Actividad: Comandos de Git

Para cada punto de la práctica, hacer una captura con el comando solicitado y su respuesta. Se muestran como ejemplo los puntos 1 y 2.

En las capturas se debe identificar en el *prompt* el nombre de la persona que hace la práctica.

**Ej.** alumno@**cliente-rodriguez-joseramon**:~/Documentos/reposgit$

Tras cada comando ejecutado que produzca alguna modificación en el repositorio, comprobar el estado del repositorio, explicando el resultado de la ejecución de dicho comando.

Siglas de uso común:

* **SA**: ***S****taging* ***A****rea* o Área de preparación/montaje.
* **WD**: ***W****orking* ***D****irectory* o Directorio de trabajo.
* **WIP**: ***W****ork* ***I****n* ***P****rogress* o Trabajo en proceso.

Toda la práctica se realizará en un directorio **reposgit**, que no es un repositorio Git en sí mismo, donde a su vez se crearán algunos directorios que sí serán repositorios Git.

Al final de la práctica, en la sección de ANEXOS, aparecen listados los comandos, tanto de [Git](#ANEXO_I_Comandos_Git), como de [Linux](#ANEXO_II_Comandos_Linux), que se usan en esta práctica.

**\* NOTA 1:** En varios de los apartados se pide que se obtenga un cierto resultado “con un solo comando”. Habitualmente esto quiere decir que hay que combinar en una misma línea varios comandos para obtener el resultado. En Linux existen operadores para ello como tubería (*pipe*) **|** o redireccionamiento **>** , **>>**, **<** o **<<**.

**Ej.** Mostrar, ordenadas alfabéticamente, las líneas del archivo **eventos** que contengan la cadena de texto 2020:

**cat eventos | grep 2024 | sort**

**\* NOTA 2:** En varios apartados se pedirá lo mismo, por ejemplo, listar todos los *commits* en una línea por *commit*, en visualización gráfica y para todas las ramas. Hay que hacer una captura diferente por cada uno de estos apartados, ya que el estado del repositorio es diferente en cada punto de la práctica.

**Trabajo con la clonación de un repositorio ya existente**

1. Clonar el repositorio ubicado en https://github.com/grayghostvisuals/practice-git.

|  |
| --- |
| A computer screen shot of a black screen  Description automatically generated |

1. Mediante la ejecución de un solo comando (una sola línea), compuesto de un comando Git para listar *commits*, usado conjuntamente con el comando de wc de Linux, indicar cuántos *commits* contiene el proyecto en su WD. Hacerlo de diferentes maneras, al menos de dos.

|  |
| --- |
|  |
|  |

1. Mostrar la lista de contribuyentes/colaboradores del repositorio, así como el número de *commits* que ha hecho cada uno. El resultado se visualizará ordenado descendentemente por número de *commits*.

|  |
| --- |
| (sigue) |

1. Mostrar, con un comando, los nombres de los contribuyentes/colaboradores que hay en el repositorio, ordenados ascendentemente.

|  |
| --- |
| (sigue) |

1. Mostrar, con un comando, el número de contribuyentes/colaboradores que hay en el repositorio.

|  |
| --- |
|  |

1. Mediante un comando de Linux, usado conjuntamente con el comando usado anteriormente, mostrar el número de commits existentes en el repositorio. Se trata de hacer la suma de los números que hay dentro de un archivo o flujo de caracteres.

|  |
| --- |
|  |

1. Mostrar el histórico de *commits* en el repositorio, en modo de un *commit* por línea, con salida gráfica y para todas las ramas del repositorio.

|  |
| --- |
|  |

1. Mostrar los nombres de todas las ramas existentes.

|  |
| --- |
|  |

1. Mostrar el estado del repositorio y explicar la información que se obtiene.

|  |
| --- |
| Primero me dice que me encuentro en la rama ‘master’.  Luego que esa rama está sincronizada con la rama ‘master’ del repositorio remoto ‘origin’.  Y por último, que no hay nada preparado para hacer ‘commit’. |

1. Mostrar las direcciones de los servidores en la nube donde se almacena el repositorio, tanto para subir contenidos (*push*), como para descargarlos (*fetch* o *pull*).

