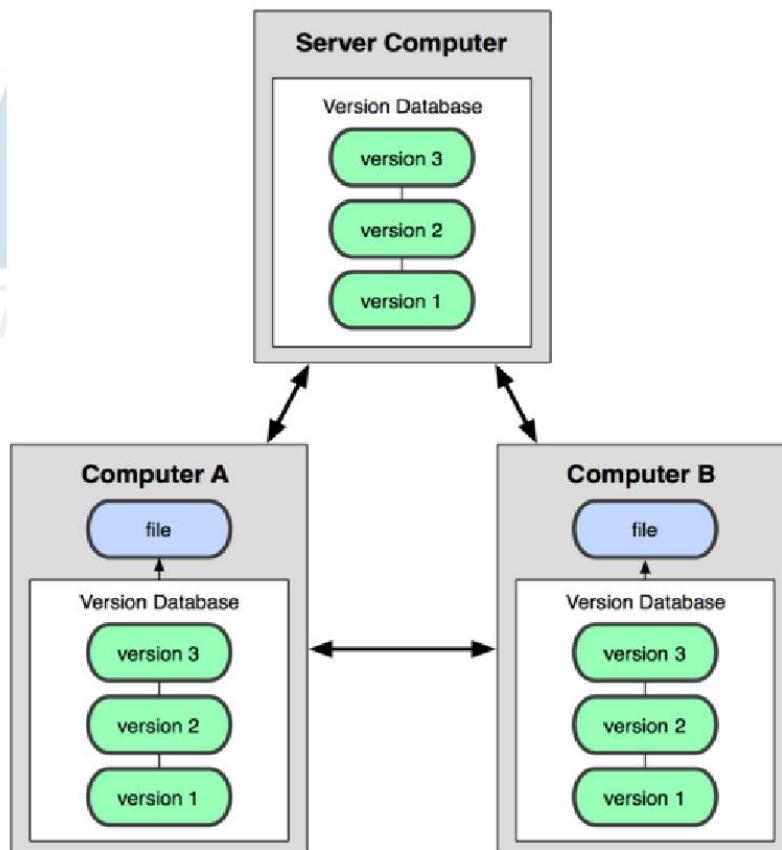
## Capítulo 13. Clientes Git

- 13.1.- Introducción.
- 13.2.- Smartgit.
- 13.3.- Interfaz Gráfica Smartgit.

## Capítulo 1. INTRODUCCION A GIT

### 1.1.- Control de Versiones

Se llama control de versiones a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo. Una versión, revisión o edición de un producto, es el estado en el que se encuentra el mismo en un momento dado de su desarrollo o modificación. En el desarrollo de software, un sistema de control de versiones brindará apoyo para gestionar de manera eficiente, cronológica y detallada las diversas versiones que se van generando en el proyecto, con el objetivo de agilizar los tiempos de desarrollo y facilitando el trabajo en equipo.



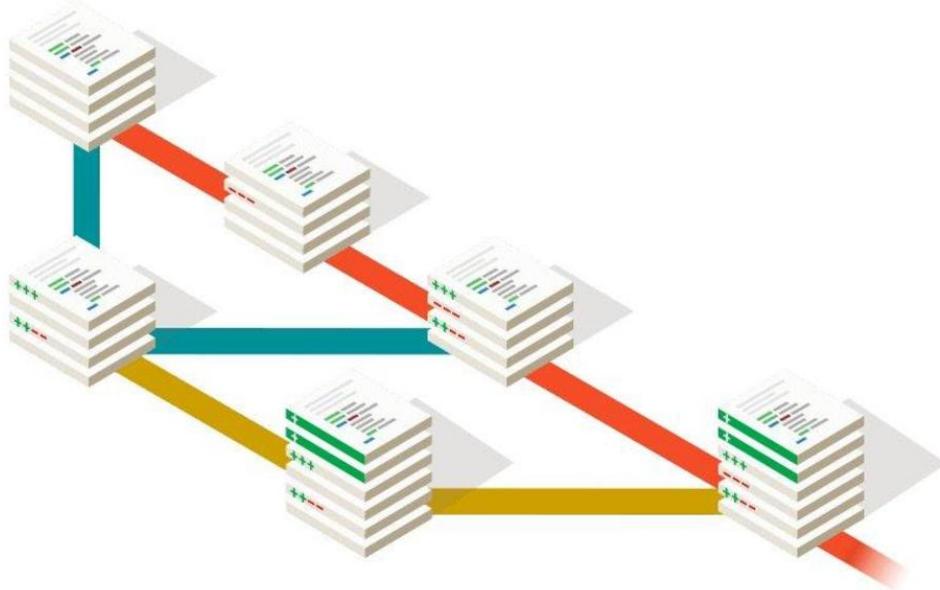
Durante los 70's hasta inicios del 2000, muchos profesionales (principalmente de software) se tuvieron que enfrentar a desarrollar proyectos de forma muy pesada. Se encontraban con 3 problemas:

- Proyectos difíciles de gestionar y liderar
- Riesgos a sobreseguir con mi código el avance formal del equipo
- La centralización y poca probabilidad de trabajar remotamente

Con esto, existieron diferentes sistemas de control de versiones (SCV) que se enfocaron fuertemente a atacar esto, con todo lo que hubiera en la época:

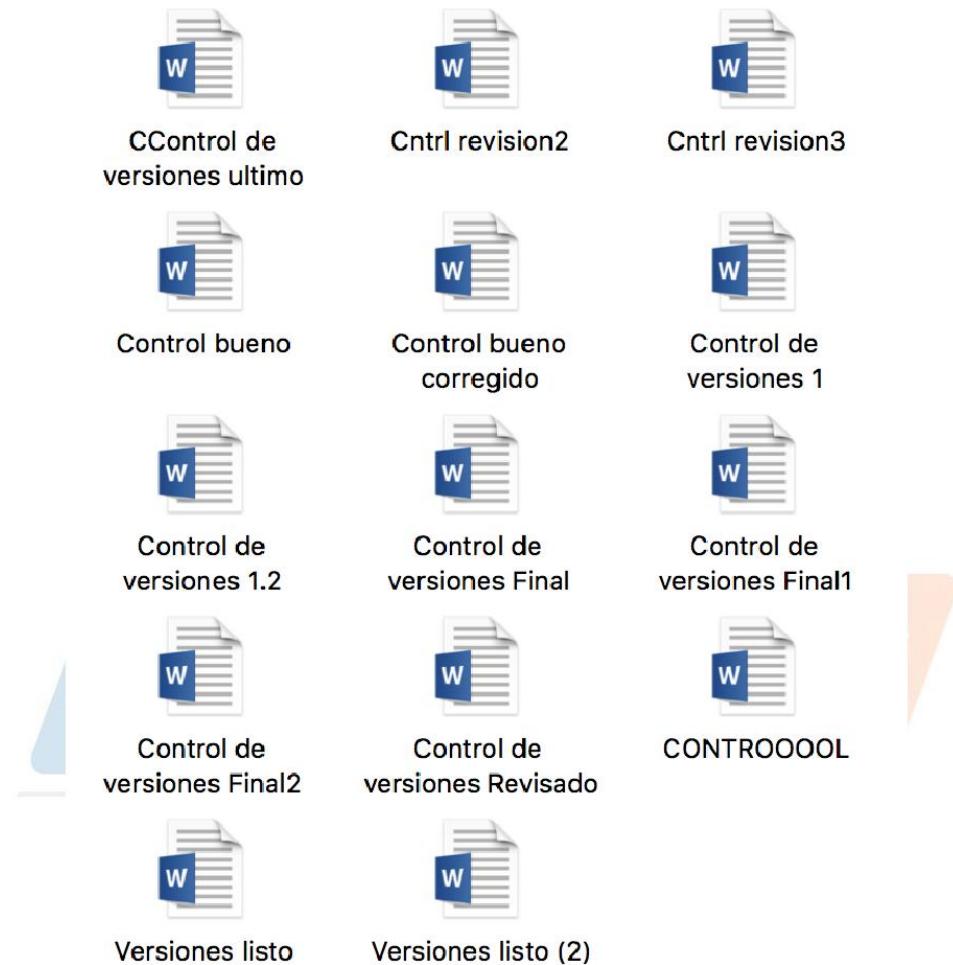
- SCCS (1972)
- RCS (1982)
- CVS (1986-1990)
- SVN (2000)
- BitKeeper SCM (2000)
- Mercurial (2005)

A pesar de que la evolución iba naturalmente bien, no fue hasta la última tecnología que encajó todas las piezas de forma funcional y usable. Además, internet como medio de comunicación principal, ofreció el último peldaño para que destacara a nivel mundial. La llegada de Git (2005) revolucionó la forma de gestionar código, la comunicación y colaboración con profesionales en proyectos de software, incluidos web.

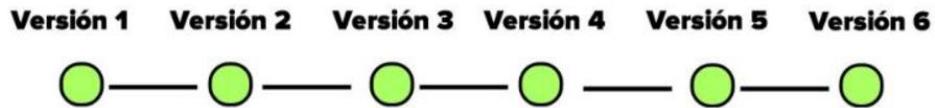


### 1.2.- Objetivos del Control de Versiones

Es muy frecuente encontrar desarrollos de proyectos en los cuales existen gran cantidad de archivos, donde cada uno es una versión mejorada del anterior lo cual hace prácticamente incierto cuales son los archivos más actualizados representando un problema muy poco práctico. Los sistemas de control de versiones buscan eliminar el problema reflejado en la siguiente imagen:



Un sistema de control de versiones busca gestionar ágilmente proyectos. Parte de su principal propósito es que puedas regresar a un estado anterior del proyecto o conocer, incluso, toda su evolución en el tiempo. Desde sus inicios hasta donde se encuentra actualizado. Puedes ver a los SCV como máquinas del tiempo, que permiten regresar a cualquier momento que quieras de tu proyecto.



### 1.3.- Git

Git es un sistema de control de versiones elaborado por Linus Torvalds de libre distribución y de código abierto diseñado para manejar todo, desde proyectos pequeños a muy grandes con rapidez y eficiencia.

El núcleo de Linux es un proyecto de software de código abierto con un alcance bastante grande. Durante la mayor parte del mantenimiento del núcleo de Linux (1991-2002), los cambios en el software se pasaron en forma de parches y archivos. En 2002, el proyecto del núcleo de Linux empezó a usar un DVCS propietario llamado BitKeeper. Luego el equipo de desarrollo de Linux encabezado por Linus Torvalds implementó Git como su propio DVCS.

Git es fácil de aprender y tiene una huella muy pequeña con un rendimiento rápido. Supera las herramientas de DVCS como Subversion, CVS, Perforce y ClearCase con funciones como ramificación local, áreas de puesta en escena convenientes y múltiples flujos de trabajo.

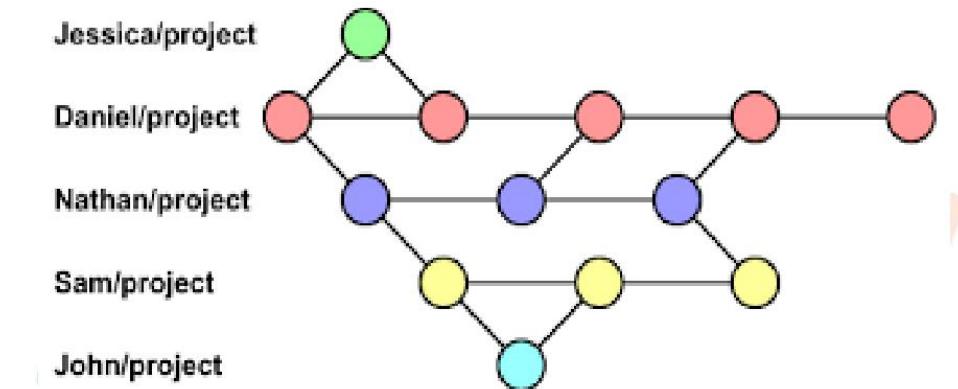


La característica de Git que realmente hace que sea distinto de casi todos los otros SCM es su modelo de ramificación. Git te permite y te anima a tener múltiples ramas locales que pueden ser totalmente independientes entre sí. La creación, la fusión y la supresión de esas líneas de desarrollo toma segundos.

