

Prácticas Profesionalizantes 3

Profesor: Goral Gonzalo
Tecnicatura en Análisis de Sistemas
ISFDYT N° 210 - La Plata 2020

Tema de la clase

Sistemas de control de
Versiones Avanzado

GIT

Objetivo de la clase

1. Repaso

- 1.1. Características de Git
- 1.2. Ciclo de vida
- 1.3. Trabajando con estados

2. Ramas/Etiquetas

- 2.1. Introducción
- 2.2. Branch (Rama)
- 2.3. Tag (Etiquetas)

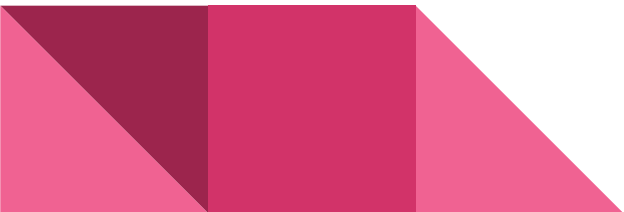
3. Formas de trabajo

- 3.1. Workflows
- 3.2. Gitlab/Github



Características de Git

1. **Snapshots, no diferencias**
2. **Operaciones, casi todas las operaciones son locales**
3. **Integridad**





Snapshots

La mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos.

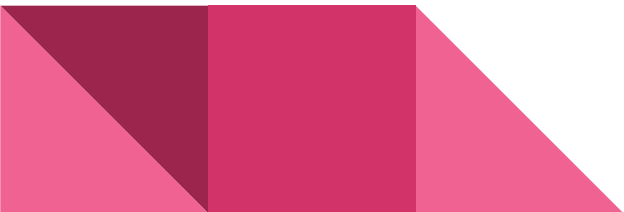
Git modela sus datos más como un conjunto de instantáneas (snapshots) de un mini sistema de archivos.

Por eficiencia, si un archivo no cambió, no vuelve a guardarlo, sólo referencia al archivo ya almacenado.

Operaciones

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para operar.

Para navegar por la historia del proyecto, Git no necesita buscarla en el servidor.



Integridad

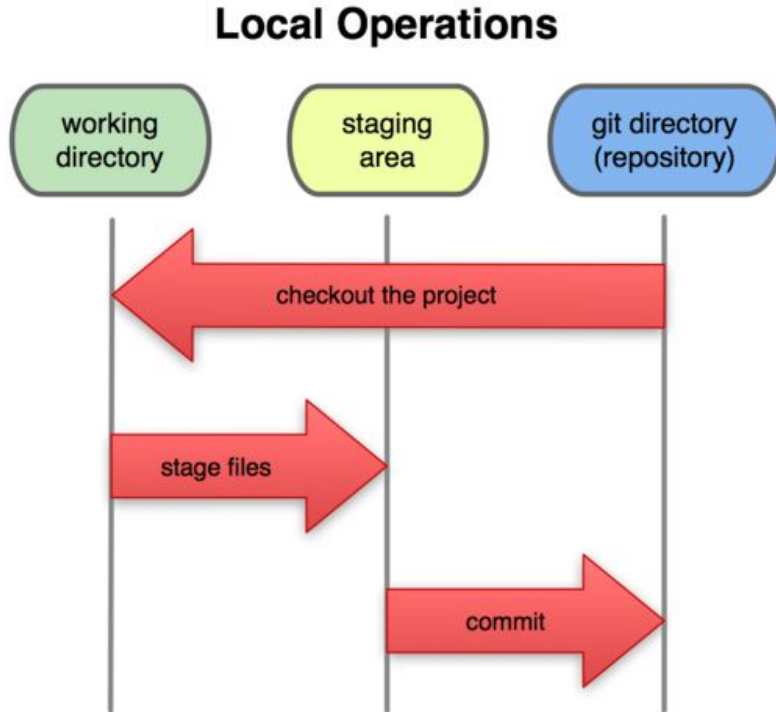
Todo en Git es verificado mediante una suma de comprobación antes de ser almacenado, y es identificado a partir de ese momento mediante dicho checksum.

Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa.

