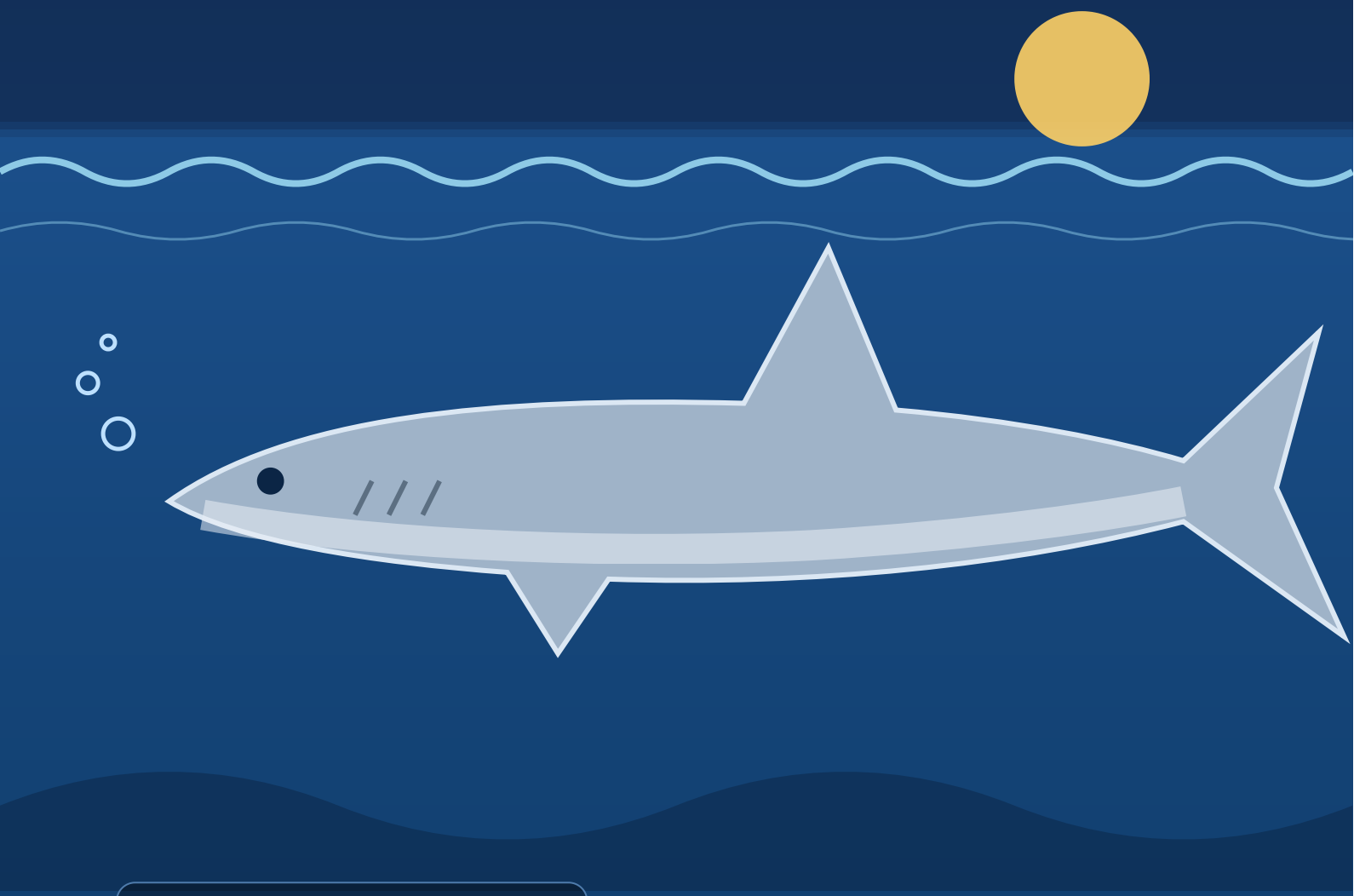
GUÍA PRÁCTICA DE COMANDOS

Git y GitHub desde la terminal

Del primer commit al Pull Request, sin salir de la línea de comandos



```
$ git push origin main
```

Marlon Falcón Hernández

Edición 2026

Git y GitHub desde la terminal

Introducción

Este libro enseña a trabajar con Git y GitHub sin salir de la terminal: desde la primera configuración hasta Pull Requests, releases y automatización. Cada capítulo combina una explicación breve con los comandos exactos que se escriben en el día a día.

