Uso de Git y Github

Como hemos visto antes, Git es un sistema de control de versiones distribuido (DVCS) creado por el equipo de Linux. Actualmente, se utiliza en muchos servidores de control de versiones, como GitHub, BitBucket o GitLab, para almacenar proyectos de forma remota. Pero, si queremos interactuar con estos proyectos o repositorios remotos desde nuestra máquina local, debemos instalar Git localmente y utilizar los diferentes comandos que proporciona. En este documento aprenderemos cómo instalar Git y cómo utilizar algunos de los comandos básicos.

Instalación de Git en tu máquina

1. Instalación y configuración de Git

La instalación de Git depende del sistema operativo en el que queremos instalarlo.

Para sistemas Linux, solo tenemos que ejecutar el comando especificado para instalar Git. Por ejemplo, en sistemas Ubuntu tenemos que ejecutar este comando:

```
sudo apt-get install git
```

Para Windows y Mac, tenemos que ir a la página web de Git y descargar la versión adecuada. En el caso de Mac, también se puede instalar Git a través de la instalación de XCode.

1.1. Configuración de Git

Antes de utilizar los comandos de Git, deberíamos configurar algunas variables predeterminadas en nuestro sistema, para conectarnos fácilmente a los servidores y almacenar nuestras credenciales para conexiones posteriores. Utilizaremos el comando `git config` para almacenar estas variables, y las podemos guardar a tres niveles diferentes:

- **Sistema**: utilizando el parámetro `--system`, la configuración se aplica a todos los usuarios del sistema.
- **Usuario**: utilizando el parámetro `--global`, la configuración se aplica solo al usuario actual del sistema. Esta es la opción que utilizaremos en esta sección.
- **Repositorio**: cada repositorio tendrá sus propios parámetros de configuración de Git.

Primero de todo, definimos nuestro nombre completo con este comando (sustituye "John Doe" por tu nombre real):

```
git config --global user.name "John Doe"
```

A continuación, especificamos el correo electrónico con el que creamos la cuenta de GitHub:

```
git config --global user.email yourEmail@server.com
```

Después, podemos especificar el editor predeterminado de Git. Este paso no es necesario, pero si Git necesita abrir un fichero de texto para mostrar información, este será el editor que utilizaremos. Por ejemplo, podemos utilizar el Bloc de notas en Windows de esta manera:

```
git config --global core.editor notepad
```

Finalmente, debemos especificar cómo Git almacenará nuestras credenciales, de manera que no tengamos que introducirlas cada vez que nos conectemos a los repositorios. El ayudante que utilizamos para guardar las credenciales depende del sistema operativo en el que estamos utilizando Git, pero el comando general es este:

```
git config --global credential.helper <helper>
```

donde `<helper>` depende del sistema operativo:

- Para Windows utilizamos `wincred`
- Para Linux utilizamos `cache`
- Para Mac OSX utilizamos `osxkeychain`

Así que, si queremos configurar el ayudante de credenciales en Windows, por ejemplo, escribimos algo así:

```
git config --global credential.helper wincred
```

De esta manera, ya estamos preparados para utilizar Git, incluso desde diferentes entornos de desarrollo, como veremos en otras secciones. Podemos comprobar la configuración actual utilizando el comando `git config --list`. También podemos comprobar la versión de Git que tenemos instalada actualmente con el comando `git version`.

2. Comandos locales básicos útiles

Veamos ahora algunos comandos que podemos utilizar para trabajar con proyectos locales (sin conectarnos a ningún repositorio o servidor remoto). Estos comandos son útiles tanto para proyectos locales como para proyectos remotos que hayamos descargado previamente, si queremos trabajar localmente con ellos durante un tiempo.

2.1. Crear un repositorio local

Si queremos inicializar o crear un nuevo repositorio local, primero debemos crear la carpeta donde se guardará este proyecto. Después, podemos inicializarla como repositorio Git con este comando (desde dentro de la carpeta del proyecto):

```
git init
```

Esto inicializará esta carpeta como carpeta Git, creando una subcarpeta oculta llamada `.git`, donde se almacenará la base de datos del repositorio. No nos debemos preocupar por esta subcarpeta.

