

Introducción al Control de Versiones

Julian F. Latorre






25 de octubre de 2024

Objetivos de la sesión

- ▶ Comprender qué es el control de versiones y por qué es importante
- ▶ Explorar los diferentes tipos de sistemas de control de versiones
- ▶ Aprender los conceptos básicos del control de versiones
- ▶ Entender las mejores prácticas en el uso de control de versiones

¿Qué es el Control de Versiones?

El control de versiones es un sistema que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.

- ▶  Mantiene un historial de cambios
- ▶  Permite revertir a versiones anteriores
- ▶  Facilita la colaboración en equipo
- ▶  Posibilita el trabajo en paralelo (ramificación)
- ▶  Proporciona respaldo y seguridad

Escenarios de uso

1. Un desarrollador introduce un error accidentalmente:

```
git revert HEAD
```

2. Varios desarrolladores trabajan en diferentes partes:

```
git merge feature-branch
```

3. Un cliente solicita una versión anterior:

```
git checkout v1.2.3
```

Tipos de Sistemas de Control de Versiones

1. Sistemas de Control de Versiones Locales
2. Sistemas de Control de Versiones Centralizados (CVCS)
3. Sistemas de Control de Versiones Distribuidos (DVCS)

Sistemas de Control de Versiones Locales

- ▶ Funcionan en un solo computador
- ▶ Ejemplos: RCS (Revision Control System)
- ▶ **Ventajas:** Simples de usar
- ▶ **Desventajas:** No permiten colaboración, riesgo de pérdida de datos

Sistemas de Control de Versiones Centralizados (CVCS)

- ▶ Utilizan un servidor central que almacena todas las versiones
- ▶ Ejemplos: Subversion (SVN), Perforce
- ▶ **Ventajas:** Facilitan la colaboración, control administrativo
- ▶ **Desventajas:** El servidor central es un punto único de fallo

Sistemas de Control de Versiones Distribuidos (DVCS)

- ▶ Cada usuario tiene una copia completa del repositorio
- ▶ Ejemplos: Git, Mercurial
- ▶ **Ventajas:** No dependen de un servidor central, permiten trabajo offline
- ▶ **Desventajas:** Curva de aprendizaje más pronunciada

Conceptos Básicos del Control de Versiones

- ▶ Repositorio
- ▶ Commit
- ▶ Branch (Rama)
- ▶ Merge (Fusión)
- ▶ Conflicto

Repositorio

Un repositorio es el lugar donde se almacenan todos los archivos de tu proyecto junto con su historial de cambios.

```
# Crear un nuevo repositorio
```

```
git init mi_proyecto
```

```
# Clonar un repositorio existente
```

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

Commit

Un commit representa un punto específico en la historia de tu proyecto. Cada commit captura un snapshot del estado de tu proyecto en ese momento.

```
# Añadir cambios al área de preparación
git add archivo.txt

# Crear un nuevo commit
git commit -m "Añadir nueva funcionalidad"
```

Branch (Rama)

Una branch es una línea independiente de desarrollo. Permite trabajar en diferentes características o experimentos sin afectar la línea principal de desarrollo.

```
# Crear una nueva rama
```

```
git branch nueva-caracteristica
```

```
# Cambiar a la nueva rama
```

```
git checkout nueva-caracteristica
```

```
# Crear y cambiar a una nueva rama en un solo comando
```

```
git checkout -b otra-caracteristica
```

Merge (Fusión)

El proceso de combinar cambios de diferentes ramas se llama merge.

```
# Fusionar la rama 'feature' en la rama actual  
git merge feature
```

```
# Si hay conflictos, resolverlos y luego:  
git add archivo_con_conflictos.txt  
git commit -m "Resolver conflictos de fusión"
```

Conflicto

Un conflicto ocurre cuando se realizan cambios incompatibles en el mismo archivo en diferentes ramas.

- ▶ Git marca las áreas conflictivas en los archivos
- ▶ El desarrollador debe resolver manualmente los conflictos
- ▶ Después de resolver, se debe hacer un nuevo commit

Mejores Prácticas en el Uso de Control de Versiones

- ▶ Commits frecuentes y pequeños
- ▶ Mensajes de commit descriptivos
- ▶ Uso efectivo de ramas
- ▶ Revisión de código
- ▶ Uso de archivos `.gitignore`

Commits frecuentes y pequeños

Realiza commits frecuentes y pequeños. Esto hace que sea más fácil entender los cambios y revertirlos si es necesario.

```
# Ejemplo de un buen commit  
git commit -m "Agregar validación de correo electrónico en el formulario de registro"
```


Mensajes de commit descriptivos

Escribe mensajes de commit claros y descriptivos. Un buen mensaje de commit debería completar la frase: "Si se aplica, este commit...".

```
# Ejemplo de un buen mensaje de commit  
git commit -m "Corregir error de cálculo en la función  
de descuento"
```

Uso efectivo de ramas

Utiliza ramas para desarrollar nuevas características o experimentar sin afectar el código principal.

```
# Crear una nueva rama
git branch nueva-caracteristica

# Cambiar a la nueva rama
git checkout nueva-caracteristica
```

Revisión de código

Implementa un proceso de revisión de código antes de fusionar cambios en la rama principal.

- ▶ Mejora la calidad del código
- ▶ Facilita la detección temprana de errores
- ▶ Promueve el intercambio de conocimientos en el equipo

Uso de archivos .gitignore

Utiliza archivos .gitignore para excluir archivos y directorios que no deben ser versionados.

```
# Ejemplo de contenido de .gitignore
node_modules/
.env
*.log
```

Herramientas populares de Control de Versiones

- ▶ Git
- ▶ Subversion (SVN)
- ▶ Mercurial

Git

Git es el sistema de control de versiones más popular actualmente.

- ▶ Distribuido
- ▶ Rápido y eficiente
- ▶ Soporta ramificación y fusión de manera eficiente
- ▶ Tiene una gran comunidad y ecosistema

Subversion (SVN)

Subversion es un sistema de control de versiones centralizado.

- ▶ Centralizado
- ▶ Más simple de aprender que Git
- ▶ Manejo eficiente de archivos binarios grandes

Mercurial

Mercurial es otro sistema de control de versiones distribuido.

- ▶ Distribuido
- ▶ Interfaz de usuario más simple que Git
- ▶ Extensible a través de extensiones

Conclusión

El control de versiones es una herramienta fundamental en el desarrollo de software moderno. Proporciona numerosos beneficios:

- ▶ Seguimiento de cambios en el código
- ▶ Facilitación de la colaboración en equipo
- ▶ Capacidad de experimentar con nuevas ideas sin riesgo
- ▶ Respaldo y recuperación de versiones anteriores

Recursos adicionales

Para seguir aprendiendo sobre control de versiones, recomendamos:

- ▶ Pro Git Book
- ▶ Atlassian Git Tutorial
- ▶ Learn Git Branching

¿Preguntas?

¿Tienen alguna pregunta sobre lo que hemos cubierto hoy?