



Control de Versiones

TB022 - Esteban - Riesgo - Kristal



Qué es?

Es la práctica de trackear y manejar cambios al código fuente.

Un sistema de Control de Versiones es una herramienta diseñada con este fin.

Que beneficios tienen:

- Trackear todos los cambios que se hicieron en una aplicación. Uno por uno. Línea por línea. Y quién y cuando se efectuaron.

- Facilitar el compartir contenido. El código fuente puede ser compartido con quien querramos. Se pueden configurar reglas de acceso tan complejas como se deseen (grupos, equipos, organizaciones, etc.)

- Facilitar la colaboración. Están diseñados para que gran cantidad de personas estén trabajando sobre la misma fuente sin generar problemas.

Alternativas



Subversion (SVN)

Desarrollado y mantenido por Apache en el año 2000 y bastante usado hasta hoy inclusive

Concurrent Version System (CVS)

Desarrollado por Dick Grune en el año 1986, uno de los primeros sistemas



Git

Desarrollado originalmente por Linus Torvalds en el 2005
Ganó mucha popularidad por ser usado para manejar el desarrollo y las versiones de Linux.
Es el sistema más usado y más completo existente hoy en día.

Git



Los sistemas de versionado (VCSs) suelen guardar un delta con los cambios que se hicieron entre cada versión del sistema.

Git en cambio guarda un snapshot (una foto) cada vez que se genera una nueva versión del contenido.

Cuando un archivo no cambia, simplemente se guarda una referencia al snapshot anterior del mismo para ahorrar espacio.

Además, la mayoría de las operaciones son locales.



Git: instalación

La mayoría de los sistemas basados en Unix deberían ya tener instalado una versión de Git.

De no tenerla, pueden instalarlo siguiendo [estas instrucciones](#).

Conceptos básicos



commit: conjunto atómico de cambios.

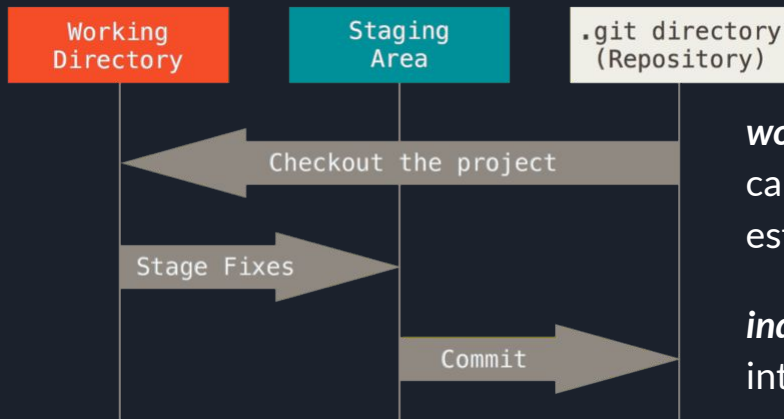
Tiene un ID único.

Todos los commits tienen un padre, salvo el primer commit.

repositorio: conjunto de commits relacionados que hacen a un proyecto

local: repositorio que se encuentra en nuestro entorno de desarrollo en el disco del ordenador siendo usado

remoto: repositorio que se encuentra en otro lado que no sea local



workdir/workspace: contiene los cambios actuales en los que estamos trabajando

index/staging area: estado intermedio al cual cambios son agregados para luego poner en un commit

Git remote repository hostings



[Student Developer Pack](#) con email FIUBA





Comandos básicos: init & clone

Iniciar un repositorio nuevo vacío

```
$ git init
```

Crea un nuevo repositorio con el mismo nombre del directorio actual (crea un directorio `.git` con la información de configuración, eso es todo)

```
$ git config [--global] user.name "<nombre completo>"  
$ git config [--global] user.email <email>
```

`--global` Guarda los datos en `~/.gitconfig`

Partir de un repo ya existente

```
git clone <url.git> [<nombre>]
```

Clona el repo remoto alojado en la URL prevista y lo coloca en un nuevo directorio `<nombre>` (si se omite, se usará el nombre del repositorio como nombre del directorio)

Si el repositorio remoto es privado, nos pedirá las credenciales para clonarlo

Todo lo que se puede configurar: [acá](#)



Git: configs

`[path]/etc/gitconfig`

Archivo en el directorio PATH. Aplica a todos los usuarios del sistema.

`~/.gitconfig` or `~/.config/git/config`

Archivo en el home del usuario. Aplica al usuario actual.

`.git/config` (en el directorio actual)

Archivo en el root path de cada repositorio. Aplica únicamente al repositorio actual.

Más general

Más específico

Las configuraciones en niveles más específicos sobrescriben a las configuraciones más generales



Comandos básicos: add & reset

```
git add [<path>] [-A, -p]
```

Nos permite agregar las modificaciones al **índice**. Podemos especificar uno a varios path para indicar directorios o archivos que queremos agregar. Es el paso previo a generar commits.

```
git restore --staged <path>
```

Acción inversa de add. Remueve un archivo del index para pasarlo al workdir.

```
git rm --cached <path>
```

Mismo efecto que el anterior



Comandos básicos: commit & reset

```
git commit [-m <mensaje>, -a]
```

Genera un nuevo commit con los cambios que estén agregados al *índice*. Si no hay cambios no se generará nada.

El ID es autogenerado en base a muchas variables (contenido, autor, committer, mensaje, snapshot del código fuente). Por más que dos commits tengan el mismo contenido, tendrán distinto ID. Salvo que el repositorio sea un clon, **NO HAY DOS COMMITS CON EL MISMO ID** en ninguno de todos los repositorios en existencia, no sólo el propio. Se utiliza el algoritmo *SHA-256* para lograrlo.

Si no se indica un mensaje, se abrirá el editor de texto estipulado en gitconfig para setearlo.

```
$ git config --global core.editor <editor>
```

```
git reset <target commit> [--soft]
```

Con esta alternativa, el comando reset modificará el historial de commits "eliminando" todos los commits entre el actual y `<target commit>` y moviendo todas las modificaciones de los mismos al *workspace*. Si se usa `--soft` los cambios irán al *index*.



Git commit

Una vez que se hizo un commit, el mismo perdurará por siempre.

Cualquier cambio que hayamos al menos commiteado una vez nunca se perderá. Nuestro historial de commits nos permitirá ver cada uno de los cambios que fuimos haciendo.

Por más que haya comandos como `git reset` que modifiquen este historial de commits, todo commit que se haya hecho se puede referenciar, ver y recuperar los cambios que allí se contengan.

El hecho de tener commits, hace que Git cumpla con la idea de trackear todos los cambios que se hicieron, por quién y cuándo.



Conceptos básicos: HEAD

El último commit (el más nuevo, el que no tiene padre aún) tiene un sobrenombre, una etiqueta: HEAD.

En lugar de poner el ID del commit, se puede utilizar la etiqueta HEAD para facilitar la operación.

Al crear un nuevo commit, el HEAD actual será hijo del nuevo, y la etiqueta de HEAD pasará a ese nuevo commit.

Algunos comandos, como `git reset` modifican el historial de commits, y por lo tanto la referencia a HEAD también.



Conceptos básicos: repositorio remoto

Como mencionamos, el repositorio local va a contener todos los cambios que nosotros vamos haciendo.

Para poder compartir nuestro desarrollo con otras personas, debemos subir nuestros cambios a un repositorio remoto.

Para ello, debemos hacer dos cosas:

1. Crear un repositorio remoto vacío en el servicio de nuestra preferencia (GitHub es el más usado).
2. Agregar la referencia al remoto en nuestro local:

```
git remote add origin git@github.com:<usuario>/<nombre repo>.git
```

Estamos diciendo que estamos agregando un remoto a nuestro repositorio y le ponemos de nombre origin (convención) para referenciarlo más fácilmente.



