Guía Git

Sitio: Instituto Superior Politécnico Córdoba
Curso: Programador Full Stack - TSDWAD - 2022
Libro: Guía Git

Imprimido por: Daniel Roberto LUJAN
Día: lunes, 28 agosto 2023, 11:10 PM

Tabla de contenidos

1. Introducción

Esta guía tiene por objetivo:

- Introducir el concepto de sistema de control de versiones, su funcionamiento general y su relevancia en el proceso de desarrollo de software.
- Comprender el proceso de instalación y de uso de la herramienta Git para el control de versiones de un proyecto de software.
- Crear y gestionar un proyecto de software a través de un repositorio en Github, generando un flujo de trabajo con ramificaciones.

2. Sistema de Control de Versiones

En el proceso de desarrollo de software es un requisito casi indispensable mantener un registro de los cambios que se realizan sobre el código fuente a lo largo del tiempo. Es debido a esto que cobran importancia los sistemas de control de versiones. Estos sistemas son herramientas que permiten realizar un seguimiento de los cambios y también permiten proteger el código de errores humanos accidentales. Además, un sistema de control de versiones facilita el trabajo en equipo a la hora de desarrollar software, ya que mientras un integrante trabaja en alguna funcionalidad específica, otro podría estar trabajando en alguna corrección de errores o bien en otra funcionalidad, para luego integrar las soluciones y realizar una sincronización del trabajo de cada uno.

El uso de un sistema de control de versiones tiene tres ventajas principales:

1. Gracias al historial de cambios se puede saber el autor, la fecha y notas escritas sobre los cambios realizados. También permite volver a versiones anteriores para ayudar a analizar causas raíces de errores y es crucial cuando hay que solucionar problemas de versiones anteriores.
2. Creación y fusión de ramas. Al tener varios integrantes del equipo trabajando al mismo tiempo, cada uno en una tarea diferente, pueden beneficiarse de tener flujos de trabajo independientes. Posteriormente se pueden fusionar estos flujos de trabajos o ramas a una principal. Los sistemas de control de versiones tienen mecanismos para identificar que los cambios entre ramas no entren en conflicto para asegurar la funcionalidad y la integración.
3. Trazabilidad de los cambios que se hacen en el software. Poder conectar el sistema de control de versiones con un software de gestión de proyectos y seguimiento de errores ayuda con el análisis de la causa raíz de los problemas y con la recopilación de información.

El concepto de versión (también llamado revisión o edición) de un proyecto (código fuente) hace referencia al estado en el que se encuentra el mismo en un momento dado de su desarrollo o modificación. Los sistemas de control de versiones utilizan repositorios para almacenar el proyecto actualizado junto a sus cambios históricos. Los sistemas de control de versiones centralizados almacenan todo el código en un único repositorio, es decir que un único servidor contiene todos los archivos versionados. Esto representa un único punto de falla dado que si el servidor no está disponible por un tiempo nadie podrá colaborar o guardar cambios en archivos en los que hayan estado trabajando.

Los sistemas de control de versiones distribuidos permiten en cambio continuar el trabajo aún cuando el repositorio de referencia no está disponible. En estos sistemas los clientes no solo descargan la última copia del código, sino que se replica completamente el repositorio con los cambios históricos (versiones). De esta manera, si un servidor deja de funcionar y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios disponibles en los clientes puede ser copiado al servidor con el fin de restaurarlo.

Actividad: Podemos clasificar los sistemas de control de versiones según la arquitectura utilizada para el almacenamiento del código fuente en: distribuida o centralizada. *Investiga ¿en qué se basan estas dos arquitecturas de almacenamiento? ¿Cuál será la más recomendable?*

3. Git

Git es un proyecto de código abierto maduro y con un activo mantenimiento desarrollado originalmente por Linus Torvalds. Este sistema de control de versiones distribuido que funciona bajo cualquier plataforma (Windows, MacOS, Linux, etc) y está integrado en una amplia variedad de entornos de desarrollo (IDEs). Este sistema presenta una arquitectura distribuida, es decir que, cada desarrollador posee una copia del trabajo en un repositorio local donde puede albergar el historial completo de todos los cambios y, mediante determinados comandos, realiza sincronizaciones al repositorio remoto.

