Xabier Unai)

¡Prepárate para desbloquear todo el potencial de GIT y llevar tus proyectos al siguiente nivel

CONCEPTOS DE GIT

[Subtítulo del documento]

Tabla de contenido

[Introducción a Git y su importancia en el desarrollo de software 2](#_Toc160529723)

[Instalación y configuración inicial de Git 2](#_Toc160529724)

[Creación de un nuevo repositorio 2](#_Toc160529725)

[Zona de Trabajo, Staging Area y Almacén. 3](#_Toc160529726)

[Estructura de un repositorio 3](#_Toc160529727)

[Uso de comandos básicos: git add, git commit y git reset**.** 3](#_Toc160529728)

[Funcionamiento Interno de Git 4](#_Toc160529729)

[Concepto de objetos en Git: blobs, árboles y confirmaciones. 4](#_Toc160529730)

[Exploración de la base de datos de Git. 4](#_Toc160529731)

[Entender cómo Git gestiona y almacena los cambios a lo largo del tiempo. 5](#_Toc160529732)

[Gestión de Ramas y Referencias 5](#_Toc160529733)

[Introducción a las ramas en Git. 5](#_Toc160529734)

[Trabajar con referencias: HEAD, ramas y etiquetas 5](#_Toc160529735)

[Creación, cambio y eliminación de ramas 5](#_Toc160529736)

[Uso de etiquetas para marcar versiones importantes. 6](#_Toc160529737)

[Encontrar y Corregir Errores en el Historial 7](#_Toc160529738)

[Identificación de errores comunes en el historial. 7](#_Toc160529739)

[Encontrar Problemas y Deshacer Cambios en Git 8](#_Toc160529740)

[Uso de git blame para identificar la autoría de cambios. 8](#_Toc160529741)

[Utilización de git bisect para encontrar la introducción de errores. 8](#_Toc160529742)

[Deshacer cambios con git reset, git checkout y git revert: 8](#_Toc160529743)

[Reescribir el Historial en Git 9](#_Toc160529744)

[Modificación del historial utilizando git commit --amend: 9](#_Toc160529745)

[Uso de git reset para deshacer cambios de forma más flexible: 9](#_Toc160529746)

[Introducción al rebase: reorganización de la historia de la rama. 10](#_Toc160529747)

[Personalización Avanzada de Git 10](#_Toc160529748)

[Creación de alias para simplificar comandos repetitivos: 10](#_Toc160529749)

[Uso de hooks para automatizar acciones en determinados eventos de Git: 11](#_Toc160529750)

[Creación y utilización de plantillas para mensajes de confirmación: 11](#_Toc160529751)

[Gestión de proyectos complejos: combinación de varios proyectos con submódulos y subárboles. 12](#_Toc160529752)

# Introducción a Git y su importancia en el desarrollo de software

Git es un sistema de control de versiones distribuido ampliamente utilizado en el desarrollo de software. Su importancia radica en su capacidad para gestionar cambios en el código fuente de manera eficiente y colaborativa, lo que lo convierte en una herramienta fundamental para cualquier equipo de desarrollo de software.

Con Git, los desarrolladores pueden realizar un seguimiento de los cambios realizados en su código, revertir a versiones anteriores si es necesario, colaborar con otros desarrolladores de manera efectiva y mantener un historial completo de todas las modificaciones realizadas en el proyecto. Esto ayuda a garantizar la integridad y la coherencia del código base a lo largo del tiempo, lo que es esencial para el éxito a largo plazo de cualquier proyecto de software.

## Instalación y configuración inicial de Git

La instalación de Git es un proceso sencillo que puede realizarse en una variedad de sistemas operativos, incluyendo Windows, macOS y Linux.

Ejemplo de instalación en sistemas basados en Unix/Linux:

bashCopy code

sudo apt update

sudo apt install git

Ejemplo de instalación en macOS usando Homebrew:

bashCopy code

brew install git

Después de instalar Git, es importante configurar algunos ajustes iniciales, como el nombre de usuario y la dirección de correo electrónico. Esto es fundamental para que Git pueda identificar quién realiza cada cambio en el repositorio.

bashCopy code

git config --global user.name "Tu Nombre"

git config --global user.email "tu@email.com"

Estos comandos establecerán globalmente el nombre de usuario y la dirección de correo electrónico asociada con tus commits en todos los repositorios de Git en tu sistema.

