* Práctica 1.3: Git Básico

P1.3 - Introducción y comandos básicos para trabajar con el control de versiones Git

A continuación, ofrecemos una guía paso a paso para aquellos usuarios principiantes interesados en trabajar con un repositorio de Git en local.

El sistema de control de versiones Git es una herramienta fundamental para muchos desarrolladores, especialmente cuando colaboran en un proyecto. Git ayuda a mantener una visión de conjunto, a preservar las versiones antiguas y a integrar los cambios de manera coherente. Para ello, Git agrupa una serie de programas de línea de comandos y crea un efectivo entorno de trabajo.

Es la mejor forma de trabajar para un desarrollador. De esta manera vais a tener siempre vuestro código a salvo en la nube. Podéis clonar el proyecto y trabajar en cualquier ordenador.

1. Instalar Git

La forma más oficial está disponible para ser descargada en el sitio web de Git. Solo tienes que visitar Download, seleccionar tu sistema operativo y la descarga empezará automáticamente.

Una vez que hayamos instalado Git en nuestro ordenador, abriremos la aplicación "Git Bash", o desde linux, podremos abrir una terminal y ejecutar el comando `git` para interactur con git.

En windows, Git Bash es la herramienta de línea de comandos que permite a los usuarios de Windows utilizar las funciones de Git. Git Bash es Git en una “Bourne Again Shell”. La aplicación contiene numerosas utilidades de Unix. Git Bash os permitirá usar herramientas MinGW/Linux Bash con Git en la línea de comandos. Todas esas cosas bonitas que se hacen en Linux también las podemos hacer en Windows a través de Git Bash.

2. Configurar Git y crear nuestro primer proyecto

1. Primero tenemos que definir nuestra identidad, para ello en la línea de comandos escribiremos las siguientes instrucciones utilizando nuestro usuario iPasen y `xxxxx@g.educaand.es`. Mas adelante podremos utilizar el correo `xxxxx@iesrafaelalberti.es`:

```
> git config --global user.name "dcansib483"
> git config --global user.email "dcansib483@g.educaand.es"
```

4. Algunos comandos básicos para navegar y trabajar con ficheros y carpetas:
    - `clear`: limpia la pantalla de la consola.
    - `pwd`: muestra el directorio actual.
    - `ls`: lista los ficheros y directorios *(parámetros `-l`: muestra la lista detallada y `-a`: muestra archivos ocultos).*

- `cd`: para movernos entre directorios *(por ejemplo `cd nombreDir`. Con `cd ..` voy al directorio anterior al actual)*.
- `mkdir`: para crear un directorio *(por ejemplo `mkdir nombreDir`)*.
- `rmdir`: para borrar un directorio *(por ejemplo `rmdir nombreDir`)*.
- `cat`: para volcar el contenido de un archivo *(por ejemplo `cat nombreFile`)*.
- `touch`: para crear un archivo *(por ejemplo `touch nombreFile`)*.
- Podemos pulsar las flechas arriba y abajo para movernos por los comandos ejecutados previamente.
- Podemos autocompletar el nombre de los directorios o ficheros.

5. El fichero de Git dónde se almacena la información de nuestra identidad está en un archivo oculto en nuestro `HOMEPATH (…users/usuarioT)` llamado *.gitconfig*, intenta encontrarlo y visualizar su contenido con el comando `cat`.
6. Accedemos a la carpeta `Documents` y creamos el directorio `ProgPython`:

```
> cd Documents
> mkdir ProgPython
```