Esto significa que usted puede hacer cosas como:

- Comutación de contexto sin fricción. Cree una rama para probar una idea, comience varias veces, cambie de nuevo a donde se ramificó, aplique un parche, cambie de nuevo a donde está experimentando y fíjelo.
- Reglas Basadas en el Rol . Tener una rama que siempre contiene sólo lo que va a la producción, otro que se fusionan en el trabajo para la prueba, y varios más pequeños para el trabajo diario.
- Flujo de trabajo basado en funciones . Cree nuevas ramas para cada nueva función en la que esté trabajando, de modo que pueda pasar sin problemas entre ellas y, a continuación, eliminar cada una de las ramas cuando se fusionen con la línea principal.
- Experimentación desecharable . Crear una rama para experimentar, darse cuenta de que no va a funcionar, y sólo eliminarlo - abandonar el trabajo - con nadie más lo vea (incluso si has empujado otras ramas en el ínterin).

Otra de las características a resaltar de git como herramienta, es que está pensada para versionar los proyectos de desarrollo gracias a su capacidad de crear una línea de tiempo de las versiones de los archivos. Git permite la creación de diversas líneas de tiempo separadas que en cualquier momento pueden ser integradas entre sí. Esto facilita en gran medida el trabajo en equipo, ya que se pueden crear líneas de tiempo del proyecto para cada tarea o cada uno de los involucrados en el desarrollo, donde dichos cambios luego puedan ser integrados a una versión central del proyecto.

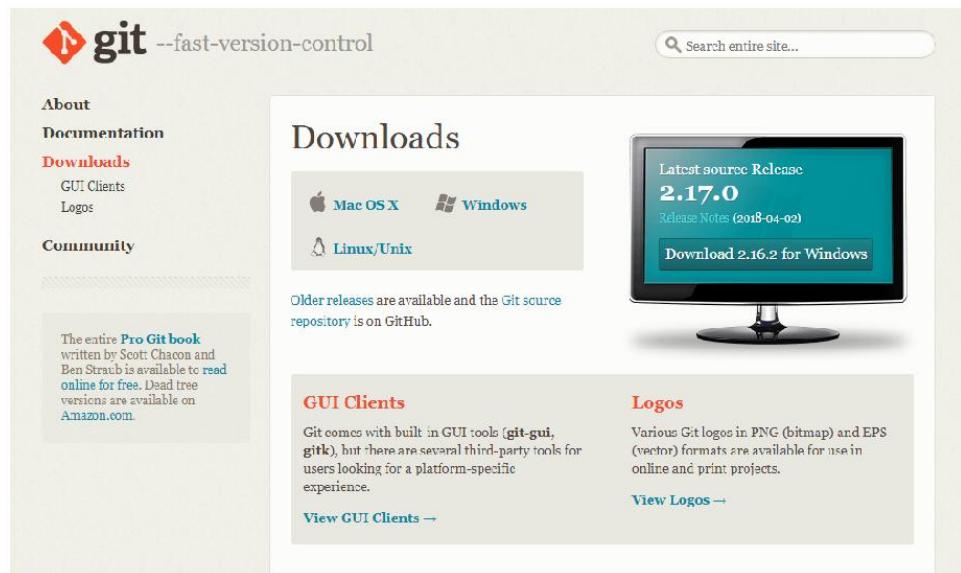


ACADEMIA DE SOFTWARE

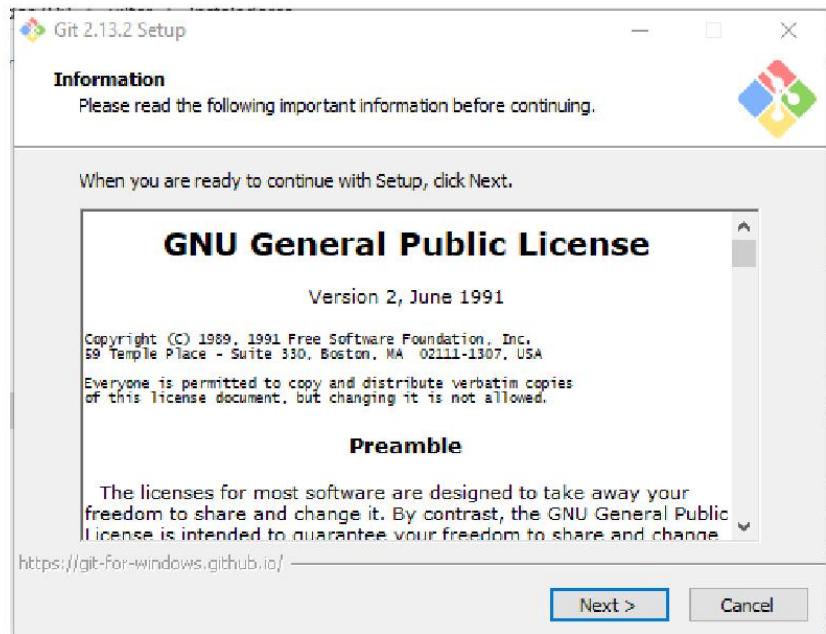
## Capítulo 2. INSTALACIÓN Y CONFIGURACIÓN

### 2.1.- Instalación de Git

El sitio oficial del proyecto Git es <https://git-scm.com>. Desde la sección de descarga se puede obtener los instaladores para diversos sistemas operativos. El sitio oficial provee un apartado de documentación para aprender los aspectos principales de la aplicación. En la imagen se muestra la página de descarga.



En el caso de Windows, Git ofrece un archivo ejecutable (Instalador), el cual permite realizar la instalación del software. Como típico instalador de Windows, la instalación de Git es muy sencilla, solo se debe seguir el proceso asistente de instalación haciendo clic en “siguiente” en todos los pasos del mismo. Para la instalación de Git no se requiere ningún software especializado previamente instalado en el computador.



Luego de terminar el proceso de con el asistente de instalación, se debe comprobar el correcto funcionamiento del software. Para esto, mediante la consola de Windows se realiza el siguiente comando para conocer la versión de Git que se ha instalado en el computador:

git versión

Se mostrará un mensaje con la versión instalada en el software. En el caso de cualquier otra salida no esperada, es indicativo de que Git no fue instalado correctamente. En la imagen se muestra la ejecución del comando y su respectiva salida

A screenshot of a Windows Command Prompt window titled "C:\WINDOWS\system32\cmd.exe". The command "git version" is entered at the prompt "C:\Users\vilfer>". The output shows "git version 2.13.2.windows.1".

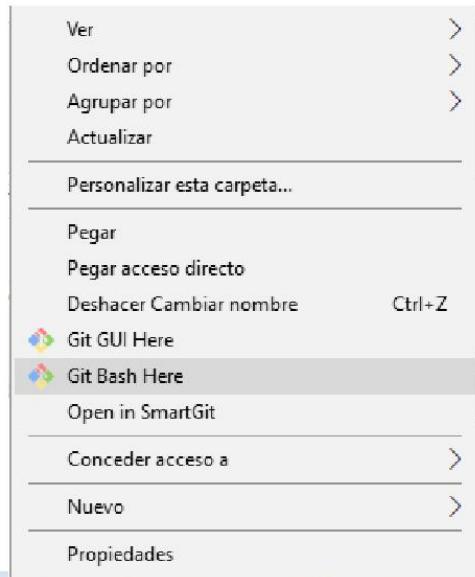
## 2.2.- Git Bash

La manera inicial de usar Git es mediante una terminal, es por esto el instalador de Git provee una aplicación llamada Git Bash, la cuales un intérprete de comandos que va a servir como medio para que el usuario pueda utilizar los comandos de Git. Existen 2 maneras de usar Git mediante consola:

- Consola del sistema operativo: Haciendo uso de la propia consola que viene con el sistema operativo
- Git Bash: Usando el programa Git Bash el cual emula una consola especial para ejecutar comandos de Git.



Este programa se puede ejecutar como cualquier aplicación de escritorio. El ejecutable se encuentra en C:\Program Files\Git\git bash.exe. Existe una manera rápida de abrir Git bash en un directorio específico, simplemente con click derecho sobre el directorio y seleccionando la opción “Git Bash here”. Esto último ejecutará el programa y ya se podrán usar los comandos de Git en el directorio especificado.



### 2.3.- Primera Configuración

Lo primero que se debe hacer cuando se instala Git es establecer un nombre de usuario y dirección de correo electrónico. Esto es importante porque las confirmaciones de cambios en Git usan esta información, y es introducida de manera inmutable en los en los cambios que se envían. Es importante mencionar que esta configuración por ser global, necesitas hacerla una sola vez.

Git trae una herramienta llamada `git config` que te permite obtener y establecer variables de configuración, que controlan el aspecto y funcionamiento de Git. El programa necesita que se coloque información de configuración inicial como nombre y correo electrónico.

El comando de Git utilizado para establecer la configuración inicial es el comando “config”. De la siguiente manera se puede configurar el nombre y el email del usuario el cual está utilizando Git en el

```
$ git config --global user.name "Vilfer Alvarez"  
$ git config --global user.email "vilfer@cadif1.com"
```

Es probable que si no se realiza este paso inicial, cuando se intente usar algún otro comando para la gestión del proyecto, git le dé la sugerencia de que establezca estas variables globales antes de seguir con el trabajo.

El comando anterior permitía establecer el nombre y email del usuario, pero en el caso que simplemente se quiera obtener y mostrar la información que contienen esas variables, se usa el mismo comando pero obviando el valor de la variable, en la imagen se muestra el ejemplo:

```
MINGW32:/d/vilfer/CADIF1/GIT/Material/img
vilfer@DESKTOP-DMOLHOS MINGW32 /d/vilfer/CADIF1/GIT/Material/img
$ git config --global user.name
Vilfer Alvarez

vilfer@DESKTOP-DMOLHOS MINGW32 /d/vilfer/CADIF1/GIT/Material/img
$ git config --global user.email
vilferalvarez@gmail.com

vilfer@DESKTOP-DMOLHOS MINGW32 /d/vilfer/CADIF1/GIT/Material/img
$ |
```

ACADEMIA DE SOFTWARE

## Capítulo 3. REPOSITORIOS LOCALES

### 3.1.- Repositorios Git

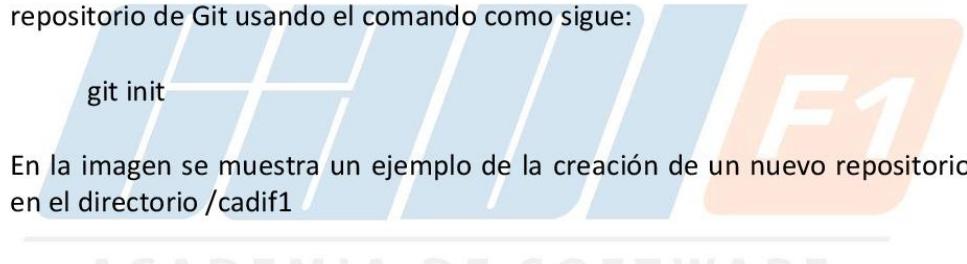
Cualquier carpeta del sistema puede ser un contenedor o repositorio para Git. Ahí se establecerá la configuración e historial bajo una carpeta nombrada .git que se encuentra oculta. No es necesario iniciar el proyecto a la par que un repositorio. De hecho, estos se pueden crear aún cuando ya se ha avanzado con nuestro código.