Git es un sistema de control de versiones distribuido: guarda la historia completa de un proyecto en tu propio equipo. **GitHub** es una plataforma que aloja repositorios Git en la nube y añade colaboración: Pull Requests, Issues, revisiones, releases y Actions. Git funciona sin GitHub; GitHub no funciona sin Git.

Trabajaremos con dos herramientas de línea de comandos:

- `git`: el cliente oficial de Git, para todo lo que ocurre en tu repositorio.
- `gh`: la CLI oficial de GitHub, para todo lo que ocurre en la plataforma (repos remotos, PRs, issues, Actions).

A quién va dirigido. A desarrolladores que usan Git a medias desde un editor y quieren dominarlo desde la terminal, y a quien empieza de cero. No se necesita experiencia previa, solo saber abrir una terminal.

Cómo leerlo. Los capítulos 1 a 6 son la base y conviene leerlos en orden. Del 7 en adelante puedes saltar a lo que necesites. El apéndice final es una chuleta para tener a mano.

Convenciones: los comandos van en bloques de código; lo que aparece entre `<` y `>` se sustituye por tu valor (por ejemplo, `<rama>` → `feature/login`). Las líneas que empiezan con `#` son comentarios.

Capítulo 1. Instalación y configuración inicial

Antes del primer commit necesitas tres cosas: Git instalado, tu identidad configurada y una forma segura de autenticarte en GitHub.

1.1 Instalar Git y la CLI de GitHub

```
# Linux (Debian/Ubuntu)
sudo apt update && sudo apt install git gh

# macOS (con Homebrew)
brew install git gh

# Windows (con winget)
winget install Git.Git GitHub.cli

# Comprobar versiones
git --version
gh --version
```

1.2 Configurar tu identidad

Cada commit lleva un nombre y un email. Configúralos una sola vez con `--global`:

```
git config --global user.name "Tu Nombre"
git config --global user.email "tu@email.com"

# Rama por defecto al crear repositorios
git config --global init.defaultBranch main

# Editor para mensajes de commit (nano, vim, code --wait...)
git config --global core.editor "nano"

# Ver toda la configuración y su origen
git config --list --show-origin
```

Usa el mismo email que tienes en GitHub para que los commits se asocien a tu perfil. Sin `--global`, la configuración aplica solo al repositorio actual: útil para separar cuentas de trabajo y personales.

1.3 Autenticar la CLI `gh`

```
gh auth login        # asistente interactivo: GitHub.com → SSH → navegador
gh auth status       # comprobar con qué cuenta estás conectado
gh auth setup-git    # usar gh como gestor de credenciales para HTTPS
```

1.4 Autenticación con SSH

GitHub ya no acepta contraseñas para operaciones Git. Las dos opciones son SSH o un token HTTPS. SSH es la más cómoda:

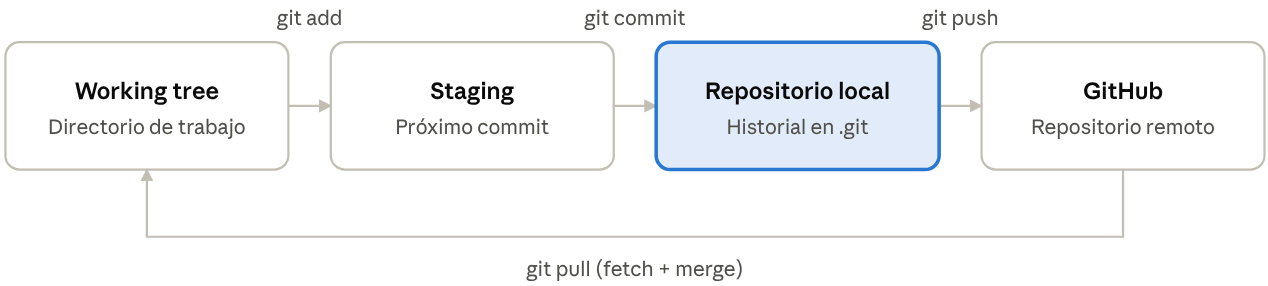
```
# Generar una clave nueva
ssh-keygen -t ed25519 -C "tu@email.com"

# Arrancar el agente y añadir la clave
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519

# Subir la clave pública a GitHub con gh
gh ssh-key add ~/.ssh/id_ed25519.pub --title "Mi portátil"

# Probar la conexión
ssh -T git@github.com
```