Repositorios remotos

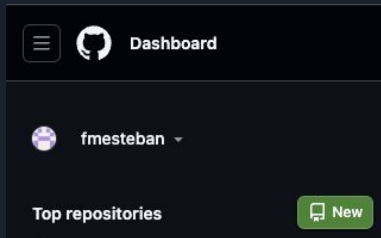
El tener el concepto de repositorios remotos hace que Git pueda permitir la colaboración entre distintos usuarios siguiendo distintas reglas de permisos que se pueden adaptar a cada casa de uso específico.

Cada usuario puede determinar quién y cómo tienen acceso a sus repositorios.

Cada empresa puede determinar quiénes de sus empleados tienen acceso a qué.

Crear repositorio en Github

Dentro del home, podemos ver a la izquierda nuestros repositorios y un botón para crear uno nuevo



Nos llevará a un formulario para completar los datos

Create a new repository

Repositories contain a project's files and version history. Have a project elsewhere? [Import a repository](#).
Required fields are marked with an asterisk (*).

1 General

Owner * **Repository name ***

/

Great repository names are short and memorable. How about **effective-bassoon**?

Description

0 / 350 characters

2 Configuration

Choose visibility *

Choose who can see and commit to this repository

Start with a template

Templates pre-configure your repository with files.

Add README ☐ Off

READMEs can be used as longer descriptions. [About READMEs](#)

Add .gitignore

.gitignore tells git which files not to track. [About ignoring files](#)

Add license

Licenses explain how others can use your code. [About licenses](#)



Repositorio remoto: credenciales

Para poder acceder a repositorios remotos debemos autenticarnos con nuestro usuario de Github

Los hostings de repositorios remotos manejan sus propios servicios de autenticación.

Cada vez que hagamos un push/pull, debemos indicar nuestras credenciales.

Para evitar este proceso engorroso se puede configurar la herramienta para que se autentique a través de un par clave privada/clave pública.

Para ellos, debemos seguir estos pasos [Paso1](#) y [Paso2](#) y **mucho muy importante**: usar la URL en formato SSH cuando se clone o se conecte a un repositorio remoto (NO USAR HTTPS).



Comandos básicos: push, fetch, merge & pull

```
git push [<remoto>] [<branch>]
```

Envía todos los commits nuevos que haya en el repositorio local al repositorio remoto `<remoto>`.

Para ser exitoso, no debe haber conflictos entre ambas versiones.

```
git fetch [<remoto>]
```

Actualiza las referencias del repositorio remoto `<remoto>` en el repositorio local.

```
git merge <remoto>/<branch>
```

Actualiza el repositorio local con los commits traídos previamente con el comando fetch del repositorio `<remoto>`. Puede generar conflictos si hay *diverging changes*.

```
git pull <remoto>/<branch>
```

Es una combinación de los dos comandos anteriores. Actualiza las referencias del remoto e intenta actualizar el repositorio local con esos cambios. Puede generar conflictos si hay *diverging changes*.



Qué es un conflicto?

Cuando se intentan combinar dos versiones del mismo repositorio en uno y los cambios no son compatibles entre sí, se genera un conflicto.

Git identifica cuáles son las partes que no ha podido combinar y las marca en los archivos que tienen estos conflictos.



Git conflicts: Ejemplo

```
→ mi_repo git:(main) git pull
Auto-merging data/trabajos.yml
CONFLICT (content): Merge conflict in data/trabajos.yml
Automatic merge failed; fix conflicts and then commit the result.
```

data/trabajos.yml

```
<<<<<< HEAD
  publicacion: 2018-04-06
  entrega: 2018-04-20
=====
  publicacion: 2018-09-14
  entrega: 2018-09-24
>>>>>> 2603ffef0d76dc00e0059354a93b442d58f1d147
```

HEAD -> El cambio local,
el mío

commit ID -> El cambio
remoto, de alguien más



Git Revert

Podemos deshacer un commit sin necesidad de modificar nuestro historial

```
git revert <commit>..<commit>
```

Crea un nuevo commit con los cambios inversos al rango de commits seleccionado.

Pueden generarse conflictos si el rango de commits seleccionado modifica archivos que fueron modificados en commits posteriores.



Comandos básicos adicionales

`git status`

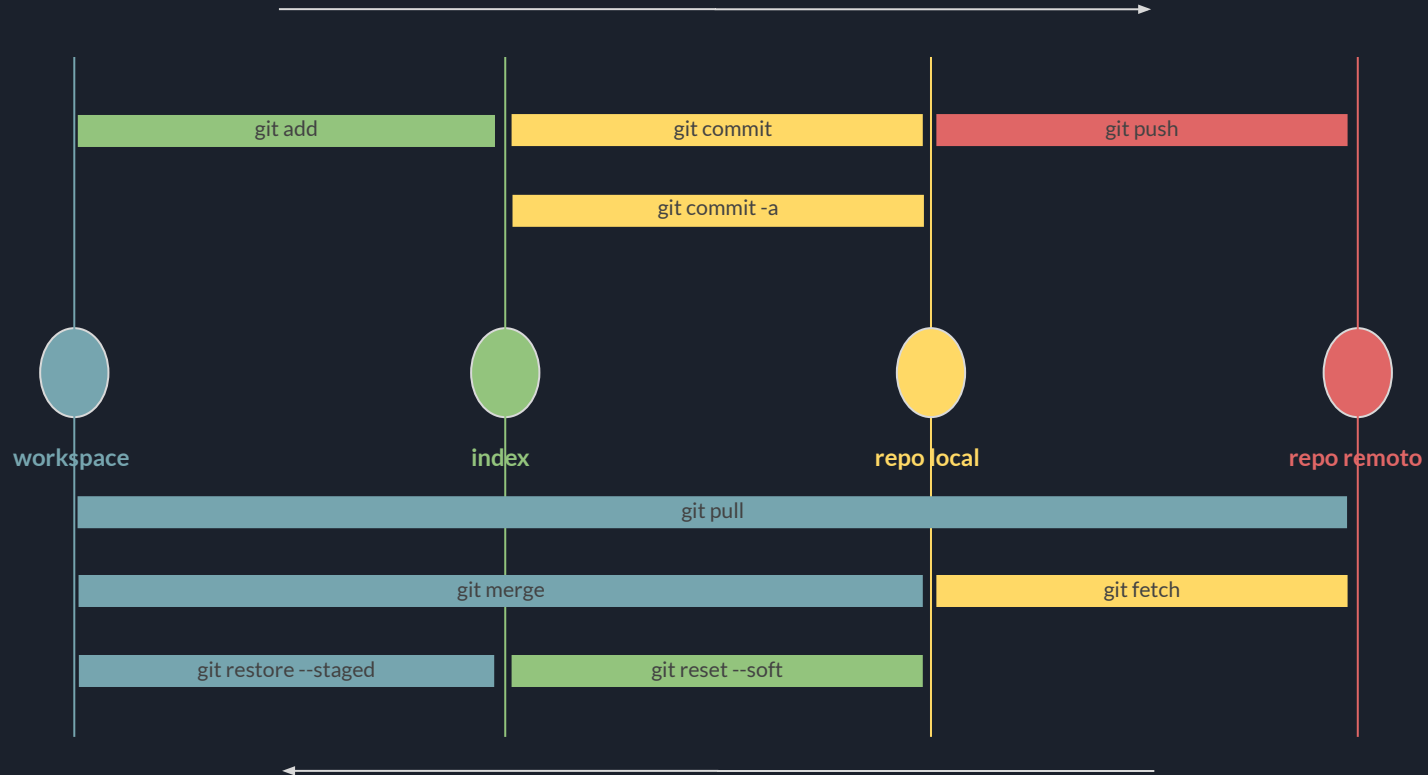
`git log`

`git diff`

`git show`

`git rm`

Comandos básicos: repaso





Análisis de Situaciones

1. Dev A y Dev B están trabajando cada uno en su cambios.
2. Dev A mergea y pushea sus cambios al remoto.
3. Dev B hace su commit y quiere llevar sus cambios al remoto.

Qué debe hacer dev A?

