UNIVERSIDAD TECNOLÓGICA DE PANAMÁ

FACULTAD DE INGENIERÍA DE SISTEMAS COMPUTACIONALES

CARRERA: LICENCIATURA EN DESARROLLO DE SOFTWARE

MATERIA: DESARROLLO DE SOFTWARE V

TEMA:

USO DE LA HERRAMIENTA GIT

PROFESOR: ERIC AGRAZAL

ESTUDIANTES:

CARLOS JAÉN	**8-861-119**
DAVID ORTEGA	**8-819-2187**
LUIS RAMÍREZ	**8-721-2435**

GRUPO: 1GS222

II SEMESTRE

FECHA: 12 DE AGOSTO DE 2024

Contenido

Uso de la Herramienta Git

A) Introducción a Git

Git es un sistema de control de versiones distribuido que se diferencia del resto en el modo en que modela sus datos. La mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos *(Fig. 1)*, mientras que Git modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos *(Fig. 2)*.

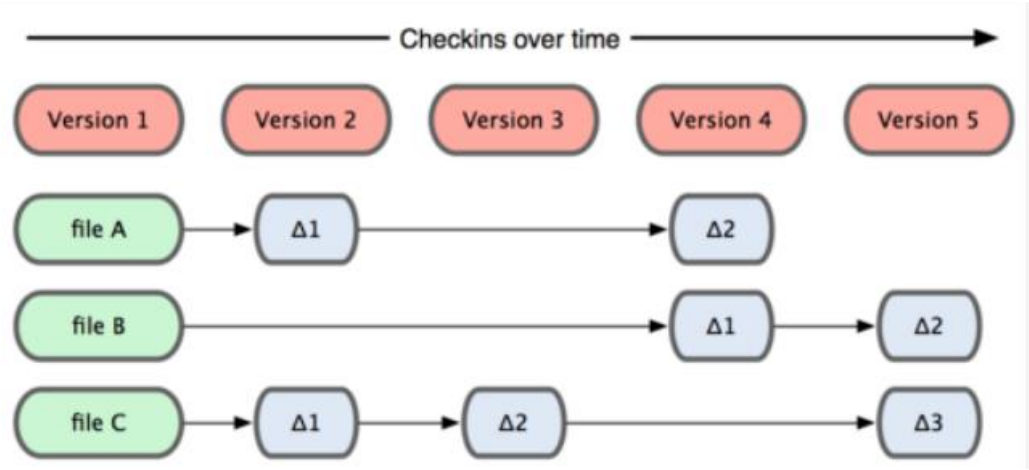

Figura 1. Modelo de datos de los sistemas distribuidos tradicionales. (Universidad de Córdoba, 2015)

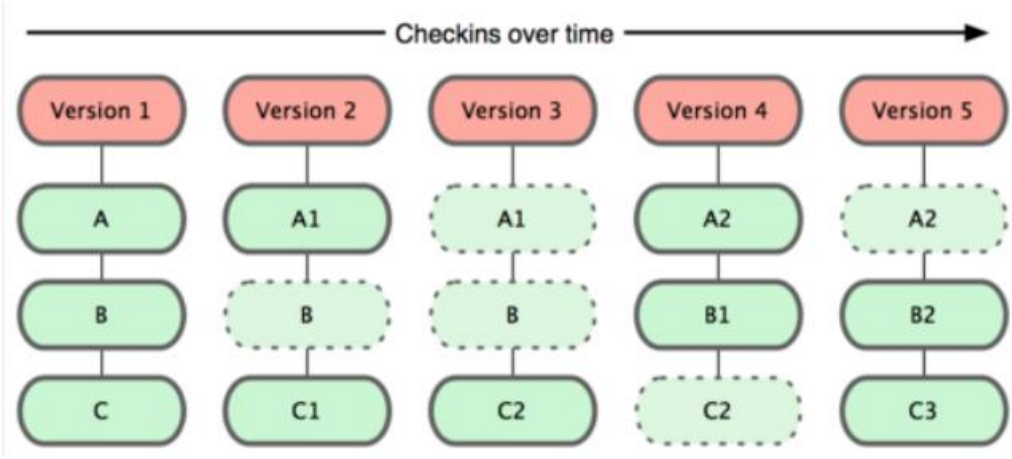

Figura 2. Modelo de datos de Git. (Universidad de Córdoba, 2015)

1. Los tres estados

Git tiene tres estados principales en los que se pueden encontrar tus archivos:

- Confirmado (committed): significa que los datos están almacenados de manera segura en tu base de datos local.