Con esto ya puedes crear repos, abrir Pull Requests y gestionar issues sin abrir el navegador.

Capítulo 2. Conceptos básicos

Git guarda instantáneas de tu proyecto, no diferencias sueltas. Para usarlo bien basta entender cinco ideas y por dónde viaja un cambio.

Un cambio recorre cuatro zonas hasta llegar a GitHub



Las cuatro zonas de Git y los comandos que mueven cambios entre ellas

Un archivo editado vive primero en tu **working tree**. Con `git add` pasa al **staging** (también llamado índice), donde eliges qué entra en el próximo commit. `git commit` lo guarda en el **repositorio local** y `git push` lo publica en **GitHub**. `git pull` trae de vuelta los cambios de otros.

Las cinco ideas clave

Concepto	Qué es
Repositorio	La carpeta `.git` con toda la historia del proyecto
Commit	Una instantánea con autor, fecha, mensaje y un identificador SHA único
Rama	Un puntero móvil a un commit; avanza solo cuando haces commits en ella
HEAD	El puntero que indica en qué rama y commit estás ahora
Remoto	Una copia del repositorio en otro sitio, normalmente llamada `origin`

Estados de un archivo

1. **Untracked**: Git no lo conoce todavía.
2. **Modified**: cambió desde el último commit, pero no está preparado.
3. **Staged**: preparado para el próximo commit.
4. **Committed**: guardado en la historia.

`git status` te dice en todo momento en qué estado está cada archivo. Es el comando que más usarás.

Capítulo 3. Crear y clonar repositorios

Hay dos formas de empezar: crear un repositorio nuevo o copiar uno que ya existe.

3.1 Empezar desde cero en local

```
mkdir mi-proyecto && cd mi-proyecto
git init                      # crea la carpeta .git
echo "# Mi proyecto" > README.md
git add README.md
git commit -m "Primer commit"
```

3.2 Publicarlo en GitHub con `gh`

```
# Crea el repo en GitHub, añade el remoto origin y sube la rama actual
gh repo create mi-proyecto --private --source=. --remote=origin --push
```

Cambia `--private` por `--public` si quieres que sea visible para todos.

3.3 Hacerlo a mano (sin `gh`)

Si ya creaste el repositorio vacío en la web de GitHub:

```
git remote add origin git@github.com:usuario/mi-proyecto.git
git branch -M main
git push -u origin main
```

3.4 Clonar un repositorio existente

```
# Con SSH
git clone git@github.com:usuario/repo.git

# Con gh (más corto)
gh repo clone usuario/repo

# En una carpeta con otro nombre
git clone git@github.com:usuario/repo.git otra-carpeta

# Solo el último commit (repos grandes)
git clone --depth 1 git@github.com:usuario/repo.git
```

3.5 Explorar repos desde la terminal

```
gh repo view usuario/repo          # descripción y README
gh repo view --web                 # abre el repo actual en el navegador
gh repo list usuario --limit 20    # repos de un usuario u organización
```

Capítulo 4. El ciclo diario

El 90 % del trabajo con Git es repetir este ciclo: mirar qué cambió, preparar, confirmar y revisar.

