ESCUELA DE
INGENIERÍA EN CIENCIAS Y SISTEMAS
FACULTAD DE INGENIERÍA
UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

ECYS | DTT
DESARROLLO DE TRANSFERENCIA TECNOLÓGICA

Día, Fecha:	Jueves, 01/08/2024
Hora de inicio:	10:40 - 12:20

Introducción a la Programación y Computación 2 [P]

Denilson Florentín de León Aguilar

https://git-scm.com/book/en/v2

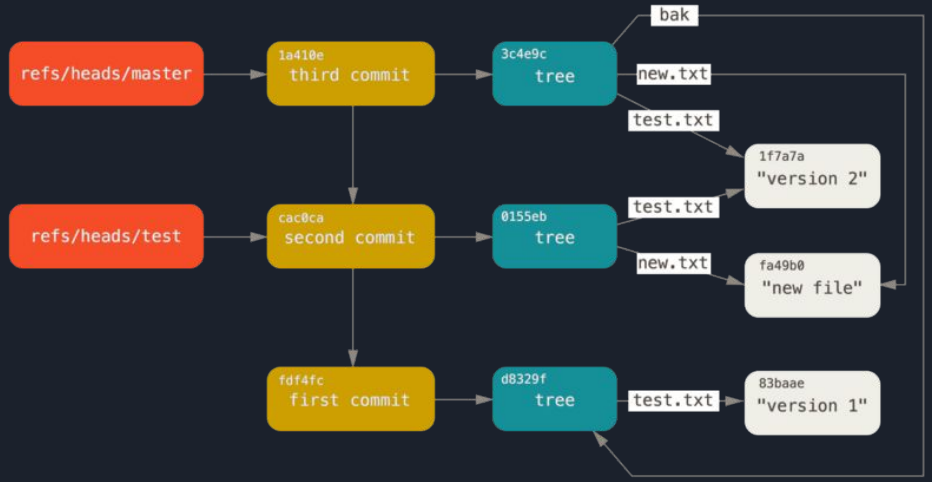
¿Qué es Git?

https://git-scm.com/book/en/v2

Git es un sistema de control de versiones distribuido utilizado para gestionar y rastrear cambios en el código fuente de un proyecto.

¿Para qué sirve?

Permite el seguimiento eficiente de cambios en el código, colaboración entre equipos de desarrollo, y la gestión de versiones para proyectos de software.



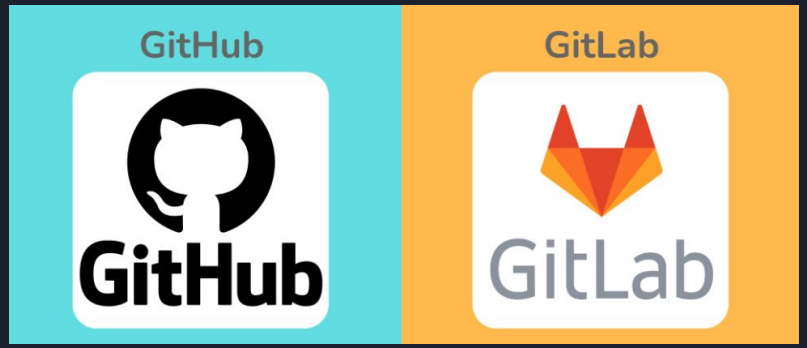
Plataformas de Git

GitHub - https://github.com/

Plataforma en línea para alojar proyectos Git de forma colaborativa. Facilita la colaboración y el seguimiento de proyectos de código abierto.

GitLab - https://gitlab.com/

Alternativa a GitHub con características adicionales, como herramientas de CI/CD (integración continua/despliegue continuo).

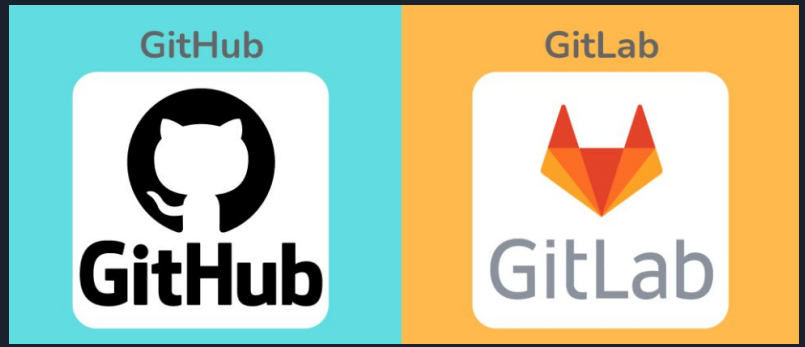


Configuración del Software a Usar

Descarga e instalación de Git:

