Unidad 2 — Bash y Git

Contenidos

Esta unidad es una introducción a dos herramientas básicas para los desarrolladores, muy utilizadas en el ámbito del desarrollo web: una interfaz de línea de comandos (usando generalmente Bash) y Git.

CLI

La interfaz de línea de comandos o *command line interface* (CLI) es una forma de interactuar con el sistema operativo. Los usuarios de una computadora hoy en día usan exclusivamente una interfaz gráfica o GUI. La línea de comandos muchas veces se asocia con el pasado, con una forma de usar la computadora cuando no existía nada mejor. Esto no es cierto y como programadores tenemos que estar familiarizados con la línea de comandos para realizar muchas tareas. Además la línea de comandos es más expresiva que una interfaz gráfica, muchas tareas sólo pueden realizarse allí, o bien de manera más simple.

Shell

La shell es un programa del sistema operativo que interpreta los comandos que escribimos y los ejecuta. Toma su nombre de que es la capa más externa del sistema operativo, con la cuál interactuamos como usuarios. En los sistemas operativos derivados de UNIX (Linux y MacOS) usamos generalmente un programa llamado `bash`.

Emuladores de terminal

Si nos encontramos usando un entorno de escritorio, en Linux por ejemplo KDE o GNOME, no

usamos directamente `bash` sino que nos comunicamos con el programa a través de un emulador de terminal. Este programa que suele aparecer simplemente como terminal en el menú de aplicaciones puede cambiarse a gusto del usuario. En GNOME por ejemplo es `gnome-terminal` y en KDE `konsole`.

El prompt

Cuando abrimos una terminal usualmente nos encontramos con algo similar a esto.

```
[user@host ~]$
```

Esto se conoce como *prompt* y aparece cuando la terminal está dispuesta a recibir comandos. Además puede personalizarse para mostrar información útil, en el caso de arriba está indicando el nombre de usuario, de máquina y el directorio actual. Si en vez de ver un signo `$` vemos un `#` significa que la sesión de esa terminal tiene privilegios de superusuario.

Comandos

Los comandos en `bash` y en otras *shells* también tienen la siguiente estructura.

```
$ comando -opciones argumentos
```

El comando es simplemente el nombre del comando, como `cd`. Las opciones están precedidas por un guión o dos guiones si se usa la versión no abreviada. Las opciones también se conocen como *flags*. Por último los argumentos dependen del comando en cuestión y muchas veces se pueden aceptar varios. Por ejemplo el comando `ls` que lista los contenidos de uno o más directorios.

```
$ ls -a /usr/bin /dev
```

En el ejemplo anterior listamos los contenidos de dos directorios o carpetas, `/usr/bin` que contiene programas instalados en Linux y `/dev` que contiene archivos que representan dispositivos de hardware entre otras cosas. Además se usa la opción `-a` que incluye en el resultado los archivos y carpetas ocultas. Usando la versión larga del *flag* `-a` y listando los contenidos del escritorio del usuario se vería así.

```
$ ls --all /home/user/Desktop
```

Cabe destacar que los espacios en los ejemplos anteriores son importantes y se utilizan para delimitar el comando de las opciones y los argumentos.

Ayuda

En los sistemas derivados de UNIX cada comando debe tener una página de manual o *man page*.

Esto es un estándar y una tradición en estos sistemas operativos. Para invocar la ayuda de cualquier comando simplemente usamos `man comando` que nos imprime la *man page* del comando que nos interesa. Por ejemplo para ver la ayuda de `ls`

```
$ man ls
```

Navegación

Veamos unos comandos básicos de navegación para moverse por la estructura de archivos o *filesystem*.

`pwd`
: *Print working directory*. Imprime el directorio actual donde estamos parados.

`cd`
: *Change directory*. Cambia de directorio al especificado como argumento.

`ls`
: *List directory*. Lista los contenidos de uno o más directorios.

Manipulación de archivos y directorios

Los comandos básicos para borrar, crear, mover y copiar archivos y directorios.

`cp`
: *Copy*. Copia archivos y directorios.

`mv`
: *Move*. Mueve archivos y directorios, sirve también para renombrar.

`rm`
: *Remove*. Elimina archivos y directorios.

`mkdir`
: *Make directory*. Crea un directorio (carpeta).

`touch`
: Sirve para crear archivos.