- Modificado (modified): significa que has modificado el archivo, pero todavía no lo ha confirmado a tu base de datos.
- Preparado (staged): significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git *(Fig. 3)*: el directorio de Git (Git directory), el directorio de trabajo (working directory), y el área de preparación (staging area).

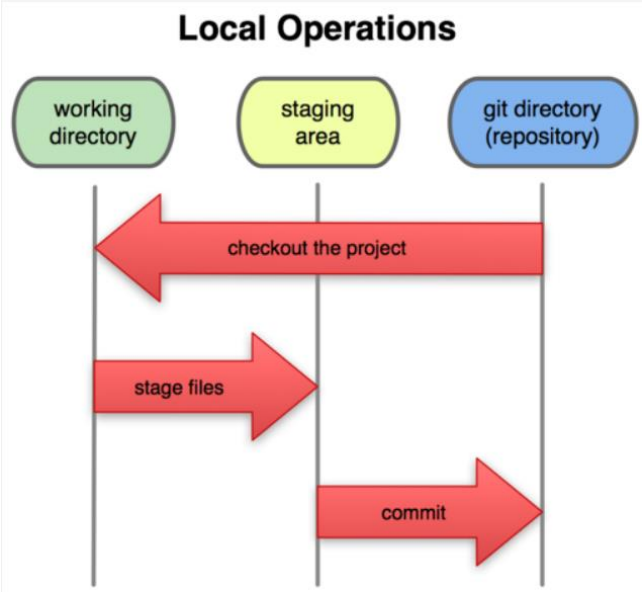

Figura 3. Directorio de trabajo, área de preparación, y directorio de Git. (Universidad de Córdoba, 2015)

2. Flujos de trabajo distribuidos con Git

Hemos visto en qué consiste un entorno de control de versiones distribuido, pero más allá de la simple definición, existe más de una manera de gestionar los repositorios. Estos son los flujos de trabajo más comunes en Git:

a) Flujo de trabajo centralizado:

Existe un único repositorio o punto central que guarda el código y todo el mundo sincroniza su trabajo con él. Si dos desarrolladores clonan desde el punto central, y ambos hacen cambios; tan solo el primero de ellos en enviar sus cambios de vuelta lo podrá hacer limpiamente. El segundo desarrollador deberá fusionar previamente su trabajo con el del primero, antes de enviarlo, para evitar el sobre escribir los cambios del primero.

Existe un único repositorio o punto central que guarda el código y todo el mundo sincroniza su trabajo con él *(Fig. 4)*. Si dos desarrolladores clonan desde el punto central, y ambos hacen cambios; tan solo el primero de ellos en enviar sus cambios de vuelta lo podrá

hacer limpiamente. El segundo desarrollador deberá fusionar previamente su trabajo con el del primero, antes de enviarlo, para evitar el sobre escribir los cambios del primero.

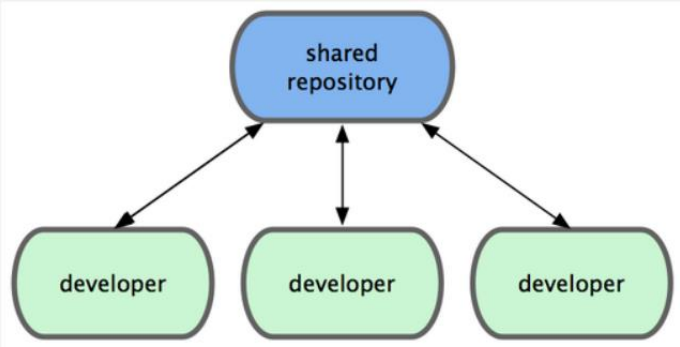

Figura 4. Flujo de trabajo centralizado. (Universidad de Córdoba, 2015)

b) Flujo de trabajo del Gestor de Integraciones:

Al permitir múltiples repositorios remotos, en Git es posible tener un flujo de trabajo donde cada desarrollador tenga acceso de escritura a su propio repositorio público y acceso de lectura a los repositorios de todos los demás *(Fig. 5)*. Habitualmente, este escenario suele incluir un repositorio canónico, representante "oficial" del proyecto. Este modelo se puso muy de moda a raíz de la forja GitHub.

