**Consideraciones iniciales:**

* Si bien Git es una herramienta de control de versiones, donde se lleva un seguimiento ordenado de todo el código fuente que se crea o modifica, a los efectos prácticos y para este caso en particular, la presente guía se ha redactado de manera totalmente agnóstica a un lenguaje en particular, utilizando en lugar de código fuente, textos de autores de la literatura universal como elementos de trabajo.
* Simularemos que somos parte de un equipo de traductores que trabaja para una editorial.
* Realizaremos parte de nuestra práctica con Github, pero no veremos todas las funcionalidades que posee esa herramienta, ya que nos tomaría mucho tiempo del que no disponemos.
* Si bien CMD es lo que más pueden conocer, el conocimiento de comandos de consola Linux se hará necesario cuando se encuentren trabajando para un proyecto real. Git fue desarrollado por Linus Torvalds (el creador del kernel de GNU/Linux) por lo que es mucho más sencillo usar Git con Git Bash que con Git CMD.
* Finalmente, se agrega una sección de ejercicios opcionales, con temas igualmente importantes pero que no se considerarán en las evaluaciones.

**¿Qué comandos de CMD son necesarios para moverse en la consola?**

* **dir:** lista el contenido del directorio. Con el modificador **/adh** se listan los directorios ocultos
* **cd:** cambiar de directorio.
* **Type <nombre\_de\_archivo>:** muestra en consola el contenido de un archivo

**¿Qué comandos de Linux son necesarios para moverse en la consola?**

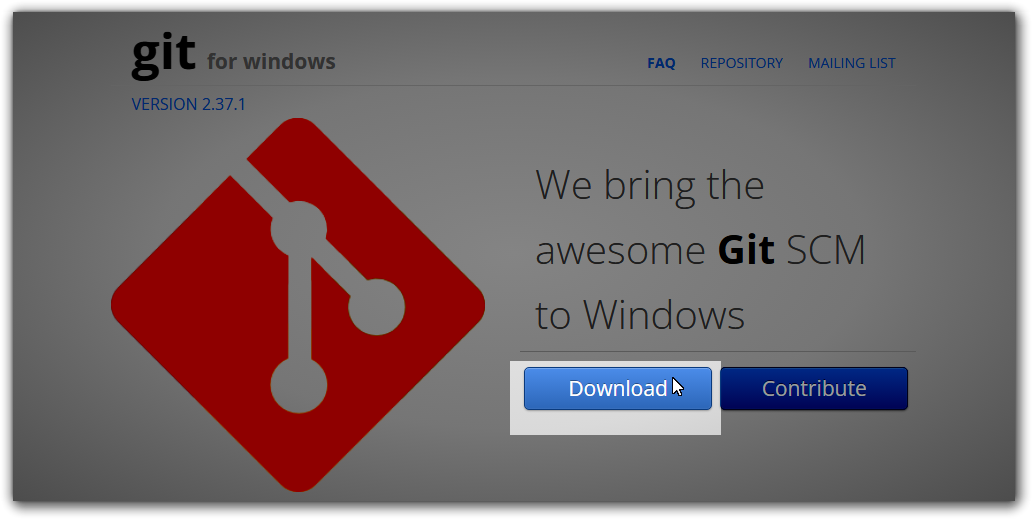
* **ls:** listar el contenido del directorio. Con el modificador **-la** se listan todos los archivos ocultos
* **pwd:** averiguar en qué directorio estamos parados
* **cd:** cambiar de directorio. A diferencia del CMD, cd es case-sensitive y emplea el espacio como separador de argumentos.
* **cat <nombre\_de\_archivo>:** muestra en consola el contenido de un archivo.
* **mkdir <nombre\_de\_directorio>:** crea un directorio en la ubicación en la que se invoca.

