	<p align="center">UNIVERSIDAD DON BOSCO FACULTAD DE INGENIERIA, ESCUELA DE COMPUTACION</p>
<p align="center">Ciclo II</p>	<p align="center">DISEÑO Y PROGRAMACION DE SOFTWARE MULTIPLATAFORMA</p> <p align="center">Guía de Laboratorio No. 1</p> <p align="center">“Versionando con Git”</p>

I. RESULTADOS DE APRENDIZAJE

Que el estudiante:

- Configure el entorno de trabajo para poder iniciar a versionar los proyectos
- Utilice Git Bash para crear repositorios locales y llevarlos a repositorios remotos

II. INTRODUCCIÓN

SISTEMA DE CONTROL DE VERSIONES

Git, es un software de control de versiones diseñado por **Linus Torvalds**. La pregunta es **¿qué es control de versiones?** Pues bien, se define como control de versiones a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo es decir a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración, y para los que aún no les queda claro del todo, control de versiones es lo que se hace al momento de estar desarrollando un software o una página web.

Exactamente es eso que haces cuando subes y actualizas tu código en la nube, o le añades alguna parte o simplemente le editas cosas que no funcionan como deberían o al menos no como tú esperarías.

Y, entonces **¿a que le llamamos sistema de control de versiones?** Muy sencillo, son todas las herramientas que nos permiten hacer todas esas modificaciones antes mencionadas en nuestro código y hacen que sea más fácil la administración de las distintas versiones de cada producto desarrollado; es decir Git.

GIT

Git fue creado pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente, es decir Git nos proporciona las herramientas para desarrollar un trabajo en equipo de manera inteligente y rápida y por trabajo nos referimos a algún software o página que implique código el cual necesitemos hacerlo con un grupo de personas.

Algunas de las características más importantes de Git son:

- Rapidez en la gestión de ramas, debido a que Git nos dice que un cambio será fusionado mucho más frecuentemente de lo que se escribe originalmente.
- Gestión distribuida; Los cambios se importan como ramas adicionales y pueden ser fusionados de la misma manera como se hace en la rama local.
- Gestión eficiente de proyectos grandes.
- Realmacenamiento periódico en paquetes.

ORDENES BÁSICAS

Iniciar un repositorio vacío en una carpeta específica.

```
git init
```

Añadir un archivo específico.

```
git add "nombre_de_archivo"
```

Añadir todos los archivos del directorio

```
git add .
```

Confirmar los cambios realizados. El "mensaje" generalmente se usa para asociar al commit una breve descripción de los cambios realizados.

```
git commit -am "mensaje"
```

Revertir el commit identificado por "hash_commit"

```
git revert "hash_commit"
```

Subir la rama(branch) "nombre_rama" al servidor remoto.

```
git push origin "nombre_rama"
```

Mostrar el estado actual de la rama(branch), como los cambios que hay sin hacer commit.

```
git status
```

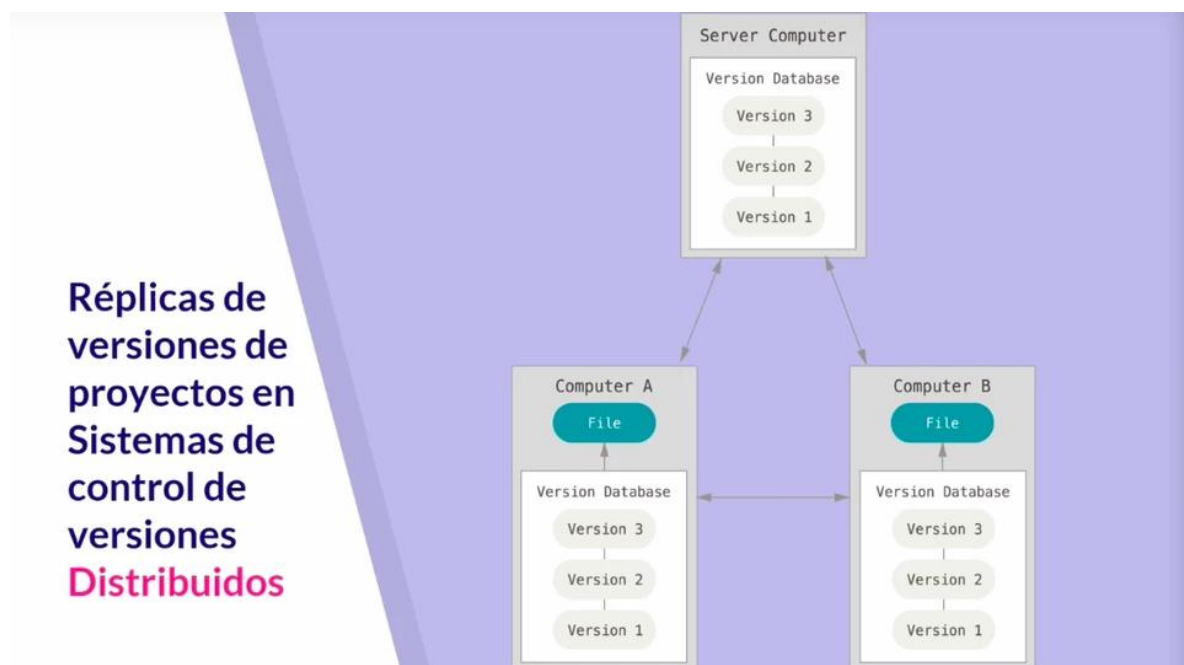
Sistemas de Control de Versiones Distribuidos

Los Sistemas de Control de Versiones Centralizados (VCS), como por ejemplo Subversion, que es una herramienta en la que se ha confiado para albergar el histórico de revisión de versiones, es un punto centralizado, lo cual puede llegar a suponer una merma de trabajo si perdemos la conectividad de la red.