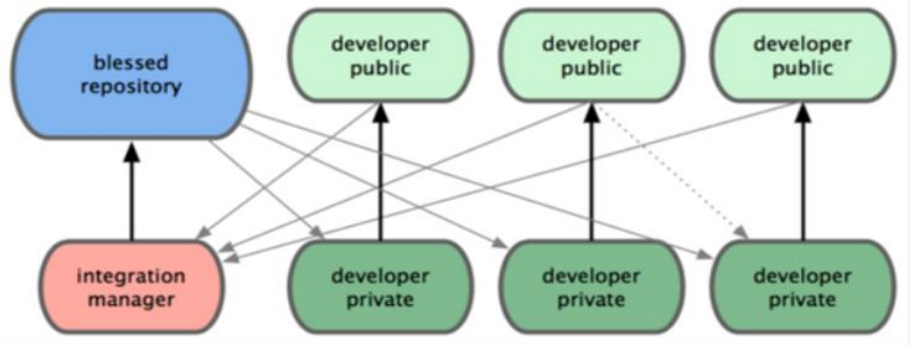

Figura 5. Flujo de Trabajo del Gestor de Integraciones. (Universidad de Córdoba, 2015)

c) Flujo de Trabajo con Dictador y Tenientes:

Es una variante del flujo de trabajo con múltiples repositorios. Se utiliza generalmente en proyectos muy grandes, con cientos de colaboradores. Un ejemplo muy conocido es el del kernel de Linux. Unos gestores de integración se encargan de partes concretas del repositorio; y se denominan tenientes. Todos los tenientes rinden cuentas a un gestor de integración; conocido como el dictador benevolente. El repositorio del dictador

benevolente es el repositorio de referencia, del que recuperan (pull) todos los colaboradores (*Fig. 6*).

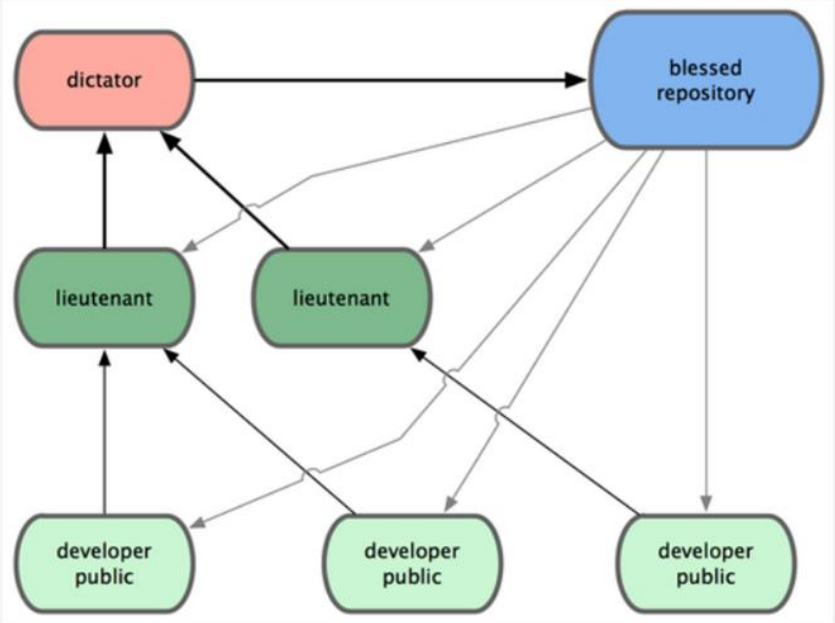

Figura 6. Flujo de trabajo con Dictador y Tenientes. (Universidad de Córdoba, 2015)

B) Funcionalidad de la herramienta Git

Git es una herramienta de control de versiones distribuido que permite a los desarrolladores rastrear, gestionar y coordinar cambios en archivos, generalmente de código fuente, a lo largo del tiempo. Su funcionalidad principal se centra en la gestión eficiente del historial de un proyecto, permitiendo la colaboración en equipos, la experimentación segura y la recuperación de versiones anteriores del código.

