

Git

Control de versiones

Algoritmos y Estructuras de Datos
(basado en el taller de la ComCom)

DC, FCEyN, UBA

September 8, 2023

Trabajando en grupo



tp.java

Trabajando en grupo



tp.java



tp_corregido.java

Trabajando en grupo



tp.java



tp_corregido.java



tp_corregido2.java

Trabajando en grupo



tp.java



tp_corregido.java



tp_corregido2.java



tp_FINAL.java

Trabajando en grupo



tp.java



tp_corregido.java



tp_corregido2.java



tp_FINAL.java



tp_FINAL_posta.java

Trabajando en grupo



tp.java



tp_corregido.java



tp_corregido2.java



tp_FINAL.java



tp_FINAL_posta.java



Controlando la situación...

Sistemas de Control de Versiones

Programas que permiten **manejar los cambios** de los archivos en un proyecto a lo largo del tiempo, a través de un **seguimiento** a sus modificaciones.

Controlando la situación...

Sistemas de Control de Versiones

Programas que permiten **manejar los cambios** de los archivos en un proyecto a lo largo del tiempo, a través de un **seguimiento** a sus modificaciones.

Esto permite, principalmente, tres cosas:

- 1 Volver a una *versión* anterior en caso de que nos equivoquemos.

Controlando la situación...

Sistemas de Control de Versiones

Programas que permiten **manejar los cambios** de los archivos en un proyecto a lo largo del tiempo, a través de un **seguimiento** a sus modificaciones.

Esto permite, principalmente, tres cosas:

- ➊ Volver a una *versión* anterior en caso de que nos equivoquemos.
- ➋ Comparar distintas etapas del proyecto.

Controlando la situación...

Sistemas de Control de Versiones

Programas que permiten **manejar los cambios** de los archivos en un proyecto a lo largo del tiempo, a través de un **seguimiento** a sus modificaciones.

Esto permite, principalmente, tres cosas:

- ➊ Volver a una *versión* anterior en caso de que nos equivoquemos.
- ➋ Comparar distintas etapas del proyecto.
- ➌ Compartir el código con otras personas y resolver *conflictos*.

Controlando la situación...

Sistemas de Control de Versiones

Programas que permiten **manejar los cambios** de los archivos en un proyecto a lo largo del tiempo, a través de un **seguimiento** a sus modificaciones.

Esto permite, principalmente, tres cosas:

- ➊ Volver a una *versión* anterior en caso de que nos equivoquemos.
- ➋ Comparar distintas etapas del proyecto.
- ➌ Compartir el código con otras personas y resolver *conflictos*.

¿Permite manejar los cambios de *cualquier* archivo?

¿Qué es Git?

Git

Sistema de control de versiones **distribuido y de código abierto**, con énfasis en el **rendimiento** (sirve hasta para manejar proyectos muy grandes), **seguridad** y **flexibilidad**.

¿Qué es Git?

Git

Sistema de control de versiones **distribuido y de código abierto**, con énfasis en el **rendimiento** (sirve hasta para manejar proyectos muy grandes), **seguridad** y **flexibilidad**.

Proveedores que lo implementan:



GitHub



Bitbucket



GitLab

¿Qué es Git?

GIT - the stupid content tracker

"git" can mean anything, depending on your mood.

- **random three-letter combination that is pronounceable**, and not actually used by any common UNIX command. The fact that it is a mispronunciation of "get" may or may not be relevant.
- **stupid**. contemptible and despicable. simple. Take your pick from the dictionary of slang.
- **"global information tracker"**: you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.
- **"goddamn idiotic truckload of sh*t"**: when it breaks

¿Qué es Git?

GIT - the stupid content tracker

"git" can mean anything, depending on your mood.

- **random three-letter combination that is pronounceable**, and not actually used by any common UNIX command. The fact that it is a mispronunciation of "get" may or may not be relevant.
- **stupid**. contemptible and despicable. simple. Take your pick from the dictionary of slang.
- **"global information tracker"**: you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.
- **"goddamn idiotic truckload of sh*t"**: when it breaks

- Mensaje del commit de Linus Torvalds, su creador, agregando el README al repositorio Git de Git

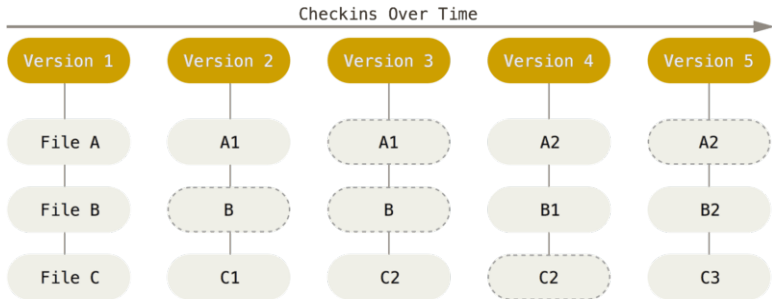
¿Cómo funciona?

Cada proyecto se almacena en un **repositorio**: directorio que contiene todos sus archivos (e.g. “taller1”, “taller2”, ...).

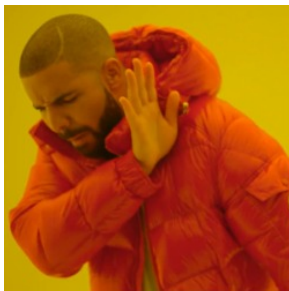
Este repositorio puede estar almacenado tanto de manera *local*, como *remota* (en un servidor web, e.g. GitLab).

¿Cómo funciona?

- 1 Piensa a los datos dentro del repositorio como una serie de 'fotos' (*snapshots*) del sistema de archivos.
- 2 Con cada `commit`, o cada vez que se guarda el *estado* del repo, saca una foto al mismo y guarda una *referencia* a ella.
- 3 Si los archivos no cambian, no saca ninguna foto: solo se toma la última.



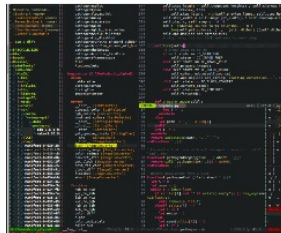
Vamos a trabajar con la consola !