### 3.2.- Creando un Repositorio

Para comenzar un proyecto en Git se debe hacer uso del comando “git init”. Basta con posicionarse en el directorio en el cual se quiere iniciar un repositorio de Git usando el comando como sigue:

```
git init
```

En la imagen se muestra un ejemplo de la creación de un nuevo repositorio en el directorio /cadif1

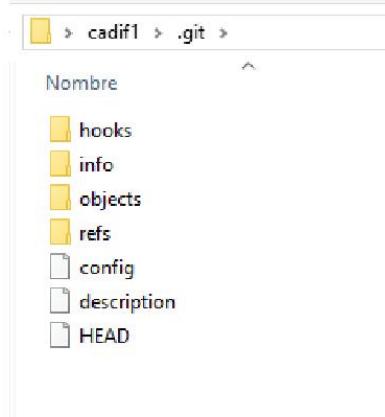


The logo features the text "ACADEMIA DE SOFTWARE" in a light blue font, with "CADIF1" in a larger orange font to the right. The background has a blue and white striped pattern.

```
MINGW32:/c/Users/vilfer/Desktop/cadif1
vilfer@DESKTOP-DM0LH0S MINGW32 ~/Desktop/cadif1
$ git init
Initialized empty Git repository in C:/Users/vilfer/Desktop/cadif1/.git/
vilfer@DESKTOP-DM0LH0S MINGW32 ~/Desktop/cadif1 (master)
$ |
```

En el directorio cadif1 se ha creado una carpeta oculta llamada .git en la cual se almacenan archivos y subdirectorios necesarios para llevar el control del repositorio.

El usuario no debería interactuar directamente con esta carpeta, ya que es de uso específico del programa.



### 3.3.- Estado Del Repositorio

Para conocer el estado de los archivos de tu repositorio debes invocar el comando “git status”. Este comando mostrará los archivos (si existen) que han tenido alguna modificación o que se hayan agregado recientemente.

En el caso de un repositorio vacío (sin archivos) al hacer un git status debe mostrar un mensaje similar al de la siguiente imagen:

```
vilfer@DESKTOP-DM0LH0S MINGW32 ~/Desktop/cadif1 (master)
$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

Recuerda que cada archivo de tu directorio de trabajo puede estar en uno de estos dos estados: bajo seguimiento (tracked), o sin seguimiento (untracked). Los archivos bajo seguimiento son aquellos que existían en la última instantánea; pueden estar sin modificaciones, modificados, o preparados. Los

archivos sin seguimiento son todos los demás —cualquier archivo de tu directorio que no estuviese en tu última instantánea ni está en tu área de preparación.

En la siguiente imagen se muestra el resultado del comando git status luego de haber agregado un archivo te texto llamado “README.txt” al repositorio:

```
vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ git status
On branch master
Initial commit

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

    README.txt

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



## Capítulo 4. CICLO DE VIDA DENTRO DEL REPOSITORIO

### 4.1.- Areas Del Repositorio

En un repositorio de Git existen 3 áreas en donde se albergan los archivos según el seguimiento que hace git de ellos. Estas áreas son las siguientes:

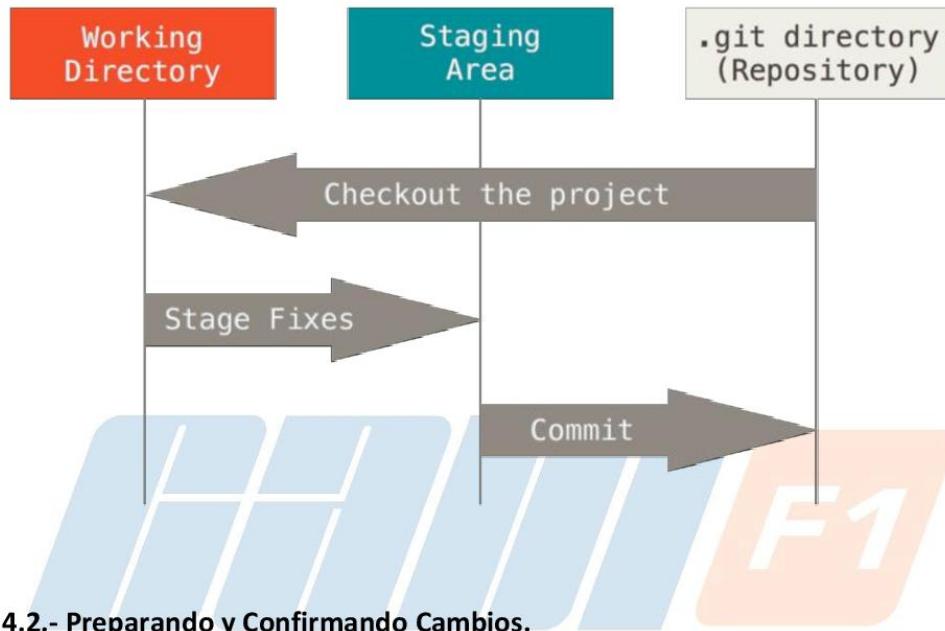
- Working Area: Es donde se encuentran los archivos que están siendo editados.
- Staging Area: Es donde se encuentran archivos que ya han sido modificados y están preparados para la confirmación.
- Local Repository: Acá se encuentran las versiones finales de los archivos que ya estaban preparados(Staging Area) para ser confirmados.

Este ciclo de vida se repite una y otra vez para todo archivo que sufra modificaciones.



Git estará monitoreando las modificaciones que se realicen sobre los archivos del repositorio, de esta manera en el Working area se encuentran los archivos en los que se está trabajando actualmente, cuando los archivos ya han sido modificados y se quieren registrar los cambios, se deben pasar a Staging área para prepararlos para la confirmación de estos cambios. La confirmación de los cambios ocurre en el momento en que se pasan los

archivos a repositorio local, es justo en ese momento en donde termina temporalmente el ciclo de vida de cambios para uno o varios archivos.



#### 4.2.- Preparando y Confirmando Cambios.

Cuando se editan archivos del repositorio, Git notará el cambio y va a etiquetar a ese archivo como “Modified”, tras hacer un “git status” se mostrarán los archivos que han sido modificados. Estos archivos seguirán catalogados como modificados hasta que pasen al Staging area, lo que indica es que se quieren preparar esos archivos para posteriormente confirmar las modificaciones que se realizaron sobre ellos. Para que un archivo pase al staging área se usa el siguiente comando:

```
git add nombreArchivo
```

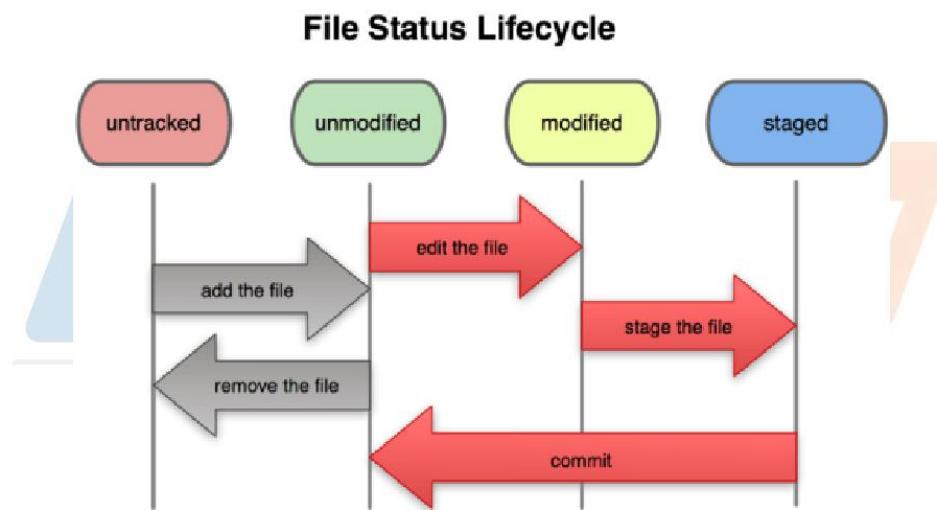
Si se necesita pasar todos los archivos que han sido modificados, se utiliza la siguiente variante el comando add:

```
git add .
```

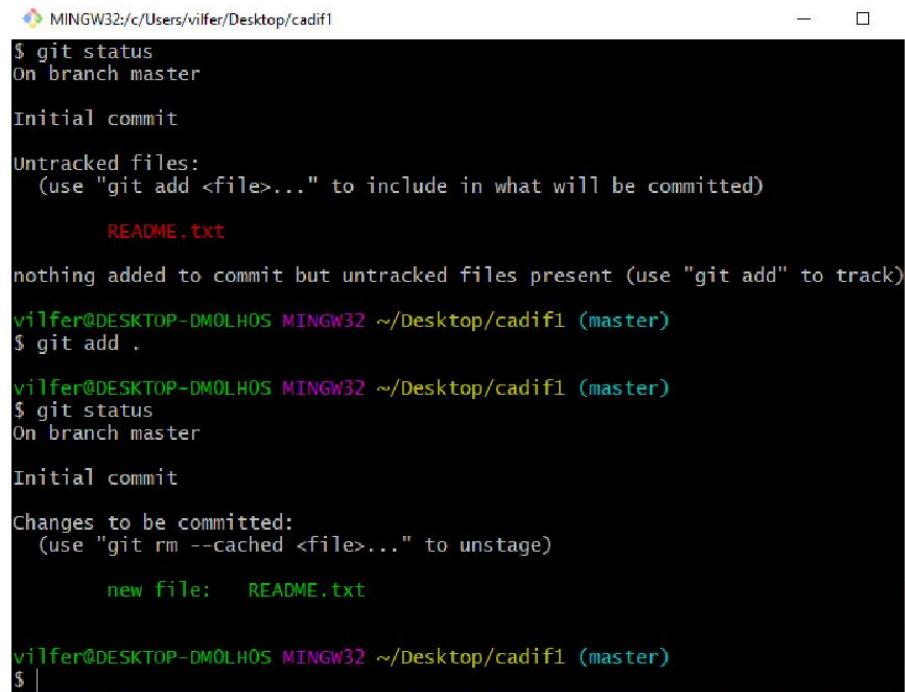
Los archivos que están en el área de preparación (staging area) serán confirmados tras realizar una ejecución del comando commit se la manera siguiente:

```
git commit -m "Mensaje para identificar el commit"
```

Donde el texto que aparece entre comillas es un texto arbitrario colocado por el usuario para dejar claro los cambios que van involucrados en esa confirmación.