Git fue diseñado teniendo en cuenta las siguientes características:

- **Rendimiento**: Los algoritmos implementados en Git aprovechan el profundo conocimiento sobre los atributos comunes de los auténticos árboles de archivos de código fuente, cómo suelen modificarse con el paso del tiempo y cuáles son los patrones de acceso. El formato de objeto de los archivos del repositorio de Git emplea una combinación de codificación delta (que almacena las diferencias de contenido) y compresión, y guarda explícitamente el contenido de los directorios y los objetos de metadatos de las versiones.
- **Seguridad**: la principal prioridad es conservar la integridad del código fuente gestionado. El contenido de los archivos y las verdaderas relaciones entre estos y los directorios, las versiones, las etiquetas y las confirmaciones, están protegidos con un algoritmo de hash criptográficamente seguro llamado "SHA1". De este modo, se salvaguarda el código y el historial de cambios frente a las modificaciones accidentales y maliciosas, y se garantiza que el historial sea totalmente trazable.
- **Flexibilidad**: es flexible en varios aspectos, en la capacidad para varios tipos de flujos de trabajo de desarrollo no lineal, en su eficiencia en proyectos tanto grandes como pequeños y en su compatibilidad con numerosos sistemas y protocolos. Se ha ideado para posibilitar la ramificación y el etiquetado como procesos de primera importancia y las operaciones que afectan a las ramas y las etiquetas (como la fusión o la reversión) también se almacenan en el historial de cambios.

Para comprenderlo mejor, te invitamos a mirar el siguiente video explicativo:

3.1. Instalación de Git

Instructivo de instalación **aquí!!**

3.2. Repositorios en Git

Un repositorio es un directorio dónde tienes alojado tu proyecto permitiéndote además, guardar versiones del código a las que puedes acceder cuando lo necesites.

¿Cómo crear un repositorio en Git?

Para crear un nuevo repositorio deberemos utilizar los comandos **git init** y **git clone**.

- **git init:** es un comando que se utiliza una sola vez durante la configuración inicial de un nuevo repositorio. Al ejecutar este comando se creará un subdirectorio oculto .git en el directorio de trabajo actual (aquí estará guardada toda la historia) y una nueva rama principal (ej. main o master).
- **git clone:** es un comando que se utiliza para clonar un repositorio remoto.

 Para comprender mejor estos conceptos y ver una demostración de cómo utilizar estos comandos y con qué herramientas, te invitamos a mirar el siguiente video:

Para más información sobre estos comandos click en **git init, git clone.**

3.3. ¿Cómo guardar los cambios en Git?

Cuando trabajamos con Git o con otro sistemas de control de versiones, el concepto de "guardar" difiere un poco al concepto convencional que tenemos actualmente de guardar.

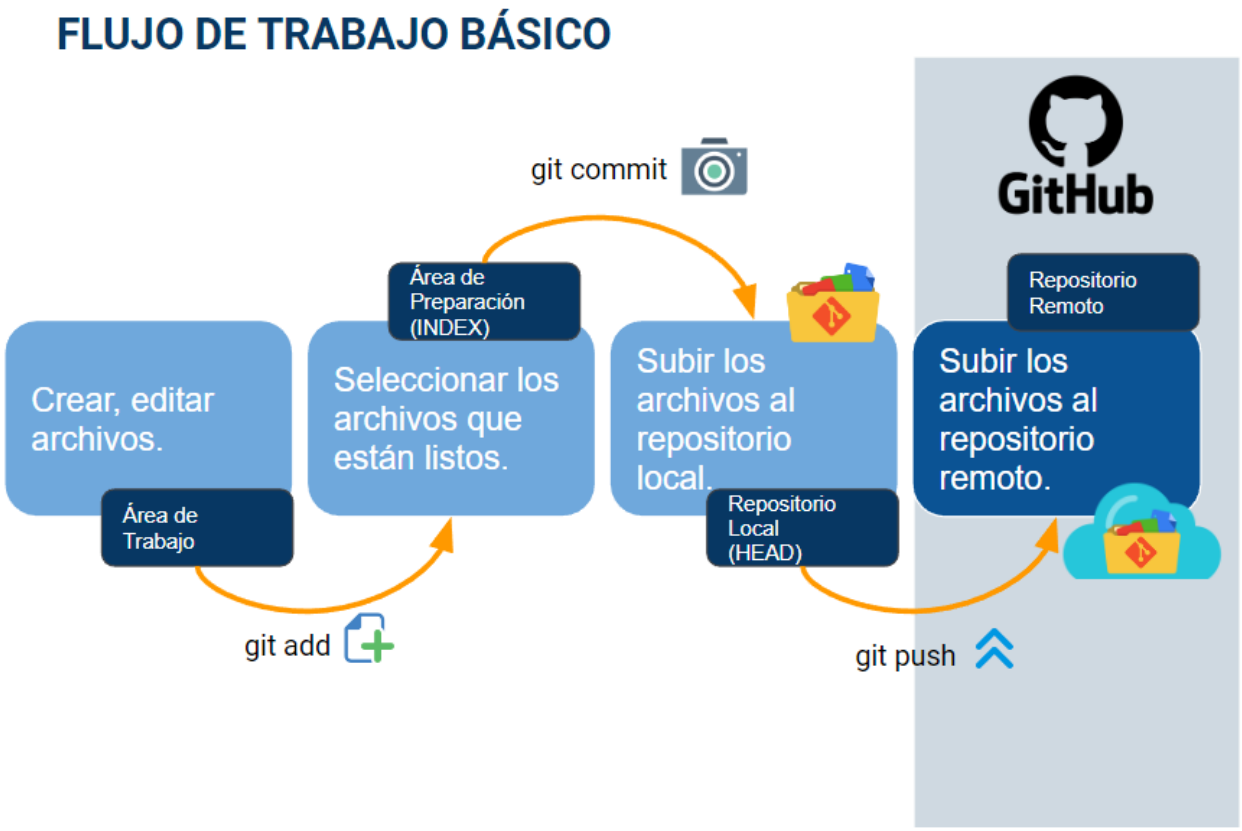
En git la expresión **"guardar"** equivale a **"confirmar"**.