Avs REASON Avs KICKSTART Avs CHANGING Avs CONTRIBUTING
Avs W... Condu... Integration

Nome	Last commit
img1	Update 6 files
img2	Update 6 files

Lenta
interfaz
web



Obteniendo un repositorio

```
git clone [URL]
```

Para obtener una copia local de un repositorio remoto, utilizamos el comando `git clone [URL]` (sin los corchetes) sobre la terminal.

Obteniendo un repositorio

```
git clone [URL]
```

Para obtener una copia local de un repositorio remoto, utilizamos el comando `git clone [URL]` (sin los corchetes) sobre la terminal.

Esto procederá a descargar el repositorio (con todos sus archivos) en una carpeta en nuestro disco.

A partir de ahora, cualquier cambio realizado sobre esos archivos será registrado por Git.

```
git clone  
https://gitlab.com/usuario/algo-2023c2-individual
```

Preparando cambios

```
git add [archivo]
```

Una vez que tenemos cambios hechos, tenemos que marcarlos como preparados antes de confirmarlos. En la jerga de Git, decimos que pasamos los cambios a *staged*.

- 1 Creamos/modificamos el archivo en cuestión.
- 2 Ejecutamos `git add [nombre del archivo]`.

Preparando cambios

```
git add [archivo]
```

Una vez que tenemos cambios hechos, tenemos que marcarlos como preparados antes de confirmarlos. En la jerga de Git, decimos que pasamos los cambios a *staged*.

- 1 Creamos/modificamos el archivo en cuestión.
- 2 Ejecutamos `git add [nombre del archivo]`.

Se puede tratar de un archivo ya existente en el repositorio (y que ustedes modificaron), como uno que previamente no existía.

```
git add taller1/Funciones.java
```

Configuraciones iniciales

Tu identidad

Es importante establecer nuestro **nombre y email**, ya que estos van a ir asociados con los cambios que hagamos:

```
git config user.name "Linus Torvalds"  
git config user.email linus@linux-foundation.org
```

Agregando la opción `--global` a los comandos anteriores, se establecen estas configuraciones para *todos* los repositorios en la computadora.

Confirmando cambios

git commit

Una vez que tenemos ciertos cambios en *staged*, podemos confirmarlos ejecutando `git commit -m [mensaje]`.

Donde [mensaje] es una breve descripción de los cambios que acabamos de confirmar.

Confirmando cambios

git commit

Una vez que tenemos ciertos cambios en *staged*, podemos confirmarlos ejecutando `git commit -m [mensaje]`.

Donde [mensaje] es una breve descripción de los cambios que acabamos de confirmar.

```
git commit -m "Primera versión del primer taller"
```

Importante: estos cambios *aún* no están reflejados en el repositorio remoto (stay tuned)

¡No seas vago con los mensajes!



	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

Fuente: <https://xkcd.com/1296/>

¿Está preparado, confirmado o ninguna de las dos?

git status

Untracked, modified, staged, committed

Las modificaciones que hacemos pueden estar en **4 estados** distintos:

¿Está preparado, confirmado o ninguna de las dos?

git status

Untracked, modified, staged, committed

Las modificaciones que hacemos pueden estar en **4 estados** distintos:

- **Sin seguimiento (untracked):** archivos que nunca fueron agregados al repositorio, por ejemplo archivos nuevos.

Output de ejemplo

```
Untracked files:
  README
```

¿Está preparado, confirmado o ninguna de las dos?

git status

Untracked, modified, staged, committed

Las modificaciones que hacemos pueden estar en **4 estados** distintos:

- **Modificado (modified):** archivos modificados que no fueron confirmados.

Output de ejemplo

```
Changes not staged for commit:  
  modified: README
```

¿Está preparado, confirmado o ninguna de las dos?

git status

Untracked, modified, staged, committed

Las modificaciones que hacemos pueden estar en **4 estados** distintos:

- **Preparado (staged):** modificaciones confirmadas (con `git add`) que irán en la próxima *confirmación de cambios* (*commit*).

Output de ejemplo

```
Changes to be committed:  
  new file:   README
```

¿Está preparado, confirmado o ninguna de las dos?

git status

Untracked, modified, staged, committed

Las modificaciones que hacemos pueden estar en **4 estados** distintos:

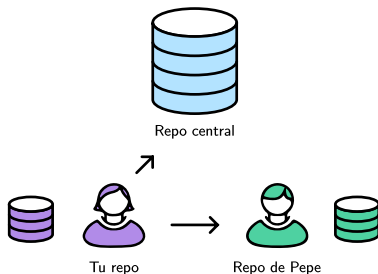
- **Confirmado (committed):** las modificaciones están guardadas con un *mensaje* que explica los cambios realizados.

Output de ejemplo

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


Colaborando con otras personas

Los repositorios remotos son *copias* de nuestro proyecto a las cuales accedemos a través de Internet. Puede haber varios, cada uno de los cuales puede ser de solo lectura o de lectura/escritura, según los permisos que tengamos.



Colaborar con otros implica gestionar estos repositorios remotos, y mandar (**push**) y recibir (**pull**) datos de ellos cuando necesites compartir cambios.

Enviando cambios

git push

Para enviar los cambios **desde nuestro repositorio local a algún repositorio remoto**, ejecutamos: `git push [remoto] [branch]`.

`remoto` es el nombre del repositorio remoto (casi siempre vamos a escribir `origin`, que refiere al cual clonamos).

Por ahora, vamos a usarlo como `git push`.

Trayendo cambios

git pull

Para traer cambios **desde un repositorio remoto a nuestro repositorio local**, ejecutamos: `git pull [remoto] [branch]`.

Igual que antes, vamos a usarlo como:
`git pull`.

Y qué pasa si... ¡BOOM!

A veces hay conflictos

- Supongamos que dos personas (**A** y **B**) están trabajando en un mismo proyecto. Es decir, ambos tienen una *copia* en su máquina.

Y qué pasa si... ¡BOOM!

