

Introducción a Git / GitHub



¿Qué es [Git](#)?

Git es una herramienta de control de versiones para gestionar el control de cambios en el código fuente y otros archivos. Permite a los desarrolladores trabajar en un mismo proyecto de forma colaborativa y coordinada, manteniendo un historial de cambios y versiones. Es una herramienta muy utilizada en el desarrollo de software y se considera esencial para cualquier programador o equipo de desarrollo.

¿Qué es [GitHub](#)?

GitHub es una plataforma en línea que se utiliza para alojar y compartir proyectos de software utilizando Git como sistema de control de versiones. Permite a los desarrolladores alojar repositorios de código fuente, colaborar en proyectos, realizar un seguimiento de los cambios y versiones, y gestionar problemas y solicitudes de cambios. Además, GitHub también ofrece herramientas de gestión de proyectos y colaboración, como wikis y tableros de proyectos.



Diferencias entre ambos

Git y Github son dos herramientas diferentes pero relacionadas que se utilizan comúnmente en la gestión de versiones de software y en la colaboración en proyectos de programación. A continuación, se presentan algunas de las principales diferencias entre Git y Github:

- Git es un sistema de control de versiones de código fuente distribuido, mientras que Github es un servicio en línea que proporciona almacenamiento en la nube y una plataforma de colaboración para proyectos de software que utilizan Git como sistema de control de versiones.
- Git es una herramienta de línea de comandos que se instala en la computadora del usuario, mientras que Github es una aplicación web que se ejecuta en línea y se puede acceder a través de un navegador web.
- Git se utiliza para realizar un seguimiento de los cambios en el código fuente y mantener un historial completo de cada cambio, mientras que Github se utiliza para alojar repositorios Git y proporcionar herramientas para colaborar y administrar el código fuente.
- Git se puede utilizar para trabajar en proyectos de software sin conexión a Internet, mientras que Github requiere una conexión a Internet para la mayoría de las funciones.
- Git es una herramienta de código abierto y se puede utilizar de forma gratuita, mientras que Github ofrece planes de pago que proporcionan características adicionales, como repositorios privados y herramientas de colaboración avanzadas.

En resumen, Git es un sistema de control de versiones de código fuente distribuido y Github es un servicio en línea que proporciona almacenamiento en la nube y herramientas de colaboración para proyectos de software que utilizan Git como sistema de control de versiones. Git se utiliza para realizar un seguimiento de los cambios en el código fuente y mantener un historial completo de cada cambio, mientras que Github se utiliza para alojar repositorios Git y proporcionar herramientas para colaborar y administrar el código fuente.

Instalación / Configuración

Según tu sistema operativo podrás descargar desde la [web principal Git](#). Podremos instalar a través de terminal o con un cliente de interfaz gráfica.

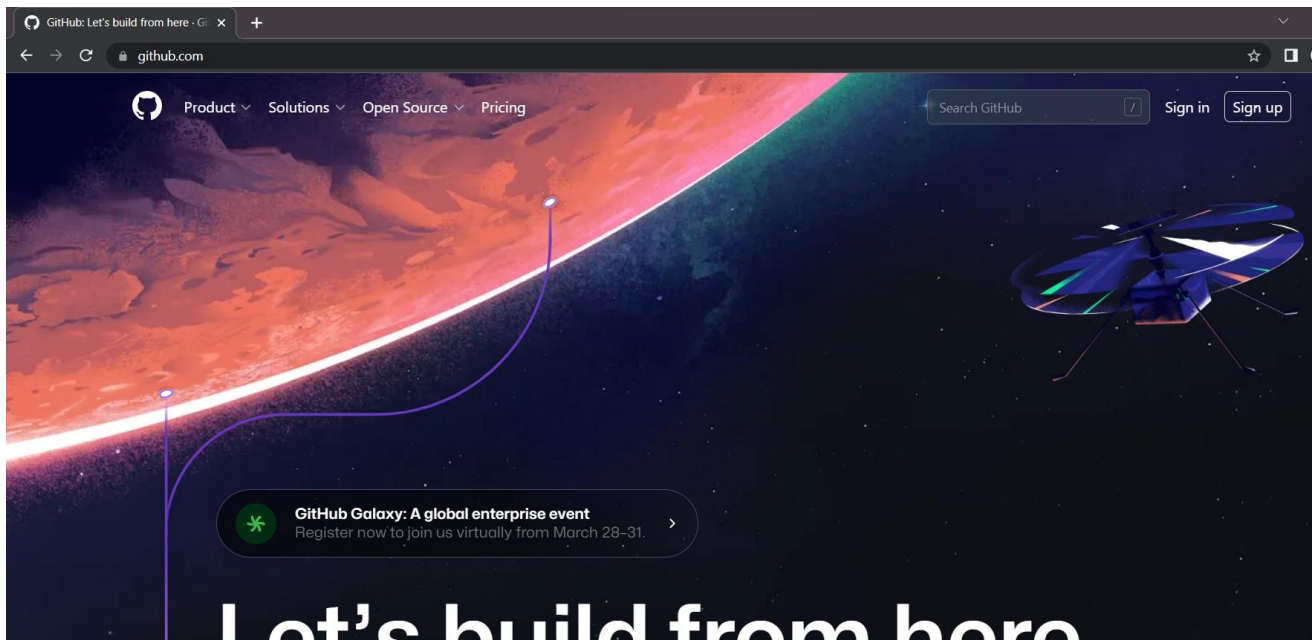


Como primer paso revisaremos si tenemos Git en nuestro sistema, abrimos nuestra terminal y escribimos **git --version**