Los Sistemas de Control de Versiones Distribuidos (DVCS) salvan este problema. Algunos ejemplos de sistemas distribuidos, aparte de **Git**, son **Mercurial**, **Bazaar** o **Darcs**. En este tipo de herramientas, los clientes replican completamente el repositorio.

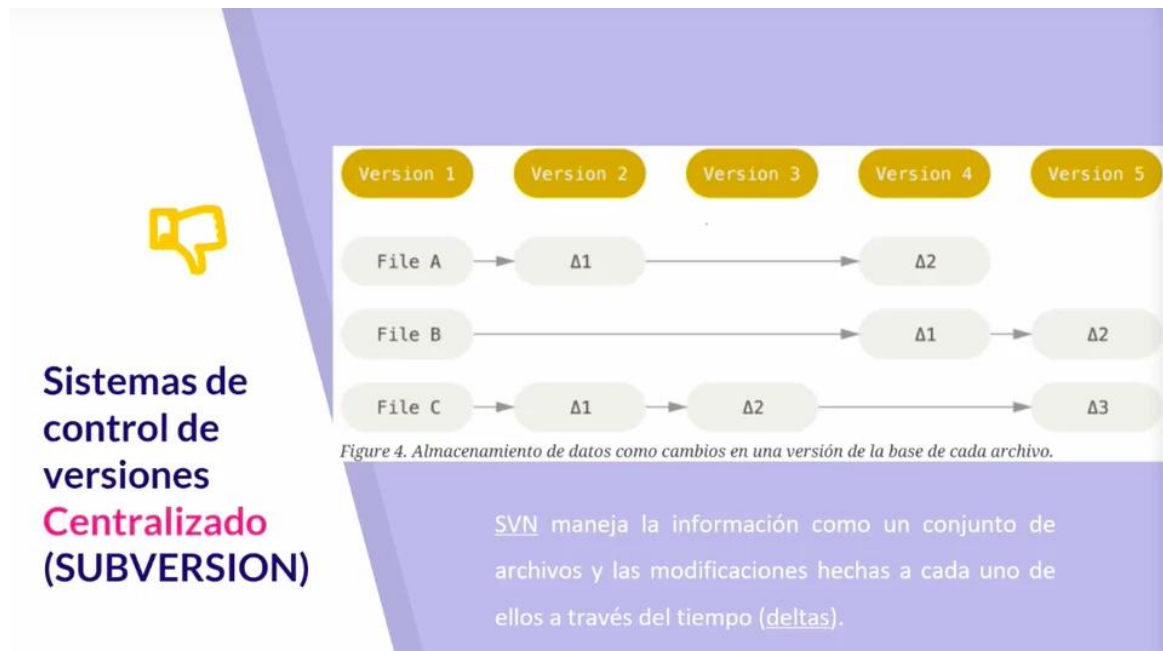
Funcionamiento de las réplicas

Las réplicas de versiones de proyectos en Sistemas de Control de Versiones Distribuidos funcionan como podemos ver en este gráfico:



Podemos tener una máquina que actúa como servidor, en el mismo se van almacenando las diferentes versiones de nuestro código, y cada uno de los clientes que participan en ese desarrollo tiene el histórico de las revisiones completa.

Sistemas de Control de Versiones Centralizados vs Sistemas de Control de Versiones Distribuidos



En **Sistemas de Control de Versiones Centralizados**, como Subversion, partimos de una versión, por ejemplo, de la versión 1 y tenemos tres ficheros, A, B y C.

De la versión 1 a la versión 2, las diferencias, como si fueran esa especie de incrementos que llamamos deltas, son almacenados por el sistema. De esta manera, por buscar algún símil, es como si Subversion trabajase con backups incrementales.

Después tenemos el funcionamiento de Sistemas de Control de Versiones de Git, en el que cada vez que hay cambios de ficheros éste es almacenado otra vez, y si no hay cambios es como si tuviésemos una especie de apuntador al fichero que no ha tenido cambios, en una revisión o en un hito del tiempo anterior.

Existen tres tipos de estado de un fichero Git:



Fundamentalmente, un proyecto Git se estructura en tres partes:

- El área del **working directory**, que es dónde vamos a tener todos nuestros ficheros, dónde estamos trabajando constantemente.
- El **staging area**, que es donde van los archivos que estamos modificando y que aceptamos para que vayan en una futura revisión.
- El **área de commit o el git directory**, que es dónde se almacenan la revisión completa. A lo largo de nuestro **curso de Git**, se explicará cómo podemos movernos a lo largo de esos tres estados, para qué sirven y por qué suponen una ventaja.

En resumen podemos decir que Git es una herramienta:

- **Potente**
- **Rápida**
- **Multiplataforma**
- Que se puede utilizar a través de la **línea de comando o con múltiples clientes**
- Es la base o el primer eslabón de herramientas de estructura a plataforma tipo **GitHub o GitLab**, que son las plataformas que se utilizan, de manera masiva, para albergar proyectos de software libre y otro tipo de proyectos que pueda tener una organización, y que quieran delegar en GitHub o en GitLab como su Servicio de Gestión de Control de Versiones Distribuidas.

Herramientas por utilizar:

git bash



Git Bash es un sistema de gestión de control de código fuente para Windows. Permite a los usuarios escribir comandos Git que facilitan la administración del código fuente a través del control de versiones y el historial de confirmación.