A veces hay conflictos

- Supongamos que dos personas (**A** y **B**) están trabajando en un mismo proyecto. Es decir, ambos tienen una *copia* en su máquina.
- Ahora imaginemos, por ejemplo, que **A** modifica la línea 23 del archivo `tp.java` y confirma los cambios.

Y qué pasa si... ¡BOOM!

A veces hay conflictos

- Supongamos que dos personas (**A** y **B**) están trabajando en un mismo proyecto. Es decir, ambos tienen una *copia* en su máquina.
- Ahora imaginemos, por ejemplo, que **A** modifica la línea 23 del archivo `tp.java` y confirma los cambios.
- Sin saberlo, **B** también modifica la línea 23 del archivo `tp.java`, pero pone algo distinto y confirma dichos cambios.

Y qué pasa si... ¡BOOM!

A veces hay conflictos

- Supongamos que dos personas (**A** y **B**) están trabajando en un mismo proyecto. Es decir, ambos tienen una *copia* en su máquina.
- Ahora imaginemos, por ejemplo, que **A** modifica la línea 23 del archivo `tp.java` y confirma los cambios.
- Sin saberlo, **B** también modifica la línea 23 del archivo `tp.java`, pero pone algo distinto y confirma dichos cambios.
- ¿Qué va a pasar cuando quieran compartir lo que hicieron?

Y qué pasa si... ¡BOOM!

A veces hay conflictos

- Supongamos que dos personas (**A** y **B**) están trabajando en un mismo proyecto. Es decir, ambos tienen una *copia* en su máquina.
- Ahora imaginemos, por ejemplo, que **A** modifica la línea 23 del archivo `tp.java` y confirma los cambios.
- Sin saberlo, **B** también modifica la línea 23 del archivo `tp.java`, pero pone algo distinto y confirma dichos cambios.
- ¿Qué va a pasar cuando quieran compartir lo que hicieron?
- ¿Qué va a hacer Git?

Y qué pasa si... ¡BOOM!

A veces hay conflictos

- Supongamos que dos personas (**A** y **B**) están trabajando en un mismo proyecto. Es decir, ambos tienen una *copia* en su máquina.
- Ahora imaginemos, por ejemplo, que **A** modifica la línea 23 del archivo `tp.java` y confirma los cambios.
- Sin saberlo, **B** también modifica la línea 23 del archivo `tp.java`, pero pone algo distinto y confirma dichos cambios.
- ¿Qué va a pasar cuando quieran compartir lo que hicieron?
Va a haber un conflicto, ya que dos personas modificaron de forma distinta la misma línea.
- ¿Qué va a hacer Git?

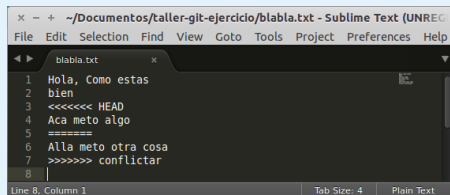
Y qué pasa si... ¡BOOM!

A veces hay conflictos

- Supongamos que dos personas (**A** y **B**) están trabajando en un mismo proyecto. Es decir, ambos tienen una *copia* en su máquina.
- Ahora imaginemos, por ejemplo, que **A** modifica la línea 23 del archivo `tp.java` y confirma los cambios.
- Sin saberlo, **B** también modifica la línea 23 del archivo `tp.java`, pero pone algo distinto y confirma dichos cambios.
- ¿Qué va a pasar cuando quieran compartir lo que hicieron?
Va a haber un conflicto, ya que dos personas modificaron de forma distinta la misma línea.
- ¿Qué va a hacer Git?
Se va a quejar. A alguno de los dos le va a tocar **incorporar a mano los cambios del otro**.

Apagando el incendio

¿Cómo se ve un conflicto?



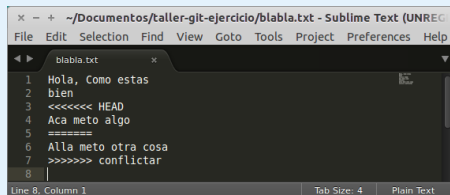
```
x - + ~/Documentos/taller-git-ejercicio/blabla.txt - Sublime Text (UNREG
File Edit Selection Find View Goto Tools Project Preferences Help
blabla.txt
1 Hola, Como estas
2 bien
3 <<<<<< HEAD
4 Aca meto algo
5 =====
6 Alla meto otra cosa
7 >>>>>> conflictar
8
Line 8, Column 1 Tab Size: 4 Plain Text
```

¿Qué hago?

- Decido cómo tiene que quedar el archivo final.

Apagando el incendio

¿Cómo se ve un conflicto?



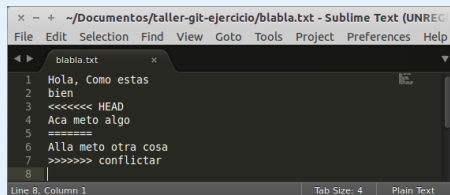
```
x - + ~/Documentos/taller-git-ejercicio/blabla.txt - Sublime Text (UNREG
File Edit Selection Find View Goto Tools Project Preferences Help
blabla.txt
1 Hola, Como estas
2 bien
3 <<<<<<< HEAD
4 Aca meto algo
5 =====
6 Alla meto otra cosa
7 >>>>>>> conflictar
8
Line 8, Column 1 Tab Size: 4 Plain Text
```

¿Qué hago?

- Decido cómo tiene que quedar el archivo final.
- Hago add.

Apagando el incendio

¿Cómo se ve un conflicto?



```
x - + ~/Documentos/taller-git-ejercicio/blabla.txt - Sublime Text (UNREG
File Edit Selection Find View Goto Tools Project Preferences Help
blabla.txt
1 Hola, Como estas
2 bien
3 <<<<<<< HEAD
4 Aca meto algo
5 =====
6 Alla meto otra cosa
7 >>>>>>> conflictar
8
Line 8, Column 1 Tab Size: 4 Plain Text
```

¿Qué hago?

- Decido cómo tiene que quedar el archivo final.
- Hago add.
- Despues `commit` normalmente, con un mensaje como 'Merge'.

Creando un repositorio vacío

git init

Crea un repositorio local vacío. Un lienzo en blanco, por así decirlo.

