Fusión y Conflictos

Merge, Rebase, Squash y el Arte de Sobrevivir

Clase 5 · Fusión y Conflictos

GLUD — Grupo GNU/Linux Universidad Distrital

Control de Versiones y Desarrollo Colaborativo

Mapa de ruta

1. El Momento de la Verdad
2. Git Merge
3. Git Rebase
4. Squash: Aplastar Commits
5. Merge vs Rebase vs Squash
6. Resolución de Conflictos

1

El Momento de la Verdad

¿Por Qué Fusionar?

El Problema

Después de trabajar en ramas separadas, eventualmente necesitas **integrar** esos cambios de vuelta.

¿Por Qué Fusionar?

El Problema

Después de trabajar en ramas separadas, eventualmente necesitas **integrar** esos cambios de vuelta.

Escenarios comunes:

- Feature completada → merge a develop
- Hotfix urgente → merge a main
- Sincronizar tu rama con main
- Integrar trabajo de otro developer

¿Por Qué Fusionar?

El Problema

Después de trabajar en ramas separadas, eventualmente necesitas **integrar** esos cambios de vuelta.

Escenarios comunes:

- Feature completada → merge a develop
- Hotfix urgente → merge a main
- Sincronizar tu rama con main
- Integrar trabajo de otro developer

El miedo real:

- ¿Y si rompo todo?
- ¿Qué son estos conflictos?
- ¿Por qué Git me odia?
- ¿Borro el repo y lo clono de nuevo?

La “Solución” del Principiante

Cuando hay un conflicto en Git...

La “Solución” del Principiante

Cuando hay un conflicto en Git...

```
rm -rf mi-proyecto/
git clone https://github.com/equipo/proyecto.git
# Copiar mis cambios manualmente...
# Llorar un poco...
git add . && git commit -m "cosas"
```

La “Solución” del Principiante

Cuando hay un conflicto en Git...

```
rm -rf mi-proyecto/
git clone https://github.com/equipo/proyecto.git
# Copiar mis cambios manualmente...
# Llorar un poco...
git add . && git commit -m "cosas"
```

“Si funciona, no es estúpido” — Mentira, sí lo es

> *Merge conflicts are not bugs — they are features that tell you two people cared enough to change the same thing.*
>
> — Sabiduría de Stack Overflow

Las Herramientas de Fusión

Git ofrece varias formas de integrar cambios:

git merge

- Combina historiales
- Preserva contexto
- Crea commit de merge
- El más común

Las Herramientas de Fusión

Git ofrece varias formas de integrar cambios:

git merge

- Combina historiales
- Preserva contexto
- Crea commit de merge
- El más común

git rebase

- Reescribe historial
- Historial lineal
- Más limpio
- Más peligroso

Las Herramientas de Fusión

Git ofrece varias formas de integrar cambios:

git merge

- Combina historiales
- Preserva contexto
- Crea commit de merge
- El más común

git rebase

- Reescribe historial
- Historial lineal
- Más limpio
- Más peligroso

git squash

- Combina commits
- Un solo commit
- Historial simple
- Pierde detalles

2

Git Merge

¿Qué es git merge?

Definición

`git merge` combina el historial de dos ramas, integrando los cambios de una rama en otra.

¿Qué es git merge?

Definición

`git merge` combina el historial de dos ramas, integrando los cambios de una rama en otra.

Sintaxis básica

```
git switch main
git merge feature/login
```

Esto trae los cambios de `feature/login` hacia `main`

¿Qué es git merge?

Definición

`git merge` combina el historial de dos ramas, integrando los cambios de una rama en otra.

Sintaxis básica

```
git switch main
git merge feature/login
```

Esto trae los cambios de `feature/login` hacia `main`

Importante

Siempre estás **trayendo** cambios **hacia** la rama en la que te encuentras.

Tipos de Merge

Git decide automáticamente qué tipo de merge hacer:

Fast-Forward

Cuándo ocurre:
La rama base no tiene commits nuevos desde que se creó la feature.

Resultado:
Git simplemente "avanza" el puntero. No crea commit de merge.

Tipos de Merge

Git decide automáticamente qué tipo de merge hacer:

Fast-Forward

Cuándo ocurre:
La rama base no tiene commits nuevos desde que se creó la feature.

Resultado:
Git simplemente "avanza" el puntero. No crea commit de merge.

3-Way Merge (Recursive)

Cuándo ocurre:
Ambas ramas tienen commits nuevos (historiales divergentes).

Resultado:
Git crea un nuevo "merge commit" con dos padres.