https://git-scm.com/

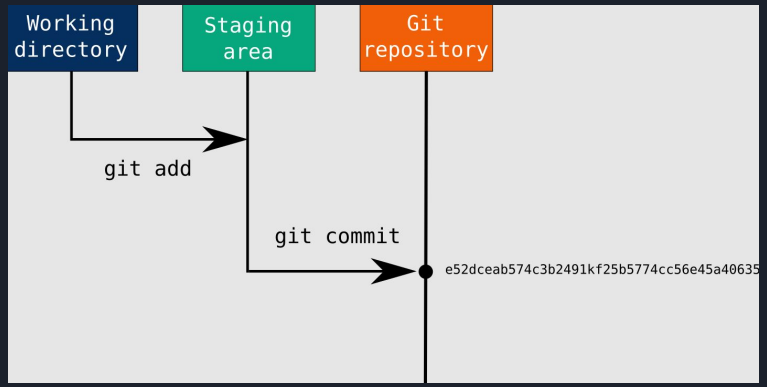
Configuración inicial del nombre de usuario y correo electrónico.



Conceptos y Fundamentos

Repositorio: Almacén que contiene todos los archivos y la historia de cambios del proyecto.

Commit: Un commit en Git es como una captura instantánea de los cambios realizados en tu proyecto en un momento específico. Cada commit tiene un mensaje que describe los cambios realizados, proporcionando una forma organizada de seguir la evolución del código a lo largo del tiempo. Los commits son la base para la gestión del historial y la colaboración en Git.



Conceptos y Fundamentos

Repositorio: Almacén que contiene todos los archivos y la historia de cambios del proyecto.

Commit: Un commit en Git es como una captura instantánea de los cambios realizados en tu proyecto en un momento específico. Cada commit tiene un mensaje que describe los cambios realizados, proporcionando una forma organizada de seguir la evolución del código a lo largo del tiempo. Los commits son la base para la gestión del historial y la colaboración en Git.

```
# Iniciar git en un carpeta
git init
```

```
# Clonar desde un repositorio remoto
git clone URL-del-repositorio
```

Implementación de Troncales y Ramas

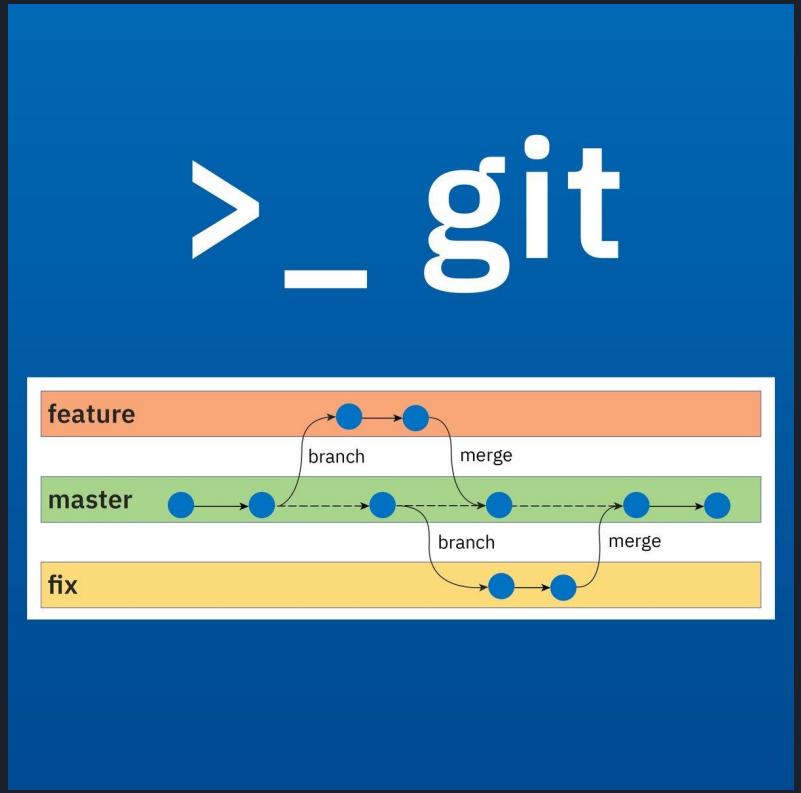
Denilson de León

Recordatorio: Captura de pantalla

Conceptos y Fundamentos

Branch (Rama)

Una rama (branch) en Git es una línea independiente de desarrollo que permite trabajar en nuevas características o correcciones sin afectar la rama principal (troncal). Cada rama tiene su propio conjunto de commits, lo que facilita la implementación de características aisladas o experimentos. Después de probar y completar el trabajo en una rama, puedes fusionarla con la rama principal para incorporar esos cambios.