- 1 Nos paramos en el directorio que queremos convertir en un repositorio.
- 2 Ejecutamos `git init`.

Esto crea un subdirectorio `.git` que tiene todos los archivos necesarios de Git.

Vinculando un repositorio remoto

git remote

Ver los repositorios remotos asociados

Ejecutamos `git remote -v`.

```
origin gitlab.com/usuario/algo-2023c2-individual.git  
(fetch)
```

```
origin gitlab.com/usuario/algo-2023c2-individual.git  
(push)
```

Vinculando un repositorio remoto

git remote

Ver los repositorios remotos asociados

Ejecutamos `git remote -v`.

Agregar un repositorio remoto

Ejecutamos `git remote add [nombre que queramos] [URL]`.
El repositorio remoto ya debe existir en el servidor web.

Resumen

Recapitulando

Pasos típicos:

- Si el repositorio no existe, lo creamos en el servidor web (e.g. GitLab).

Recapitulando

Pasos típicos:

- Si el repositorio no existe, lo creamos en el servidor web (e.g. GitLab).
- Obtener una copia local del repositorio (`git...`).

Recapitulando

Pasos típicos:

- Si el repositorio no existe, lo creamos en el servidor web (e.g. GitLab).
- Obtener una copia local del repositorio (`git clone`).

Recapitulando

Pasos típicos:

- Si el repositorio no existe, lo creamos en el servidor web (e.g. GitLab).
- Obtener una copia local del repositorio (`git clone`).
- Si ya lo teníamos clonado, actualizarlo (`git...`).

Recapitulando

Pasos típicos:

- Si el repositorio no existe, lo creamos en el servidor web (e.g. GitLab).
- Obtener una copia local del repositorio (`git clone`).
- Si ya lo teníamos clonado, actualizarlo (`git pull`).

Recapitulando

Pasos típicos:

- Si el repositorio no existe, lo creamos en el servidor web (e.g. GitLab).
- Obtener una copia local del repositorio (`git clone`).
- Si ya lo teníamos clonado, actualizarlo (`git pull`).
- Trabajar sobre los archivos. Realizar modificaciones.

Recapitulando

Pasos típicos:

- Si el repositorio no existe, lo creamos en el servidor web (e.g. GitLab).
- Obtener una copia local del repositorio (`git clone`).
- Si ya lo teníamos clonado, actualizarlo (`git pull`).
- Trabajar sobre los archivos. Realizar modificaciones.
- Marcar cambios como preparados (*staged*) (`git...`),

Recapitulando

Pasos típicos:

- Si el repositorio no existe, lo creamos en el servidor web (e.g. GitLab).
- Obtener una copia local del repositorio (`git clone`).
- Si ya lo teníamos clonado, actualizarlo (`git pull`).
- Trabajar sobre los archivos. Realizar modificaciones.
- Marcar cambios como preparados (*staged*) (`git add`),

Recapitulando

Pasos típicos:

- Si el repositorio no existe, lo creamos en el servidor web (e.g. GitLab).
- Obtener una copia local del repositorio (`git clone`).
- Si ya lo teníamos clonado, actualizarlo (`git pull`).
- Trabajar sobre los archivos. Realizar modificaciones.
- Marcar cambios como preparados (*staged*) (`git add`), y confirmar estos cambios (`git...`).

Recapitulando

Pasos típicos:

- Si el repositorio no existe, lo creamos en el servidor web (e.g. GitLab).
- Obtener una copia local del repositorio (`git clone`).
- Si ya lo teníamos clonado, actualizarlo (`git pull`).
- Trabajar sobre los archivos. Realizar modificaciones.
- Marcar cambios como preparados (*staged*) (`git add`), y confirmar estos cambios (`git commit`).

Recapitulando

Pasos típicos:

- Si el repositorio no existe, lo creamos en el servidor web (e.g. GitLab).
- Obtener una copia local del repositorio (`git clone`).
- Si ya lo teníamos clonado, actualizarlo (`git pull`).
- Trabajar sobre los archivos. Realizar modificaciones.
- Marcar cambios como preparados (*staged*) (`git add`), y confirmar estos cambios (`git commit`).
- Ver el estado actual de nuestros cambios (`git...`).

Recapitulando

Pasos típicos:

- Si el repositorio no existe, lo creamos en el servidor web (e.g. GitLab).
- Obtener una copia local del repositorio (`git clone`).
- Si ya lo teníamos clonado, actualizarlo (`git pull`).
- Trabajar sobre los archivos. Realizar modificaciones.
- Marcar cambios como preparados (*staged*) (`git add`), y confirmar estos cambios (`git commit`).
- Ver el estado actual de nuestros cambios (`git status`).

Recapitulando

Pasos típicos:

- Si el repositorio no existe, lo creamos en el servidor web (e.g. GitLab).
- Obtener una copia local del repositorio (`git clone`).
- Si ya lo teníamos clonado, actualizarlo (`git pull`).
- Trabajar sobre los archivos. Realizar modificaciones.
- Marcar cambios como preparados (*staged*) (`git add`), y confirmar estos cambios (`git commit`).
- Ver el estado actual de nuestros cambios (`git status`).
- Enviar nuestros cambios al repositorio remoto (`git...`).

Recapitulando

Pasos típicos:

- Si el repositorio no existe, lo creamos en el servidor web (e.g. GitLab).
- Obtener una copia local del repositorio (`git clone`).
- Si ya lo teníamos clonado, actualizarlo (`git pull`).
- Trabajar sobre los archivos. Realizar modificaciones.
- Marcar cambios como preparados (*staged*) (`git add`), y confirmar estos cambios (`git commit`).
- Ver el estado actual de nuestros cambios (`git status`).
- Enviar nuestros cambios al repositorio remoto (`git push`).

Ramificaciones en Git

Rama (*branch*)

Representa una línea independiente de desarrollo.

Al crear nuevas ramas, podemos pensar que nuestro proyecto *diverge* en dos líneas distintas: los cambios que hagamos en una no impactan a la otra.