A través de Git, los usuarios pueden crear repositorios, que son espacios donde se almacena el código y su historial. Cada vez que se realiza un cambio en los archivos del proyecto, este puede ser registrado en el repositorio mediante un proceso llamado commit, que guarda un "snapshot" de los archivos en ese momento. Estos commits se encadenan cronológicamente, formando un historial detallado de todas las modificaciones realizadas en el proyecto.

Una de las características más destacadas de Git es su capacidad para trabajar con ramas (branches). Las ramas permiten desarrollar nuevas características, corregir errores o experimentar con cambios sin alterar el código principal. Una vez que el trabajo en una rama se completa, puede fusionarse (merge) con otras ramas, incorporando los cambios de forma controlada.

Git es una herramienta distribuida, lo que significa que cada copia del repositorio en un equipo contiene todo el historial del proyecto. Esto hace que el trabajo en equipo sea más eficiente, ya que no depende de un único servidor central. Los desarrolladores pueden trabajar de manera independiente y luego sincronizar sus cambios con otros miembros del equipo, lo que permite una colaboración robusta y flexible.

Además, Git facilita la resolución de conflictos cuando varios desarrolladores modifican el mismo archivo o área del código. Esto, junto con su capacidad para revertir cambios y experimentar con diferentes versiones del código, lo convierte en una herramienta esencial en el desarrollo moderno de software.

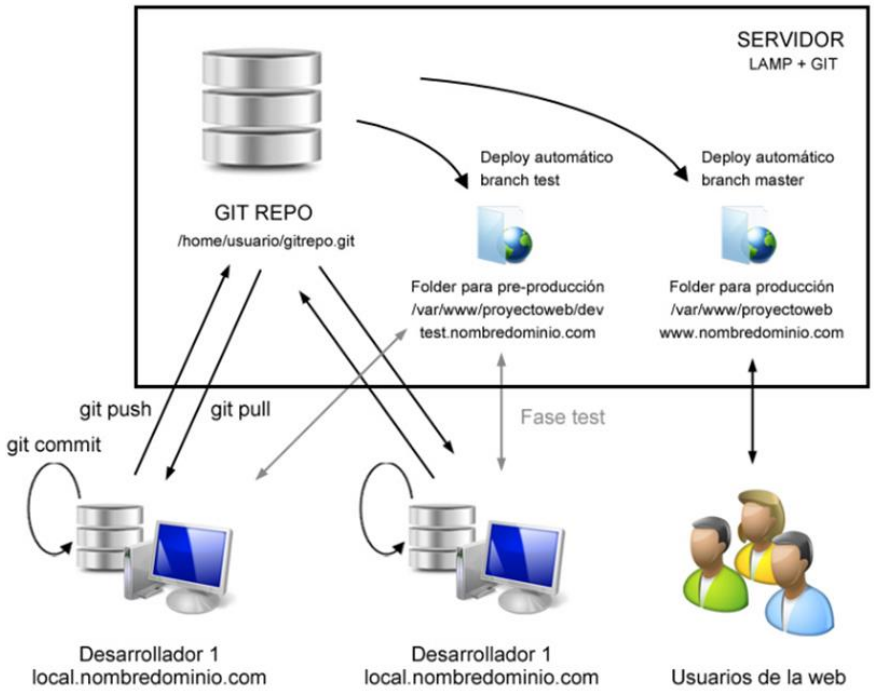

Figura 7. Funcionalidad de GIT (Danielnavarroymas.com, 2019)