Ejemplo (SHA-1 de 40 caracteres)



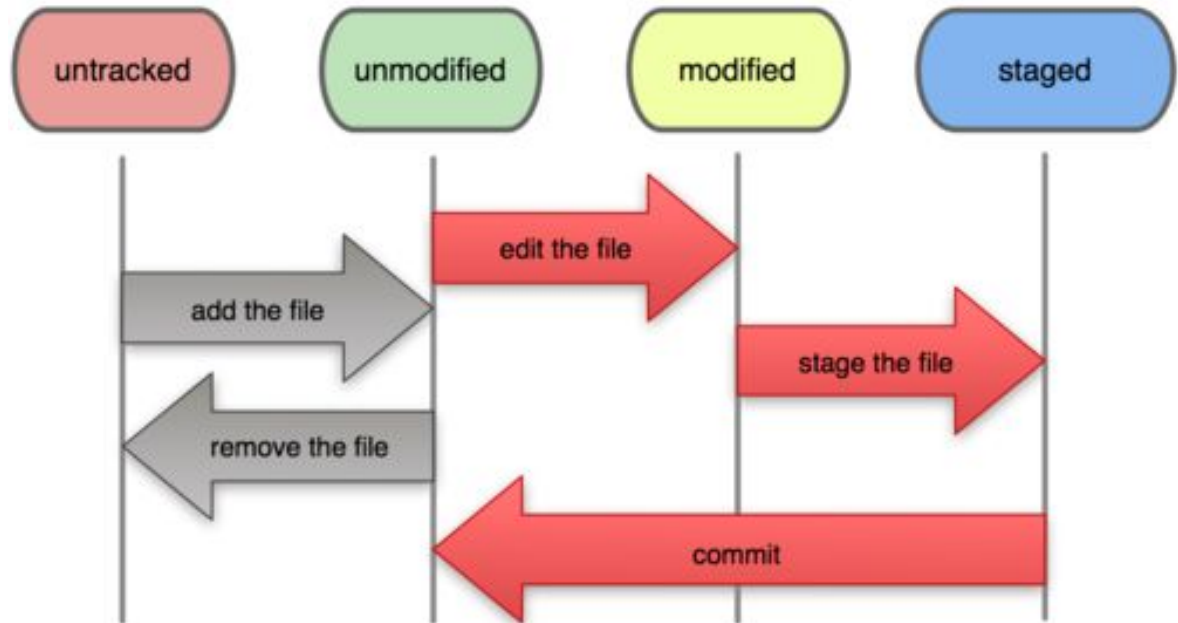
Los tres estados



- **WORKING DIRECTORY**
- **STAGING AREA**
- **GIT DIRECTORY**

Ciclo de Vida

File Status Lifecycle



Comprobando el estado del repositorio

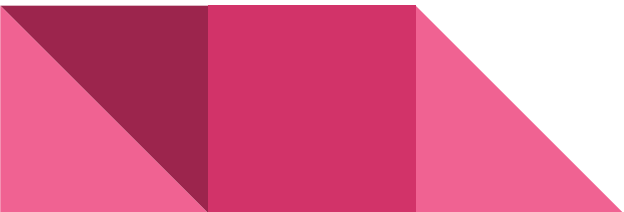
RECORDAR:

`$ git add -> pasa de untracked o de modified a staged.`

`$ git unstage -> pasa de staged a untracked o modified.`

`$ git commit -> pasa de staged a unmodified.`

`#Los archivos unmodified son los únicos correctamente “committed”.`



Estados (status)

Local:

- Untracked files
- Changes not staged for commit
- Changes to be committed

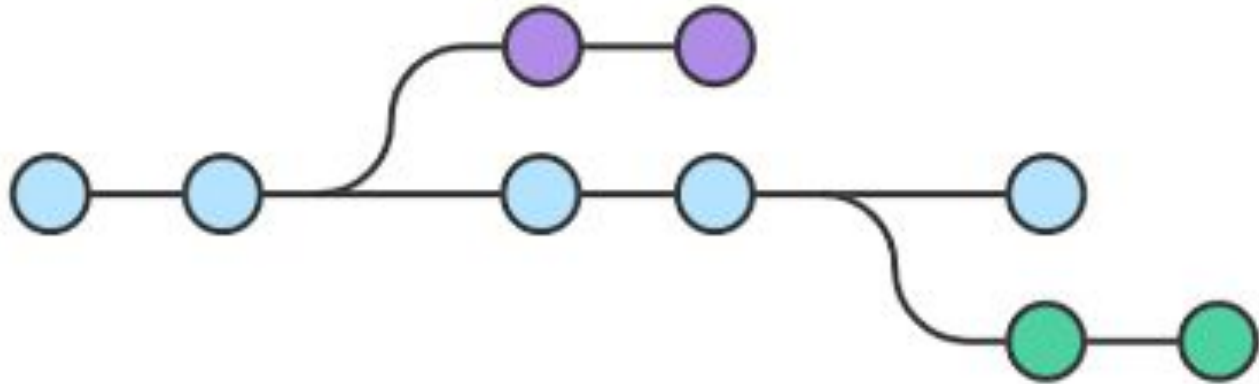
Con remotos:

- Your branch is ahead of 'origin/master' by 1 commit.
-> push
- Your branch is behind 'origin/master' by 1 commit...
-> pull
- Your branch and 'origin/master' have diverged, and have 1 and 1 different commit(s) each, respectively.
-> pull y después → push



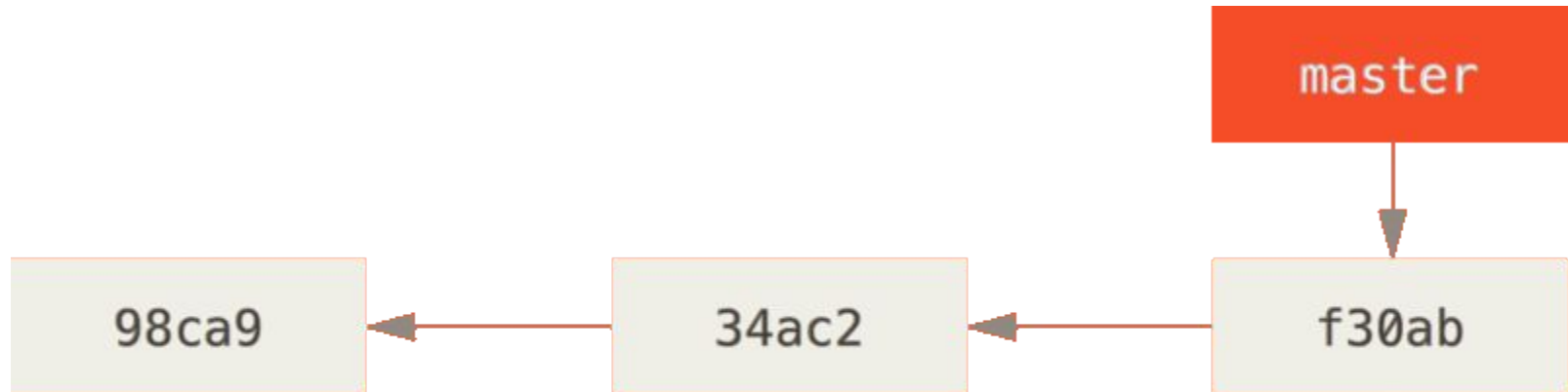
Branch (Rama) - Conceptualmente

Una rama (o branch) en Git representa una línea independiente de desarrollo. Al crear nuevas ramas, podemos pensar que nuestro proyecto diverge en dos distintos: los cambios que hagamos en uno no impactan al otro. Entonces uno puede desarrollar una nueva funcionalidad independientemente sin interferir con la línea principal.



Branch (Rama) - Master

Por defecto git `init` crea una rama por defecto para trabajar: master.



Branch (Rama) - Creación

Para crear una rama nueva, podemos ejecutar el comando

```
$ 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, podemos ejecutar:

```
$ git branch
```

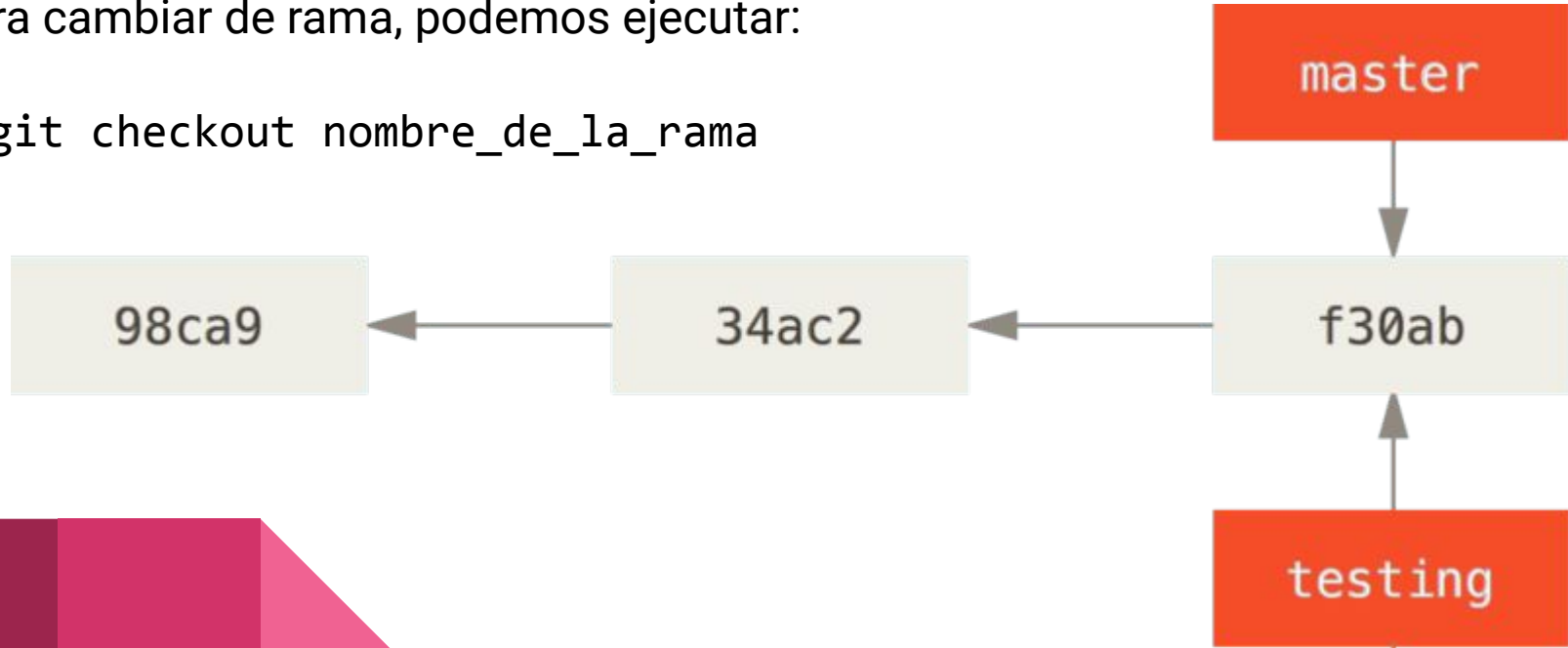
Ejercicio: Crear una rama llamada “testing” en algún repositorio de la clase pasada.



Branch (Rama) - Cambiando de Rama

Para cambiar de rama, podemos ejecutar:

```
$ git checkout nombre_de_la_rama
```



Branch (Rama) - Merge (Fusión)

El comando `git merge` nos permite fusionar las historias de dos ramas distintas (podria haber conflictos). La sintaxis es:

```
$ git merge [nombre de la rama origen]
```

Importante: Este comando fusiona la rama origen en la rama en la que estamos parados (rama destino).

Ejercicio: Crear un archivo en rama “testing” y llevarlo a la rama “master”.





Branch (Rama) - Conflictos

¿Que es un conflicto? Básicamente es modificar uno o más archivos en los mismos puntos de dos ramas.

Si al actualizar el repositorio local con el remoto (`git fetch + git merge = git pull`) no se puede realizar el merge o fusión por estar las mismas líneas modificada se produce un CONFLICTO

Git nos avisa que debemos resolver nosotros el conflicto manualmente


```
Auto-merging example.rb
CONFLICT (content): Merge conflict in example.rb
Automatic merge failed; fix conflicts and then commit the result.
```

Branch (Rama) - Conflictos

Para resolver un conflicto debemos editar el archivo que va a tener marcadas las líneas en los bloques con conflicto:

```
<<<<<<< HEAD
puts "Hola Práctica Profesional!!!"
=====
puts "hello World!!!!!!!!!!!!!!!"
>>>>>>>
```

una vez que elegimos una de esos dos bloques, se vuelve a agregar con `$ git add` y aceptar el cambio (Commitarlo) `git commit`.



Branch (Rama) - Comandos

Repositorio Local:

\$ git branch <nombre> → Crear un branch

\$ git checkout <nombre> → Cambia a un branch

\$ git diff <nombre> → Diferencia entre el branch actual y el indicado

\$ git merge <nombre> → Incorpora los cambios del branch indicado en el actual



Branch (Rama) - Comandos

Repositorio Remoto:

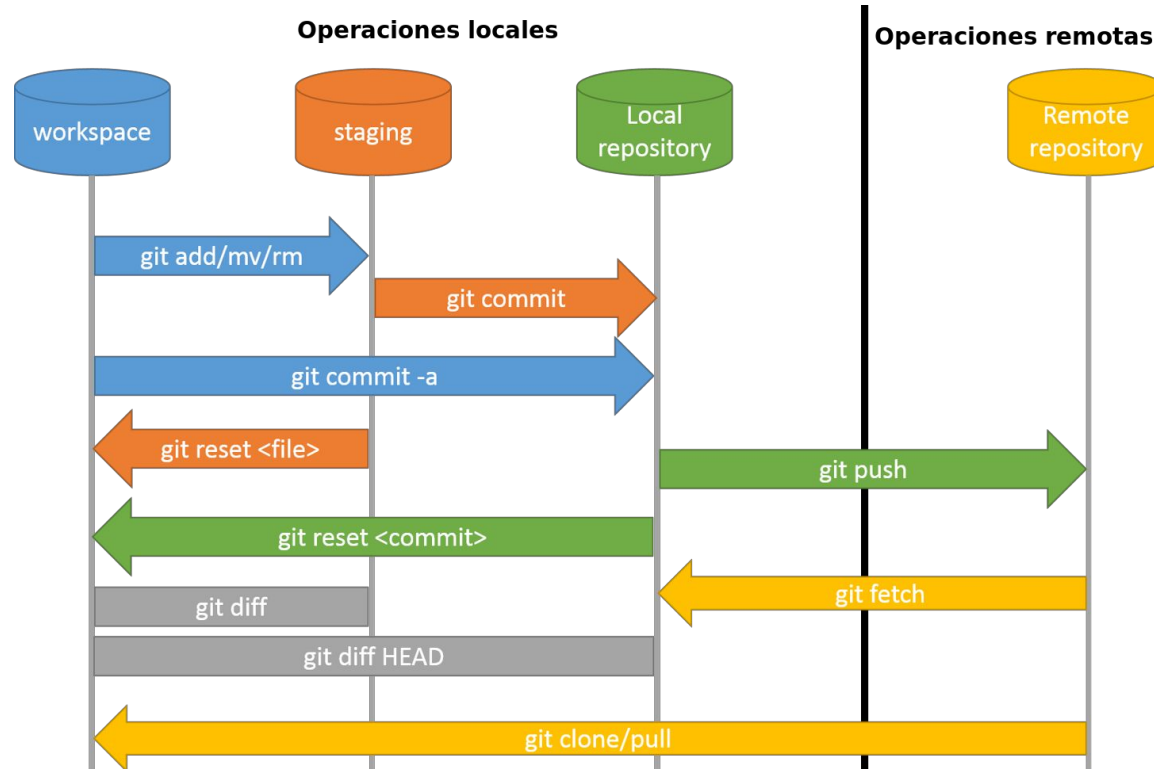
\$ git branch -a -> Para ver todos los branches

\$ git push <remoto> <branch> -> Para subir un branch a un remoto

\$ git branch -d <branch> -> Para borrar un branch localmente

\$ git branch <remoto> :<branch> -> Para borrar un branch en el servidor

Branch (Rama) - Flujo completo



Tags (Etiquetas)

Git nos da la posibilidad de marcar o taggear puntos específicos dentro de la historia de nuestro repositorio. El más habitual es marcar las versiones del desarrollo.

Listado de tags:

```
$ git tag
```

```
v1.0
```

```
v2.0
```



Tags (Etiquetas)


Git soporta 2 tipos de tags: livianos o anotados.

Los tags livianos son como ramas que nunca cambian, punteros a commits.