Si queremos trabajar usando `rm`, `mv` y `cp` con directorios tenemos que indicar la opción `-r` (recursivo) para que copie, mueva o elimine un directorio con todos sus subdirectorios y archivos. Si no queremos que nos pida confirmación (con archivos de solo lectura) podemos usar el *flag* `-f` (*force*).

Redirección

Para entender el concepto de **redirección** y **piping** primero hay que entender los *streams* estándar. Hay tres *streams* llamados **stdin** (entrada estándar), **stdout** (salida estándar) y **stderr** (error estándar). Estos tres *streams* o flujos de datos están conectados por defecto al teclado (stdin) y a la pantalla de la terminal (stdout y stderr).

Los operadores de redirección sirven para llevar la salida o la entrada estándar a un archivo distinto los que están configurados por defecto. Un ejemplo sirve para clarificar. Supongamos que queremos guardar la palabra "Hola" en un archivo llamado `hola.txt`. Podemos escribir "Hola" en pantalla usando `echo`.

```
$ echo Hola
$ Hola
```

Eso sucede porque `echo` imprime sus argumentos en *stdout*. Podemos imprimir la palabra en un archivo usando el operador `>` de redirección de esta manera.

```
$ echo Hola > hola.txt
$ cat hola.txt
$ Hola
```

Efectivamente estamos redireccionando *stdout* al archivo `hola.txt` en vez de la pantalla de la terminal. Después simplemente usamos `cat` para mostrar el archivo en pantalla.

Ahora supongamos que tenemos un archivo de texto con los siguientes contenidos:

```
$ cat archivo.txt
4
2
3
0
```

Si queremos ordenarlos podemos usar `sort` y redireccionar `stdin` al archivo en vez del teclado de la siguiente manera:

```
$ sort < archivo.txt
0
2
3
4
```

Por último vemos el operador `|` (*pipe* o tubería). A menudo tenemos que ver una archivo de texto en la terminal que es muy largo para la pantalla. Podemos usar `|` y `less` que es un *pager*, un programa que realiza paginación.

```
$ cat archivo-muy-largo.txt | less
```

Estamos redireccionando la salida de `cat` a la entrada de `less`, haciendo una especie de "tubería" entre los dos comandos.

Git

Git es un **VCS** (*Version Control System*). Un sistema de control de versiones. Este tipo de herramientas sirven para tener un control y *backup* de los cambios realizados en una serie de archivos. Originalmente se crearon para que los programadores pudieran mantener su código de manera más eficiente. Pero no necesariamente tienen que usarse para mantener archivos de código.

Git fue creado por Linus Torvalds, el mismo del kernel de Linux, para este último proyecto. Hoy en día gracias a GitHub (una plataforma para compartir repositorios de Git) es el VCS más usado para desarrollo de software.

Git nació como una herramienta de UNIX para ser utilizada en la CLI, pero también existen entornos gráficos para utilizar Git como GitHub Desktop. De todas maneras cualquier GUI que sirva para usar Git solo tiene un subconjunto de todos los comandos posibles en la interfaz de línea de comandos.

Otros VCS que existen y compiten con Git son por ejemplo: Mercurial, CVS y Subversion. También la plataforma de Google Drive funciona en cierta medida como un VCS.

La característica fundamental de Git es que es un sistema de control de versiones distribuido, esto quiere decir que no depende de la conexión a un servidor remoto para trabajar. Todo es local.

Git en la línea de comandos

Todos los comandos en Git tienen la forma de `git <verbo>` donde reemplazamos verbo por los distintos comandos específicos.

Por ejemplo para iniciar un nuevo repositorio en la carpeta actual ejecutamos `git init`, (iniciar). Los comandos de Git podemos dividirlos en los comandos de inicialización como `init`. Los que me permiten guardar cambios en un repositorio y hacer ramas. Y por último los que usamos para trabajar con repositorios remotos.

Creación de repositorios

Un repositorio de Git es una carpeta o directorio donde quiero realizar un control de los cambios en sus archivos a lo largo del tiempo. Supongamos que es una carpeta llamada `sitio` donde voy a guardar los archivos de un sitio web. Para convertir la carpeta a un repositorio de Git ejecutamos dentro de la carpeta el comando `git init` y listo.

Hay otra manera de obtener un repositorio, que es clonarlo de un servidor. Para eso usamos el comando `git clone <url>` donde reemplazamos `<url>` por la URL del repositorio en el servidor,

como por ejemplo `https://github.com/escuela-tecnica-35/oixum-docs.git`. Después de clonar obtenemos un directorio de igual nombre al repositorio remoto.

