CURSO DE GIT PARA DESARROLLADORES

Contenido creado por
Francisco Javier Salvado de la Llave
para la ***Bosonit Academy***

MÓDULO 1

INTRODUCCIÓN, INSTALACIÓN Y PRIMEROS PASOS

¿Qué es Git?

Git es un **sistema de control de versiones distribuido** que permite rastrear cambios en el código fuente durante el desarrollo de software. Creado por Linus Torvalds en 2005, es la herramienta estándar en la industria para colaborar en proyectos de desarrollo.

Características clave:

- ✓ **Distribuido:** Cada desarrollador tiene una copia completa del repositorio.
- ✓ **Rápido:** Operaciones locales sin necesidad de conexión a internet.
- ✓ **No lineal**: Soporta múltiples líneas de desarrollo (ramas).
- ✓ **Seguro**: Usa SHA-1 para identificar commits de forma única.

Instalación

- **Windows:**
  1. *Descargar desde https://git-scm.com/*
  2. *Ejecutar el instalador y seguir el asistente*
  3. *Seleccionar "Use Git from the Windows Command Prompt"*

- **macOS:**

```
# Usando Homebrew
brew install git
```

```
# O desde Xcode Command Line Tools
xcode-select --install
```

- **Linux (Ubuntu/Debian):**

```
sudo apt update
sudo apt install git
```

Configuración Inicial

```
# Configurar nombre y email (se usa en commits)
git config --global user.name "Tu Nombre"
git config --global user.email "tu-email@ejemplo.com"

# Configurar editor predeterminado
Git config --global core.editor "code --wait"  # Para VS Code

# Ver configuración
git config --list
```

Comandos básicos para trabajar en local

- **git init**: Inicializar repositorio.
- **git add**: Añadir archivos al staging.
- **git commit**: Crear commit.
- **git status**: Ver estado del repositorio.
- **git checkout**: Cambiar de rama.
- **git log**: Ver historial de commits (la opción más usada es git log --oneline --graph --all).
- **git branch**: ver ramas.
- **git tag**: gestión de etiquetas.

Primeros Pasos

A continuación, veremos cómo inicializar un nuevo repositorio Git en local, aunque típicamente en proyecto real lo normal será que ya exista un repositorio inicializado y simplemente tengamos que descargarlo (clonarlo) en nuestra máquina para empezar a trabajar con él. En este caso veremos más adelante cómo actuar.

1. Crear un repositorio nuevo

```
mkdir mi-proyecto
cd mi-proyecto
git init
```

2. Crear archivo y hacer primer commit

```
echo "# Mi Proyecto" > README.md
git add README.md
git commit -m "Inicial: Crear README"
```

3. Ver estado

```
git status
git log --oneline
```

RAMAS EN GIT

Las ramas permiten trabajar en diferentes versiones de un proyecto simultáneamente sin afectar el trabajo principal, lo que las hace ideales para colaborar en equipo, experimentar con nuevas funcionalidades o corregir errores sin riesgos.