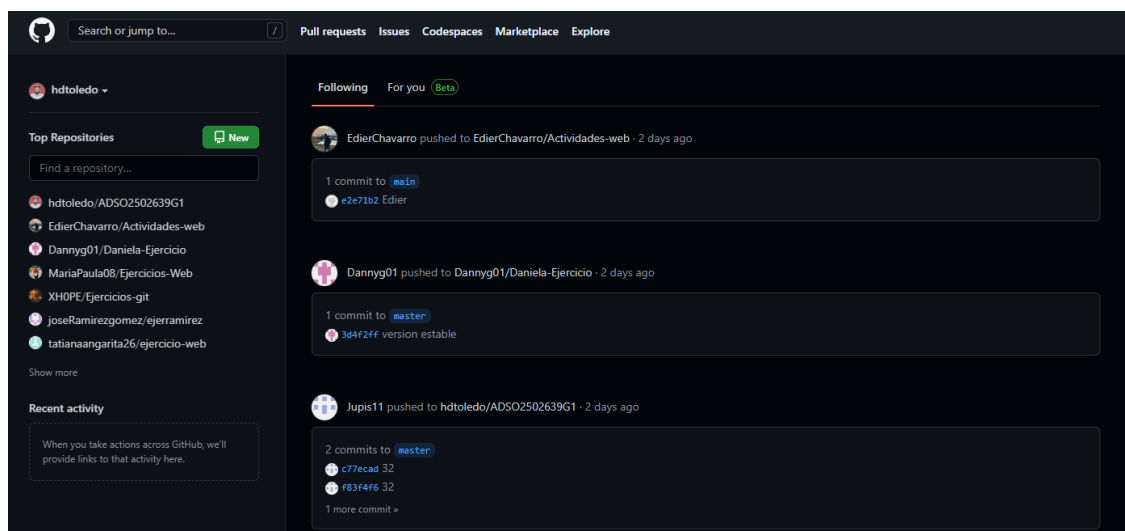
```
hdtolledo@HDMSIGF65THIN: ~$ git --version
git version 2.25.1
hdtolledo@HDMSIGF65THIN: ~$
```

Sino lo tenemos instalaremos el paquete según la versión de nuestro sistema operativo.

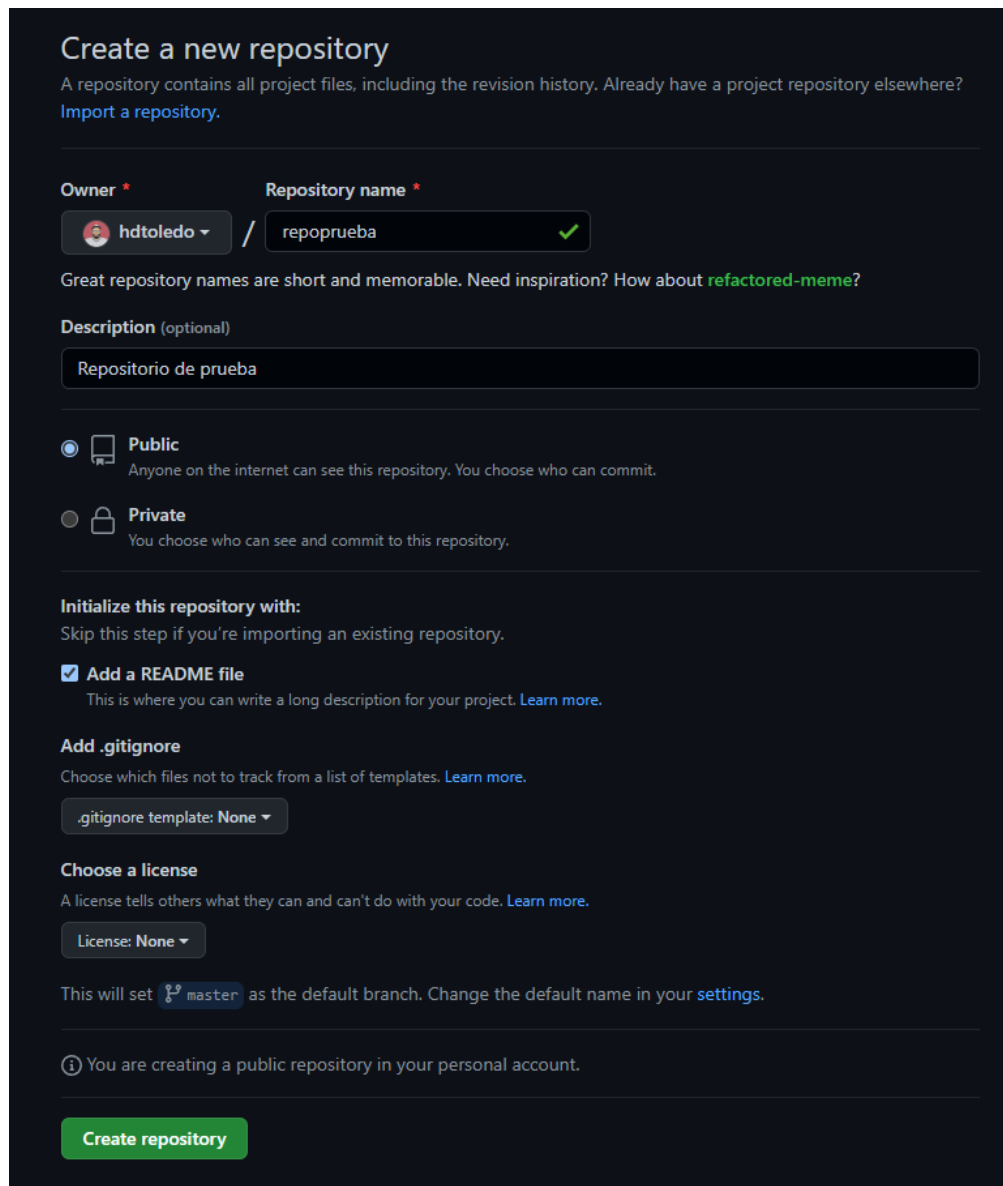
Si ya lo tenemos simplemente pasaremos al paso de la creación de nuestra cuenta en [github](https://github.com) o iniciar la sesión.