|  |
| --- |
|  |

1. Mostrar los *commits* realizados por el contribuyente **grayghostvisuals** durante el año 2019.

|  |
| --- |
| (sigue) |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Trabajo con un nuevo repositorio actividad**

1. Configurar el usuario y su correo.

|  |
| --- |
|  |

1. Configurar el nombre de la rama por defecto de todo nuevo repositorio local para que sea **main**.

|  |
| --- |
|  |

1. Configurar los siguientes alias:
   1. st 🡪 status
   2. lg 🡪 log --oneline --graph --all

|  |
| --- |
|  |

1. Crear, dentro del directorio reposgit, un repositorio de nombre actividad.

|  |
| --- |
|  |

1. Mostrar el histórico de *commits* del repositorio. Explicar el resultado obtenido.

|  |
| --- |
| Usamos el alias que acabamos de crear y el resultado nos sale vacío porque todavía no se han realizado commits en este repositorio. |

1. Mostrar los nombres de los contribuyentes al repositorio con el número de *commits* que ha realizado cada uno. Explicar el resultado obtenido.

|  |
| --- |
| Sucede más o menos lo mismo, como no hay commits manda ese mensaje que nos indica que GIT está esperando recibir mensajes de ‘log’ desde la entrada estándar (standard input). |

1. Mostrar la información disponible del último *commit*. Explicar el resultado obtenido.

|  |
| --- |
| Más de lo mismo, aquí GIT nos lo informa. |

1. Mostrar el nombre de las ramas existentes. Explicar el resultado obtenido.

|  |
| --- |
| Sigue sin mostrar ningún resultado, en este caso podríamos suponer que sí hay una rama: la rama ‘main’, pero esta es un puntero que apunta al último commit, si no hay commits es como si no existiera, o lo que es lo mismo: empezará a existir en cuanto hagamos un commit. |

1. Crear una nueva rama de nombre bifurca. Explicar el resultado obtenido.

|  |
| --- |
| Sucede lo mismo, una rama, cualquiera que sea no es más que un puntero a un commit específico, si no hay commits entonces no hay nada a lo que ese puntero pueda hacer referencia. |

1. Mostrar las direcciones de los servidores en la nube donde se almacena el repositorio, tanto para subir contenidos (*push*), como para descargarlos (*fetch* o *pull*).

|  |
| --- |
| Este comando no muestra nada porque todavía no hemos configurado un repositorio remoto. |

1. Verificar la conectividad y validez de los objetos de la base de datos de Git. Explicar el resultado obtenido.

|  |
| --- |
| La primera línea verifica que la estructura de los directorios están en su lugar, complete y sin problemas.  La Segunda línea explicita lo que ya hemos mencionado: que el punter ‘HEAD’ apunta a una rama ‘no nacida’ (“unborn”) que será ‘main’.  La última línea señala que no hay referencias predeterminadas en el repositorio, que solo tienen sentido si hemos creado alguna rama. |

**A partir de aquí, se irán realizando operaciones, tras cada una de las cuáles se irá mostrando el estado del repositorio.**

1. Crear un archivo de nombre **letras** que contenga la letra **a**.

|  |
| --- |
|  |

1. Añadir el archivo letras al SA. ¿Ha cambiado el contenido del archivo?

|  |
| --- |
| El contenido no ha cambiado: |

1. Ejecutar el comando git rm letras. Explicar la respuesta obtenida.

|  |
| --- |
| Como el archive está en la SA antes de eliminarlo GIT nos advierte de que se perdería, luego nos explica cómo podríamos forzar la eliminación y cómo podríamos quitarlo de la SA. |

1. Sacar el archivo del SA (*unstage*).

|  |
| --- |
|  |

1. Añadir de nuevo el archivo al SA. Realizar un *commit* con el mensaje *Creado el archivo* ***letras***.

|  |
| --- |
|  |

1. Mostrar los nombres de los archivos contenidos en este *commit* recién creado. Únicamente los nombres.

|  |
| --- |
|  |

1. Mostrar los nombres de los archivos contenidos en el *commit* recién creado, con sus SHA y su tipo.

|  |
| --- |
|  |

1. ¿Cuál es el SHA del archivo **letras** en el WD?