Conceptos y Fundamentos

Branch (Rama)

`git branch`

- **Crear una Nueva Rama:**

```
git branch nombre-de-la-rama
```

- **Listar Todas las Ramas:**

```
git branch
```

- **Cambiar el Nombre de una Rama:**

```
git branch -m nombre-antiguo nombre-nuevo
```

- **Eliminar una Rama:**

```
git branch -d nombre-de-la-rama
```

Conceptos y Fundamentos

Branch (Rama)

- **Forzar la Eliminación de una Rama (sin fusionar cambios):**

```
git branch -D nombre-de-la-rama
```

- **Renombrar la Rama Actual:**

```
git branch -m nuevo-nombre
```

Conceptos y Fundamentos

Branch (Rama)

`git checkout`

- **Cambiar a una Rama Existente:**

```
git checkout nombre-de-la-rama
```

- **Crear y Cambiar a una Nueva Rama:**

```
git checkout -b nombre-de-la-rama
```

- **Cambiar a una Revisión o Commit Específico:**

```
git checkout commit-hash-o-nombre-de-revision
```

Conceptos y Fundamentos

Branch (Rama)

- **Desvincular Archivos (quitar cambios no deseados):**

```
git checkout -- nombre-del-archivo
```

- **Desvincular Todos los Archivos (descartar todos los cambios no comprometidos):**

```
git checkout -- .
```

- **Crear una Rama y Cambiar a Ella (forma más reciente y preferida):**

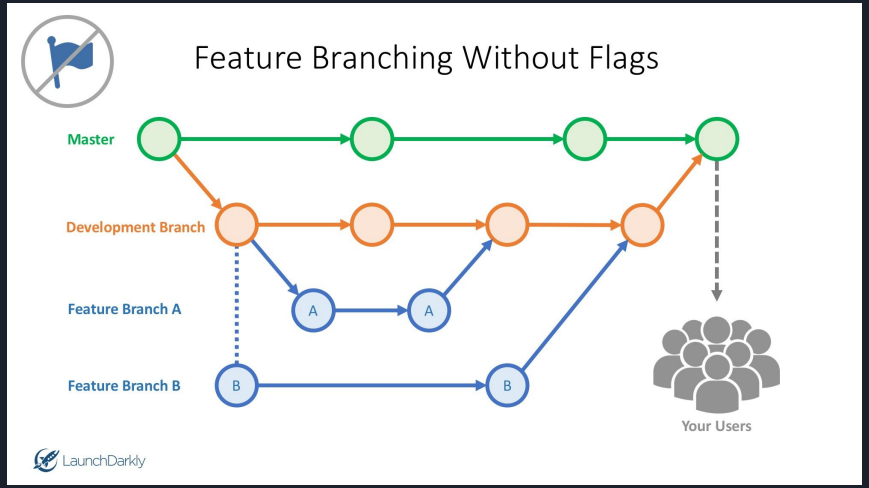
```
git switch -c nombre-de-la-rama
```

Conceptos y Fundamentos

Git Branching Workflow

La metodología de ramificación en Git es una práctica común para gestionar y organizar el desarrollo de software. Un flujo de trabajo popular sigue el estándar:

1. **main o master**
2. **develop**
3. **feature**
4. **fix**

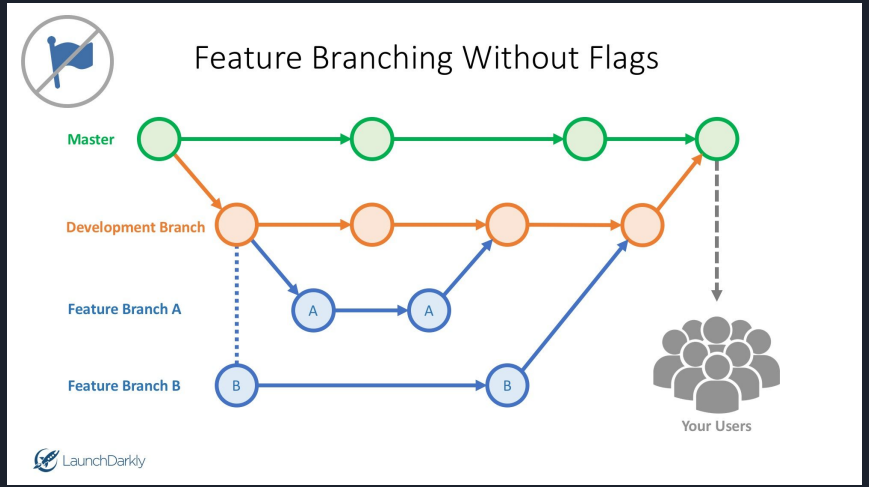


Conceptos y Fundamentos

Git Branching Workflow

`main` (o `master`):

- Representa la rama principal del proyecto.
- Contiene la versión estable y despliegable del software.
- Idealmente, esta rama siempre debería contener código que está listo para ser lanzado en producción.

