

# Taller de Introducción a Git

## "Introducción a Git: Gestión de Versiones desde Cero"

### [Objetivos del Taller](#)

### [Plan Detallado del Taller](#)

1. [Introducción](#)
2. [Fundamentos Teóricos y Configuración Inicial](#)
3. [Primeros Pasos Prácticos](#)
4. [Introducción a Ramas y Flujo de Trabajo](#)
5. [Trabajo con Repositorios Remotos](#)
6. [Resolución de Conflictos y Buenas Prácticas](#)

## Objetivos del Taller

- Comprender qué es Git y para qué sirve.
  - Familiarizarse con la terminología básica (repositorio, commit, branch, merge, etc.).
  - Aprender a configurar y utilizar Git en un entorno local.
  - Conocer el flujo básico de trabajo con repositorios remotos (GitHub/GitLab).
  - Practicar la resolución de conflictos y otras operaciones comunes.
- 

## Plan Detallado del Taller

### 1. Introducción

- **Contextualización y Motivación:**








- **¿Qué es Git?**

Git es un **sistema de control de versiones distribuido** que permite a los desarrolladores rastrear cambios en el código, colaborar en equipo y gestionar diferentes versiones de un proyecto de software.

#### ◆ **Características clave de Git**

- ✅ **Distribuido:** Cada usuario tiene una copia completa del historial del repositorio, lo que permite trabajar sin conexión.
- ✅ **Velocidad y eficiencia:** Opera rápidamente incluso en proyectos grandes.
- ✅ **Branching y Merging:** Facilita la creación de ramas (branches) para desarrollar nuevas funciones sin afectar el código principal.
- ✅ **Seguridad:** Usa funciones criptográficas para garantizar la integridad de los datos.
- ✅ **Colaboración remota:** Se integra con plataformas como **GitHub**, **GitLab** y **Bitbucket** para compartir código con otros.

## **Comandos básicos de Git**

-  `git init` – Inicializa un repositorio Git en una carpeta.
-  `git add` – Agrega archivos al área de preparación (staging).
-  `git commit -m "mensaje"` – Guarda los cambios en el historial.
-  `git push` – Envía los cambios a un repositorio remoto.
-  `git pull` – Obtiene actualizaciones del repositorio remoto.
-  `git branch` – Muestra o crea nuevas ramas.
-  `git merge` – Fusiona ramas.

### ◦ **Importancia del control de versiones en el desarrollo de software.**

El control de versiones es una **práctica esencial** en el desarrollo de software moderno, ya que permite gestionar cambios en el código, facilitar la colaboración y mejorar la seguridad del proyecto.

## **Beneficios Clave**

### ◆ **Historial de Cambios:**

Cada modificación queda registrada, lo que permite recuperar versiones anteriores y entender la evolución del proyecto.

### ◆ **Colaboración Eficiente:**

Varios desarrolladores pueden trabajar en el mismo código sin conflictos, ya que las ramas permiten trabajar en paralelo.

#### ◆ **Seguridad y Recuperación:**

Si un error crítico ocurre, se puede revertir fácilmente a una versión estable anterior.

#### ◆ **Trazabilidad y Responsabilidad:**

Cada cambio tiene un autor y un mensaje de commit, lo que ayuda a identificar quién hizo qué y por qué.


#### ◆ **Automatización e Integración Continua (CI/CD):**

Facilita pruebas automatizadas y despliegues sin intervención manual, mejorando la calidad del software.

### **Ejemplo Real en Equipos Colaborativos**

Imagina un equipo de 5 desarrolladores trabajando en una aplicación. Sin control de versiones:

- ✗ **Sobrescriben archivos** al compartir código manualmente.
- ✗ **Dificultad para sincronizar cambios**, causando errores frecuentes.
- ✗ **No hay historial de cambios**, lo que dificulta revertir errores.

Con Git, cada miembro trabaja en su propia rama, prueba cambios antes de fusionarlos y mantiene la estabilidad del código principal. 