En la imagen se muestra el resultado del estado el repositorio (git status) tras haber hecho un git "add ." para agregar al área de preparación al archivo README.txt



```

MINGW32:/c/Users/vilfer/Desktop/cadif1
$ git status
On branch master

Initial commit

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

    README.txt

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

vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ git add .

vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ git status
On branch master

Initial commit

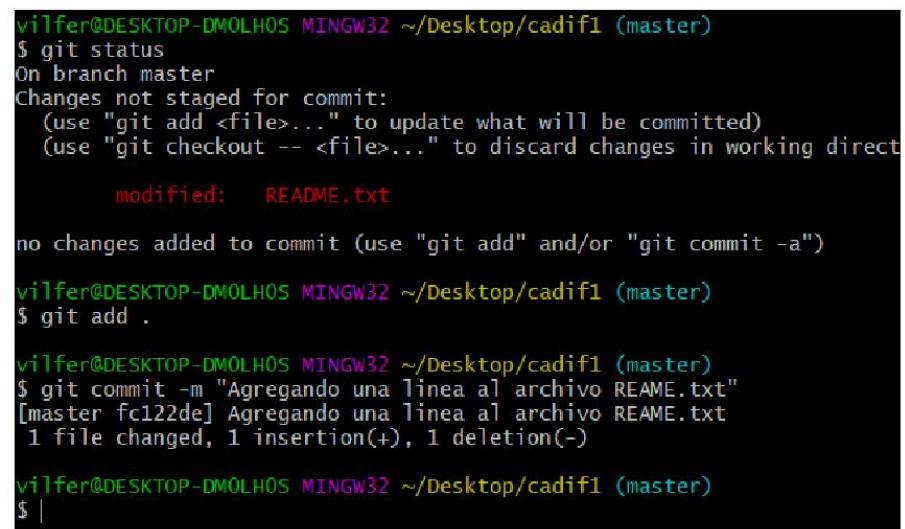
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   README.txt

vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ |

```

Al agregar una línea de texto al archivo README.txt, git catalogará al archivo como modificado. La imagen muestra la preparación y confirmación de los cambios al archivo



```

vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working direct
modified:   README.txt

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

vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ git add .

vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ git commit -m "Agregando una linea al archivo REAME.txt"
[master fc122de] Agregando una linea al archivo REAME.txt
 1 file changed, 1 insertion(+), 1 deletion(-)

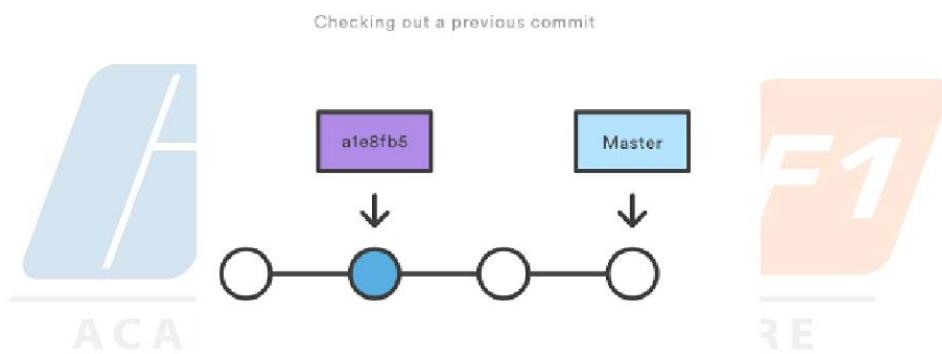
vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ |

```

## Capítulo 5. VIAJANDO EN EL TIEMPO

### 5.1.- Historial de Confirmaciones

El objetivo de cualquier sistema de control de versiones es registrar los cambios en su código. Esto le da el poder de volver al historial de su proyecto para ver quién contribuyó con qué, descubrir dónde se introdujeron los errores y revertir los cambios problemáticos. Pero, tener todo este historial disponible es inútil si no sabes cómo navegarlo. Ahí es donde entra el comando git log.



A veces necesitamos examinar la secuencia de commits (confirmaciones) que hemos realizado en la historia de un proyecto, para saber dónde estamos y qué estados hemos tenido a lo largo de la vida del repositorio. Esto lo conseguimos con el comando de Git "log".

Una vez que se tenga el listado en la terminal se pueden observar los mensajes de los commits y su identificador, pero también se podría obtener más información de un commit en concreto, sus modificaciones en el código, o tener la referencia para moverse hacia atrás en el histórico de commits.

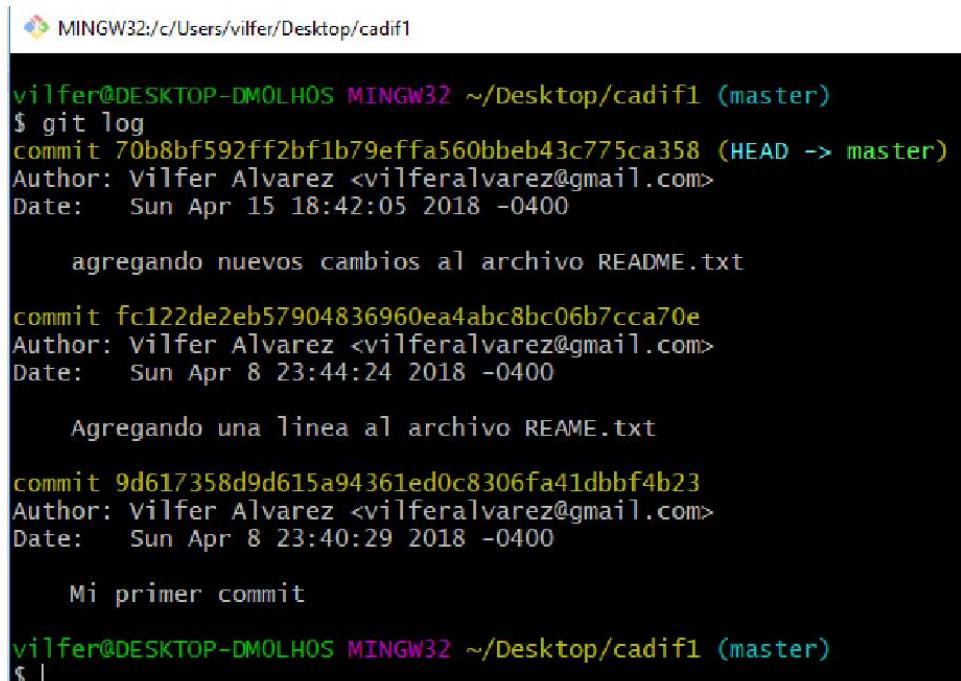
El uso del comando para mostrar el historial de confirmaciones del proyecto es el siguiente:

```
git log
```

Una variante para este comando seria:

```
git log -n
```

Donde el parámetro “n” es la cantidad de los últimos commit que desean ser consultados. En la imagen se muestra el uso del comando y la salida que presenta



```
MINGW32:/c/Users/vilfer/Desktop/cadif1
vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ git log
commit 70b8bf592ff2bf1b79effa560bbeb43c775ca358 (HEAD -> master)
Author: Vilfer Alvarez <vilferalvarez@gmail.com>
Date:   Sun Apr 15 18:42:05 2018 -0400

    agregando nuevos cambios al archivo README.txt

commit fc122de2eb57904836960ea4abc8bc06b7cca70e
Author: Vilfer Alvarez <vilferalvarez@gmail.com>
Date:   Sun Apr 8 23:44:24 2018 -0400

    Agregando una linea al archivo REAME.txt

commit 9d617358d9d615a94361ed0c8306fa41dbbf4b23
Author: Vilfer Alvarez <vilferalvarez@gmail.com>
Date:   Sun Apr 8 23:40:29 2018 -0400

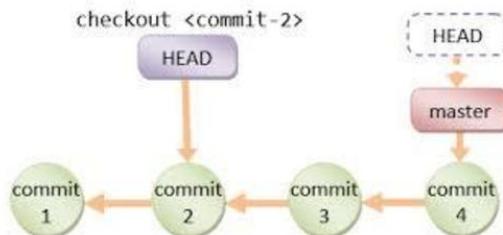
    Mi primer commit

vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ |
```

## 5.2.- Comando Checkout

Git provee al usuario la capacidad de literalmente “viajar en el tiempo”. No basta con conocer el historial de todos los cambios que se han realizado en el

proyecto, sino poder retroceder o adelantarse en el tiempo según sea necesario para beneficio del proyecto.



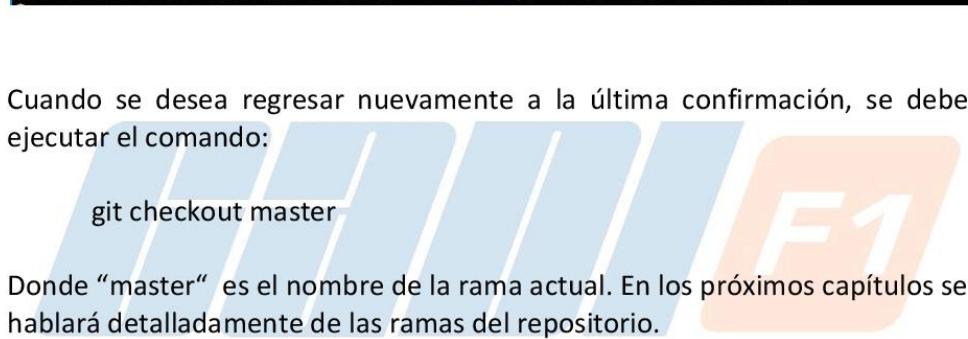
Git identifica a cada commit con un código inequívoco dentro del proyecto, de manera que se puede viajar a un commit en específico conociendo su código. El comando “git checkout” permite viajar en el tiempo de la siguiente manera:

`git checkout <codigoCommit>`

Donde el parámetro `códigoCommit` simplemente indica el código del commit al que se desea viajar.

Es válido acotar que al moverse en el tiempo, los commits no sufren ninguna modificación, es decir, la línea de tiempo sigue siendo la misma, así se puede regresar al último commit del proyecto sin problema.

En la imagen se muestra el uso del comando `checkout` de git para viajar al segundo commit del historial de confirmaciones del repositorio:



```

MINGW32:/c/Users/vilfer/Desktop/cadif1
vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ git checkout fc122de2eb57904836960ea4abc8bc06b7cca70e
Note: checking out 'fc122de2eb57904836960ea4abc8bc06b7cca70e'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

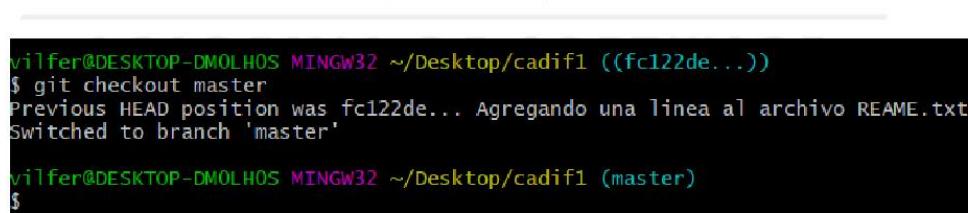
HEAD is now at fc122de... Agregando una linea al archivo REAME.txt
vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 ((fc122de...))

```

Cuando se desea regresar nuevamente a la última confirmación, se debe ejecutar el comando:

git checkout master

Donde “master” es el nombre de la rama actual. En los próximos capítulos se hablará detalladamente de las ramas del repositorio.



```

vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 ((fc122de...))
$ git checkout master
Previous HEAD position was fc122de... Agregando una linea al archivo REAME.txt
Switched to branch 'master'

vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ 

```

### 5.3.- Comando Reset

Los viajes en el tiempo que se han hecho hasta ahora no han realizado cambios ni eliminado ninguna de las confirmaciones anteriores. ¿Qué tal que lo que se desea es retroceder en el tiempo pero a su vez eliminar todas las confirmaciones hasta el punto hasta el cual retrocede?. Esto con el objetivo de no dejar rastros de esas confirmaciones y que no aparezca en el historial

Git provee un comando que permite hacer lo anteriormente descrito, es decir, retroceder a algún commit anterior pero con la peculiaridad de eliminar los commit anteriores. El comando es el siguiente:

```
git reset
```

El comando reset puede ser usado en alguno de estos modos:

--soft

No toca el archivo de índice o el árbol de trabajo en absoluto (pero restablece el encabezado a <commit>, al igual que todos los modos). Esto deja todos los archivos modificados "Cambios que se comprometerán", como lo indicaría el estado de git.

