USUARIO A

EJERCICIO 1: INVITAR AL COMPAÑERO/A (en pareja)

Como usuario A, te toca ser el que comparta tu repositorio de GitHub entre los dos. Tienes que invitar a tu compañero B para que podáis trabajar ambos en el mismo repositorio.

1.1) Añade a tu compañero como colaborador en tu repositorio de prácticas -> *Settings* -> *Collaborators* -> *Add people* -> *Manage access* -> *Add people* -> Añade el usuario de GitHub de tu compañero/a B e invítalo/a.

1.2) Haz clone de tu proyecto en tu equipo.

Solución: -> `git clone git@github.com:...`

1.3) Comprueba que se ha clonado con el método SSH.

Solución: -> `git remote -v`

Espera al compañero/a B antes de continuar

EJERCICIO 2: TRABAJANDO CON RAMAS (en pareja)

2.1) Crea una issue de nombre “Testing file (A)” con la descripción que quieras. Debes añadir el nombre de usuario de tu compañero/a B en “Assigness” en la parte de la derecha. Como propietario del repositorio, tendrás que activar la opción de issues en *Settings* -> *General* -> *Features* -> *Issues*

Espera al compañero/a B antes de continuar

2.2) Crea una rama local de nombre “feature/practicandogit_a” y sitúate en ella.

Solución -> `git checkout -b feature/practicandogit_a`

2.3) Comprueba que estás situado en esa rama.

Solución -> `git branch`

2.4) Súbela al repositorio remoto.

Solución -> `git push -u origin feature/practicandogit_a`

2.5) Crea, en la raíz del proyecto, un archivo llamado “practicandogit_a.txt” que contenga una sola línea con tu uvus.

Solución -> `echo "tu_uvus" > practicandogit_a.txt`

2.6) Comprueba que ese archivo lo está rastreando git.

Solución -> `git status`

2.7) Añade ese archivo en concreto al área de preparación (stage).

Solución -> `git add practicandogit_a.txt`

2.8) Comprueba que ese archivo lo está rastreando git y ya está en el área de preparación

Solución -> `git status`

2.9) Crea un commit de nombre "feat: Añade archivo de prueba" pero no lo subas aún al repositorio remoto.

Solución -> `git commit -m "feat: Añade archivo de prueba"`

2.10) ¡Ay! Que nos hemos equivocado, que todo debería ir en inglés. Cambia, entonces, el título del commit por "feat: Add testing file" y cerrando la issue de tu compañero "Testing file (B)" mediante su ID

Solución ->
`git commit --amend -m "feat: Add testing file. Closes #<ID>"`

2.11) Sube el nuevo commit a tu rama remota.

Solución -> `git push`

Espera al compañero/a B antes de continuar

EJERCICIO 3: PULL REQUEST (en pareja)

Vamos a hacer una petición de cambio que tendrá que aceptar nuestro compañero B.

3.1) Ve a la pestaña *Pull requests* -> *New pull request*. Asegúrate, a la hora de comparar, que en "*base*" aparece "*main*" y que en "*compare*" aparece "*feature/practicandogit_a*"

3.2) Crea una pull request e invita al compañero B a que la revise. Debes añadir su nombre de usuario en "Assigness" en la parte de la derecha.

Espera al compañero/a B antes de continuar

3.3) En la pestaña *Pull requests* debería haber una petición de cambio pendiente que tiene que ser aceptada por nosotros. Acéptala con *Merge pull request -> Confirm merge.* **No borres la rama, nos seguirá haciendo falta.**

Espera al compañero/a B antes de continuar

EJERCICIO 4: CONFLICTOS (en pareja)

Vamos a trabajar con conflictos, algo habitual en el desarrollo de proyectos.

4.1) Sitúate en la rama principal

Solución: > `git checkout main`

4.2) Actualiza todos los cambios del repositorio remoto en tu repositorio local

Solución -> `git fetch`

4.3) Tráete a tu repositorio local la rama "feature/practicandogit_b"

Solución -> `git checkout feature/practicandogit_b`

4.4) Actualiza los posibles cambios de esa rama.

