22 DE ENERO DE 2023

PAT:
PRÁCTICA 1

Instalación de herramientas

Ernesto Hidalgo Felipe
3º GITT+BA

Contenido

1. Objetivo de la práctica

Esta práctica busca tener unas nociones de cómo usar GitHub como plataforma de desarrollo, Git sistema de control de versiones de código fuente de Software y empezar a trabajar con sus comandos y funcionalidades.

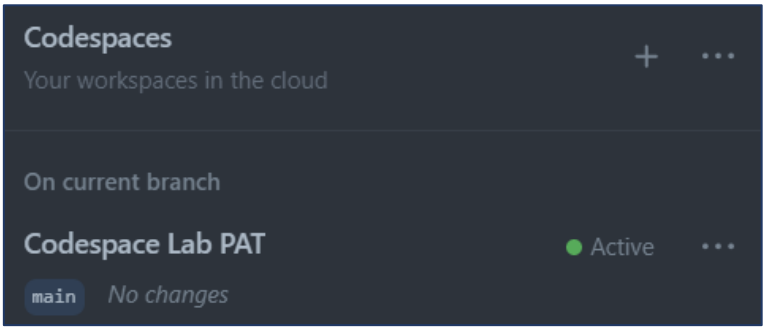
2. Desarrollo

2.1 Prerrequisitos

Para trabajar con el repositorio proporcionado, el primer paso es crearnos una cuenta de GitHub. GitHub utiliza el sistema de control de versiones Git, y nos sirve para alojar en la nube nuestros proyectos. En mi caso, mi nombre de usuario en la plataforma es @merovynn.

Una vez creada, hacemos un ***fork*** del repositorio github.com/gitt-3-pat/hello-world. Lo que conseguimos con esto es replicar dicho repositorio para poder trabajar con él. Si los cambios los hiciésemos desde nuestro Git sobre el primer repositorio, al pretender actualizarlos en la nube se nos denegaría la petición, ya que no tenemos permisos sobre él. Al replicarlo, creamos una copia que nos pertenece y que podemos modificar como queramos.

Tras hacer el *fork*, procedemos estableciendo un ***codespace*** de Git. Si tuviésemos Git instalado en nuestro ordenador podríamos omitir este paso, pero se nos da la oportunidad de trabajar desde un entorno de desarrollo en la nube de GitHub que simula el programa y nos ahorra el descargárnoslo y configurarlo.



Cabe resaltar que GitHub limita el uso del *codespace* a 60 horas mensuales de uso, por lo que tendremos que tener cuidado con cuánto y cómo lo usamos. También es recomendable limitar a 15 minutos el tiempo de inactividad previo a la suspensión del espacio.

2.2 Ejecución de comandos

Comenzamos entrando en el *codespace* ya creado. Dentro de este procedemos a realizar un ejemplo de línea de código con cada comando de entre los más básicos:

- git clone

El comando ***clone*** nos sirve para replicar un repositorio a un directorio local para así trabajar con él. A diferencia del *fork* que realizamos antes, la clonación crea una copia local completamente independiente del repositorio origen.

Para probarlo en nuestro repositorio nos desplazamos a la carpeta temporal /tmp, en la cual situamos la copia exacta del repositorio /hello-world del que disponíamos.

```
@merovynn ➜/workspaces/hello-world (main) $ cd /tmp
@merovynn ➜/tmp $ git clone https://github.com/merovynn/hello-world
Cloning into 'hello-world'...
remote: Enumerating objects: 41, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 41 (delta 1), reused 3 (delta 1), pack-reused 34
Unpacking objects: 100% (41/41), 59.81 KiB | 1.50 MiB/s, done.
```

Así, nos podemos desplazar a la clonación y comprobar su contenido, que es el mismo que teníamos en el repositorio original.

```
@merovynn ➜/tmp $ dir
codespaces_logs  hsperfdata_codespace        vscode-ipc-01c63a20-4716-497a-91
dockerd.log      sshd.log                    vscode-ipc-67ba1ce6-022a-43b2-8d
hello-world      vscode-git-746d5a5694.sock  vscode-ipc-74f733f6-ccb2-4fb6-a7
@merovynn ➜/tmp $ cd hello-world
@merovynn ➜/tmp/hello-world (main) $ dir
comandos.txt  LICENSE  mvnw  mvnw.cmd  pom.xml  README.md  src
```

- git status, add, commit

El comando ***status*** nos sirve para comprobar el estado de los archivos en nuestro proyecto (si se han creado, modificado o eliminado). A la hora de trabajar con Git, nos valdremos de los comandos *add* y *commit* para indexar los cambios y versionarlos. *status* nos permite ver en qué estado está nuestro proyecto: qué archivos han cambiado y cuáles de estos cambios se consideran para el siguiente *commit*.