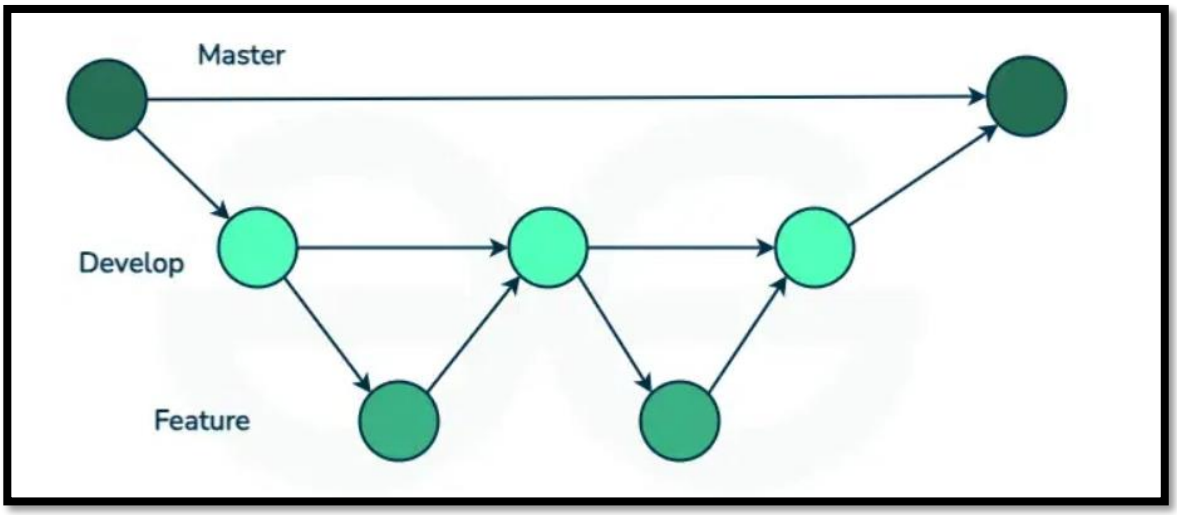
Cada rama representa una línea independiente de desarrollo, pero todas comparten el mismo historial de commits hasta el punto en que se separan. Esto permite a los desarrolladores trabajar en tareas específicas (como añadir una funcionalidad, solucionar un bug o probar algo experimental) de forma aislada, para luego combinar los cambios en la rama principal cuando estén listos.

Cuando inicializas un repositorio con git init o clonas uno con git clone, Git automáticamente crea una rama por defecto llamada main (o master en versiones antiguas de Git, aunque main es el estándar moderno desde 2020). Esta rama principal actúa como la base oficial del proyecto, donde normalmente se encuentra la versión estable y lista para producción.

A partir de la rama principal (**main**), los equipos suelen adoptar un flujo de trabajo basado en ramas para estructurar el desarrollo. Uno de los enfoques más comunes incluye:

- Rama de desarrollo (**develop**): Una rama dedicada a integrar cambios que aún no están listos para producción. Sirve como un "espacio de pruebas" donde se combinan las nuevas funcionalidades o correcciones antes de pasar a main.

- Ramas de funcionalidades (**feature**/*): Cada nueva funcionalidad o tarea se desarrolla en una rama específica (por ejemplo, feature/login-system, feature/payment-gateway). Estas ramas se crean desde develop (o directamente desde main en flujos más simples) y se fusionan de vuelta cuando la funcionalidad está completa.



MÓDULO 2

MANEJO DE GIT

Anteriormente hemos visto cómo trabajar con Git en local, pero en este apartado vamos a partir de la base de que todos tenemos cuenta en alguna plataforma para alojar repositorios en la nube como por ejemplo GitHub. A fin de cuentas, Git está pensado para entre otras cosas poder trabajar en equipo y de forma remota.

Comandos básicos para trabajar en remoto

- **git clone**: copia repositorio remoto en tu máquina local.
- **git push**: sube commits locales a repositorio remoto.
- **git pull**: descarga y fusiona cambios del repositorio remoto a la rama local.
- **git feth**: se trae los cambios realizados en el repositorio remoto, pero sin fusionar con ramas locales.

Sincronizar proyecto en la nube

```
# Crear repositorio remoto (en GitHub)
git remote add origin https://github.com/usuario/mi-proyecto.git

# Subir tus cambios al repositorio remote (opción –u solo la primera vez)
git push -u origin main
```

Opción 2: Clonar repositorio remoto existente

```
# Caso de uso: Unirse a proyecto existente, colaboración
git clone https://github.com/usuario/proyecto-existente.git
cd proyecto-existente

# Ya tienes historial completo y conexión remota configurada
git branch  # Ver rama actual (normalmente main)
```

Flujo de Trabajo Básico (desde clonado)

1. **Crear y cambiar a nueva rama:**

```
git checkout -b feature/login
```

Tener en cuenta que el comando anterior crea una nueva rama de nombre “feature/login” con un contenido idéntico al de la rama actual desde dónde lanzamos el comando. Esto sirve para crear una nueva rama, pero si simplemente queremos cambiar a otra rama ya creada previamente, lo haremos simplemente con git checkout, sin añadir la opción –b.