Solución -> `git pull origin feature/practicandogit_b`

Nota: obviamente, no habrá ningún cambio nuevo, dado que la rama "main" está perfectamente sincronizada con el resto de ramas. Pero esto no tiene por qué ser así siempre. Siempre que cambiemos a una rama local, debemos sincronizarla con la rama remota.

4.5) Modifica la primera y única línea del archivo "practicandogit_b.txt" con "*esta es una nueva línea remota*".

4.6) Sube los cambios a la rama remota "feature/practicandogit_b" con el mensaje de commit "fix: Change testing file"

Solución ->

```
git add practicandogit_b.txt
git commit -m "fix: Change testing file"
git push
```

¡MUY IMPORTANTE! Espera al compañero/a B antes de continuar

4.7) Sitúate en tu rama "feature/practicandogit_a"

Solución -> `git checkout feature/practicandogit_a`

4.8) Modifica la primera y única línea del archivo "practicandogit_a.txt" con "*esta es una nueva linea local*". Añádelo al área de stage y commitéalo, pero NO LO SUBAS AL REMOTO.

Solución ->
```
git add practicandogit_a.txt
git commit -m "feat: Whatever"
```

4.9) Actualiza tu rama local

Solución -> `git pull origin feature/practicandogit_a`

¡Vaya! Ha debido mostrarse el mensaje "*Las ramas se han divergido y hay que especificar cómo reconciliarlas".* Ambas han recibido nuevos commits que no existen en la otra rama. Cuando esto sucede, Git necesita saber cómo quieres combinar los cambios, ya que no puede realizar la operación de forma automática sin que indiques cómo proceder.

4.10) Intenta realizar la fusión de la rama remota con la rama local sin hacer rebase.

Solución ->
```
git pull --no-rebase origin feature/practicandogit_a
```

4.11) ¡Hay un conflicto! Claro, la fusión no es posible porque se está superponiendo la misma línea en el mismo archivo (practicandogit_a.txt). Git no sabe qué cambio es el válido, si el tuyo en local (*esta es una nueva linea local*) o el remoto (*esta es una nueva línea remota*)

Un archivo con conflicto se ve así:

```
<<<<<<< HEAD
// Esta es tu versión (la versión local) del código.
código que tú modificaste
=======
// Esta es la versión del código en la otra rama (la
versión remota).
código que alguien más modificó (compañero B)
>>>>>>> nombre-de-la-rama-conflictiva
```

Arreglar el conflicto no es más que decidir con qué cambio me quedo, si con el local o con el remoto. Quédate con el cambio remoto y sube el cambio con el mensaje "fix: Fix conflicts"

Solución ->

(luego de haber editado el archivo)

```
git add practicandogit_a.txt
git commit -m "fix: Fix conflicts"
git push
```

Espera al compañero/a B antes de continuar

EJERCICIO 5: CHERRY-PICKING (en pareja)

Un escenario común en el desarrollo es cuando un desarrollador, trabajando en su propia rama, encuentra y soluciona un fallo crítico que necesita ser corregido de inmediato. Sin embargo, la rama en la que está trabajando no está lista para ser fusionada (*merge*) con la rama principal (por ejemplo, main o develop) porque aún tiene cambios pendientes o no ha sido revisada completamente. En este caso, **ese commit específico** que soluciona el fallo es el único que debe estar disponible en todas las ramas, sin necesidad de fusionar toda la rama incompleta. Aquí es donde entra en juego el concepto de **cherry-picking**. **Cherry-picking** en Git es el proceso de tomar un commit específico de una rama y aplicarlo en otra sin fusionar el resto del historial de esa rama. Este proceso te permite "elegir" (como si cogieras una cereza de un árbol, de ahí el nombre) un commit en particular y aplicarlo en otro lugar, sin traer todos los commits de la rama original.