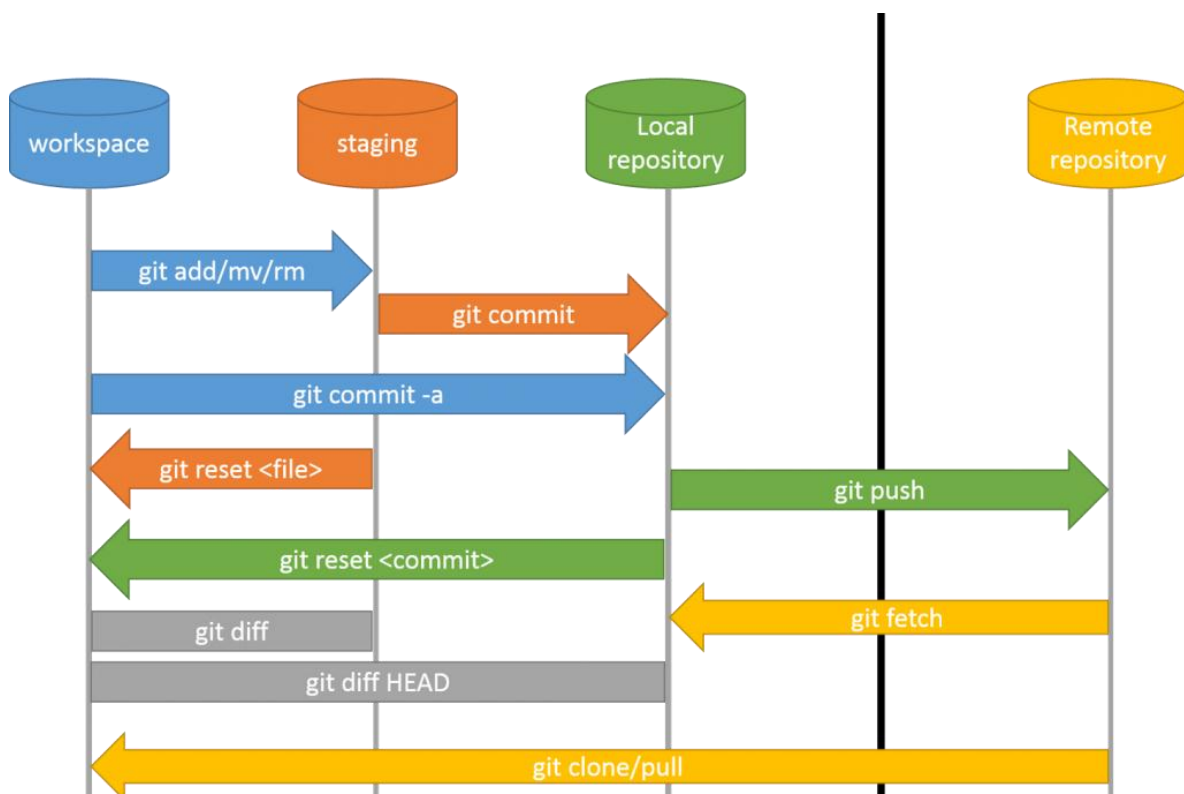
Descargue e instale git bash en su equipo:

<https://git-scm.com/downloads>

Crear cuenta **Github**

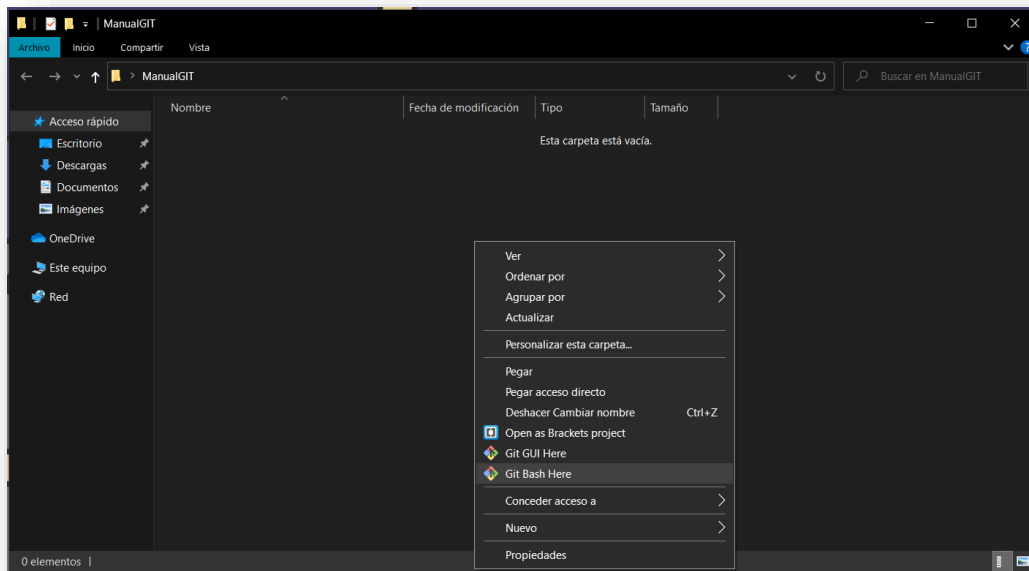


<https://github.com/>



IV DESARROLLO

Crear la carpeta donde tendremos nuestro repositorio, ingresamos a dicha carpeta y damos clic derecho dentro de la carpeta vacía y veremos la opción que dice “Git Bash”



Nos saldrá una ventana de este estilo

```
MINGW64/c/Users/PANDABOX/desktop/ManualGIT
PANDABOX@yuuta MINGW64 ~
$ cd desktop
PANDABOX@yuuta MINGW64 ~/desktop
$ cd ManualGIT
PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT
$ |
```

Dentro de la carpeta donde crearemos nuestro repositorio comenzaremos con algunos comando básicos que son muy necesarios.