Cada fichero dentro de este repositorio estará en uno de los tres estados mencionados en secciones anteriores (committed, staged o modified), y podemos cambiar el estado de cada fichero escribiendo algunos de los comandos que veremos ahora. También podemos comprobar el estado del repositorio en cualquier momento con el comando `git status` (lo debemos ejecutar desde la carpeta raíz del repositorio). Nos informará si todo está "commitado", o si hay algún fichero con cambios sin guardar.

> **Actividad**
>
> 1. Instala Git en tu ordenador, en la máquina virtual facilitada en clase o en una máquina virtual. En este último caso, puedes descargar una imagen de Linux como por ejemplo Lubuntu y utilizar VirtualBox (o cualquier otro). Para poder utilizar una máquina virtual en tu ordenador, **debes permitirlo en la configuración de la BIOS** (permitir virtualización).
> 2. Configura Git con tu cuenta creada en el apartado anterior.
> 3. Crea una carpeta llamada **GitExercises** en tu sistema. Almacenaremos algunos repositorios dentro de ella. Para empezar, crea dentro de esta carpeta una subcarpeta nueva llamada **MyFirstLocalRepo**, entra dentro de esta carpeta y ejecuta el comando `git init` para inicializar esta carpeta como repositorio Git.

2.2. Añadir o editar ficheros en el repositorio

Si añadimos algún fichero nuevo a la carpeta del repositorio (por ejemplo, un fichero llamado `file.txt`) y ejecutamos el comando `git status`, Git mostrará que hay algunos ficheros que deben ser añadidos al repositorio.
Estos ficheros se encuentran en estado "modificado". Si utilizamos el comando `git add`, el (los) fichero(s) se marcarán como "preparados" (staged). Si solo queremos añadir un único fichero, especificaremos este nuevo fichero como parámetro:

```
git add file.txt
```

No obstante, puede haber muchos cambios en nuestro repositorio. Si queremos añadirlos todos a la vez, utilizamos `.` como parámetro:

```
git add .
```

Después de cada nuevo cambio que hagamos en el repositorio (ya sea añadiendo, editando o eliminando ficheros), debemos repetir este comando para preparar los cambios. Una vez los cambios estén preparados, este es el resultado del comando `git status`:

Como puedes ver en la imagen anterior, podemos utilizar el comando `git rm` para desmarcar este fichero, si queremos:

```
git rm --cached file.txt
```

2.3. Guardar o *commitear* cambios

Después de añadir o preparar los cambios, debemos hacer un último paso para actualizar la base de datos del repositorio. Esta operación es el "**commit**", y la podemos hacer mediante el comando `git commit` . Podemos ejecutarla después de una o varias operaciones de `git add` que hayan preparado uno o más ficheros.

Esta es la estructura general del comando `git commit` :

```
git commit -m "Mi primer commit"
```

El parámetro `-m` nos permite especificar un mensaje de commit. Este mensaje es obligatorio para guardar el commit, de manera que, si queremos recuperarlo más adelante, podemos identificar este mensaje en la lista de commits. Después de commitear los cambios, si ejecutamos `git status` , deberíamos ver que no hay nada por commitear:

```
On branch master
nothing to commit, working tree clean
```

Alternativamente, también podemos utilizar el parámetro `-a` para añadir o preparar automáticamente los cambios antes de commitearlos. Este comando combina un `git add .` y un comando `git commit` :

```
git commit -a -m "Tu mensaje de commit"
```

Mostrar el historial de commits

Si queremos ver el historial de commits de nuestro repositorio, podemos escribir este comando:

```
git log
```

Cada commit tiene una etiqueta que consiste en una secuencia de dígitos y letras. En el ejemplo anterior, nuestro commit se ha etiquetado como `08f4ed1751…` . Esta etiqueta será útil para comprobar el commit más adelante, aunque no es necesario recordar todos estos caracteres, solo el prefijo inicial.