--mixed

Restablece el índice pero no el árbol de trabajo (es decir, los archivos modificados se conservan pero no se marcan para la confirmación) e informa lo que no se ha actualizado. Esta es la acción por defecto.

--hard

Restablece el índice y el árbol de trabajo. Cualquier cambio en los archivos rastreados en el árbol de trabajo desde <commit> se descarta

En la imagen se muestra el uso del comando reset

```
vilfer@DESKTOP-DMOLH0S MINGW32 ~/Desktop/cadif1 (master)
$ git reset --soft fc122de2eb57904836960ea4abc8bc06b7cca70e

vilfer@DESKTOP-DMOLH0S MINGW32 ~/Desktop/cadif1 (master)
$ git log
commit fc122de2eb57904836960ea4abc8bc06b7cca70e (HEAD -> master)
Author: Vilfer Alvarez <vilferalvarez@gmail.com>
Date:   Sun Apr 8 23:44:24 2018 -0400

        Agregando una linea al archivo REAME.txt

commit 9d617358d9d615a94361ed0c8306fa41dbbf4b23
Author: Vilfer Alvarez <vilferalvarez@gmail.com>
Date:   Sun Apr 8 23:40:29 2018 -0400

        Mi primer commit
```

## Capítulo 6. RAMAS. PARTE 1

### 6.1.- Definición

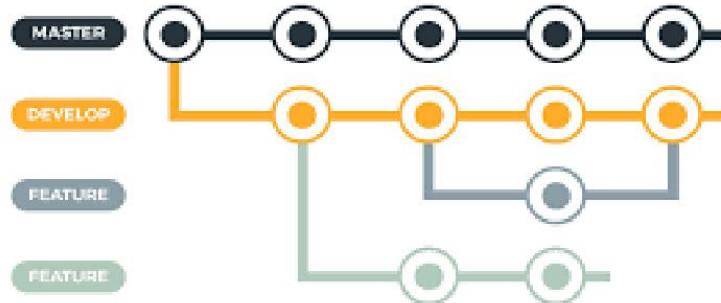
En el día a día del trabajo con Git una de las cosas útiles que se puede hacer es trabajar con ramas. Las ramas son caminos que puede tomar el desarrollo de un software, algo que ocurre naturalmente para resolver problemas o crear nuevas funcionalidades. En la práctica permiten que nuestro proyecto pueda tener diversos estados y que los desarrolladores sean capaces de pasar de uno a otro de una manera ágil.

Ramas podrás usar en muchas situaciones. Por ejemplo imagina que estás trabajando en un proyecto y quieres implementar una nueva funcionalidad en la que sabes que quizás tengas que invertir varios días. Mediante el uso de ramas se puede crear prácticamente un proyecto paralelo al proyecto central sin necesidad de alterar el código de este último, donde luego si es necesario, se pueden agregar esos nuevos cambios al proyecto.

Cuando inicializamos o clonamos un repositorio, siempre obtenemos una rama principal por defecto llamada master a no ser que se especifique lo contrario.

Esta rama es la principal de tu proyecto y a partir de la que podrás crear nuevas ramas cuando lo necesites.

En la imagen se muestra la rama master además de 3 ramas. La rama develop nace a partir del primer commit de la rama master.



Es común encontrar nombres de ramas con nombres de participantes del proyecto o cualquier otro identificador no muy específico. Las buenas prácticas indican que se deben tener por lo menos la rama central del proyecto (master), alguna rama de desarrollo (develop) en la cual se hagan las pruebas necesarias para los nuevos cambios que deban integrarse al proyecto y finalmente ramas que respondan a nombres de características que se estén desarrollando en el momento. Por ejemplo la rama “Validación de inicio de sesión” es una rama en la cual se está desarrollando el inicio de sesión de usuario y que luego será integrada a la rama Develop.

Entre los objetivos de usar ramas está:

- Agilizar el trabajo en equipo
- Separar el ambiente de desarrollo del de producción
- Llevar un mejor control y seguimiento de la realización de tareas dentro del equipo

### 6.2.- Creando Ramas

Se pueden observar las ramas presentes en el repositorio con el siguiente comando:

```
git branch
```

En la imagen se muestra el resultado de este comando en el repositorio.

Nota: El símbolo \* identifica la rama en la que se encuentra el repositorio actualmente.

```
vilfer@DESKTOP-DM0LH0S MINGW32 ~/Desktop/cadif1 (master)
$ git branch
* master

vilfer@DESKTOP-DM0LH0S MINGW32 ~/Desktop/cadif1 (master)
$
```

Para crear una nueva rama, se debe usar el comando anterior agregándole el nombre de la nueva rama como sigue:

```
git branch nuevarama
```

La rama “nuevarama” será una copia exacta de la rama desde la cual se generó. Todos los commits de la rama anterior pasan a la nueva rama.

```
vilfer@DESKTOP-DM0LH0S MINGW32 ~/Desktop/cadif1 (master)
$ git branch desarrollo

vilfer@DESKTOP-DM0LH0S MINGW32 ~/Desktop/cadif1 (master)
$ git branch
    desarrollo
* master

vilfer@DESKTOP-DM0LH0S MINGW32 ~/Desktop/cadif1 (master)
$ ~|
```

La idea de crear nuevas ramas es el de usarlas para alguna razón. Es por esto que se puede hacer uso del comando git checkout para moverse entre las ramas del repositorio.

Para moverse a la rama “desarrollo” se debe usar el siguiente comando:

```
git checkout desarrollo
```

Cuando se pasa a una nueva rama, ahora esta será la rama actual del repositorio y además se estará posicionado en el último commit de esa rama.

```
vilfer@DESKTOP-DMOLH05 MINGW32 ~/Desktop/cadif1 (master)
$ git checkout desarrollo
Switched to branch 'desarrollo'
M      README.txt

vilfer@DESKTOP-DMOLH05 MINGW32 ~/Desktop/cadif1 (desarrollo)
$ git branch
* desarollo
  master

vilfer@DESKTOP-DMOLH05 MINGW32 ~/Desktop/cadif1 (desarrollo)
$
```

El comando checkout tiene la posibilidad de permitirte crear una rama nueva y moverte a ella en un único paso. Para crear una nueva rama y situarte sobre ella tendrás que darle un nombre y usar el parámetro -b.

git checkout -b otrarama

### 6.3.- Eliminando Ramas

Las ramas en un repositorio son temporales, una vez utilizada y cumplida su función por lo general las ramas son eliminadas para mantener un orden dentro del repositorio. Eliminar una rama simplemente sería quitar la línea de tiempo paralela creada, no afectará en lo absoluto a las demás ramas.

Para proceder a la eliminación de una rama, se usa el siguiente comando:

```
git branch -d nombrerama
```

si la rama tiene cambios sin fusionar dará un error como el siguiente:

```
error: The branch desarollo is not an ancestor of your current HEAD.
```

If you are sure you want to delete it, run git branch -D desarollo.

```
vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$ git branch -d desarollo
Deleted branch desarollo (was fc122de).
vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (master)
$
```

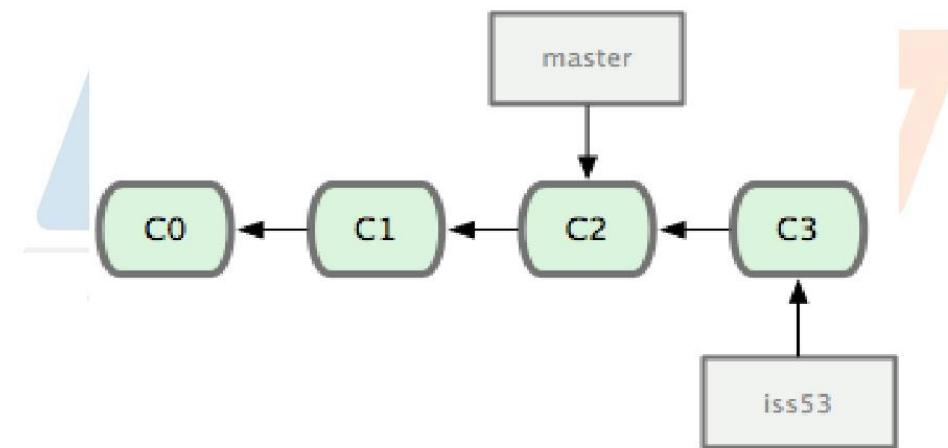


## Capítulo 7. RAMAS. PARTE 2

### 7.1.- Unión de Ramas

Se había dicho que las ramas serían la manera para trabajar en ciertas características o problemas se deseen solventar en el proyecto sin tocar la rama central (master), cambios que luego de ser verificados y aprobados, podrán ser integrados a la rama principal.

Suponga ahora el caso en que se crea una nueva rama a partir de la master y se hacen confirmaciones en esa rama. Visualmente pasaría algo como lo muestra la imagen:



Se puede observar como ahora la rama desarrollo tiene 1 commit más que la rama master. Si se revisa el historial de confirmaciones de la rama desarrollo se nota como git indica que el HEAD (último commit de la rama actual) está por encima del último commit de la rama master que es la rama original de la cual se generó la rama desarrollo.

```
vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (desarrollo)
$ git log
commit 7da7e71a40d307be483d4469c0bdfeee5d2aed88 (HEAD -> desarrollo)
Author: Vilfer Alvarez <vilferalvarez@gmail.com>
Date:   Wed Apr 18 01:40:02 2018 -0400

    Agregando una tercera linea al archivo README.txt

commit fc122de2eb57904836960ea4abc8bc06b7cca70e (master)
Author: Vilfer Alvarez <vilferalvarez@gmail.com>
Date:   Sun Apr 8 23:44:24 2018 -0400

    Agregando una linea al archivo REAME.txt

commit 9d617358d9d615a94361ed0c8306fa41dbbf4b23
Author: Vilfer Alvarez <vilferalvarez@gmail.com>
Date:   Sun Apr 8 23:40:29 2018 -0400

    Mi primer commit

vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/cadif1 (desarrollo)
$ |
```

¿Cómo unir los cambios de una rama con otra?

En Git se pueden unir ramas con el comando merge como sigue:

Git merge ramaAFusionar

La unión de la ramaAFusionar se va a realizar con la rama en la cual se está posicionado actualmente.

En la fusión de ramas generalmente ocurre uno de los siguientes casos

-Fast-Forward: No existen conflictos de cambios similares entre las ramas

-Manual Merge: Existen conflictos en cambios hechos en ambas ramas sobre los mismos archivos.