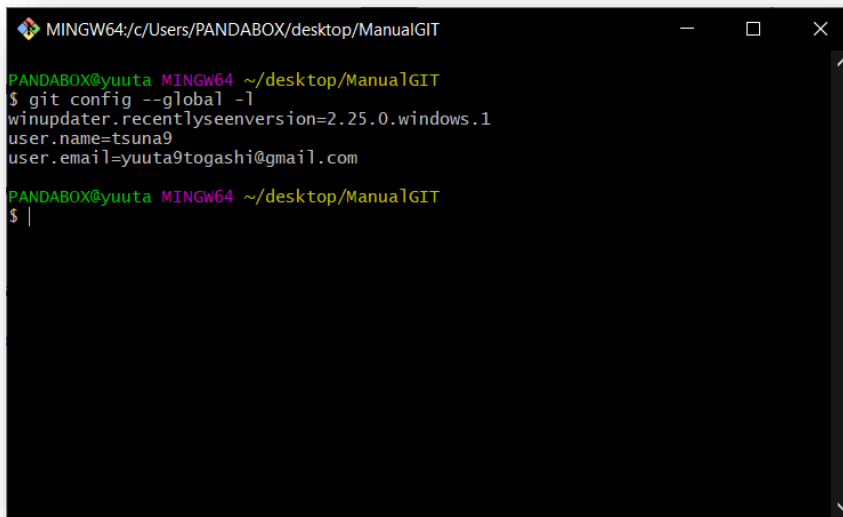
Necesitamos ejecutar el siguiente comando para iniciar nuestro usuario (anteriormente creado) donde escribiremos nuestro usuario y correo

```
git config --global user.name "Usuario"
git config --global user.email "correodeusuario@dominio.com"
```

Se nos pedirá la contraseña y con esto ya tendremos nuestro usuario abierto, entonces procedemos a crear nuestro repositorio con el siguiente comando. Para poder revisar la configuración de nuestro git solo tendremos que escribir el siguiente comando

```
git config --global -l
```

Donde se nos mostrara una ventana como la siguiente:

A screenshot of a terminal window titled 'MINGW64:/c/Users/PANDABOX/desktop/ManualGIT'. The prompt is 'PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT'. The command '\$ git config --global -l' has been executed, and the output is displayed: 'winupdater.recentlyseenversion=2.25.0.windows.1', 'user.name=tsuna9', and 'user.email=yuuta9togashi@gmail.com'. The prompt '\$ |' is visible at the bottom.

Caso tengamos otro usuario ya configurado, pero queremos agregar uno nuevo entonces con el siguiente comando podremos quitar toda la configuración existente.

```
git config --global --unset-all gui.recentrepo
```

Para establecer la configuración de nombre de usuario / correo electrónico específica del repositorio: Desde la línea de comando, cambie al directorio del repositorio.

Establece tu nombre de usuario:

```
git config user.name "FIRST_NAME LAST_NAME"
```

Configura tu dirección de correo electrónico:

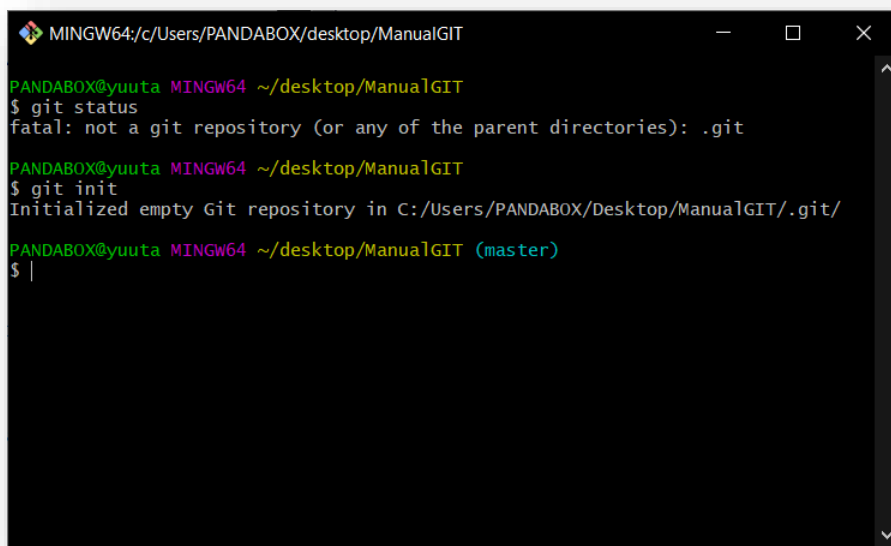
```
git config user.email "MY_NAME@example.com"
```

Verifique su configuración mostrando su archivo de configuración: **cat .git/config**

Ya configurado nuestro usuario, pasamos a iniciar nuestro repositorio local con el siguiente comando

```
git init
```

Caso no tengamos un repositorio creado y usemos el comando, y no tengamos un repositorio creado veremos el siguiente mensaje de error, luego de ejecutar el comando anterior podremos ver el siguiente mensaje y a la vez podremos observar que ya tenemos creado nuestro repositorio y nos muestra la opción que estamos en la raíz (master)

A screenshot of a terminal window titled 'MINGW64: c:/Users/PANDABOX/desktop/ManualGIT'. The prompt is 'PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT'. The first command entered is '\$ git status', which results in a 'fatal: not a git repository (or any of the parent directories): .git' error. The second command is '\$ git init', which successfully initializes an empty Git repository in 'C:/Users/PANDABOX/Desktop/ManualGIT/.git/'. The prompt then changes to 'PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT (master)'.