4.1 Ver el estado

```
git status          # informe completo
git status -s       # versión corta: M = modificado, A = añadido, ?? = nuevo
```

4.2 Preparar cambios (staging)

```
git add archivo.py          # un archivo
git add src/                # una carpeta
git add .                   # todo lo modificado y nuevo en la carpeta actual
git add -p                  # elegir trozo a trozo qué entra (muy
recomendable)
```

4.3 Ver las diferencias

```
git diff                    # cambios aún no preparados
git diff --staged           # lo que entrará en el próximo commit
git diff main..mi-rama      # diferencias entre dos ramas
```

4.4 Confirmar (commit)

```
git commit -m "Añade validación del formulario de login"
git commit                  # abre el editor para un mensaje largo
git commit -am "Mensaje"    # add + commit de archivos ya rastreados
```

4.5 Revisar la historia

```
git log                             # historia completa
git log --oneline                   # una línea por commit
git log --oneline --graph --all     # árbol de todas las ramas
git log -p archivo.py               # cambios de un archivo, commit a commit
git log --author="Ana" --since="2 weeks ago"
git show <sha>                      # detalle de un commit
git blame archivo.py                # quién cambió cada línea y cuándo
```

4.6 Descartar y mover archivos

```
git restore archivo.py              # descarta cambios no preparados
(¡irreversible!)
git restore --staged archivo.py     # saca el archivo del staging, conserva
cambios
git rm archivo.py                   # borra y registra el borrado
git mv viejo.py nuevo.py            # renombra y registra el cambio
```

Regla práctica: antes de cada commit ejecuta `git status` y `git diff --staged`. Así nunca subes algo por accidente.

Capítulo 5. Ramas

Una rama te deja trabajar en algo nuevo sin tocar `main`. Crear una cuesta milisegundos, así que úsalas para cada tarea.

5.1 Crear, cambiar y listar

```
git branch                      # ramas locales (* = la actual)
git branch -a                   # locales y remotas
git switch -c feature/login     # crear y cambiar a una rama nueva
git switch main                 # volver a main
git switch -                    # volver a la rama anterior
```

`git switch` sustituye al antiguo `git checkout <rama>`, que sigue funcionando pero hace demasiadas cosas a la vez.

5.2 Renombrar y borrar

```
git branch -m nombre-nuevo              # renombrar la rama actual
git branch -d feature/login             # borrar (solo si ya está fusionada)
git branch -D feature/login             # forzar el borrado
git push origin --delete feature/login  # borrar la rama en GitHub
```

5.3 Fusionar con merge

```
git switch main
git merge feature/login                 # trae los commits de la rama a main
git merge --no-ff feature/login         # fuerza un commit de merge visible
```

Si `main` no avanzó, Git hace un *fast-forward*: solo mueve el puntero. Si ambas ramas avanzaron, crea un commit de merge con dos padres.

5.4 Reorganizar con rebase

```
git switch feature/login
git rebase main          # reaplica tus commits encima del main actual
```

Rebase deja una historia lineal y limpia. **Regla de oro:** no hagas rebase de commits que ya compartiste en una rama pública; reescribe la historia de los demás.

5.5 Resolver conflictos

Un conflicto aparece cuando dos ramas cambian las mismas líneas. Git marca el archivo así:

```
<<<<<<< HEAD
color = "azul"
=======
color = "verde"
>>>>>>> feature/login
```

Pasos para resolverlo:

1. Ejecuta `git status` para ver qué archivos están en conflicto.
2. Edita cada archivo, deja la versión correcta y borra las marcas.
3. Marca el archivo como resuelto con `git add archivo`.
4. Termina con `git commit` (en un merge) o `git rebase --continue` (en un rebase).

```
git merge --abort        # cancelar el merge y volver al estado anterior
git rebase --abort       # cancelar el rebase
```

Capítulo 6. Remotos: sincronizar con GitHub

Un remoto es otra copia del repositorio. Casi siempre trabajarás con `origin` (tu copia en GitHub) y, si contribuyes a proyectos ajenos, con `upstream` (el original).