C) Conceptos

1. Repositorio (Repository) en Git

Un **repositorio** en Git es una estructura de datos que almacena todas las versiones de los archivos de un proyecto. Es el lugar donde Git guarda toda la información necesaria para gestionar el historial de cambios y las versiones de los archivos. Hay dos tipos principales de repositorios en Git:

a) **Repositorio Local**: Es el repositorio que existe en tu máquina local. Aquí es donde realizas la mayoría de las operaciones de Git, como commits, branches, merges, etc. Cada desarrollador tiene su propio repositorio local donde puede hacer cambios y probarlos antes de compartirlos con otros.

b) **Repositorio Remoto**: Es una copia del repositorio que está alojada en un servidor (por ejemplo, GitHub, GitLab, Bitbucket, etc.) y se utiliza para compartir el trabajo con otros miembros del equipo. Los repositorios remotos permiten la colaboración entre varios desarrolladores. Los comandos git push, git pull, y git fetch son utilizados para interactuar con el repositorio remoto *(fig. 8)*.

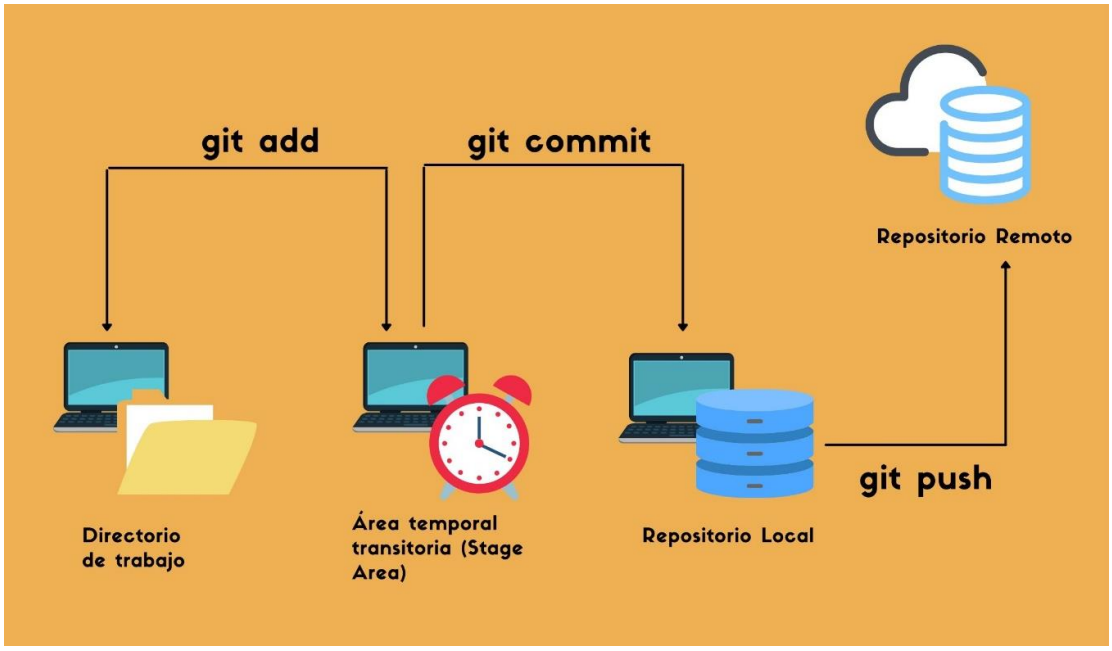

Figura 8. Flujo de Trabajo básico con un repositorio remoto (Hellenbar, s.f.)

2. Directorio de Trabajo (Working Directory) en Git

El **Directorio de Trabajo** o **Working Directory** es la carpeta en tu sistema de archivos donde están ubicados los archivos de tu proyecto en su versión actual. Este directorio contiene una copia de los archivos que están siendo versionados por Git y es donde trabajas directamente, realizando ediciones, añadiendo nuevos archivos, o eliminando otros.

