Git de Básico a Intermedio

Jonathan Hechenleitner

22 de septiembre – 27 de septiembre

Introducción a Git

Git es un **sistema de control de versiones distribuido (DVCS)**.
Cada desarrollador tiene una copia completa del repositorio, lo que permite trabajar offline y sincronizar después.

- **Comandos**

```
git init        # Inicializa un repositorio vacío
git --version   # Verifica la versión instalada
```

- **Caso de uso**
  - Un equipo de desarrollo trabaja en paralelo en distintas funcionalidades y necesita un historial confiable.
- **Buenas prácticas**
  - Usar Git desde el inicio del proyecto.
  - Mantener un repositorio central (GitHub/GitLab/Bitbucket).
- **Errores comunes**
  - Usar Git solo como **backup** sin aprovechar ramas ni historial.

Instalación y Configuración Inicial

Antes de empezar a trabajar, debes configurar tu identidad en Git y preparar el entorno.

- **Comandos**

```
git config --global user.name "Tu Nombre"
git config --global user.email "tuemail@dominio.com"
git config --list
```

- **Caso de uso**
  - Cada commit en el historial tendrá asociado un autor y correo válido.
- **Buenas practices**
  - Usar el correo asociado a tu cuenta de GitHub/GitLab.
- **Errores comunes**
  - No configurar usuario → commits sin autor.Configurar mal los line endings en Windows.

Flujo de Trabajo Básico

El ciclo de cambios en Git tiene 3 etapas:
Working Directory → Staging Area → Repository

- **Comandos**

```
git status
git add archivo.txt
git commit -m "feat: agrega archivo inicial"
git log --oneline
```

- **Caso de uso**
  - Registrar cambios en pasos pequeños y entendibles.
- **Buenas prácticas**
  - Commits atómicos.
  - Mensajes claros.
- **Errores comunes**
  - Hacer un commit enorme con cambios mezclados.

Repositorios: Locales y Remotos

Un repositorio puede estar en tu máquina (**local**) o en un servidor (**remoto**).

- **Comandos**

```
git clone https://github.com/usuario/proyecto.git
git remote add origin https://github.com/usuario/proyecto.git
git push -u origin main
git pull origin main
```

- **Caso de uso**
  - Conectar un repo local a GitHub para colaboración en equipo.
- **Buenas prácticas**
  - Usar origin como nombre del remoto principal.
  - Configurar upstream si trabajas con forks.
- **Errores comunes**
  - No hacer git pull antes de trabajar.

Commits

Un commit es una “foto” del estado del proyecto en un momento específico.

- **Comandos**

```
git commit -m "fix: corrige error en login"
git add -p archivo.txt   # Commits parciales
```

- **Caso de uso**
  - Corregir un bug y dejar constancia en el historial.
- **Buenas prácticas**
  - Mensajes con convención: feat, fix, docs.
  - Un commit = un propósito.
- **Errores comunes**
  - Usar mensajes genéricos: “cambios varios”.

Inspección de Cambios

Permite revisar el historial y diferencias en el código.

- **Comandos**

```
git log --oneline
git diff
git diff --staged
git blame archivo.txt
```

- **Caso de uso**
  - Saber quién introdujo un bug en una línea de código.
- **Buenas prácticas**
  - Usar git log --stat para tener contexto de cambios.
- **Errores comunes**
  - No revisar antes de integrar ramas.

Branches (Ramas)

Las ramas permiten trabajar en paralelo sin afectar la principal.

- **Comandos**

```
git branch
git switch -c feature/login
git branch -d feature/login
```

- **Caso de uso**
  - Crear una rama feature/login para desarrollar login sin romper main.
- **Buenas prácticas**
  - Nombres claros: feature/, bugfix/, hotfix/.
- **Errores comunes**
  - Trabajar todo en main.

Merge y Conflictos

merge integra cambios de una rama en otra. Puede generar conflictos.

- **Comandos**

```
git switch main
git merge feature/login
# Resolver conflictos
git add archivo_conflictivo
git commit
```

- **Caso de uso**
  - Integrar un **feature** a la rama principal.
- **Buenas prácticas**
  - Hacer **commits** pequeños para reducir conflictos.
- **Errores comunes**
  - Resolver conflictos sin revisar qué se elimina.

Rebase y Cherry-pick

- **Rebase**: reaplica commits sobre otra rama.
- **Cherry-pick**: trae commits puntuales.
- **Comandos**

```
git switch feature/login
git rebase main
git cherry-pick abc123
```

- **Caso de uso**
  - Mantener historial lineal o traer un bugfix puntual.
- **Buenas prácticas**
  - Usar rebase en ramas locales, no en ramas compartidas.
- **Errores comunes**
  - Usar push --force en ramas compartidas.