Mostrar los cambios

También podemos ver los cambios entre dos versiones consecutivas del repositorio. Hay muchas maneras de hacerlo:

- `git show` : muestra los cambios realizados en el último commit.
- `git show cb1fd6f8` : muestra los cambios hechos en el commit etiquetado con el prefijo `cb1fd6f8` (como se puede ver, no necesitamos escribir toda la etiqueta).
- `git diff` : muestra los cambios realizados en la última versión que todavía no se ha commiteado.
-

> **Actividad**
>
> Haz los siguientes cambios en el repositorio **MyFirstLocalRepo** que has creado en el ejercicio anterior:
>
> 1. Crea un fichero nuevo llamado `file.txt` con el texto "Mi primer fichero de texto". Guarda los cambios en este fichero.
> 2. Ejecuta el comando `git add .` para preparar este fichero.
> 3. Ejecuta el comando `git commit` con el mensaje "Mi primer commit" para guardar los cambios a la base de datos.
> 4. Edita el fichero `file.txt` y añade una segunda línea con tu nombre.
> 5. Ejecuta el comando `git commit –a –m` para preparar y commitear automáticamente los cambios con el mensaje "Mi segundo commit".
> 6. Ejecuta el comando `git log` para ver el historial de commits. Deberías ver algo similar a esto:

```
commit 08f4ed1751…
Author: John Doe <john@example.com>
Date: Wed Apr 14 15:00 2023
    Mi primer commit
commit cb1fd6f8…
Author: John Doe <john@example.com>
Date: Wed Apr 14 16:00 2023
    Mi segundo commit
```

> 7. Ejecuta el comando `git show` para ver los cambios hechos en el último commit. Deberías ver algo parecido a esto:
>
> Los nuevos cambios se muestran en verde si han sido añadidos (en este caso, tu nombre al final del contenido del fichero), o en rojo si han sido eliminados.

Etiquetar commits

Podemos añadir manualmente etiquetas a un commit determinado, de manera que lo podemos encontrar fácilmente más adelante cuando queramos mostrar sus cambios. Utilizamos el comando `git tag` seguido del nombre de la etiqueta:

```
git tag v1.0
```

Esto se aplica al último commit enviado. A continuación, podemos mostrar los cambios de este commit con este comando:

```
git show v1.0
```

Si queremos etiquetar un commit que no es el último, entonces debemos especificar la etiqueta anterior de ese commit (o su prefijo inicial), después de la nueva etiqueta que queremos asignarle:

```
git tag v1.0 cb1fd6f8
```

2.4. Deshacer cambios

¿Y si queremos volver a un commit anterior y deshacer los cambios hechos en el (los) último(s) commit(s)? Podemos utilizar el comando `git reset` . Este comando se puede utilizar de muchas maneras, pero aquí explicaremos una de ellas: debemos identificar la etiqueta del commit que queremos establecer como el actual y después escribir este comando:

```
git reset --hard 0305afd
```

donde `0305afd` es el prefijo de la etiqueta del commit que queremos establecer como nuestro estado actual activo.

2.5. El fichero .gitignore

En cada repositorio Git, podemos añadir manualmente un fichero llamado `.gitignore` . Es solo un fichero de texto que contiene una lista de ficheros y carpetas que deben ser ignorados cuando subamos nuevos cambios. Por ejemplo, si estamos trabajando en un proyecto C#, no necesitamos subir ficheros `.exe` al repositorio, ya que podemos recompilar el proyecto de nuevo. Así que podemos editar este fichero y especificarlo así:

```
*.exe
```

Esto saltará todos los ficheros `.exe` de la carpeta principal del proyecto. De la misma manera, podemos añadir tantos ficheros y carpetas como necesitemos en este fichero. Por ejemplo:

```
node_modules/
*.exe
*.tmp
```

Esto omite la carpeta `node_modules` y todos los ficheros `.exe` o `.tmp` en la carpeta raíz. Aquí puedes encontrar ficheros `.gitignore` típicos para muchos tipos diferentes de proyectos, como proyectos Node, Laravel, etc.

NOTA: El fichero `.gitignore` **NO** excluye ficheros que ya han sido *commiteados* previamente. Por ejemplo, si le decimos a este fichero que ignore ficheros `.exe` pero previamente hemos commiteado un fichero `.exe` al repositorio, este fichero no se eliminará del mismo.

3. Trabajando con repositorios remotos

Ahora que hemos aprendido cómo añadir y editar contenido en un repositorio local, veamos cómo conectarnos a un repositorio remoto de GitHub para descargar o subir los cambios. Primeramente, si queremos trabajar con repositorios remotos almacenados en GitHub, necesitaremos crear este repositorio remoto en GitHub.

3.1. Clonar repositorios

Una vez tengamos nuestro repositorio creado en GitHub, necesitaremos copiarlo a nuestra máquina local. Esta operación se conoce como una operación de **clonación**, y la hacemos con el comando `git clone`, especificando la URL del repositorio, la cual se puede recuperar del botón *Clone or download* en el mismo repositorio.