Fast-Forward Merge: Visualización

ANTES del merge:

Fast-Forward Merge: Visualización

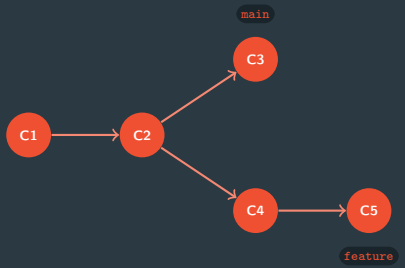
ANTES del merge:

DESPUÉS de `git merge feature`:

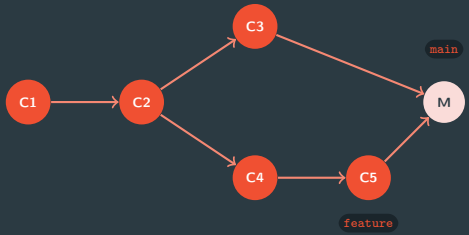
main simplemente "avanzó" hasta donde estaba feature. No hay commit nuevo.

3-Way Merge: Visualización

ANTES del merge (historiales divergentes):



DESPUÉS de `git merge feature`:



Git crea un "merge commit" (M) que tiene DOS padres: C3 y C5

Forzar el Tipo de Merge

Forzar merge commit

```
git merge --no-ff feature
```

Aunque sea posible fast-forward, crea un commit de merge. Útil para mantener registro de qué fue una feature.

Forzar el Tipo de Merge

Forzar merge commit

```
git merge --no-ff feature
```

Aunque sea posible fast-forward, crea un commit de merge. Útil para mantener registro de qué fue una feature.

Forzar fast-forward

```
git merge --ff-only feature
```

Solo hace merge si es fast-forward. Falla si hay divergencia. Útil para scripts de CI.

Forzar el Tipo de Merge

Forzar merge commit

```
git merge --no-ff feature
```

Aunque sea posible fast-forward, crea un commit de merge. Útil para mantener registro de qué fue una feature.

Forzar fast-forward

```
git merge --ff-only feature
```

Solo hace merge si es fast-forward. Falla si hay divergencia. Útil para scripts de CI.

Recomendación

En Git Flow, usa `--no-ff` para features. Así puedes ver en el historial dónde empezó y terminó cada feature.

Ejemplo Práctico: Merge Exitoso

Flujo completo de merge

```
# Asegurarse de estar en main actualizado
git switch main
git pull origin main

# Hacer el merge
git merge feature/login

# Si todo sale bien...
git push origin main

# Limpiar la rama
git branch -d feature/login
```

3

Git Rebase

¿Qué es git rebase?

Definición

`git rebase` **reescribe** el historial de commits, moviendo tu rama a una nueva base (el tip de otra rama).

¿Qué es git rebase?

Definición

`git rebase` **reescribe** el historial de commits, moviendo tu rama a una nueva base (el tip de otra rama).

Concepto:

- "Re-basar" = cambiar la base
- Toma tus commits
- Los "replanta" sobre otro punto
- Crea nuevos commits (nuevos hashes)

¿Qué es git rebase?

Definición

`git rebase` **reescribe** el historial de commits, moviendo tu rama a una nueva base (el tip de otra rama).

Concepto:

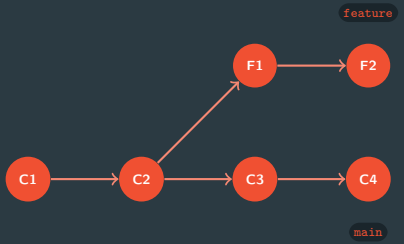
- "Re-basar" = cambiar la base
- Toma tus commits
- Los "replanta" sobre otro punto
- Crea nuevos commits (nuevos hashes)

Resultado:

- Historial lineal
- Sin commits de merge
- Más limpio visualmente
- Pero... reescribe historial

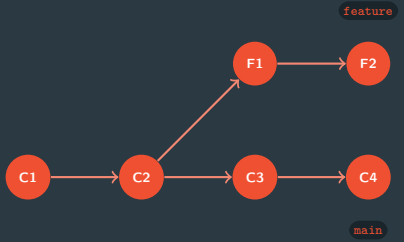
Rebase: Visualización

ANTES del rebase:



Rebase: Visualización

ANTES del rebase:



DESPUÉS de `git rebase main` (estando en feature):

Cuándo Usar Rebase

Usar rebase:

- Actualizar tu feature con cambios de main
- Antes de crear un PR (historial limpio)
- En ramas locales no compartidas
- Para limpiar historial personal

Cuándo Usar Rebase

Usar rebase:

- Actualizar tu feature con cambios de main
- Antes de crear un PR (historial limpio)
- En ramas locales no compartidas
- Para limpiar historial personal

NO usar rebase:

- En ramas públicas/compartidas
- Después de hacer push
- En main o develop
- Si no entiendes las consecuencias

Cuándo Usar Rebase

Usar rebase:

- Actualizar tu feature con cambios de main
- Antes de crear un PR (historial limpio)
- En ramas locales no compartidas
- Para limpiar historial personal

NO usar rebase:

- En ramas públicas/compartidas
- Después de hacer push
- En main o develop
- Si no entiendes las consecuencias

La Regla de Oro del Rebase

NUNCA hagas rebase de commits que ya hayas pusheado. Otros pueden tener esos commits y causarás un caos dimensional.

Ejemplo Práctico: Rebase

Actualizar feature con cambios de main

```
# Estás en tu feature, main tiene cambios nuevos
git switch feature/login

# Traer cambios de main
git fetch origin main

# Rebasar tu feature sobre main
git rebase origin/main

# Si hay conflictos, resolverlos y:
git rebase --continue

# Si todo sale mal:
git rebase --abort
```

> *Do not rebase commits that exist outside your repository and that people may have based work on.*
>
> — Pro Git Book

4

Squash: Aplastar Commits

¿Qué es Squash?

Definición

`squash` combina múltiples commits en uno solo, simplificando el historial.

¿Qué es Squash?

Definición

`squash` combina múltiples commits en uno solo, simplificando el historial.

¿Por qué aplastar commits?

Tu historial real:

- "WIP"
- "arreglo typo"
- "ahora sí"
- "por qué no funciona"
- "FUNCIONA!!!"

¿Qué es Squash?

Definición

`squash` combina múltiples commits en uno solo, simplificando el historial.

¿Por qué aplastar commits?

Tu historial real:

- "WIP"
- "arreglo typo"
- "ahora sí"
- "por qué no funciona"
- "FUNCIONA!!!"

Lo que quieres en main:

- "feat: implementar sistema de login con OAuth2"

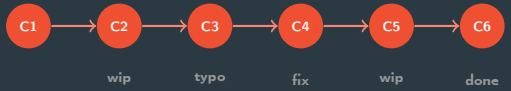
Un commit limpio y descriptivo

Squash: Visualización

ANTES (6 commits):

Squash: Visualización

ANTES (6 commits):



DESPUÉS del squash (1 commit):

Todos los commits intermedios se combinan en uno solo con un mensaje limpio

Formas de Hacer Squash

1. Squash en merge (GitHub/GitLab)

```
# En la interfaz web, botón "Squash and Merge"
```

Formas de Hacer Squash

1. Squash en merge (GitHub/GitLab)

```
# En la interfaz web, botón "Squash and Merge"
```

2. Squash con rebase interactivo

```
git rebase -i HEAD~5 # Últimos 5 commits
```

Formas de Hacer Squash

1. Squash en merge (GitHub/GitLab)

```
# En la interfaz web, botón "Squash and Merge"
```

2. Squash con rebase interactivo

```
git rebase -i HEAD~5 # Últimos 5 commits
```

3. Merge con squash

```
git merge --squash feature/login
git commit -m "feat:  implementar login"
```

Rebase Interactivo: El Editor

Al ejecutar `git rebase -i HEAD~5`, Git abre un editor:

Editor de rebase interactivo

```
pick a1b2c3d feat:  inicio login
pick b2c3d4e wip
pick c3d4e5f arreglo typo
pick d4e5f6g ahora sí funciona
pick e5f6g7h fix lint

# Comandos:
# p, pick = usar commit
# s, squash = usar commit, pero fusionar con el anterior
# f, fixup = como squash, pero descartar mensaje
# d, drop = eliminar commit
```

Rebase Interactivo: El Editor

Al ejecutar `git rebase -i HEAD~5`, Git abre un editor:

Editor de rebase interactivo

```
pick a1b2c3d feat:  inicio login
pick b2c3d4e wip
pick c3d4e5f arreglo typo
pick d4e5f6g ahora sí funciona
pick e5f6g7h fix lint

# Comandos:
# p, pick = usar commit
# s, squash = usar commit, pero fusionar con el anterior
# f, fixup = como squash, pero descartar mensaje
# d, drop = eliminar commit
```

Cambiar a:

```
pick a1b2c3d feat:  inicio login
squash b2c3d4e wip
squash c3d4e5f arreglo typo
squash d4e5f6g ahora sí funciona
squash e5f6g7h fix lint
```

5

Merge vs Rebase vs Squash

Tabla Comparativa

Aspecto	Merge	Rebase	Squash
Historial	Preservado	Reescrito	Simplificado
Commits nuevos	Merge commit	Nuevos hashes	Un commit
Linealidad	Ramificado	Lineal	Lineal
Conflictos	Una vez	Por cada commit	Una vez
Seguridad	Alto	Bajo	Medio
Trazabilidad	Completa	Perdida	Perdida

¿Cuál Usar?