Solución

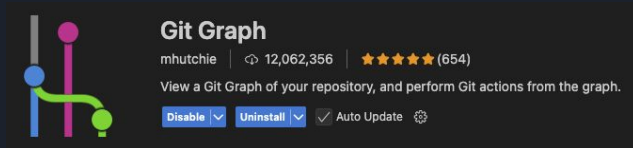
1. Dev B actualiza su repositorio con:
 - a. Merge con main
 - b. Rebase sobre main
2. Soluciona conflictos localmente si hay
3. Pushea nuevamente al remoto

Herramientas de visualización



gitk

VS Code git extension





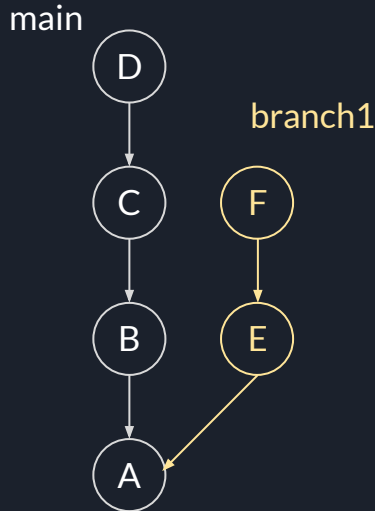
Manejo de Branches

TB022 - Esteban - Riesgo - Kristal

Branches

Para efectivamente trabajar entre muchas personas en un mismo proyecto, es necesario tener en simultáneo muchas versiones distintas que luego puedan ser combinadas fácilmente.

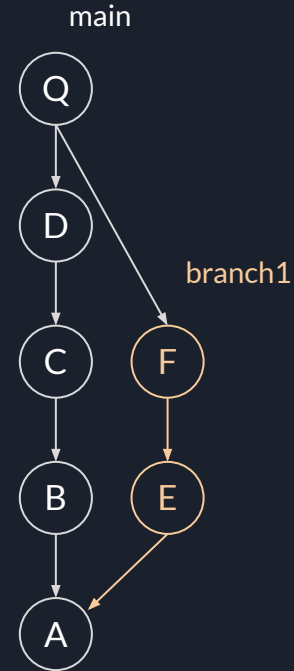
Una branch (rama) es una bifurcación en el historial



Un commit puede tener múltiples hijos, cada uno de una branch distinta.

Para unir las ramas, se generará un *merge commit*.

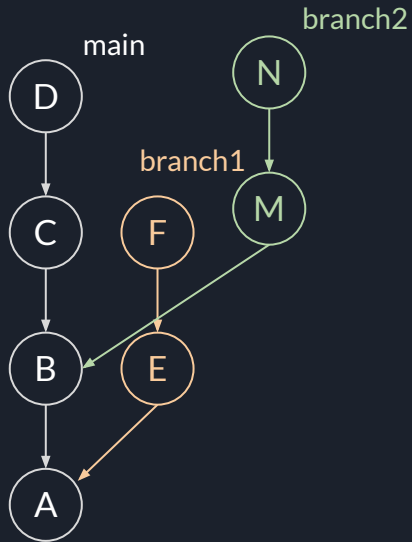
Un commit puede también tener entonces múltiples padres.



Branches

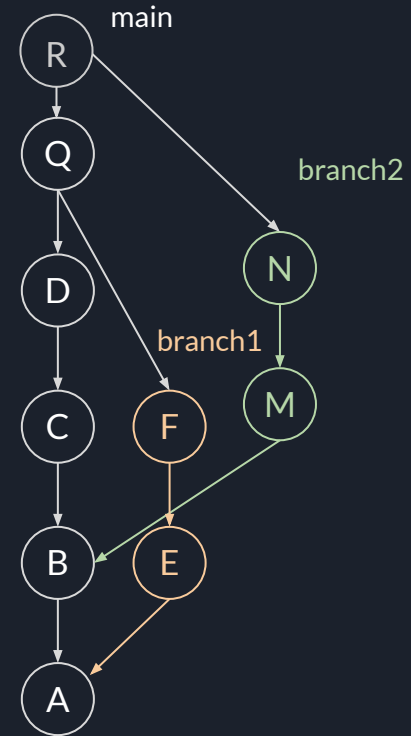
El beneficio de poder tener branches poder hacer cambios en paralelo minimizando la cantidad de conflictos y ordenando el trabajo.

Cada persona trabaja sobre una branch distinta.



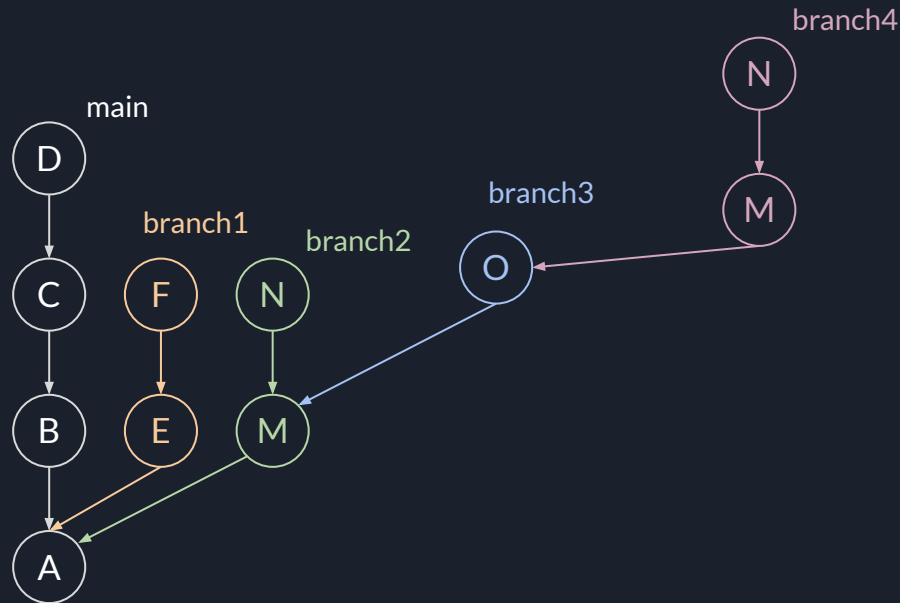
A medida que cada uno termina lo que estaba haciendo, lleva sus cambios a `main`. El resto, sigue haciendo sus cosas.

Cada nueva branch, va a salir de `main` cada vez que algún participante haga algún feature nuevo.



Branches

Cada branch puede partir de cualquier commit. Se termina generando una suerte de árbol



Por comodidad, es normal visualizar siempre la branch `main` a un costado, y todas las demás ramas del mismo lado.

`main` es una rama como cualquier otra, pero la vamos a tratar de manera preferencial. Es la rama principal de todos nuestros repositorios.

El historial de cada branch contiene sus propios commits y el de las branches que tienen commits que son padres de sus commits.



Branches: branch & checkout

```
git branch <nombre>
```

Crea una nueva branch en el commit actual. La branch original y la nueva apuntarán al mismo commit.

```
git switch [-c] <branch>
```


El repositorio local pasará a apuntar al último commit de la branch `<nombre>` y moverá la etiqueta HEAD a ese commit. Si se usa `-c` se creará la branch especificada antes de cambiar.

```
git restore <branch> <path>
```

Trae a la rama local, la versión de el/los archivo/s en `<path>` en la rama `<branch>`.

```
git checkout <nombre>
```

Es el comando original que permite hacer todos los comandos anteriores.



Branch name == commit ID

En todos los comandos que reciban un nombre de una branch, en realidad lo que reciben un commit ID.

El nombre de una branch sirve como referencia a un commit ID en particular. Por lo tanto, todos estos comandos que estuvimos viendo, pueden recibir un commit ID (o cualquier otro identificador hacia un commit).

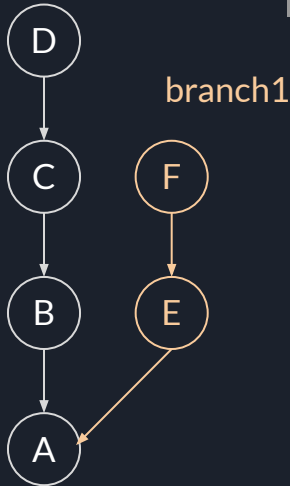
Branches: merge

```
git merge <branch>
```

Genera un nuevo commit **en la branch actual** combinando los cambios de la misma con los de **<branch>**

De haber conflictos al generar el merge no se creará el commit. Los conflictos y cambios se añadirán al workspace para ser solucionados y luego commiteados.