### **Conclusión**

El control de versiones es una herramienta fundamental para **desarrollar software de manera organizada, segura y colaborativa**. Git se ha convertido en el estándar de la industria gracias a su eficiencia y flexibilidad.

#### ◦ **Ejemplos de casos reales y beneficios en equipos colaborativos.**

El uso de **Git y control de versiones** ha transformado la forma en que los equipos de desarrollo trabajan, permitiéndoles colaborar de manera eficiente y mantener la calidad del código. Veamos algunos **casos reales** y sus **beneficios**.

## ◆ Caso 1: Desarrollo de Software en una Startup 🚀

### Escenario:

Un equipo de 5 desarrolladores trabaja en una aplicación móvil. Sin control de versiones, compartir código sería un caos: archivos sobrescritos, pérdida de cambios y dificultad para rastrear errores.

### Solución con Git:

- Cada desarrollador trabaja en una **rama** independiente para nuevas funcionalidades.
- Una vez probados los cambios, se fusionan (merge) con la rama principal ( `main` ).
- Se usa **GitHub/GitLab** para revisar el código antes de aceptarlo (Pull Requests).

### Beneficios:

- ✓ Evita sobrescribir código de otros.
  - ✓ Permite trabajar en paralelo sin conflictos.
  - ✓ Mantiene un historial claro de cada cambio.
- 

## ◆ Caso 2: Mantenimiento de un Proyecto de Código Abierto 🌐

### Escenario:

Un repositorio de código abierto en GitHub recibe contribuciones de cientos de desarrolladores de diferentes países.

### Solución con Git:

- Los colaboradores hacen un **fork** del proyecto y trabajan en sus propios repositorios.
- Proponen cambios mediante **Pull Requests**, que los mantenedores revisan antes de fusionarlos.
- Se identifican y corrigen errores rápidamente gracias al historial de commits.

### Beneficios:

- ✓ Posibilita la colaboración global en proyectos grandes.
  - ✓ Permite validar cada cambio antes de incorporarlo.
  - ✓ Mejora la calidad del código con revisiones de la comunidad.
- 

## ◆ Caso 3: Corrección Rápida de Errores en Producción ⚠

### Escenario:

Una gran empresa de software lanza una actualización, pero los usuarios reportan un error crítico.

### Solución con Git:

- Se crea una **rama de emergencia** ( `hotfix` ) para solucionar el problema sin afectar las nuevas funcionalidades en desarrollo.
- Se implementa y prueba el parche, luego se fusiona con la rama principal.
- Se despliega una nueva versión con el error corregido.

### Beneficios:

- ✓ Responde rápidamente a incidentes sin afectar el desarrollo normal.
  - ✓ Permite probar soluciones antes de implementarlas.
  - ✓ Evita interrupciones prolongadas en los servicios.
- 

## 🎯 Conclusión

El uso de **Git y control de versiones** en equipos de desarrollo permite:

- ◆ **Trabajo colaborativo eficiente** sin conflictos.
- ◆ **Historial de cambios claro y recuperable.**
- ◆ **Integración con flujos de desarrollo modernos** como CI/CD.
- ◆ **Mayor calidad y seguridad del código.**

- **Agenda y Materiales:**

- Repaso del cronograma del taller.

- Distribución de material de apoyo (cheatsheet, links a recursos en línea).
  - Proporcionar el Cheat-Sheet
  - Proporcionar PDF con toda la información del taller
  - Links de aprendizaje:
    - <https://learngitbranching.js.org/>
    - <https://docs.github.com/es/get-started/start-your-journey/git-and-github-learning-resources>

## 2. Fundamentos Teóricos y Configuración Inicial

- **Conceptos Básicos:**

- Repositorio, commit, branch, merge, pull, push, etc.

A continuación, te explico los conceptos clave en **Git y control de versiones**:

---

### **Repositorio**

Es el lugar donde se almacena el código y su historial de cambios. Puede ser:

- **Local:** En tu computadora.
- **Remoto:** En plataformas como GitHub, GitLab o Bitbucket.