Para comprenderlo, te invitamos a mirar el siguiente video explicativo:

Por ende, para guardar los cambios en Git, se sigue un proceso distinto. Es fundamental entender la áreas y estados por los que pueden pasar los archivos durante el flujo de trabajo.

En un proyecto de Git hay 4 secciones fundamentales:

- **Área de trabajo (*Working Directory*):** es una copia de una versión del proyecto, Son archivos sacados de la base de datos comprimida y se colocan en el disco para poder ser usados o modificados.
- **Área de preparación (*Staging Area - INDEX*):** es un archivo que se encuentra dentro del directorio de Git y que contiene información acerca de lo que va a ir en la próxima confirmación.
- **Repositorio Local(*Local Repository - HEAD*):** es el lugar donde se almacenan los metadatos y la base de datos de objetos del proyecto. Es lo que se copia cuando se clona un repositorio desde otra fuente.
- **Repositorio Remoto (*Remote Repository*):** es el repositorio que se encuentra en un servidor remoto y con el que eventualmente se sincronizan los trabajos entre los diferentes integrantes del equipo.



Comandos básicos para guardar cambios en Git

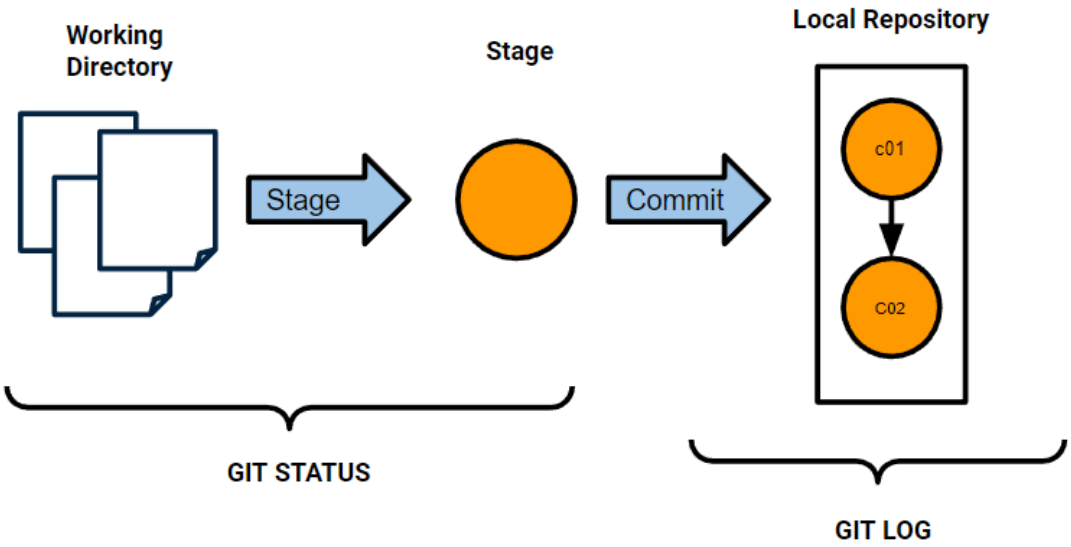
Git se puede ver como un set de herramientas muy completo, pero para un manejo básico de repositorios en Git es necesario conocer, por lo menos, los siguientes comandos:

- **git add <file>** luego de la creación, modificación o eliminación de un archivo, los cambios quedan únicamente en el área de trabajo, por lo tanto es necesario pasarlos al área de preparación mediante el uso del comando **git add**, para que sea incluido dentro de la siguiente confirmación (*commit*).
- **git commit**, con este comando se confirman todos los cambios registrados en el área de preparación, o lo que es lo mismo, se pasan los cambios al repositorio local.
- **git push** es el comando que se utiliza para enviar todas las confirmaciones registradas en el repositorio local a un repositorio remoto.

3.4. ¿Cómo inspeccionar los cambios en un repositorio de Git?

A menudo necesitamos inspeccionar los cambios en nuestros repositorios. Para ello, Git nos provee los siguientes comandos:

- **git status:** comando que muestra el estado del área de trabajo y del área de preparación permitiendo ver los cambios que se han preparado, los que no y los que Git no lleva el seguimiento. Es importante agregar en este punto que, este comando no muestra ninguna información relativa a la historia del proyecto.
- **git log:** comando que muestra las instantáneas confirmadas. Permite ver el historial del proyecto, filtrarlo y buscar cambios concretos.



Para comprenderlo mejor, te invitamos a mirar el siguiente video explicativo:

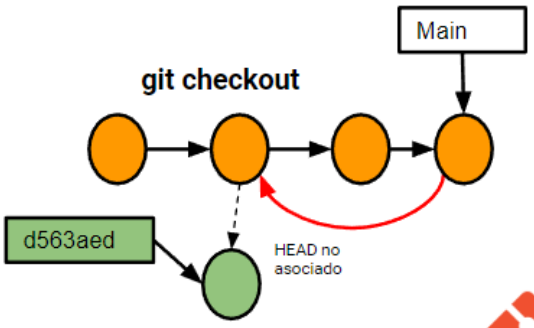
Actividad: A continuación, te invitamos a encontrar otras formas de filtrar confirmaciones. Ej. filtrar commits de todas o una rama en particular, por rango de fechas, etc.
Comparte en el foro!!

3.5. ¿Cómo deshacer los cambios en Git?

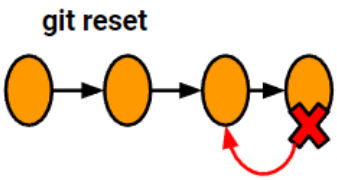
A menudo necesitamos deshacer acciones en confirmaciones de nuestro proyecto. Al igual que para el procedimiento de guardar, Git no cuenta con el sistema tradicional para deshacer acciones tal como lo conocemos. Es decir, no existe el **ctrl + z.**

Para ello, Git nos provee los siguientes comandos:

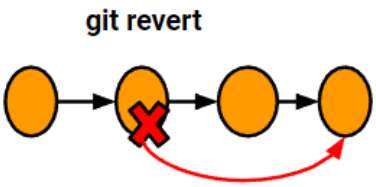
- **git reset:** es la opción más adecuada para deshacer cambios privados locales.
- **git revert:** es la mejor opción para deshacer cambios públicos compartidos.
- **git checkout:** se utiliza para desplazarte y revisar el historial de confirmaciones.



- git checkout <commit>
- git checkout -b <new branch name>



- git reset <commit>



- git revert <commit>
- git revert <commit> --no-edit
- git revert <commit> --no-commit

A continuación, te invitamos a ver un video explicativo:

Para más información, consulta: **git checkout**, **git reset** y **git revert**.

