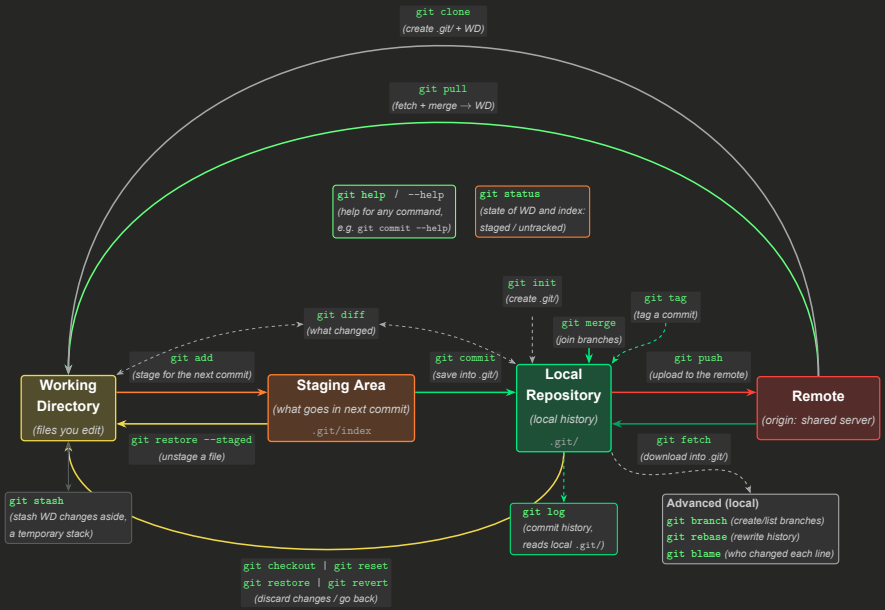
Git para operaciones (2/2)



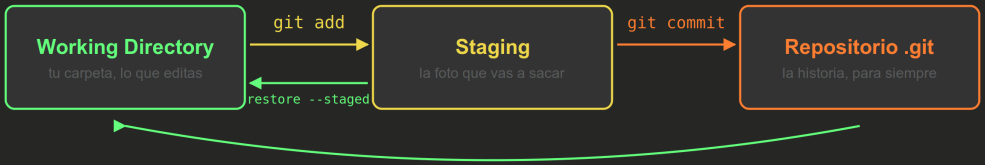
Introducción: contexto

Taller de Git, segunda de dos sesiones.

- Ya sabemos guardar, buscar y volver atrás, solos.
- Hoy: el trabajo sale del computador.
- Además: nombrar la entrega y reparar lo que se rompió.

Git no solo guarda: respalda, sincroniza y permite volver atrás.

git reset: repaso de la clase 1



`git restore .`
volver al último commit

`git status` ¿qué cambió?

`git diff` ¿en qué cambió?

nada sale de tu computador

git help: los 16 comandos de la clase 1

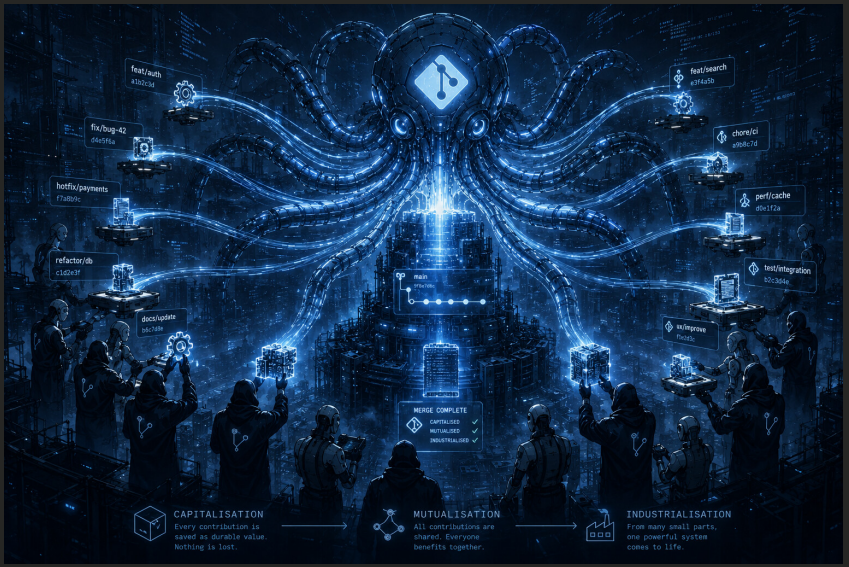
Iniciar	Capitalizar	Historiar	Recuperar
help	diff	log	restore
init	add	shortlog	checkout
config	commit	blame	stash
status	show	grep	clean

