Git y GitHub

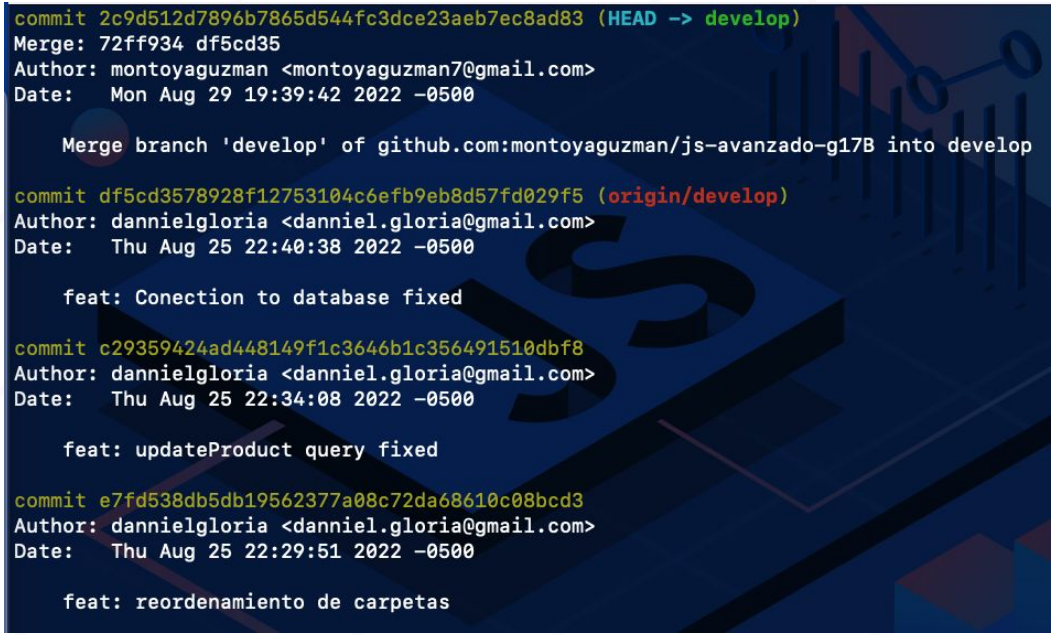
DEV.F::
DESARROLLAMOS(PERSONAS);

Git y github no son lo mismo, cada que piensas eso muere un gatito.

Git
Sistema de Control de Versiones

Sistema de control de versiones

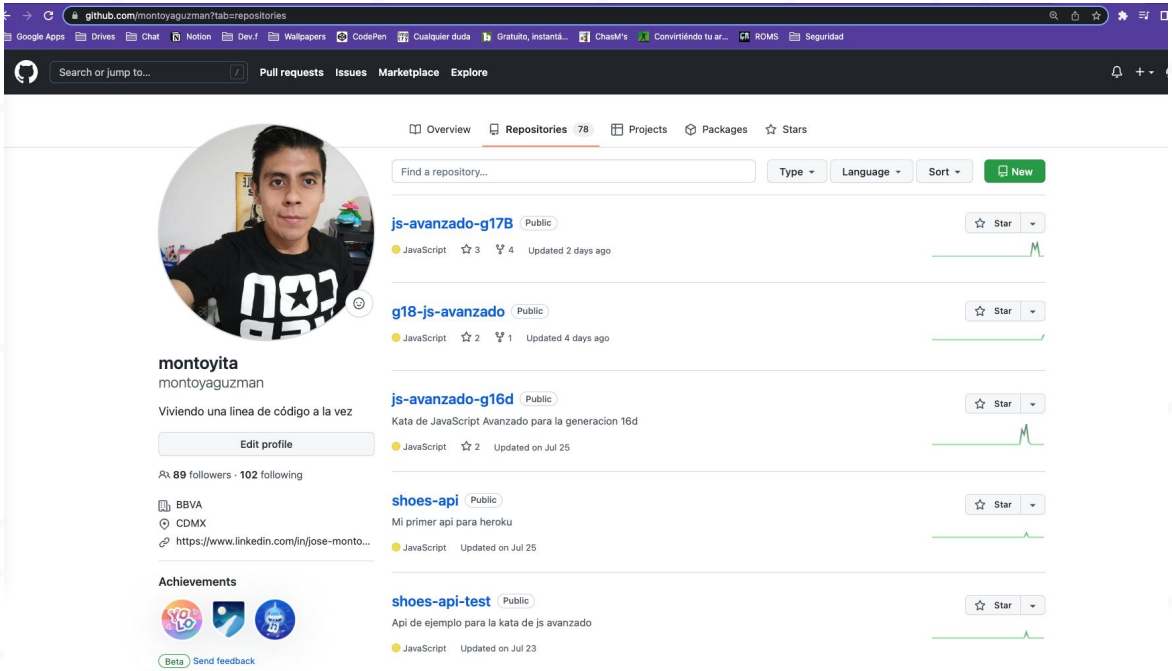
Git nos va a permitir **controlar nuestro historial de cambios de código** y **saber que, cuando y quien modifico algo.** En sistemas de software profesionales se tienen muchos archivos y distintos cambios que controlar, para ello nos ayudan los controladores de versiones.