El Fast-Forward ocurre Cuando la rama que se desea fusionar es hija de la rama actual, la cual no ha sufrido cambios. En este caso simplemente se fusionan las ramas, se agregan los nuevos commit a la rama actual y se posiciona el HEAD en el último nuevo commit. Es decir, la rama actual queda

ahora con los mismos commit que la rama que se fusionó. Siguiendo con el ejemplo de la rama desarrollo:

```
vilfer@DESKTOP-DM0LHOS MINGW32 ~/Desktop/cadif1 (master)
$ git merge desarrollo
Updating fc122de..7da7e71
Fast-forward
 README.txt | 4 +---
 1 file changed, 3 insertions(+), 1 deletion(-)

vilfer@DESKTOP-DM0LHOS MINGW32 ~/Desktop/cadif1 (master)
$
```

## 7.2.- Resolviendo Conflictos

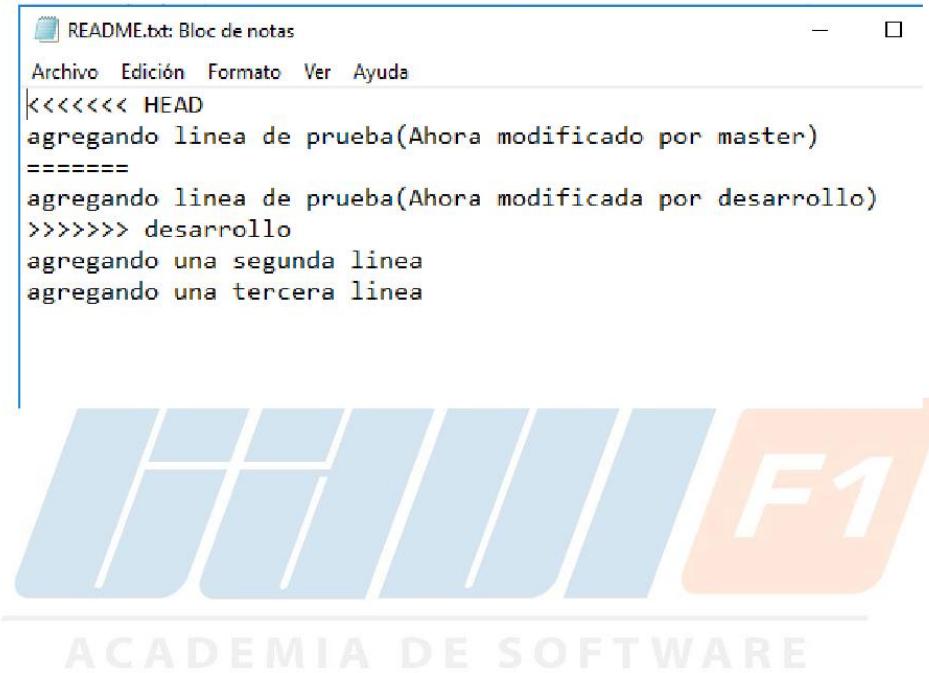
El segundo caso que puede ocurrir al hacer una fusión entre ramas, es que exista confirmaciones de cambios en ambas ramas que hayan afectado a las mismas líneas de uno o más archivos. En este caso Git muestra la advertencia y las líneas en las cuales existen conflictos, así que se debe proceder manualmente a solventar el problema para luego hacer un nuevo commit.

Ejemplo: En la rama master y en la rama desarrollo se modificó la primera línea del archivo README.txt. Al tratar de hacer una unión de desarrollo con master sucede lo que muestra la imagen:

```
vilfer@DESKTOP-DM0LHOS MINGW32 ~/Desktop/cadif1 (master)
$ git merge desarrollo
Auto-merging README.txt
CONFLICT (content): Merge conflict in README.txt
Automatic merge failed; fix conflicts and then commit the result.

vilfer@DESKTOP-DM0LHOS MINGW32 ~/Desktop/cadif1 (master|MERGING)
$ |
```

Git escribe en las líneas de los archivos donde existe conflicto. Es por eso que se deben revisar esos archivos, modificarlos y dejar lo que se conservará y confirmas cambios. La imagen muestra el archivo README.txt de ejemplo:



```
 README.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
<<<<< HEAD
agregando linea de prueba(Ahora modificado por master)
=====
agregando linea de prueba(Ahora modificada por desarrollo)
>>>> desarrollo
agregando una segunda linea
agregando una tercera linea
```

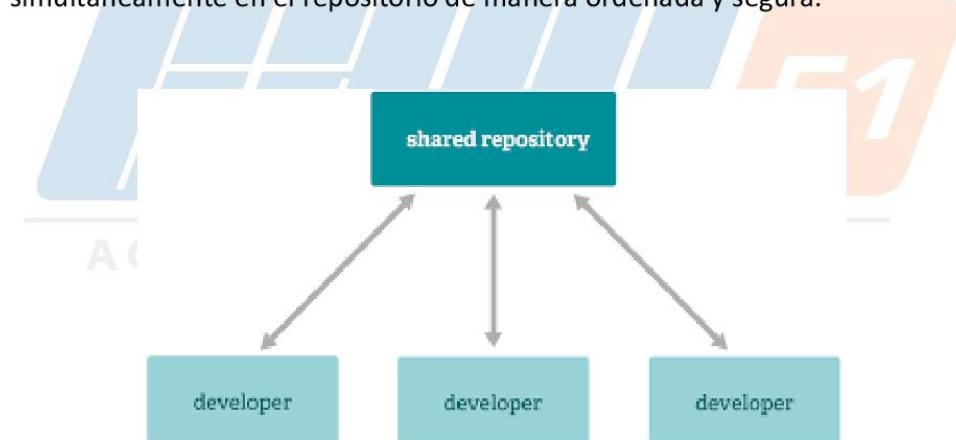


## Capítulo 8. GITHUB. PARTE 1

### 8.1.- Plataformas Para el Trabajo en Equipo

El trabajo en equipo es fundamental para lograr los objetivos de algún proyecto de manera efectiva, además, que cada vez es más común encontrar equipos de trabajo donde existe una separación física de sus integrantes, lo que se convierte en un reto para las nuevas organizaciones. Es por ello que se ha pensado en el desarrollo de ciertas plataformas que apoyen la integración del trabajo en equipo, facilitando a sus miembros la realización de sus actividades.

Estas plataformas permiten manejar repositorios git pero ahora de manera compartida con múltiples usuarios, donde todos ellos pueden hacer cambios simultáneamente en el repositorio de manera ordenada y segura.



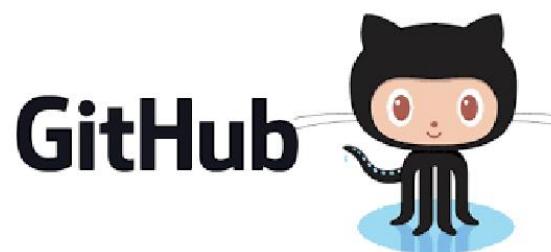
Entre las plataformas más utilizadas hoy en día que usan git como controlador de versiones están:

- Github
- Bitbucket
- Gitlab
- Coding



### 8.2.- Github

GitHub es una forja (plataforma de desarrollo colaborativo) para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de código fuente de programas de computadora. El software que opera GitHub fue escrito en Ruby on Rails. Desde enero de 2010, GitHub opera bajo el nombre de GitHub, Inc. Anteriormente era conocida como Logical Awesome LLC. El código de los proyectos alojados en GitHub se almacena típicamente de forma pública, aunque utilizando una cuenta de pago, también permite hospedar repositorios privados.

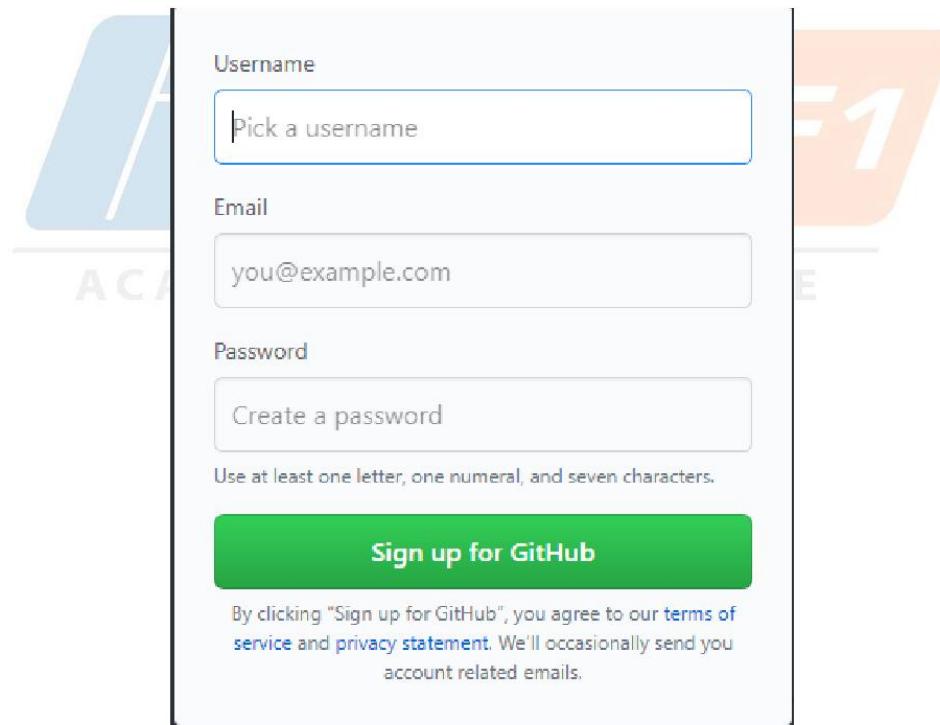


GitHub es el host más grande para los repositorios de Git, y es el punto central de colaboración para millones de desarrolladores y proyectos. Un

gran porcentaje de todos los repositorios de Git están alojados en GitHub, y muchos proyectos de código abierto lo usan para el alojamiento de Git, el seguimiento de problemas, la revisión de códigos y otras cosas. Entonces, si bien no es una parte directa del proyecto de código abierto de Git, hay muchas posibilidades de que desee o necesite interactuar con GitHub en algún momento mientras usa Git profesionalmente.

### 8.3.- Usando Github

Lo primero es crear una cuenta en GitHub. Para eso debe ir al sitio oficial <https://github.com/> donde podrá encontrar un formulario para crear una nueva cuenta. Se le solicitará información personal como nombre, email y password.



En la sección de “mi perfil” se puede culminar de añadir la información personal relevante que le sea solicitada. Ya en el dashboard de GitHub se pueden observar características como una barra la búsqueda tanto de personas como de proyectos en los cuales se deseé colaborar.



En la parte derecha de la pantalla inicial de GitHub, muestra un listado con los repositorios en los cuales se ha trabajado. Esta sección debe estar vacía cuando se es usuario nuevo de la plataforma. De igual manera, se encuentra un botón para crear nuevos repositorios.

A screenshot of the GitHub dashboard showing the user's repository list. The interface includes a sidebar with the letters "ACA" and "ed". The main area has a title "Your repositories 9" with a green "New repository" button. Below it is a search bar with placeholder text "Find a repository...". Underneath are filter buttons for "All", "Public", "Private", "Sources", and "Forks", with "All" being underlined. The repository list shows 2449 entries, each with a small icon and a blue link. The entries listed are: "cadif1", "viferalvarez.github.io", "prueba", "asd", "pagina", and "luisffg24/ucline". At the bottom of the list is a "Show more" link.

## Capítulo 9. GITHUB. PARTE 2

### 9.1.- Creando Repositorios Remotos

Hasta ahora se han manejado repositorios solamente de manera local (Computador personal), de manera que el repositorio es accesible solo por un usuario que es el dueño del mismo.

Los repositorios remotos son versiones de tu proyecto que se encuentran alojados

Entre las ventajas de los repositorios remotos está:

- Tener copias del trabajo en la web, lo que las hace accesibles en cualquier momento.
- Gestión de proyectos en equipo
- Control de actividades en el desarrollo
- Colaboración en proyectos de terceros en que usen Git como sistema de control de versiones.
- Crear portafolios de trabajos realizados por el usuario

Para crear un nuevo repositorio en GitHub, luego de iniciar sesión con el usuario respectivo, debe hacer click en la sección de repositorios en el botón “new repository” el cual es un enlace a la ruta <https://github.com/new>. Allí se debe colocar el nombre del repositorio, una breve descripción, tipo de repositorio (público o privado). En la imagen se muestra la creación del repositorio Cadif1Project.

