

El arte de la analítica:

Módulo 1: Manejo de Consola y Manejo de Versiones

MSc Edoardo Bucheli

1 Introducción

En este primer módulo aprenderemos a utilizar la consola y a llevar a cabo manejo de versiones con **git** y **Github**.

2 Uso de Consola

Cuando hablamos de la consola nos referimos a la interfaz de texto de una computadora. También se le llama, shell, terminal, prompt, entre otros.

En Mac y Linux, podemos acceder a la consola por medio de las aplicaciones de Terminal. Aunque esta aplicación suele venir instalada con ambos sistemas operativos, existen también aplicaciones hechas por terceros con diferentes ventajas. En Windows existe el Command Prompt, el Power Shell, Anaconda Prompt entre otros, cada uno con funcionalidades un poco distintas.

La consola como aparece en Mac y Linux es la más común pues ambas surgen de Unix. Los comandos para Power Shell o Command Prompt pueden variar un poco pero aprenderemos los comandos para sistemas basados en Unix pues Linux es lo más común cuando accedemos a servidores ya sea locales o en la nube.

La buena noticia es que Windows en colaboración con Canonical (la compañía detrás de Ubuntu) crearon Windows Subsystem for Linux, el cual nos permite correr una instancia virtual de diferentes distros de Linux como Ubuntu o Debian.

2.1 Comandos básicos para la consola

Existen miles de comandos para la consola que vienen incluidos en el sistema operativo o que se pueden instalar pero en esta sección aprenderemos los más básicos.

- **pwd**: Ver el directorio actual.
- **cd**: Cambiar el directorio actual.
- **ls**: Listado del directorio actual.
- **mkdir**: Crear un nuevo directorio.
- **mv**: Mover un archivo.
- **rm**: Borrar un archivo.

- **rmdir**: Borrar un directorio.
- **cat**: Crear, ver y concatenar archivos.
- **touch**: Crear un nuevo archivo.
- **nano**: Editar un archivo.
- **sudo**: Correr un comando con permiso de administrador.

3 Git y Github

En esta sección aprenderemos sobre dos de las herramientas para gestión de control de versiones más utilizadas hoy en día, git y Github. Quizá tu primera duda sea, cual es la diferencia entre las dos así que empecemos con eso.

Git es una herramienta de manejo de versiones (o *version control* en inglés). Con esta herramienta, a través de lo que se conoce como un repositorio, podemos tener acceso a casi cualquier versión anterior de nuestra base de código con mucha facilidad. Nos permite también colaborar y experimentar con nuestro código sin miedo de perder información o arruinar el funcionamiento del programa.

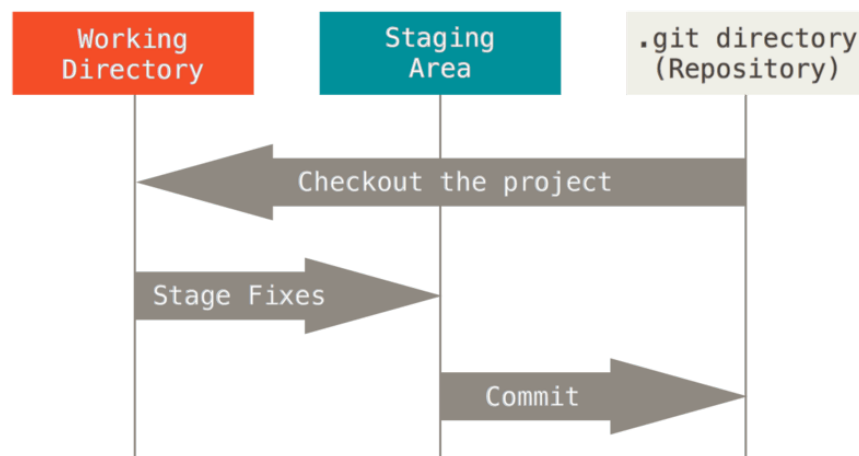
Por otro lado, Github es un servicio en línea que nos permite manejar repositorios remotos y añade otras funcionalidades como seguridad, repositorios privados y herramientas de colaboración.

Básicamente, Git es un pedazo de software *open source* para manejo de versiones y Github es una empresa que nos permite manejar repositorios remotos de Git con algunas funcionalidades añadidas.

3.1 Fundamentos de Git

Existen muchos workflows dentro de git pero empezaremos con el uso más común, crear un repositorio, hacer cambios y guardarlos. Antes de continuar, hablemos de las secciones de un repositorio y las maneras en las que podemos clasificar los archivos en git.