Github
Repositorio remoto

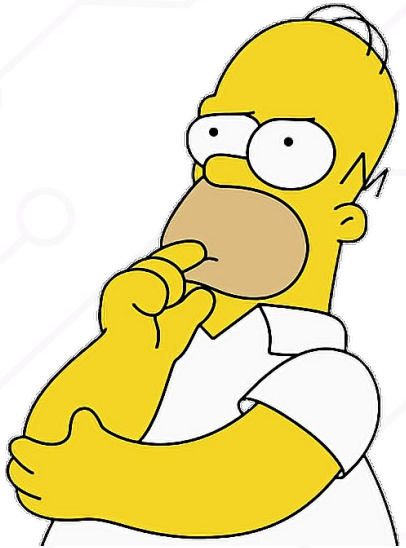
Repositorio remoto

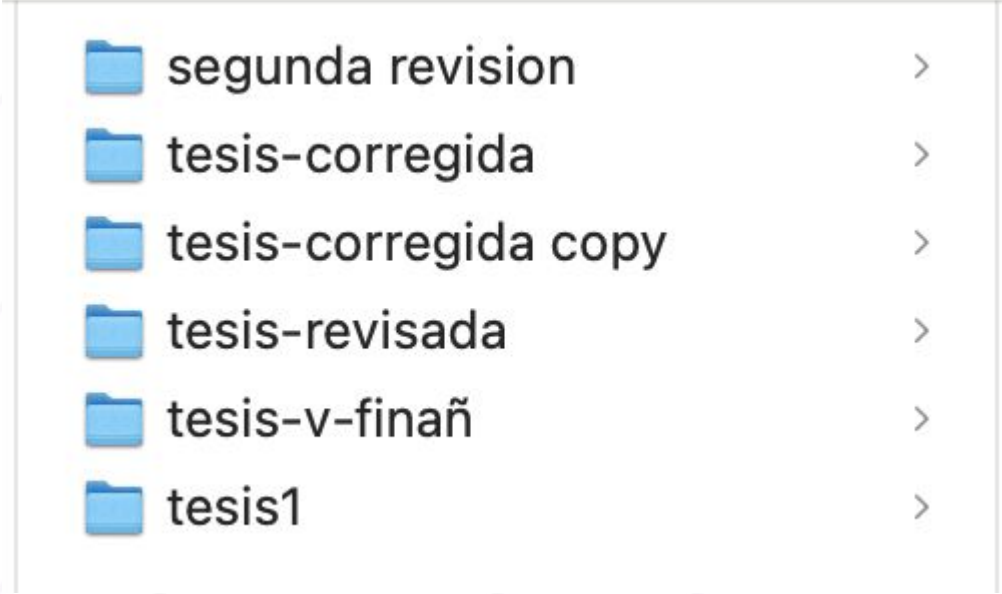
Github funge como **la plataforma en internet a donde subimos el código para compartirlo con otros** desarrolladores**.**