Ejemplo de página principal del repositorio:

Si tuviéramos un repositorio en GitHub, el comando adecuado para clonarlo sería así:

```
git clone https://github.com/johndoe/test
```

Este comando creará una carpeta llamada **test** en el directorio desde el cual estamos ejecutando esta comando, así que asegúrate de ejecutarla dentro de la carpeta donde quieras colocar tu proyecto.

> **Actividad**
> Clona el repositorio donde se almacenan estos apuntes:
> https://github.com/arturoalbero/Materiales-EDE-2425

3.2. Actualizar los cambios remotos en local

Una vez tengamos el repositorio clonado localmente, si estamos trabajando en equipo o gestionando el mismo repositorio desde diferentes ordenadores, quizás necesitaremos descargar los últimos cambios de este repositorio para actualizar nuestra copia local. Este paso es esencial para asegurarnos que tenemos los contenidos actualizados antes de hacer nuevos cambios.

Para hacer esto, simplemente utilizamos el comando `git pull` desde la carpeta del repositorio:

```
git pull
```

Esto descarga automáticamente los últimos cambios y actualiza los ficheros afectados.

3.3. Subir los cambios locales al repositorio remoto

Si tenemos nuestro repositorio local actualizado y hacemos nuevos cambios a cualquier fichero, podemos subir estos cambios al repositorio remoto. Los pasos necesarios son los siguientes:

1. Haz cambios en el(los) fichero(s) deseado(s).
2. Marca los ficheros como preparados con el comando `git add .` que ya hemos visto antes.
3. Haz un **commit** de los cambios localmente con el comando `git commit` que también hemos visto antes.
4. Sube este commit (o los últimos commits, si hay más de uno) con el comando `git push` .

```
git push
```

Ejercicio 3:

1. Clona el repositorio de GitHub **MyFirstRepo** que habrás creado en el documento anterior. Clónalo dentro de la misma carpeta principal donde estás creando el resto de repositorios locales de este documento. Verás una nueva carpeta llamada **MyFirstRepo** que contiene todos los elementos de tu repositorio remoto.
2. Aplica estos cambios:
    - Añade un nuevo fichero llamado `shopping_list.txt` con una lista de artículos que quieras comprar.
    - Sube este fichero al repositorio remoto (recuerda, primero añade los cambios, después haz el commit y finalmente haz el push).
3. Ve a GitHub y comprueba que el nuevo fichero se ha subido correctamente.
4. Ve a una carpeta diferente del ordenador y clona una copia nueva del mismo repositorio.

5. Desde esta segunda carpeta, añade un nuevo fichero llamado `to_do.txt` y añade algunas tareas pendientes para las próximas semanas.
6. Sube los cambios al repositorio remoto.
7. Vuelve a tu carpeta original **MyFirstRepo** y haz un comando `git pull`. Comprueba si el nuevo fichero **to_do.txt** se ha descargado en esta copia local.

3.4 Crear un repositorio remoto a partir de nuestro repositorio local

Para crear un repositorio a partir de nuestro repositorio local, lo que debemos hacer es conectar nuestro repositorio local a un repositorio remoto vacío (para evitar conflictos de fusión). Para ello, seguimos estos pasos:

1. **Creamos el repositorio de Git si no lo tenemos con `git init`**
2. **Añadimos los archivos que queramos y hacemos al menos un commit**
3. **Creamos un Nuevo Repositorio en GitHub:**

- Ve a GitHub.com e inicia sesión en tu cuenta.
- Haz clic en el signo **+** en la esquina superior derecha y selecciona **"New repository"** (Nuevo repositorio).
- Dale un **nombre** a tu repositorio (normalmente, es buena idea que coincida con el nombre de tu carpeta local, pero no es obligatorio).
- Puedes añadir una **descripción** si quieres.
- **¡Importante!** No marques las opciones "Initialize this repository with a README", "Add .gitignore" ni "Choose a license". Así evitamos conflitos.
- Finalmente, haz clic en el botón **"Create repository"** (Crear repositorio).