## Creación de un nuevo repositorio

Una vez que Git está instalado y configurado, puedes crear un nuevo repositorio para comenzar a rastrear los cambios en tu proyecto. Para esto, simplemente navega hasta el directorio de tu proyecto y ejecuta el siguiente comando:

bashCopy code

git init

Este comando inicializa un nuevo repositorio Git en el directorio actual. A partir de este momento, Git comenzará a rastrear todos los cambios que realices en los archivos de este directorio.

En resumen, Git es una herramienta esencial en el desarrollo de software moderno. Su instalación y configuración inicial son pasos importantes para comenzar a aprovechar sus capacidades de control de versiones. Además, la creación de un nuevo repositorio marca el punto de partida para gestionar de manera eficiente los cambios en tu proyecto de software.

# Zona de Trabajo, Staging Area y Almacén.

## Estructura de un repositorio

En un repositorio Git, hay tres áreas principales que se utilizan para gestionar los cambios en el código fuente: la zona de trabajo (working directory), el área de preparación (staging area o index) y el almacén (repository o historial).

**1. Zona de Trabajo (Working Directory):** La zona de trabajo es el directorio de tu sistema de archivos donde trabajas con tus archivos de código fuente. Es aquí donde modificas, creas y eliminas archivos según sea necesario para desarrollar tu proyecto.

**Ejemplo:** Imagina que estás trabajando en un proyecto web y tienes un directorio llamado proyecto. Dentro de este directorio, encontrarás todos los archivos relacionados con tu proyecto, como HTML, CSS, JavaScript, imágenes, etc. Este directorio proyecto sería tu zona de trabajo.

**2. Área de Preparación (Staging Area o Index):** La zona de preparación es un área intermedia donde se preparan los cambios antes de confirmarlos en el repositorio. Permite seleccionar qué cambios se incluirán en el próximo commit. Los archivos en esta área se encuentran en un estado "preparado" para ser confirmados.

**Ejemplo:** Supongamos que has realizado cambios en tres archivos: index.html, style.css y script.js. Sin embargo, solo deseas confirmar los cambios en index.html y style.css, mientras que deseas dejar script.js para una confirmación posterior. Para hacer esto, añades solo los archivos index.html y style.css al área de preparación.

**3. Almacén (Repository o Historial):** El almacén es donde Git almacena el historial completo de tu proyecto, incluyendo todos los commits, ramas, etiquetas y demás metadatos. Cada vez que confirmas cambios en Git, estos cambios se registran en el almacén, lo que te permite realizar un seguimiento completo de la evolución de tu proyecto a lo largo del tiempo.

**Ejemplo:** Cuando confirmas los cambios en los archivos index.html y style.css, Git toma una instantánea de estos archivos en el momento de la confirmación y los almacena en el almacén como un nuevo commit junto con un mensaje descriptivo que explica los cambios realizados.

## Uso de comandos básicos: git add, git commit y git reset**.**

* git add: Se utiliza para agregar archivos o cambios específicos al área de preparación.

bashCopy code

git add nombre\_del\_archivo

* git commit: Se utiliza para confirmar los cambios en el área de preparación y almacenarlos en el repositorio.

bashCopy code

git commit -m "Mensaje descriptivo de los cambios"

* git reset: Se utiliza para deshacer cambios en el área de preparación o para revertir un commit.

bashCopy code

git reset nombre\_del\_archivo # para quitar un archivo del área de preparación

git reset HEAD~1 # para deshacer el último commit

En resumen, la zona de trabajo, el área de preparación y el almacén son componentes fundamentales en la gestión de cambios con Git. Comprender cómo funcionan y cómo utilizar los comandos básicos asociados (git add, git commit y git reset) es esencial para trabajar de manera efectiva con control de versiones en Git.

