GIT IPC2

`git init`

El comando `git init` se utiliza para iniciar un nuevo repositorio de Git. Este comando crea un nuevo directorio oculto llamado `.git` en el directorio actual y configura todos los archivos y subdirectorios necesarios para comenzar a realizar un seguimiento de versiones con Git.

```
git init
```

- **Uso:**
  - Este comando se ejecuta en el directorio del proyecto que deseas convertir en un repositorio de Git.
  - Después de ejecutar `git init`, el directorio se convierte en un repositorio de Git local.

`git clone`

El comando `git clone` se utiliza para crear una copia exacta de un repositorio Git existente. Puedes clonar un repositorio desde un servidor remoto (como GitHub) o desde otro directorio local.

```
git clone URL-del-repositorio
```

- **Uso:**
  - `URL-del-repositorio` es la dirección del repositorio que deseas clonar.
  - Este comando descarga todos los archivos del repositorio y crea una copia local en tu máquina.

Iniciar un Nuevo Repositorio con `git init`:

1. **Navega al Directorio del Proyecto:**
   - Abre una terminal y navega al directorio del proyecto que deseas convertir en un repositorio de Git.
2. **Ejecuta `git init`:**
   - Ejecuta el siguiente comando en el directorio del proyecto:

     ```
     git init
     ```

3. **Repositorio Inicializado:**
   - Ahora tu directorio local es un repositorio de Git. Puedes comenzar a realizar un seguimiento de versiones de tus archivos.

Clonar un Repositorio Existente con `git clone`:

1. **Ejecuta `git clone`:**
   - Abre una terminal y ejecuta el siguiente comando para clonar un repositorio existente:

     ```
     git clone URL-del-repositorio
     ```

- Sustituye `URL-del-repositorio` con la dirección del repositorio que deseas clonar.

2. **Copia Local Creada:**
    - Git descargará todos los archivos del repositorio y creará una copia local en tu máquina.
3. **Repositorio Clonado:**
    - Ahora tienes una copia del repositorio en tu máquina y puedes comenzar a trabajar en él.

Estos comandos son esenciales para iniciar y colaborar en proyectos utilizando Git. `git init` es usado para iniciar un nuevo repositorio local, mientras que `git clone` se usa para obtener una copia de un repositorio existente, ya sea en un servidor remoto o en otro directorio local.

Comandos generales de GIT

`git add .`

El comando `git add .` se utiliza para agregar todos los cambios realizados en los archivos del directorio de trabajo al área de preparación (staging area) en Git. Este comando selecciona todos los archivos modificados, eliminados o creados y los prepara para ser incluidos en el próximo commit.

```
git add .
```

- **Uso:**
    - Este comando selecciona automáticamente todos los archivos y cambios en el directorio de trabajo para ser incluidos en el área de preparación.

`git commit`

El comando `git commit` se utiliza para registrar los cambios preparados en el área de preparación en un nuevo commit en la historia del repositorio.

```
git commit
```

- **Uso:**
    - Este comando abre un editor de texto predeterminado para que puedas ingresar una descripción detallada de los cambios realizados.
    - Después de ingresar la descripción, guardas y cierras el editor, lo que crea un nuevo commit con los cambios y la descripción proporcionada.

`git commit -m "descripción"`

El comando `git commit -m "descripción"` es una forma más rápida de realizar un commit sin necesidad de abrir un editor de texto. Puedes incluir directamente la descripción del commit en la línea de comandos.

```
git commit -m "Agregado nuevo feature"
```

- **Uso:**
  - Esta opción te permite agregar una descripción en línea, lo que es útil para cambios pequeños y rápidos.

Recomendaciones para Descripciones de Commit:

1. **Sé Descriptivo:**
   - Proporciona una descripción clara y detallada de los cambios realizados en el commit.
2. **Usa Presente y Afirmativo:**
   - Utiliza oraciones en tiempo presente y afirmativas para describir qué hace el commit.
3. **Limítate a un Tema o Funcionalidad:**
   - Cada commit debe abordar un solo tema o funcionalidad. Evita incluir cambios no relacionados en el mismo commit.
4. **Mantén la Longitud Razonable:**
   - Intenta mantener las descripciones concisas y enfocadas. Evita descripciones excesivamente largas.
5. **Explica el "por qué":**
   - Además de explicar qué cambios se realizaron, proporciona contexto sobre por qué se realizaron esos cambios.

Siguiendo estas recomendaciones, se logra una historia de commits clara y fácil de entender, lo que facilita la colaboración y la revisión de código por parte de otros desarrolladores.

`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
  ```

- **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
```

`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
```

- **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
```

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`):**
    - 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.
2. **`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.
3. **`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.

Ejemplo de Flujo de Trabajo:

1. **Inicio de una Nueva Característica:**

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

2. **Desarrollo de la Característica:**

   - Realiza cambios en la rama `feature/nueva_caracteristica`.

3. **Finalización de la Característica:**

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

4. **Integración en `develop`:**

   ```
   git checkout develop
   git pull origin develop
   git merge --no-ff feature/nueva_caracteristica
   ```

5. **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`.

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.

Adoptar un modelo de ramificación estructurado mejora la colaboración entre equipos y facilita la gestión de versiones en proyectos de desarrollo de software.

`git merge`

- **Fusionar una Rama en Otra:**

```
git merge nombre-de-la-rama
```

- **Fusionar y Permitir Fast-Forward (si es posible):**

```
git merge --ff nombre-de-la-rama
```

`git rebase`

- **Reorganizar la Historia de Commits (Rebase):**

```
git rebase nombre-de-la-rama
```

Documentación de Git para obtener más detalles sobre estas y otras opciones: Documentación de Git Branch y Documentación de Git Checkout.

`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.

`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.

`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.

`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.

`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.

`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.

`git merge [alias]/[branch]`

- **Descripción:**
  - Fusiona la rama remota especificada (`[branch]`) del repositorio remoto identificado por `[alias]` en la rama local actual.
  - Asegura que la rama local esté actualizada con los cambios de la rama remota.

`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`).

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.

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.

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.

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.

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
```

Resolución de 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]
      ```

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.

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.