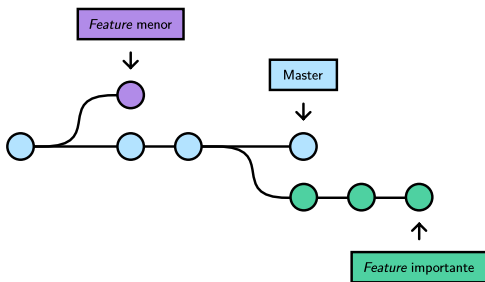
Ramificaciones en Git

Rama (*branch*)

Representa una línea independiente de desarrollo.

Al crear nuevas ramas, podemos pensar que nuestro proyecto *diverge* en dos líneas distintas: los cambios que hagamos en una no impactan a la otra.

Un ejemplo visual:



Creando ramas

```
git branch [nombre de la rama]
```

Para crear una rama nueva, ejecutamos `git branch [nombre de la rama]`.

Creando ramas

```
git branch [nombre de la rama]
```

Para crear una rama nueva, ejecutamos `git branch [nombre de la rama]`.

Nótese que este comando no nos mueve a la nueva rama, solo la crea.

Para ver las ramas de nuestro repositorio local: `git branch`.

Las ramas remotas se pueden listar con `git branch -a`.

Cambiando de rama

```
git checkout [nombre de la rama]
```

Para cambiar de rama, ejecutamos `git checkout [nombre de la rama]`.

Cambiando de rama

```
git checkout [nombre de la rama]
```

Para cambiar de rama, ejecutamos `git checkout [nombre de la rama]`.

Si se trata de una rama remota, debemos crearla localmente y *luego* obtenerla:

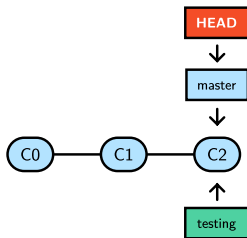
```
git checkout -b [nombre de la rama] origin/[nombre de la rama]
```

Cambiando de rama

`git checkout [nombre de la rama]`

Para cambiar de rama, ejecutamos `git checkout [nombre de la rama]`.

Un ejemplo visual:

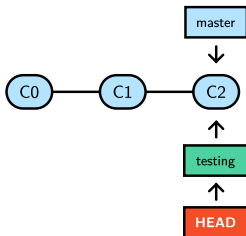


Cambiando de rama

`git checkout [nombre de la rama]`

Para cambiar de rama, ejecutamos `git checkout [nombre de la rama]`.

Un ejemplo visual:



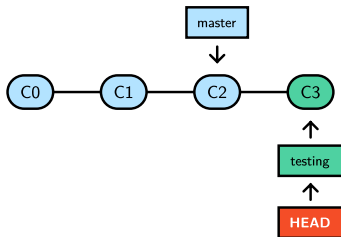
Acá cambiamos a la rama “testing”, ejecutando `git checkout testing`.

Cambiando de rama

`git checkout [nombre de la rama]`

Para cambiar de rama, ejecutamos `git checkout [nombre de la rama]`.

Un ejemplo visual:



Y los siguientes *commits* serán agregados a la rama “testing”.

Fusionando ramas

```
git merge [nombre de la rama a fusionar]
```

Nos permite fusionar las historias de dos ramas distintas (podría haber conflictos).

Fusionando ramas

```
git merge [nombre de la rama a fusionar]
```

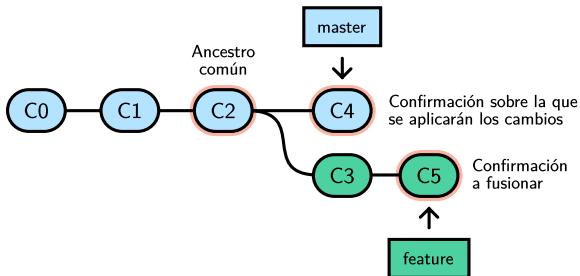
Nos permite fusionar las historias de dos ramas distintas (podría haber conflictos).

Importante: este comando fusiona la rama que le decimos **en la rama en la que estamos parados**.

Fusionando ramas

`git merge` [nombre de la rama a fusionar]

Un ejemplo visual:

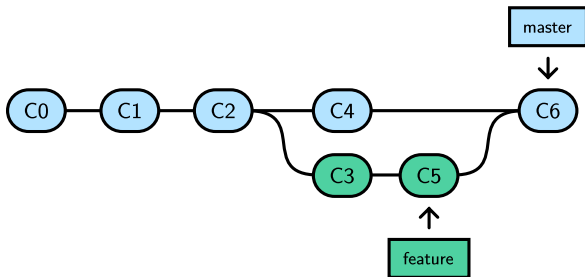


Antes del *merge*.

Fusionando ramas

`git merge` [nombre de la rama a fusionar]

Un ejemplo visual:



Después de pararnos en la rama “master” (`git checkout master`) y haber fusionado la rama “feature” (`git merge feature`).

Otros comandos útiles

```
git rm [archivo]
```

Permite borrar un archivo y marcar este cambio como *staged*.

```
git mv [archivo] [nuevo nombre/ubicación]
```

Otros comandos útiles

```
git rm [archivo]
```

Permite borrar un archivo y marcar este cambio como *staged*.

```
git mv [archivo] [nuevo nombre/ubicación]
```

Permite mover/renombrar un archivo y marcar este cambio como *staged*.

Inspeccionando los cambios

git diff

Muestra las diferencias entre el estado actual de los archivos y la última vez que hicimos `git add` (cambios marcados como *staged*).

Inspeccionando los cambios

git diff

Muestra las diferencias entre el estado actual de los archivos y la última vez que hicimos `git add` (cambios marcados como *staged*).

Si, en cambio, queremos ver las diferencias entre los cambios marcados como *staged* y los que confirmamos en el último *commit*, podemos usar `git diff --staged`.

Viendo la historia de los *commits*

git log

Muestra el historial de *commits* para saber cuáles son las modificaciones que se hicieron (siempre y cuando hayamos elegido mensajes descriptivos!).

Herramientas

Existen diversas herramientas gráficas interesantes (MUCHAS).