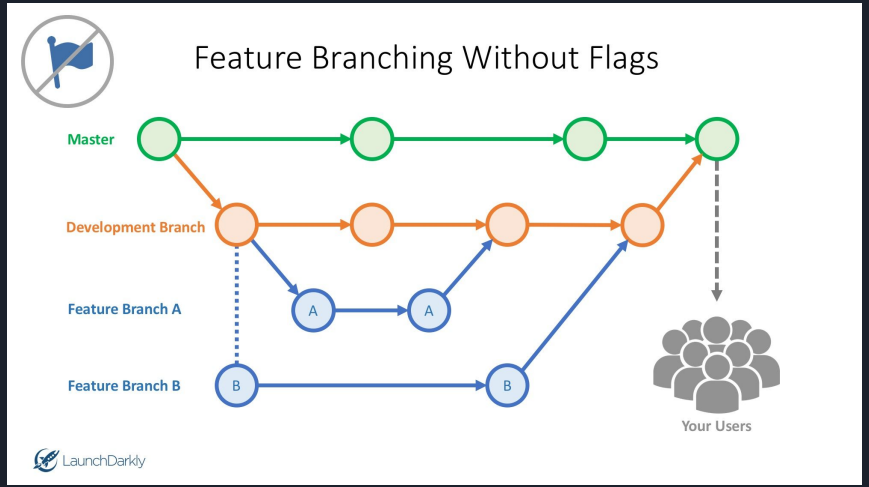


Conceptos y Fundamentos

Git Branching Workflow

`develop`:

- Es la rama de desarrollo principal.
- Se utiliza como punto de integración para todas las características completadas.
- Contiene el código más reciente, aunque no necesariamente listo para producción.

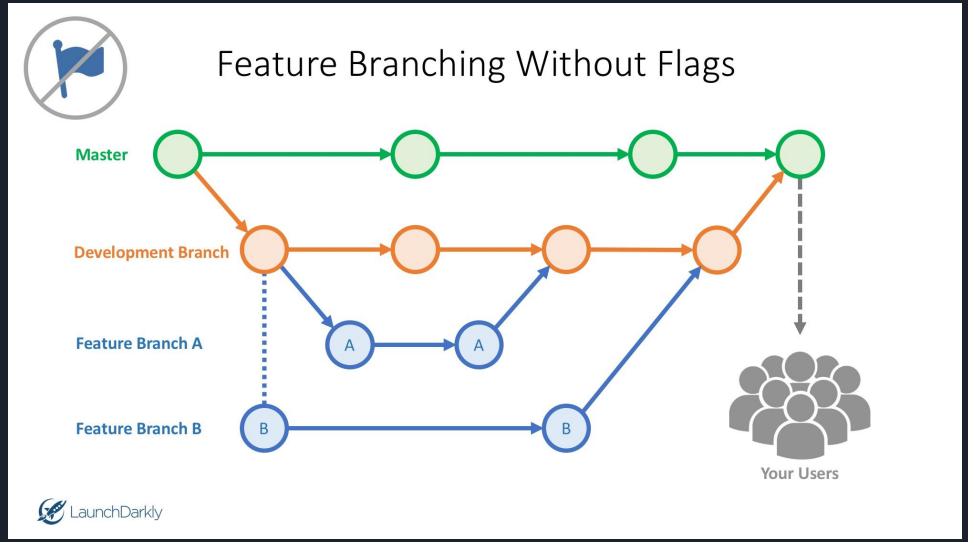


Conceptos y Fundamentos

Git Branching Workflow

`feature/nombre_feature`:

- Cada nueva característica o mejora se desarrolla en una rama separada.
- Se ramifica desde `develop` y se fusiona de nuevo en `develop` cuando la característica está completa.
- El nombre de la rama sigue el formato `feature/nombre_feature` para una fácil identificación.



Conceptos y Fundamentos

Git Branching Workflow

Ejemplo de Flujo de Trabajo:

1. Inicio de una Nueva Característica:
2. Desarrollo de la Característica
   a. Realiza cambios en la rama feature/nueva_caracteristica.
3. Finalización de la Característica:
4. Integración en develop:

```
git checkout develop
git pull origin develop
git checkout -b feature/nueva_caracteristica
```

```
git add .
git commit -m "Implementar nueva característica"
```

```
git status # comprobar que estamos en la rama
git checkout develop
git pull origin develop
git merge --no-ff feature/nueva_caracteristica
```