Dentro del directorio de trabajo, hay tres estados clave que un archivo puede tener:

a) **Archivos Modificados (Modified)**: Son archivos que han sido cambiados en el directorio de trabajo, pero aún no han sido preparados para el commit.

b) **Archivos en Stage (Staged)**: Son archivos que han sido marcados para ser incluidos en el próximo commit. Esto se realiza utilizando el comando git add. Estos archivos están en una especie de "área de preparación" antes de ser confirmados en el historial de cambios.

c) **Archivos Confirmados (Committed)**: Son archivos que han sido guardados en el repositorio local. Una vez que un archivo ha sido confirmado (usando el comando git commit), se convierte en parte del historial de versiones del proyecto.

Resumen de los Conceptos

- **Repositorio**: Es el almacenamiento de todos los archivos y sus historiales de cambio. Puede ser local o remoto.
- **Directorio de Trabajo**: Es la copia de los archivos en los que trabajas y que están controlados por Git. Aquí se hacen modificaciones que luego se pueden añadir y confirmar en el repositorio.

3. Índice

¿Qué es Staging Area en Git?

El Staging Area de Git, o git staged area, es el lugar en el que se encuentran datos de un proyecto y sus cambios. En esta área puedes asignarle un nombre a una nueva versión y crear una "copia" de cómo quedaría dicha versión en el repositorio en producción.

En otras palabras, el Staging Area es un archivo sencillo, generalmente almacenado en tu directorio de Git, que recopila información acerca de lo que va a estar presente en la próxima confirmación, listos para ser enviados al repositorio de Git.

Algunas veces se denomina Índice o Index, pero se está convirtiendo en estándar el referirse a ello como el área de preparación (staging area). Entonces, ¿qué es el staging área de Git?

En un proyecto de Git, el Staging Area -que muchos confunden llamándolo el staging area de Github-, se utiliza junto al directorio de trabajo (working directory) y el repositorio (repository), que es una copia de las versiones del proyecto en sí y el lugar donde se almacenan los datos de los elementos para el proyecto, respectivamente.

Para qué sirve Staging Area de Git

El funcionamiento del Staging Area depende directamente del Working Copy, ya que el área de preparación es la segunda zona dentro de las 3 zonas principales de Git. Pese a que cada una de las zonas sea fundamental para el funcionamiento de las otras, estas no están directamente conectadas y se necesita el uso de diferentes comandos para interconectarlas: git add, git commit staged y git push (o git push -u).

La función principal del Index es ser el puente de paso de los cambios realizados en el código de trabajo que se realizan dentro del Working Copy, para luego, por medio del comando git commit, confirmar dichos cambios y pasarlos directamente a tu repositorio local.

Dentro del repositorio local hay dos caminos a seguir que puedes tomar. Volver a enviar tu código al Working Copy para seguir haciendo ajustes o, por el contrario, enviar todos los cambios ya

confirmados desde tu repositorio local y sincronizarlos al servidor Git que estés usando para trabajar en conjunto con tu equipo de trabajo.

En conclusión, el staging area. no solo hace parte de las 3 zonas de trabajo principal dentro de Git, es la parte que permite conectar cualquier cambio o ajuste dentro de los códigos de trabajo que estés realizando en tu proyecto para luego ser confirmados y enviados a los servidores Git.

4. Head

El concepto de HEAD es muy simple: se refiere al commit en el que está tu repositorio posicionado en cada momento. Por regla general HEAD suele coincidir con el último commit de la rama en la que estés, ya que habitualmente estás trabajando en lo último. Pero si te mueves hacia cualquier otro commit anterior entonces el HEAD estará más atrás.

De hecho, si tienes el repositorio actualizado (te has traído los últimos cambios de origin) y estás trabajando en la rama main lo más habitual es que coincidan las tres cosas.

Origin: como seguramente sabrás Git es un sistema de control de código distribuido. Esto quiere decir que, aunque todos los desarrolladores tienen una copia exacta del mismo repositorio en su disco duro, existen uno o más repositorios remotos contra los que trabajamos, y que son los que almacenan el estado final del producto. Estos repositorios remotos se suelen llamar simplemente "remotos", y no todo el mundo tiene permiso para enviar commits hacia ellos (lo que se llama hacer un push).