#### ◆ **Ejemplo:**

```
git init # Crea un nuevo repositorio local
```

---

### **Commit**

Un commit es un **registro de cambios** en el código. Es como una "foto" del estado actual del proyecto en ese momento.

#### ◆ **Ejemplo:**

```
git add archivo.txt # Agrega un archivo al área de preparación
git commit -m "Agregando archivo de configuración" # Guarda los cambios con un mensaje
```

✅ **Beneficio:** Permite regresar a versiones anteriores en caso de errores.

## **Branch (Rama)**

Una rama es una línea independiente de desarrollo. Permite trabajar en nuevas funciones sin afectar la rama principal.

### ◆ **Ejemplo:**

```
git branch nueva-rama # Crea una nueva rama
git checkout nueva-rama # Cambia a la nueva rama
```

✅ **Beneficio:** Permite que varios desarrolladores trabajen en paralelo sin conflictos.

## **Merge (Fusión de ramas)**

El merge combina los cambios de una rama en otra, generalmente integrando nuevas funciones en `main`.

### ◆ **Ejemplo:**

```
git checkout main # Cambiar a la rama principal
git merge nueva-rama # Fusionar la rama nueva-rama con main
```

✅ **Beneficio:** Une el trabajo de distintos desarrolladores en un solo flujo de trabajo.

## **Pull**

El comando `git pull` trae los cambios de un repositorio remoto al local.

### ◆ **Ejemplo:**

```
git pull origin main # Descarga y fusiona cambios de la rama main del remoto
```

✅ **Beneficio:** Mantiene tu repositorio actualizado con los cambios hechos por otros.

---

## Push

El comando `git push` envía los cambios locales al repositorio remoto.

### ◆ Ejemplo:

```
git push origin main # Sube los commits locales a GitHub
```

✅ **Beneficio:** Permite compartir el código con el equipo y guardar cambios en la nube.

---

## Clone

Clonar un repositorio significa descargar una copia completa en tu máquina.

### ◆ Ejemplo:

```
git clone https://github.com/usuario/repositorio.git
```

✅ **Beneficio:** Cualquiera puede obtener una copia exacta del código y contribuir.

---

## Status y Log

Para ver el estado del repositorio y el historial de commits:

### ◆ Ejemplo:

```
git status # Muestra cambios pendientes de commit  
git log # Muestra el historial de commits
```



✅ **Beneficio:** Permite revisar cambios y ver quién hizo qué.

---

## 🎯 Conclusión

💠 **Git facilita la colaboración, la organización del código y la recuperación de versiones anteriores.**

💠 **Es clave en cualquier equipo de desarrollo moderno.**

- **Diferencia entre repositorios locales y remotos.**

En **Git**, trabajamos con dos tipos de repositorios principales: **locales y remotos**. Ambos cumplen funciones esenciales en el control de versiones y el trabajo colaborativo.

---

## 📌 Repositorio Local

📌 **Definición:** Es el repositorio que está almacenado en tu computadora. Contiene todos los archivos, el historial de cambios y las ramas.

📌 **Características:**

✅ Permite trabajar sin conexión.

✅ Contiene el historial completo del proyecto.

✅ Solo es accesible desde tu máquina.

📌 **Ejemplo:** Crear un repositorio local

```
git init # Inicializa un nuevo repositorio local
git add archivo.txt # Agrega un archivo al área de preparación
git commit -m "Primer commit" # Guarda los cambios en el historial
```

## 🌐 Repositorio Remoto

📌 **Definición:** Es un repositorio alojado en una plataforma en línea como **GitHub, GitLab o Bitbucket**. Permite compartir y colaborar con otros desarrolladores.

📌 **Características:**

✅ Permite que varias personas trabajen en el mismo proyecto.

- ✓ Sirve como copia de seguridad en la nube.
- ✓ Se sincroniza con el repositorio local mediante `push` y `pull`.