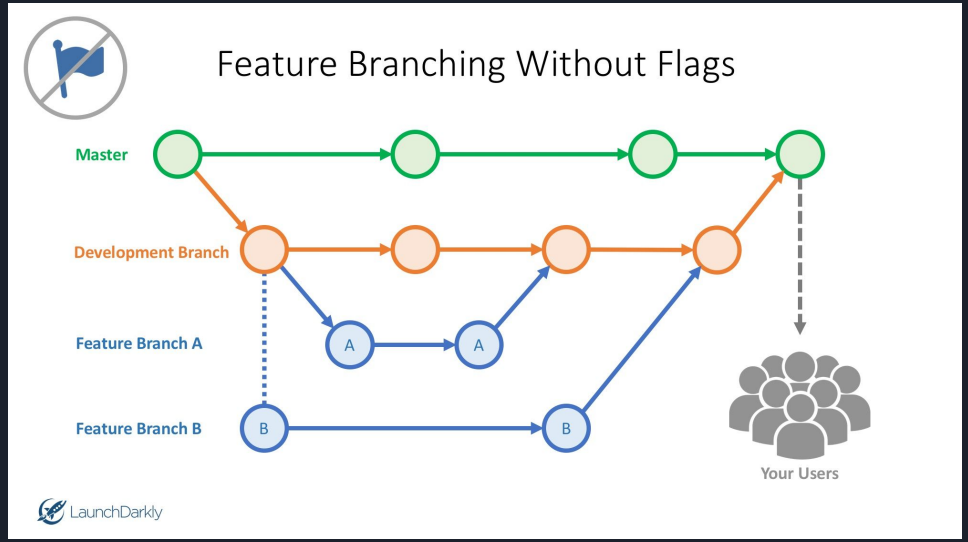
Conceptos y Fundamentos

Git Branching Workflow

Preparación para un Release:

Cuando todas las características para un lanzamiento están integradas en **`develop`**, se crea una nueva rama **`release`**.

Las correcciones de errores se realizan en la rama `release` antes de la fusión final en **`main`**.



Conceptos y Fundamentos

Git Branching Workflow

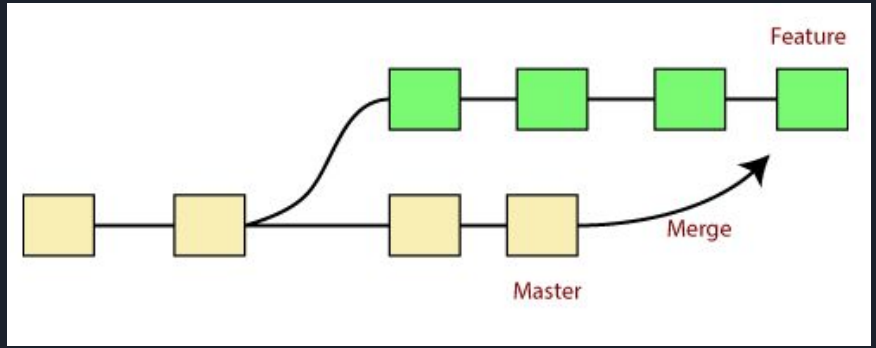
Ventajas del Modelo de Ramificación:

- **Seguimiento de Versiones:**
  - Facilita la gestión de versiones del software, ya que `main` siempre representa la última versión estable.
- **Desarrollo Paralelo:**
  - Permite que varios desarrolladores trabajen en características diferentes de manera simultánea sin interferencias.
- **Estabilidad:**
  - `main` siempre contiene código listo para producción, lo que asegura que los lanzamientos sean estables.
- **Historial Limpio:**
  - Mantiene un historial de cambios limpio y estructurado, con la posibilidad de identificar fácilmente cuándo se agregaron nuevas características.

Conceptos y Fundamentos

Merge (Fusión)

El proceso de merge (fusión) en Git implica combinar los cambios de una rama en otra. Por lo general, esto significa llevar los cambios de una rama secundaria a la rama principal. Git realiza automáticamente la fusión si no hay conflictos entre los cambios. El objetivo es consolidar el trabajo realizado en diferentes ramas en una única línea de desarrollo.

