ENERO 2024

PRACTICA 1: ENTORNO DE DESARROLLO

FERNANDO PEREZ ALBA

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.

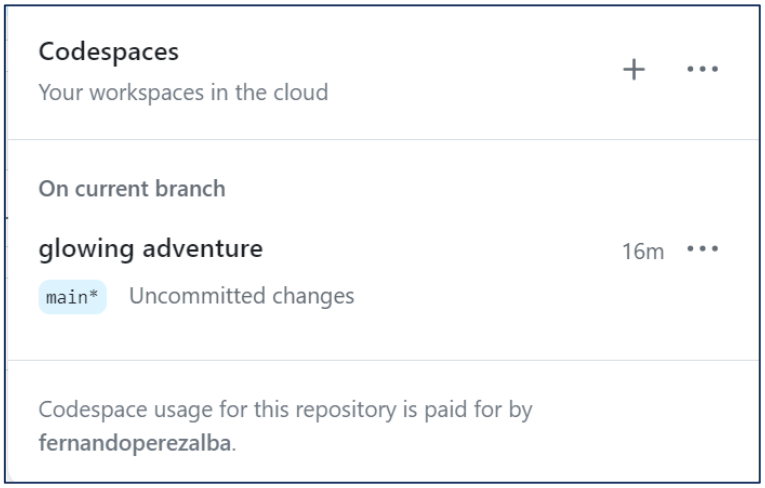
2. Desarrollo

2.1 Prerrequisitos

Para iniciar el trabajo con el repositorio proporcionado, el paso inicial es registrarse y crear una cuenta en GitHub. Esta plataforma emplea el sistema de control de versiones Git y es ideal para hospedar proyectos en la nube. Por ejemplo, mi nombre de usuario en GitHub es @fernandoperezalba.

Después de establecer nuestra cuenta, procedemos a realizar un fork del repositorio en https://github.com/gitt-3-pat/p1. Al hacer esto, estamos duplicando el repositorio original para poder trabajar sobre él de manera independiente. Si intentáramos hacer cambios directamente en el repositorio original desde nuestro Git, nuestra solicitud de actualización sería rechazada en la nube, debido a la falta de permisos. Al duplicar el repositorio, obtenemos una copia propia que podemos alterar según nuestras necesidades.

Después de realizar el fork, seguimos con la configuración de un codespace de Git. Este paso es opcional si ya tenemos Git instalado en nuestra computadora; sin embargo, GitHub nos ofrece la ventaja de trabajar en un entorno de desarrollo en la nube. Este entorno simula el funcionamiento de Git, evitando la necesidad de descargar y configurar el programa en nuestro equipo local.



Es importante mencionar que GitHub establece un límite de 120 horas mensuales para el uso del codespace. Por tanto, es esencial ser conscientes de cuánto tiempo y de qué manera lo utilizamos. Además, se aconseja configurar el codespace para que se suspenda automáticamente tras 15 minutos de inactividad, con el fin de optimizar el uso de estas horas disponibles.

2.2 Ejecución de comandos

niciamos accediendo al codespace previamente creado. Una vez dentro, procederemos a demostrar un ejemplo de línea de código utilizando algunos de los comandos más fundamentales:

- ### git clone

El comando clone se utiliza para copiar un repositorio a un directorio local, permitiéndonos trabajar con él. A diferencia del fork que se llevó a cabo anteriormente, clonar un repositorio genera una copia local que es completamente independiente del repositorio original.

- ### git status, add, commit

El comando status es útil para verificar el estado actual de los archivos en nuestro proyecto, identificando si se han creado, modificado o eliminado. En el flujo de trabajo con Git, utilizamos los comandos add y commit para indexar y versionar los cambios. El comando status nos proporciona una visión clara del estado en el que se encuentra nuestro proyecto: muestra qué archivos han experimentado cambios y cuáles de estos cambios están preparados para ser incluidos en el próximo commit.

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

```
@fernandoperezalba →/workspaces/p1-fork (main) $ 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)
        p1/
        pruebstatus.txt

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

Como se observa, la rama está actualmente sincronizada. Sin embargo, se hace mención de un nuevo archivo que aún no ha sido incluido para el próximo commit.

A continuación, utilizaremos el comando add, el cual añadirá nuestro nuevo archivo al índice de Git, preparándolo así para ser incluido en el siguiente commit..