|  |
| --- |
|  |

1. ¿Cuál es el SHA del archivo **letras** en el SA?

|  |
| --- |
|  |

1. Añadir la letra **b** al archivo **letras**. Subir el archivo al SA. Sacar el archivo del SA. ¿Ha cambiado el contenido del archivo?

|  |
| --- |
| El archive no ha cambiado: |

1. Añadir de nuevo el archivo **letras** al SA. Añadir la letra **c** al archivo **letras**. Comprobar el estado del repositorio. Hacerlo de nuevo con el comando **git status -s**. Explicar la salida devuelta por ambos comandos de comprobación del estado del repositorio.

|  |
| --- |
| Me informa que hay una versión de letras en la SA y que el archivo letras ha sido modificado, luego me dice como puedo revertir los cambios del archivo o como puedo volver a subirlo con esas modificaciones a la SA.    Mustra la misma información pero abreviada: ‘MM’ indica que ‘letras’ ha sido modificado tanto en la SA como en el WD |

1. ¿Qué sucede si se ejecuta el comando **git restore letras**?

*Sucede que el segundo paso del punto 34 (añadir la letra ‘C’) que no se había añadido a la SA desaparece.*

1. ¿Qué sucede si se ejecuta el comando **git restore --staged letras**?

*Sucede que el último ‘add’ que había realizado (después de añadir la letra ‘b’)*

*se revierte y deja de estar en la SA.*

1. Para este apartado, volver al estado en el que se encuentra el repositorio justo detrás del punto 28.

Mostrar un listado del histórico de *commits*.

|  |
| --- |
|  |

1. Mostrar la información disponible del último *commit*. Explicar el resultado obtenido.

|  |
| --- |
| Nos dice el SHA del commit, que estamos en la rama ‘main’ y que ahora mismo el ‘HEAD’ apunta a esa rama y ese commit (el único), el autor y su mail, la fecha y la hora en la que hizo el commit, nos muestra el mensaje del commit, nos muestra las diferencias de las distintas versiones el archivo (después de diff); ‘new file’ indica que se ha creado un nuevo archivo, ‘mode’ nos indica con los últimos 3 dígitos (644) los permisos del archivo = lectura y escritura para el propietario y sólo de lectura para el grupo y otros), index nos informa sobre los SHA (000000 indica que es el primero, o mejor dicho: que no había un archivo anterior); ---/dev/null, indica que previamente se apuntaba a nada, otra vez: que no había un archivo previo, +++ b/letras = indica el nuevo archivo que hemos commiteado; @@ -0,0 +1 @@ nos habla de las diferencias (“-0,0” dice que no había líneas en la versión anterior, y “+1” que se ha añadido una línea); por último se nos indica precisamente qué línea se ha añadido representado con un signo ‘+’: = ‘a’. |

1. Etiquetar el último *commit* como v1.0.

|  |
| --- |
|  |

1. Listar las etiquetas definidas.

|  |
| --- |
|  |

1. Mostrar el nombre de las ramas existentes. Explicar el resultado obtenido.

|  |
| --- |
| A diferencia de antes, como ya hemos un commit ya nos aparece la rama ‘main’ acompañada de un asterisco y coloreada de verde para indicarnos que estamos en ella. |

1. Verificar la conectividad y validez de los objetos de la base de datos de Git. Explicar el resultado obtenido.

|  |
| --- |
| La primera línea nos dice que la verificación de directorios es correcta y se ha completado con éxito (‘done’); el ‘dangling blob’ (“blob colgante”) es un objeto de datos que existe en la base de datos de objetos de GIT pero que no está referido por ningún commit, árbol o etiqueta, es decir: aquellos objetos que entraron en la SA pero que fueron eliminados antes de haber hecho commit (tal como hemos hecho más arriba); el ‘dangling tree’ (“árbol colgante”) es lo mismo que lo anterior pero referido a una estructura de directorios.  + Estos objetos colgantes existen en la base de datos de objetos de GIT pero no tienen ninguna referencia active, estos objetos pueden perderse cuando GIT realice sus procesos de limpieza automática. |