**¿Qué comandos GIT son necesarios para realizar la practica?**

* **git init:** inicializa el repositorio en el directorio especificado, tomándolo como raíz.
* **git status:** permite chequear el estado de nuestro repositorio (en que rama estamos, si se realizaron commits, los archivos que hemos agregado para commitear, etc)
* **git add [<nombre\_de archivo> | modificador]**: agrega archivos a la lista del próximo commit. El modificador **-A** indica que se desea agregar todos los archivos que registren cambios. Se pueden agregar individualmente indicando el nombre del archivo.
* **git commit [modificador]:** prepara y envía los archivos al repositorio, agregando el N° de versión y el tag con el que se identifican los cambios. El modificador -m “” permite agregar un tag a todos los cambios que se aplican. Es muy saludable, tanto para el desarrollador como para el equipo el identificar con un tag los cambios que se realizaron.
* **git branch** **[modificador]:** permite cambiar de rama, según las tengamos definidas. El modificador **-M main** nos ubica en la rama **main**.
* **git checkout <rama>**: establece como HEAD la **<rama>** que se le indica. Permite pasar de una rama a otra. Si pasamos **-b <rama>** creamos una nueva rama y nos posicionamos en ella en un único paso.
* **git remote [-v | add <rama> <ruta.git> | remove <rama>]:** nos permite trabajar con los repositorios remotos. Se debe destacar que Git asocia la rama con la ruta. Nunca vamos a tener una ruta sin asociar a una rama, aunque si podemos tener ramas sin rutas.
* **git pull <remote><rama>**: descarga el contenido del repositorio remoto, chequeando los cambios y uninéndolos

**EJERCICIO 1:**

Descargar GIT for Windows desde <https://gitforwindows.org/> e instalarlo con todas las opciones por defecto.



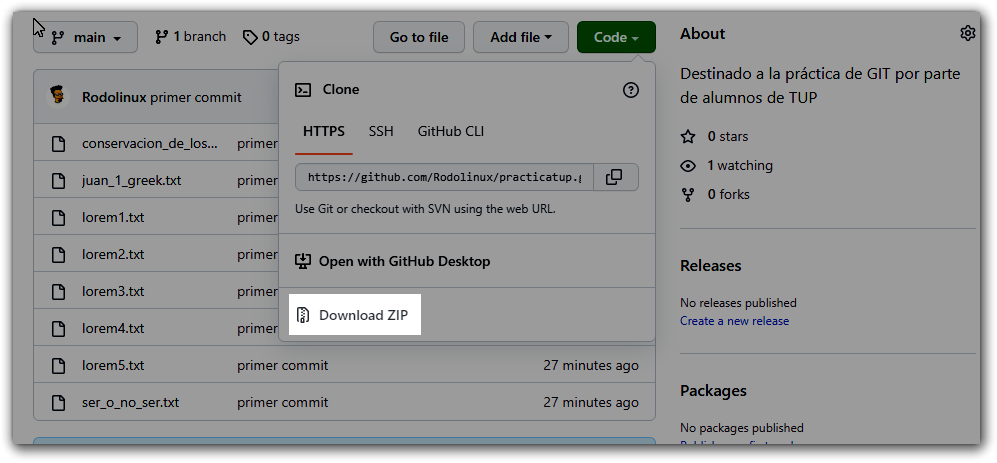
**EJERCICIO 2:**

Ingresar a github.com y crear una cuenta de usuario, seteando un repositorio como público con el nombre **“practicatup”**

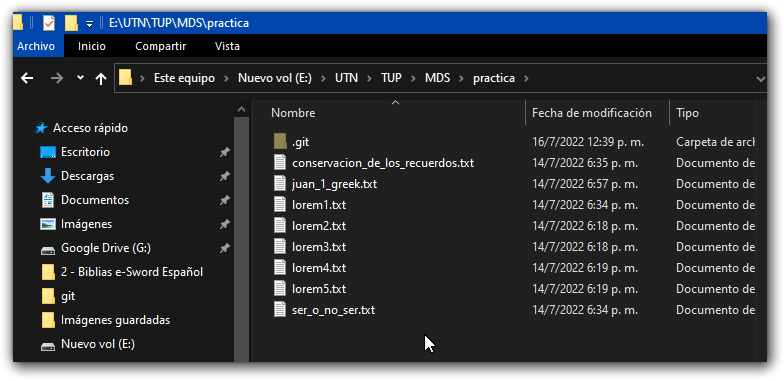


**EJERCICIO 3:**

* Descargar el archivo de práctica **“prácticagit.zip”** del url <https://github.com/Rodolinux/practicatup>

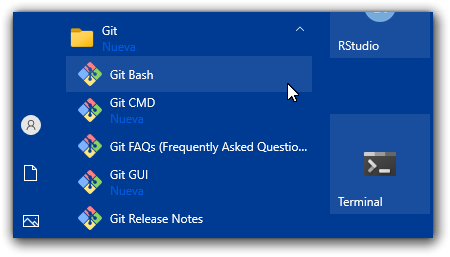


* descomprimirlo en un directorio en ruta fácil de acceder (en lo posible en una ruta sin espacios)