Ninguno salía de tu computador. Eso cambia hoy.

git fetch: contenido

Capítulo	Sesión 1	Sesión 2	Sesión 3	Sesión 4
Mutualizar	clone	remote	pull	push
Ramificar	branch	switch	merge	revert
Entregar	tag	show	describe	for-each-ref
Reparar	reflog	reset	rebase	filter-repo

Capítulo 5: Mutualizar



git clone

Traer un repositorio que vive en un servidor.

```
git clone https://github.com/tinmarino/stock
cd stock
```

Bajar todo: toda la historia de todos los archivos.

git clone (liviano)

Cuando el repo es enorme y no necesitas todo.

```
git clone --depth 1 <url>       # solo lo último (shallow)
git fetch --unshallow           # traer la historia después

git clone --filter=blob:none --sparse <url>   # monorepo
git sparse-checkout set stock/ tests/         # elige carpetas
```

No siempre se necesitan los 10 años de historia.

git remote

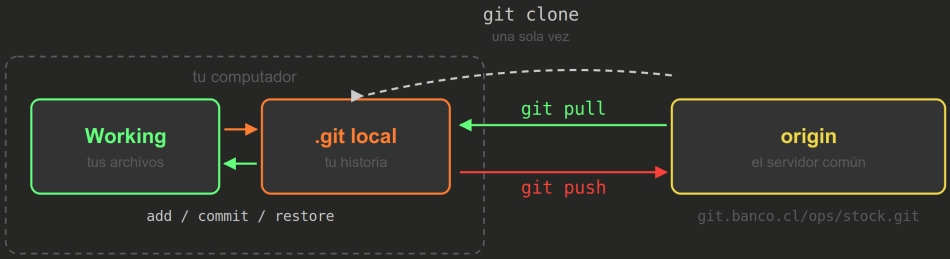
‘origin‘ es solo un apodo para una URL.

```
git remote -v
# origin  https://git.example.com/ops/stock.git (fetch)
# origin  https://git.example.com/ops/stock.git (push)

git remote add origin https://git.example.com/ops/stock.git
git remote set-url origin git@git.example.com:ops/stock.git
```

- Un repo local puede hablar con varios remotos.

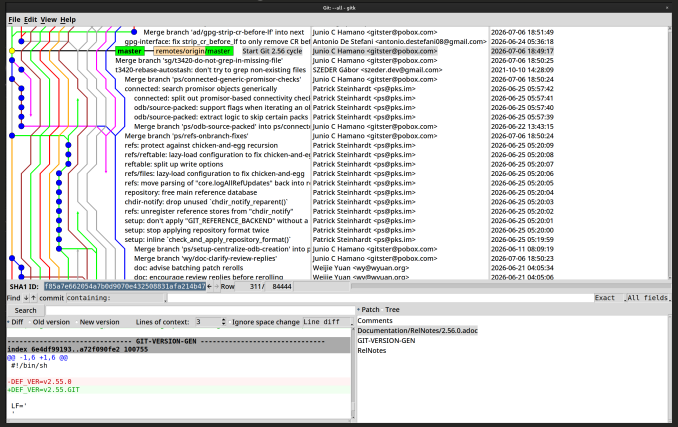
git remote: del computador al servidor



Primero recibir, después enviar.

git pull (merge)

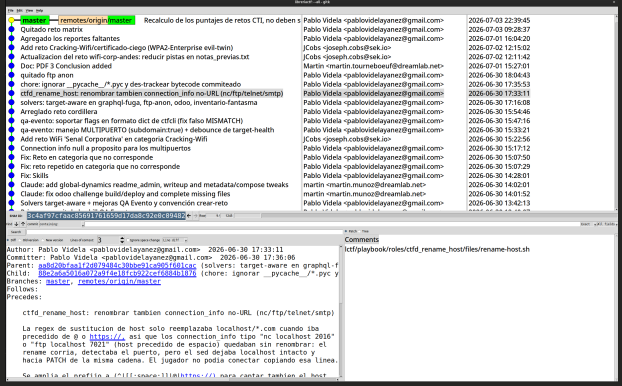
`git pull` = `fetch` + `merge`: juntar el trabajo local y el remoto.



Dar es recibir: primero recibir lo de los otros.

git pull `--rebase` (línea recta)

Reaplica los commits locales encima de lo remoto: una sola línea, sin nudos.