1. Realizar un *commit* con el mensaje *Modificado el directorio* ***letras***.

|  |
| --- |
|  |

1. Crear un archivo vacío de nombre **vacio**. Añadirlo al SA. Enmendar el *commit* anterior para que añada este nuevo archivo y modifique el mensaje para que sea *Modificado el archivo* ***letras***.

|  |
| --- |
|  |

1. Mostrar el contenido del archivo **letras** en el *commit* actual y en el anterior.

|  |
| --- |
|  |

1. Modificar el contenido del archivo **letras** para que contenga las letras a y c, cada una en una línea, dejando una línea en blanco entre la a y la c.

Crear un archivo **numeros**, que contenga una línea con los números impares 1, 3 y 5 (separados por comas), y otra con los números pares 2, 4 y 6 (separados por comas).

Añadir ambos archivos al SA. Comprobar el estado del repositorio, en el formato largo y en el corto.

|  |
| --- |
| En la versión corta ahora también podemos ver la letra ‘A’ de ‘added’ = añadido / agregado. |

1. Confirmarlos con el mensaje *Modificado el archivo leras y añadido el archivo números*. Explicar el resultado del comando.

|  |
| --- |
| Nos dice la rama ‘main’ el SHA, el mensaje del commit, el númro de archivos cambiados, los tipos de los cambios (4 inserciones y una eliminación, la letra ‘b’) y los permisos del nuevo archive numeros. |

1. Mostrar el contenido del archivo números en el commit actual y en el anterior. Explicar el resultado obtenido.

|  |
| --- |
| El archive números no existía en el commit anterior, pore eso no muestra nada. |

1. Mostrar las diferencias del archivo letras en sus versiones del *commit* anterior y la del actual. Utilizar el operador **~**.

|  |
| --- |
|  |

1. Mostrar las diferencias entre el *commit* anterior y el actual. Utilizar el operador **^**.

|  |
| --- |
|  |

1. Modificar el archivo **numeros** para que tenga: en la primera línea, los números impares del 1 al 9, separados por comas; en la segunda, los números pares 2 y 4 separados por comas; la tercera línea en blanco y la cuarta línea, los números 30, 20 y 10 separados por comas. En un solo comando, añadir el archivo al SA y confirmarlo con el mensaje *Retocado el archivo numeros*.

|  |
| --- |
|  |

1. Mostrar el histórico de confirmaciones en formato de una línea por *commi*t , gráfico y para todas las ramas.

|  |
| --- |
|  |

1. Eliminar completamente el último *commit* y mostrar el histórico de confirmaciones en formato de una línea por *commit*, gráfico y para todas las ramas.

|  |
| --- |
|  |

1. Arreglar el árbol no referenciado del apartado anterior (su id es **4b825dc**) y volver a probar el comando **git fsck**.

|  |
| --- |
|  |

1. Verificar la conectividad y validez de los objetos de la base de datos de Git. Explicar el resultado obtenido.

|  |
| --- |
| Tal como se ve, se verifican los directorios y los objetos como ya hemos mencionado anteriormente, la novedad está en que ahora también lo ha hecho con los commits. |

1. Crear una rama de nombre **ramifica**. Mostrar las ramas existentes. Cambiarle el nombre a **bifurca**. Mostrar las ramas existentes.

|  |
| --- |
|  |

1. Crear un archivo llamado **simbolos** en **main**. Añadirle los caracteres **!**, **·** y **$** (uno por línea). Cambiar a **bifurca**, ¿existe el archivo símbolos? Explicar la respuesta.

|  |
| --- |
| Sí existe, porque hasta que no haga ningún commit las ramas no se separarán (pese a su nombre, ‘bifurca’ aún no se ha bifurcado), ambas apuntan al mismo commit que es el HEAD. |

1. En **bifurca**, añadir al SA el archivo **simbolos**. Comprobar el estado del repositorio. Listar el contenido del WD. Cambiar a **main**. Comprobar el estado del repositorio. Comparar los resultados obtenidos en una y otra ramas. Explicar la respuesta.

|  |
| --- |
| Al no haber hecho todavía ningún commit y al no haber desplazado el puntero de HEAD, es como si todavía estuviésemos en una única rama. |