4. **Conecta tu Repositorio Local con el de GitHub:**

Después de crear el repositorio en GitHub, verás una página con instrucciones. Busca la sección que dice "...or push an existing repository from the command line." (o sube un repositorio existente desde la línea de comandos). Usaremos los siguientes dos comandos:

- **Añade el origen remoto:** Este comando le dice a tu Git local dónde está tu repositorio de GitHub. Asegúrate de reemplazar `TU_NOMBRE_DE_USUARIO` y `TU_NOMBRE_DEL_REPOSITORIO` con tu información real.

```
git remote add origin https://github.com/TU_NOMBRE_DE_USUARIO/TU_NOMBRE_DEL_REPOSITORIO
```

Ejemplo: `git remote add origin https://github.com/octocat/mi-proyecto-genial.git`

- **Verifica el remoto (Opcional pero recomendado):** Puedes comprobar que el remoto se añadió correctamente:

```
git remote -v
```

Deberías ver algo así:

```
origin  https://github.com/TU_NOMBRE_DE_USUARIO/TU_NOMBRE_DEL_REPOSITORIO.git (fetch)
origin  https://github.com/TU_NOMBRE_DE_USUARIO/TU_NOMBRE_DEL_REPOSITORIO.git (push)
```

`fetch` es recoger, `push` es subir.

5. **Sube tu Repositorio Local a GitHub:**

- `git branch -M main` : Renombra tu rama actual a `main` . GitHub usa `main` como nombre de rama principal por defecto. Si tu rama local ya se llama `main` o prefieres mantenerla como `master` , puedes omitir este comando o ajustarlo (por ejemplo, `git push -u origin master`).
- `git push -u origin main` : Sube tu rama local `main` al repositorio remoto llamado `origin` . La opción `-u` (o `--set-upstream`) es crucial porque establece la rama remota de seguimiento, lo que significa que en el futuro, los comandos `git push` y `git pull` sabrán con qué rama remota interactuar sin que tengas que especificarla cada vez.

```
git branch -M main
git push -u origin main
```

Tabla resumen de comandos básicos

Comando	Utilidad
`git init`	Inicializa un nuevo repositorio Git en la carpeta actual.
`git status`	Muestra el estado del repositorio y los ficheros modificados o no commiteados.
`git add <fichero>`	Añade un fichero específico al área de "stage" para el siguiente commit.
`git add .`	Añade todos los ficheros modificados al área de "stage".

Comando	Utilidad
`git commit -m "<mensaje>"`	Realiza un commit de los cambios con un mensaje descriptivo.
`git commit -a -m "<mensaje>"`	Añade y commitea automáticamente los cambios sin necesidad de `git add .`
`git log`	Muestra el historial de commits del repositorio.
`git show`	Muestra los cambios realizados en el último commit o en un commit específico.
`git diff`	Compara los cambios entre las versiones no commiteadas del repositorio.
`git rm --cached <fichero>`	Elimina un fichero del área de "stage" sin borrarlo del sistema.
`git reset --hard <commit>`	Restaura el estado del repositorio a un commit anterior.
`git tag <etiqueta>`	Añade una etiqueta a un commit específico.
`git clone <URL>`	Clona un repositorio remoto en la máquina local.
`git pull`	Actualiza la copia local con los cambios del repositorio remoto.
`git push`	Sube los cambios locales al repositorio remoto.
`git config --global user.name "<nombre>"`	Configura el nombre de usuario para los commits.
`git config --global user.email "<email>"`	Configura el correo electrónico para los commits.
`git config --list`	Muestra la configuración actual de Git.
`git version`	Muestra la versión de Git instalada.

Uso de ramas en Git

En esta segunda parte, nos centraremos en el trabajo con **ramas** (**branches**) de Git, una de las funcionalidades más poderosas de este sistema de control de versiones. Las ramas permiten trabajar en funciones o mejoras de manera independiente sin afectar la rama principal del proyecto (normalmente llamada `main` o `master`). Así, podemos desarrollar, probar y modificar nuevas funcionalidades sin interferir en la versión estable del proyecto.

1. ¿Qué es una rama en Git?

Una rama en Git es esencialmente una secuencia de commits que comienza a partir de un punto específico en la historia del repositorio. La rama principal del proyecto es generalmente `main` o `master`, pero podemos crear tantas ramas como necesitemos para desarrollar nuevas características o corregir errores de manera paralela.

Cuando trabajamos con ramas, podemos hacer cambios en una rama sin que afecten a las otras, y posteriormente podemos combinar las ramas para integrar esos cambios en el proyecto principal.