`--rebase`: historia lineal, sin «merge branch main».

git push

Enviar el trabajo al servidor común.

```
git push
# To git.example.com/ops/stock.git
#    aa62183..d1bb173  main -> main

git push -u origin main    # fija el upstream
git push                   # de ahí en más, basta esto
```

Recién ahí el trabajo es de todos.

git push `--force`: no

`--force` reescribe la historia del servidor: los demás pierden commits.

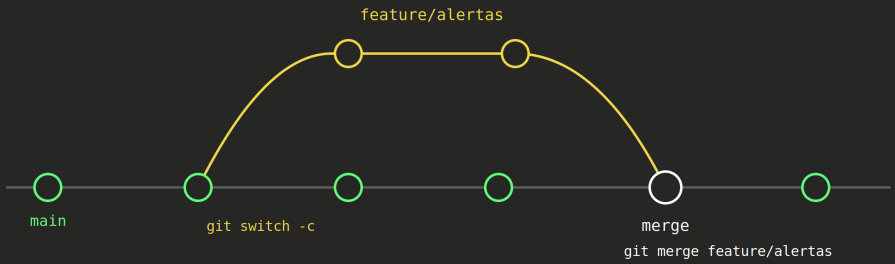
```
git push --force              # nunca sobre main
git push --force-with-lease # en tu rama, si hay que hacerlo
```

- En `main` la rama está protegida: sin force, con revisión de un par.

Lo compartido no se reescribe: se corrige hacia adelante.

Capítulo 6: Ramificar

Una rama para no ensuciar main



Git junta solo lo que no choca; el resto es conflicto.

git branch

Trabajar en paralelo sin tocar `main`.

```
git branch                  # listar las ramas locales
git branch -a               # incluye las del remoto
git branch -d experimento   # borrar (segura)
```

Una idea nueva no debería ensuciar lo que ya funciona.

git switch

```
git switch -c feature/alertas   # crear rama y cambiar
git switch main                 # volver
git switch -                    # a la anterior

git push -u origin feature/alertas   # publicarla
```

Una rama es un post-it sobre un commit, no una copia.

git merge

Juntar dos líneas de trabajo en una.

```
git merge feature/alertas
# Merge made by the 'recursive' strategy.
```

- Git combina solo lo que no choca.
- Si dos tocaron lo mismo: conflicto (se resuelve a mano).

El trabajo de cada uno se mutualiza.

git merge: conflicto, quedarse con un lado

No siempre hay que editar a mano: puedes elegir un lado entero.

```
git checkout --ours   config.py   # tu versión
git checkout --theirs config.py   # la que entra

git merge -X theirs feature       # gana feature
git merge --abort                 # el botón de pánico
```

`--abort` deja todo como estaba.

git revert

Deshacer un commit ya compartido, de forma segura.

```
git revert d1bb173
# crea un commit nuevo que anula al anterior

git revert -m 1 <merge>   # deshacer un merge
```

- No borra historia: la corrige hacia adelante.
- Seguro incluso si ya está en el servidor.

Capítulo 7: Entregar

El tag: un nombre para una entrega

`vX.Y.Z`	Release . SubRelease . Build
`-prod`	sufijo del tag que sí llegó a producción

git tag

Un nombre estable para un commit: la entrega.

```
git tag                          # listar
git tag v1.2.0                   # liviano: solo un puntero
git tag -a v1.2.0 -m "Release julio 2026"   # anotado

git push origin v1.2.0           # el tag NO se sube con git push
git push --tags                  # todos los tags
```

- El commit se mueve; el tag no.

git tag: anotado o liviano

Para una entrega formal: siempre anotado.

```
git tag -a v1.2.0 -m "Release julio 2026"   # autor y fecha
git tag -s v1.2.0 -m "Release julio 2026"   # y firmado (GPG)

git cat-file -p v1.2.0          # el objeto del tag
git tag -v v1.2.0               # verificar la firma
```

- El liviano no deja quién ni cuándo: no sirve como evidencia.

Un tag anotado y firmado es trazabilidad para auditoría.

git tag: nomenclatura `vX.Y.Z`

Parte	Significa	Sube cuando
`X`	Release	cambia el alcance funcional
`Y`	SubRelease	se agregan funciones menores
`Z`	Build	corrección o recompilación

- Sufijo `-prod`: el tag que sí entró a producción.
- Sin sufijo: candidato que quedó huérfano.