📌 **Ejemplo:** Subir un repositorio local a GitHub

```
git remote add origin https://github.com/usuario/repositorio.git # Vincula el repositorio local con GitHub
git push -u origin main # Sube los cambios al repositorio remoto
```

## 🔄 Diferencias Clave

Característica	Repositorio Local 🖥️	Repositorio Remoto 🌐
<b>Ubicación</b>	En la computadora del usuario	En servidores en la nube (GitHub, GitLab, etc.)
<b>Accesibilidad</b>	Solo el usuario lo puede ver	Puede ser accedido por todo el equipo
<b>Conectividad</b>	No necesita internet	Requiere conexión para sincronizar
<b>Trabajo en equipo</b>	No permite colaboración directa	Facilita la colaboración entre desarrolladores
<b>Seguridad</b>	Puede perderse si la PC falla	Permite recuperar el código en caso de pérdida

## 🎯 Conclusión

- ◆ **El repositorio local** permite trabajar en el código de manera individual.
- ◆ **El repositorio remoto** facilita la colaboración y sirve como respaldo en la nube.

### • Instalación y Configuración de Git:

- Instalación en diferentes sistemas operativos (Windows, macOS, Linux).
- Configuración básica: `git config --global user.name "Tu Nombre"` y `git config --global user.email "tuemail@ejemplo.com"`.

Antes de empezar a usar Git, es importante configurarlo con tu nombre y correo electrónico. Esta información se usará para registrar la autoría de

cada **commit** que realices.

---

## **Configurar Nombre y Correo Electrónico**

Ejecuta los siguientes comandos en la terminal:

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

### **Ejemplo:**

```
git config --global user.name "Juan Pérez"
git config --global user.email "juan.perez@example.com"
```

 Ahora, cada commit que hagas llevará tu nombre y correo como autor.

---

## **Verificar la Configuración Actual**

Si quieres asegurarte de que tu configuración es correcta, usa:

```
git config --global --list
```


### Esto mostrará algo como:

```
user.name=Juan Pérez
user.email=juan.perez@example.com
```

## **Configurar Git Solo para un Proyecto Específico**

Si deseas que la configuración solo se aplique a un repositorio específico (sin afectar otros proyectos), omite `--global` :

```
git config user.name "Otro Nombre"
git config user.email "otro.email@example.com"
```

 **Importante:** Esto se aplicará solo dentro del repositorio actual.

---

## Eliminar o Cambiar Configuración

Si necesitas cambiar el usuario o correo, simplemente ejecuta los comandos con la nueva información.

Si deseas eliminar la configuración global:

```
git config --global --unset user.name
git config --global --unset user.email
```

## Conclusión

- ◆ Configurar tu **nombre y correo** es esencial para que Git pueda identificarte en cada commit.
- ◆ Puedes aplicar la configuración **globalmente** o **por proyecto**.
- ◆ Usar `git config --list` te ayuda a verificar la configuración actual.
- Revisión de herramientas auxiliares (terminal, Git Bash, Visual Studio Code, etc.).

## 3. Primeros Pasos Prácticos

- **Creación de un Repositorio Local:**
  - Iniciar un nuevo proyecto con `git init`.
  - Agregar archivos y realizar commits básicos:
    - Creación y edición de un archivo README.md.
    - Uso de `git add` y `git commit` con mensajes claros.
- **Exploración de Comandos Básicos:**
  - `git status` y `git log`: revisión del estado del repositorio y del historial de commits.
  - Ejercicio práctico: Los participantes crearán su propio repositorio local y harán al menos 3 commits.
- **Ejercicio Guiado en Vivo:**

- El instructor realiza una demostración en tiempo real en pantalla compartida.
- Los alumnos siguen paso a paso en sus propias máquinas.

## 4. Introducción a Ramas y Flujo de Trabajo

- **Conceptos de Branching y Merging:**

- ¿Qué es una rama y por qué usarla?

En **Git**, una **rama** es una línea independiente de desarrollo dentro de un proyecto. Permite trabajar en nuevas funcionalidades, corregir errores o experimentar sin afectar la versión principal del código.