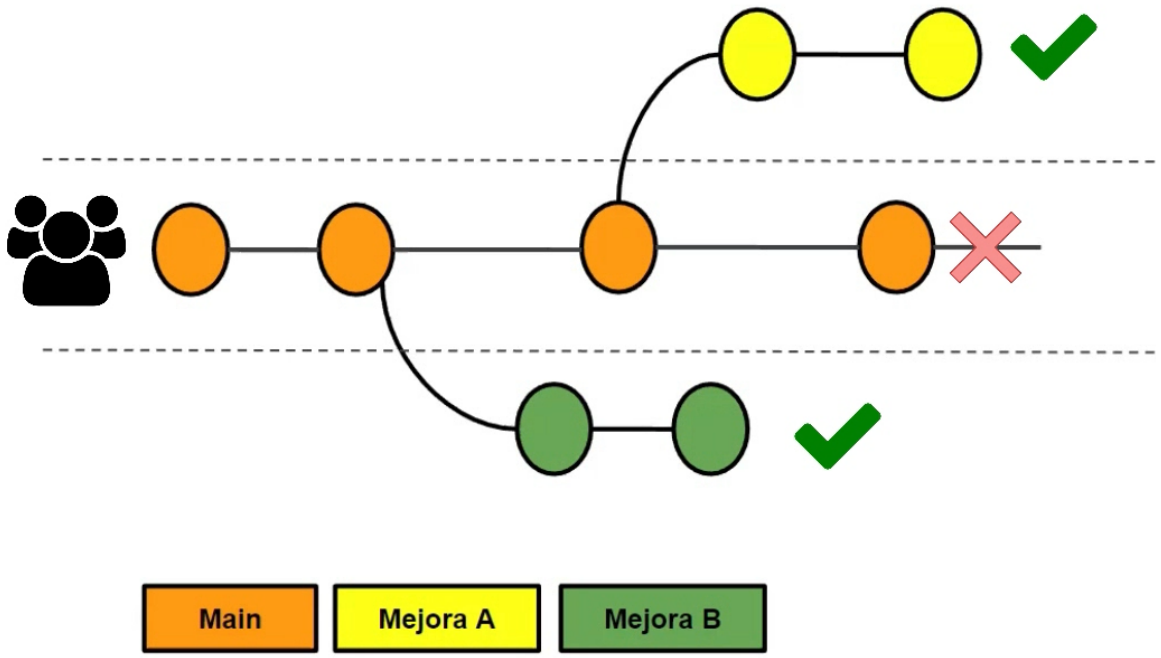
3.6. ¿Cómo trabajar con ramas en Git?

Ramas (branches)

Una rama o branch, es una línea de tiempo que nos provee un entorno de trabajo independiente ya sea para agregar una nueva funcionalidad, para solucionar fallos o bugs, etc. sin afectar la rama principal. Por ende, las ramas en Git son parte del desarrollo diario brindándonos la posibilidad de trabajar en diferentes versiones del producto, en diferentes entornos de trabajo (ej. desarrollo, testing, producción) .

Git parte de una rama principal (master o main) de la cual parten todas las demás.

El sistema de ramificación nos permite entonces, trabajar más ordenados y sin interrumpir el trabajo de otros colegas. Además, nos permite limpiar el historial antes de fusionarlo a la rama principal.



Para ello, Git nos provee los siguientes comandos:

- **git branch,** permite crear, enumerar, cambiar el nombre y eliminar ramas.
- **git checkout,** permite cambiar de rama.
- **git merge**, permite fusionar ramas.

A continuación, te invitamos a mirar un video explicativo:

3.7. ¿Cómo reescribir la historia de Git?

A menudo deseamos reescribir el historial de nuestro proyecto ya sea porque, olvidamos agregar un archivo en una confirmación o porque nos equivocamos al momento de escribir el mensaje de la confirmación.

Si bien el trabajo principal de Git es garantizar que nunca se pierda una confirmación, el mismo está diseñado también para otorgarnos un control total sobre tu workflow de desarrollo.

Para ello, Git nos provee dos comandos:

- **git commit --amend,** permite modificar la confirmación más reciente. Sin embargo, es importante agregar que, el comando no se limita a alterar el commit más reciente, sino que lo reemplaza por completo, por lo que el commit corregido será una entidad nueva con su propia referencia.
- **git rebase,** permite modificar el historial y, reorganizar tu trabajo haciendo el historial del proyecto limpio y lineal.

Para comprenderlo mejor, te invitamos a ver el siguiente video:

Actividad: En el video pudimos observar el comando **git rebase --continue**. Investiga **git rebase --interactive**.