3.1.1 Áreas de un repositorio



- **Working Directory/Tree:** Es la versión o *checkout* del proyecto en la que se está trabajando.
- **Staging Area:** Aquí se guardan los cambios que irán al siguiente commit.
- **Git Directory:** Aquí se guarda la metadata y base de datos de objetos del proyecto. Esto es lo que se copia cuando hacemos un *clone*.

3.1.2 Clasificación de archivos

1. **Untracked:** El archivo no es parte del repositorio y no se analizan sus versiones.
2. **Unmodified:** El archivo no tiene cambios respecto a la versión anterior. Ya sea porque no sea ha modificado o porque acabamos de hacer commit.
3. **Modified:** El archivo tiene cambios pero no le hemos indicado a git que queremos guardar los cambios.
4. **Staged:** El archivo tiene cambios y lo preparamos para guardar los cambios.

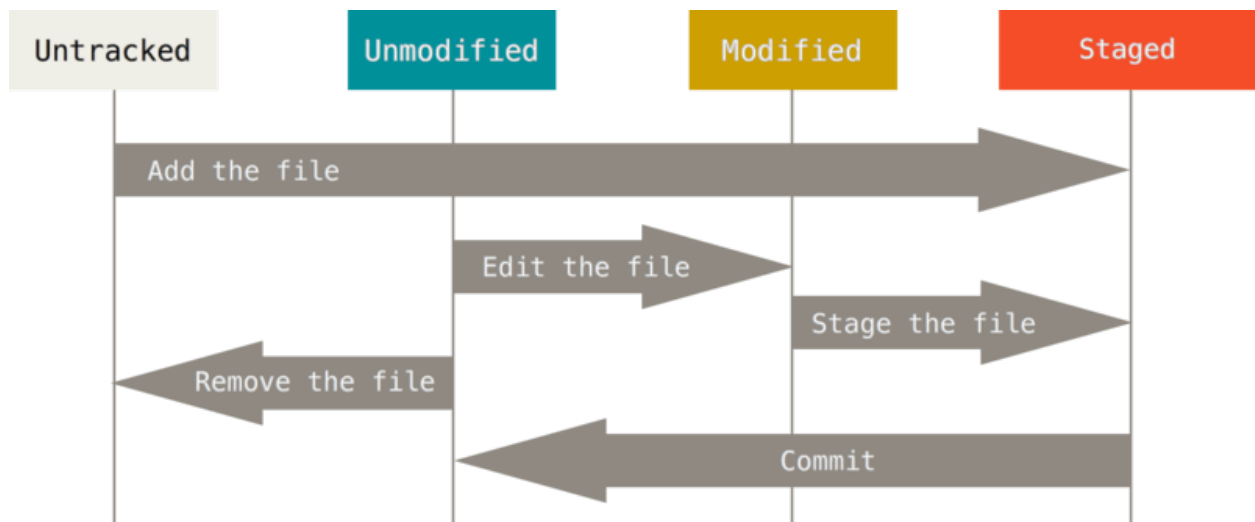


Figure 1: El ciclo de archivos en git.

3.2 Uso básico de git sin repositorio remoto

Para crear un nuevo repositorio de git, utilizamos el comando `git init` dentro del directorio que queremos convertir en un repositorio de git. Esto crea ciertos archivos dentro de nuestro repositorio donde git guardará los snapshots de nuestro proyecto para poder referenciarlos más adelante.

Ahora podemos crear los archivos de nuestro proyecto y agregarlos al repositorio usando el comando `git add [nombre-del-archivo]`. Este comando hace que nuestro archivo pase de ser *untracked* a *staged*.

Específicamente podemos hacer lo siguiente.

1. Crear un nuevo directorio y convertirlo en repositorio de git.

```
$ mkdir my_repository
$ cd my_repository
$ git init
```

2. Crea un nuevo archivo y ponle el nombre que gustes, en este caso crearé un archivo llamado "README.md" y otro llamado "test.py". Ahora queremos editar los archivos, podemos hacer esto en un editor de text, en un IDE o, para hacerlo desde la consola, utilizamos el comando **nano**.

```
$ touch test.py README.md
$ nano test.py
$ nano README.md
```

3. Una vez que hagamos los cambios podemos pasarlos al *staging area* de la siguiente forma,

```
$ git add test.py
$ git add README.md
```

Ahora nuestros archivos pasaron de ser untracked al staging area.