# Funcionamiento Interno de Git

## Concepto de objetos en Git: blobs, árboles y confirmaciones.

En Git, los archivos y directorios se manejan como objetos. Los tres tipos principales de objetos en Git son:

1. **Blob (Binary Large Object)**: Representa el contenido de un archivo en un estado específico. Cada versión diferente de un archivo se almacena como un blob distinto en la base de datos de Git.

Ejemplo: Si tienes un archivo llamado archivo.txt y realizas cambios en él, cada versión diferente de archivo.txt se almacenará como un blob separado en Git.

1. **Árbol (Tree)**: Representa un directorio en un momento específico. Contiene una lista de referencias a blobs (archivos) y a otros árboles (subdirectorios) junto con sus nombres y permisos.

Ejemplo: Si tienes un directorio llamado proyecto que contiene archivos y subdirectorios, la estructura de este directorio en un momento específico se representa como un árbol en Git.

1. **Confirmación (Commit)**: Representa un conjunto de cambios en el repositorio en un momento específico. Contiene referencias a uno o más árboles, además de metadatos como el autor, el mensaje del commit y un puntero al commit anterior (padre), lo que permite construir una línea de tiempo de los cambios en el repositorio.

Ejemplo: Cuando realizas un commit en Git, estás creando una nueva confirmación que registra los cambios realizados desde el último commit, junto con la información sobre quién realizó los cambios, cuándo se realizaron y un mensaje descriptivo.

## Exploración de la base de datos de Git.

La base de datos de Git es donde se almacenan todos los objetos del repositorio, incluidos blobs, árboles y confirmaciones. Estos objetos se almacenan de manera eficiente utilizando una estructura de datos llamada **grafo dirigido acíclico** (DAG). Este grafo representa la historia del proyecto, donde cada commit apunta a su commit padre o padres.

Puedes explorar la base de datos de Git utilizando comandos como git log para ver la historia de los commits, git show para ver los detalles de un commit específico y git ls-tree para ver el contenido de un árbol específico.

## Entender cómo Git gestiona y almacena los cambios a lo largo del tiempo.

Git gestiona y almacena los cambios a lo largo del tiempo mediante la creación de commits que representan instantáneas del estado del repositorio en diferentes momentos. Cada commit contiene referencias a los árboles que representan el estado del directorio en ese momento, lo que permite reconstruir cualquier versión anterior del proyecto.

Cuando realizas cambios en un archivo y realizas un commit, Git crea nuevos blobs para los archivos modificados y un nuevo árbol que representa el estado del directorio después de los cambios. Luego, crea una nueva confirmación que apunta al nuevo árbol y al commit anterior, formando así una cadena de commits que registra la historia del proyecto.

En resumen, Git maneja y almacena los cambios a lo largo del tiempo mediante la creación de objetos como blobs, árboles y confirmaciones, que se organizan en una base de datos utilizando una estructura de grafo dirigido acíclico. Esto permite gestionar eficientemente la historia del proyecto y reconstruir cualquier versión anterior del mismo.

# Gestión de Ramas y Referencias

## Introducción a las ramas en Git.

Las ramas en Git son una característica poderosa que permite trabajar en diferentes líneas de desarrollo de forma independiente. Cada repositorio Git comienza con una rama principal llamada master (o main en algunos casos), pero es común crear y trabajar en varias ramas para desarrollar nuevas características, corregir errores o experimentar con ideas sin afectar la rama principal.

Trabajar con referencias: HEAD, ramas y etiquetas

* **HEAD**: Es un puntero especial que indica la última confirmación realizada en la rama actual. En otras palabras, HEAD apunta al commit en el que te encuentras actualmente.
* **Ramas**: Son punteros móviles que apuntan a commits específicos en la línea de tiempo del proyecto. Cuando creas un nuevo commit en una rama, esta se mueve hacia adelante para apuntar al nuevo commit.
* **Etiquetas**: Son nombres descriptivos asociados con commits específicos y se utilizan para marcar versiones importantes de tu proyecto. A diferencia de las ramas, las etiquetas no se mueven una vez que se crean y se utilizan comúnmente para identificar versiones estables o lanzamientos.