Una vez que nos registremos o iniciemos sesión veremos la interfaz de bienvenida en donde nos dirigimos a crear nuestro nuevo repositorio.



Iniciaremos con un repositorio de prueba, en donde estableceremos varias prácticas.



The screenshot shows the 'Create a new repository' page on GitHub. At the top, it says 'Create a new repository' and provides a brief explanation of what a repository is. Below this, there are two main sections: 'Owner' and 'Repository name'. The 'Owner' is set to 'hdtledo' and the 'Repository name' is 'repoprueba'. A green checkmark is next to the repository name. Below these fields, there is a suggestion for repository names: 'Great repository names are short and memorable. Need inspiration? How about **refactored-meme**?'. The 'Description' field is optional and contains the text 'Repositorio de prueba'. There are two radio buttons for visibility: 'Public' (selected) and 'Private'. Below these, there is a section 'Initialize this repository with:' which includes a checkbox for 'Add a README file' (checked) and a dropdown for '.gitignore template' (set to 'None'). There is also a section 'Choose a license' with a dropdown set to 'None'. At the bottom, there is a green button labeled 'Create repository'.

¿Público o Privado?

La principal diferencia entre un repositorio público y uno privado en GitHub es la visibilidad de su contenido.

Repositorio público: es visible para cualquier persona en Internet y puede ser encontrado mediante búsquedas en GitHub y en motores de búsqueda como Google. Cualquier usuario de GitHub puede ver y clonar el repositorio público, así como hacer "**fork**" (crear su propia copia) y enviar "**pull requests**" (solicitudes de cambios).

Repositorio privado: es visible solo para los usuarios que tienen acceso al repositorio. Los colaboradores que no están en la lista de colaboradores del repositorio no podrán ver ni clonar el contenido. Además, los motores de búsqueda no indexarán el contenido del repositorio, por lo que no se mostrará en los resultados de búsqueda.



Los repositorios privados son útiles para proyectos que contienen información confidencial o propiedad intelectual que no se desea que sea pública. También son útiles para proyectos que aún están en desarrollo y no están listos para ser compartidos públicamente.

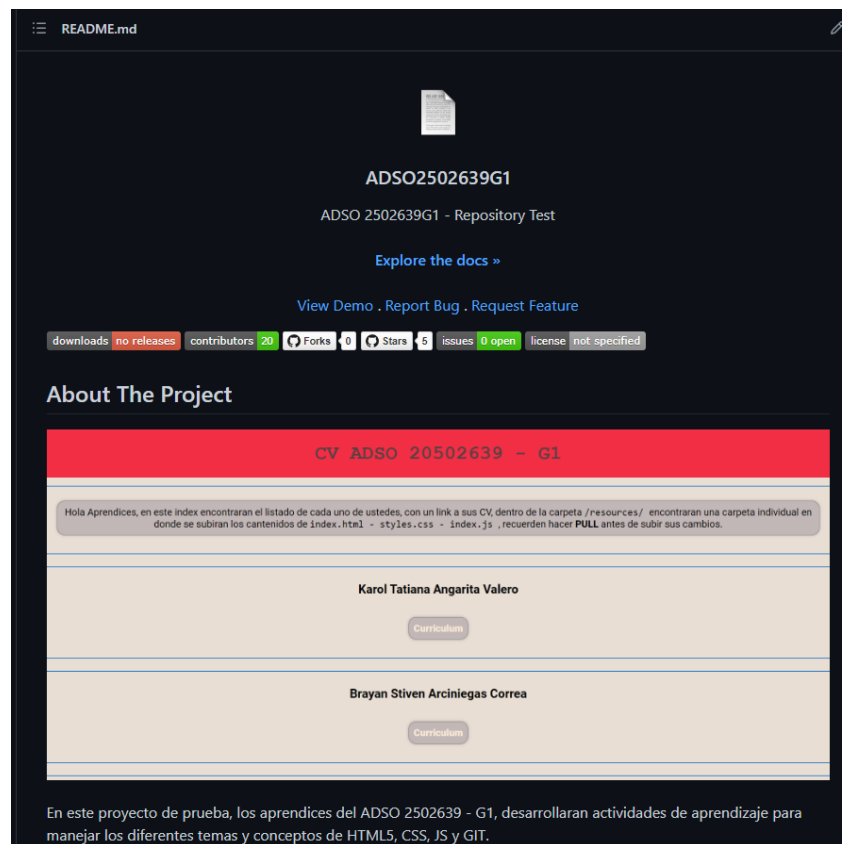
Los repositorios públicos son excelentes para proyectos de código abierto, ya que permiten la colaboración abierta y la participación de la comunidad. También son útiles para proyectos educativos, proyectos personales y proyectos que buscan una mayor visibilidad y exposición en la comunidad de GitHub.