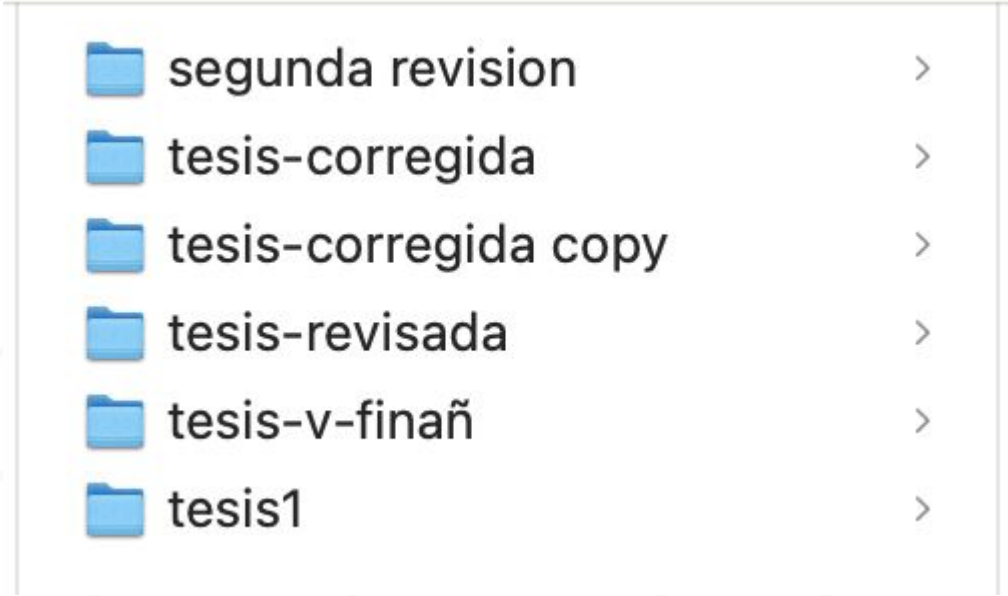
¿Por qué necesito un controlador de versiones (GIT)?

¿Qué significa controlar una versión?





¿Qué significa controlar una versión?



Sistema de control de versiones (VCS)

Un **sistema de control de versiones (VCS - Version control system)** es aquel que nos permite llevar un historial y control de cambios a medida que una o más personas colaboran en un proyecto.

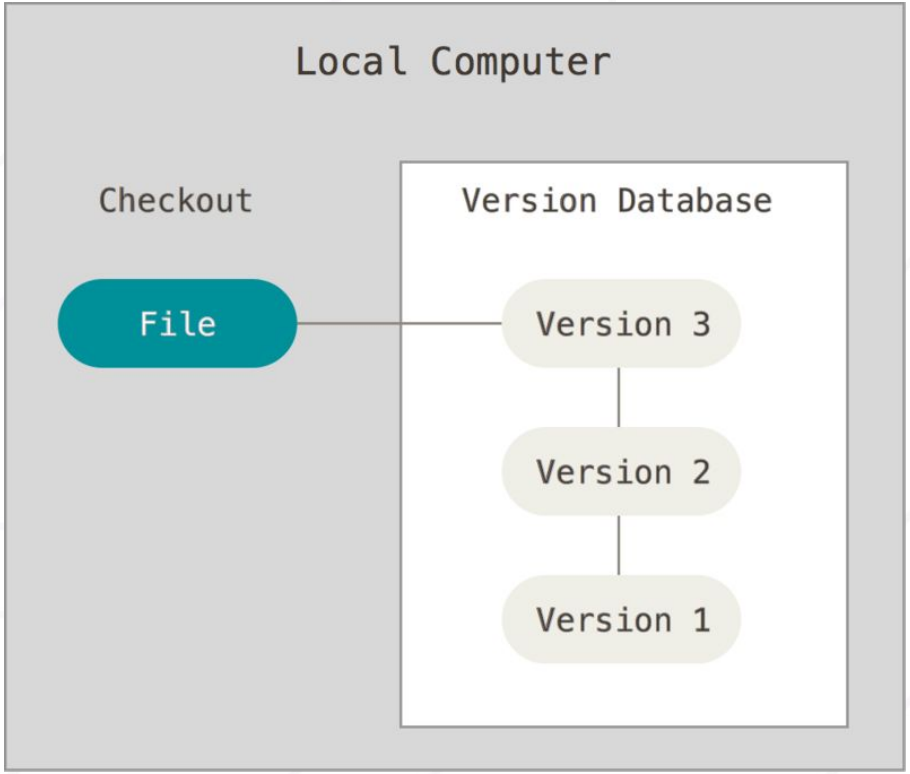
- ¿Que cambios se hicieron?
- ¿Quién hizo los cambios?
- ¿Cuando se hicieron los cambios?
- ¿Por qué fueron requeridos los cambios?