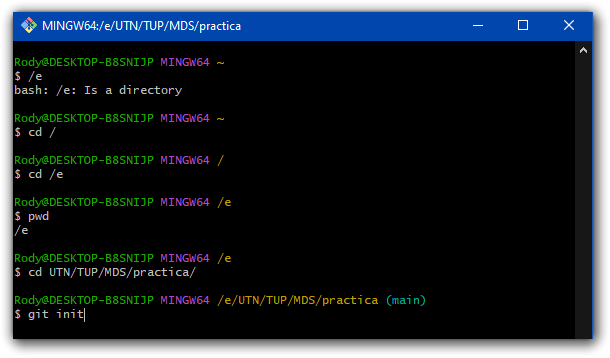
**EJERCICIO 4:**

* Iniciar Git Bash preferentemente o GIT CMD

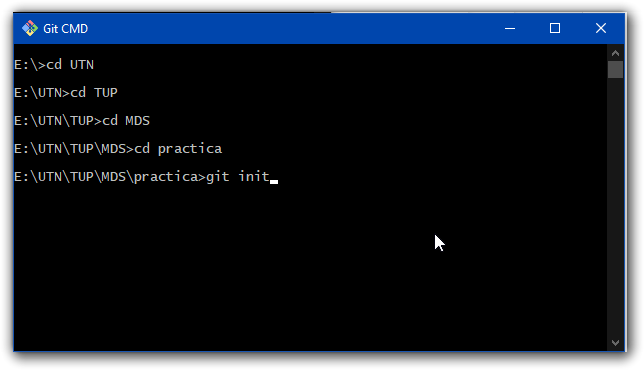


* Definir la configuración global de GIT en nuestro equipo. Si no hacemos esto no podremos hacer ningún commit.
  + git config --global user.name “Juan Perez”
  + git config --global user.email “juan@perez.org”
* Acceder al directorio donde están los archivos con los que vamos a trabajar e inicializar el repositorio con **“git init”**

**Git bash:**

****

**Git CMD**

****

* ¿Puede citar las diferencias entre Git Bash y Git CMD?
* Listar el directorio y ver si ha habido algún cambio. ¿Hay alguna diferencia al listar con **ls** y con **ls -la**?
* Acceder al directorio .git con **“cd .git”** e inspeccionar su contenido. ¿Nota algo especial al final de la línea?
* inspeccionar el contenido de los archivos con **cat <nombre\_de\_archivo>**.

Git Bash tiene algo muy útil que es el histórico de los comandos que ingresamos. Presionando tecla arriba podemos ver todos los comandos que hemos ingresado. Este histórico es persistente, lo que significa que permanece luego de que cerremos la consola.

**EJERCICIO 6:**

Abrir el archivo **conservacion\_de\_recuerdos.txt**, cortar el texto “**(Julio Cortázar)”** que se encuentra a la derecha del título del cuento, y pegarlo al final del documento. Guardar el archivo.

* **git commit -m “primer commit”**

¿Qué resultado nos da? ¿Se realizó o no el commit? ¿Qué nos faltó?

**EJERCICIO 7:**

* Verificar el estado de nuestro repositorio.
  + **git status**

¿Qué información podemos observar? ¿En qué **rama** estamos? ¿En qué estado se encuentran nuestros archivos? ¿Hubo algún **commit** registrado?

**master** es la rama que Git crea por defecto cuando recién empezamos con un git init. Se puede cambiar sin problemas, es solo un nombre.

* **Staging** es el lugar donde van los archivos a ser commiteados. Debemos incorporar al **staging** los archivos destinados al primer commit mediante el comando **git add**. En nuestro caso en particular agregaremos todos.
  + **git add -A**
* Verificar nuevamente el estado de nuestro repositorio
  + **git status**

¿Qué información podemos observar? ¿En qué **rama** estamos? ¿En qué estado se encuentran nuestros archivos? ¿Hubo algún commit registrado?

* Veamos el log de cambios que lleva git con el comando **git log**

**EJERCICIO 8:**

**master** es la rama que Git crea por defecto cuando recién empezamos con un **git init**. Se puede cambiar sin problemas, es solo un nombre. Lo que indica la rama activa es HEAD.

* Vamos a cambiar el nombre de la rama **master** por **main**
  + **git branch -M main**

Si usamos Git Bash notaremos el cambio enseguida mientras que con Git CMD no se percibe nada.