Algunas:

- git gui - Realizar commits y otros gráficamente

Herramientas

Existen diversas herramientas gráficas interesantes (MUCHAS).

Algunas:

- git gui - Realizar commits y otros gráficamente
- Meld - Diff gráfico

Herramientas

Existen diversas herramientas gráficas interesantes (MUCHAS).

Algunas:

- git gui - Realizar commits y otros gráficamente
- Meld - Diff gráfico
- gitk - Git log gráfico, cambios y varias cosas más

Herramientas

Existen diversas herramientas gráficas interesantes (MUCHAS).

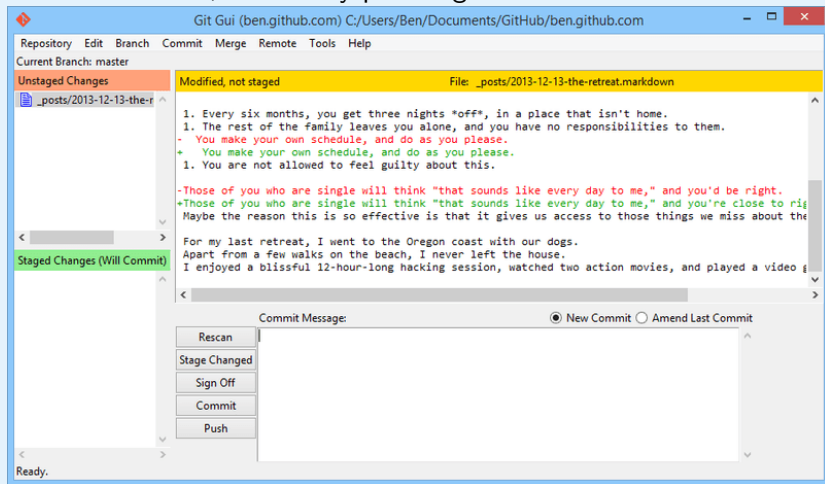
Algunas:

- git gui - Realizar commits y otros gráficamente
- Meld - Diff gráfico
- gitk - Git log gráfico, cambios y varias cosas más
- tig - Gitk por línea de comandos - NCURSES

Herramientas - git gui

git gui

Mostrar cambios, commits y pushes gráfico



Herramientas - meld

meld

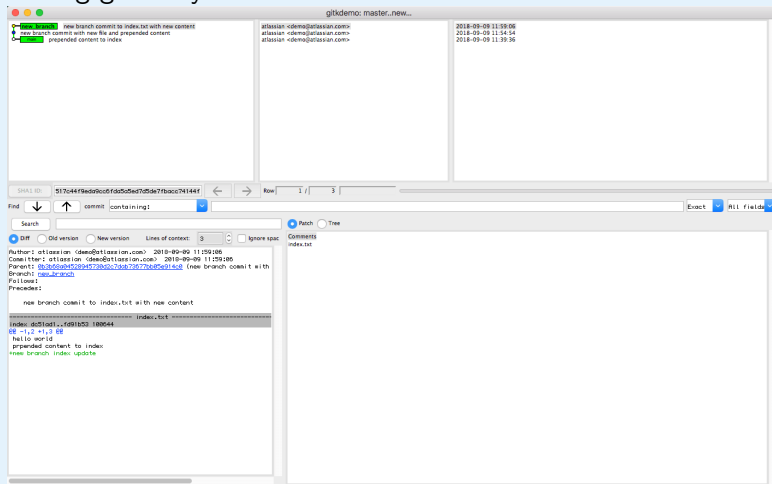
Diff gráfico

```
2130 self.recompute_label()
2131 index = self.textbuffer.index(buf)
2132 self.readonlytoggle[index].props.visib
2133 self.set_buffer_editable(buf, writable
2134
2135 def set_buffer_editable(self, buf, editabl
2136 index = self.textbuffer.index(buf)
2137 self.readonlytoggle[index].set_active(
2138 self.readonlytoggle[index].get_child() →
2139 'changes-allow-symbolic' if editab
2140 'changes-prevent-symbolic')
2141 self.textview[index].set_editable(edit
2142 self.on_cursor_position_changed(buf, N
2143
2144 @with_focused_pane
2145 def action_save(self, pane, *args): →
2146 self.save_file(pane)
2147
2148 @with_focused_pane
2149 def action_save_as(self, pane, *args): →
2150 self.save_file(pane, saveas=True)
2151
2152 def action_save_all(self, *args): →
2153 for i in range(self.num_panes):
2154 if self.textbuffer[i].get_modified
2155 self.save_file(i)
2156
1650 self.readonlytoggle[index].props.visib
1651 self.set_buffer_editable(buf, writable
1652
1653 def set_buffer_editable(self, buf, editabl
1654 index = self.textbuffer.index(buf)
1655 self.readonlytoggle[index].set_active(
←1656 self.readonlytoggle[index].props.icon_
1657 'changes-allow-symbolic' if editab
1658 'changes-prevent-symbolic')
1659 self.textview[index].set_editable(edit
1660 self.on_cursor_position_changed(buf, N
←1661 for linkmap in self.linkmap:
1662 linkmap.queue_draw()
1663
1664 @with_focused_pane
←1665 def save(self, pane):
1666 self.save_file(pane)
1667
1668 @with_focused_pane
←1669 def save_as(self, pane):
1670 self.save_file(pane, saveas=True)
1671
←1672 def on_save_all_activate(self, action):
1673 for i in range(self.num_panes):
1674 if self.textbuffer[i].get_modified
1675 self.save_file(i)
1676
```