Tipos de controladores de versiones

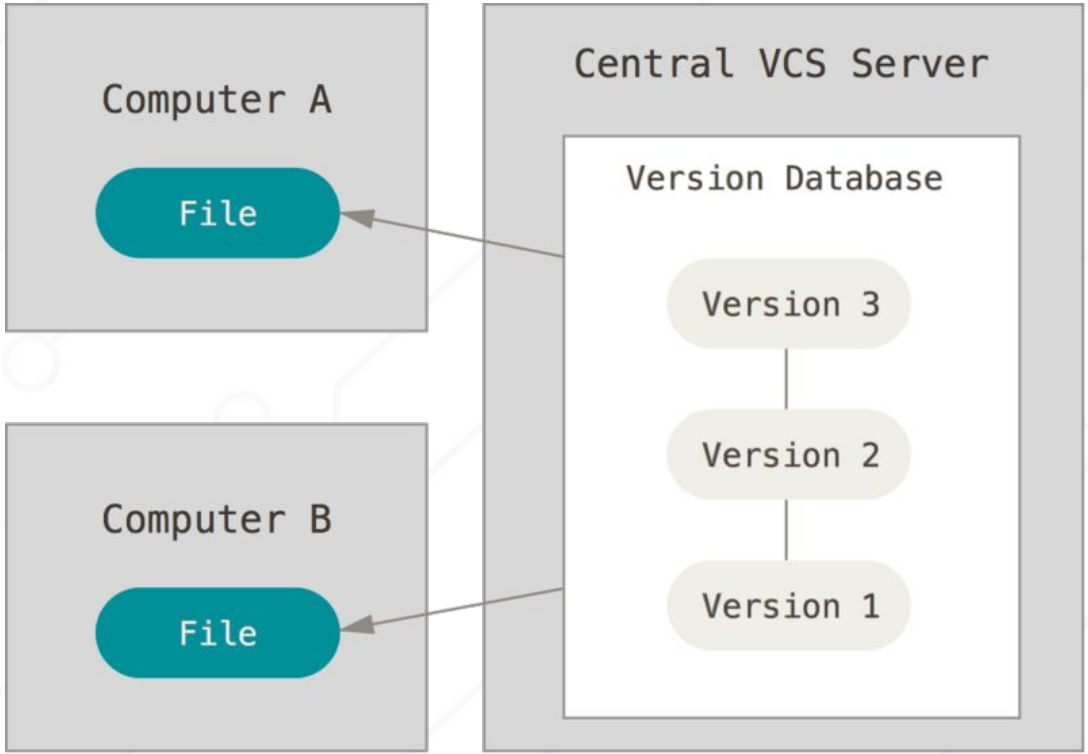
Repositorio remoto

- **Locales:** Todos los datos del proyecto se almacenan en una sola computadora y los cambios realizados en los archivos del proyecto se almacenan como revisiones.
- **Centralizados:** Utiliza un servidor central para almacenar todos los archivos y permite el trabajo colaborativo de un equipo. Trabaja sobre un repositorio único al que los usuarios pueden acceder desde un servidor central.
- **Distribuidos:** Aparecen para superar el inconveniente del sistema de control de versiones centralizado. Los clientes clonan completamente el repositorio, incluido su historial completo. Si algún servidor está inactivo o desaparece, cualquiera de los repositorios del cliente se puede copiar en el servidor para restaurarlo..

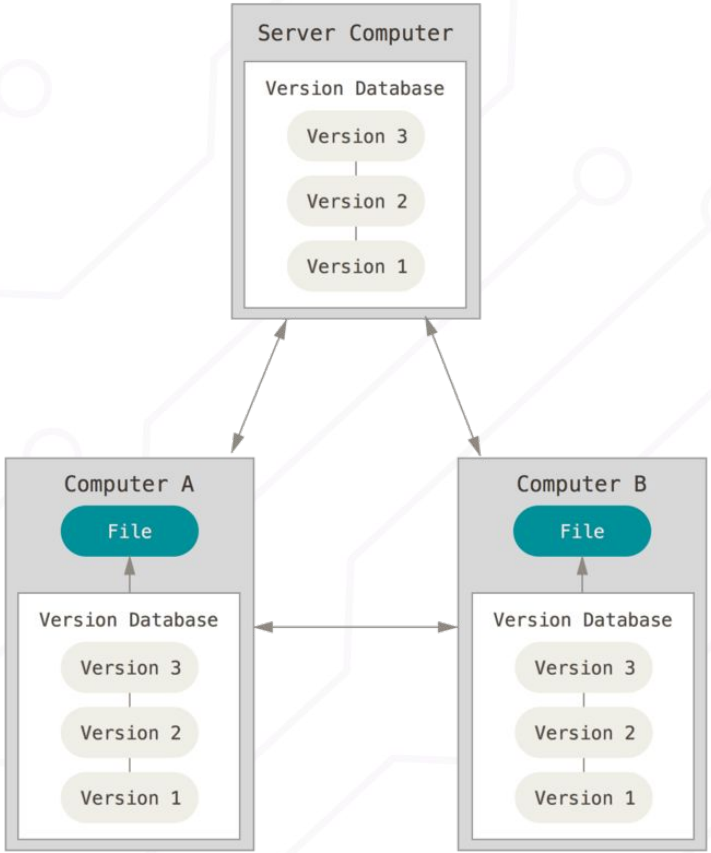
Tipos de VCS (Local)