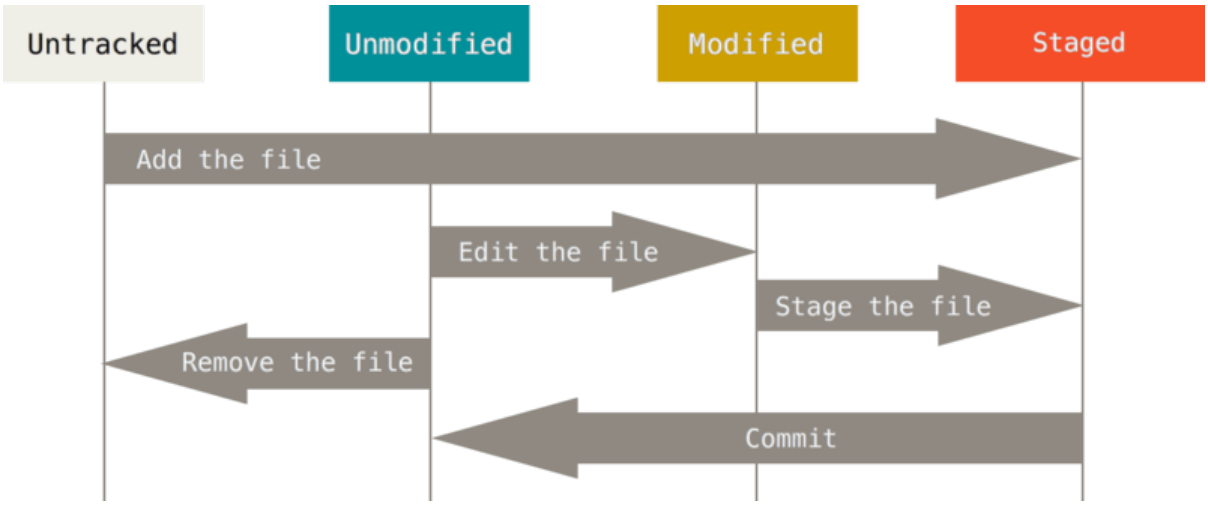
En ambos casos podemos constatar que tenemos un directorio es un repositorio de Git si encontramos una carpeta oculta con el nombre `.git`. En ese directorio Git guarda una pequeña base de datos con las distintas versiones históricas de mis archivos y otras cosas que utiliza la aplicación.

Trabajando con repositorios

Para entender Git, lo primero que hay que entender son los estados en los que pueden estar los archivos en un repositorio. En primer lugar todos los archivos se dividen en *tracked* y *untracked*. Es decir los archivos que están siendo **versionados** y se guarda su historia son los que están "trackeados". De los que no nos interesa seguir sus cambios decimos que están sin "trackear" (*untracked*).

En segundo lugar, de los archivos "trackeados" decimos que pueden estar modificados o sin modificar (*modified* y *unmodified*). Es decir que Git sabe cuando editamos un archivo y lo guardamos (en un editor de texto por ejemplo).

Por último los archivos modificados pueden o no estar en el *stage*. El *stage* o *staging area* es el lugar donde están los archivos que van a entrar en el próximo *commit*. Un *commit* es una foto de un momento determinado de nuestra carpeta. En cierta medida es una versión de nuestro proyecto que guardamos y a la que podemos volver de ser necesario.



En resumen los estados son cuatro: *untracked*, sin modificar, modificado y *staged*.

Los comandos de Git relevantes aquí son en esencia dos: `git add` y `git commit`. El comando `add` es multiuso, sirve para empezar a "trackear" archivos y también sirve para agregarlos al *stage*. Solo agrega archivos al *stage* si fueron modificados desde el último *commit* o si no estaban siendo "trackeados" aún. Generalmente lo usamos escribiendo `git add .` que significa agregar todos los archivos del directorio, incluyendo subdirectorios. También se puede usar para poner en el *stage* archivos individuales escribiendo `git add <archivo>`.

Para realizar un *commit* de lo que está en el *stage* usamos `git commit`. Si lo ejecutamos así se abre un editor de texto (generalmente Vim) para que ingresemos un mensaje que describa el por qué del *commit*. Cuando "commiteamos" los archivos que estaban en el *stage* pasan al estado de *unmodified* como muestra la imagen. Podemos evitar que se abra el editor de texto usando el *flag* `-m` (mensaje)

de la siguiente manera: `git commit -m "mensaje"`. También podemos saltear la parte de agregar al *stage* usando `git commit -am "mensaje"`, pero cabe aclarar que no se agregan archivos no "trackeados" al *stage*, a diferencia de usar `git add .`.