4. Ahora hacemos el *commit*. Una vez hecho esto, podremos acceder en cualquier momento a esta versión del proyecto.

```
$ git commit -m "Initial commit"
```

Presta atención a la nomenclatura. Te darás cuenta que utilizamos el *flag* **-m** seguido de un string. Aquí guardamos un comentario que nos ayuda a entender en qué consiste el cambio actual.

5. Digamos que ahora queremos hacer un pequeño cambio, editaremos el archivo **test.py**. Una vez que hacemos el cambio podemos generar un nuevo *commit*.

```
$ nano test.py
$ git add test.py
$ git commit -m "Changed print message"
```

3.3 Repositorios remotos

Para colaborar con personas o entre nuestros dispositivos utilizamos repositorios remotos. Los repositorios remotos son versiones del proyecto ubicadas en el internet o algún otro tipo de red. Puede haber muchos de ellos en los cuales podemos tener permisos para leer/escribir o solo leer. Ya que configuramos los repositorios remotos podemos hacer *pull* (jalar de un remoto) o *push* (empujar hacia el remoto).

Hagamos un repositorio remoto para nuestro repositorio actual. En Github podemos crear un nuevo repositorio, aquí podemos seleccionar un nombre, descripción y agregar unos archivos. En este momento solo le pondremos un nombre y descripción a nuestro proyecto. Podemos ver esto en la figura 2.


Aquí Github nos ayuda de varias formas, usaremos la tercera recomendación, *push an existing repository from the command line*.

```
$ git remote add origin [url-del-repositorio]
$ git branch -M master
$ git push -u origin master
```

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner *

 ebucheli ▾

Repository name *

/ TC1002S 

Great repository names are short and memorable. Need inspiration? How about **musical-winner**?

Description (optional)

Repositorio para la Semana Tec TC1002s



Public

Anyone on the internet can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.



Add a README file

This is where you can write a long description for your project. [Learn more.](#)



Add .gitignore

Choose which files not to track from a list of templates. [Learn more.](#)



Choose a license

A license tells others what they can and can't do with your code. [Learn more.](#)

Create repository

Figure 2: Crear un nuevo repositorio de Github

Ahora nuestro repositorio se encuentra respaldado por un repositorio remoto con el nombre *origin*. Ahora que tenemos este repositorio remoto vamos a añadir dos comandos al workflow que definimos en la sección anterior. Imaginemos que cambiamos el archivo README.md y ahora queremos hacer un commit para guardar nuestros cambios, también queremos guardar estos cambios en el repositorio remoto utilizando el comando `git push [remoto] [branch]`.

```
$ git add README.md
$ git commit -m "Changed README.md"
$ git push origin master
```

Ahora, imaginemos que colaboramos con alguien o que nos cambiamos de computadora y queremos seguir trabajando. Sin embargo, el *remote* debe ser diferente a nuestra versión local. Por lo tanto podemos

usar el comando,

```
git pull
```

Con esto, descargamos la versión más nueva en el repositorio remoto y hacemos merge con nuestra versión local.

3.4 Branching

Una de las herramientas más poderosas dentro de Git es la capacidad de generar *branches*. Los *branches* nos permiten saltar entre versiones alternas del código lo cual trae muchas ventajas. Algunos ejemplos son, manter una versión estable principal junto con una versión de desarrollo. Hacer experimentos raros que no sabemos si vamos a terminar usando. Permitir que distintos colaboradores trabajen en distintos problemas y al final unamos sus contribuciones entre muchas otras.

El branch principal se llama **master** por default. Si queremos generar un nuevo branch llamado "development" y cambiarnos a él podemos ejecutar el comando,

```
$ git branch development
$ git checkout development
```

Ahora podemos hacer todos los cambios que queramos, si en cualquier momento quiero regresar al estado de mi repositorio master, solo necesito correr el comando **git checkout master**. Git se encargará de poner nuestro Working Directory de acuerdo al branch en el que estemos posicionados.

De aquí podemos hacer todos los cambios que queramos, en general, una vez que estemos conformes con los cambios en el nuevo branch queremos pasarlos a *master*. Podemos hacer esto con el commando **merge** de la siguiente manera.

1. Hacer checkout desde development hacia master,

```
$ git checkout master
```

2. Hacer merge

```
$ git merge development
```

Este paso puede ser muy complicado o muy sencillo dependiendo de si hay conflictos o no. En general, si nuestro nuevo branch solo es una versión actualizada de *master* entonces no habrá problema, pero si *master* tuvo algún cambio antes de crear development u algún otro merge podemos tener riesgo de que haya algún conflicto.