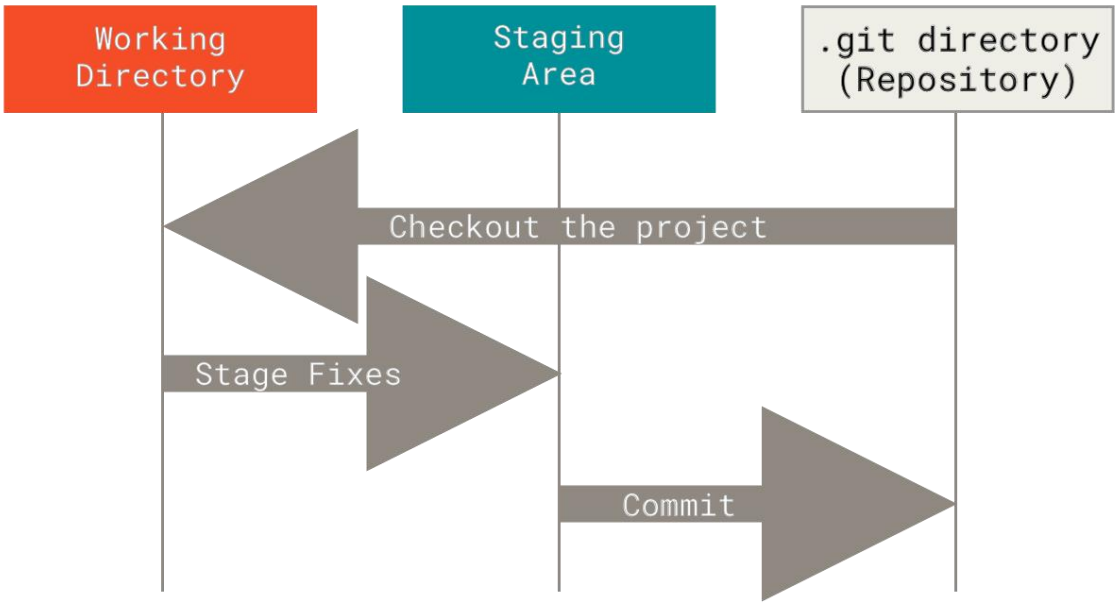
9. En esta carpeta vamos a crear nuestro proyecto de Git. Inicializamos Git en este directorio para indicarle que esta carpeta es nuestra área de trabajo:

```
> cd ProgPython
> git init
```

¿Cómo vamos a trabajar con Git (Proyecto)?

De manera muy básica, en un proyecto de Git vamos a trabajar con 3 secciones o áreas principales:

- **Área de trabajo** *(creamos carpetas y ficheros, modificamos el contenido de los ficheros y ejecutamos el comando `add` para agregarlos al área de preparación del proyecto)*
- **Área de preparación** *(staging area)*
- **Repositorio** *(commit)*



12. Nos creamos un primer programa en Python y lo ejecutamos *(todo desde la línea de comandos por ahora)*:

```
> touch holamundo.py
```

14. Podemos crearnos el programa y editarlo de manera gráfica en el Explorador de Windows, pero esta vez vamos a usar un editor de consola para escribir el contenido de nuestro programa. Después mostraremos su contenido en la terminal con el comando cat y mediante el intérprete de Python lo ejecutaremos:

```
> nano holamundo.py
```

    - Dentro del fichero escribimos `print("Hola mundo DAM-DAW!")`
    - Guardamos y salimos *(leer las opciones en la barra inferior)*
    - Para comprobar el contenido del fichero utilizamos el comando siguiente:

```
> cat holamundo.py
```

    - Vamos a ejecutar el programa realizado en Python:

```
> python holamundo.py
```

18. Ahora vamos a mirar el estado de nuestra área de trabajo... a ver que nos dice Git:

```
> git status
```

¿Qué va a pasar si vamos al directorio justo anterior y volvemos a ejecutar el mismo comando?

20. Volvemos a la carpeta de nuestro proyecto de Git... y volvemos ver el estado de nuestro proyecto... vemos que nos está indicando que existe un fichero nuevo sin añadir a nuestra área de preparación. La añadimos:

```
> git add holamundo.py
```

22. Cómo es un programa muy pequeño y ya lo hemos terminado, vamos a confirmar que es un buen punto de partida para hacer un commit, es decir, lo pasamos a nuestro repositorio o área de producción:

```
> git commit -m "Primera versión de hola mundo"
```

Si volvemos a ejecutar el estado del proyecto de Git veremos que no tenemos ningún cambio pendiente... está todo en el repositorio.