## **Creación, cambio y eliminación de ramas**

* Para crear una nueva rama, utiliza el comando git branch seguido del nombre de la nueva rama:

bashCopy code

git branch nombre\_de\_la\_rama

* Para cambiar a una rama existente, utiliza el comando git checkout seguido del nombre de la rama:

bashCopy code

git checkout nombre\_de\_la\_rama

* También puedes crear una nueva rama y cambiar a ella en un solo paso utilizando el comando git checkout -b:

bashCopy code

git checkout -b nombre\_de\_la\_rama

* Para eliminar una rama, utiliza el comando git branch -d seguido del nombre de la rama:

bashCopy code

git branch -d nombre\_de\_la\_rama

## **Uso de etiquetas para marcar versiones importantes.**

Para crear una etiqueta en un commit específico, utiliza el comando git tag seguido del nombre de la etiqueta y el hash del commit:

bashCopy code

git tag nombre\_de\_la\_etiqueta hash\_del\_commit

Puedes listar todas las etiquetas existentes en el repositorio utilizando git tag. También puedes crear etiquetas anotadas que incluyan información adicional como el autor, la fecha y un mensaje descriptivo utilizando la opción -a:

bashCopy code

git tag -a nombre\_de\_la\_etiqueta -m "Mensaje descriptivo" hash\_del\_commit

En resumen, las ramas en Git te permiten trabajar en diferentes líneas de desarrollo de forma independiente, mientras que las etiquetas se utilizan para marcar versiones importantes del proyecto. Con una comprensión sólida de cómo crear, cambiar y eliminar ramas, así como de cómo utilizar etiquetas para marcar versiones, puedes gestionar eficazmente la historia de tu proyecto en Git.

# Encontrar y Corregir Errores en el Historial

## Identificación de errores comunes en el historial.

Los errores comunes en el historial de Git pueden incluir commits con errores de código, cambios incorrectos en archivos, confirmaciones incompletas o commits que rompen la funcionalidad del proyecto.

**Uso de comandos como git log y git diff para revisar cambios:**

* **git log**: Permite revisar el historial de commits en el repositorio. Muestra información como el hash del commit, el autor, la fecha y el mensaje del commit. Puedes utilizar opciones adicionales para personalizar la salida del registro.

bashCopy code

git log

* **git diff**: Permite ver los cambios entre dos commits, entre el área de preparación y la zona de trabajo, o entre un commit y el área de preparación. Esto te ayuda a identificar los cambios realizados en los archivos.

bashCopy code

git diff commit\_1 commit\_2

**Corrección de errores utilizando git revert y git cherry-pick:**

* **git revert**: Se utiliza para deshacer los cambios introducidos por un commit específico. Crea un nuevo commit que revierte los cambios realizados en el commit objetivo. Esto es útil para corregir errores sin eliminar la historia de commits.

bashCopy code

git revert hash\_del\_commit

* **git cherry-pick**: Se utiliza para aplicar los cambios de un commit específico a la rama actual. Esto es útil cuando necesitas traer cambios específicos de otra rama sin fusionar toda la rama.

bashCopy code

git cherry-pick hash\_del\_commit

**Ejemplo de uso:**

Supongamos que has identificado un error en el commit con el hash abcd1234 y necesitas corregirlo sin eliminar la historia de commits.

1. Utiliza git revert para deshacer los cambios introducidos por ese commit:

bashCopy code

git revert abcd1234

1. Git creará un nuevo commit que revierte los cambios introducidos por el commit abcd1234, manteniendo la historia intacta.
2. Si necesitas traer cambios específicos de otro commit a la rama actual, puedes utilizar git cherry-pick. Por ejemplo, para aplicar los cambios del commit defg5678:

bashCopy code

git cherry-pick defg5678

Estos comandos te ayudarán a identificar y corregir errores en el historial de Git de manera efectiva, manteniendo la integridad y la historia del proyecto.

# Encontrar Problemas y Deshacer Cambios en Git

En el desarrollo de software, es común encontrarse con situaciones en las que es necesario identificar problemas en el historial de cambios o deshacer modificaciones realizadas. Git proporciona herramientas que facilitan estas tareas.