1. En **main**, confirmar con el mensaje *Añadido el archivo simbolos*. Comprobar el estado del repositorio. Comparar los resultados obtenidos en una y otra ramas. Explicar la respuesta. Listar el contenido del WD.

|  |
| --- |
| En ambas al hacer git status apreciamos que está limpio, la diferencia fundamental es que al haber hecho commit en main podemos apreciar como simbolos aparece listado en esa rama pero no en ‘bifurca’. |

1. Cambiar a **bifurca**. Comprobar el estado del repositorio. Listar el contenido del WD. ¿Coincide el contenido del WD en una y otra rama? Explicar la respuesta.

|  |
| --- |
| Respondido en el ejercicio anterior. |

1. En la rama **bifurca**., crear el archivo **simbolos**. Añadirle, en líneas separadas, los caracteres **%**, **/** y **=**. Comprobar el estado del repositorio.

|  |
| --- |
|  |

1. Añadir **simbolos** al **SA**. Comprobar el estado del repositorio. Confirmar con el mensaje *Añadido el archivo símbolos en bifurca*.

|  |
| --- |
|  |

1. En **bifurca**, mostrar el histórico de *commits* (una línea por *commit* y gráfico) Cambiar a **main**. Mostrar el histórico de *commits*, ¿coinciden ambos históricos? Explicar la respuesta.

|  |
| --- |
| La única diferencia es a dónde apunta el puntero HEAD. |

1. En **main**, mostrar el histórico de *commits* de todas las ramas. Cambiar a **bifurca**. Mostrar el histórico de *commits*, de todas las ramas.

|  |
| --- |
| Respondido en el ejercicio anterior. |

1. Cambar a **main**. Crear un archivo de nombre **nuevo\_en\_main** que contenga una línea con el texto ***una línea***. Añadirlo al SA. Añadir al archivo recién creado el texto ***otra\_línea***. Mostrar el estado el simplificado repositorio. Mostrar el contenido del archivo **nuevo\_en\_main**. Explicar el resultado.

|  |
| --- |
| El status indica lo que hemos hecho: creamos un nuevo archivo (‘new file’) y lo añadimos a la SA (por eso sale en verde), y acto seguido lo modificamos pero sin añadirlo (por eso dice ‘modified’ = “modificado” y sale en rojo); eso no quita que el contenido si se haya modificado, tal como se aprecia cuando hacemos cat nuevo\_en\_main: |

1. En **main**, listar el contenido del WD. Reservar el trabajo en curso. Listar el contenido del WD. Mostrar el estado del repositorio. Mostrar la lista de reservas. Sacar el trabajo reservado al WD. Listar el contenido del WD. Mostrar el estado del repositorio. ¿Coincide el contenido del WD y el estado del repositorio cuando se ha hecho la reserva y cuando se deshace? Explicar la respuesta.

|  |
| --- |
| El estado coincide porque git stash mantiene el WD exactamente como yo lo guardé, tanto si había añadido o no el contenido a la SA[[1]](#footnote-1). |

1. Fusionar la rama **bifurca** en la rama **main**. ¿Qué sucede?, ¿qué habría que hacer ahora para no perder archivos? Ejecutar los comandos necesarios para que no se pierda el trabajo que hubiera en curso, sino que se guarde en el repositorio.

|  |
| --- |
|  |

1. Si la rama bifurca no se fusionó a **main** anteriormente, hacerlo ahora. ¿Qué sucede? Explicar el resultado del comando. Muestra el estado del repositorio. ¿Hay que hacer algo para que el repositorio no tenga “nada pendiente”? En caso afirmativo, hacerlo.

|  |
| --- |
| Hay un conflicto porque los archivos ‘simbolos’ de cada rama son distintos, si hacemos git status lo podemos apreciar:    Para resolver los conflictos debemos editar el archivo problemático, hacemos: nano símbolos =    La interfaz de ‘nano’ cambia automáticamente y se divide mostrándonos el contenido de ambas ramas separados por una línea de signos ‘=’; + yo en este archivo puedo hacer lo que quiera antes de hacer commit y el resultado de la fusión quedará exactamente como lo que yo edite, elijo preservar las aportaciones de ambos archivos y elimino las referencias automáticas de GIT:    Procedemos a hacer commit con el mensaje “Hemos hecho merge en ambas ramas juntado el contenido de ‘simbolos’”: |