```
@fernandoperezalba →/workspaces/p1-fork (main) $ git add .
@fernandoperezalba →/workspaces/p1-fork (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:   p1
        new file:   pruebstatus.txt
```

Correcto, ahora el cambio está preparado para el commit. Cabe destacar que también se podría haber utilizado el comando add <ruta-del-archivo> para ser más específicos. El uso del punto en el comando add . añade todos los cambios existentes al índice, lo cual no siempre es lo deseado, dependiendo de la situación.

Una vez que hemos añadido los cambios al índice con add, procedemos a ejecutar el comando commit. Este paso es como tomar una “fotografía” de nuestro proyecto en su estado actual, empaquetando los cambios y describiéndolos con un mensaje breve y explicativo. Este proceso es fundamental para llevar un registro claro y detallado de la evolución del proyecto.

```
@fernandoperezalba →/workspaces/p1-fork (main) $ git commit -m "Prueba commit"
[main a5d5b13] Prueba commit
 2 files changed, 1 insertion(+)
 create mode 160000 p1
 create mode 100644 pruebstatus.txt
@fernandoperezalba →/workspaces/p1-fork (main) $ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

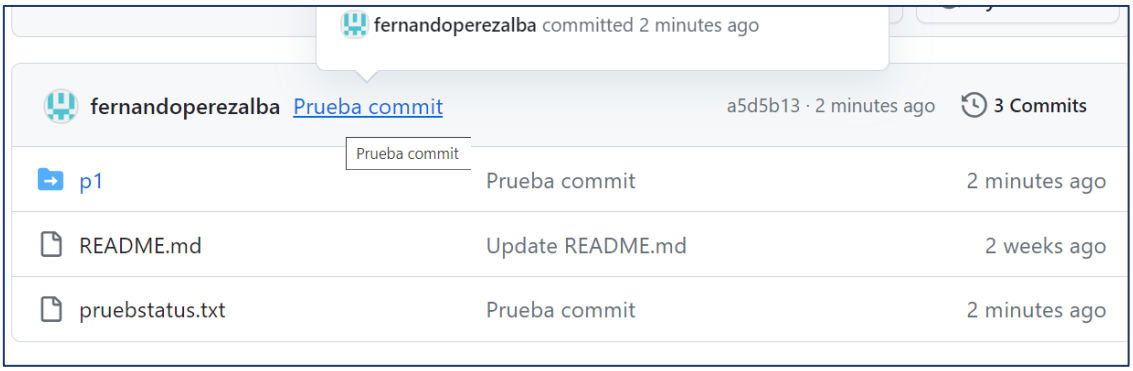
Ahora sí, el *status* no menciona ningún cambio no registrado. Hemos hecho *commit* con éxito.

- git push

Si revisamos el último mensaje de status, notaremos que indica que nuestra rama está un commit adelante del repositorio original. Esto ocurre porque, aunque hemos registrado los cambios localmente, todavía no los hemos publicado en el repositorio remoto. Para actualizar el repositorio original (el que se encuentra en nuestro GitHub) con los cambios locales, utilizamos el comando push. Este comando es el que finalmente sincroniza nuestro trabajo local con el repositorio en la nube, asegurando que todos los cambios recientes estén disponibles y visibles en GitHub.

```
@fernandoperezalba →/workspaces/p1-fork (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% (2/2), done.
Writing objects: 100% (3/3), 332 bytes | 332.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/fernandoperezalba/p1-fork
   07720b5..a5d5b13  main -> main
```

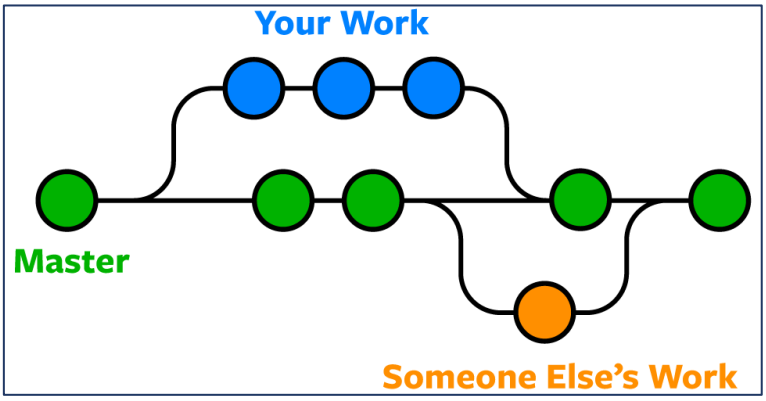
Al acceder al repositorio en GitHub, podemos verificar que el cambio ha sido registrado exitosamente. Además, GitHub ofrece la capacidad de ver a qué commit específico corresponde cada cambio. Esta característica es extremadamente valiosa para proyectos modulares o aquellos desarrollados en colaboración. Permite a los colaboradores seguir el historial de cambios de manera precisa, comprendiendo quién hizo cada modificación y en qué momento. Esto facilita enormemente la coordinación y el seguimiento del progreso en entornos de desarrollo colaborativo.