## **Uso de git blame para identificar la autoría de cambios.**

El comando git blame muestra quién modificó cada línea de un archivo y en qué confirmación se realizaron esos cambios. Esto puede ser útil para entender quién introdujo un determinado cambio o para encontrar a la persona responsable de una sección específica del código.

bashCopy code

git blame nombre\_del\_archivo

## **Utilización de git bisect para encontrar la introducción de errores.**

git bisect es una herramienta de búsqueda binaria que te ayuda a encontrar qué commit introdujo un error específico. Comienza especificando un commit que contiene el error (conocido como "malo") y otro commit en el que sabes que el error no existe (conocido como "bueno"). Git utiliza la búsqueda binaria para identificar rápidamente el commit que introdujo el problema.

bashCopy code

git bisect start

git bisect bad # Indica el commit actual como "malo"

git bisect good <commit> # Indica un commit conocido como "bueno"

Git te llevará al commit medio entre el "bueno" y el "malo". Debes revisar si el error persiste y luego indicar si ese commit es "bueno" o "malo" utilizando los comandos git bisect good o git bisect bad. Repite este proceso hasta que Git identifique el commit que introdujo el error.

## **Deshacer cambios con git reset, git checkout y git revert:**

* git reset: Se utiliza para deshacer cambios en el repositorio. Puedes utilizarlo para retroceder a un estado anterior eliminando commits o cambios del área de preparación.

bashCopy code

git reset --hard HEAD~1 # Deshace el último commit y elimina los cambios realizados

* git checkout: Se utiliza para cambiar entre ramas o para revertir cambios en archivos específicos. También se puede utilizar para deshacer cambios locales no confirmados.

bashCopy code

git checkout nombre\_de\_la\_rama # Cambia a la rama especificada

git checkout -- nombre\_del\_archivo # Revierte los cambios no confirmados en el archivo

* git revert: Se utiliza para deshacer un commit específico creando un nuevo commit que revierte los cambios realizados en el commit original.

bashCopy code

git revert hash\_del\_commit

Estas herramientas te permiten encontrar problemas en el historial de cambios y deshacer modificaciones en Git de manera efectiva, lo que te ayuda a mantener la integridad de tu proyecto y a solucionar problemas de manera rápida y eficiente.

# Reescribir el Historial en Git

En ocasiones, es necesario modificar el historial de Git para corregir errores, mejorar la claridad o reorganizar la línea de tiempo del proyecto. Git proporciona varias herramientas para reescribir el historial de manera efectiva.

## **Modificación del historial utilizando git commit --amend:**

El comando git commit --amend te permite realizar cambios en el commit más reciente sin crear un nuevo commit. Esto puede ser útil para corregir errores tipográficos en el mensaje del commit, añadir archivos olvidados al commit o incluso modificar los cambios realizados en los archivos.

bashCopy code

git add archivo\_modificado

git commit --amend

Este comando abrirá tu editor de texto predeterminado para que puedas realizar los cambios necesarios en el mensaje del commit. Si solo deseas cambiar el mensaje del commit, puedes utilizar la opción -m seguida del nuevo mensaje:

bashCopy code

git commit --amend -m "Nuevo mensaje del commit"

## **Uso de git reset para deshacer cambios de forma más flexible:**

git reset se utiliza para deshacer cambios en el historial, pero también puede utilizarse para deshacer cambios en el área de preparación o en el directorio de trabajo.

* git reset --soft: Deshace el commit, pero mantiene los cambios en el área de preparación.
* git reset --mixed: Deshace el commit y los cambios en el área de preparación, pero mantiene los cambios en el directorio de trabajo (archivos modificados).
* git reset --hard: Deshace el commit y los cambios en el área de preparación, eliminando también los cambios en el directorio de trabajo.

bashCopy code

git reset --soft HEAD~1 # Deshace el último commit, manteniendo los cambios en el área de preparación

git reset --mixed HEAD~1 # Deshace el último commit y los cambios en el área de preparación, pero mantiene los cambios en el directorio de trabajo

git reset --hard HEAD~1 # Deshace el último commit y todos los cambios realizados en él