5.1) Sitúate en la rama “feature/practicandogit_a”

Solución -> `git checkout feature/practicandogit_a`

5.2) Crea el archivo “fix_bug_A.py” con el contenido que quieras

Solución -> `echo "" > fix_bug_A.py`

5.3) Realiza el commit “fix: Fix important bug (developer A)” con el archivo anterior.

Solución ->

```
git add fix_bug_A.py
git commit -m "fix: Fix important bug (developer A)"
git push
```

Espera al compañero/a B antes de continuar

5.4) Actualiza los cambios remotos y obtén el hash del commit de tu compañero, es decir, “fix: Fix important bug (developer B)”

Solución ->

```
git fetch
git log –oneline --all (hash del compañero/a B)
```

5.5) Tráete ese commit a tu rama

Solución -> `git cherry-pick (hash)`

5.6) Sube los cambios

Solución -> `git push`

EJERCICIO 6: STASHING (individual)

Una situación común en el desarrollo es la siguiente: estás trabajando en tu rama local y realizando cambios que aún no están listos para ser confirmados (commits), pero de repente necesitas cambiar de rama para revisar o trabajar en algo más, y no puedes simplemente "saltarte" a la otra rama porque perderías los cambios actuales. Aquí es donde entra en juego la **pila stash** de Git.

El *stash* en Git es como un "cajón temporal" donde puedes guardar tus cambios no confirmados (modificaciones y archivos nuevos o cambiados) sin tener que hacer un commit. Esto te permite limpiar tu área de trabajo (working directory) momentáneamente, cambiar de rama o hacer cualquier otra operación, y luego **recuperar esos cambios** cuando lo necesites. El *stash* funciona como una **pila (stack)**, es decir, puedes hacer varios "stash" de cambios y almacenarlos de manera secuencial. Luego, cuando los necesites, puedes recuperar el último *stash* o cualquiera de los que hayas guardado previamente.

6.1) Asegúrate que estás en la rama “feature/practicandogit_a”

Solución -> `git branch`

6.2) Crea, en la raíz, un archivo de nombre “estoestasinterminar.txt” con el contenido que quieras

Solución ->
```
echo "blablabla" > estoestasinterminar.txt
```

6.3) Tu jefe te pide que revises algo urgente en la rama “main” pero no puedes subir ese archivo sin terminar ni tampoco descartarlo. Usa la pila stash. Pero, ¡cuidado! **La pila stash solo tiene en cuenta los archivos en seguimiento**, y “estoestasinterminar.txt” no lo esta (a o no ser que se use el flag “-u” de untracked)

Solución ->
```
git add estoestasinterminar.txt
git stash
```

6.4) Comprueba que puedes saltar a la rama “main” sin problemas. Vuelve a la rama “feature/practicandogit_a”

Solución ->
```
git checkout main
git checkout feature/practicandogit_a
```

6.5) Muestra el contenido de la pila stash

Solución -> `git stash list`

Cada entrada tiene un número (stash@{0}, stash@{1}, etc.), lo que te permite referenciar un *stash* específico.

6.6) Aplica el stash guardado sin eliminarlo de la pila

Solución -> `git stash apply`

6.7) El último stash sigue en la pila. Ya no nos hace falta. Borra la pila stash.

Solución -> `git stash drop`

EJERCICIO 7: REVIRTIENDO CAMBIOS (individual)

Resulta que ya no nos hace falta el archivo "estoestasinterminar.txt". Además, supón que hemos hecho unos cuantos cambios intentando arreglar el sistema por un bug y lo hemos empeorado. ¡Queremos volver al instante cuando todo iba bien!

¡Saquemos al Delorean del garaje!

7.1) Localiza el hash correspondiente al commit "fix: Fix missing MariaDB port in Docker entrypoints"

Solución -> `git log --oneline (de1fffd)`

7.2) Has realizado cambios locales en tu rama, pero necesitas deshacer todos los commits realizados después de un commit específico, manteniendo los cambios en el área de trabajo para seguir editando. Consulta también el estado del stage.