Para probar que hemos creado dicho repositorio entonces agregaremos un archivo a la carpeta, un ejemplo sencillo con el siguiente código

```
<HTML>

<HEAD>

<TITLE>ejemplo hola mundo</TITLE>

</HEAD>

<BODY>

<P>Hola Mundo</P>

</BODY>

</HTML>
```

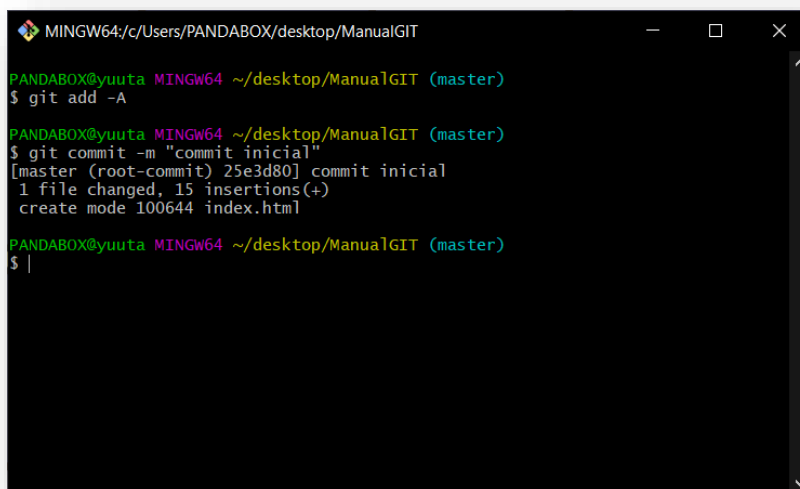
Lo ideal seria utilizar el siguiente comando agregando todos los cambios que le realicemos a nuestro repositorio local.

```
git add -A
```

Ahora para hacer commit (donde subiremos los cambios a nuestro repositorio en línea) usaremos el siguiente comando

```
git commit -m "commit inicial"
```

La estructura del anterior comando es primero la palabra reservada “git” la acción que realizaremos, y un mensaje describiendo lo que estamos cambiando, normalmente el mensaje tiene un limite de caracteres por lo que se recomienda escribir un mensaje preciso respecto al cambio que estamos haciendo.

A screenshot of a terminal window titled "MINGW64: c:/Users/PANDABOX/desktop/ManualGIT". The terminal shows the following commands and output:

```
PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT (master)
$ git add -A

PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT (master)
$ git commit -m "commit inicial"
[master (root-commit) 25e3d80] commit inicial
1 file changed, 15 insertions(+)
create mode 100644 index.html

PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT (master)
$ |
```

Observamos que se ha agregado un archivo al repositorio local, tenemos la opción de corroborar los commit que hemos hecho en esta rama con los siguientes comandos.

```
git log
git log --oneline --decorate --all --graph
```

Ya sea cualquiera de los dos podremos ver los commit realizados, solo que para el caso del segundo los podremos observar de forma ordenada y más limpia

```
MINGW64/c/Users/PANDABOX/desktop/ManualGIT

PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT (master)
$ git add -A

PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT (master)
$ git commit -m "commit inicial"
[master (root-commit) 25e3d80] commit inicial
1 file changed, 15 insertions(+)
create mode 100644 index.html

PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT (master)
$ git log --oneline --decorate --all --graph
* 25e3d80 (HEAD -> master) commit inicial

PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT (master)
$ git log
commit 25e3d80260483dfe27f0c769b326708f081f8b7e (HEAD -> master)
Author: tsuna9 <yuuta9togashi@gmail.com>
Date:   Wed Jul 8 23:29:33 2020 -0600

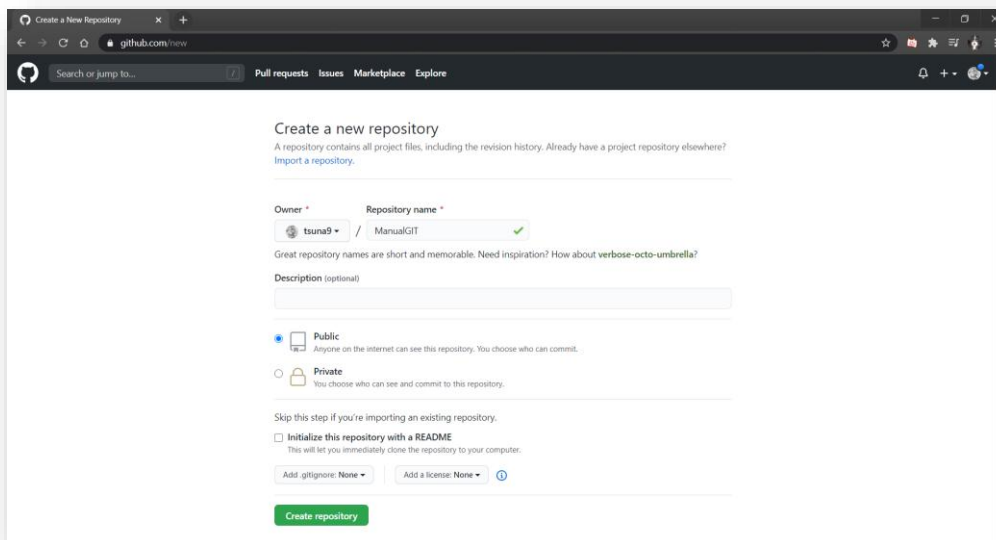
    commit inicial

PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT (master)
$ |
```