- git checkout

El último comando fundamental que exploraremos es checkout. Este comando se utiliza para varias funciones esenciales en Git, como cambiar entre diferentes ramas del proyecto, crear ramas nuevas y revisar versiones anteriores de estas. El checkout es una herramienta versátil que permite a los desarrolladores navegar y gestionar eficientemente las diversas líneas de desarrollo dentro de un proyecto. Por ejemplo, puedes cambiar a una rama específica para trabajar en una nueva característica o revisar el estado de tu proyecto en un punto anterior, todo ello utilizando este comando.

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



Las ramas de GitHub son una herramienta fundamental para la gestión de proyectos, ya que permiten a los usuarios desarrollar nuevas características, realizar experimentos o corregir errores sin alterar la rama principal del proyecto. Esto es especialmente útil para mantener la integridad del código principal mientras se llevan a cabo diferentes tareas de desarrollo.

Cuando se trabaja con un repositorio en GitHub, se pueden ver todas las ramas existentes. Cada rama representa una línea independiente de desarrollo. Para ver una lista completa de las ramas disponibles en un repositorio local, se utiliza el comando git branch -a. Este comando muestra todas las ramas, tanto locales como remotas.

Para moverse entre diferentes ramas, se emplea el comando checkout <nombre-de-rama>. Esto permite a los desarrolladores alternar entre las diversas ramas del proyecto, facilitando la revisión y la contribución a diferentes aspectos del mismo. Por lo tanto, las ramas en GitHub son esenciales para un flujo de trabajo colaborativo y eficiente, permitiendo a múltiples usuarios trabajar simultáneamente en diferentes secciones de un mismo proyecto sin interferir unos con otros.
Por supuesto al acabar este informe hay que realizarlo todo otra vez con este archivo para que figure en el repositorio.

- tracking branches (adicional)

Las ramas de seguimiento, o "tracking branches", en Git son un concepto específico que amplía la funcionalidad de las ramas tradicionales. Estas ramas se establecen para tener una correspondencia directa con ramas remotas en un repositorio en GitHub u otro sistema de control de versiones remoto. La principal ventaja de una rama de seguimiento es que simplifica el proceso de mantener sincronizados los cambios locales con los del repositorio remoto.

Cuando realizas un push en una rama de seguimiento, los cambios se envían directamente a la rama remota asociada. De manera similar, al ejecutar un pull, los cambios de la rama remota se integran automáticamente en tu rama de seguimiento local. Esta sincronización bidireccional es esencial para un flujo de trabajo colaborativo eficiente.

El comando fetch juega un papel importante en este proceso. Al llamar a fetch, Git actualiza la información de tu rama local con la rama remota asociada, permitiendo ver los cambios más recientes sin necesariamente fusionarlos en tu trabajo actual. Esto es útil para mantenerse al tanto de los desarrollos en la rama remota sin afectar tu propio progreso.

Este tipo de ramas es particularmente valioso en escenarios donde diferentes departamentos o equipos trabajan en distintas secciones de un proyecto más grande. Cada equipo puede trabajar de manera independiente en su propia rama maestra, sin interferir con el trabajo de los demás. Además, para proyectos que dependen o interactúan con varios repositorios, las ramas de seguimiento facilitan la gestión y el seguimiento de los cambios en estos repositorios de manera ordenada y eficiente.

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\fpa20>mvn -version
Apache Maven 3.9.6 (bc0240f3c744dd6b6ec2920b3cd08dcc295161ae)
Maven home: C:\Program Files\apache-maven-3.9.6
Java version: 17.0.10, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-17
Default locale: es_ES, platform encoding: Cp1252
OS name: "windows 11", version: "10.0", arch: "amd64", family: "windows"

C:\Users\fpa20>java --version
java 17.0.10 2024-01-16 LTS
Java(TM) SE Runtime Environment (build 17.0.10+11-LTS-240)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.10+11-LTS-240, mixed mode, sharing)
```

4. Referencias

https://github.com/gitt-3-pat/p1.
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