¿Readme?

Un archivo README.md es un archivo de texto que se utiliza para proporcionar información y documentación sobre un proyecto de software. El propósito principal de un archivo README.md es proporcionar una descripción general del proyecto, explicar cómo se instala y se utiliza, y proporcionar información sobre cómo contribuir al proyecto.

El archivo README.md es una parte importante del repositorio de un proyecto, ya que es la primera fuente de información que los usuarios y los colaboradores verán cuando visiten el repositorio en línea. Proporciona información sobre el propósito del proyecto, los requisitos de instalación, la estructura del directorio, las instrucciones de uso, las posibles limitaciones y problemas conocidos, y cualquier otra información relevante.

El archivo README.md también se puede utilizar para proporcionar información sobre cómo colaborar en el proyecto, cómo informar problemas y cómo contribuir con código. En resumen, el archivo README.md es una forma importante de comunicar información crítica sobre un proyecto a sus usuarios y colaboradores.



¿.gitignore?

Un archivo .gitignore es un archivo de texto que se utiliza para especificar qué archivos o carpetas deben ser ignorados por Git al rastrear cambios en un repositorio.

Cuando trabajas en un proyecto, es posible que haya archivos o carpetas que no desees incluir en el control de versiones de Git. Por ejemplo, los archivos generados automáticamente por herramientas de compilación, los archivos de configuración local, los archivos de caché y los archivos de registro pueden no ser útiles para otros desarrolladores que colaboran en el mismo proyecto.

Para evitar que estos archivos sean incluidos en el control de versiones de Git, se puede crear un archivo .gitignore en la raíz del repositorio. Este archivo contiene una lista de patrones de nombres de archivos que Git debe ignorar al rastrear cambios en el repositorio. Por ejemplo, si deseas ignorar todos los archivos con extensión ".log", puedes agregar la siguiente línea al archivo .gitignore:

```
bash
*.log
```

También puedes utilizar patrones más complejos y especificar carpetas enteras que deben ser ignoradas. Por ejemplo, si deseas ignorar todos los archivos en una carpeta llamada "temp", puedes agregar la siguiente línea al archivo .gitignore:

```
temp/
```

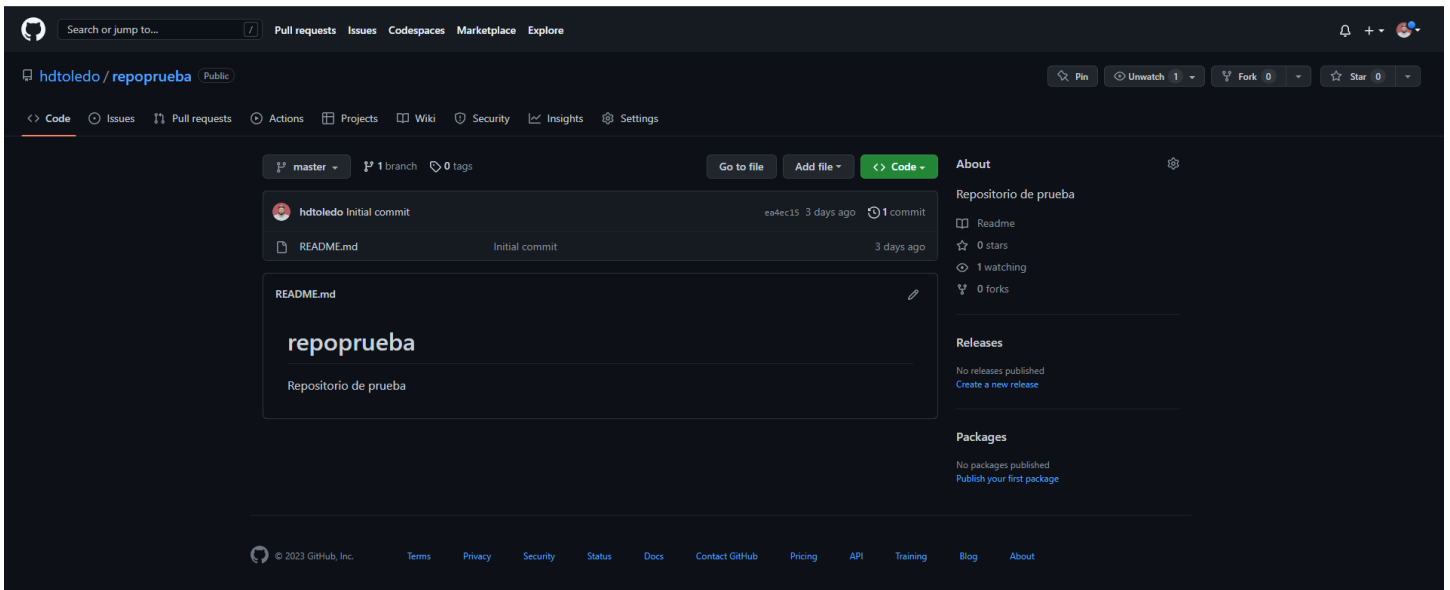
Es importante tener en cuenta que, una vez que un archivo o carpeta se ha agregado al control de versiones de Git, no se puede ignorar con un archivo .gitignore. Para evitar que un archivo o carpeta ya rastreados sean incluidos en el control de versiones, debes eliminarlos del repositorio utilizando el comando **git rm** y luego hacer un **commit** de los cambios.