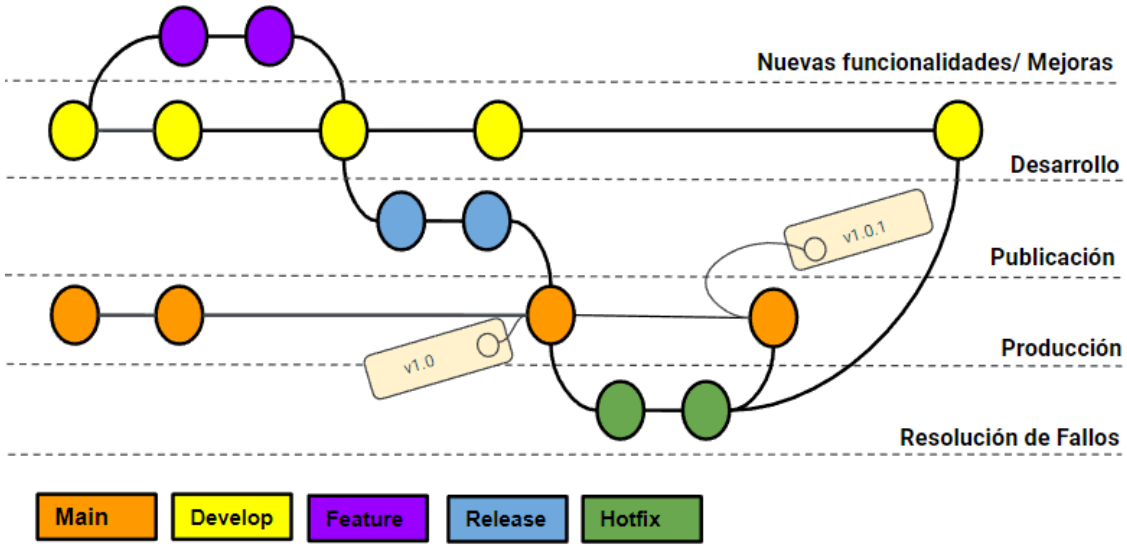
3.8. Git Flow

GITFLOW

La tarea de administrar el fuente confeccionado por un equipo de desarrollo de software puede ser desafiante dado que es un proceso complejo dónde muchas personas no sólo utilizan distintas herramientas y procesos sino que también, muchas veces necesitan trabajar sobre los mismos archivos. *Un flujo de trabajo requiere entonces, controlar los procesos, y el trabajo de cada uno de los integrantes del equipo de desarrollo para así garantizar que cada persona pueda trabajar de forma correcta e independiente. No existe una solución universal para todas las necesidades del equipo pero un buen flujo de trabajo ofrece procesos a fin de que los integrantes del equipo trabajen de forma óptima.*

El sistema de branching o ramificación es la forma que Git tiene para administrar el desarrollo de software en paralelo. Git nos permite crear, eliminar y combinar ramas con un mínimo de esfuerzo y éstas pueden existir el tiempo que necesitemos. Uno de los flujos de trabajo de ramificación más sencillos y que, nos proporciona un marco sólido para gestionar proyectos es **GitFlow**. El mismo fué publicado por *Vincent Driessen* en nvie.

En este flujo de trabajo, nuestro proyecto tiene dos ramas principales o maestras: la rama principal (master o main) y la de desarrollo (develop). La rama master siempre reflejará el estado más actual y listo para ir a producción mientras que, la rama de desarrollo refleja el último desarrollo para la siguiente publicación o release. También disponemos de ramas de respaldo con fines específicos. Las ramas de nuevas funcionalidades y mejoras que tienen por propósito preparar el desarrollo que se incluirá en la próxima publicación. Y las ramas de Hotfix o resolución de fallos, nos permiten hacer cambios críticos y urgentes en la rama maestra sin impedir el trabajo de desarrollo o la siguiente publicación.



Para utilizar este flujo de trabajo puedes ejecutar el comando: **git flow init**

Para más información consulta: https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow

3.9. ¿Cómo trabajo el versionado en Git?

Existen muchas formas de asignar una versión al software. Cada empresa puede establecer sus propias reglas pero, como todo en el área de desarrollo de software, se busca una nomenclatura común o pautas que sirvan de guía.

Existen diferentes propuestas para asignar versiones:

- Versiones por número.
- Versiones por estabilidad.
- Versiones parche.
- Versiones por fecha.

Ejemplo:

Versiones por número

Seguramente te habrás encontrado con versiones establecidas por medio de 3 números: A.B.C dónde cada uno indica una cosa diferente.