Herramientas - gitk

gitk

Git log gráfico y más



Herramientas - tig

gitk

tig

```
2004-05-09 19:22 Junio C Hamano      [master] [next] Merge branch 'master' into next
2004-05-09 19:24 Junio C Hamano      Merge branch 'fix'
2006-05-09 19:23 Junio C Hamano      checkout: use --aggressive when running a 3-way merge (-m).
2004-05-09 19:22 Linus Torvalds       revert/cherry-pick: use aggressive merge.
2004-05-09 14:52 Junio C Hamano      Merge branch 'jc/clean'
2004-05-09 14:45 Junio C Hamano      Merge branch 'mw/alternates'
2004-05-09 14:44 Junio C Hamano      Merge branch 'jc/xshal'
2004-05-09 14:40 Junio C Hamano      Merge branch 'jc/again'
2004-05-09 14:40 Junio C Hamano      Merge branch 'np/delta'
2004-05-09 14:16 Junio C Hamano      Merge branch 'jc/bindiff'

[main] 8d7a397aab561d3782f531e733b617e0e211f04a - commit 3 of 577 (0%)
commit 8d7a397aab561d3782f531e733b617e0e211f04a
Author: Junio C Hamano <junkio@cox.net>
Date:   Tue May 9 19:23:23 2006 -0700

    checkout: use --aggressive when running a 3-way merge (-m).

    After doing an in-index 3-way merge, we always do the stock
    "merge-index merge-one-file" without doing anything fancy;
    use of --aggressive helps performance quite a bit.

    Signed-off-by: Junio C Hamano <junkio@cox.net>

---
git-checkout.sh |    2 +-
1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/git-checkout.sh b/git-checkout.sh
index 463ed3e..a11e939 100755
--- a/git-checkout.sh
+++ b/git-checkout.sh
[diff] 8d7a397aab561d3782f531e733b617e0e211f04a - line 1 of 28 (3%)
Loaded 28 lines in 0 seconds
```


Git en Visual Studio Code

Posicionarse sobre la carpeta raíz del repositorio

The screenshot shows the Visual Studio Code interface with the following components:

- Left Panel:** Contains the Explorer, Search, and Run and Debug views. The Explorer shows the project structure with a folder named "Carpeta".
- Git Panel (Top Left):** Displays the status of the repository. It shows "Archivos modificados ó sin seguimiento (git status)" and "Descartar cambios (git restore) ó prepararlos (git add)".
- Commit History (Bottom Left):** A list of commits with their messages and authors. The first commit is "Añadir carpeta de ejemplo" by "Carpeta".
- Main Editor:** Displays the "Cambios efectuados en el archivo seleccionado (git diff)" view. It shows the differences between the current state and the last commit. The diff is for the file "Carpeta\ejercicio1\ejercicio1.cs".

The diff view shows the following changes:

```
diff --git a/Carpeta/ejercicio1/ejercicio1.cs b/Carpeta/ejercicio1/ejercicio1.cs
index 1234567..8901234
+using System;
+using System.Collections.Generic;
+using System.Linq;
+using System.Text;
+using System.Threading.Tasks;
+
+namespace Carpeta.ejercicio1
+{
+    class Program
+    {
+        static void Main(string[] args)
+        {
+            Console.WriteLine("Hola mundo!");
+        }
+    }
+}
```

¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): **A** y **B**

¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): **A** y **B**

- 1 **A**: Hacer un *fork* en su cuenta de GitLab de este repositorio: <https://gitlab.com/algo2-catedra/tallergit>, y darle permiso a **B** para hacer *push*.

¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): **A** y **B**

- 1 **A**: Hacer un *fork* en su cuenta de GitLab de este repositorio: <https://gitlab.com/algo2-catedra/tallergit>, y darle permiso a **B** para hacer *push*.
- 2 **A** y **B**: obtener una copia local del repositorio.

¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): **A** y **B**

- 1 **A**: Hacer un *fork* en su cuenta de GitLab de este repositorio: <https://gitlab.com/algo2-catedra/tallergit>, y darle permiso a **B** para hacer *push*.
- 2 **A** y **B**: obtener una copia local del repositorio.
- 3 **A** y **B**: corregir el bug que hay en el código (¿también refactorizar?)

¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): **A** y **B**

- 1 **A**: Hacer un *fork* en su cuenta de GitLab de este repositorio: <https://gitlab.com/algo2-catedra/tallergit>, y darle permiso a **B** para hacer *push*.
- 2 **A** y **B**: obtener una copia local del repositorio.
- 3 **A** y **B**: corregir el bug que hay en el código (¿también refactorizar?)
- 4 **A** o **B** (quien haya terminado primero): hacer *push* de los cambios al repositorio remoto.

¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): **A** y **B**

- 1 **A**: Hacer un *fork* en su cuenta de GitLab de este repositorio: <https://gitlab.com/algo2-catedra/tallergit>, y darle permiso a **B** para hacer *push*.
- 2 **A** y **B**: obtener una copia local del repositorio.
- 3 **A** y **B**: corregir el bug que hay en el código (¿también refactorizar?)
- 4 **A** o **B** (quien haya terminado primero): hacer *push* de los cambios al repositorio remoto.
- 5 **A** o **B**: intentar hacer *push* de los cambios al repositorio remoto. ¿Qué pasó?

¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): **A** y **B**

- 1 **A**: Hacer un *fork* en su cuenta de GitLab de este repositorio: <https://gitlab.com/algo2-catedra/tallergit>, y darle permiso a **B** para hacer *push*.
- 2 **A** y **B**: obtener una copia local del repositorio.
- 3 **A** y **B**: corregir el bug que hay en el código (¿también refactorizar?)
- 4 **A** o **B** (quien haya terminado primero): hacer *push* de los cambios al repositorio remoto.
- 5 **A** o **B**: intentar hacer *push* de los cambios al repositorio remoto. ¿Qué pasó?
- 6 **A** o **B**: bajarse los cambios del repositorio remoto. ¿Anduvo?

¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): **A** y **B**

- 1 **A**: Hacer un *fork* en su cuenta de GitLab de este repositorio: <https://gitlab.com/algo2-catedra/tallergit>, y darle permiso a **B** para hacer *push*.
- 2 **A** y **B**: obtener una copia local del repositorio.
- 3 **A** y **B**: corregir el bug que hay en el código (¿también refactorizar?)
- 4 **A** o **B** (quien haya terminado primero): hacer *push* de los cambios al repositorio remoto.
- 5 **A** o **B**: intentar hacer *push* de los cambios al repositorio remoto. ¿Qué pasó?
- 6 **A** o **B**: bajarse los cambios del repositorio remoto. ¿Anduvo?
- 7 **A** o **B**: resolver los conflictos que haya.