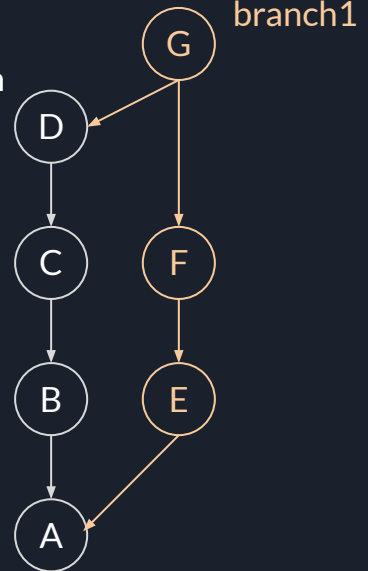
main



```
→ mi_repo git:(branch1) git merge main
```

El resultado acá es el opuesto al que veníamos mostrando. Estamos actualizando la branch **branch1** con los cambios que hay en **main**.

main



Branches: rebase

```
git rebase <branch>
```

Modifica el historial de commits de la branch actual de manera que todos los commits que no se encuentren en <branch> serán reaplicados uno por uno sobre el último commit encontrado.





Branches: rebase

Los commits de `branch1` se aplican uno por uno sobre los commits de `main`.

Si no hay conflictos, aplicará todos los commits (como no son los mismos de antes, aunque tengan exactamente el mismo contenido, tendrán **un nuevo ID**)

Si alguno de los commits presenta un conflicto, el rebase se detiene. Se deben solucionar los mismos, agregar los archivos al índice y luego continuar.

Para continuar, **NO HAY QUE HACER UN COMMIT NUEVO CON `git commit`**, sino que hay que usar la instrucción

```
git rebase --continue
```

Si en cambio se desea abortar el rebase

```
git rebase --abort
```

Branches: rebase -i

Modo interactivo del rebase. Permite elegir qué hacer con cada uno de los commits a los cuales se les aplica el rebase.

```
pick a0e2dc4f5de Add enrichment for metadata  
pick fb899fbaefd Fix tests
```

Commits sobre los cuales se aplica el rebase. El más antiguo arriba.

```
# Rebase fed6c127ed3..fb899fbaefd onto fed6c127ed3 (2 commands)
```

```
#
```

```
# Commands:
```

```
# p, pick <commit> = use commit
```

```
# r, reword <commit> = use commit, but edit the commit message
```

```
# e, edit <commit> = use commit, but stop for amending
```

```
# s, squash <commit> = use commit, but meld into previous commit
```

```
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous commit's log message
```

```
# x, exec <command> = run command (the rest of the line) using shell
```

```
# b, break = stop here (continue rebase later with 'git rebase --continue')
```

```
# d, drop <commit> = remove commit
```

```
# l, label <label> = label current HEAD with a name
```

```
# t, reset <label> = reset HEAD to a label
```

```
# These lines can be re-ordered; they are executed from top to bottom.
```

```
#
```

```
# If you remove a line here THAT COMMIT WILL BE LOST.
```

```
#
```

```
# However, if you remove everything, the rebase will be aborted.
```

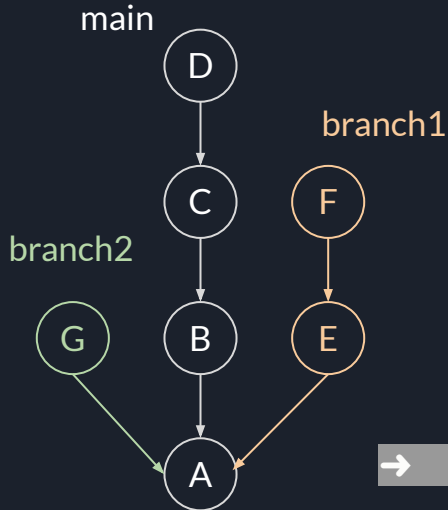
- Inicio del rango
- Último commit del rango
- HEAD commit de la branch sobre la cual se aplica el rebase

Todo esto (y más) se muestra en el editor de texto al hacer el rebase

Branches: cherry-pick

```
git cherry-pick <commit>
```

Intenta aplicar el commit indicado en la branch actual

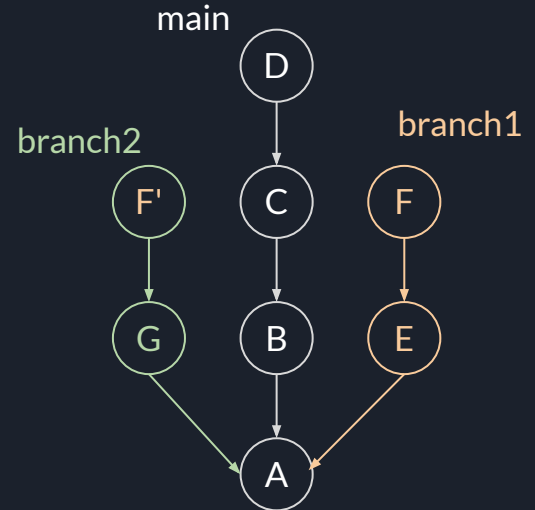


Igual que el rebase, no hay que hacer el commit manualmente, si hay conflictos:

```
git cherry-pick --continue
```

```
git cherry-pick --abort
```

```
→ mi_repo git:(branch2) git cherry-pick F
```





Git Tag

Un tag es una referencia a un cambio (un commit) particular en cualquier parte del repositorio.

Se pueden usar para identificar commits (para cualquiera de los comandos antes vistos).

Se suelen usar para indicar hitos importantes. Por ejemplo, alguna versión particular del software que se está armando.

```
git tag -a <tag_name> -m "<Mensaje de tag>"
```

Crea un tag con nombre `<tag_name>` y se le agrega el mensaje como metadata.

```
git tag -d <tag_name>
```

Elimina el tag del repositorio local

```
git push <remote> --tags
```

Para pushear los tags a un remoto, es necesario pasar el argumento `--tags`



Git Alias

En el archivo `.gitconfig` (puede ser global o local al repo) agregamos la siguiente sección:

```
[alias]
```

```
alias-name = "some command"
```

Ejemplo

Status

```
git config alias.st status
```

En vez de `git status` de ahora en más podemos hacer `git st`



Comandos avanzados adicionales

`git branch -D`

`git remote prune`

`git reflog`

`git commit --amend [--no-edit]`




Pull/Merge Requests

Es un pedido formal para mergear nuevos cambios propios en el repositorio remoto compartido, generalmente en la branch principal.

Los cambios deben ser revisados y aprobados por un tercero (otro integrante del equipo generalmente) para poder ser mergeados a la branch principal.

Se pueden configurar chequeos automáticos de calidad de código y cobertura de pruebas.

Al hacer el merge del Pull Request (PR) se pueden mantener todos los commits de la branch original o se pueden combinar en un único commit.



Flujo de trabajo en equipo

TB022 - Esteban - Riesgo - Kristal



Flujo de trabajo en un repo compartido

1. Existe una branch principal, por ejemplo `main` o `master`
2. **No se hacen commits a `main` directamente.**
3. Cada colaborador va a trabajar en una branch por cada nueva feature que quiera incluir.
4. Una vez que está conforme con lo que hizo, va a llevar sus cambios a `main` a través de un *Pull Request*.
5. Se realiza el proceso de revisión, corrección, resolución de conflictos y eventualmente se hace un merge de ese Pull Request a la branch principal.
6. Desde mi local, me traigo esos cambios nuevos (pull) y borro las branches viejas
7. Se repite el proceso para cada colaborador para cada nueva tarea a implementar.

Flujo de trabajo: integrar una nueva feature

Queremos agregarle nueva funcionalidad a nuestra aplicación

Estado Inicial

Repositorio Local

main



La rama `main` va a tener la versión última del programa.