Para listar todas las ramas disponibles solo debemos escribir git branch sin modificadores

* **git branch**

¿Qué sucede si empleamos el modificador **-v** con **git branch**? ¿Qué cree que significa el **\*** que precede a la rama **main**?

* **git branch -v**

Git branch crea ramas, pero no las activa

**EJERCICIO 9:**

Aplicar los cambios almacenados temporalmente en el **staging** al repositorio.

* **git commit -m “primer commit”**

Si hicimos todo bien, los cambios se aplicaron a nuestro repositorio local en la rama **“main”**.

Cada commit siempre registra:

1. Hash del commit. ¿Por qué podemos suponer que es un hash de 40 caracteres y no un simple número?
2. Nombre y correo del autor
3. Fecha. La fecha y hora al momento de realizar el commit
4. Mensaje del commit. Es lo que se escribe luego del -m

**EJERCICIO 10:**

Ejecutemos **git log** para ver los cambios aplicados

* **git log**

Ahora agreguemos el modificador **-p** a **git log**

* **git log -p**

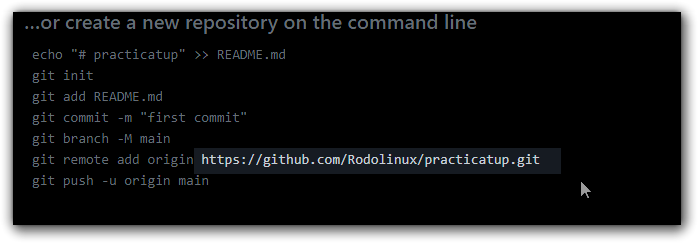
¿Qué nos muestra **git log -p** ahora? Con **tecla arriba** y **tecla abajo** podemos recorrer el texto que se nos muestra. Se sale presionando **q**

**EJERCICIO 11:**

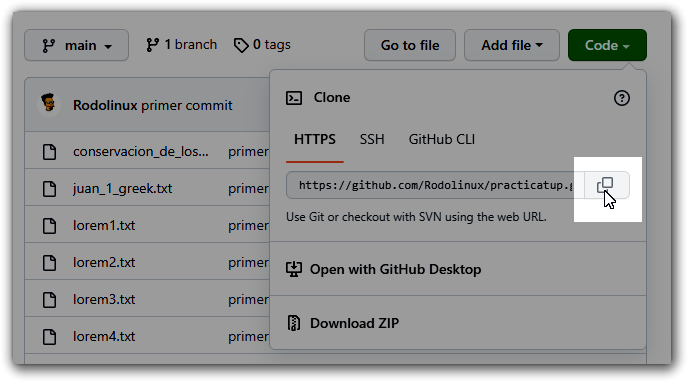
Hasta ahora solo hemos trabajado en nuestro repositorio local pero como Git nos permite compartir nuestro trabajo con otros, vamos a agregar un repositorio remoto a nuestro git local de manera de poder enviar nuestro trabajo desde consola, sin tener que pensar en seleccionar los archivos a enviar, crear un comprimido y enviarlo por correo o subirlo a un FTP. Git ya dispone de una función para que podamos hacerlo desde la misma consola.

Debido a que creamos una repositorio en github, es allí donde alojaremos nuestro trabajo, por lo que le indicaremos a nuestro Git que trabajaremos con Github.

Para obtener la dirección en Github, la primera vez la tendremos disponible de esta manera



Después de haber subido código, esta pantalla inicial desaparecerá y se podrá obtener la dirección Github de la siguiente forma:



Con la dirección git copiada, debemos decirle a nuestro Git local en donde estará el Git remoto con el que trabajaremos

* **git remote add origin** [**https://github.com/**](https://github.com/)**<nombre\_de\_usuario\_elegido>/practicatup2023.git**

Verificar el estado de las conexiones remotas

* **git remote -v**

¿Por qué cree que hay dos entradas para remote, a pesar que escribimos una sola? ¿Qué es **push** y **fetch**?

**EJERCICIO 12:**

transferir los cambios desde el repositorio local al remoto en Github

* **git push -u origin main**

¿Que nos informa en consola?

**origin**, al igual que **main** y **master** son branches que se crean con Git. En el caso de **origin**, es la rama que Git crea para identificar la fuente remota. En este caso, **origin** fue creado por el Git de Github.

Es importante comprender que no hay una conexión persistente entre el git local y el remoto. Todo acontece por medio de los comandos que escribimos por consola.