6.1 Gestionar remotos

```
git remote -v                                    # listar remotos y sus URLs
git remote add upstream git@github.com:org/repo.git
git remote set-url origin git@github.com:usuario/nuevo.git
git remote rename origin github
git remote remove upstream
```

6.2 Traer cambios: fetch vs pull

```
git fetch origin           # descarga cambios sin tocar tu trabajo
git fetch --all --prune    # todos los remotos y limpia ramas borradas
git pull                   # fetch + merge en la rama actual
git pull --rebase          # fetch + rebase (historia más limpia)
```

`fetch` es siempre seguro: solo actualiza las referencias remotas (`origin/main`). `pull` además integra esos cambios en tu rama. Para que `pull` use rebase por defecto:

```
git config --global pull.rebase true
```

6.3 Enviar cambios: push

```
git push                                # rama actual a su upstream
git push -u origin feature/login        # primera vez: crea la rama remota y
la enlaza
git push origin --tags                  # subir etiquetas
git push --force-with-lease             # forzar de forma segura tras un
rebase
```

Usa siempre `--force-with-lease` en lugar de `--force` : se niega a sobrescribir si alguien subió commits que tú no tienes.

6.4 Ramas de seguimiento (upstream)

```
git branch -vv                                  # ver qué remota sigue cada
rama
git branch --set-upstream-to=origin/main main   # enlazar manualmente
git status                                      # indica si vas por delante o
por detrás
```

Si `git status` dice *Your branch is behind 'origin/main' by 3 commits*, toca hacer `git pull`. Si dice *ahead*, toca `git push`.

Capítulo 7. Flujo de trabajo con GitHub: forks y Pull Requests

Un Pull Request (PR) propone fusionar una rama en otra y abre un espacio para revisar el código antes de integrarlo. Con `gh` todo el flujo se hace desde la terminal.

7.1 Flujo completo en un repositorio propio

```
git switch main && git pull                 # 1. partir de main actualizado
git switch -c feature/exportar-pdf          # 2. rama para la tarea
# ... editar, git add, git commit ...
git push -u origin feature/exportar-pdf     # 3. subir la rama
gh pr create --fill                         # 4. abrir el PR (título y cuerpo
desde los commits)
```

Opciones útiles de `gh pr create`:

```
gh pr create --title "Exportar a PDF" --body "Closes #42" \
  --base main --reviewer ana,luis --label mejora --draft
```

Escribir `Closes #42` en el cuerpo cierra el issue 42 automáticamente al fusionar.

7.2 Contribuir a un proyecto ajeno (fork)

```
gh repo fork org/proyecto --clone    # crea tu fork, lo clona y configura
upstream
cd proyecto
git switch -c fix/typo-readme
# ... cambios y commit ...
git push -u origin fix/typo-readme
gh pr create --repo org/proyecto --fill
```

Mantén tu fork al día con el original:

```
git fetch upstream
git switch main
git merge upstream/main     # o: gh repo sync
git push origin main
```

7.3 Revisar Pull Requests

```
gh pr list                            # PRs abiertos
gh pr status                          # tus PRs y los que te piden revisar
gh pr view 57                         # detalle del PR 57
gh pr diff 57                         # ver los cambios
gh pr checkout 57                     # bajar la rama del PR para probarla
gh pr review 57 --approve
gh pr review 57 --request-changes --body "Falta un test para el caso vacío"
gh pr review 57 --comment --body "Buena idea, un detalle menor"
```

7.4 Fusionar y limpiar

```
gh pr checks 57                       # estado de los tests de CI
gh pr merge 57 --squash --delete-branch
git switch main && git pull           # actualizar tu main local
git branch -d feature/exportar-pdf  # borrar la rama local
```

Estrategia	Qué hace	Cuándo usarla
`--merge`	Crea un commit de merge con todos los commits	Conservar la historia completa
`--squash`	Une todos los commits en uno	PRs con muchos commits de prueba
`--rebase`	Reaplica los commits sin commit de merge	Historia lineal con commits limpios

Capítulo 8. Issues, tags y releases

Los issues organizan el trabajo pendiente; los tags y releases marcan versiones publicadas. Ambos se gestionan sin salir de la terminal.