‘v1.2.0-prod‘ responde «¿qué versión está arriba?».

git show, sobre un tag

Ver y comparar entregas.

```
git show v1.2.0                       # el tag y su commit
git show --stat v1.2.0                # qué archivos cambiaron
git diff v1.1.0..v1.2.0               # todo lo que entró
git diff --stat v1.1.0-prod..v1.2.0-prod

git checkout v1.1.0-prod              # reproducir la prod
```

El changelog de la entrega sale del repositorio.

git describe

¿En qué punto exacto estoy, con un nombre legible?

```
git describe --tags
# v1.2.0-4-g3164b36
#  ^tag    ^commits   ^SHA corto

git describe --tags --dirty
# v1.2.0-4-g3164b36-dirty   <- hay cambios sin commitear
```

- Sirve como número de versión del artefacto que se compila.

Si dice ‘-dirty‘, ese binario no es reproducible.

git for-each-ref

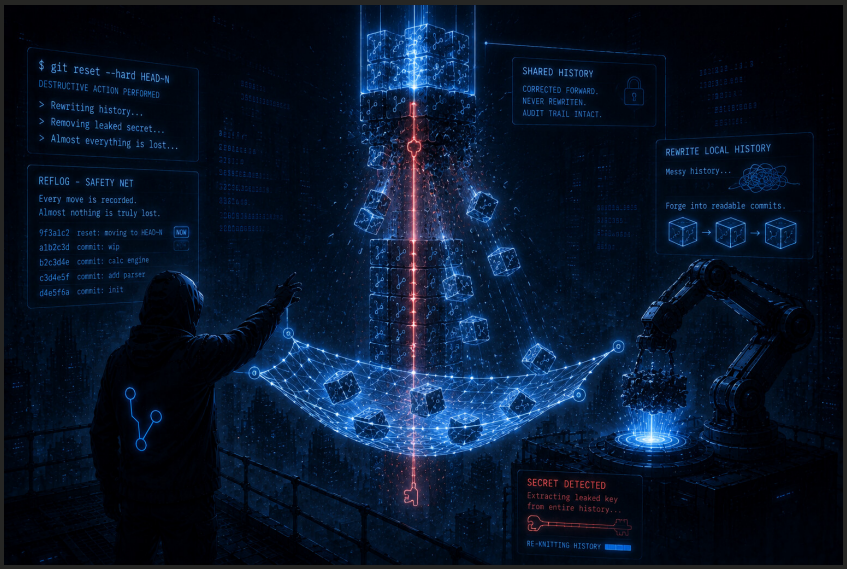
Los tags, ordenados por fecha: la historia de las entregas.

```
git for-each-ref --sort=-creatordate \
  --format='%(refname:short) %(creatordate:short) %(subject)' \
  refs/tags

git tag --list 'v1.*-prod'      # solo lo que entró a producción
git tag --contains d1bb173      # ¿en qué entrega salió?
```

«¿Ese parche está en prod?» es una consulta, no un correo.

Capítulo 8: Reparar



git reflog: la red de seguridad

Recuperar commits “perdidos” tras un reset o rebase equivocado.

```
git reflog                    # cada movimiento de HEAD
git reset --hard HEAD@{2}     # vuelve 2 pasos atrás
```

En Git casi nada se pierde de verdad.

git reset: las 3 caras

```
git reset --soft  HEAD~1   # deja los cambios STAGED
git reset --mixed HEAD~1   # cambios sin stagear (default)
git reset --hard  HEAD~1   # BORRA los cambios (¡peligro!)
```

- `--soft` / `--mixed`: seguros. `--hard`: peligroso (usa reflog para volver).

‘reset‘ es local; para lo compartido, ‘revert‘.

git rebase -i: reescribir lo local

Limpiar 10 commits caóticos en 3 legibles antes de compartir.

```
git rebase -i HEAD~5     # pick / squash / reword / drop
```

Reescribe libre lo tuyo; nunca lo que ya compartiste.

git filter-repo: borrar un archivo de TODA la historia

Un secreto que se coló: borrarlo con un commit nuevo no basta.

```
pip install git-filter-repo
git filter-repo --path data/secretos.env --invert-paths
git push --force --all        # y avisar: todos deben reclonar
```

- Antes se hacía con `git filter-branch` (lento, propenso a errores).
- Si el secreto llegó a un remoto: considéralo comprometido, rótalo.

Conclusión: lo aprendido

Mutualizar	Ramificar	Entregar	Reparar
1. clone	5. branch	9. tag	13. reflog
2. remote	6. switch	10. show	14. reset
3. pull	7. merge	11. describe	15. rebase
4. push	8. revert	12. for-each-ref	16. filter-repo

16 más: ahora el trabajo es de todos.