Owner                          Repository name

 vilferalvarez / Cadif1Project 

Great repository names are short and memorable. Need inspiration? How about [improved-octo-train](#).

Description (optional)

Primer repositorio de prueba

 Public  
Anyone can see this repository. You choose who can commit.

 Private  
You choose who can see and commit to this repository.

Initialize this repository with a README  
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None | 

**Create repository**

Allí se debe colocar el nombre del repositorio, una breve descripción, tipo de repositorio (público o privado). El archivo README es importante, por lo cual se debe habilitar la opción para crear con el proyecto un archivo README, el cual servirá para especificar información del repositorio:

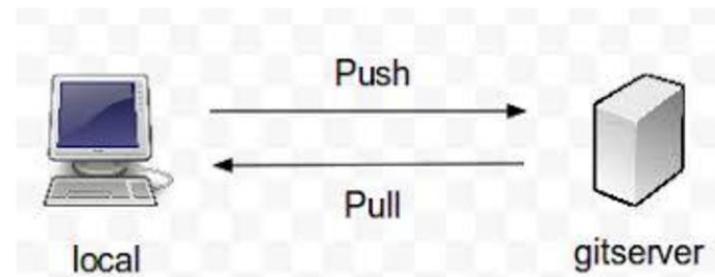
- Owner: Dueño del proyecto
- ProjectName: Nombre del proyecto
- Description: Descripción básica del repositorio
- Public ó Private: Especifica la seguridad del repositorio.

Por defecto los repositorios en GitHub son públicos.

## 9.2.- Repositorio Remoto& Repositorio Local

Una vez se dispone de un repositorio local y uno remoto en los que trabajar, se debe aprender a subir los cambios que se han hecho en el repositorio local al repositorio remoto. Se deben asociar los repositorios para que la

plataforma redireccione hacia el repositorio remoto los cambios que se hacen en local.



Git provee de el comando “git remote -v” para verificar el o los repositorios remotos a los cuales está asociado el repositorio local. En la imagen se muestra la salida de un repositorio local el cual no está asociado a ningún repositorio remoto.

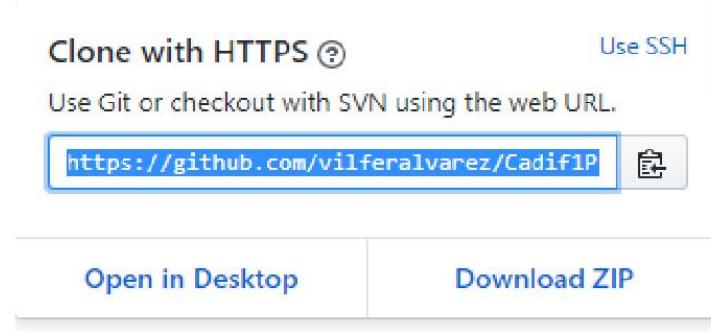
```
vilfer@DESKTOP-DM0LH0S MINGW32 ~/Desktop/cadif1 (master)
$ git remote -v

vilfer@DESKTOP-DM0LH0S MINGW32 ~/Desktop/cadif1 (master)
$ |
```

El repositorio remoto tiene asociado una url dentro de la plataforma con el siguiente patrón:

<https://github.com/vilferalvarez/Cadif1Project.git>

La URL se crea al inicializarse el proyecto y se puede encontrar en el botón “clone o download de la ventana principal del repositorio:



Para establecer la conexión entre los 2 repositorios, se debe usar una variante del comando remote" como sigue:

Git remote add nombre url

Donde "nombre" es el nombre para identificar el repositorio remoto y la "url" para especificar la dirección exacta de ese repositorio en la web. En la imagen siguiente se muestra la conexión con el repositorio creado en prácticas anteriores:

```
vilfer@DESKTOP-DMOLH05 MINGW32 ~/Desktop/cadif1 (master)
$ git remote add origin https://github.com/vilferalvarez/Cadif1Project.git
vilfer@DESKTOP-DMOLH05 MINGW32 ~/Desktop/cadif1 (master)
$ git remote -v
origin https://github.com/vilferalvarez/Cadif1Project.git (fetch)
origin https://github.com/vilferalvarez/Cadif1Project.git (push)
```

Para subir los cambios al repositorio remoto solo se debe tener alguna confirmación preparada en Local y proceder a usar el comando:" push". El comando push es el encargado de enviar los cambios de Local al repositorio remoto.

git push [nombre-remoto][nombre-rama].

Si se desea enviar la rama maestra (master) al servidor origen (origin), se debe usar el siguiente comando:

```
git push origin master
```

### 9.3.- Clonando Repositorios

Es muy común tener la necesidad de obtener algún proyecto de un repositorio remoto que ya esté adelantado, solo para agregarlo al ambiente local. Para obtener un repositorio remoto de GitHub sin tener ningún repositorio local, el usuario debe colocarse en la carpeta donde necesita que se cree el repositorio y usar el comando “clone” de git.

Para clonar un repositorio se necesita la URL(La misma utilizada en el capítulo anterior) del repositorio. La estructura del comando clone es la siguiente:

```
Git clone url
```

```
vilfer@DESKTOP-DMOLHOS MINGW32 ~/Desktop/proyecto2
$ git clone https://github.com/vilferalvarez/Cadif1Project.git
Cloning into 'Cadif1Project'...
```

Los repositorios clonados tendrán la misma estructura y características del repositorio remoto, eso incluye commits y ramas.

## Capítulo 10. GITHUB. PARTE 3

### 10.1.- Contribuyendo en Proyectos de Terceros

Uno de los motivos de la creación de GitHub era la de crear un ambiente de colaboración entre desarrolladores de todo el mundo. GitHub le da la posibilidad al usuario de revisar repositorios de terceros y contribuir con piezas de código que sean necesarias. Hasta el momento en este material se ha trabajado con un repositorio local(propio del usuario). Pero ¿Qué tal si lo que se desea es contribuir en un proyecto en el cual no tenemos permisos para modificarlo? Es allí donde aparece el término “Fork”.

La palabra fork se traduce al castellano, dentro del contexto que nos ocupa, como bifurcación. Cuando se hace un fork de un repositorio, se hace una copia exacta en crudo (en inglés “bare”) del repositorio original que se puede utilizar como un repositorio git cualquiera. Despues de hacer Fork se tendrán dos repositorios git idénticos pero con distinta URL. Justo después de hacer el Fork, estos dos repositorios tienen exactamente la misma historia, son una copia idéntica. Finalizado el proceso, se tendrán dos repositorios independientes que pueden cada uno evolucionar de forma totalmente autónoma. De hecho, los cambios que se hacen el repositorio original NO se transmiten automáticamente a la copia (Fork). Esto tampoco ocurre a la inversa: las modificaciones que se hagan en la copia (Fork) NO se transmiten automáticamente al repositorio original.



Cuando se realiza el Fork de un repositorio, se crea un repositorio idéntico en la cuenta del usuario que ha hecho el Fork. Para hacer un Fork se debe dirigir al repositorio al que desea contribuir y hacer click en el icono de Fork de la

parte superior derecha. Automáticamente se creará en su cuenta un repositorio con el mismo nombre

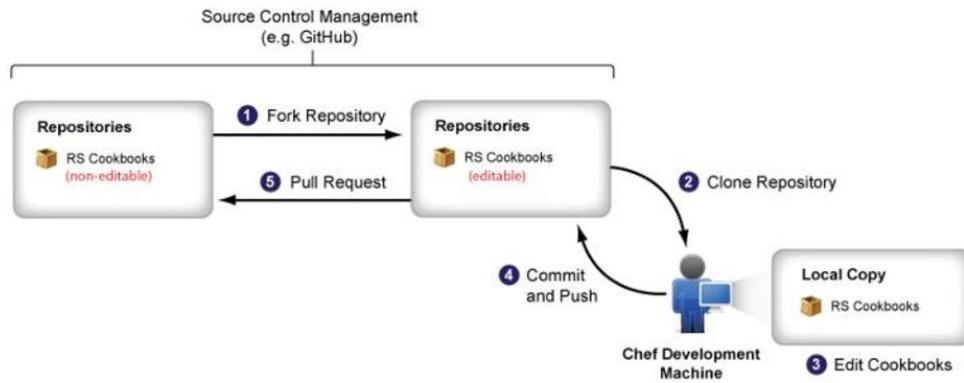


Los repositorios los cuales son producto de un Fork y que ahora están en la cuenta del usuario contribuyente con el proyecto, será un repositorio mas en su cuenta, es decir, puede clonarlo y comenzar a trabajar sobre el, posteriormente subir cambios a este proyecto copia del original.

### 10.2.- Pull Request

Los cambios hechos en los proyectos bifurcados de otros repositorios, solo serán visibles en la copia del repositorio y no en la versión original del mismo, esto quiere decir que hasta ahora se ha colaborado propiamente con el proyecto. Esta es una de las diferencias notables con el procedimiento de clonado, ya que al hacer Fork no se pueden hacer cambios directamente sobre el proyecto original, lo que da más seguridad a la integridad del proyecto.

La imagen muestra el flujo de trabajo para contribuir con un repositorio de un tercero:



Los Pull Request serán peticiones que se hacen al dueño del proyecto para que integre los cambios que ha realizado un tercero sobre una copia de su repositorio. Será decisión del dueño del repositorio si integra los cambios o no. En el momento en que se hagan cambios locales y se suban al repositorio copia (Fork), GitHub habilita un botón identificado “Pull Request” el cual será el encargado de enviar la petición de integración de cambios al dueño del repositorio original.

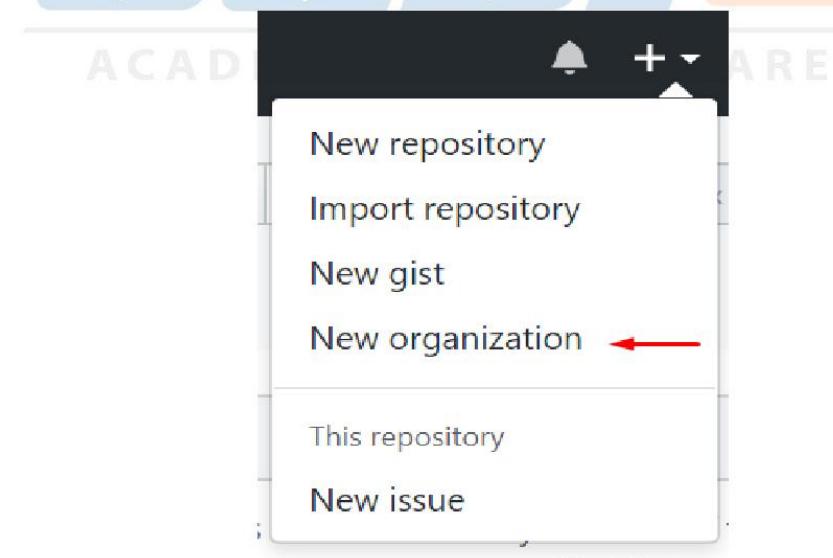
ACADEMIA DE SOFTWARE

## Capítulo 11. GITHUB. PARTE 4

### 11.1.- Organizaciones

Además de las cuentas de usuario, GitHub tiene Organizaciones. Al igual que las cuentas de usuario, las cuentas de organización tienen un espacio donde se guardarán los proyectos, pero en otras cosas son diferentes. Estas cuentas representan un grupo de personas que comparte la propiedad de los proyectos, y además se pueden gestionar estos miembros en subgrupos. Normalmente, estas cuentas se usan en equipos de desarrollo de código abierto (por ejemplo, un grupo para “perl” o para “rails) o empresas (como sería ``google” o “twitter”).