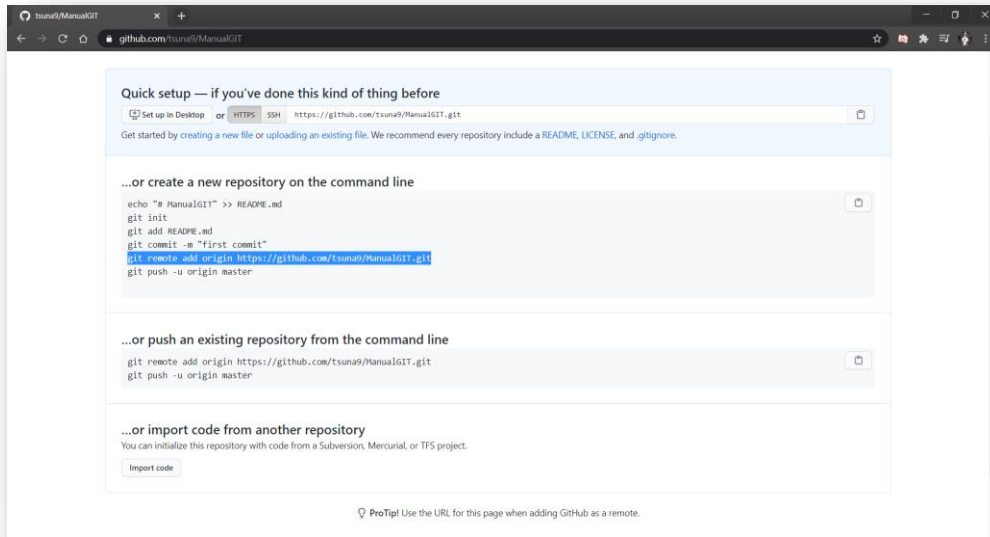
Para listar todos los repositorios usaremos el siguiente comando

```
git remote -v
```

En este caso como no tenemos ninguno, no saldrá nada, entonces con el siguiente comando subiremos los cambios a nuestro repositorio que para este caso tendremos que ir a la pagina de GitHub primero y crearemos el repositorio remoto.



Una vez creado nuestro repositorio remoto usaremos el siguiente comando que también se nos muestra al haber completado la creación del repositorio

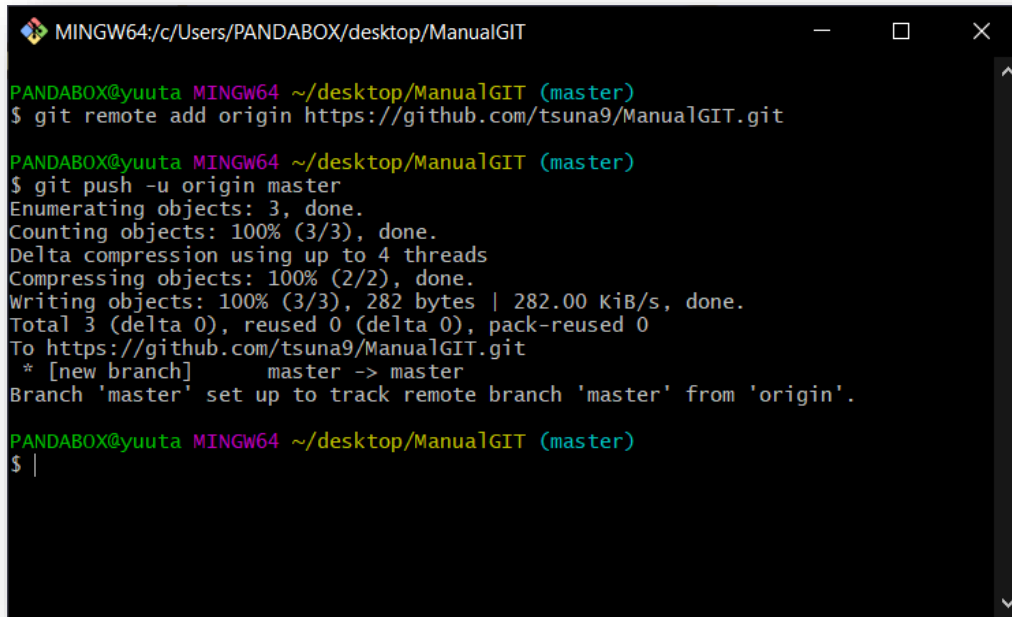


El comando es el siguiente

```
git remote add origin https://github.com/tsuna9/ManualGIT.git
```

