USUARIO B

EJERCICIO 1: QUE TE INVITE EL COMPAÑERO/A (en pareja)

Como usuario B, te toca ser invitado por tu compañero a su repositorio de GitHub para que podáis trabajar ambos en el mismo repositorio.

1.1) Para aceptar la invitación, en tu dashboard de GitHub, justo en el icono a la izquierda de tu foto pone "*You have unread notifications*". Acepta la invitación.

1.2) Haz clone del proyecto de tu compañero/ A 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 A antes de continuar

EJERCICIO 2: TRABAJANDO CON RAMAS (en pareja)

2.1) Crea una issue de nombre "Testing file (B)" con la descripción que quieras. Debes añadir el nombre de usuario de tu compañero/a A en "Assigness" en la parte de la derecha.

Espera al compañero/a A antes de continuar

2.2) Crea una rama local de nombre "feature/practicandogit_b" y sitúate en ella.

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

2.3) Comprueba que estás situado en esa rama.

git status también valdría

Solución -> `git branch`

2.4) Súbela al repositorio remoto.

Una vez que los dos hemos subido las ramas, para que me aparezca que su rama existe tengo que actualizar: git fetch

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

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

Solución -> `echo "tu_uvus" > practicandogit_b.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_b.txt`

git add . añade todo

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 (A)" mediante su ID

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

ammend solo vale para el último commit

2.11) Sube el nuevo commit a tu rama remota.

Solución -> `git push`

Espera al compañero/a A antes de continuar

EJERCICIO 3: PULL REQUEST (en pareja)

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

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_b*"

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

Espera al compañero/a A 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 A 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`

pull nos lo traería todo y aplicaría los cambios en la rama actual si hubiese. Pero no hace falta porque no hay cambios

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

Solución -> `git checkout feature/practicandogit_a`

4.4) Actualiza los posibles cambios de esa rama.

Solución -> `git pull origin feature/practicandogit_a`

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_a.txt" con "*esta es una nueva línea remota*".

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

Solución ->

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

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

4.7) Sitúate en tu rama "feature/practicandogit_b"

Solución -> `git checkout feature/practicandogit_b`

4.8) Modifica la primera y única línea del archivo "practicandogit_b.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_b.txt
git commit -m "feat: Whatever"
```

4.9) Actualiza tu rama local

Solución -> `git pull origin feature/practicandogit_b`

¡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.

en la consola da varias opciones. P.ej.: git config pull.rebase false

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

4.11) ¡Hay un conflicto! Claro, la fusión no es posible porque se está superponiendo la misma línea en el mismo archivo (practicandogit_b.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 A)
>>>>>>> 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_b.txt
git commit -m "fix: Fix conflicts"
git push
```

Espera al compañero/a A 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_b”

Solución -> `git checkout feature/practicandogit_b`

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

Solución -> `echo "" > fix_bug_B.py`

5.3) Realiza el commit “fix: Fix important bug (developer B)” con el archivo anterior.
y haz push

Solución ->
```
git add fix_bug_B.py
git commit -m "fix: Fix important bug (developer B)"
git push
```

Espera al compañero/a A 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 A)”
o si no, pull, checkout a la otra rama, git log, copiamos el hash, y checkout de vuelta a mi rama

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_b”

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_b”

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

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"

git show :/<COMMIT_TITLE> (sin poner comillas)

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.

Pero no lo hagas aún que en el 7.3 nos dicen que hagamos otra cosa

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 --mixed de1fffd`

7.4) Haz de nuevo "git status". What? Tu rama remota "feature/practicandogit_b" 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.

Mucho ruido y pocas nueces. Mucho paja pero es una tontería. Lo que quiere decir es que si borras en local, te lo va a comparar con el remoto y te va a decir que vas por atrás. Así que haz un force push. Push normal no dejaría.

Solución ->
```
git reset --hard de1fffd
git push origin feature/practicandogit_b --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"

Para encontrar el hash hacemos el show :/ de antes

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_b", en .git/hooks/, un archivo "commit-msg"

La carpeta .git está oculta. Se puede acceder desde la terminal como se ve aquí, desde el explorador de archvos usando ctrl+h, para que te muestre los ocultos, o desde VSC configurando que te deje ver los ocultos.

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”. Antes hay que darle permisos de ejecutable al hook. En las notas generales lo tengo puesto.

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"`