En resumen, un archivo .gitignore es una herramienta importante para evitar la inclusión de archivos o carpetas no deseados en el control de versiones de Git, lo que ayuda a mantener el repositorio limpio y ordenado.

¿Qué sigue?

Una vez que tengamos listo nuestro repositorio en github, lo siguiente será clonarlo y dejarlo listo en nuestro equipo, la vista que tendrás de tu repositorio será la siguiente:



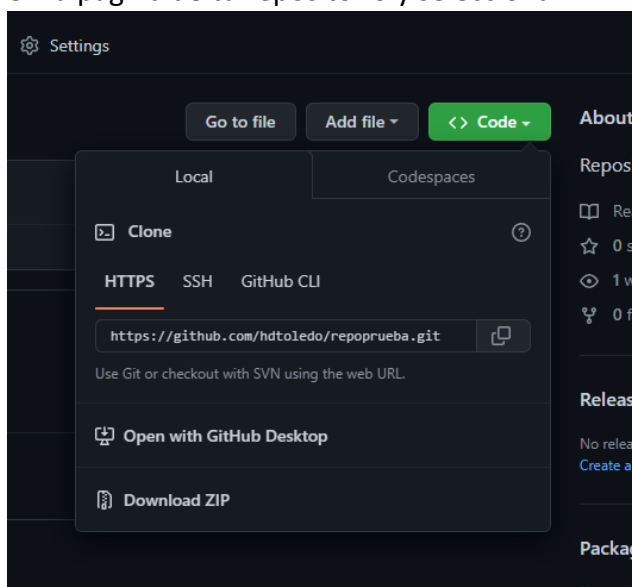


¡Listo! Ahora, para configurar el repositorio en tu equipo y comenzar a trabajar en él, sigue los siguientes pasos:

- Abre tu terminal o línea de comandos y navega hasta la carpeta donde deseas clonar el repositorio de GitHub. Puedes usar el comando `cd` para cambiar de directorio.



- Copia la URL de clonación de tu repositorio en GitHub. Para hacerlo, haz clic en el botón verde "Code" en la página de tu repositorio y selecciona "HTTPS". Copia la URL que aparece.



- En tu terminal, escribe el siguiente comando para clonar el repositorio en tu equipo:

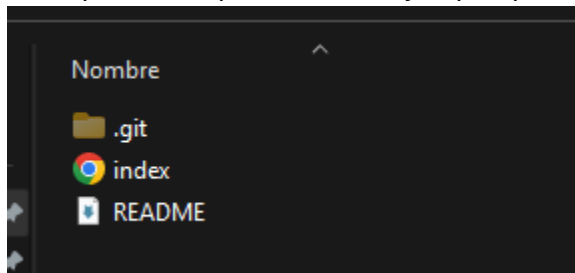
```
git clone <URL del repositorio>
```

```
MINGW64:/d/Data/Escritorio
hdtol@HDMSIGF65THIN MINGW64 /d/Data/Escritorio
$ git clone https://github.com/hdtoledo/repoprueba.git
Cloning into 'repoprueba'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

- Ahora, navega al directorio que acabas de clonar con el siguiente comando:

```
hdtol@HDMSIGF65THIN MINGW64 /d/Data/Escritorio
$ cd repoprueba/
```

- Para comenzar a trabajar en tu repositorio, crea un nuevo archivo en tu equipo y guárdalo en la carpeta del repositorio "prueba". Por ejemplo, puedes crear un archivo llamado "index.html".



- Luego, agrega el archivo a tu repositorio local con el siguiente comando:

```
hdtol@HDMSIGF65THIN MINGW64
$ git add .
```

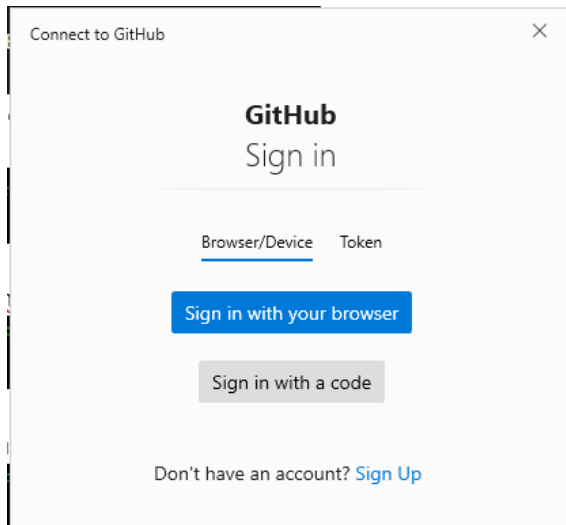
- Realiza un commit de tus cambios con el siguiente comando:

```
hdtol@HDMSIGF65THIN MINGW64 /d/Data/Escritorio
$ git commit -m "Agregado archivo index"
```

- Finalmente, sube tus cambios al repositorio en GitHub con el siguiente comando:

```
hdtol@HDMSIGF65THIN MINGW64 /d/Data/Escritorio
$ git push
```


En este punto nos solicitara iniciar sesión:



Validamos nuestra sesión, ya sea con el cliente de github o nuestro navegador o nuestra app de github, de esa forma podremos realizar nuestro primer push a nuestro repositorio.