Solución ->
```
git reset --soft de1fffd
git status
```

7.3) El problema es que, despues del commit *de1fffd* hubo varios otros, y cada uno aplicó X cambios. La opción "--soft" deja todos los cambios que habías hecho después de ese commit en el área de preparación (stage). Elimina también esos cambios del stage.

Solución -> `git reset --hard de1fffd`

7.4) Haz de nuevo "git status". What? Tu rama remota "feature/practicandogit_a" tiene varios commits adicionales que no están en tu copia local. Estos commits no se ven afectados por el "reset --hard" que has hecho localmente. Es decir, aunque resetees tu rama local al commit *de1fffd*, el repositorio remoto sigue teniendo esos commits adicionales. Arréglalo actualizando la rama.

Solución -> `git pull`

7.5) ¡Esto parece un sinsentido! En mi repositorio local vuelvo a un estado anterior, pero Git siempre detecta que estamos varios commits por detrás. ¿Qué pasa si verdaderamente quiero volver a un estado anterior? Que tengo que forzar un *push* a mi rama remota para eliminar los posteriores commits adicionales.

Solución ->
```
git reset --hard de1fffd
git push origin feature/practicandogit_a --force
```

Es importante entender que "git reset" afecta al repositorio local, no al remoto. Si el remoto tiene commits adicionales, estos siguen ahí después del reset. Entonces Git te pide hacer un pull porque tu local está "detrás" del remoto. Si quieres que el remoto también vuelva a un commit anterior, debes hacer un git push --force.

7.6) Deshaz los cambios del commit "fix: Change Vagrant box to jammy64"

Solución ->
```
git revert 44d5fe6
git push
```

7.7) Y, ya que estamos, compara el commit de HEAD con ese commit, para ver qué cambios acabamos de deshacer (**rojo = lo que había hecho, verde = lo que hemos deshecho**)

Solución -> `git diff HEAD 44d5fe6`

EJERCICIO 8: HOOKS (individual)

Los **hooks de Git** son **scripts que Git ejecuta automáticamente** en determinados momentos del ciclo de vida del repositorio. Los hooks permiten personalizar y automatizar tareas en un proyecto al interactuar directamente con ciertos eventos de Git, como la creación de commits, la fusión de ramas, el envío de cambios a un repositorio remoto, y más. No obstante, los hooks de Git no forman parte del repositorio en sí, es decir, no se suben ni se descargan automáticamente junto con el código del repositorio.

8.1) Crea en tu rama "feature/practicandogit_a", en .git/hooks/, un archivo "commit-msg"

Solución ->
```
cd .git/hooks
touch commit-msg
```

8.2) Vamos a crear un script (hook) que compruebe que el mensaje de un commit sigue unas normas. Si no las sigue, no nos dejará continuar. Es un pre-commit.

Añade el siguiente contenido a commit-msg:

```
#!/bin/bash

# Leemos el mensaje del commit desde el archivo temporal
que Git genera
mensaje_commit=$(cat "$1")

# Verificamos si el mensaje de commit cumple con el formato
if ! [[ "$mensaje_commit" =~ ^(feat|fix|chore): ]]; then
  echo "ERROR: El mensaje del commit debe comenzar con uno
de los siguientes prefijos: feat:, fix:, chore:"
  exit 1  # Evitar el commit si el formato es incorrecto
fi
```

8.3) En la raíz, crea un archivo "testing_hook.txt" con el contenido que tu quieras, añádelo a tu área de preparación y crea un commit con el mensaje "Esto es un mensaje inválido".

Solución ->
```
echo "blablabla" > testing_hook.txt
git add testing_hook.txt
git commit -m "Esto es un mensaje inválido"
```

8.4) Vaya, no hemos podido. Claro, es que no sigue nuestras normas. Haz un commit que sea "feat: Esto sí un mensaje válido"

Solución -> `git commit -m "feat: Esto sí un mensaje válido"`

EJERCICIO 9: GESTIÓN AVANZADA DE CAMBIOS (individual)

El proceso de *commit* en Git es fundamental. Sin embargo, a menudo realizamos múltiples cambios en varios archivos antes de hacer git add . y luego git commit. Las herramientas que vamos a practicar ahora (git add -p, git diff, y el manejo de parches) nos otorgan un control de granularidad excepcional. Esto nos permite separar y enviar solo partes específicas de un archivo modificado, creando *commits* más limpios, enfocados y fáciles de revisar.

git add -p (o *patch*) convierte el proceso de preparación (*staging*) en un diálogo interactivo, dándonos el poder de inspeccionar y decidir, línea por línea, qué incluir o excluir de nuestro próximo *commit*.