Así, empezamos creando un archivo *pruebastatus.txt*, para luego ejecutar por primera vez *status*.

```
@merovynn ➜/workspaces/hello-world (main X) $ git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        pruebastatus.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Como podemos apreciar, la rama está sincronizada, pero se menciona el nuevo archivo, todavía no considerado para el *commit*.

Procedemos a utilizar ***add***, que añadirá al index nuestro nuevo archivo.

```
@merovynn →/workspaces/hello-world (main X) $ git add .
@merovynn →/workspaces/hello-world (main) $ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   pruebastatus.txt
```

Como vemos, el cambio está listo para el *commit*. También se podría haber indicado *add <ruta-de-archivo>*, ya que lo que consigue el punto utilizado es añadir al index todos los cambios existentes y no siempre es lo que vamos a querer.

Una vez añadido ejecutamos ***commit***, empaquetando los cambios y describiéndolos con un mensaje breve. El *commit* es como una "fotografía" de nuestro proyecto en este momento.

```
@merovynn →/workspaces/hello-world (main) $ git commit -m "Segunda prueba de P1"
[main 0c9ae02] Segunda prueba de P1
 1 file changed, 1 insertion(+)
 create mode 100644 pruebastatus.txt
@merovynn →/workspaces/hello-world (main) $ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)
```

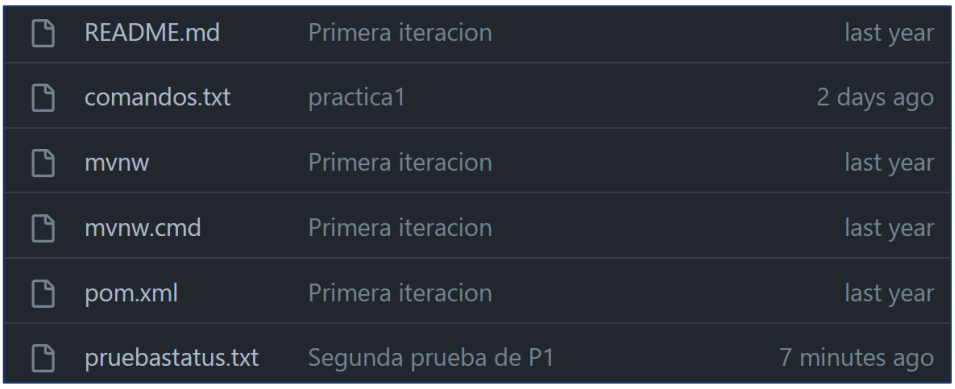
Ahora sí, el *status* no menciona ningún cambio no registrado. Hemos hecho *commit* con éxito.

- git push

Si nos fijamos en el último *status*, podemos ver que se menciona que nuestra rama está un *commit* por delante del repositorio original. Esto se debe a que, pese a haber registrado los cambios, todavía no los hemos publicado. Para ello existe el comando ***push***, que actualiza el repositorio original, el que se encuentra en nuestro GitHub, con los cambios locales que hemos realizado.

```
@merovynn →/workspaces/hello-world (main) $ git push origin main
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 2 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 356 bytes | 356.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/merovynn/hello-world
   e7ca55b..0c9ae02  main -> main
```

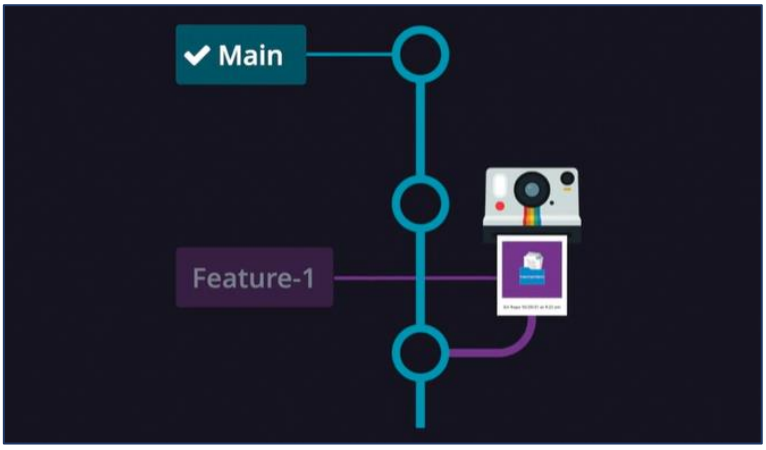
Al revisar el repositorio en GitHub podemos comprobar que, en efecto, el cambio se ha registrado. No solo eso; también podemos ver a qué *commit* concreto corresponde cada cambio, herramienta muy útil a la hora de elaborar proyectos modulares y/o entre varias personas.



- git checkout

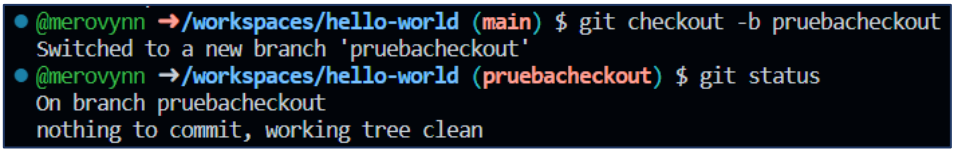
El último comando de entre los básicos a investigar es el comando *checkout*. Este sirve para cambiar entre ramas en las que estemos trabajando, crear nuevas y revisar antiguas versiones de estas, entre otras posibilidades.

Para entender mejor el concepto de ramas, podemos estudiar la siguiente ilustración:



Como vemos, una rama nos permite realizar cambios, o bien ver los cambios que han propuesto otros usuarios, sin necesidad de alterar el camino principal del programa.

Comenzamos creando una rama local *pruebacheckout* con el comando *git* ***checkout*** *-b /pruebacheckout*.



A partir de ahora, los cambios que realicemos los publicaremos en esta nueva rama de prueba.

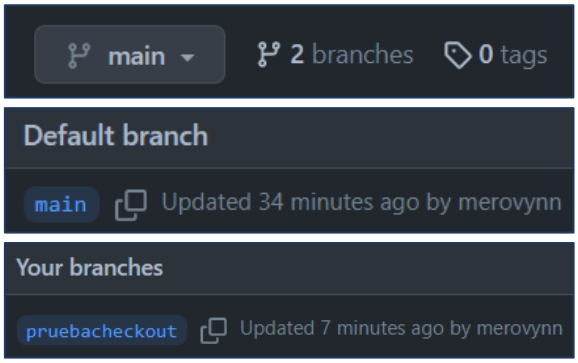
Por ejemplo, creamos un simple archivo *.txt*. Procedemos a ejecutar el correspondiente *add*, *commit* y *push*.