8.1 Issues

```
gh issue list                                    # issues abiertos
gh issue list --assignee @me --label bug         # filtrar
gh issue create --title "Error al exportar" --body "Pasos para reproducir..."
--label bug
gh issue view 42
gh issue comment 42 --body "Lo reviso hoy"
gh issue edit 42 --add-assignee @me
gh issue close 42 --comment "Resuelto en #57"
gh issue develop 42 --checkout                   # crea una rama enlazada al
issue
```

8.2 Tags (etiquetas de versión)

```
git tag                                  # listar
git tag -a v1.2.0 -m "Versión 1.2.0"   # tag anotado (recomendado)
git tag v1.2.0-beta                      # tag ligero
git show v1.2.0
git push origin v1.2.0                   # subir un tag
git push origin --tags                   # subir todos
git tag -d v1.2.0-beta                   # borrar local
git push origin --delete v1.2.0-beta     # borrar en GitHub
```

Sigue el versionado semántico `MAYOR.MENOR.PARCHE`: sube la mayor si rompes compatibilidad, la menor si añades funciones y el parche si solo corriges errores.

8.3 Releases

Una release es un tag con notas y archivos descargables publicados en GitHub.

```
gh release create v1.2.0 --generate-notes            # notas automáticas
desde los PRs
gh release create v1.2.0 dist/app.zip --title "v1.2.0" --notes-file
CHANGELOG.md
gh release list
gh release view v1.2.0
gh release download v1.2.0 --pattern "*.zip"
gh release delete v1.2.0-beta
```

Capítulo 9. Deshacer errores

Casi todo en Git tiene arreglo. La clave es saber si el error ya está publicado: en local puedes reescribir la historia; en GitHub, mejor crear un commit que lo corrija.

Situación	Comando
Mensaje del último commit mal escrito	`git commit --amend -m "Nuevo mensaje"`
Olvidé un archivo en el último commit	`git add archivo && git commit --amend --no-edit`
Deshacer el último commit, conservando cambios	`git reset --soft HEAD~1`
Deshacer el último commit y sacar del staging	`git reset HEAD~1`
Borrar el último commit y sus cambios	`git reset --hard HEAD~1`
Revertir un commit ya publicado	`git revert <sha>`
Descartar cambios de un archivo	`git restore archivo`
Recuperar un commit "perdido"	`git reflog` y luego `git reset --hard <sha>`

9.1 Los tres modos de reset

- `--soft`: mueve la rama; los cambios quedan en staging.
- `--mixed` (por defecto): mueve la rama; los cambios quedan en el working tree.
- `--hard`: mueve la rama y **borra** los cambios. Úsalo con cuidado.

9.2 Revert: deshacer sin reescribir

```
git revert a1b2c3d            # crea un commit que invierte a1b2c3d
git revert -m 1 <sha-merge>   # revertir un commit de merge
```

Es la opción segura en ramas compartidas porque no altera la historia existente.

9.3 Reflog: la red de seguridad

`git reflog` registra cada movimiento de HEAD durante unos 90 días. Si borraste una rama o hiciste un `reset --hard` por error, el commit sigue ahí:

```
git reflog                    # buscar el estado anterior, p. ej. HEAD@{3}
git reset --hard HEAD@{3}     # volver a ese punto
git switch -c rescate HEAD@{3} # o crear una rama desde ahí
```

9.4 Stash: guardar trabajo a medias

```
git stash push -m "login a medias"   # guardar y dejar el working tree limpio
git stash -u                         # incluir archivos nuevos
git stash list
git stash pop                        # recuperar el último y borrarlo de la
lista
git stash apply stash@{1}            # recuperar uno concreto sin borrarlo
git stash drop stash@{1}
```

Útil cuando tienes que cambiar de rama urgentemente y no quieres hacer un commit a medias.

Capítulo 10. Técnicas avanzadas

Estas herramientas no se usan a diario, pero resuelven en minutos problemas que de otro modo llevan horas.

10.1 Cherry-pick: copiar un commit concreto

```
git switch main
git cherry-pick a1b2c3d            # aplica ese commit sobre la rama actual
git cherry-pick a1b2c3d..f4e5d6c   # rango: no incluye a1b2c3d
```

Caso típico: un arreglo hecho en una rama de desarrollo que necesitas ya en producción.

10.2 Rebase interactivo: limpiar la historia

```
git rebase -i HEAD~4
```

Se abre el editor con los últimos cuatro commits. Cambia la palabra inicial de cada línea:

- `pick`: mantener el commit.
- `reword`: cambiar el mensaje.
- `squash` / `fixup`: unirlo al anterior (con o sin su mensaje).
- `drop`: eliminarlo.
- Reordenar líneas reordena los commits.