Todo cambio que está en `main` está probado y debe estar funcionando.

Repositorio Remoto

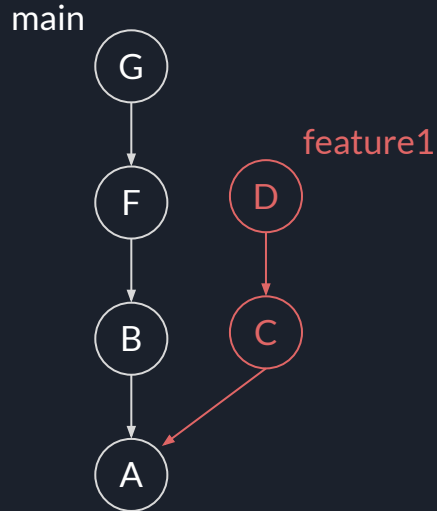
main



Flujo de trabajo: integrar una nueva feature

Yo empiezo a trabajar en una feature

Repositorio Local



Creo mi propia branch para trabajar en mis cosas.

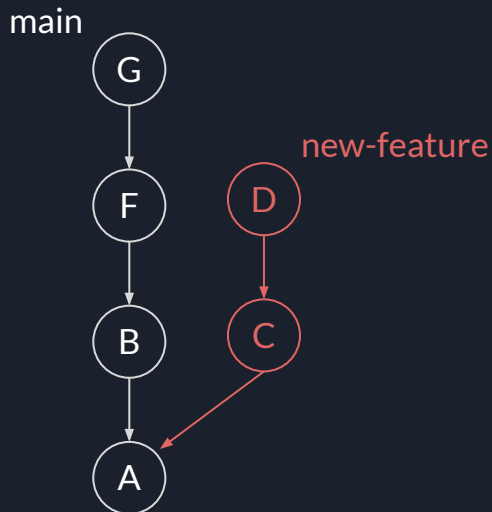
Repositorio Remoto



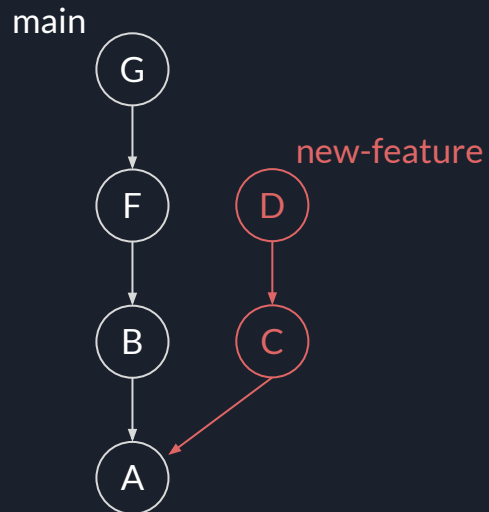
Flujo de trabajo: integrar una nueva feature

Pusheo mis cambios al remoto

Repositorio Local

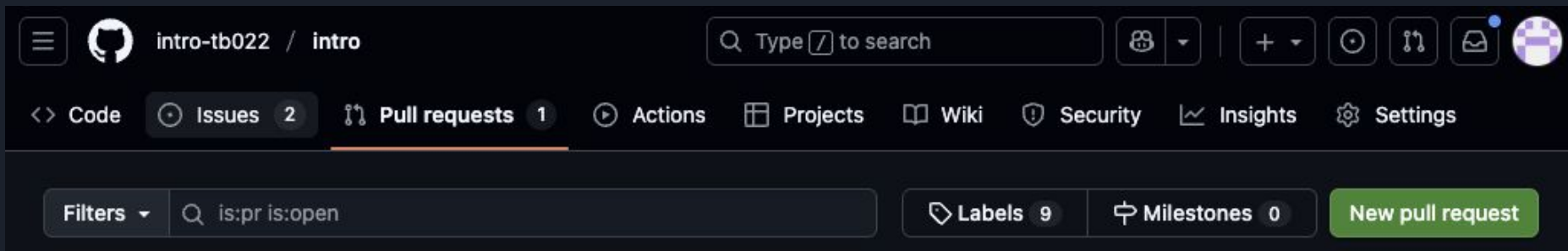


Repositorio Remoto



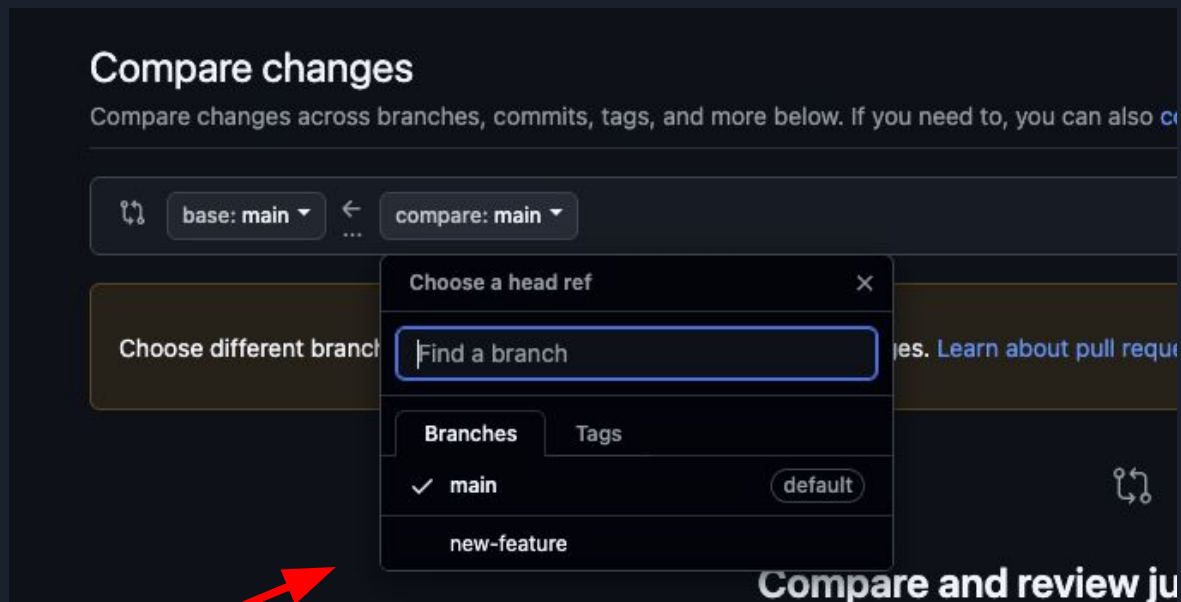
Flujo de trabajo: ejemplo

Cuando estoy listo, hago un pull request a main para que revisen mi feature



Flujo de trabajo: ejemplo

Hago un pull request a main para que
revisen mi feature



Flujo de trabajo: ejemplo

Previsualizo los cambios, le pongo una descripción al PR y lo creo

The screenshot shows the GitHub interface for creating a new pull request. At the top, the base branch is 'main' and the compare branch is 'new-feature'. The 'Add a title' field contains 'Cambio sol del ej1 para que resuelva ej2 tambien'. The 'Add a description' field contains 'Permito que ahora el script reciba de stdin y chequee si el archivo existe.' A red arrow points to the 'Create pull request' button. The right sidebar shows 'Reviewers' (No reviews), 'Assignees' (No one—assign yourself), 'Labels' (None yet), 'Projects' (None yet), 'Milestone' (No milestone), and 'Development' (Use Closing keywords in the description to automatically close issues). Below the description field, it says 'Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).' The commit history shows '1 commit' and '1 file changed'. The commit message is 'Cambio sol del ej1 para que resuelva ej2 tambien' by 'lmesteban' committed 3 minutes ago. The diff shows changes to 'ejercicio.sh' with 7 additions and 1 deletion. The diff content is:

```
@@ -1,7 +1,13 @@  
  
1 count=0  
  
1 + #!/usr/bin/bash  
2 +  
3 + count=0  
4 + if [[ "$1" ]] && [[ ! -f "$1" ]]; then  
5 +   echo $count  
6 +   exit 0  
7 + fi
```