1. Mostrar el histórico de confirmaciones del repositorio, en formato de un *commit* por línea, gráfico y para todas las ramas.

|  |
| --- |
|  |

1. Mostrar el histórico de confirmaciones del repositorio en los que ha sido modificado el archivo **simbolos**, en formato de un *commit* por línea, gráfico y para todas las ramas.

|  |
| --- |
|  |

1. El archivo **letras** ha sido confirmado tres veces. Mostrar en contenido en la versión que hay en el WD. Mostrar el contenido del archivo existente en el primer *commit*. Recuperar en el WD la versión que del mismo hay en el segundo *commit*. Mostrar el contenido del archivo letras en el WD. Mostrar el estado del repositorio.

|  |
| --- |
| Mostramos el historial de letras para movernos por sus commits:    Aprovechamos que habíamos etiquetado el primer commit y extraemos letras usando checkout y luego hacemos cat:    Extraemos letras del segundo commit:    Hacemos git status:    Al estar todavía en main y traer una versión anterior del archivo letras, el archivo viejo pasa directamente a la SA, en caso de que hiciéramos commit volveríamos al estado del segundo commit. |

1. Descartar completamente la versión recuperada del archivo letras, es decir, que el archivo vuelva a estar como antes de recuperar la versión anterior y no quede rastro de la versión recuperada. Mostrar el contenido del archivo letras. Explicar los pasos que se dan, es decir, los comandos ejecutados.

|  |
| --- |
| Volvemos al commit del HEAD = 4179ff5, haciendo checkout y hacemos cat:    También podríamos haber hecho: ***git checkout HEAD letras*** |

1. Crear un nuevo archivo de nombre **mar\_de\_dudas**, que contenga una línea con el texto ***No sé qué será de mi vida***. Mostrar el estado del repositorio. Crear un archivo **.gitignore** que contenga como texto únicamente el nombre del archivo recién creado. Mostrar el estado del repositorio. Explicar qué ha sucedido. Confirmar el trabajo que deba ser confirmado. Mostrar el histórico de confirmaciones del repositorio, en formato de un *commit* por línea, gráfico y para todas las ramas.

|  |
| --- |
| Al incluir el archivo ‘mar\_de\_dudas’ en el .gitignore, GIT deja hacerle un control de versiones, o lo que es lo mismo: lo ignora. |

1. Mostrar las diferencias entre el primer y el último *commit* del repositorio.

|  |
| --- |
| Aprovechamos el tag que le pusimos a nuestro primer commit y hacemos un diff entre el tag y el HEAD: |

1. Instalar la [herramienta gráfica **Meld**](http://meldmerge.org/) para mostrar diferencias y asistir en los procesos de fusionado (*merge*) de *commits*.

|  |
| --- |
| Vamos al sitio: <https://meldmerge.org/#features>, y nos lo descargamos: |

1. Configurar **Meld** como herramienta gráfica en Git para mostrar diferencias, y configurarla también para asistir en los procesos de fusionado (*merge*) de *commits*.

|  |
| --- |
| Nos vamos a nuestro usuario y abrimos con nano el archivo .*gitconfig*:    Añadimos las líneas que se remarcan a continuación: |

1. Mostrar las diferencias entre el primer y el último *commit* del repositorio usando la herramienta gráfica recién configurada.

|  |
| --- |
| Se van abriendo uno a uno los archivos y se ven como a continuación, por ejemplo ‘letras’:    Así se ve ‘simbolos’: |

1. Instalar la [herramienta gráfica **KDiff3**](https://kdiff3.sourceforge.net/) para mostrar diferencias y asistir en los procesos de fusionado (*merge*) de *commits*.

|  |
| --- |
| Vamos al sitio: <https://sourceforge.net/projects/kdiff3/> y nos descargamos el instalador:    Se nos abre automáticamente un README: |