Ideal para dejar un PR limpio antes de pedir revisión.

10.3 Bisect: encontrar el commit que rompió algo

```
git bisect start
git bisect bad                  # la versión actual falla
git bisect good v1.1.0          # esta versión funcionaba
# Git va saltando entre commits; en cada uno pruebas y respondes:
git bisect good                 # o git bisect bad
git bisect reset                # al terminar, vuelve a donde estabas

# Automático con un script que devuelve 0 si todo va bien
git bisect run npm test
```

Con 1.000 commits de por medio, bisect encuentra el culpable en unos 10 pasos.

10.4 .gitignore

El archivo `.gitignore` en la raíz lista lo que Git nunca debe rastrear:

```
# Dependencias y compilados
node_modules/
__pycache__/
dist/

# Secretos y configuración local
.env
*.key

# Sistema y editores
.DS_Store
.vscode/
```

```
git rm --cached .env          # dejar de rastrear un archivo ya subido
git check-ignore -v archivo   # saber qué regla lo ignora
```

Si subiste un secreto por error, bórralo del repositorio **y cámbialo**: sigue estando en la historia.

10.5 Submódulos

```
git submodule add git@github.com:org/libreria.git libs/libreria
git clone --recurse-submodules git@github.com:usuario/repo.git
git submodule update --init --recursive
git submodule update --remote          # actualizar a lo último de cada
submódulo
```

10.6 Alias: atajos propios

```
git config --global alias.st status
git config --global alias.co switch
git config --global alias.lg "log --oneline --graph --all --decorate"
git config --global alias.deshacer "reset --soft HEAD~1"

git lg          # ahora funciona como atajo
```

`gh` también admite alias: `gh alias set prs 'pr list --author @me'`.

10.7 Worktrees: dos ramas a la vez

```
git worktree add ../repo-hotfix hotfix/urgente   # otra carpeta con otra rama
git worktree list
git worktree remove ../repo-hotfix
```

Permite atender un arreglo urgente sin hacer stash ni interrumpir lo que tienes a medias.

Capítulo 11. GitHub Actions desde la terminal

GitHub Actions ejecuta tareas automáticas (tests, compilación, despliegues) cuando ocurre algo en el repositorio, como un push o un PR. Los flujos se definen en archivos YAML dentro de `.github/workflows/`.

11.1 Un workflow mínimo

Crea `.github/workflows/ci.yml`:

```
name: CI
on:
  push:
    branches: [main]
  pull_request:

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.12"
      - run: pip install -r requirements.txt
      - run: pytest
```

```
git add .github/workflows/ci.yml
git commit -m "Añade CI con pytest"
git push
```

11.2 Vigilar ejecuciones con `gh`

```
gh run list                       # últimas ejecuciones
gh run watch                      # seguir en directo la ejecución en curso
gh run view 1234567 --log-failed  # ver solo los logs de los pasos que
fallaron
gh run rerun 1234567 --failed     # relanzar solo los jobs fallidos
gh run download 1234567           # descargar artefactos generados
```

11.3 Lanzar y gestionar workflows

```
gh workflow list
gh workflow run deploy.yml --ref main -f entorno=produccion   # requiere on:
workflow_dispatch
gh workflow disable ci.yml
gh workflow enable ci.yml
```

11.4 Secretos y variables

Nunca escribas contraseñas o tokens en el YAML. Guárdalos como secretos y úsalos con `${{ secrets.NOMBRE }}`:

```
gh secret set API_TOKEN                      # pide el valor de forma oculta
gh secret set API_TOKEN < token.txt          # desde un archivo
gh secret list
gh variable set ENTORNO --body "staging"     # variables no secretas
```

Capítulo 12. Buenas prácticas y convenciones

Un repositorio ordenado ahorra horas de revisión y facilita encontrar errores meses después.

12.1 Mensajes de commit

- Primera línea de menos de 72 caracteres, en imperativo: *Añade*, *Corrige*, *Elimina*.
- Una línea en blanco y, si hace falta, un cuerpo que explique **por qué**, no qué.
- Un commit = un cambio lógico. Mejor varios pequeños que uno enorme.