Flujo de trabajo: integrar una nueva feature

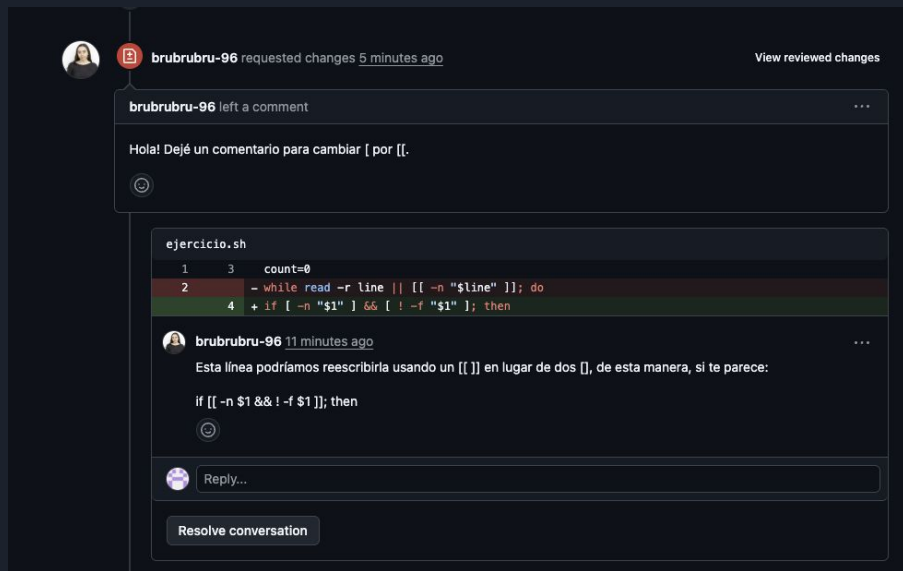
Con el PR creado, espero que alguien me haga un review

El sistema me permite mergear el pull request y llevar los cambios a `main`, pero **debemos** esperar que alguien lo revise.

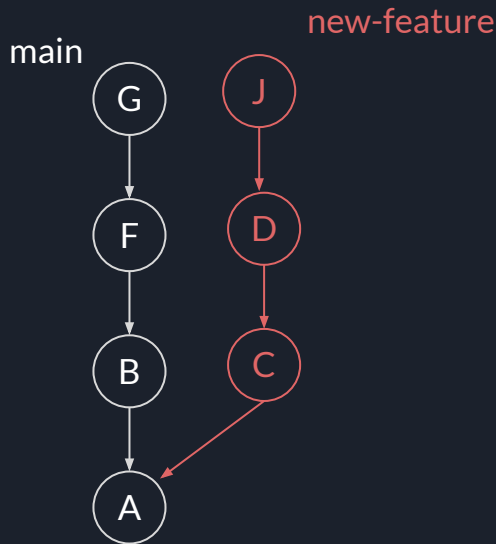


Flujo de trabajo: integrar una nueva feature

Reviso los comentarios que me hayan hecho. Si es necesario, implemento los cambios en mi repo local y vuelvo a hacer un push a la branch correspondiente. Vuelvo a pedir que revisen el PR.

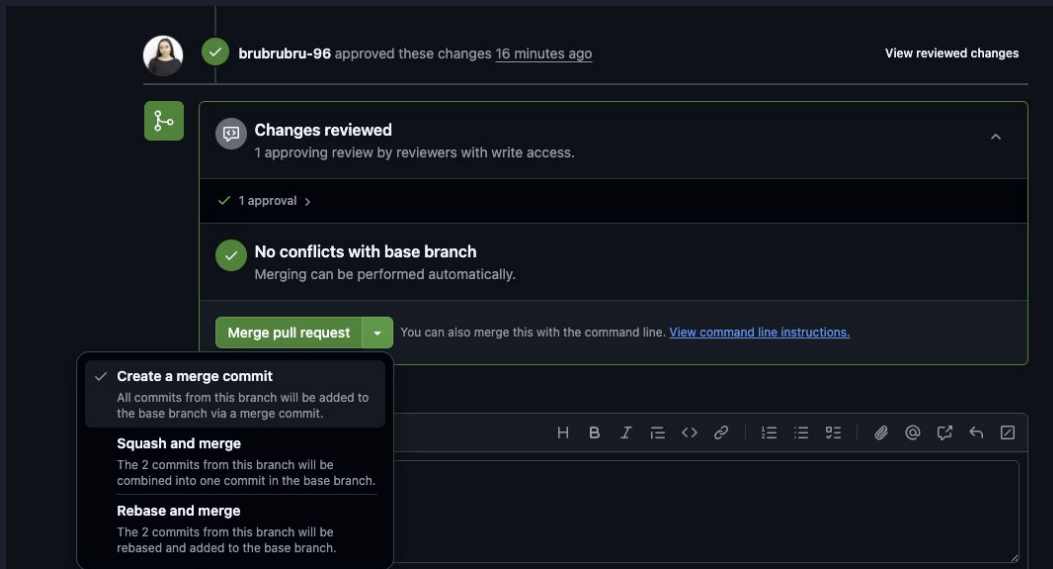


The screenshot shows a GitHub pull request interface. At the top, a user profile icon and the text "brubrubru-96 requested changes 5 minutes ago" are visible, along with a "View reviewed changes" link. Below this, a comment box shows a comment from "brubrubru-96" stating "Hola! Dejé un comentario para cambiar [por [[". Underneath the comment is a code diff for a file named "ejercicio.sh". The diff shows three lines: line 1 with "count=0", line 2 with "- while read -r line || [[-n \"\$line\"]]; do", and line 4 with "+ if [-n \"\$1\"] && [! -f \"\$1\"]; then". Below the diff, another comment from "brubrubru-96" 11 minutes ago suggests rewriting a line using "[[]]" instead of "dos []". The comment text is "Esta línea podríamos reescribirla usando un [[]] en lugar de dos [], de esta manera, si te parece: if [[-n \$1 && ! -f \$1]]; then". At the bottom, there is a "Reply..." input field and a "Resolve conversation" button.



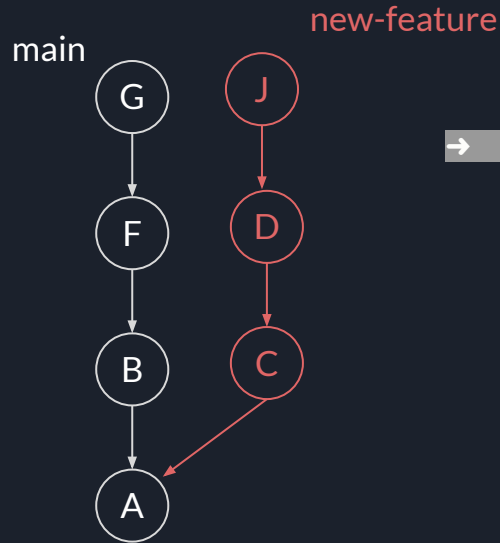
Flujo de trabajo: integrar una nueva feature

Una vez que el PR está aprobado. Lo mergeo en main.
Puedo combinar todos mis commits en uno solo (Squash and merge), o mantener el historial como está (Merge pull request).



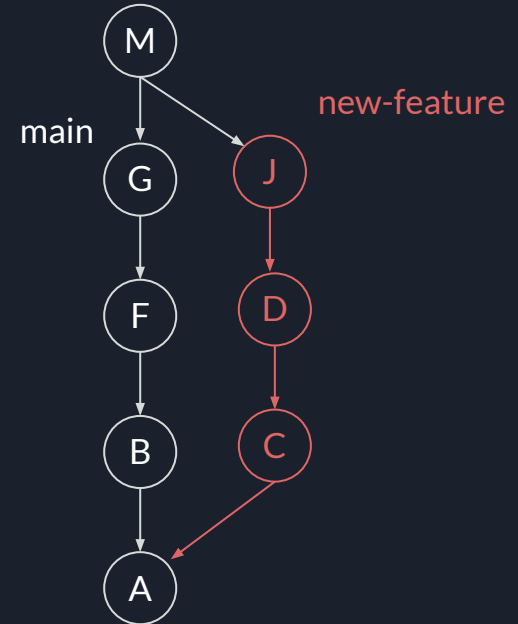
Flujo de trabajo: integrar una nueva feature

Una vez mergeado el PR necesito actualizar mi local



```
→ mi_repo git:(main) git pull origin main
```

Luego, para limpiar, borro la rama **new-feature** local y la remota.