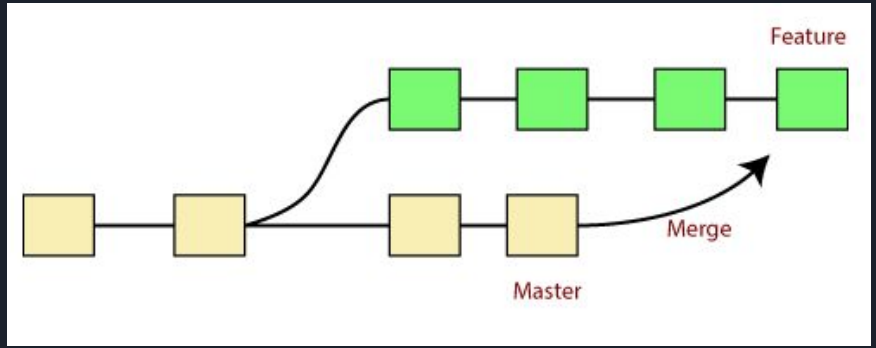



```
# Ejemplo de merge //en este ejemplo uniremos a maser/main
git checkout main
git merge feature-branch
```

Conceptos y Fundamentos

Fast-Forward (Avance Rápido)

En un fast-forward merge (fusión de avance rápido), Git puede realizar la fusión directa de una rama en otra sin crear un nuevo commit de fusión. Esto ocurre cuando todos los cambios en la rama secundaria que se va a fusionar están directamente accesibles desde la rama principal. Es una fusión sencilla y lineal, y el historial de commits sigue siendo claro.

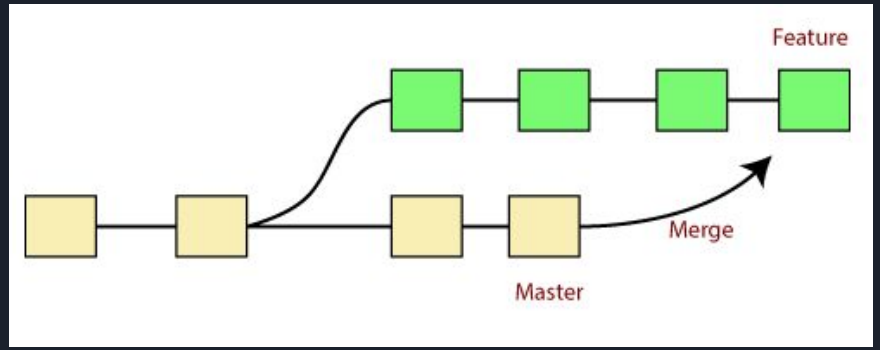



```
# Ejemplo de fast-forward merge
git checkout main
git merge feature-branch
```

Conceptos y Fundamentos

No Fast-Forward (No Avance Rápido)

En un no fast-forward merge (fusión sin avance rápido), Git crea un nuevo commit de fusión incluso si los cambios en la rama secundaria son accesibles directamente desde la rama principal. Esto se hace para conservar una referencia específica al punto en el tiempo en que se fusionaron las ramas, lo que puede ser útil para el seguimiento del historial y la auditoría.

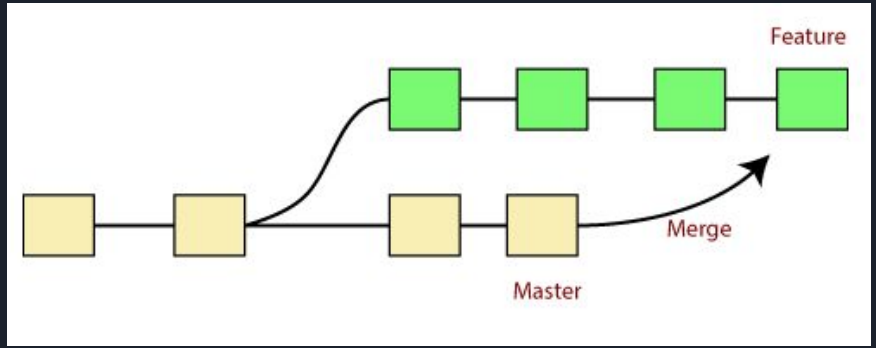



```
# Ejemplo de no fast-forward merge
git checkout main
git merge --no-ff feature-branch
```

Conceptos y Fundamentos

Rebase

La operación de rebase en Git reorganiza el historial de commits, moviendo la rama actual hacia adelante en el tiempo. A diferencia del merge, el rebase no crea un commit de fusión, sino que coloca los commits de la rama actual encima de la rama a la que se está rebasando. Esto puede dar como resultado una historia de commits más limpia y lineal, pero debes tener cuidado al reescribir el historial si otros




```
# Ejemplo de rebase
git checkout feature-branch
git rebase main
```

Comparación

Fast-Forward vs. No Fast-Forward en Merge:

Fast-Forward: Se utiliza cuando los cambios en la rama secundaria son directamente accesibles desde la rama principal.

No Fast-Forward: Se utiliza para conservar una referencia específica al punto en el tiempo de la fusión, incluso si los cambios son accesibles directamente.

Merge vs. Rebase:

Merge: Combina cambios de dos ramas y crea un nuevo commit de fusión. Conserva la historia original.

Rebase: Reorganiza la historia de commits, colocando los commits de la rama actual sobre la rama a la que se está rebasando.