- El primer número (A) se le conoce como versión mayor y nos indica la versión principal del software. Ej. 1.0.0
- El segundo número (B) se le conoce como versión menor y nos indica que el software tienen nuevas funcionalidades. Ej. 1.2.0
- El tercer número (C) se le conoce como revisión y nos indica que la versión del software resuelve algún fallo. Ej. 1.2.3

Desafío. Ahora que conocemos el significado de cada número *¿Cómo identificamos cuándo cambiar la versión del software y cuál número cambiar?*

Versiones por estabilidad

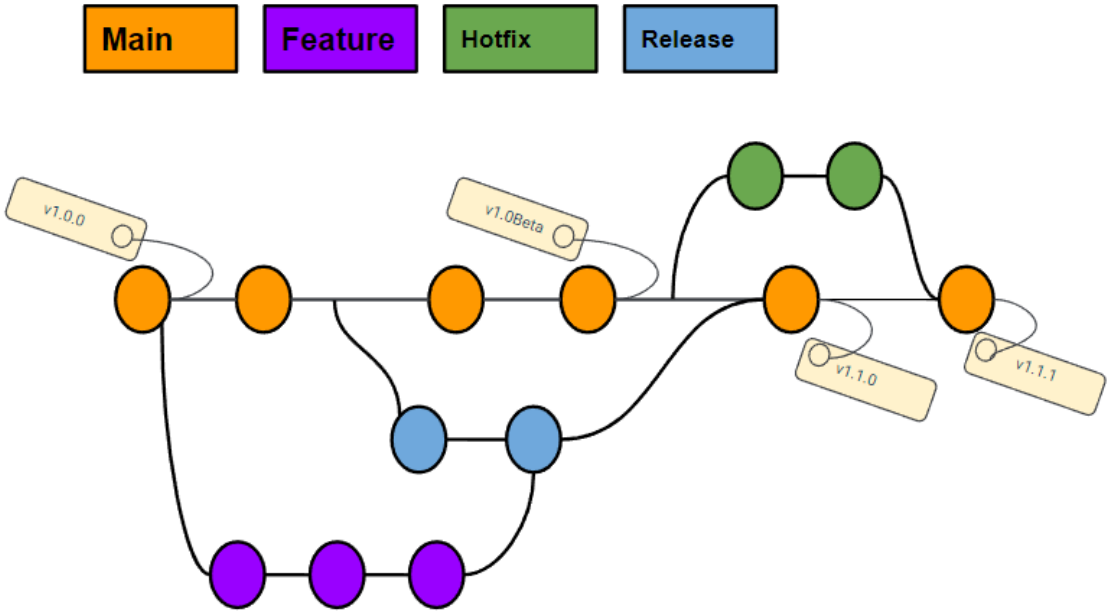
Además de versionar nuestro software por números, podemos clasificarlo por estabilidad: Alpha y Beta.

Alpha, es una versión inestable que probablemente tenga cosas por mejorar pero queremoss sea aprobada para poner a prueba las funcionalidades y encontrar fallos. Ej. 1.0Alpha o 1.0a1.1

Beta, es una versión más estable que Alpha dado que, encontramos el software completo. El objetivo es realizar pruebas de rendimiento, usabilidad y funcionalidad. Ej. 2.0Beta, 2.0b.

Desafío. Investiga sobre el versionamiento parche y por fecha y luego responde. *¿Cuándo será indicado utilizar uno u otro?*

¿Cómo versionamos nuestro software en Git?



Git nos permite listar, observar y etiquetar el versionado de nuestro software a través de los comandos:

- **git tag**
- **git show**

En este punto es importante mencionar que, cuando hablamos del comando **git tag** no nos estamos refierndo a la versión de un archivo en particular, sino del todo el proyecto de manera global. Por ello es que, este comando es útil para trabajar el versionado.

Como se mencionó arriba, la forma de versionado la define el desarrollador pero es importante definir a priori cómo se llevará a cabo. Es decir, no debemos crearlas de manera arbitraria.

Ejemplo:

Supongamos que comenzamos a trabajar en un nuevo repositorio y aún no tenemos una numeración de versión. Entonces, creamos la primer versión.

git tag v0.0.1 -m "Primer versión"

Como puedes observar, el número de versión va acompañado de un mensaje (al igual que el commit).

¿Cómo ver los versionados?

Al igual que con otros comandos, el comando **git tag** (a secas) nos muestra un listado con todas las versiones de etiquetado establecidadas hasta el momento.