¡A practicar!

Ejercicio de a 2 máquinas (preferiblemente 2 personas): **A** y **B**

- 1 **A**: Hacer un *fork* en su cuenta de GitLab de este repositorio: <https://gitlab.com/algo2-catedra/tallergit>, y darle permiso a **B** para hacer *push*.
- 2 **A** y **B**: obtener una copia local del repositorio.
- 3 **A** y **B**: corregir el bug que hay en el código (¿también refactorizar?)
- 4 **A** o **B** (quien haya terminado primero): hacer *push* de los cambios al repositorio remoto.
- 5 **A** o **B**: intentar hacer *push* de los cambios al repositorio remoto. ¿Qué pasó?
- 6 **A** o **B**: bajarse los cambios del repositorio remoto. ¿Anduvo?
- 7 **A** o **B**: resolver los conflictos que haya.
- 8 **A** o **B**: añadir y confirmar el archivo que tenía conflicto.

Ejercicio de a 2 máquinas (preferiblemente 2 personas): **A** y **B**

- 1 **A**: Hacer un *fork* en su cuenta de GitLab de este repositorio: <https://gitlab.com/algo2-catedra/tallergit>, y darle permiso a **B** para hacer *push*.
- 2 **A** y **B**: obtener una copia local del repositorio.
- 3 **A** y **B**: corregir el bug que hay en el código (¿también refactorizar?)
- 4 **A** o **B** (quien haya terminado primero): hacer *push* de los cambios al repositorio remoto.
- 5 **A** o **B**: intentar hacer *push* de los cambios al repositorio remoto. ¿Qué pasó?
- 6 **A** o **B**: bajarse los cambios del repositorio remoto. ¿Anduvo?
- 7 **A** o **B**: resolver los conflictos que haya.
- 8 **A** o **B**: añadir y confirmar el archivo que tenía conflicto.
- 9 **A** o **B**: *pushear* estos nuevos cambios.

Ejercicio de a 2 máquinas (preferiblemente 2 personas): **A** y **B**

- 1 **A**: Hacer un *fork* en su cuenta de GitLab de este repositorio: <https://gitlab.com/algo2-catedra/tallergit>, y darle permiso a **B** para hacer *push*.
- 2 **A** y **B**: obtener una copia local del repositorio.
- 3 **A** y **B**: corregir el bug que hay en el código (¿también refactorizar?)
- 4 **A** o **B** (quien haya terminado primero): hacer *push* de los cambios al repositorio remoto.
- 5 **A** o **B**: intentar hacer *push* de los cambios al repositorio remoto. ¿Qué pasó?
- 6 **A** o **B**: bajarse los cambios del repositorio remoto. ¿Anduvo?
- 7 **A** o **B**: resolver los conflictos que haya.
- 8 **A** o **B**: añadir y confirmar el archivo que tenía conflicto.
- 9 **A** o **B**: *pushear* estos nuevos cambios.
- 10 **A** o **B**: bajarse los nuevos cambios.

Revirtiendo cambios

`git commit --amend`

Podemos usarlo para **arreglar**, por ejemplo, el mensaje del último *commit* que hicimos: `git commit --amend -m [nuevo mensaje]`.

`git revert [hash]`

Revirtiendo cambios

git commit --amend

Podemos usarlo para **arreglar**, por ejemplo, el mensaje del último *commit* que hicimos: `git commit --amend -m [nuevo mensaje]`.

git revert [hash]

Permite **revertir** exactamente los cambios introducidos por un *commit*. Buscamos el *hash* del *commit* en cuestión usando `git log`, y luego ejecutamos `git revert [hash]`.

Otro comando útil

git stash

Guarda el estado actual de los archivos modificados y nos deja el directorio limpio.

Para volver a mostrar los cambios guardados ejecutamos `git stash apply`.

Extras: Ignorando archivos

El archivo .gitignore

No solemos subir logs, pdfs, binarios, por lo cual está bueno que no figuren cada vez que ejecutamos `git status`.

Especificamos qué archivos (o extensiones) **ignorar** por completo en un archivo especial llamado `.gitignore`.

Extras: Ignorando archivos

El archivo .gitignore

No solemos subir logs, pdfs, binarios, por lo cual está bueno que no figuren cada vez que ejecutamos `git status`.

Especificamos qué archivos (o extensiones) **ignorar** por completo en un archivo especial llamado `.gitignore`.

Por ejemplo, para ignorar la carpeta *target*:

- 1 Crear un archivo llamado `.gitignore` en el directorio principal del proyecto.
- 2 Adentro escribir: *target/*

Más ejemplos de `.gitignore`:

<https://github.com/github/gitignore>.

Extras: Más comandos

- `git fetch [remote repository]`: Para traer todos los datos de un repositorio remoto.
- `git reset`: Permite deshacer cambios; sirve para revertir modificaciones en el área de trabajo, pero también puede eliminar por completo *commits* anteriores. ¡Usar con mucho cuidado!
- `git rebase [branch]`: Aplica todos los *commits* que difieren entre una rama y aquella en la que estamos parados. Así podemos incorporar cambios realizados en otras ramas manteniendo lineal el historial de la rama actual.
- `git blame [archivo]`: Para ver qué *commit* modificó por última vez cada línea de un archivo, y quién fue su autor. ¡Así, podemos saber a quién *culpar* cuando haya problemas!
- `git bisect`: Encuentra cuál fue el *commit* que introdujo cierto error, haciendo búsqueda binaria en el historial de *commits*.

Bibliografía

- Git Community book, disponible online y en español:
<https://git-scm.com/book/es/v2>
- `git help [command]` para ver la documentación de cualquier comando de Git.
- A visual Git reference: <http://marklodato.github.io/visual-git-guide/index-es.html>
- Try Git online: <https://try.github.io>
- git gui: <https://git-scm.com/docs/git-gui/>
- Meld: <https://meldmerge.org/>
- Gitk: <https://www.atlassian.com/es/git/tutorials/gitk>
- tig: <https://jonas.github.io/tig/>
- Git en vscode: <https://code.visualstudio.com/docs/sourcecontrol/intro-to-git>