```
hdtol@HDSIGF65THIN MINGW64 /d/Data/Escritorio/repoprueba (master)
$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 278 bytes | 278.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/hdtoledo/repoprueba.git
ea4ec15..314d24d master -> master
```

Clave SSH

Una clave SSH es un tipo de clave criptográfica que se utiliza para autenticarse y establecer una conexión segura entre un cliente (como tu computadora) y un servidor (como un repositorio de GitHub).

El protocolo SSH (Secure Shell) utiliza claves criptográficas para autenticar y cifrar las comunicaciones entre el cliente y el servidor. La clave SSH consta de dos partes: una clave pública y una clave privada. La clave pública se comparte con el servidor remoto, mientras que la clave privada se mantiene en el cliente.

Cuando el cliente intenta conectarse al servidor, el servidor solicita la clave pública del cliente. Si la clave pública coincide con la clave privada del cliente, el servidor autentica al cliente y se establece una conexión segura cifrada entre el cliente y el servidor.

Las claves SSH son una forma segura de autenticación porque las claves privadas no se comparten con el servidor remoto. Además, las claves SSH se pueden proteger con una contraseña adicional para aumentar la seguridad de la conexión.



Hagamos una clave SSH

Para crear una clave SSH de GitHub en tu PC, sigue los siguientes pasos:

1. Abre la terminal en tu PC.
2. Ejecuta el siguiente comando para verificar si ya tienes una clave SSH en tu PC:

```
ls -al ~/.ssh
```

3. Si el comando anterior muestra que no tienes una clave SSH, entonces ejecuta el siguiente comando para generar una nueva clave SSH:

```
ssh-keygen -t rsa -b 4096 -C "tu_correo_electronico"
```

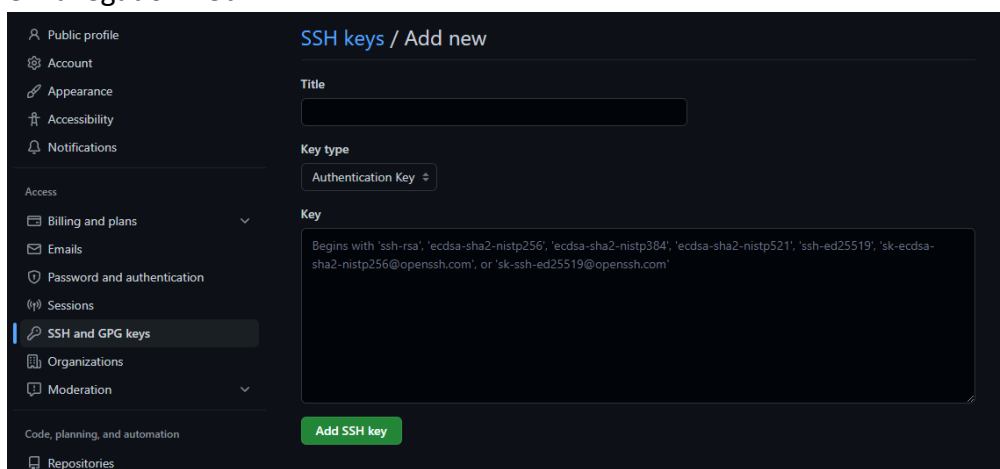
4. Cuando se te solicite, presiona Enter para aceptar la ubicación predeterminada del archivo de clave SSH.
5. A continuación, se te solicitará que ingreses una frase de contraseña para proteger la clave SSH. Puedes ingresar una contraseña o dejarla en blanco si no deseas proteger la clave SSH con una contraseña.
6. Una vez que se haya generado la clave SSH, ejecuta el siguiente comando para agregar la clave SSH a tu agente SSH:

```
eval "$(ssh-agent -s)"  
ssh-add ~/.ssh/id_rsa
```

7. Ahora, copia la clave SSH pública para agregarla a tu cuenta de GitHub. Para hacer esto, ejecuta el siguiente comando:

```
cat ~/.ssh/id_rsa.pub
```

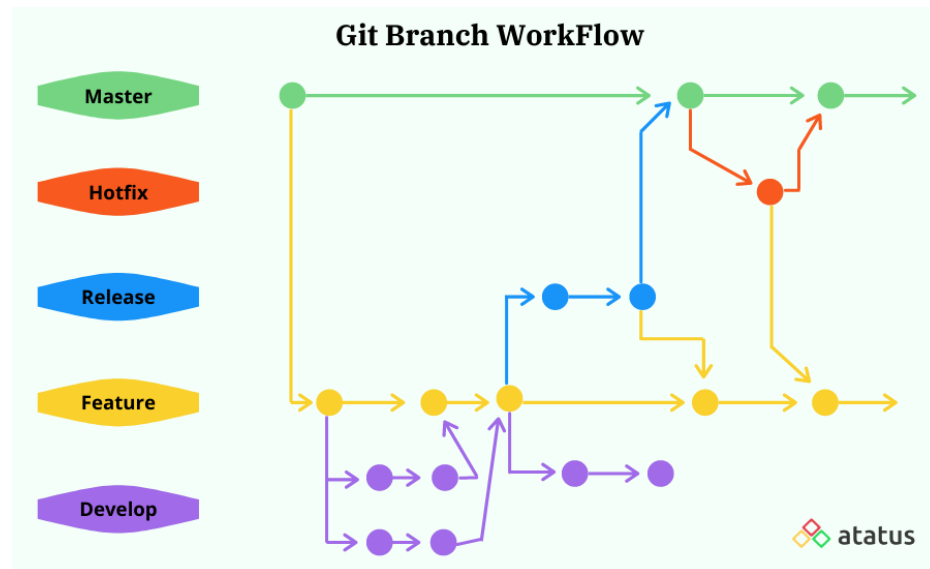
8. Copia la salida del comando anterior y pégala en la sección "Claves SSH" de tu cuenta de GitHub en el navegador web.



Con esto, has creado una clave SSH de GitHub en tu PC y la has asociado con tu cuenta de GitHub. Ahora puedes utilizar esta clave SSH para autenticarte con GitHub sin tener que ingresar tus credenciales de inicio de sesión cada vez que accedas a un repositorio.

¿Ramas/branch?

Las ramas en Git son referencias a una línea de desarrollo independiente y aislada del proyecto principal. En otras palabras, una rama es una versión alternativa de tu proyecto en la que puedes hacer cambios sin afectar la versión principal del proyecto.



Git por defecto crea una rama principal llamada "master" o "main" que contiene la versión actual del proyecto. Puedes crear nuevas ramas a partir de esta rama principal o de otras ramas ya existentes para trabajar en diferentes características o problemas de tu proyecto de manera aislada.

Cada rama tiene su propio historial de commits, lo que significa que puedes realizar cambios y hacer commits en una rama sin afectar el historial de commits de otras ramas. Además, puedes fusionar ramas para combinar los cambios de una rama en otra.

Las ramas son una característica poderosa y útil de Git porque permiten a los desarrolladores trabajar en diferentes características o soluciones de forma aislada y organizada, lo que facilita el mantenimiento y la colaboración en proyectos de software.

Ejercicio para las Branch

Para la revisión del ejercicio agregaras a tu equipo de proyecto junto con el instructor al repositorio, allí cada uno clonara el repo y subirá un cambio para luego fusionar las ramas en el master.

- Crea un nuevo repositorio en GitHub llamado "**ramas-prueba**".
- Clona el repositorio en tu equipo con el comando **git clone** <URL del repositorio>.
- Crea una nueva rama en tu repositorio local con el comando **git checkout -b nueva-rama**.
- Crea un archivo llamado "**archivo-rama.txt**" en tu repositorio local y agrega algún contenido.
- Agrega el archivo al área de preparación con el comando **git add archivo-rama.txt**.
- Realiza un **commit** con el comando **git commit -m "Agregado archivo-rama.txt a la rama nueva-rama"**.

- Empuja la rama **nueva-rama** y sus cambios al repositorio en GitHub con el comando **git push -u origin nueva-rama**.
- En la página de GitHub del repositorio, cambia a la rama "**nueva-rama**" en el menú desplegable de ramas.
- Crea un archivo llamado "**nuevo-archivo-rama.txt**" en GitHub y agrega algún contenido.
- Haz un **commit** de los cambios en la rama "**nueva-rama**" en GitHub.
- Vuelve a tu repositorio local y cambia a la rama principal con el comando **git checkout main**.
- Fusiona los cambios de la rama "**nueva-rama**" en la rama principal con el comando **git merge nueva-rama**.
- Resuelve cualquier conflicto que surja durante la fusión.
- Haz un **commit** de los cambios fusionados en la rama principal con el comando **git commit -m "Fusión de cambios de la rama nueva-rama en la rama principal"**.
- Empuja los cambios a la rama principal en el repositorio en GitHub con el comando **git push origin main**.

¡Y eso es todo! Con este ejercicio puedes practicar la creación de ramas, la edición de archivos en diferentes ramas, la fusión de ramas y la resolución de conflictos.

Algunos comandos básicos:

- **git init**: Inicializa un nuevo repositorio Git en el directorio actual.
- **git clone** <URL del repositorio>: Clona un repositorio existente en tu equipo.
- **git add** <nombre del archivo>: Agrega el archivo especificado al área de preparación para ser confirmado.
- **git commit -m** "Mensaje de confirmación": Crea un nuevo commit con los cambios agregados al área de preparación y un mensaje de confirmación que describe los cambios realizados.
- **git push**: Envía los cambios confirmados al repositorio remoto en GitHub u otro servicio de alojamiento de Git.
- **git pull**: Obtiene los cambios más recientes del repositorio remoto y los fusiona con tu rama actual.
- **git status**: Muestra el estado actual del repositorio, incluyendo los archivos que han sido modificados y los archivos que se han agregado o eliminado.
- **git log**: Muestra el historial de confirmaciones del repositorio, incluyendo los mensajes de confirmación y los autores de los cambios.
- **git branch**: Muestra una lista de todas las ramas en el repositorio y muestra la rama actual.
- **git checkout** <nombre de la rama>: Cambia a la rama especificada.
- **git merge** <nombre de la rama>: Fusiona la rama especificada en la rama actual.
- **git stash**: Guarda temporalmente los cambios que no están listos para confirmar, lo que te permite cambiar de rama o hacer otras tareas sin perder tu trabajo actual.
- **git remote add** <nombre> <URL del repositorio>: Agrega un nuevo repositorio remoto con el nombre especificado.

Estos son solo algunos de los comandos más comunes en Git. Hay muchos otros comandos que puedes usar para trabajar con tu repositorio, pero estos deberían darte una buena base para comenzar.



Git Cheat Sheet

Setup

Set the name and email that will be attached to your commits and tags