Bien, pues origin es simplemente el nombre predeterminado que recibe el repositorio remoto principal contra el que trabajamos. Cuando clonamos un repositorio por primera vez desde GitHub o cualquier otro sistema remoto, el nombre que se le da a ese repositorio "maestro" es precisamente origin.

Main: Como sabes, los sistemas de control de código fuente como Git trabajan con el concepto de ramas (o branches en inglés). Dicho de manera rápida y básica, una rama no es más que un nombre que se da a un commit, a partir del cual se empieza a trabajar de manera independiente y con el que se van a enlazar nuevos commits de esa misma rama. Las ramas pueden mezclarse de modo que todo el trabajo hecho en una de ellas pase a formar parte de otra.

Por si no lo tienes claro, un commit es un conjunto de cambios en los archivos que hemos dado por buenos y que queremos almacenar como una instantánea de cara al futuro. Los commits se relacionan unos con otros en una o varias secuencias para poder ir viendo la historia de un determinado archivo a lo largo del tiempo. Es el concepto central de todo sistema de control de código.

Existe una rama predeterminada que se crea automáticamente cuando se crea un repositorio que se llama rama main.

Actualización posterior: en realidad, hasta mediados del año 2020 esta rama se llamaba "master" y no "main". Pero por aquel entonces, aunque existen diversas teorías sobre el origen de ese nombre, se decidió que era una referencia al esclavismo ("master" de "amo", frente a "slave" de "esclavo. Otras teorías hablan, con más lógica IMHO, de que viene de "master copy" o copia maestra o principal) y que había que renombrarla. Así que ahora es "main" y no "master".

Por regla general a main se la considera la rama principal y la raíz de la mayoría de las demás ramas. Lo más habitual es que en main se encuentre el "código definitivo", que luego va a producción, y es la rama en la que se mezclan todas las demás tarde o temprano para dar por finalizada una tarea e incorporarla al producto final:

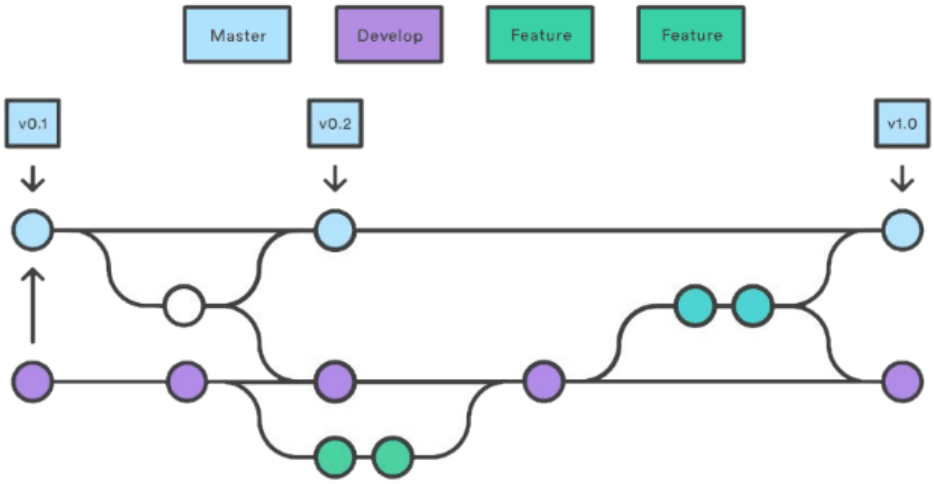

Figura 9. Rama Máster y develop (Alarcón, s.f.)

Fig. 9 que ilustra la rama máster y algunas otras como develop o ramas para el desarrollo de características concretas, y como luego se mezclan entre sí

Esta manera de trabajar con ramas nos permite llevar en paralelo varios desarrollos relacionados sin importar que cada uno de ellos termine en momentos muy diferentes, ya que no interfieren, pudiendo mezclarlos todos al final.

Lo más habitual es que para poder mezclar otra rama cualquiera con main haya que pedir permiso y que alguien lo revise todo antes de permitirlo. Es lo que se denomina un "pull request". O simplemente que la rama main se encuentre protegida de modo que solo si se pasan todos los test y aseguramos que el producto funciona, sea posible mezclarse con ella. De este modo impedimos que cualquiera pueda llevar al producto final código que no cumple unos mínimos de calidad.