2. **Hacer cambios y commitear** ***(****a continuación, se describe un simple ejemplo. Tu puedes realizar un cambio cualquiera, como introducir un “Hola mundo” en tu archivo Python o lo que sea que tengas entre manos****):***

```
# Crear archivos de login
touch src/login.js src/login.css
echo "Función de login" > src/login.js

# Añadir al staging
git add .
git commit -m "feat: implementar sistema de login"
```

3. **Subir cambios al remoto:**

```
git push origin feature/login
```

Actualización de ramas

Una de las operaciones más comunes a la hora de trabajar con git es actualizar ramas. En este sentido, main suele contener actualizaciones como correcciones de bugs, nuevas dependencias o features completadas por otros desarrolladores. Actualizar tu rama evita que trabajes sobre una base obsoleta, reduciendo sorpresas al final.

```
# Nos situamos sobre main
git checkout main

# Descargamos a local lo último de la rama main en remoto
git pull origin main

# Nos situamos sobre la rama que queremos actualizar
git checkout feature/login
```

```
# Introducimos en nuestra rama feature/login los ultimos cambios que haya en main
git merge main
```

Resolución de conflictos

Un conflicto ocurre cuando dos personas (o tú mismo en distintas ramas) modifican la misma parte de un archivo y luego intentas fusionar esos cambios. Cuando esto sucede, Git no sabe cuál versión debe conservar, así que te pide que elijas manualmente qué parte del código mantener.

Ejemplo:

Supongamos que tienes un archivo llamado **index.html** con el siguiente contenido:

```
<h1>Bienvenido</h1>
```

Posteriormente, en la rama **main**, cambias el texto por:

```
<h1>Bienvenido a mi web</h1>
```

En una rama llamada **feature-login**, cambias lo mismo por:

```
<h1>Página de inicio de sesión</h1>
```

Cuando intentas fusionar la rama **feature-login** en **main**, Git te avisa que hay un conflicto porque ambas ramas modificaron la misma línea.

El mensaje típico que aparece en consola es algo parecido a:

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

¿Cómo se ve el conflicto?

Si abres el archivo **index.html**, verás algo así:

```
<<<<<<< HEAD
<h1>Bienvenido a mi web</h1>
=======
<h1>Página de inicio de sesión</h1>
>>>>>>> feature-login
```

Explicación de las marcas:

- Lo que está entre `<<<<<<< HEAD` y `=======` corresponde al contenido de tu rama actual (en este caso, **main**).
- Lo que está entre `=======` y `>>>>>>> feature-login` es el contenido que viene de la otra rama (**feature-login**).

Cómo resolver el conflicto

Tú decides qué versión conservar o puedes combinar ambas.

Por ejemplo, podrías dejarlo así:

```
<h1>Bienvenido a mi web - Inicio de sesión</h1>
```

Una vez que hayas modificado el archivo y estés conforme con el resultado:

- ✓ Guarda el archivo.
- ✓ Marca el conflicto como resuelto con el comando:

```
git add index.html
```

- ✓ Cierra la fusión con:

```
git commit -m "Resuelve conflicto en index.html"
```

Después de esto, Git considerará que el conflicto está solucionado.

Resumen de pasos

1. Intentas fusionar ramas con **git merge**
2. Git detecta un **conflicto**
3. Abres el archivo y **eliges qué versión mantener**. Este paso suele ser cómodo hacerlo mediante la interfaz gráfica del plugin de Git que muchas herramientas como VS Code ofrecen.
4. Guardas los cambios y los marcas como resueltos con **git add**
5. Finalizas el proceso con **git commit**

Listar y eliminar Ramas

```
# Listar ramas
git branch -a

# Eliminar rama local
git branch -d rama-antigua

# Eliminar rama remota
git push origin --delete rama-antigua
```

Gestión de cambios:

```
# Ver diferencias
git diff              # Cambios no stageados
git diff --staged       # Cambios stageados
git diff HEAD~1         # Diferencias con commit anterior
```

```
# Deshacer cambios
git checkout -- archivo.txt  # Descartar cambios no commiteados
git restore archivo.txt      # Git 2.23+
git restore --staged archivo.txt  # Des-staging
```

```
# Commits avanzados
git commit --amend          # Modificar último commit
git reset HEAD~1            # Deshacer commit (mantener cambios)
git reset --hard HEAD~1     # Deshacer commit y cambios
```