Muchos equipos usan **Conventional Commits**:

```
feat: añade exportación a PDF
fix(login): evita doble envío del formulario
docs: actualiza instrucciones de instalación
refactor: separa la validación en su propio módulo
chore: actualiza dependencias
```

12.2 Nombres de ramas

Prefijo	Uso	Ejemplo
`feature/`	Funcionalidad nueva	`feature/exportar-pdf`
`fix/`	Corrección de un error	`fix/login-doble-envio`
`hotfix/`	Arreglo urgente en producción	`hotfix/caida-pagos`
`docs/`	Solo documentación	`docs/readme-instalacion`
`chore/`	Mantenimiento	`chore/actualizar-deps`

12.3 Flujos de trabajo habituales

- **GitHub Flow**: `main` siempre desplegable; cada cambio en una rama corta que entra por PR. Sencillo y recomendado para la mayoría de equipos.
- **Git Flow**: ramas `main`, `develop`, `release/*` y `hotfix/*`. Útil con versiones planificadas y varias versiones en mantenimiento.
- **Trunk-based**: ramas que viven horas, integración continua a `main` y uso de *feature flags*.

12.4 Proteger la rama principal

En GitHub, configura reglas de protección para `main`: exigir PR, al menos una aprobación y que pasen los checks de CI. Así nadie sube directamente por error.

12.5 Lista de comprobación antes de abrir un PR

- [ ] Rama actualizada con `main` (`git pull --rebase origin main`).
- [ ] Tests en verde en local.
- [ ] `git diff main` revisado: sin archivos de depuración ni secretos.
- [ ] Commits limpios (rebase interactivo si hace falta).
- [ ] Descripción del PR con contexto y el issue que cierra.

Apéndice. Chuleta de comandos

Los comandos esenciales del libro en una sola tabla, agrupados por tarea.

Tarea	Comando
Configurar identidad	`git config --global user.name "Nombre"`
Autenticar gh	`gh auth login`
Crear repo local	`git init`
Crear y publicar en GitHub	`gh repo create nombre --private --source=. --push`
Clonar	`gh repo clone usuario/repo`
Ver estado	`git status -s`
Preparar cambios	`git add -p`
Ver diferencias preparadas	`git diff --staged`
Confirmar	`git commit -m "Mensaje"`
Historia en árbol	`git log --oneline --graph --all`
Nueva rama	`git switch -c feature/x`
Cambiar de rama	`git switch main`
Fusionar	`git merge feature/x`
Reorganizar	`git rebase main`
Traer cambios	`git pull --rebase`
Subir rama nueva	`git push -u origin feature/x`
Forzar de forma segura	`git push --force-with-lease`
Abrir PR	`gh pr create --fill`
Probar un PR	`gh pr checkout 57`
Fusionar PR	`gh pr merge 57 --squash --delete-branch`
Crear issue	`gh issue create --title "..."`
Crear release	`gh release create v1.0.0 --generate-notes`
Corregir último commit	`git commit --amend`
Deshacer último commit	`git reset --soft HEAD~1`

Tarea	Comando
Revertir commit publicado	`git revert <sha>`
Guardar trabajo a medias	`git stash push -m "..."`
Recuperar lo perdido	`git reflog`
Copiar un commit	`git cherry-pick <sha>`
Buscar el commit culpable	`git bisect start`
Ver ejecuciones de CI	`gh run watch`

Domina Git y GitHub con la terminal

Un recorrido práctico y directo por los comandos que se usan cada día: desde configurar tu identidad hasta revisar Pull Requests, publicar releases y automatizar con GitHub Actions.

En este libro aprenderás a:

- Configurar Git, SSH y la CLI `gh`
- Dominar el ciclo diario: `status`, `add`, `commit`, `log`
- Trabajar con ramas, merge, rebase y conflictos
- Colaborar con forks, Pull Requests e issues
- Deshacer errores con `reset`, `revert` y `reflog`
- Usar técnicas avanzadas: cherry-pick, bisect, worktrees
- Automatizar con GitHub Actions

Incluye una chuleta final con los 30 comandos esenciales.

Marlon Falcón Hernández

Git · GitHub · CLI