Flujo de trabajo: revisar un PR

Cuando nuestros compañeros hagan su trabajo y creen sus PRs, debemos revisarlos y dejar nuestros comentarios en los mismos. Abrimos el PR y vamos a ver los cambios

Cambio sol del ej1 para que resuelva ej2 tambien #1

[Try the new experience](#) [Edit](#) [Code](#)

[Open](#) **fmesteban** wants to merge 2 commits into `main` from `new-feature`

[Conversation](#) 2 [Commits](#) 2 [Checks](#) 0 [Files changed](#) 1 +7 -1

[Changes from all commits](#) [File filter](#) [Conversations](#) [Jump to](#) [Settings](#) 0 / 1 files viewed [Review in codespace](#) [Review changes](#)

8 `ejercicio.sh`

Viewed

```
...      @@ -1,7 +1,13 @@
1      + #!/usr/bin/bash
2      +
1 3      count=0
4      + if [[ -n "$1" && ! -f "$1" ]]; then
5      +   echo $count
6      +   exit 0
7      + fi
2 8      while read -r line || [[ -n "$line" ]]; do
3 9          for word in $line; do
4 10             count=$((count + 1))
5 11             done
6      - done
12 + done < "${1:-/dev/stdin}"
7 13     echo "$count"
```

Flujo de trabajo: revisar un PR

Podemos hacer comentarios inline y dejar una review general aprobando o rechazando el PR

The screenshot displays a GitHub Pull Request (PR) review interface. At the top, navigation tabs include 'Conversation' (2), 'Commits' (2), 'Checks' (0), and 'Files changed' (1). A status bar on the right shows '+7 -1' with a green progress indicator. Below the tabs, a dropdown menu shows 'Changes from all commits', and a 'File filter' is active. The main area shows a diff for the file 'ejercicio.sh'. The diff highlights changes in green (additions) and red (deletions). The code includes a shebang, a count variable, and a loop that reads lines and counts words. A review sidebar on the right is titled 'Finish your review'. It contains a text area for a comment, a 'Write' button, and a 'Preview' button. Below the text area, there are options to 'Comment', 'Approve', or 'Request changes'. The 'Approve' option is selected. At the bottom right, there is a 'Submit review' button.

Conversation 2 Commits 2 Checks 0 Files changed 1 +7 -1

Changes from all commits File filter Conversations Jump to 0 / 1 files viewed Review in codespace Review changes

8 ejercicio.sh

```
... 1,7 +1,13
1 + #!/usr/bin/bash
2 +
1 3 count=0
4 + if [[ -n "$1" && ! -f "$1" ]]; then
5 + echo $count
6 + exit 0
7 + fi
2 8 while read -r line || [[ -n "$line" ]]; do
3 9 for word in $line; do
4 10 count=$((count + 1))
5 11 done
6 - done
12 + done < "${1:-/dev/stdin}"
7 13 echo "$count"
```

Finish your review

Write Preview H B I ≡ <> 🔗 ⋮ ⋮ ⋮ 📎 @ ...

Leave a comment

Markdown is supported Paste, drop, or click to add files

☐ **Comment**
Submit general feedback without explicit approval.

☒ **Approve**
Submit feedback and approve merging these changes.

☐ **Request changes**
Submit feedback that must be addressed before merging.

Submit review

© 2025 GitHub, Inc. Terms Privacy Security Status



Análisis de Situación 1

1. Dev A partió de main con su branch deva
2. Dev B partió de main con su branch devb
3. Dev A mergea sus cambios primero, Dev b hace un PR y no lo puede mergear porque tiene conflictos

Qué debe hacer?

Solución

1. Dev B actualiza su branch con:
 - a. Merge con main
 - b. Rebase sobre main
2. Soluciona conflictos localmente según el caso
3. Pushea nuevamente al remoto (con -f si hizo un rebase)



Análisis de Situación 2

1. Dev A partió de main con su branch deva
2. Hace los commits J, K
3. Dev B necesita algunos de los cambios que él tiene. Crea su branch devb partiendo desde deva, en el commit K, y hace sus propios cambios en los commits L, M
4. Dev A hace un PR y mergea sus cambios a main
5. Dev B hace un PR pero tiene conflictos.

Qué debe hacer?

Solución

1. Actualiza su branch con rebase a main (usando el modo interactivo o seleccionando el rango)
2. Selecciona sólo los commits que él hizo, L y M
3. Soluciona conflictos localmente que pueda haber
4. Pushear nuevamente al remoto con -f



Análisis de Situación 3

1. Dev A partió de main con su branch
2. Hace los commits J, K
3. Dev B necesita algunos de los cambios que él tiene. Crea su branch devb partiendo desde deva, en el commit K, y hace sus propios cambios en los commits L, M
4. Dev A modifica el historial de commits, haciendo que los commits que tenía Dev B, él ya no los tenga y viceversa; y luego hace un PR y mergea a master.
5. Dev B termina su funcionalidad y quiere hacer el PR

Qué debe hacer?

Solución

1. Actualiza su branch con rebase a main (usando el modo interactivo o seleccionando el rango)
2. Selecciona sólo los commits que él hizo, L y M
3. Soluciona conflictos localmente que pueda haber
4. Pushear nuevamente al remoto con -f



Forks

Es una desviación del repositorio original.

Un *fork* es un clon de un repo puntual, pero copiando todo el contenido en un nuevo repo remoto.

Los hosting de repositorios permiten crearlos desde su interfaz.

Por ejemplo, los repos de los tps que ustedes tienen son forks de un repo base el cual tiene la consigna.

Dato de color: en *GitHub* los *PullRequests* se llaman así porque originalmente estaban orientados a usarse en forks.

Para poder participar de proyectos open source cada developer que quiera colaborar se hace un fork del repositorio, hace sus cambios en su propio repositorio, y luego hace un *Pull Request* de su fork al repositorio original.



.gitignore

Este archivo indica qué otros archivos deben ser ignorados por Git para no incluirse en el repositorio.

Es muy común tener archivos autogenerados al compilar o correr código. O también archivos de configuración del IDE.

Estos archivos nunca vamos a querer subirlos al repositorio. Para este caso vamos a usar el .gitignore.

El en root path del repositorio debemos crear un archivo .gitignore donde incluiremos los paths a todos los archivos que queremos excluir de ser subidos al repo. Se pueden incluir archivos puntuales, directorios enteros, o grupos de directorios siguiendo las reglas de wildcards que se aplican en la terminal.



Enlaces útiles

Manual de usuario oficial

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

Herramientas interactivas

<https://learngitbranching.is.org/>

<https://ohmygit.org/>



Bonus Tracks



Semantic Versioning

Los programas de software que se publican para ser usados por otras personas, suelen (hoy en día, todos) tener distintas versiones durante el tiempo.

Cómo se organizan estas versiones? Una técnica muy usada es la de [Semantic Versioning](#) (semver).

Una versión particular cuenta de tres partes:

`MAJOR.MINOR.PATCH`

Ejemplos: 1.2.4, 1.120.0, 2.0.0, 0.7.123

Donde el orden de versiones es $0.7.123 < 1.2.4 < 1.120.0 < 2.0.0$ etc.

PATCH: el número de menor denominación. Reservado para resolución de bugs que son retrocompatibles con la versión anterior.

MINOR: cambios a la funcionalidad del software (nuevas features) que siguen siendo retrocompatibles con versiones anteriores.

MAJOR: cambios que no son retrocompatibles con versiones anteriores.