```
$ git config --global user.name "Danny Adams"
$ git config --global user.email "my-email@gmail.com"
```

Start a Project

Create a local repo (omit <directory> to initialise the current directory as a git repo)

```
$ git init <directory>
```

Download a remote repo

```
$ git clone <url>
```

Make a Change

Add a file to staging

```
$ git add <file>
```

Stage all files

```
$ git add .
```

Commit all staged files to git

```
$ git commit -m "commit message"
```

Add all changes made to tracked files & commit

```
$ git commit -am "commit message"
```

Basic Concepts

main: default development branch
origin: default upstream repo
HEAD: current branch
HEAD^: parent of HEAD
HEAD~4: great-great grandparent of HEAD

By @DoableDanny

Branches

List all local branches. Add -r flag to show all remote branches. -a flag for all branches.

```
$ git branch
```

Create a new branch

```
$ git branch <new-branch>
```

Switch to a branch & update the working directory

```
$ git checkout <branch>
```

Create a new branch and switch to it

```
$ git checkout -b <new-branch>
```

Delete a merged branch

```
$ git branch -d <branch>
```

Delete a branch, whether merged or not

```
$ git branch -D <branch>
```

Add a tag to current commit (often used for new version releases)

```
$ git tag <tag-name>
```

Merging

Merge branch a into branch b. Add --no-ff option for no-fast-forward merge



New Merge Commit (no-ff)



```
$ git checkout b
$ git merge a
```

Merge & squash all commits into one new commit

```
$ git merge --squash a
```

Rebasing

Rebase feature branch onto main (to incorporate new changes made to main). Prevents unnecessary merge commits into feature, keeping history clean



```
$ git checkout feature
$ git rebase main
```

Iteratively clean up a branches commits before rebasing onto main

```
$ git rebase -i main
```

Iteratively rebase the last 3 commits on current branch

```
$ git rebase -i Head~3
```

Undoing Things

Move (&/or rename) a file & stage move

```
$ git mv <existing_path> <new_path>
```

Remove a file from working directory & staging area, then stage the removal

```
$ git rm <file>
```

Remove from staging area only

```
$ git rm --cached <file>
```

View a previous commit (READ only)

```
$ git checkout <commit_ID>
```

Create a new commit, reverting the changes from a specified commit

```
$ git revert <commit_ID>
```

Go back to a previous commit & delete all commits ahead of it (revert is safer). Add --hard flag to also delete workspace changes (BE VERY CAREFUL)

```
$ git reset <commit_ID>
```

Review your Repo

List new or modified files not yet committed

```
$ git status
```

List commit history, with respective IDs

```
$ git log --oneline
```

Show changes to unstaged files. For changes to staged files, add --cached option

```
$ git diff
```

Show changes between two commits

```
$ git diff commit1_ID
commit2_ID
```

Stashing

Store modified & staged changes. To include untracked files, add -u flag. For untracked & ignored files, add -a flag.

```
$ git stash
```

As above, but add a comment.

```
$ git stash save "comment"
```

Partial stash. Stash just a single file, a collection of files, or individual changes from within files

```
$ git stash -p
```

List all stashes

```
$ git stash list
```

Re-apply the stash without deleting it

```
$ git stash apply
```

Re-apply the stash at index 2, then delete it from the stash list. Omit stash@{n} to pop the most recent stash.

```
$ git stash pop stash@{2}
```

Show the diff summary of stash 1. Pass the -p flag to see the full diff.

```
$ git stash show stash@{1}
```

Delete stash at index 1. Omit stash@{n} to delete last stash made

```
$ git stash drop stash@{1}
```

Delete all stashes

```
$ git stash clear
```

Synchronizing

Add a remote repo

```
$ git remote add <alias>
<url>
```

View all remote connections. Add -v flag to view urls.

```
$ git remote
```

Remove a connection

```
$ git remote remove <alias>
```

Rename a connection

```
$ git remote rename <old>
<new>
```

Fetch all branches from remote repo (no merge)

```
$ git fetch <alias>
```

Fetch a specific branch

```
$ git fetch <alias> <branch>
```

Fetch the remote repo's copy of the current branch, then merge

```
$ git pull
```

Move (rebase) your local changes onto the top of new changes made to the remote repo (for clean, linear history)

```
$ git pull --rebase <alias>
```

Upload local content to remote repo

```
$ git push <alias>
```

Upload to a branch (can then pull request)

```
$ git push <alias> <branch>
```