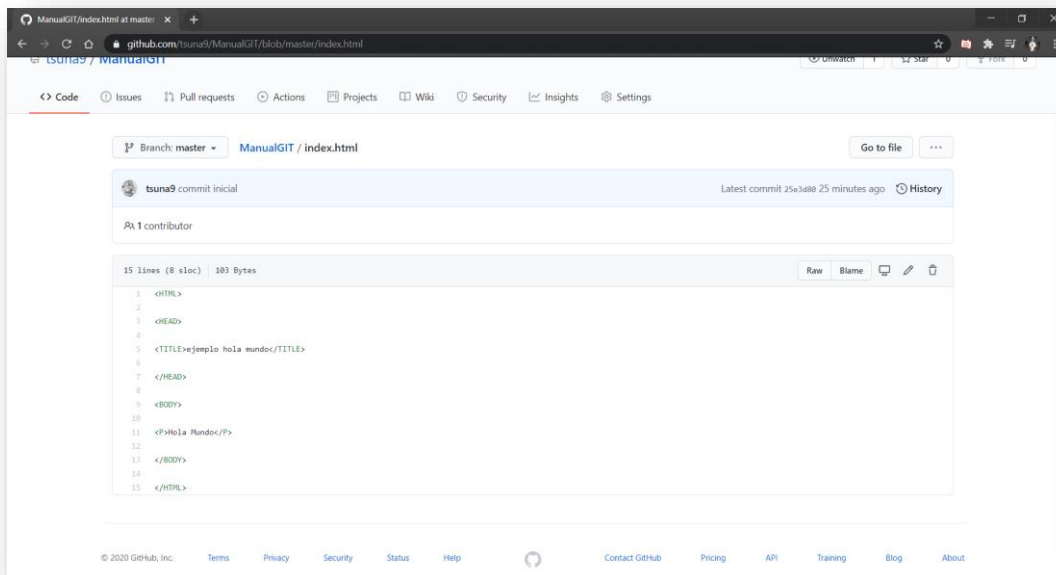
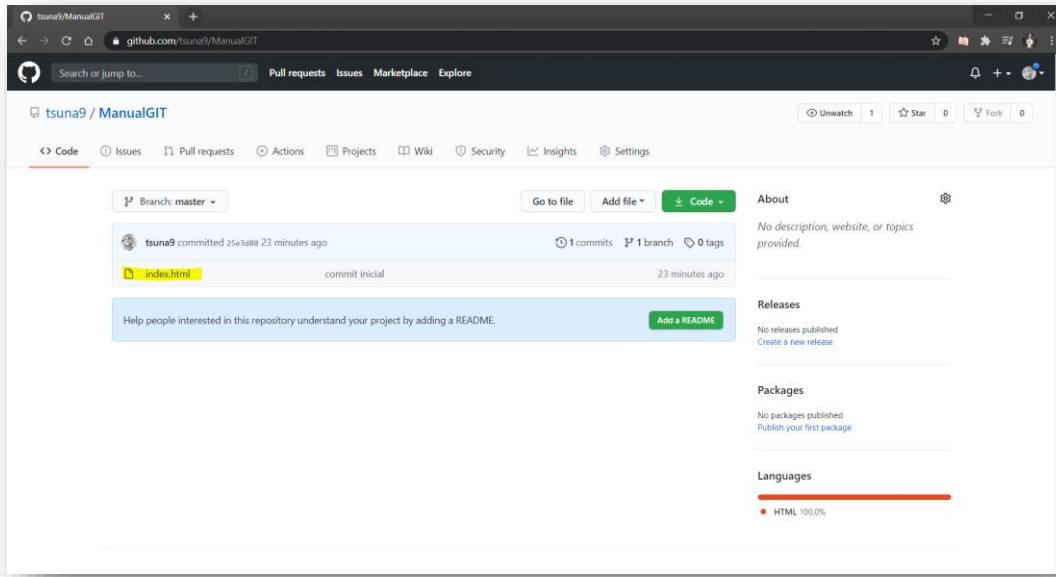
En el caso de dicho comando podríamos ponerle otro nombre a donde dice “origin” uno mas personalizado, pero en este caso lo dejaremos como se nos sugiere y ejecutamos dicho comando. Una vez ejecutado ya solo nos queda un paso para enviar nuestros cambios locales al repositorio remoto, eso lo haremos con el siguiente comando

```
git push -u origin master
```

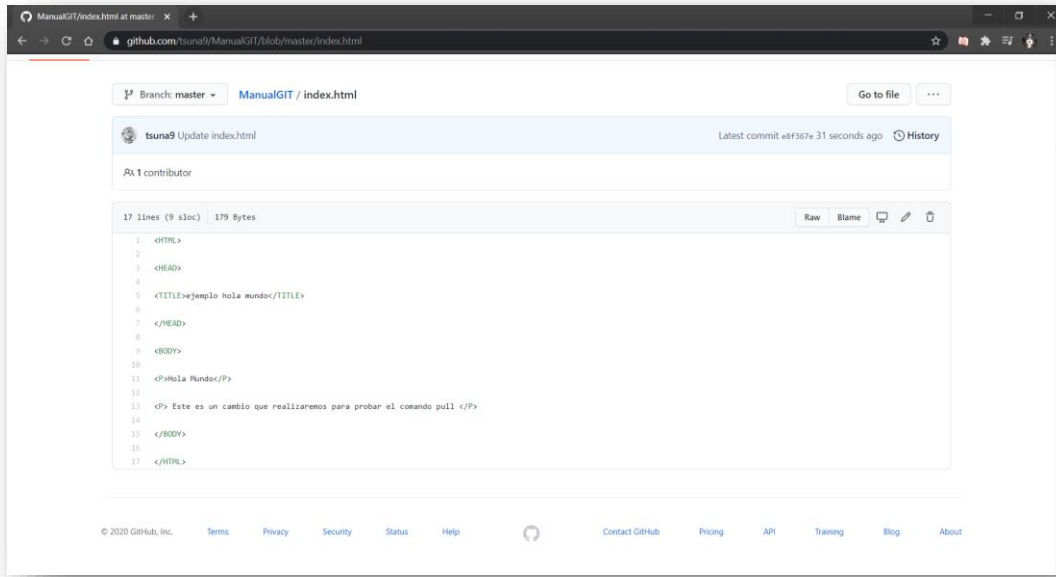
A screenshot of a terminal window titled "MINGW64:/c/Users/PANDABOX/desktop/ManualGIT". The prompt is "PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT (master)". The first command is "\$ git remote add origin https://github.com/tsuna9/ManualGIT.git". The second command is "\$ git push -u origin master". The output shows the push process: "Enumerating objects: 3, done.", "Counting objects: 100% (3/3), done.", "Delta compression using up to 4 threads", "Compressing objects: 100% (2/2), done.", "Writing objects: 100% (3/3), 282 bytes | 282.00 KiB/s, done.", "Total 3 (delta 0), reused 0 (delta 0), pack-reused 0", "To https://github.com/tsuna9/ManualGIT.git", "* [new branch] master -> master", and "Branch 'master' set up to track remote branch 'master' from 'origin'.". The prompt returns to "\$ |".

```
MINGW64:/c/Users/PANDABOX/desktop/ManualGIT
PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT (master)
$ git remote add origin https://github.com/tsuna9/ManualGIT.git
PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT (master)
$ git push -u origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 282 bytes | 282.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/tsuna9/ManualGIT.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT (master)
$ |
```