```
$ git tag v1.4
```

Los tags anotados, en cambio, son guardados como un objeto completo en git. Se le aplica un checksum conteniendo el nombre del autor, su email y la fecha, tiene un mensaje y pueden ser firmados y verificados con GNU Privacy Guard (GPG).

```
$ git tag -a v1.4 -m "mi version 1.4"
```

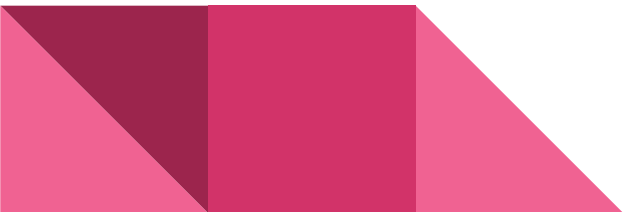


Workflows

Gitflow

Gitflow es un flujo de trabajo basado en ramas (branches) propuesto por Vincent Driessen en 2010.

Propone una serie de "reglas" para organizar el trabajo del equipo.



Workflows

Gitflow - Dos ramas principales

master: Cualquier commit que pongamos en esta rama debe estar preparado para subir a producción.

develop: Rama en la que está el código que conformará la siguiente versión planificada del proyecto.

Cada vez que se incorpora código a master, tenemos una nueva versión





Workflows

Gitflow - Ramas auxiliares

Feature: Se originan e incorporan siempre a develop, son las nuevas características de la app.

Release: Se originan en develop y se incorporan a master y develop. Se utilizan para preparar el siguiente código en producción.

Hotfix: Se originan en master y se incorporan a master y develop. Se utilizan para corregir errores y bugs en el código en producción.

Estas ramas auxiliares suelen desaparecer una vez incorporadas.

Plataformas: GitLab - GitHub

Son dos plataformas que prestan un servicio web de control de versiones y desarrollo de software colaborativo basado en Git. Además de gestor de repositorios, el servicio ofrece también alojamiento de wikis y un sistema de seguimiento de errores, en ambas se puede trabajar de la misma manera.

GitLab está liberada bajo licenciamiento open source.

GitLab es una aplicación opensource que nos permite administrar repositorios en git mediante una interfaz web.





Gitlab

GitLab utiliza claves SSH para permitir trabajar con los repositorios.

Las claves SSH son utilizadas para establecer una conexión segura entre los repositorios y GitLab.

Con lo cual lo primero que necesitamos hacer es subir nuestra clave pública al proyecto.

Si no realizamos esto el usuario no podrá subir los cambios realizado en su repositorio local al proyecto de GitLab!



Gitlab

Generación de llaves

```
ssh-keygen -t rsa -C "your_email@example.com"
```

Ir a la sección de claves SSH de su perfil del usuario y agregar la clave pública

```
.ssh/id_rsa.pub
```

Se va a utilizar para identificar y autenticar cada interacción con el servidor



Gitlab

Configurar el usuario

```
$ git config --global user.name "John Doe"
```

```
$ git config --global user.email "johndoe@mail.com"
```



Gitlab

Inicializar el repositorio Git de nuestro proyecto GitLab

Tenemos dos opciones:

Crear un repositorio vacío y enlazarlo al repositorio local de nuestro proyecto en GitLab.

Utilizar un repositorio ya creado y sólo debemos asignarlo al proyecto de GitLab.



Gitlab

Teniendo nuestro repositorio git creado, podemos empezar a utilizarlo y en GitLab tener:

- El último estado de nuestros archivos.
- Seguimiento de los commits realizados y las diferencias aplicadas.
- Una red con el crecimiento de las versiones y bifurcaciones que va tomando nuestro repositorio.
- Gráficos con estadísticas de uso.
- Creación y seguimiento de tareas (o issues) relativas al proyecto.
- Una wiki con información propia de cada proyecto.



Referencias

Libro de git:

<https://git-scm.com/book/es/v2>

Taller de git:

<https://comcom.dc.uba.ar/talleres/git/que-es-git/>

Tutorial de git de Atlassian:

<https://www.atlassian.com/es/git/tutorials/learn-git-with-bitbucket-cloud>



¿Preguntas?
En el foro