Parámetros Básicos (git add -p)

Antes de empezar, es crucial entender qué es un **hunk** en el contexto de git.

Un **hunk** (que se traduce como "trozo" o "bloque") es una unidad lógica y contigua de cambios que Git ha detectado entre la versión actual del archivo en tu directorio de trabajo y la última versión en el área de *stage* o en el *HEAD* (último *commit*).

Es decir, git agrupa automáticamente las líneas que has añadido, modificado o eliminado en bloques de código. git add -p te presenta estos *hunks* uno por uno, y te pregunta si quieres incluir todo el bloque en el próximo *commit*. Si un *hunk* es muy grande y solo quieres la mitad, puedes dividirlo.

Para comenzar el dialogo interactivo, git te hará la siguiente pregunta: *Stage this hunk [y,n,q,a,d,s,e,?]?*, los comandos más comunes son:

Comando	Significado	Acción
y	Yes	Prepara (stage) este hunk completo.
n	No	NO prepara este hunk (lo deja sin preparar).
q	Quit	Sale del modo parche sin preparar los hunks restantes.
s	Split	Intenta dividir el hunk actual en trozos más pequeños.
e	Edit	Abre un editor de texto para seleccionar líneas manualmente.
?	Help	Muestra la guía de comandos.

9.1) Asegúrate de que tu directorio de trabajo esté limpio antes de continuar.

Solución ->
`git reset --hard HEAD` `git status` (debe decir "nothing to commit, working tree clean")

9.2) Crea un archivo *config.ini* con el siguiente contenido y haz el commit inicial.

```
[Settings]\nUSER_ID = 1001\nLOG_LEVEL = INFO\nPORT =
8080\n[Database]\nURL =
mariadb://localhost\nCONNECTION_POOL = 10\nTIMEOUT = 5000"
> config.ini git add config.ini git commit -m "feat: Archivo
de configuracion base
```

Solución ->

```
echo -e "[Settings]\nUSER_ID = 1001\nLOG_LEVEL = INFO\nPORT
= 8080\n[Database]\nURL =
mariadb://localhost\nCONNECTION_POOL = 10\nTIMEOUT = 5000"
> config.ini git add config.ini

git commit -m "feat: Archivo de configuracion base"
```

9.3) Realiza los siguientes 3 cambios lógicos.

1. Cambia `LOG_LEVEL = INFO` a `LOG_LEVEL = DEBUG`.
2. Añade una nueva línea 5, después de `PORT`: `API_KEY = abc-123`.
3. Añade una nueva línea 8, después de `CONNECTION_POOL:` `RETRY_COUNT = 3`.

9.4) Inicia el modo parche (`git add -p config.ini`. SOLO queremos subir el cambio del LOG_LEVEL (Hunk 1).

9.5) Cuando git muestre el primer gran *hunk* que contiene la modificación:

- Presiona **s** (split) para separarla en distintos *hunks*
- Una vez que el hunk `LOG_LEVEL` se separe, presiona **y**.
- Para el resto de hunks (`API_KEY` y `RETRY_COUNT`), presiona **n** para dejarlos sin preparar.

9.6) Confirma que el `LOG_LEVEL` está en el *stage* y haz el commit.

Solución -> `git status git commit -m "fix: Cambiado LOG_LEVEL a DEBUG"`

9.7) Vuelve a iniciar el modo parche, ahora SOLO queremos subir el cambio `RETRY_COUNT` (*hunk* 3).