1. Configurar **KDiff3** como herramienta gráfica en Git para mostrar diferencias, y configurarla también para asistir en los procesos de fusionado (*merge*) de *commits*.

|  |
| --- |
| Repetimos el mismo modus operandi que con Meld, y abrimos con nano el archivo .*gitconfig* para añadirle las siguientes líneas (en sustitución de las anteriores de Meld, para evitar conflictos, aunque podrían estar ambas siempre que le indiquemos a la configuración cual preferimos usar): |

1. Mostrar las diferencias entre el primer y el último *commit* del repositorio usando la herramienta gráfica recién configurada.

|  |
| --- |
| Así se ve el .gitignore:    Así se ve ‘nuevo\_en\_main’:    Así se ve ‘números’: |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Manejo de Git con GitHub**

De cada acción que se realice en GitHub se debe hacer una captura de la pantalla.

1. En GitHub, crear un repositorio público y sin archivos, de nombre **actividad**.

|  |
| --- |
|  |

1. Comprobar las conexiones que tiene el repositorio local con repositorios remotos. Conectar el repositorio local con el remoto antes creado. Volver a comprobar las conexiones que tiene el repositorio local con repositos remotos.

|  |
| --- |
|  |

1. Descargar el contenido de la rama **main** del repositorio remoto a local. ¿Se descarga? Explicar la respuesta. Si no se pudo realizar la descarga, arreglarlo para que se pueda.

|  |
| --- |
| No se puede porque aunque hayamos conectado nuestro repositorio local con el remoto aún no le hemos subido ningún contenido, deberíamos hacer antes un git push origin main:    + Comprobamos que se actualiza el repositorio en GitHub:    Probamos hacer una vez más git pull origin main:    La frase “Already up to date” nos informa que no hay cambios nuevos en la rama ‘main’ del repositorio remoto, le agregamos un README para poder realizar la descarga: |

1. Mostrar el histórico de confirmaciones del repositorio (un *commit* por línea, en modo gráfico y para todas las ramas). Mostrar las referencias existentes en el repositorio remoto.

|  |
| --- |
| Las referencias del repositorio remoto están escritas en Naranja y precedidas por la palabra ‘origin’. |

1. Subir la rama **bifurca** a remoto. Explicar el resultado devuelto por el comando y la acción realizada. Mostrar las referencias existentes en el repositorio remoto.

|  |
| --- |
| Nos indica en la primera línea que no se han transferido objetos nuevos, la palabra entre paréntesis “delta” de los campos subsiguientes hace referencia a los ‘cambios incrementales’ que nos informan sobre la diferencia entre 2 versiones de un archivo o conjunto de archivos: + y nos dice que no han habido cambios de este tipo que se necesitarán enviar, o que se reutilizan objetos existentes en el remoto (“reused”), y que tampoco se reutilizaron paquetes de objetos (pack files) en el remote.  + Todos las líneas que se inician con “remote” son mensaje que se refieren a GitHub: y nos dice que hemos creado una ‘pull request’ para la rama ‘bifurca’ desde el repositorio remote, la URL para hacerlo, y la URL desde donde se hizo ‘push’; la última línea indica que hemos creado una rama ‘bifurca’ en el remote que está sincronizada con la rama ‘bifurca’ en Nuestro repositorio local.  + Comprobamos que esto es así en la interfaz de GitHub: |

1. Crear un archivo de nombre **comun**, tanto en local como en remoto. En local contendrá el texto ***¿Y ahora qué, remoto?*** En remoto contendrá el texto ***¿Y ahora qué, local?*** Añadir cada archivo al repositorio donde fue creado. Sincronizar ambos repositorios para que contengan la misma información. Explicar todos los pasos y el resultado de cada acción usando para esto los comandos que se vienen usando anteriormente: **git log**, **git status**…