Usa Merge

- Ramas públicas
- Cuando la historia importa
- En main/develop
- Con --no-ff en Git Flow

Usa Rebase

- Ramas locales
- Antes de PR
- Para historial limpio
- Actualizar feature

Usa Squash

- Features pequeñas
- Muchos commits WIP
- PR final a main
- Limpieza de historial

¿Cuál Usar?

Usa Merge

- Ramas públicas
- Cuando la historia importa
- En main/develop
- Con --no-ff en Git Flow

Usa Rebase

- Ramas locales
- Antes de PR
- Para historial limpio
- Actualizar feature

Usa Squash

- Features pequeñas
- Muchos commits WIP
- PR final a main
- Limpieza de historial

Estrategia común de equipos

`rebase` para actualizar feature branches + `squash merge` al fusionar PR = Historial lineal y limpio en main

6

Resolución de Conflictos

¿Qué es un Conflicto?

Definición

Un **conflicto** ocurre cuando Git no puede determinar automáticamente cómo combinar cambios en las mismas líneas de un archivo.

¿Qué es un Conflicto?

Definición

Un **conflicto** ocurre cuando Git no puede determinar automáticamente cómo combinar cambios en las mismas líneas de un archivo.

Git puede resolver automáticamente:

- Cambios en archivos diferentes
- Cambios en líneas diferentes del mismo archivo
- Agregar contenido nuevo sin solapar

¿Qué es un Conflicto?

Definición

Un **conflicto** ocurre cuando Git no puede determinar automáticamente cómo combinar cambios en las mismas líneas de un archivo.

Git puede resolver automáticamente:

- Cambios en archivos diferentes
- Cambios en líneas diferentes del mismo archivo
- Agregar contenido nuevo sin solapar

Git NO puede resolver:

- Cambios en las MISMAS líneas por diferentes personas
- Eliminación de un archivo que otro modificó
- Cambios que dependen lógicamente entre sí

Anatomía de un Conflicto

Así se ve un conflicto en el archivo

```
def calcular_precio(cantidad):
<<<<< HEAD
    return cantidad * 100 # Tu cambio
=======
    return cantidad * 150 # Cambio del otro
>>>>> feature/precios
```

Anatomía de un Conflicto

Así se ve un conflicto en el archivo

```
def calcular_precio(cantidad):
<<<<<< HEAD
    return cantidad * 100 # Tu cambio
=======
    return cantidad * 150 # Cambio del otro
>>>>>> feature/precios
```

Marcadores:

- `<<<<<< HEAD` — Inicio de tu versión
- `=======` — Separador
- `>>>>>>>` — Fin de la otra versión

Tu trabajo:

- Decidir qué versión conservar
- O combinar ambas manualmente
- Eliminar los marcadores

Resolviendo el Conflicto

1. Identificar archivos con conflictos

```
git status # Muestra "both modified"
```

Resolviendo el Conflicto

1. Identificar archivos con conflictos

```
git status # Muestra "both modified"
```

2. Editar el archivo y elegir la solución

Archivo resuelto

```
def calcular_precio(cantidad, premium=False):
precio_base = 100
if premium:
return cantidad * 150
return cantidad * precio_base
```

Resolviendo el Conflicto (cont.)

3 Marcar como resuelto y continuar

```
git add precio.py
git commit -m "fix:  resolver conflicto de precios"
```

Herramientas para Resolver Conflictos

VS Code

- Vista de 3 paneles
- Botones: Accept Incoming/Current/Both
- Integración con GitLens
- ¡Lo más fácil!

Terminal

```
# Usar herramienta visual
git mergetool

# Abortar merge
git merge --abort

# Ver versiones
git checkout --ours file
git checkout --theirs file
```

Herramientas para Resolver Conflictos

VS Code

- Vista de 3 paneles
- Botones: Accept Incoming/Current/Both
- Integración con GitLens
- ¡Lo más fácil!

Terminal

```
# Usar herramienta visual
git mergetool

# Abortar merge
git merge --abort

# Ver versiones
git checkout --ours file
git checkout --theirs file
```

Consejo

Configura VS Code como tu merge tool: `git config --global merge.tool vscode`

Ours vs Theirs

¡Cuidado!