2. Crear y cambiar de ramas

2.1. Crear una nueva rama

Para crear una nueva rama, utilizamos el comando `git branch`, seguido del nombre de la nueva rama. Esta nueva rama se creará a partir del estado actual del repositorio, es decir, del commit donde estamos en este momento.

```
git branch nombre-rama
```

Por ejemplo, si quieres crear una rama para desarrollar una nueva funcionalidad llamada "nueva-funcion", puedes hacerlo así:

```
git branch nueva-funcion
```

2.2. Cambiar de rama

Una vez creada la nueva rama, si queremos trabajar en ella, debemos cambiar a ella. Para ello, utilizamos el comando `git checkout` seguido del nombre de la rama a la que queremos cambiar:

```
git checkout nueva-funcion
```

A partir de este momento, cualquier cambio que hagamos se aplicará a esta rama, sin afectar la rama principal (`main` o `master`).

Nota: A partir de Git 2.23, también podemos utilizar `git switch` para cambiar de rama, una alternativa más clara a `git checkout` :

```
git switch nueva-funcion
```

2.3. Crear y cambiar de rama en un solo paso

Es posible crear una rama y cambiar a ella en un solo paso con este comando:

```
git checkout -b nombre-rama
```

Por ejemplo:

```
git checkout -b nueva-funcion
```

Este comando crea la rama `nueva-funcion` y automáticamente nos cambia a ella.

3. Trabajar en una rama

Una vez estemos trabajando en una nueva rama, podemos hacer modificaciones, añadir ficheros y realizar commits de la misma manera que lo haríamos en la rama principal. Estos cambios quedarán aislados dentro de la rama y no afectarán a ninguna otra rama hasta que decidamos combinarlas.

3.1. Ver las ramas disponibles

Podemos ver todas las ramas existentes en nuestro repositorio utilizando este comando:

```
git branch
```

La rama en la que estamos trabajando actualmente estará marcada con un asterisco `*` .

4. Fusionar ramas (merge)

Una vez hayamos terminado el desarrollo o la corrección de errores en nuestra rama, es momento de fusionar los cambios con la rama principal. Este proceso se denomina **fusión** o **merge**.

Antes de realizar la fusión, es recomendable asegurarse de que estamos en la rama donde queremos aplicar los cambios (normalmente `main` o `master`). Para ello, cambiamos a la rama con:

```
git checkout main
```

Una vez estamos en la rama principal, podemos utilizar el comando `git merge` para fusionar los cambios de la rama secundaria en la que hemos estado trabajando (por ejemplo, `nueva-funcion`):

```
git merge nueva-funcion
```

Si no hay conflictos entre los cambios, Git combinará automáticamente las dos ramas. Si hay conflictos (por ejemplo, si se han modificado las mismas líneas de código en ambas ramas), Git nos avisará y tendremos que resolverlos manualmente.

5. Resolver conflictos de fusión

Cuando aparecen conflictos durante una fusión, Git marca las líneas de código en los ficheros afectados que tienen conflictos. Deberíamos revisar estas líneas manualmente, escoger qué versión queremos mantener, y después marcar los conflictos como resueltos con `git add`. Finalmente, hacemos un commit para completar la fusión:

```
git add <fichero>
git commit
```

6. Borrar ramas

Después de fusionar los cambios de una rama secundaria a la rama principal, es una buena práctica eliminar esa rama para mantener el repositorio limpio. Podemos hacerlo con el comando `git branch -d`:

```
git branch -d nueva-funcion
```

Si la rama no ha sido fusionada, Git no permitirá borrarla. Si estamos seguros de que queremos eliminarla aunque no haya sido fusionada, podemos forzar la eliminación con `git branch -D` :

```
git branch -D nueva-funcion
```

7. Trabajo con ramas remotas

Cuando trabajamos con repositorios remotos (como GitHub o GitLab), las ramas locales no se suben automáticamente al servidor remoto. Para subir una nueva rama, debemos utilizar el comando `git push` especificando el nombre de la rama:

```
git push origin nueva-funcion
```

Una vez la rama esté subida al repositorio remoto, otros colaboradores podrán acceder a ella.