Branching

Una característica importante de Git es que permite hacer ramas (*branches*) de manera eficiente. Todo repositorio tiene al menos una rama, generalmente llamada `master` por defecto. Si trabajamos con una sola rama la historia de nuestro repositorio es como una línea del tiempo, un *commit* sucede a otro.

Crear ramas nos permite armar historias divergentes dentro de nuestro desarrollo. Esto es útil porque podemos realizar experimentos fuera de nuestra rama principal y elegir entre fusionar la rama con `master` o descartarla completamente.

Los comandos que usamos son tres: `git branch`, `git merge` y `git checkout`. Si ejecutamos `git branch` sin argumentos nos da una lista de todas las ramas del repositorio. Usando `git branch rama` creamos una nueva rama con el nombre "rama". Conviene darle un nombre que describa el propósito de la rama como `arregla-bug-25` o `algoritmo-factorial-recursivo`. Para cambiar de la rama en la que estamos a otra usamos `git checkout <rama>` reemplazando con el nombre de la rama. Podemos crear una rama y cambiar a la misma usando el comando `git checkout -b <rama>`. Por supuesto podemos eliminar una rama usando `git branch -d <rama>`. Cuando estamos listos para fusionar una rama a la rama `master` conviene cambiar primero a `master` y después fusionar la rama. Lo hacemos de la siguiente manera.

```
$ git branch
On branch hotfix
$ git checkout master
$ git merge hotfix
$ git branch -d hotfix
```

Cabe aclarar que para cambiar de ramas con `checkout` tenemos que estar en un estado que Git llama *working tree clean*. Eso quiere decir que si tenemos cambios que no "commiteamos" Git no me va a dejar cambiar de rama. Una vez que fusionamos una rama y ya no la necesitamos conviene borrarla.

Comandos que muestran información

Para ver la historia de *commits* de un repositorio podemos usar `git log`. Para ver el estado de los archivos en el repositorio `git status` es muy útil. Los dos comandos aceptan varias *flags*. Para ver la ayuda de un comando en particular podemos usar `git help <comando>`. Por ejemplo `git help status` nos da la *man page* de `git status`.

Trabajando con repositorios remotos

Para trabajar con repositorios en la red (generalmente Internet) utilizamos `git push`, `git remote` y `git pull`.

Cuando clonamos un repositorio de GitHub, hacemos cambios localmente y queremos subir esos cambios a GitHub usamos simplemente `git push`. Podemos, y a veces tenemos que ser más explícitos y usamos `git push origin master` donde `origin` se refiere al repositorio remoto (el de GitHub) y `master` a la rama local `master`. También podemos reemplazar `master` por otra rama, y si tenemos el repositorio remoto en GitHub eso nos dará la opción de abrir un *pull request* o PR.

Pero si tenemos un repositorio local, que aún no está en un servidor, usamos `git remote add origin <url>` para enlazar nuestro repositorio local con un repositorio remoto. El nombre `origin` es una convención, pueden ponerle otro nombre, aunque no lo recomiendo. Cuando hacemos un *push* por primera vez, conviene usar la opción `-u` así `git push -u origin master` para decirle a Git que "trackee" el repositorio remoto y me diga si hay diferencias entre la versión local y la que está en Internet.

Si estamos trabajando con más programadores, y suponemos que entre la última vez que clonamos el repositorio alguien más pudo hacer cambios en el servidor usamos `git pull` para traer los cambios de la rama `master` remota a la rama `master` local de nuestra computadora.

GitHub

GitHub es una plataforma online para subir repositorios de Git, trabajar colaborativamente con otros programadores y varias cosas más.

No es el único lugar que provee *hosting* de repositorios de Git, pero sí el más utilizado. Para nosotros lo más importante de GitHub es lo que se conoce como *pull requests*. Las *pull requests* son solicitudes que un colaborador realiza a los otros colaboradores de un repositorio de fusionar una rama con código nuevo a `master`. Las *pull requests* permiten a los colaboradores discutir los cambios en una especie de foro o chat antes de realizar el *merge*.

Ejercicios

Completar...