El significado cambia según la operación

Durante MERGE

`--ours` = rama donde estás (main)
`--theirs` = rama que fusionas (feature)

```
git switch main
git merge feature
# ours = main
# theirs = feature
```

Ours vs Theirs

¡Cuidado!

El significado cambia según la operación

Durante MERGE

`--ours` = rama donde estás (main)
`--theirs` = rama que fusionas (feature)

```
git switch main
git merge feature
# ours = main
# theirs = feature
```

Durante REBASE

`--ours` = rama donde rebaseas (main)
`--theirs` = tus commits (feature)

```
git switch feature
git rebase main
# ours = main (!)
# theirs = feature (!)
```

Tips de Supervivencia

Antes del merge:

- `git stash` si tienes cambios sin commit
- `git pull` para tener todo actualizado
- Revisar `git log` de ambas ramas
- Hacer backup mental del estado

Tips de Supervivencia

Antes del merge:

- `git stash` si tienes cambios sin commit
- `git pull` para tener todo actualizado
- Revisar `git log` de ambas ramas
- Hacer backup mental del estado

Durante el conflicto:

- Mantener la calma
- Resolver un archivo a la vez
- Probar el código después
- Pedir ayuda si es necesario

Tips de Supervivencia

Antes del merge:

- `git stash` si tienes cambios sin commit
- `git pull` para tener todo actualizado
- Revisar `git log` de ambas ramas
- Hacer backup mental del estado

Durante el conflicto:

- Mantener la calma
- Resolver un archivo a la vez
- Probar el código después
- Pedir ayuda si es necesario

Comandos de emergencia

```
git merge --abort    # Cancelar merge en progreso
git reset --hard HEAD # Deshacer TODO (peligroso)
git reflog           # Ver historial completo
```

Lo que NO Debes Hacer

Las "Soluciones" que No Son Soluciones:

- × Borrar el repositorio y clonarlo de nuevo
- × Copiar los archivos a otra carpeta y empezar de cero
- × Hacer `git checkout --theirs .` sin revisar
- × Editar el archivo remoto directamente en GitHub
- × Crear una rama "backup" cada vez que hay conflicto
- × Ignorar los marcadores de conflicto y hacer push

Lo que NO Debes Hacer

Las "Soluciones" que No Son Soluciones:

- × Borrar el repositorio y clonarlo de nuevo
- × Copiar los archivos a otra carpeta y empezar de cero
- × Hacer `git checkout --theirs .` sin revisar
- × Editar el archivo remoto directamente en GitHub
- × Crear una rama "backup" cada vez que hay conflicto
- × Ignorar los marcadores de conflicto y hacer push

Todos hemos hecho al menos una de estas... pero hoy aprendemos a hacerlo bien.

7

Resumen y Próximos Pasos

Resumen de la Clase

Conceptos clave:

- **Merge:** Combina historiales, preserva contexto
- **Rebase:** Reescribe historial, mantiene linealidad
- **Squash:** Combina commits en uno
- **Conflictos:** Git necesita tu ayuda

Resumen de la Clase

Conceptos clave:

- **Merge:** Combina historiales, preserva contexto
- **Rebase:** Reescribe historial, mantiene linealidad
- **Squash:** Combina commits en uno
- **Conflictos:** Git necesita tu ayuda

Reglas importantes:

- Nunca rebase commits pusheados
- Merge para ramas públicas
- Resolver conflictos uno a uno
- `git reflog` es tu amigo

Comandos Clave

Merge

```
git merge feature
git merge --no-ff feature
git merge --squash feature
git merge --abort
```

Rebase

```
git rebase main
git rebase -i HEAD~5
git rebase --continue
git rebase --abort
```

Conflictos

```
git status
git diff
git add archivo
git checkout --ours file
git checkout --theirs file
```

Emergencias

```
git reflog
git reset --hard HASH
git stash
git stash pop
```

Próxima Clase: Flujos Remotos

Clase 6

Hasta ahora hemos trabajado localmente. Es hora de conectar con el mundo exterior: **GitHub y GitLab**.

En la Clase 6 aprenderemos:

- Autenticación SSH (clave pública/privada)
- Conceptos de `origin` y `upstream`
- Forks y contribución a proyectos ajenos
- Configurar repositorios remotos
- Flujos de trabajo y gestion de ramas y repositorio

¡Preparen sus cuentas de GitHub/GitLab!

> *The only way to learn Git is to use Git — especially when things go wrong.*
>
> — Scott Chacon, Pro Git

¡Gracias!

Sobrevivieron al Laboratorio de Caos