## **Introducción al rebase: reorganización de la historia de la rama.**

git rebase es una herramienta poderosa que te permite reorganizar la historia de una rama al reubicar los commits sobre otra base. Esto puede ser útil para mantener un historial de cambios limpio y lineal, evitando bifurcaciones innecesarias.

bashCopy code

git checkout mi\_rama

git rebase rama\_base

Este comando toma todos los commits realizados en mi\_rama que no están en rama\_base, deshace esos commits temporalmente, aplica los commits de rama\_base y luego vuelve a aplicar los commits de mi\_rama uno por uno. Esto crea una línea de tiempo lineal y limpia.

En resumen, estas herramientas te permiten reescribir el historial de Git de manera efectiva, ya sea corrigiendo errores en commits anteriores, deshaciendo cambios de forma flexible o reorganizando la historia de la rama utilizando rebase. Sin embargo, ten en cuenta que reescribir el historial puede afectar a otros colaboradores del proyecto, por lo que es importante utilizar estas herramientas con cuidado y comunicar los cambios realizados.

# Personalización Avanzada de Git

Git ofrece varias características avanzadas que permiten personalizar y optimizar tu flujo de trabajo. Estas incluyen la creación de alias para comandos repetitivos, el uso de hooks para automatizar acciones, la creación y utilización de plantillas para mensajes de confirmación, y la gestión de proyectos complejos mediante submódulos y subárboles.

## **Creación de alias para simplificar comandos repetitivos:**

Los alias en Git te permiten crear atajos para comandos largos o frecuentemente utilizados. Puedes definir alias a nivel global para que estén disponibles en todos tus repositorios.

bashCopy code

git config --global alias.co checkout

git config --global alias.st status

git config --global alias.br branch

Esto te permitirá usar git co, git st y git br en lugar de git checkout, git status y git branch, respectivamente.

## **Uso de hooks para automatizar acciones en determinados eventos de Git:**

Los hooks de Git son scripts que se ejecutan automáticamente en respuesta a eventos específicos de Git, como la confirmación, el push, o la fusión de ramas. Puedes utilizarlos para realizar acciones personalizadas, como ejecutar pruebas automatizadas, actualizar la documentación, o notificar a otros miembros del equipo sobre cambios importantes.

Por ejemplo, puedes crear un script pre-commit que ejecute pruebas antes de cada confirmación:

bashCopy code

#!/bin/bash

make test

## **Creación y utilización de plantillas para mensajes de confirmación:**

Puedes crear plantillas para mensajes de confirmación que ayuden a mantener un formato consistente en tus mensajes de commit y a incluir la información relevante.

1. Crea un archivo .gitmessage con el formato deseado:

plaintextCopy code

Asunto:

Descripción:

Tareas:

- [ ]

Refs:

1. Configura Git para usar esta plantilla:

bashCopy code

git config --global commit.template ~/.gitmessage

Ahora, cuando hagas una confirmación, se abrirá automáticamente el editor de texto con la plantilla especificada.

## Gestión de proyectos complejos: combinación de varios proyectos con submódulos y subárboles.

* **Submódulos:** Permiten incluir un repositorio Git dentro de otro como una subcarpeta. Esto puede ser útil cuando deseas incluir otro proyecto en tu repositorio pero mantenerlo como un proyecto separado y poder actualizarlo independientemente.

bashCopy code

git submodule add <URL\_del\_repositorio> <ruta\_local>

* **Subárboles:** Son similares a los submódulos, pero en lugar de incluir otro repositorio como una subcarpeta, importan el historial de otro repositorio como una carpeta dentro del árbol de directorios del repositorio principal.

bashCopy code

git subtree add --prefix=<ruta\_local> <URL\_del\_repositorio> <rama>

En resumen, la personalización avanzada de Git te permite optimizar tu flujo de trabajo, automatizar tareas repetitivas y gestionar proyectos complejos de manera más eficiente. Con alias, hooks, plantillas y la capacidad de trabajar con submódulos y subárboles, puedes adaptar Git a las necesidades específicas de tu proyecto y equipo de desarrollo.