**EJERCICIO 13:**

Hasta ahora hemos trabajado con una sola rama activa, sea **master** o **main.** Supongamos que se nos encarga traducir al inglés el soliloquio de Hamlet ubicado en el archivo **ser\_o\_no\_ser.txt**, mientras otro miembro del equipo también lo traducirá, y luego el editor en jefe seleccionará la mejor traducción. ¿Como hacemos para que el trabajo que haremos no pise al del otro traductor o viceversa? Usaremos branches o ramas, que permitirán mantener los contextos separados.

Solo podemos trabajar en una rama por vez, esto es, la rama activa.

Crear una nueva rama llamada **traduccion-ingles:**

* **git branch traduccion-ingles**

Verificar la rama que creamos con git branch -v

* **git branch -v**

¿Cuál es la rama activa?

**EJERCICIO 14:**

Cambiar de la rama **main** a **traduccion-ingles**

* **git checkout traduccion-ingles** (no llevan acento ni traduccion ni ingles)

Esto provoca que:

* **traduccion-ingles** sea la rama HEAD vigente
* Se reemplazan los archivos en el directorio de trabajo para que coincidan exactamente con la revisión en la que se encuentra en la rama **traduccion-ingles**.

Esto se puede corroborar con git status:

* **git status**

Todos los cambios que apliquemos sobre los archivos se mantendrán en la rama HEAD.

**EJERCICIO 15:**

* Verificar que estamos en la rama **traduccion-ingles**
* Con el explorador de archivos de Windows nos ubicamos en la carpeta en donde tenemos nuestro directorio de trabajo. Abrimos el archivo **ser\_o\_no\_ser.txt**  y traducimos con Google Translate su contenido. Agregamos la traducción en inglés a continuación del texto original en castellano. Guardamos el archivo con el mismo nombre, **ser\_o\_no\_ser.txt**
* Ejecutar:
  + **git status** para ver el estado de nuestro directorio de trabajo
  + **git add -A** para colocar en staging los cambios en **ser\_o\_no\_ser.txt**
  + **git commit -m “traducción al inglés del soliloquio”** para commitear el cambio en la rama **traduccion-ingles**
* Verificamos abriendo desde el explorador de archivos el contenido de ser\_o\_no\_ser.txt. Debería estar en inglés.
* Ejecutar:
  + **git checkout main** para cambiar a la rama **main**.
  + **git status** para verificar que efectivamente estamos en **main**
  + Con el explorador de archivos abrimos el archivo **ser\_o\_no\_ser.txt**. Debería estar español sin el texto agregado

Mas allá que el explorador de Windows no muestre ningún cambio aparente, Git actúa como un control de versiones efectivo, separando los contextos de desarrollo. Podemos tener tantas ramas o branches como queramos.

**EJERCICIO 16:**

Si bien es útil trabajar con ramas, en algún momento debemos unir nuestro trabajo paralelo para mejorar el existente. Para esto se emplea **git merge.**

* **Ejecutar:**
  + **checkout main ,** únicamente si estamos en la otra rama
  + **merge traduccion-ingles**

Deberíamos obtener una salida como esta:

*Rody@*DESKTOP-B8SNIJP *MINGW64 /e/UTN/TUP/MDS/practica (main)*

*$ git merge traduccion-ingles*

*Updating 52566ef..31db569*

*Fast-forward*

*ser\_o\_no\_ser.txt | 47 ++++++++++++++++++++++++++++++++++++++++++++++-*

*1 file changed, 46 insertions(+), 1 deletion(-)*

Al verificar el archivo **ser\_o\_no\_ser.txt** deberíamos poder ver los textos en español e inglés

**EJERCICIO 17:**

Hasta ahora vimos la creación de un proyecto desde cero en Git, inicializando el repositorio, agregando los archivos, haciendo commit, pusheando nuestros cambios a nuestro repositorio. ¿Hay alguna forma de poder replicar un repositorio de manera más rápida que realizar lo anteriormente citado? Si, con **git clone.**

* Vamos a clonar nuestro repositorio en un nuevo directorio. Podemos hacerlo sobre el vigente, pero Git asumirá que es un subdirectorio en el proyecto actual y le aplicará todo lo que hemos venido trabajando.
* Ejecutar en la consola:
  + **cd ..** . Esto hace que subamos un nivel en la estructura de directorios. Podemos corroborarlo con **ls** o **dir**
  + **git clone** [**https://github.com/Rodolinux/practicatup**](https://github.com/Rodolinux/practicatup) **practup2023** . Este comando realiza una copia del repositorio **practicatup** y lo almacena en la carpeta destino **practup2023**. Si no indicamos la carpeta destino, pueden suceder dos cosas:
    - Que git nos avise que ya existe una carpeta con el mismo nombre si es que así la creamos
    - Que cree la carpeta y que nos muestre algo como esto:

*$ git clone https://github.com/Rodolinux/practicatup*

*Cloning into '/practicatup'...*

*remote: Enumerating objects: 10, done.*

*remote: Counting objects: 100% (10/10), done.*

*remote: Compressing objects: 100% (10/10), done.*

*remote: Total 10 (delta 0), reused 10 (delta 0), pack-reused 0*

*Receiving objects: 100% (10/10), 4.97 KiB | 4.97 MiB/s, done.*

* **cd practup2023** Ingresamos a la carpeta.
* **git status.** ¿Qué vemos de diferente con el proyecto anterior?
* **git remote -v**. ¿Tenemos rutas remotas?
* **git branch -v**. ¿Qué ramas tenemos activas?

EJERCICIO 18

* Si bien clonamos el repositorio público en una carpeta, solo tenemos la rama principal.

¿Cómo hacemos para obtener información del repositorio remoto sin afectar los archivos del repositorio local? Usando **git fetch <remoto><rama>**

* Ejecutemos:
  + **git fetch origin traduccion-ingles**

¿Qué efecto tiene **git fetch**? ¿Han cambiado nuestros archivos locales?

* El jefe nos avisa que hubo modificaciones en la rama traduccion-ingles. En función de esto, es tarea nuestra ponernos al día con los cambios. ¿Que habrá cambiado? ¿Cómo actualizamos esa rama? Con **git pull <remoto> <rama>**
  + **git pull origin traduccion-ingles**

Podríamos decir que **git pull** combina **git fetch** con **git merge**

**EJERCICIOS OPCIONALES:**

**gitignore**

Algunas veces es necesario definir qué archivos vamos a ignorar en los commits ya que no son necesarios. Por lo general son archivos de control de los diferentes IDEs que usemos y que solo tienen valor local y que no son necesarios compartir, y en otros casos por seguridad, como por ejemplo credenciales de acceso a bases de datos que se almacenan en texto plano en el proyecto. Se puede consultar más información al respecto en https://git-scm.com/docs/gitignore

A los efectos de la siguiente práctica, vamos a ignorar los siguientes archivos: (lorem2.txt, lorem3.txt, lorem4.txt y lorem5.txt)

* Crear un archivo de texto e incorporar los archivos indicados, uno debajo del otro. Puede usar el Bloc de Notas, Notepad+, VS Code o el editor de texto de su preferencia.
* Guardarlo como **.gitignore** en el directorio raíz del proyecto. El archivo no lleva extensión y comienza con un punto. En Linux significa que el archivo o directorio está oculto.
* Ejecutar **git add .gitignore** para incluir el archivo

A partir del siguiente commit, se ignorarán los archivos indicados en **.gitignore**.

**stash**

Cuando nos encontramos en el medio de un trabajo y se nos asigna una tarea urgente. ¿Qué hacer? ¿Commitear el trabajo por la mitad para no perderlo y para pasar a otra cosa? ¿Acaso git tiene alguna herramienta que nos permita guardarlo en un lugar seguro para luego retomarlo donde lo dejamos?

* **git stash:** guarda en un espacio temporal nuestro trabajo, dejando el directorio de trabajo limpio (con **git status** podemos comprobarlo)y listo para recibir lo que sea.
* **git stash list**: nos proporciona la lista de stashes que tenemos almacenados.
* Para recuperar el trabajo podemos usar:
  + **git stash pop**: nos entrega el último de la pila
  + **git stash apply <stashname>**: nos permite seleccionar qué stash queremos recuperar.

**git stash** nos permite “salvar las papas”, actuando como un portapapeles local, guardando temporalmente en nuestro repositorio local el trabajo a medio hacer y dejando el directorio de trabajo limpio.

Es útil para respaldar nuestro trabajo antes de:

* cambiar a una rama diferente
* hacer pull de cambios remotos
* hacer un merge con una rama

**Borrar ramas remotas**

Con **git push <remoto> --delete <rama\_a\_borrar>** se borran ramas remotas.

Bibliografía sugerida:

* GÜNTHER, T. Learn Version Control with Git\_ A step-by-step course for the complete beginner.