Comandos

Comandos

`git status`

- **Descripción:**
  - Muestra los archivos modificados en el directorio de trabajo y los archivos preparados para el próximo commit (en el área de preparación).

`git add [file]`

- **Descripción:**
  - Agrega un archivo tal como está en el momento actual al área de preparación para el próximo commit.

`git reset [file]`

- **Descripción:**
  - Deshace el estado de preparación (unstaging) de un archivo, pero conserva los cambios en el directorio de trabajo.

`git diff`

- **Descripción:**
  - Muestra las diferencias entre lo que está modificado pero no está preparado (staged) y lo que está en el directorio de trabajo.

Comandos

`git diff --staged`

- **Descripción:**
  - Muestra las diferencias entre lo que está preparado (staged) pero no ha sido confirmado y la última confirmación.

`git commit -m "[descriptive message]"`

- **Descripción:**
  - Realiza un nuevo commit con los cambios que están preparados en el área de preparación.
  - El mensaje entre comillas proporciona una descripción clara y concisa del commit.

`git config --global user.name "[firstname lastname]"`

- **Descripción:**
  - Configura el nombre del usuario que será identificable en el historial de versiones.

Comandos

```
git config --global user.email "[valid-email]"
```

- **Descripción:**
  - Configura la dirección de correo electrónico que se asociará con cada marcador de historial.

```
git config --global color.ui auto
```

- **Descripción:**
  - Configura el color automático en la línea de comandos para una revisión fácil de Git.

```
git init
```

- **Descripción:**
  - Inicializa un directorio existente como un nuevo repositorio de Git.

Comandos

`git clone [url]`

- **Descripción:**
  - Clona un repositorio existente desde una ubicación hospedada mediante su URL.

`git branch`

- **Descripción:**
  - Lista todas las ramas presentes en el repositorio, marcando con un asterisco la rama activa.

`git branch [branch-name]`

- **Descripción:**
  - Crea una nueva rama en el commit actual.

Comandos

`git branch [branch-name]`

- **Descripción:**
  - Crea una nueva rama en el commit actual.

`git checkout`

- **Descripción:**
  - Cambia a otra rama y la trae al directorio de trabajo.

`git merge [branch]`

- **Descripción:**
  - Fusiona la historia de la rama especificada con la rama actual.

Comandos

`git push [alias] [branch]`

- **Descripción:**
  - Transmite los commits locales de la rama local actual al repositorio remoto identificado por `[alias]` y la rama remota `[branch]`.
  - Actualiza la rama remota con los cambios locales.

`git pull`

- **Descripción:**
  - Realiza dos acciones en una:
    1. Recupera cambios del repositorio remoto (equivalente a `git fetch`).
    2. Fusiona los cambios recuperados en la rama local actual (equivalente a `git merge`).

Comandos

`git log`

- **Descripción:**
  - Muestra todos los commits en la historia de la rama actual.

`git remote add [alias] [url]`

- **Descripción:**
  - Añade un alias (`[alias]`) para una URL de un repositorio remoto (`[url]`).
  - Facilita el acceso al repositorio remoto utilizando un nombre más corto y legible.

`git fetch [alias]`

- **Descripción:**
  - Recupera toda la información del repositorio remoto identificado por `[alias]` sin realizar cambios locales.
  - Descarga los commits, ramas y objetos asociados, actualizando la información local.

Comandos

Tracking Path Changes:

`git rm [file]`

- **Descripción:**
  - Elimina un archivo del proyecto y coloca la eliminación en el área de preparación para el próximo commit.

`git mv [existing-path] [new-path]`

- **Descripción:**
  - Cambia la ubicación de un archivo en el proyecto y prepara el cambio en el área de preparación.
  - Útil para gestionar cambios de ruta de archivos.

`git log --stat -M`

- **Descripción:**
  - Muestra todos los registros de commits, indicando cualquier cambio de ruta (`-M`).
  - Proporciona una visión general de la historia del proyecto, incluyendo movimientos de archivos.

Comandos

Temporary Commits:

`git stash`

- **Descripción:**
  - Guarda temporalmente las modificaciones y archivos preparados sin realizar un commit.
  - Permite cambiar de rama o realizar otras operaciones sin comprometer los cambios.

`git stash list`

- **Descripción:**
  - Muestra la pila de cambios almacenados temporalmente, indicando la orden de apilamiento.

`git stash pop`

- **Descripción:**
  - Aplica y elimina el último cambio almacenado temporalmente de la pila.
  - Útil para retomar los cambios guardados.

`git stash drop`

- **Descripción:**
  - Descarta el último cambio almacenado temporalmente de la pila sin aplicarlo.
  - Elimina el cambio de la pila.