En GitHub:

|  |
| --- |
| Creación de comun en GitHub: |

En local:

|  |
| --- |
| Creación de comun en Nuestro repositrio local, + commit, + git lg:    Hacemos git push origin main desde local:    No nos deja, porque el repositorio remote contiene trabajo que nosotros todavía no temenos en local, nos sugiere hacer git fetch primero, o en todo caso hacer git pull, probamos a hacer git fetch:    Al hacer fetch vemos que delta ya no está a ‘0’, y marca ‘1’; + también se nos indica en la última línea que en remote se ha hecho un commit y nos da el SHA de ese commit.  + Cuando hacemos git lg Podemos ver las bifurcaciones entre local y remote, y como ambas tienen distintos commits.  +Nos movemos con checkout al commit de remoto (5c96a99) para ver el contenido de ‘comun’:    = Es el contenido que hemos realizado en GitHub; volvemos a nuestra rama principal en local e intentamos hacer un git pull:      Abrimos ‘comun’ con nano:    Lo editamos, lo añadimos a la SA, hacemos commit, y listamos los commits con git lg:      Hacemos git push para sincronizarlo con remoto y comprobamos los cambios en GitHub: |

1. Mostrar el histórico de *commits* del repositorio local (una línea por commit, modo gráfico, todas las ramas) y las referencias existentes en el repositorio remoto.

|  |
| --- |
| Vuelven a estar sincronizados. |

**Trabajo con un nuevo repositorio clonaactividad**

1. Crear un nuevo repositorio de nombre **clonactividad**, como una clonación del repositorio **LOCAL** de nombre **actividad** que se está usando en la práctica. Entra en el WD del nuevo repositorio **clonactividad**, muestra el estado del repositorio, el histórico de confirmaciones (una línea por *commit*, modo gráfico, todas las ramas) y los servidores remotos a los que está conectado.

|  |
| --- |
|  |

1. Crear un nuevo archivo **me\_revierto** con el texto ***Me voy a revertir***. Confirmarlo. Cambiarle el nombre a **revertiendo**. Confirmarlo. A partir de aquí, comprobar el estado del repositorio tras la ejecución de cada comando. Cambiar el nombre del archivo **me\_revierto** por **revertiendo**. Añadir el cambio al SA. Confirmarlo. Explicar el resultado de las operaciones.

|  |
| --- |
|  |

1. Revertir el último *commit* y mostrar el histórico de *commits*. Explicar las operaciones usando para esto los comandos que se vienen usando anteriormente: **git log**, **git status**…

|  |
| --- |
| Hacemos:    Se nos abre un editor de Vim, cambiamos el nombre del commit y salimos guardando la edición:    Hacemos git lg:    Git revert, revierte el commit que le indicamos pero lo hace creando un nuevo commit. |

1. Realizar una “recogida de basura” en el repositorio.

|  |
| --- |
|  |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**ANEXO I: Listado de comandos de Git que se pueden necesitar en la práctica** (algunos no es imprescindible usarlos)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | **git init** | | **git clone** | | git help | | **git config** | | **git status** | | **git log** | | **git shortlog** | | git reflog | | git whatchanged | | **git show** | | |  | | --- | | **git add** | | **git rm** | | **git mv** | | git clean | | **git commit** | | git commit-tree | | **git diff** | | git difftool | | git tag | | **git branch** | | |  | | --- | | **git checkout** | | **git switch** | | git stash | | **git merge** | | git mergetool | | **git rebase** | | **git remote** | | **git restore** | | **git revert** | | **git reset** | | |  | | --- | | **git push** | | **git pull** | | **git fetch** | | git ls-files | | git ls-tree | | git ls-remote | | git rev-list | | git hash-object | | git fsck | | git gc | |

**ANEXO II: Listado de comandos de Linux que se pueden necesitar en la práctica** (algunos no es imprescindible usarlos)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **awk** | **cd** | **cat** | **cut** | **echo** | **head** | **ls** | **mkdir** |
| **mv** | **pwd** | **rm** | **sort** | **touch** | **wc** |  |  |

1. En este ejercicio hay incongruencias que se aprecian en el nombre del commit, eso es porque al pegar las capturas de pantalla se me desplazó el ejercicio y no lo vi avanzando con el siguiente (el 67), me di cuenta que me faltaba más adelante cuando iba por el ejercicio 70. Intenté enmendarlo volviendo a añadir una línea en “nuevo\_en\_main”, dado que entendía que el ejercicio sólo pretendía aplicar *git stash*. [↑](#footnote-ref-1)