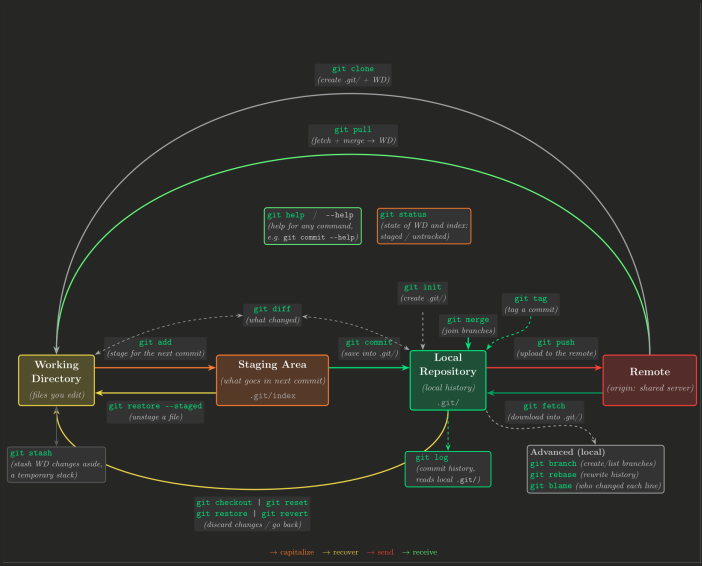
git worktree: la rutina diaria (sync)

```
#!/usr/bin/env bash

sync(){
  git commit -a -m "${1:-new commit}"   # snapshot
  git pull --rebase                     # download and rebase
  git push                              # upload
}
```

Guardar seguido, con mensajes claros, y compartir.

git worktree: las áreas de Git



git worktree: Git por dentro (plumbing)

Git es una base de datos de objetos direccionados por su SHA.

```
git rev-parse HEAD              # una ref -> su SHA
git cat-file -t <sha>           # blob / tree / commit / tag
git cat-file -p <sha>           # contenido "pretty" del objeto
git hash-object stock/cli.py    # el SHA de ese contenido
```

Un commit = un árbol + metadatos, todo con su huella SHA.

git worktree: otros comandos (avanzado / plumbing)

Plumbing (bajo nivel)

```
git cat-file
git hash-object
git rev-parse
git ls-tree
git ls-files
git write-tree
git symbolic-ref
git update-index
git for-each-ref
git count-objects
git show-ref
```

Reescribir historia

```
git rebase
git filter-repo
git replace
git reflog
```

Buscar / inspeccionar

```
git blame
git grep
git bisect
git shortlog
git describe
git whatchanged
```

Remoto / transporte

```
git fetch
git remote
git bundle
git format-patch
git am
git archive
git request-pull
```

Mantenimiento

```
git gc
git fsck
git maintenance
git prune
git repack
```

Otros útiles

```
git worktree
git notes
git rerere
git clean
git cherry-pick
git mergetool
git submodule
```

git grep: ejercicios de mutualizar

Repo: `git clone https://github.com/tinmarino/stock && cd stock`

#	Reto	Pista
5	Entre `diff-antes` y `diff-despues`: la flag.	`git diff tag1 tag2`
7	Rama `pista/rama-oculta`: un archivo.	`git branch -a`
10	Un merge tiene secreto en su body.	`git log --merges`
11	`pista/genesis` tiene su propio root.	`git rev-list`

git grep: ejercicios de entregar y reparar

#	Reto	Pista
8	Tag `oculto`: en su **objeto**, no el commit.	`git cat-file -p`
17	Commit “perdido” tras un reset.	`git reflog`
19	El tag más **nuevo** trae la flag.	`git for-each-ref`
20	v0.01 SHA 8 chars = nombre de un tag.	`git rev-parse`

git grep: ejercicios de hooks

`cp hooks/* .git/hooks/` activa los hooks del repo.

#	Reto	Pista
21	Activa hooks y commitea `ENTREGA.md`.	`post-commit`
22	En `pista/para-rebase`: squash 3 en 1.	`post-rewrite`
23	Cambia a `pista/destino-secreto`.	`post-checkout`

11 flags hoy; 23 en total con la clase 1.

git grep: flags (solucionario)

#	Flag
5	flag-05-diff-tags
7	flag-07-branch
8	flag-08-cat-file
10	flag-10-merge
11	flag-11-genesis
17	flag-17-reflog

#	Flag
19	flag-19-tag-date
20	flag-20-rev-parse
21	flag-21-hook-commit
22	flag-22-hook-rebase
23	flag-23-hook-checkout

git notes: preguntas y referencias

¿Preguntas?

Recurso	URL
Pro Git (libro gratis)	https://git-scm.com/book/es/v2
Learn Git Branching	https://learngitbranching.js.org
Semantic Versioning	https://semver.org/lang/es
Conventional Commits	https://www.conventionalcommits.org/es

Guardar, compartir, entregar con nombre.