Integración con IDEs y otras herramientas

Herramientas como VS Code, IntelliJ IDEA y muchas otras, ofrecen plugins nativos para Git que facilitan el workflow. En nuestra opinión, algunas operaciones como la resolución de conflictos son significativamente más cómoda desde un IDE gracias a interfaces visuales que muestran diferencias lado a lado y herramientas de merge automáticas. Sin embargo, para operaciones avanzadas o debugging, la consola sigue siendo insustituible por su precisión y control total.

MÓDULO 3
BUENAS PRÁCTICAS Y EJERCICIO FINAL

Buenas Prácticas

- **Uso de etiquetas**

  El comando git tag en Git permite crear, listar y gestionar etiquetas (tags), que son referencias estáticas a commits específicos en el historial del repositorio. Estas etiquetas actúan como marcadores permanentes, ideales para señalar puntos importantes como versiones de lanzamiento, hitos del proyecto o correcciones críticas. A diferencia de las ramas, que se mueven con nuevos commits, los tags son inmutables y proporcionan una forma clara de identificar y volver a un estado exacto del código en el tiempo.

  El uso de tags es una buena práctica porque mejora la organización, facilita la colaboración en equipos y permite rastrear el progreso del proyecto. Al integrar metadatos (con tags anotados), también ofrecen contexto como el autor, la fecha y un mensaje descriptivo, lo que es especialmente útil en entornos profesionales o con flujos como Gitflow. Además, los tags son compatibles con herramientas de integración continua (CI/CD) para automatizar despliegues basados en versiones.

```
git tag -a v1.0.0 -m "Lanzamiento estable 1.0.0"
git push origin v1.0.0
```

- **Uso de Pull Requests (PRs):**

  Las Pull Requests (PRs) son solicitudes para integrar cambios de una rama (como feature/*) en otra (como develop o main) ya sea en GitHub o cualquier otro repositorio remoto. Sirven para revisar código, colaborar y asegurar que los cambios sean seguros antes de fusionarlos. Incluyen comentarios, revisiones y pruebas automáticas.

- **Commits atómicos**

  Un commit = una idea
  Tamaño manejable (ideal: < 400 líneas).
  Funciona independientemente.

- **Mensajes de Commit efectivos:**

```
# ✗ Mal
git commit -m "fix"
# ✓ Bien
git commit -m "fix: corregir validación de email en formulario"
```

Convención a la hora de escribir mensajes de commit

git commit -m "feat: agregar autenticación OAuth2"

git commit -m "fix: resolver error 404 en rutas API"

git commit -m "docs: actualizar documentación de instalación"

- **Estrategia de Ramas**

main → Producción estable

develop → Integración continua

feature/* → Nuevas funcionalidades

hotfix/* → Correcciones urgentes

release/* → Preparación de versiones

- **Uso del archivo *.gitignore***

El archivo .gitignore es un archivo de texto en un repositorio Git que especifica qué archivos y directorios deben ser **ignorados** por Git. Esto significa que los archivos listados en .gitignore no serán rastreados, no aparecerán en git status, y no se incluirán en comandos como git add ..Su propósito es evitar que archivos innecesarios, temporales o sensibles (como configuraciones locales, archivos generados o credenciales) se añadan al repositorio.

Ubicación y Características

- Ubicación: Generalmente en la raíz del repositorio.
- Formato: Cada línea especifica un patrón (e.g., nombres de archivos, directorios, o comodines como *.log).
- No automático: No se crea automáticamente con git init; debe crearse manualmente o por herramientas como GitHub o IDEs.

Ejercicio Final

1. Implementar Git Flow completo (ramas main, develop y auxiliares).
2. Resolver merge conflicts intencionalmente.
3. Usar rebase como alternativa al merge para limpiar commits.
4. Eliminar ramas auxiliaries tanto en local como en remoto.