Comandos

Git Reset Hard:

```
git reset --hard [commit]
```

- **Descripción:**
  - Borra el área de preparación y reescribe el árbol de trabajo a partir de un commit específico.
  - Útil para deshacer cambios y retroceder a un estado anterior del proyecto.

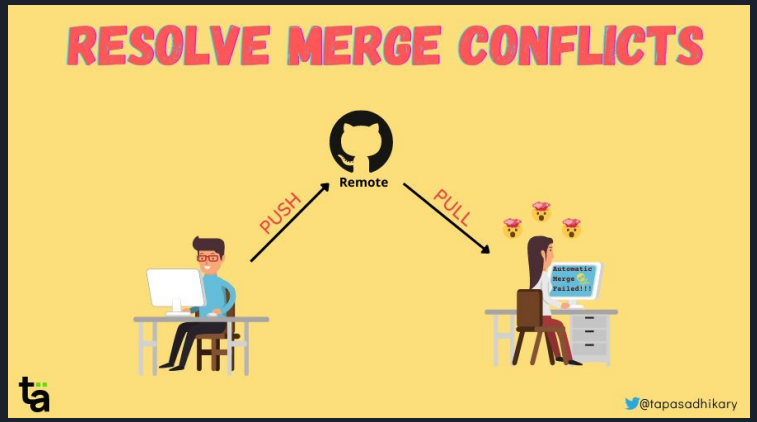
Conflictos

Conflictos

Definición

Un conflicto en Git ocurre cuando dos ramas tienen cambios conflictivos en la misma parte de un archivo. Por ejemplo, si ambas ramas modifican las mismas líneas de código, Git no puede determinar automáticamente cuál es la versión correcta. En este caso, se produce un conflicto y se requiere intervención manual para resolverlo. El manejo de conflictos es crucial durante el proceso de fusión para garantizar la coherencia del código.

Para arreglarlo hay que usar git pull **SIEMPRE** a la rama donde se quiera hacer merge



Conflictos

Cuando se trabaja en equipo, puede haber conflictos cuando dos o más ramas modifican el mismo archivo en líneas cercanas. Para resolver un conflicto:

1. **Obtener el Estado Actual:**

   - Al intentar fusionar o hacer un pull, se notificará un conflicto. El estado del conflicto se verá así:

```
<<<<<<< HEAD
Código en la rama actual
=======
Código en la rama que se está fusionando
>>>>>>> [branch-name]
```

Conflictos

La resolución de conflictos es una parte esencial del trabajo colaborativo con Git. Es crucial comunicarse con el equipo para entender los cambios propuestos y asegurarse de que la fusión sea coherente con la lógica del proyecto.

2. **Resolución Manual:**

   - Edita el archivo para conservar las líneas que deseas mantener.
   - Elimina las marcas `<<<<<<< HEAD`, `=======`, y `>>>>>>> [branch-name]`.
   - El código resultante después de la resolución manual será el que se fusionará.

3. **Agregar y Confirmar:**

```
git add [archivo]
git commit -m "Resolver conflicto"
```

4. **Finalizar la Fusión o Pull:**

   - Continuar con el proceso de fusión o pull.

Git Ignore

Git ignore

`git config --global core.excludesfile [file]`

- **Descripción:**
  - Configura un archivo global de patrones de ignorados para todos los repositorios locales.
  - Los patrones especificados en este archivo evitarán que ciertos archivos o directorios sean rastreados por Git.

`.gitignore`

- **Descripción:**
  - Este archivo se utiliza para especificar patrones de nombres de archivos o directorios que Git debería ignorar.
  - Los patrones pueden incluir nombres específicos, comodines y reglas de exclusión.

Git ignore

Este archivo se aplicará al directorio donde se coloca, esto incluye todas las subcarpetas, además es posible tener más de un archivo .gitignore

Ejemplo de `.gitignore`:

```
# Ignorar archivos de compilación
*.class
*.o

# Ignorar archivos de respaldo de texto
*~

# Ignorar directorios de construcción
/build/

# Ignorar archivos sensibles
secreto.txt
```

Git ignore

Cuando trabajas en proyectos Python, hay ciertos archivos y directorios comunes que a menudo quieres ignorar para evitar problemas de versionamiento y mantener el repositorio limpio. Aquí hay algunos ejemplos de patrones comunes que puedes agregar a tu archivo .gitignore en proyectos Python:

Git ignore

```
# Archivos y directorios generados por el entorno virtual de Python
__pycache__/
venv/
env/

# Archivos de configuración locales
*.pyc
*.pyo
*.pyd
*.db
*.sqlite3
*.log
*.csv
*.xlsx
```

Documentación

https://education.github.com/git-cheat-sheet-education.pdf

https://git-scm.com/doc