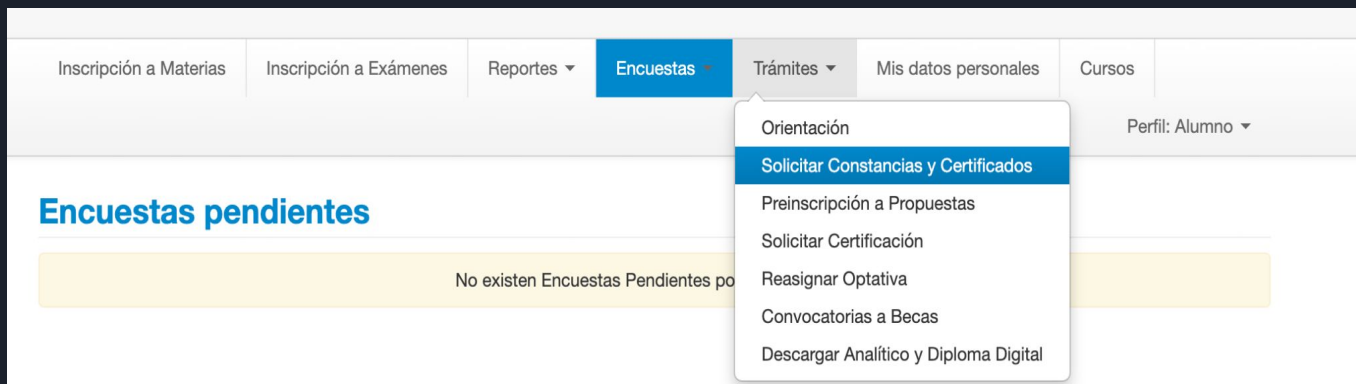


Github Education

Es un pack que contiene gran cantidad de herramientas que normalmente no están disponibles de manera gratuita.

Para obtener GitHub Education es presentar una prueba de que somos alumnos de una facultad.

Se puede descargar la constancia de alumno regular en el siu guaraní de FIUBA. En la sección de trámites y luego en solicitar constancias y certificados.



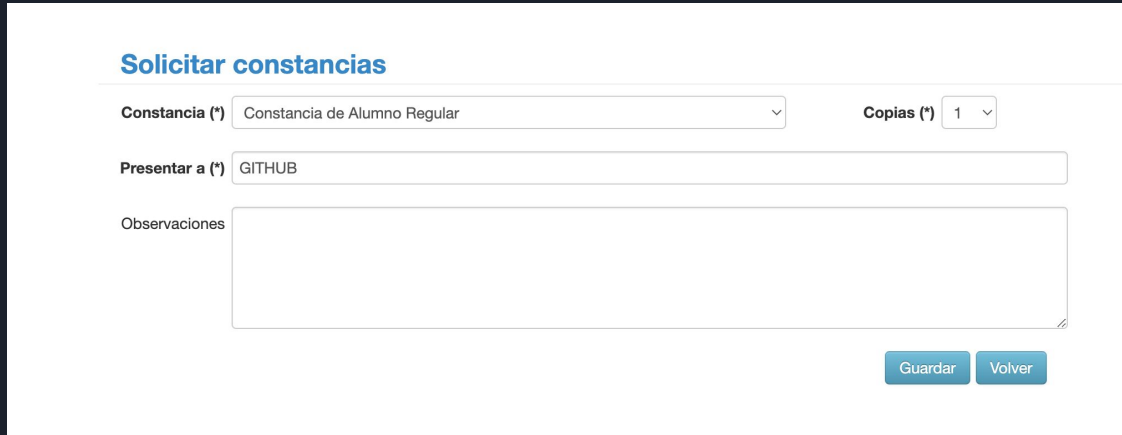


Github Education

Luego de eso estarán en la página de solicitudes. Para iniciar una nueva, deben hacer clic en el botón 'Nueva solicitud'.

Al hacer clic, serán redirigidos a la página donde podrán completar la solicitud de la constancia.

En esta van a tener que elegir en donde dice constancias la de alumno regular y a quién se lo van a presentar, en este caso Github.



Solicitar constancias

Constancia (*) Copias (*)

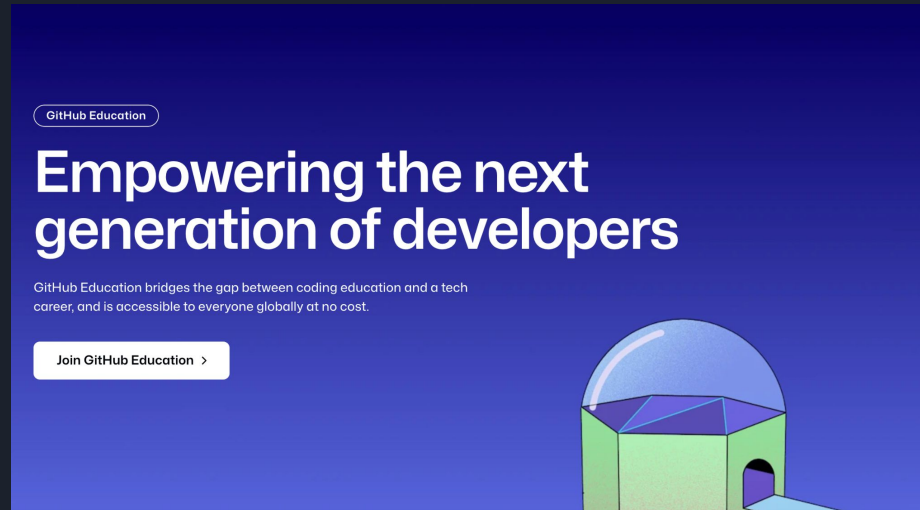
Presentar a (*)

Observaciones



Github Education

Una vez completado el paso anterior, deberán descargar el PDF de la constancia y tomarle una captura de pantalla. Ya teniendo la constancia, ahora si podremos ingresar a la página de Github Education e ir a “Join Github Education”





Github Education

Al ingresar, deberán seleccionar o verificar que esté seleccionado el rol de estudiante.

En esa misma página deben ir hasta abajo de todo para ingresar el nombre de la facultad en la que estudian. En caso de que no aparezca, ingresan el nombre completo y continúan.

Application

Required fields are marked with an asterisk *

What is the name of your school? *

Note: If your school is not listed, then enter the full school name and continue. You will be asked to provide further information about your school on the next page. **A minimum of two characters is required to find your school.**

When you click "Continue" you will be prompted to share your location with us. Providing your current location helps us verify your affiliation with your chosen school.

Continue



Github Education

En el anteúltimo paso, verán una página con un mapa. En la parte superior, deberán seleccionar “Higher Education” y, en la inferior, ingresar la dirección de la Facultad de Ingeniería.

Haciendo eso algunas casillas para ingresar datos más abajo se completan solas, pero otras tendrán que hacerlo ustedes con sus datos

Instructions

Your location is displayed in the map below.
Please select your school from the list of school campuses near to you.

You may move your location on the map or zoom in/out to display additional school campuses near to you.

If you are not currently near your school campus, then use the search box at the bottom of the list to find your school.

Type of School


- ☒ Higher Education (university, college)
- ☐ Secondary Education (high school, middle school)
- ☐ Informal Learning (bootcamp, code school)

Github Education

Ahora sí, luego de completar ese paso, deberán subir la constancia de alumno regular como prueba de sus estudios e indicar el tipo de documento, seleccionando “Dated enrollment letter on school letterhead”.

Please upload proof of your academic status.

Snap a picture of your qualifying proof of current academic status using your HD webcam or smartphone camera.



Take a picture
or [upload a photo](#)

Photo must be JPEG or PNG, at least 1024x768 resolution, and between 100KB and 1MB in size

Proof Type *

3. Dated enrollment letter on school letterhead ▾

Good proof types indicate academic affiliation documentation most likely to help you be approved
Fair proof types may lower your chances of being approved
Poor proof types are unlikely to be acceptable

Importante: Luego de enviar la foto, recibirán un mail de GitHub, que puede tardar algunos días.



Fin