Deshacer Cambios

En Git hay **varias formas de deshacer cambios**, y cada una aplica a un contexto distinto:

- **Descartar cambios locales** → antes de hacer commit.
- **Revertir un commit ya hecho** → mantener historial limpio.
- **Resetear commits** → retroceder en el tiempo, opcionalmente borrando cambios.
- **Casos de uso**
  - **Olvidaste sacar un archivo sensible** antes del commit → git reset HEAD archivo.txt.
  - **Necesitas deshacer un commit que rompió producción** → git revert <hash>.
  - **Quieres reescribir commits en tu rama local antes de publicar** → git reset --soft o git commit --amend

- **Buenas prácticas**
  - Usa **git revert** en ramas compartidas.
  - Usa **reset** solo en ramas locales o commits no compartidos.
  - Haz **commits pequeños** para que revertir sea más fácil.
  - Verifica con git log o git diff antes de cualquier acción destructiva.

- **Errores comunes**
  - Usar git reset --hard en una rama compartida → se pierden commits para el equipo.
  - Confiar en git revert sin probar → a veces genera conflictos.
  - Olvidar git reflog → muchos piensan que lo perdido no se puede recuperar.

Deshacer cambios locales

1) Descartar cambios en archivos del *working directory*

- Util cuando editaste algo por error y no quieres ni siquiera staged.

```
# Descarta cambios locales (archivo vuelve al último commit)
git restore archivo.txt
# Para todos los archivos modificados
git restore .
```

2) Sacar archivos del *staging area*

- Útil cuando agregaste muchos archivos al git add . pero uno no debía incluirse.

```
# Quita archivo del staging, pero conserva cambios en WD
git restore --staged archivo.txt
```

Deshacer cambios - revertir

3) Revertir commits (forma segura, conserva historial)

- Git crea un **nuevo commit inverso**. Es la forma más recomendada en proyectos colaborativos, porque no rompe el historial compartido.

Deshacer cambios - resetear

4) Resetear commits (peligroso si ya hiciste push)

```
# Retrocede un commit pero mantiene cambios en staging
git reset --soft HEAD~1
# Retrocede y deja cambios en working directory (unstaged)
git reset --mixed HEAD~1   # por defecto
# Retrocede y BORRA los cambios (⚠️ peligroso)
git reset --hard HEAD~1
```

- Úsalo solo en ramas locales donde no compartiste tu trabajo aún.

Trabajo Colaborativo

Un **repositorio remoto** (GitHub, GitLab, Bitbucket, Azure DevOps, etc.) actúa como punto central de sincronización.
Las operaciones clave son:

- **fetch** → trae cambios del remoto, pero no los mezcla.
- **pull** → trae y mezcla (es fetch + merge).
- **push** → envía tus commits al remoto.
  Comandos principales
  - **1) Traer cambios del remote**

- 👉 Útil para revisar qué hay en remoto **antes de integrarlo**.
-

```
# Descarga referencias del remoto (no integra aún)
git fetch origin
# Ver diferencias entre tu rama y el remoto
git log origin/main..main
```

- **2) Sincronizar tu rama con el remoto**
- # Opción segura (solo integra si no hay conflictos)
- git pull --ff-only

- # Opción habitual (merge automático si es necesario)
- git pull origin main

- # Mantener historial lineal (rebase sobre el remoto)
- git pull --rebase origin main
- 👉 Se recomienda --ff-only o --rebase para evitar commits de merge innecesarios.
-

- **3) Subir tus cambios**
- # Subir commits a la rama remota
- git push origin feature/login

- # Configurar upstream la primera vez
- git push -u origin feature/login
- 👉 Con -u tu rama local queda enlazada a la remota, simplificando futuros git push y git pull.

Trabajo Colaborativo - fetch

1) Traer cambios del remote

- Útil para revisar qué hay en remoto **antes de integrarlo**.

```
# Descarga referencias del remoto (no integra aún)
git fetch origin
# Ver diferencias entre tu rama y el remoto
git log origin/main..main
```

Trabajo Colaborativo - pull

2) Sincronizar tu rama con el remoto

- Se recomienda --ff-only o --rebase para evitar commits de merge innecesarios.

```
# Opción segura (solo integra si no hay conflictos)
git pull --ff-only
# Opción habitual (merge automático si es necesario)
git pull origin main
# Mantener historial lineal (rebase sobre el remoto)
git pull --rebase origin main
```

Trabajo Colaborativo - push

- **3) Subir tus cambios**
  - Con -u tu rama local queda enlazada a la remota, simplificando futuros git push y git pull.

```
# Subir commits a la rama remota
git push origin feature/login
# Configurar upstream la primera vez
git push -u origin feature/login
```

Stash

Permite guardar cambios sin hacer commit.

- **Comandos**

```
git stash save "avance en login"
git stash list
git stash pop
```

- **Caso de uso**
  - Interrumpir tu trabajo para atender un bug urgente.
- **Buenas prácticas**
  - Nombrar los stashes.
- **Errores comunes**
  - Olvidar aplicar stashes y perderlos.

Gitignore

Define qué archivos no deben versionarse.

- **Ejemplo .gitignore**

```
node_modules/
.env
*.log
```

- **Caso de uso**
  - Evitar subir dependencias y secretos.
- **Buenas prácticas**
  - Compartir un .gitignore en el repo.
- **Errores comunes**
  - No usar .gitignore y subir basura.

JONATHAN HECHENLEITNER