Otro comando interesante de ver es el **git show**. El mismo nos permite ver cómo estaba el repositorio en cada versión.

Ejemplo:

git show v0.0.2

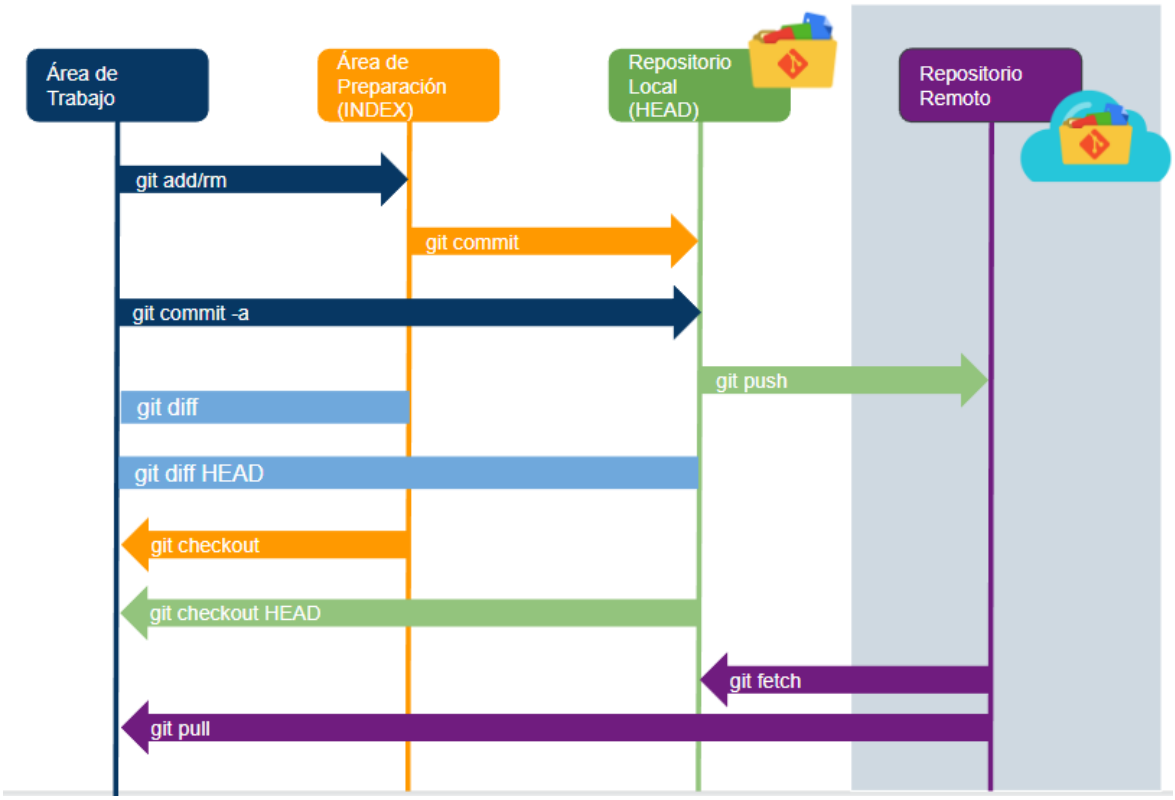
Desafío. Investiga cómo subir las etiquetas de versionado a un repo remoto y cómo eliminarlas.

Para más información consulta el siguiente enlace: https://git-scm.com/book/es/v2/Fundamentos-de-Git-Etiquetado

3.10. ¿Cómo trabajar colaborativamente en Git?

Git nos permite el trabajo colaborativo a través de los siguientes comandos:

- **git remote**. Permite ver crear y eliminar conexiones a otros repositorios.
- **git fetch.** Descarga confirmaciones de un repositorio remoto a uno local.
- **git push.** Permite subir las confirmaciones locales a un repositorio remoto.
- **git pull.** Descarga confirmaciones de un repositorio remoto al repositorio local actualizando el área de trabajo.



Para más información, consulta el siguiente enlace: https://www.atlassian.com/es/git/tutorials/syncing

3.11. Resumen de comandos

Puedes ver un resumen con todos los comandos en **https://training.github.com/downloads/es_ES/github-git-cheat-sheet/**

3.12. Learn Git Branching

Si deseas aprender haciendo, te dejamos a continuación un enlace: **https://learngitbranching.js.org/?locale=es_ES**