Podemos observar que los cambios han sido enviados, ahora solo queda corroborar si el archivo que creamos anteriormente se ha subido al repositorio remoto. Una nota es que a veces cuando se hace un “push” al remoto pide de nuevo el usuario y contraseña.



Podemos observar que justamente es nuestro archivo. Ahora modificamos dicho archivo para traer esos cambios a nuestro repositorio local.



Se agrego una párrafo al archivo, entonces ahora con el siguiente comando traeremos dichos cambios al local.

```
git pull origin master
```

```
MINGW64:/c/Users/PANDABOX/desktop/ManualGIT
PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT (master)
$ git pull origin master
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), 1.35 KiB | 4.00 KiB/s, done.
From https://github.com/tsuna9/ManualGIT
* branch      master      -> FETCH_HEAD
   25e3d80..e8f367e  master  -> origin/master
Updating 25e3d80..e8f367e
Fast-forward
 index.html | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)

PANDABOX@yuuta MINGW64 ~/desktop/ManualGIT (master)
$ |
```

Podremos ver que ha bajado los cambios y ahora revisemos el archivo dentro de la carpeta local.

Ejercicios para entregar:

Para todos los ejercicios lo que deben de entregar es la Url del repositorio público, por ejemplo : <https://github.com/AlexanderSiguenza/pruebasGit.git>

- Crear un repositorio local en una carpeta llamada **PruebasGit** , dentro de la carpeta crear una página html con el nombre **primerversion.html** con el siguiente párrafo.

“Git fue creado pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente, es decir Git nos proporciona las herramientas para desarrollar un trabajo en equipo de manera inteligente y rápida y por trabajo nos referimos a algún software o página que implique código el cual necesitemos hacerlo con un grupo de personas”

- Crear un repositorio local y versionarlo al repositorio remoto, con una estructura de carpetas de la siguiente manera: (principal) **GuiLabDPS**
(subcarpetas) guia01, guia02, guia03, guia04, guia05, guia06, guia07, guia08, guia09, guia10
- Realizar cambios en el archivo **primerversion.html** , pero desde el repositorio remoto, ingresar a github y buscar el repositorio y agregar a la página el siguiente Párrafo.

“Los Sistemas de Control de Versiones Distribuidos (DVCS) salvan este problema. Algunos ejemplos de sistemas distribuidos, aparte de Git, son Mercurial, Bazaar o Darcs. En este tipo de herramientas, los clientes replican completamente el repositorio.”

Luego de agregar el párrafo, deben de actualizar el repositorio local, con los cambios realizados en el remoto.

- Para estructura **GuiLabDPS** , clonar la página **primerversion.html** a la carpeta **guia01** y buscar 1 colaborador para trabajar la página en conjunto con un compañero, ambos deben de realizar cambios a ambos párrafos, puede ser agregar texto o quitar, no deben de comunicarse que harán , los cambios los van a realizar por separado, lo que se busca es que existan conflictos.

Luego de los cambios realizados por ambos colaboradores, el dueño del repositorio llevara los cambios al repositorio remoto y luego pedirá al colaborador hacer lo mismo , tomar captura de pantalla y comentar que sucede.

- Para estructura **GuiLabDPS** , clonar la página **primerversion.html** a la carpeta **guia02** y hacer tres ramas de trabajo , **desarrollo** , **producción**, **pruebas** , agregar un nuevo párrafo a la rama desarrollo “rama desarrollo” luego de hacer los cambios , guardarlos en el repositorio local y llevarlos a la rama pruebas guardar los cambios y llevarlos a la rama producción , para después llevar el repositorio local al repositorio remoto.
- Para estructura **GuiLabDPS** , clonar la página **primerversion.html** a la carpeta **guia03** y hacer tres ramas de trabajo , **des** , **pro**, **pru** , agregar un nuevo párrafo a la rama desarrollo “rama desarrollo pruebas” luego de hacer los cambios , guardarlos en el repositorio local y llevarlos a la rama pruebas y guardar los cambios , luego de hacer las pruebas se decide **deshacer** los cambios y se quita el párrafo “rama desarrollo pruebas” y se llevan estos nuevos cambios a las rama **des**, para después llevar el repositorio local al repositorio remoto.