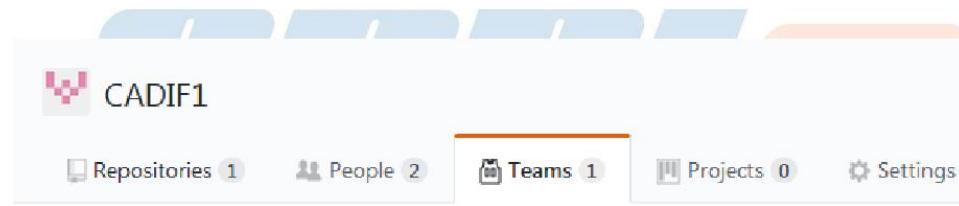
Para crear una nueva organización, simplemente se debe pulsar en el icono “+” en el lado superior derecho y selecciona “New organization”. En primer lugar tienes que decidir el nombre de la organización y una dirección de correo que será el punto principal de contacto del grupo. A continuación puedes invitar a otros usuarios a que se unan como co-propietarios de la cuenta.



En las organizaciones serán creados los repositorios en los cuales trabajarán los miembros de la organización. Los miembros podrán realizar cambios directamente en los repositorios según los permisos que se les sean asignados, demostrando una vez más la seguridad a nivel de acceso que posee la plataforma.

### 11.2.- Equipos

Los Team (equipos) dentro de una organización ayudan a dividir de manera organizada las responsabilidades que pueden tener los miembros de la organización para el aporte a los repositorios. En una organización se pueden crear los equipos que se consideren necesarios y además agregar en cada equipo a los miembros de la organización que se consideren. En la imagen se muestra la organización CADIF1 en la cual existe un Team.



Para crear un nuevo equipo dentro de la organización, se debe ir a la pestaña “Team” y hacer click en el botón “New Team”. Se debe colocar un nombre y una breve descripción al equipo

## Create new team

### Team name

What is the name of this team?

You'll use this name to mention this team in conversations.

### Description

What is this team all about?

### Parent team

Select parent team ▾

### Team visibility

**Visible** Recommended

A visible team can be seen and @mentioned by every member of this organization.

**Secret**

A secret team can only be seen by its members.

**Create team**

## Capítulo 12. GITHUB PAGES

### 12.1.- Pagina Personal

GitHub permite crear páginas personales para tu organización y/o proyecto, sirviendo de Hosting para almacenar sitios web sencillos como presentación tanto personal como de los proyectos realizados, a manera de crear un portafolio de proyectos realizados. Basta con conocer un poco de HTML para poder crear en cualquier repositorio un sitio web.

Como las GitHub Pages están asociadas a los repositorios creados en GitHub, crear una nueva página no es tan complicado. Se debe distinguir la creación de GitHub Pages tanto para presentación personal del usuario, de las páginas diseñadas para proyectos.

- GitHub Pages personales: Creadas para brindar información del usuario.
- GitHub Pages para proyectos Creadas para brindar información e interactividad para los usuarios con el proyecto que se ha creado o se está creado en ese repositorio.

Se debe destacar que las páginas deben ser estáticas, es decir, sin uso de lenguajes de programación del lado del servidor.

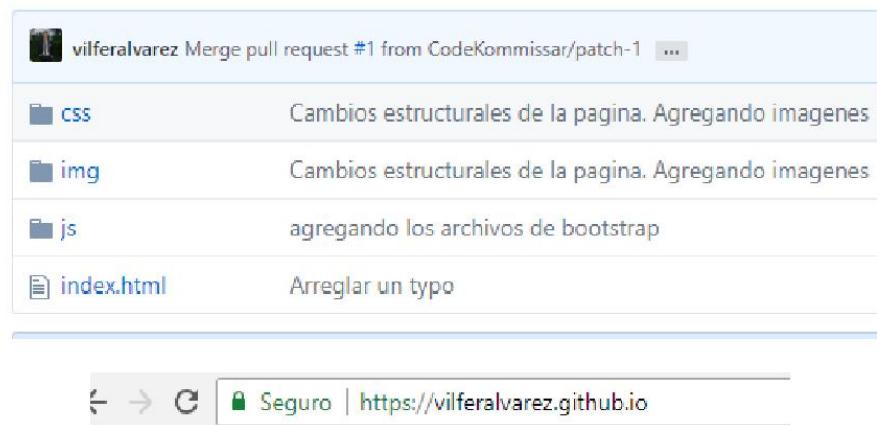
Para crear una página personal dentro de GitHub, basta con crear un nuevo repositorio con el nombre del usuario y crear en este los archivos necesarios para la visualización de la página (HTML, CSS y JS). El nombre del repositorio debe ser con el siguiente patrón:

nombreUsuario.github.io

El nombre de usuario debe ser el nombre con el que se creó la cuenta de github.

Los archivos a subir en este repositorio deben ser archivos con extensiones .html, css y js, ya que son interpretados únicamente por el navegador. La visualización de la página se hará mediante el navegador usando la siguiente url:

<https://nombreUsuario.github.io/>



## ACADEMIA DE SOFTWARE

### 12.2.- Páginas Para Repositorios

En el capítulo anterior se mostró la creación de páginas personales para los usuarios de GitHub, páginas únicas identificadas por nombres de usuarios. En este capítulo se entenderá la creación de páginas para cada uno de los proyectos alojados en una cuenta GitHub

Es importante generar de manera visual y agradable información relacionada con el proyecto, para así mostrar el funcionamiento o lo objetivos del desarrollo que se lleva a cabo, para facilitar la contribución de otros usuarios GitHub al repositorio.

Para crear una GitHub Page de cualquier proyecto, se debe crear un nuevo proyecto o utilizar uno ya existente, sabiendo que debe estar clonado

localmente para poder hacer las modificaciones respectivas. Se debe crear una rama llamada gh-pages

`git branch gh-pages`

Esta es la rama que debe albergar los archivos de la página. Los cambios se deben subir a esta rama mediante el comando `git push origin gh-pages`.

Para acceder la página del repositorio, basta con colocar la url que siga la siguiente estructura:

`nombreUsuario.github.io/nombreRepositorio`

Donde “nombreUsuario” es el nombre de usuario registrado en GitHub y “nombreRepositorio” es el nombre del repositorio en el cual hemos subido los archivos a su rama gh-pages



## Capítulo 13. CLIENTES GIT

### 13.1.- Introducción

Git es, sin duda, el sistema de control de versiones más utilizado. La mayoría de los proyectos por la mayor cantidad de empresas se ejecutan en repositorios de Git. Git no solo hace que sea más fácil codificar su aplicación, sino que también le ayuda con algunas características de colaboración para que pueda trabajar de manera eficiente con su equipo en un proyecto.

Git es también una habilidad que todo desarrollador debe tener. Comprender todas las operaciones y comandos puede ser difícil para los principiantes a veces. Pero un grupo de herramientas lo hace más simple al proporcionar alternativas de GUI a la línea de comando de Git.

### 13.2.- Smartgit

Entre los clientes gráficos Git más populares se encuentran los siguientes:

Github Desktop ([desktop.github.com](http://desktop.github.com)) Windows(64 Bits) y Mac

SmartGit ([www.syntevo.com/smartgit/](http://www.syntevo.com/smartgit/)) Windows, Linux y Mac

GitKraken ([www.gitkraken.com/](http://www.gitkraken.com/)) Windows, Linux y Mac

Hay una gran variedad de Clientes Git que se pueden utilizar, todo va a depender del gusto del usuario y el soporte que tenga para su sistema operativo.

SmartGit es probablemente uno de los más útiles. La herramienta le proporciona una interfaz de usuario maravillosa que le permite administrar su código sin escribir ningún comando en la ventana de comandos.



Puede crear repositorios nuevos, agregar repositorios locales y realizar la mayoría de las operaciones de Git desde la interfaz de usuario. SmartGit es un buen cliente para rastrear tus cambios y realizar operaciones de Git sobre la marcha. El programa se puede encontrar en el sitio oficial de sysntevo <https://www.syntevo.com/smartgit/>. Existe una versión portable para Windows y su instalación es igual a cualquier programa para Windows. En el proceso de instalación se preguntar email y nombre de usuario que serán usados para los commits.

Se puede obtener desde el sitio oficial <https://www.syntevo.com/smartgit/>

ACADEMIA DE SOFTWARE



**Get your commit done.**

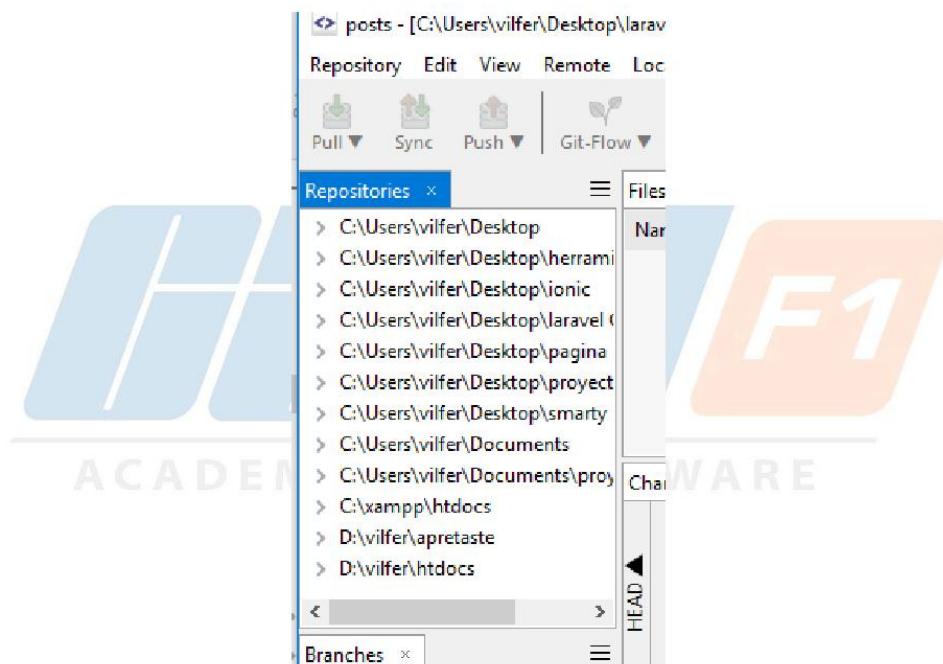
SmartGit is a Git client with support for [GitHub Pull Requests+Comments](#) and [SVN](#). It runs on Mac OS X, Windows and Linux.

[Purchase](#)

[Download](#)

### 13.3.- Interfaz Gráfica Smartgit

Al iniciar SmartGit, automáticamente reconocerá los repositorios previamente usados en ese computador y los mostrará en la parte izquierda de la ventana principal del programa. Simplemente se hace click en el repositorio a utilizar. De igual manra se pueden crear nuevos repositorios en el menú principal Repository -> Add or Create . Para crear un nuevo repositorio debe seleccionar una carpeta vacía de su proyecto.



Luego de crear o seleccionar un repositorio, tenemos a la mano sus archivos y de manera gráfica los comandos de Git en la barra superior de SmartGit. Se pueden hacer las mismas funciones que se realizan desde la consola de Git.