Solución -> `git add -p config.ini` (Usar **n** para el hunk del `API_KEY`, y **y** para el *hunk* del `RETRY_COUNT`.)

9.8) Confirma que solo el `RETRY_COUNT` está en el *stage* y haz el commit.

Solución -> `git status git commit -m "feat: Añadido RETRY_COUNT a DB"`

9.9) Solo debe quedar un cambio (`API_KEY`). Descarta este cambio.

Solución -> `git checkout -- config.ini`

En ocasiones, git no nos dará la opción de s (split) o intentará dividir el *hunk* y fallará (por ejemplo, cuando el bloque de código es muy pequeño y carece de suficiente contexto de líneas sin modificar).

En estos casos, tenemos que recurrir a la opción de edición interactiva (e), que nos da el control total sobre el parche:

1. Activación: El comando e abre el *hunk* en el editor de texto predeterminado de tu sistema (Nano, Vim, etc.).
2. Sintaxis: Dentro del editor, cada línea de cambio estará precedida por un símbolo que define su acción:
    - +: Indica una línea que se añadirá al *stage*.
    - -: Indica una línea que se eliminará de la versión anterior (si estuvieras editando una eliminación).
    - Líneas sin prefijo (+, -, #): Se consideran contexto y se mantienen como están.
3. Acciones Posibles:
    - Excluir una Línea: Para que una línea no se añada al *stage*, simplemente elimina el signo + que la precede.
    - Incluir una Línea Eliminada: Si tu *hunk* muestra una línea eliminada (-) y quieres que se añada de nuevo, cambia el - por un +.
    - Modificar el Contenido: Incluso puedes editar directamente el texto de la línea si eso ayuda a aislar el cambio deseado.

⚠️ Recordatorio Crítico: Si usas e, debes tener mucho cuidado de no modificar ni eliminar las líneas de metadatos o comentarios (especialmente las que empiezan por # o definen los límites del parche), ya que esto provocará el error corrupt patch y el proceso fallará.

9.10) Abre el archivo `config.ini` (que ahora está limpio) y realiza una modificación en una sola línea que combine un *fix* y una *feature*: en la línea 2, cambia el contenido de `USER_ID = 1001` a: `USER_ID = 1001 # El ID fue actualizado`.

9.11) Inicia el modo parche:

Solución -> `git add -p config.ini`

9.12) git te mostrará el *hunk* de la línea 2. Ya que no hay forma de separar el comentario del ID, y el comando **s** fallará, presiona **e (edit/editar)**.

9.13) Queremos crear un commit **solo para el comentario (`# El ID fue actualizado`)**.

- Dentro del editor, localiza la línea de adición (+) y **elimina la parte del ID** para dejar solo el comentario.
- **Original:** `+USER_ID = 1001 # El ID fue actualizado`
- **Lo que debes dejar:** `+# El ID fue actualizado`

9.14) El comentario (`# El ID fue actualizado`) está ahora en el *stage*. Confírmalo con un commit.

Solución -> `git commit -m "chore: Documentación de la actualización del ID"`

EJERCICIO 10: CREACIÓN Y TRATAMIENTO DE PARCHES (individual)

Aunque la mayoría de las colaboraciones en equipos se realizan mediante fusiones (merge) o pull requests, existen escenarios donde esta metodología es inviable. Los parches (.patch o .diff) son la herramienta de git para la **transportabilidad absoluta** de los cambios.

Un parche permite encapsular un conjunto de modificaciones (un *commit* o un grupo de cambios no confirmados) en un **archivo de texto independiente** que se puede enviar por correo electrónico, copiar entre sistemas sin conexión o aplicar a cualquier repositorio, incluso si no comparten un historial directo.

En este ejercicio, exploraremos las dos herramientas clave:

1. **git diff / git apply:** Para generar y aplicar parches a partir de **cambios en el directorio de trabajo** (ideales para correcciones rápidas).
2. **git format-patch / git am:** Para generar y aplicar parches que **contienen metadatos de un *commit*** (autor, fecha y mensaje), perfectos para el envío formal de contribuciones.