> **Actividad**
> Trabajando con ramas en Git
>
> En este ejercicio, aprenderás a trabajar con ramas para desarrollar nuevas funcionalidades o hacer correcciones sin afectar el código de la rama principal. Seguirás el proceso completo de creación de ramas, fusión de cambios y resolución de conflictos, todo en un escenario práctico.
>
> **Escenario:**
>
> Añade al repositorio antes creado, MyFirstRepo, un fichero llamado `index.html` a la rama principal (`main`). Tu objetivo es añadir dos nuevas funcionalidades de manera independiente, trabajando en ramas separadas y, después, fusionarlas con la rama principal. Documenta los pasos con capturas de pantalla y comparte la memoria del ejercicio.
> Puedes emplear el siguiente código como base del fichero:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Archivo</title>
</head>
<body>

</body>
</html>
```

> **Pasos a seguir:**
>
> 1. **Clona el repositorio y sitúate en la rama principal:**

```
# Si aún no lo has hecho
git clone <URL_DE_TU_REPOSITORIO_MyFirstRepo>
cd MyFirstRepo
git checkout main # Asegúrate de estar en la rama main
```

> Crea el archivo `index.html` con el contenido proporcionado en la rama `main` y súbelo al repositorio remoto.
>
> 2. **Crea una nueva rama para añadir un título a la página web:**
>
> - Crea la rama `añadir-titulo` y cambia a esta rama:

```
git checkout -b añadir-titulo
```

> - Abre el fichero `index.html` y añade un título dentro de la etiqueta `<body>` :

```
<h1>Bienvenidos a mi página web</h1>
```

> - Guarda los cambios y haz un commit:

```
git add index.html
git commit -m "Añadido título a la página web"
```

> 3. **Crea otra rama para añadir un párrafo de introducción:**

- Cambia a la rama principal (`main`) y después crea una nueva rama `añadir-introduccion` :

```
git checkout main
git checkout -b añadir-introduccion
```

- Modifica el fichero `index.html` y añade un párrafo de introducción bajo el título:

```
<p>Esta es una página web de ejemplo creada para aprender Git.</p>
```

- Guarda los cambios y haz un commit:

```
git add index.html
git commit -m "Añadido párrafo de introducción"
```

4. **Fusiona las ramas con la rama principal:**

- Vuelve a la rama principal:

```
git checkout main
```

- Fusiona la rama `añadir-titulo` con la rama principal:

```
git merge añadir-titulo
```

- A continuación, fusiona la rama `añadir-introduccion` con la rama principal:

```
git merge añadir-introduccion
```

5. **Resuelve un conflicto de fusión (opcional):**

Si hubiera conflictos durante la fusión (por ejemplo, si el título y el párrafo hubieran sido añadidos en las mismas líneas del fichero `index.html`), resuélvelos de la siguiente manera:

- Abre el fichero conflictivo (`index.html`) y modifica las líneas marcadas por Git para mantener las dos funcionalidades (el título y el párrafo).
- Después, marca los conflictos como resueltos:

```
git add index.html
```

> - Haz un commit para completar la fusión:

```
git commit -m "Resueltos conflictos y fusionadas ramas"
```

> 6. **Borra las ramas:**
>
> Después de fusionar las ramas, puedes borrarlas para mantener el repositorio limpio:

```
git branch -d añadir-titulo
git branch -d añadir-introduccion
```

Tabla resumen de comandos de ramas en Git

Comando	Utilidad
`git branch`	Lista todas las ramas del repositorio.
`git branch <nombre-rama>`	Crea una nueva rama con el nombre especificado.
`git checkout <nombre-rama>`	Cambia a la rama especificada.
`git checkout -b <nombre-rama>`	Crea una nueva rama y cambia a ella en un solo paso.
`git switch <nombre-rama>`	Cambia a la rama especificada (alternativa a `git checkout`).
`git merge <nombre-rama>`	Fusiona la rama especificada con la rama actual.
`git branch -d <nombre-rama>`	Borra una rama local después de fusionarla.
`git branch -D <nombre-rama>`	Fuerza la eliminación de una rama no fusionada.
`git push origin <nombre-rama>`	Sube una nueva rama al repositorio remoto.
`git pull origin <nombre-rama>`	Descarga e integra los cambios de una rama remota en la rama local.
`git stash`	Guarda los cambios no commiteados temporalmente para cambiar de rama.
`git stash apply`	Recupera los cambios guardados anteriormente con `git stash`.