24. Vamos a crear otro programa, pero esta vez desde un IDE (Entorno de desarrollo integrado: Pycharm o Visual Code)

Primero creamos un directorio que se llame `ejercicios1` y después un fichero que se llame `prueba1.py` que contenga el siguiente código:

```
edad = int(input("Introduzca su edad: "))
if edad >= 18:
    print("Toma una cerveza!")
else:
    print(f"Toma un zumo de piña, con {edad} años eres menor.")
```

Esta vez lo vamos a abrir con el IDE, diciéndole que nos cree un proyecto en la carpeta `ProgPython`... podemos crear el directorio y el programa vacío con los comandos de consola **(`mkdir` y `touch`)** y después abrir el fichero con el IDE **(En windows: botón derecho desde el Explorador de archivos)**

A continuación, lo vamos a ejecutar dentro del IDE para ver cómo funciona el programa...

25. Esto nos ha generado en la carpeta del proyecto el directorio: `ejercicios1` y puede ser que otros directorios propios del IDE, como por ejemplo `.idea` **(usado por el IDE Pycharm para la gestión del proyecto)**. Si comprobamos el estado del proyecto **(`git status`)** nos muestra el directorio `.idea` para que lo añadamos también. Si no queremos subir a nuestro repositorio esta carpeta, podemos indicarle a Git que la ignore. Para ello, vamos a crearnos, en la carpeta del proyecto de Git, el fichero `.gitignore` que contendrá los archivos y carpetas que deseamos que Git ignore al comprobar el estado de los archivos del proyecto. En nuestro caso, solo tendrá una línea **(podemos hacerlo con el editor `nano`)**:
26. `> nano .gitignore`

    Al que añadiremos la linea siguiente:

    ```
    .idea
    ```

    A continuación debemos añadir al repositorio el fichero `.gitignore` y la carpeta `ejercicios1`:

    ```
    > git add .gitignore
    > git add ejercicios
    > git commit -m "Primera versión de la carpeta ejercicios1"
    ```

27. Ahora mismo tenemos todo actualizado en nuestro repositorio... para ver todos los commits que hemos realizado **(observad el código HASH que tiene cada commit)**:
28. `> git log`
29. Podemos hacer ahora una modificación al programa `prueba1.py`
30. `edad = input("Introduzca su edad: ")`
31. `if edad >= 18:`
32. `    print("Toma una cerveza!")`
33. `else:`
34. `    print(f"Toma un zumo de piña, con {edad} años eres menor.")`

    Si volvemos a comprobar el estado del proyeto de Git nos dirá que hay un fichero modificado... vamos a añadirlo y hacer commit.

    ```
    > git add ejercicios1
    > git commit -m "Segunda versión con un error"
    > git log
    ```

    Si ejecutamos el programa desde el IDE PyCharm observamos que nos da un gran error... nos hemos equivocado y está en el repositorio final :-(

    ¿Qué hacemos ahora?

35. No pasa nada, para eso tenemos un control de versiones... primero hacemos `git log`, copiamos el número que está a la derecha del commit de la versión a la que queremos volver y lo pegamos detrás del siguiente comando:

36. `> git log --oneline`
37. `> git reset --hard <commit_hash>`
38. Si volvemos al IDE, observamos que volvemos a tener la primera versión de nuestro programa que funciona perfectamente. Observad también con el comando `git log` que ya no aparece la segunda versión, pues el comando `git checkout` la ha eliminado del control de versiones para ir a una versión anterior *(esto no se puede deshacer)*.
39. Otros comandos que nos pueden ayudar:
    - Para añadir TODOS los ficheros de un directorio => `git add .`
    - Para añadir TODOS los cambios pendientes de una sola vez => `git add -A`
    - Para deshacer un `git add` antes de hacer un `git commit` => `git reset nombreArchivo` o `git reset` para deshacer todos los cambios. También podemos usar `git checkout ..`
    - Para cambiar el HEAD al commit especificado, pero sin afectar la rama: `git checkout <commit_hash>`
    - Para mover la rama y borrar todos los cambios posteriores a ese commit: `git reset --hard <commit_hash>`
    - Para mover la rama pero mantener los cambios locales en staging: `git reset --soft <commit_hash>`
    - Para cambiar el nombre de la rama master por main, es decir, renombrar la rama => `git branch -m main`

Otros enlaces

- Ayuda visual
- Guia rápida
- Documentación de referencia de git
- Libro de Git