Tipos de VCS (Centralizado)



Tipos de VCS (Distribuido)



GIT

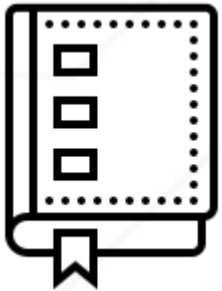
Git

Git es un (VCS) de tipo **distribuido** de código abierto y actualmente el más usado por los desarrolladores gracias a sus beneficios para individuos y equipos de trabajo.

- Acceso detallado a la historia del proyecto.
- Colaboración en cualquier momento y lugar.

Su uso principal es mediante Interfaz de línea de comandos **(CLI - Command line interface)**

¿Qué es un repositorio?



Un **repositorio** es un espacio de almacenamiento donde se organiza, mantiene y difunde información.

El **repositorio** es la carpeta del proyecto donde estará la colección de archivos y carpetas junto al historial de cambios.

Instalación y manejo de Git

DEV.F:
DESARROLLAMOS(PERSONAS);

Instalación

- Windows.
- Mac OS.
- Linux.

Configuración de GIT

DEV.F:
DESARROLLAMOS(PERSONAS);

Configuración inicial de Git

• Desde consola, se puede ver a la configuración de **Git** con el comando:

git config --list

• Se recomienda establecer una identidad en **Git**, para ello se usan los comandos:

git config user.name
git config user.email

• Usando el flag “--global” podemos establecer la configuración de forma global y realizarla una sola vez.

Comandos parte 1 (Configuración)

Posterior a la instalación es necesario indicar a git el usuario y correo que se utilizara para registrar cada uno de los commits realizados en el equipo.

git config --global user.name -> Poner un nombre de usuario global a la configuración de nuestro git.

git config --global user.email -> Poner un nombre de usuario global a la configuración de nuestro git.

NOTA: Está configuración se realiza solo una vez por equipo.

Trabajando con el repositorio local

DEV.F::
DESARROLLAMOS(PERSONAS);

Repositorio local

El repositorio local es una carpeta de nuestro proyecto en local que fue inicializada con el comando **git init** y **consta de 3 partes fundamentales:**

- **Working directory:** Nuestro disco duro o sistema de archivos.
- **Staging area:** Lo que está listo para agregarse al historial (Área de indexado de git).
- **Repo local:** Es lo que se encuentra en el historial del commit.

Estados de Git

working directory

git add

staging area

git commit

repository


I HAVE NO
IDEA WHAT
I'M DOING

Comandos parte 2 (Creación de un repo y status)

Inicialización de un repo
git init -> Inicializamos repositorio.

Seguimiento del repo
git status -> Nos muestra el estado de working y staging area.
git log --oneline -> *Ver los commits que hemos realizado*
git log --graph -> *Ver los commits como una línea de tiempo*

Comandos parte 3 (Haciendo commit)

Stage/unstage

git add . -> *Agregamos todos los archivos al staging area.*
git add archivo.txt -> *Agregamos el archivo.txt al staging area.*
git rm --cached archivo.txt -> *Quitamos el archivo.txt del staging area.*
git restore --staged archivo.txt -> *Quitamos el archivo.txt del staging area.*

Commits

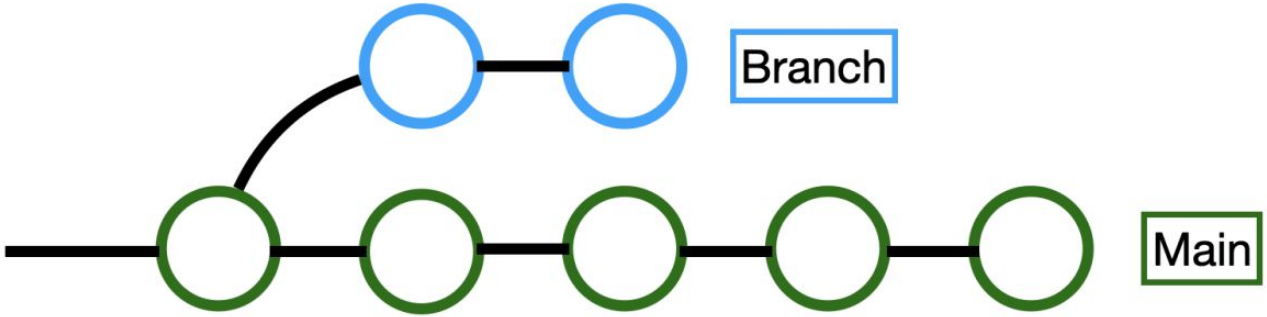
git commit -m “Comentario” -> Se crea un punto en la historia con un mensaje.
git commit -am “Comentario” -> *Agregamos el archivo.txt al staging area.*
git commit --amend -m “Comentario” -> *Actualiza el último mensaje del commit*