11.1) Asegúrate de que tu rama local esté limpia (git status debe decir "nothing to commit").

Solución -> `git status` *(Si hay cambios, haz un commit o descártalos con `git reset --hard HEAD`)*

11.2) Crea una nueva rama llamada fix/patch_source y un archivo llamado cambios_urgentes.txt con este contenido: Este es el archivo con el bug inicial. Confirma los cambios en un commit.

Seguidamente, cambia a la rama feature/practicandogit_a y fusiona la nueva rama para incorporar aquí el nuevo commit con el nuevo fichero.

Solución -> `git checkout -b fix/patch_source`

```
echo "Este es el archivo con el bug inicial." >
cambios_urgentes.txt
git add cambios_urgentes.txt
git commit -m "feat: Archivo fuente de cambios"
git checkout feature/prancticandogit_a
git merge fix/patch_source
```

11.3) Vuelve a la rama fix/patch_source y modifica el archivo para simular el arreglo que queremos exportar del siguiente modo:

```
echo "Este es el archivo con el bug inicial." >
cambios_urgentes.txt echo

echo "La línea con el fix (Bug Solucionado)" >>
cambios_urgentes.txt

echo "Linea adicional para el parche" >>
cambios_urgentes.txt
```

11.4) Exporta los cambios a un parche: Usa `git diff` para comparar los cambios en tu directorio de trabajo (working directory) con el último *commit* (HEAD) y guarda la salida en un archivo `bugfix.patch`.

Solución -> `git diff > bugfix.patch`

11.5) Inspecciona el parche: Mira el contenido del archivo `bugfix.patch`. Deberías ver el encabezado `diff --git...` y las líneas de adición (+) y eliminación (-).

Solución -> `cat bugfix.patch`

11.6) Descarta los cambios locales en el archivo `cambios_urgentes.txt` (ya están guardados en el parche). Luego, vuelve a la rama principal.

Solución ->

```
git restore cambios_urgentes.txt ó también git checkout –
cambios_urgentes.txt
git checkout feature/practicandogit_a
```

11.7) Aplica el parche sin confirmarlo: Usa `git apply` para aplicar los cambios en tu directorio de trabajo. Puedes usar primero `git apply --stat` para verificar el contenido del parche.

Muestra el contenido del fichero y confirma en un commit los cambios en el fichero pero no el parche.

Solución ->

```
git apply --stat bugfix.patch
git apply bugfix.patch
cat cambios_urgentes.txt
git add cambios_urgentes.txt
git commit -m "fix: Aplica parche"
```

Si bien `git diff` es útil para exportar rápidamente cambios locales (que aún no son un *commit*), en la práctica, a menudo queremos enviar un *commit* completo que ya está en nuestro historial.

El comando `git format-patch` está diseñado para este propósito. No solo exporta las diferencias de código, sino que también incluye los metadatos del *commit* original: el autor, la fecha y el mensaje descriptivo.

Al aplicar este tipo de parches, se utiliza `git am` (Apply Mailbox), que es más inteligente que `git apply`. `git am` no solo aplica los cambios, sino que también recrea el *commit* completo en el historial de la rama de destino, simplificando enormemente el proceso de contribución y manteniendo intacta la autoría original.

11.8) Vuelve a la rama `fix/patch_source` y crea un nuevo fichero `archivo_nuevo.txt` con el contenido "Este es un nuevo commit" y confírmalo en un commit.

Después, crea un parche a partir de ese commit usando `git format -1 HEAD`. Esto crea un archivo con extensión. patch que contiene solo los cambios del último commit.

Solución ->

```
git checkout fix/patch_source
echo "Este es un nuevo commit" > archivo_nuevo.txt
git add archivo_nuevo.txt
git commit -m "feat: Commit a exportar"
git format-patch -1 HEAD
```

11.9) Cambia a la rama `feature/practicandogit_a` y aplica el parche usando `git am`.

Solución ->

```
git checkout feature/practicandogit_a
git am 0001-feat-Commit-a-exportar.patch
```