```
@merovynn →/workspaces/hello-world (pruebacheckout X) $ git add .
@merovynn →/workspaces/hello-world (pruebacheckout) $ git status
On branch pruebacheckout
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   pruebacheckout.txt

@merovynn →/workspaces/hello-world (pruebacheckout) $ git commit
[pruebacheckout 8edb1e8] Prueba rama 2
 1 file changed, 1 insertion(+)
 create mode 100644 pruebacheckout.txt
```

```
@merovynn →/workspaces/hello-world (pruebacheckout) $ git push --set-upstream origin pruebacheckout
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 2 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 318 bytes | 318.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'pruebacheckout' on GitHub by visiting:
remote:      https://github.com/merovynn/hello-world/pull/new/pruebacheckout
remote:
To https://github.com/merovynn/hello-world
 * [new branch]      pruebacheckout -> pruebacheckout
Branch 'pruebacheckout' set up to track remote branch 'pruebacheckout' from 'origin'.
```

Si nos dirigimos al repositorio *hello-world* de GitHub veremos que aparecen ambas ramas. La primera, la principal, no contiene este último archivo, mientras que la secundaria, *pruebacheckout*, sí lo incluye.



Y con el comando *git branch -a* podemos revisar qué ramas existen actualmente.

```
@merovynn →/workspaces/hello-world (pruebacheckout) $ git branch -a
  main
* pruebacheckout
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
  remotes/origin/pruebacheckout
```

Para cambiar de una a otra bastará con ejecutar ***checkout*** *<nombre-de-rama>*.

· tracking branches (adicional)

A modo de indagar más en este tema de las ramas, podemos hablar de un tipo concreto de ramas: las *tracking branches*. Estas se asocian a una rama remota directamente, por lo que cualquier cambio que se incluya en un *push* (o cualquiera que venga de un *pull*) corresponderá con la rama asociada.

Para sincronizar nuestro trabajo con el de la rama remota, llamamos al *fetch*, que actualizará nuestra rama con aquella a la que se asocia. Esta funcionalidad es considerablemente útil si queremos, por ejemplo, que distintos departamentos de un mismo proyecto trabajen a partir de distintas ramas maestras sin dificultar el progreso del resto de equipos, que pueden tener otros fines en ese momento.

También podría ser útil si nuestro proyecto bebe de distintos repositorios, ya que permite interactuar con estos por separado.

3. Descarga de programas

Las siguientes capturas corroboran que el equipo incorpora los programas indicados en el enunciado (Java, Maven, Docker y un editor de código fuente, VSCode).

```
C:\Users\Ernes>java --version
java 16.0.1 2021-04-20
Java(TM) SE Runtime Environment (build 16.0.1+9-24)
Java HotSpot(TM) 64-Bit Server VM (build 16.0.1+9-24, mixed mode, sharing)

C:\Users\Ernes>mvn --version
Apache Maven 3.8.7 (b89d5959fcde851dcb1c8946a785a163f14e1e29)
Maven home: C:\Program Files\apache-maven-3.8.7-bin\apache-maven-3.8.7
Java version: 16.0.1, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-16.0.1
Default locale: es_ES, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

4. Referencias

https://github.com/gitt-3-pat/hello-world
https://git-scm.com/
https://www.gitkraken.com/learn/git/git-checkout
https://git-scm.com/book/en/v2/Git-Branching-Remote-Branches
https://stackoverflow.com/questions/4693588/what-is-a-tracking-branch