Administración de branches

DEV.F:
DESARROLLAMOS(PERSONAS);

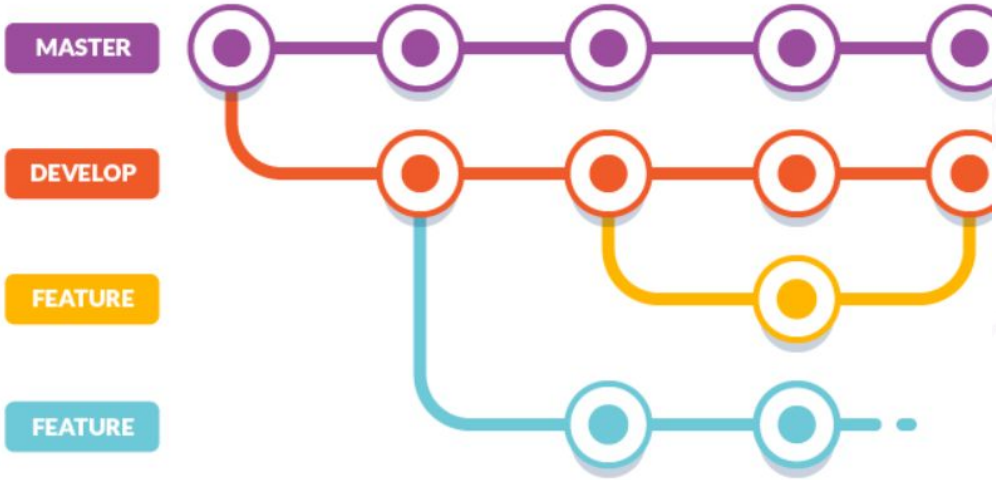
Ramas (Branch)

Una **rama** Git **es un apuntador móvil a un punto de confirmación en la historia de otra rama**. La rama por defecto de Git es la rama master. Con la primera confirmación de cambios que realicemos, se creará esta rama principal master apuntando a dicha confirmación. En cada confirmación de cambios que realicemos, la rama irá avanzando automáticamente.



Ramas (Branch)

Las **branch** son bifurcaciones o variantes de un repositorio, estas pueden contener diferentes archivos y carpetas o tener todo igual excepto por algunas líneas de código.



DEV.F

Comandos parte 4 (Administración de branchs)

git branch -> Mostrar las ramas que tenemos.

git branch newBranchName -> Creamos una nueva rama.

git checkout branchName -> Nos cambiamos a la rama nombre.

git checkout -b newBranchName -> Crear y cambiarse a una nueva rama.

git switch -c branchName -> Crear y cambiarse a una nueva rama.

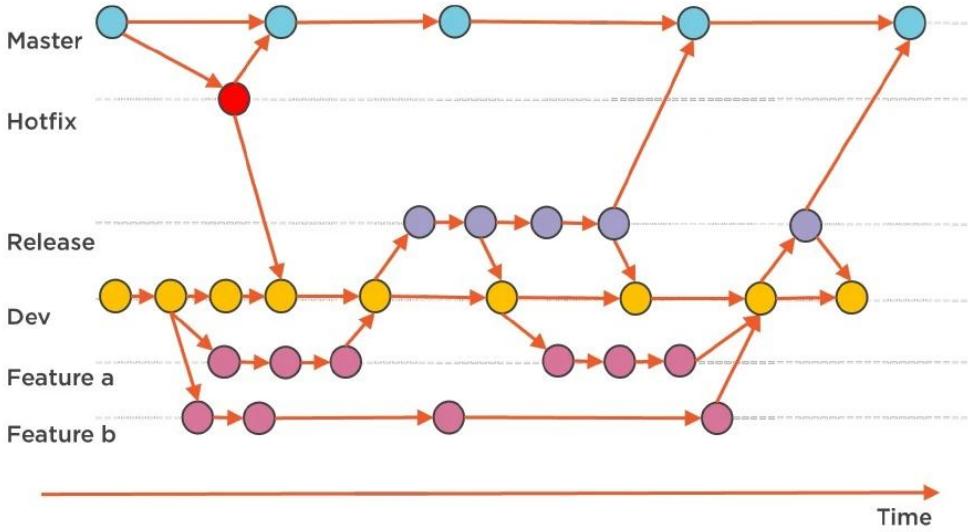
git checkout hash(id del commit) -> Cambiarnos a un commit en específico.

git checkout . -> Regresar al commit más reciente de la rama actual.

git merge sourceBranchName -> Unimos cambios de una rama.