### ¿Cómo Funciona una Rama?

- La rama principal se llama `main` o `master`.
- Puedes crear nuevas ramas para desarrollar funciones sin modificar `main`.
- Una vez que terminas los cambios en tu rama, puedes **fusionarlos** (**merge**) con `main`.

---

### ¿Por qué usar ramas en Git?

#### Trabajo en Paralelo

Diferentes desarrolladores pueden trabajar en distintas funcionalidades sin interferencias.

#### Seguridad y Organización

Las ramas permiten probar nuevas ideas sin afectar el código estable.

#### Facilita la Colaboración

Cada programador puede trabajar en su propia rama y luego integrar sus cambios al proyecto.

#### Corrección de Errores sin Riesgos

Si surge un error en producción, se puede crear una **rama de emergencia (hotfix)** sin alterar el desarrollo en curso.

## Comandos Básicos de Branching en Git

### 1. Ver las Ramas Disponibles

```
git branch
```

◆ Muestra la lista de ramas en el repositorio.

### 2. Crear una Nueva Rama

```
git branch nueva-funcionalidad
```

◆ Crea una rama llamada `nueva-funcionalidad`, pero no cambia a ella.

### 3. Cambiar a Otra Rama

```
git checkout nueva-funcionalidad
```

◆ Cambia a la rama `nueva-funcionalidad`.

|  TIP: Desde Git 2.23 puedes usar:

```
git switch nueva-funcionalidad
```

### 4. Crear y Cambiar de Rama en un Solo Paso

```
git checkout -b nueva-funcionalidad
```

◆ Crea la rama y cambia a ella inmediatamente.



TIP: También puedes usar:

```
git switch -c nueva-funcionalidad
```



## 5. Fusionar una Rama con `main` (Merge)

Cuando termines de trabajar en una rama, puedes fusionarla con `main`:

```
git checkout main # Cambiar a la rama principal  
git merge nueva-funcionalidad # Fusionar la rama con main
```



## 6. Eliminar una Rama

Si ya no necesitas una rama, puedes eliminarla:

```
git branch -d nueva-funcionalidad
```

- ◆ Solo se eliminará si la rama ha sido fusionada.
- ◆ Para forzar la eliminación, usa `-D` en lugar de `-d`.



## Conclusión

- ◆ Las ramas en Git permiten trabajar en nuevas funcionalidades de forma segura.
  - ◆ Facilitan el trabajo en equipo y la corrección de errores sin afectar la versión estable.
  - ◆ Aprender a manejar ramas es clave para cualquier flujo de desarrollo moderno.
- **Ejercicio Práctico:**
    - Crear una nueva rama para implementar una pequeña funcionalidad o cambio.

- Realizar cambios en la rama y luego integrarlos (merge) a la rama principal.
- Simulación de un escenario de colaboración: asignar roles a los participantes (por ejemplo, "desarrollador" y "integrador").

## 5. Trabajo con Repositorios Remotos

- **Introducción a GitHub/GitLab:**

- Creación de una cuenta en GitHub o GitLab.
- Creación y clonación de un repositorio remoto.

- **Flujo de Trabajo Remoto:**

- Comandos: `git remote add` , `git push` , `git pull` , `git clone` .
- Ejercicio práctico:
  - Los participantes subirán su repositorio local a GitHub.
  - Realizarán cambios en el repositorio remoto y actualizarán su copia local.

## 6. Resolución de Conflictos y Buenas Prácticas

- **Simulación de Conflictos:**

- Crear una situación intencional de conflicto (por ejemplo, dos participantes modifican la misma línea en un archivo).
- Demostrar cómo se produce un conflicto al hacer merge y cómo resolverlo manualmente.

- **Buenas Prácticas:**

- Mensajes de commit descriptivos.
- Frecuencia adecuada de commits.
- Uso de ramas para funcionalidades o correcciones específicas.
- Revisión y código en equipo.