¿Por qué usar ramas y commits?

Las ramas y los commits son las herramientas que nos van a permitir controlar el flujo de cambios e historial de los mismos, haciendo nuestra distribución del código mucho más controlada y administrable.



Regresar en el tiempo

Resolver conflictos

DEV.F::
DESARROLLAMOS(PERSONAS);

Tipos de merge

Auto Merging

Al hacer un merge la fusión de archivos la resuelve git de forma automática.

Conflict

Git no puede hacer el auto merging porque se modificaron las mismas líneas de código. Debe arreglarse el conflicto manualmente y hacer otro commit.

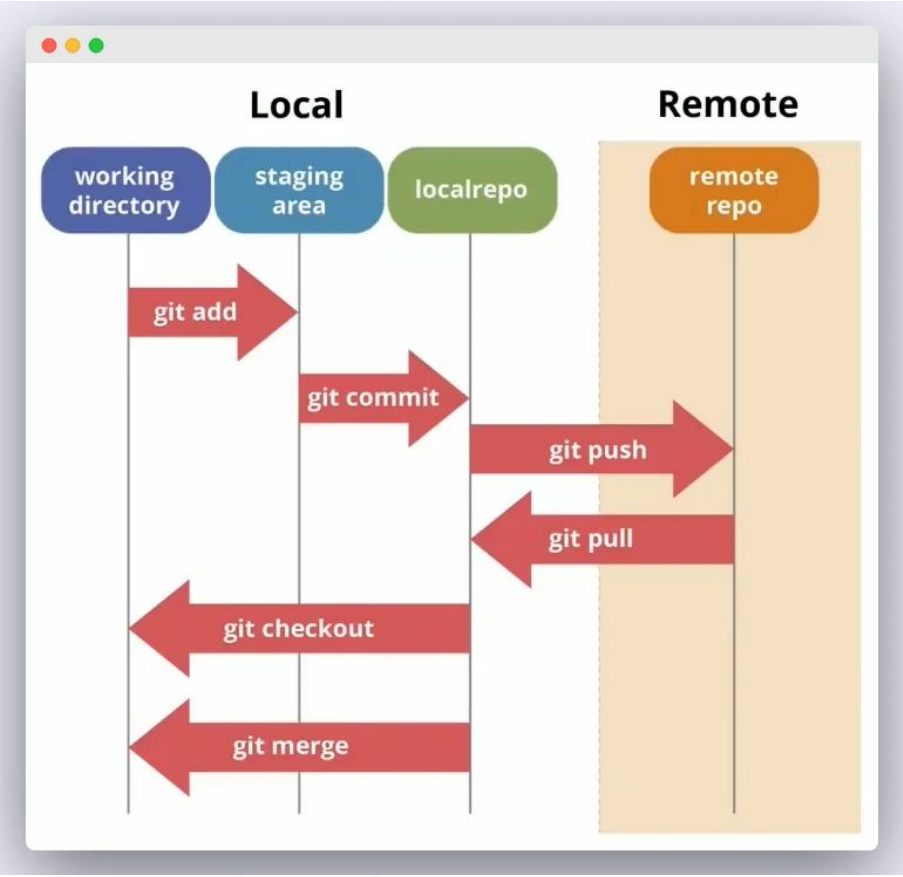
Trabajando con el repositorio remoto

DEV.F::
DESARROLLAMOS(PERSONAS);

Conectar un repositorio local a uno remoto

- El historial de cambios es gestionado por GIT.
- Los repos remotos son el respaldo de nuestro local y medio de distribución de código.

Flujo con repositorio remoto



Comandos parte 5 (Trabajando con el repo remoto)

git remote -v -> Ver si nuestro repo local esta conectado a algun repo remoto

git remote add origin *url* -> Agregar la conexión de nuestro repo local al remoto.

git remote set-url aliasName myNewUrl -> Agregar la conexión de nuestro repo local al remoto.

git clone *url* -> Clonar repositorio existente.

git push *alias branch* -> Enviamos cambios a repositorio remoto.

git push *--all origin* -> Subir todas las ramas desde local a remoto.

git pull *alias branch* -> Obtenemos cambios más recientes de la rama.

¿Cómo conectarse a un repo remoto?

DEV.F::
DESARROLLAMOS(PERSONAS);

Conectar un repo local a repo remoto

Existen 3 casos:

1. **Comenzar desde 0** creando el repo en github, clonarlo y comenzar a trabajar con él.
2. Tener código en local, es decir, **previamente escrito (opcional)**. Crear un repo remoto para subirlo.
3. Crear la carpeta y repo en local, crear un repo remoto y **relacionarlos**.

Desde 0

1. Crear un repositorio en remoto y clonarlo.
2. Comenzar a agregar archivos, commits y push.

NOTA: Utilizada cuando no tenemos nada de código y el repo es nuevo.

Con código previamente escrito (opcional)

1. Crear una nueva carpeta local.
2. Inicializarla como un repo de git y hacer commit.
3. Crear un repositorio remoto.
4. Agregar el origin del repo remoto al repo local (**git remote add origin *url***).
5. Realizar el push.

NOTA: Utilizada cuando tenemos código existente y queremos subirlo a un repo remoto.

Relacionar repos distintos

Working..

Fusión de ramas
Merges

DEV.F:
DESARROLLAMOS(PERSONAS);

Conectar un repositorio local a un remoto

Existen dos formas de hacer merge.

1. Merge local.
2. Pull request.

Flujo merge local

1. Cambiarse a la rama de destino.

   ```
   git checkout main
   ```

2. Ejecutar el comando merge en la rama destino

   ```
   git merge develop
   ```

Flujo merge por pull request

1. Hacer un commit.

```
git add .
git commit -m “Comentario”
```

2. Enviarlo al repositorio remoto.

```
git push alias branch
```

3. Crear la pull request en github (rama base y rama destino) y agregar revisores.
4. Los revisores aceptan la PR (Pull request) y se hace el merge.
5. Obtener los cambios mediante git pull origin develop.

Otros comandos útiles

DEV.F:: DESARROLLAMOS(PERSONAS);

Otros comandos útiles

git reset --hard HEAD^ -> Descartar cambios y eliminarlos del stagin y working directory

git reset --soft HEAD~1 -> Descartar cambios y eliminarlos del stagin y working directory

git rm --cached . r -> Limpia la cache de git

git stash -> stashear

git stash pop -> stashear

git stash list -> ver la pila stash

git log --all --decorate --oneline --graph -> Muestra todas las ramas con sus distintos commit de forma gráfica.

Buenas prácticas

DEV.F::
DESARROLLAMOS(PERSONAS);

Gitflow

Gitflow es un **modelo alternativo de creación de ramas** en Git en el que se utilizan ramas de función y varias ramas principales.
Según este modelo, los desarrolladores crean una rama de función y retrasan su fusión con la rama principal base hasta que la función está completa. Estas ramas de función de larga duración requieren más colaboración para la fusión y tienen mayor riesgo de desviarse de la rama troncal.

- **main/master**
- **develop**
- **release/appV1.0.0**
- **feature/login**
- **hotfix/logo-app**

Atomic commits

Un commit atómico es un commit que está enfocado en subir una sola cosa, puede ser un feature, resolver un bug, un refactor, una actualización, una tarea, etc.

Ejemplos

git commit -m “feat: agrega el menu”
git commit -m “fix: corrige la paleta de colores”
git commit -m “docs: cuando agregan documentacion para las paginas de pagos”

Conventional commits

Es una convención en el formato de los mensajes de los commits. Esta convención define una serie de reglas que hacen muy sencillo tanto la legibilidad del histórico del repositorio como el poder tener herramientas que automaticen procesos basándose en el historial de commits.

Tipos

- **feat:** Nuevas características.
- **chore:** Cosas que no aportan un req funcional pero posiblemente si un req no funcional.
- **fix:** Corrección de errores.
- **docs:** Commits con documentación o comentarios.
- **style:** Cambios de legibilidad o formateo de código que no afecta a funcionalidad.
- **refactor:** cambio de código o arquitectura que no corrige errores ni añade funcionalidad, pero mejora el código.
- **test:** Para añadir o arreglar tests.

Ejercicios

DEV.F:
DESARROLLAMOS(PERSONAS);

Práctica 1

Práctica 2