Tengamos un ejemplo de las definiciones que hemos expuesto. Digamos que este es el log, mostrando las ramas, del repositorio de un proyecto Open Source en el que participo. El HEAD local se muestra como un punto con borde azul. Como se puede observar, en el momento en el que saqué la captura (fig. 10) tenía en local la rama master (hoy en día sería main) y estaba ubicado en el último commit (cuyo hash identificador comienza por bdf09f20) y coincide con el último commit en el origen:

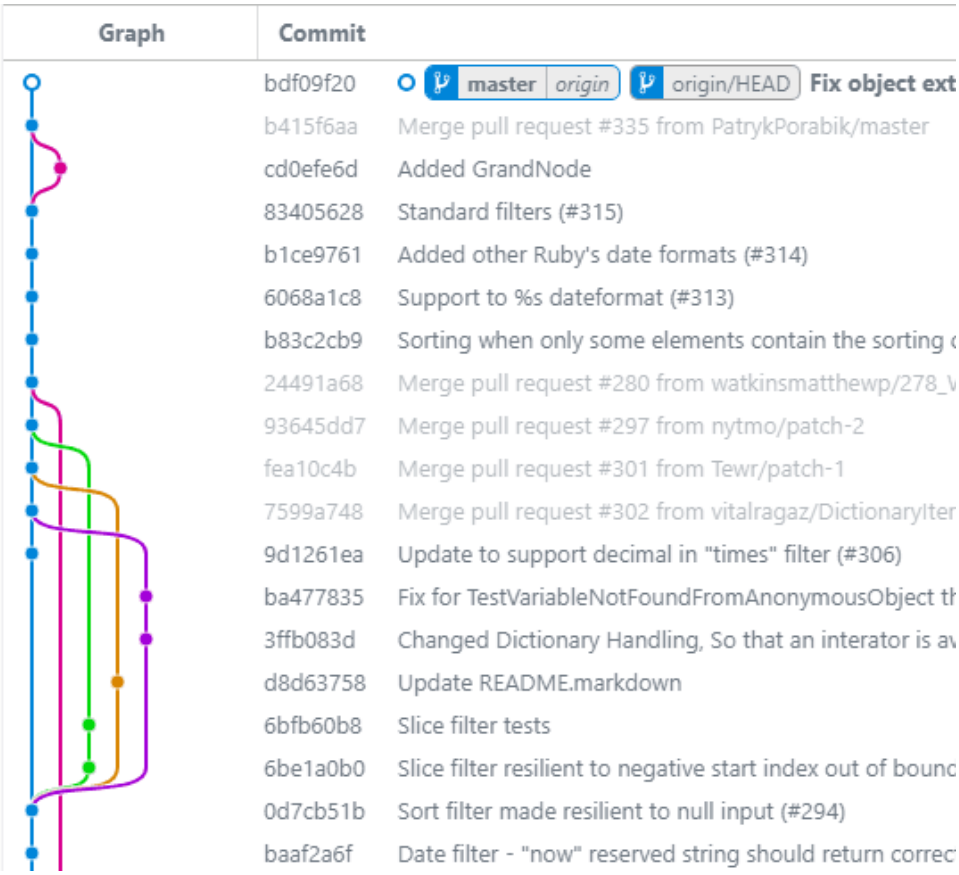

Figura 10. Log mostrando ramas del repositorio de un proyecto Open Source (Alarcón, s.f.)

Si por el contrario me muevo a otra rama o a un commit anterior, o si el repositorio de origen en la misma rama va por delante de lo que yo tengo en local, ninguno de los tres tiene por qué coincidir. Por ejemplo, me he movido a un commit anterior (el 6068a1c8): y ahora mismo mi HEAD local está 5 commits por detrás (es la que está en negrita con un punto de borde azul), mientras que la rama máster (que replica a la misma en el origen) está 5 commits por delante. El HEAD del repositorio remoto (origen) está en el último commit (*fig. 11*):

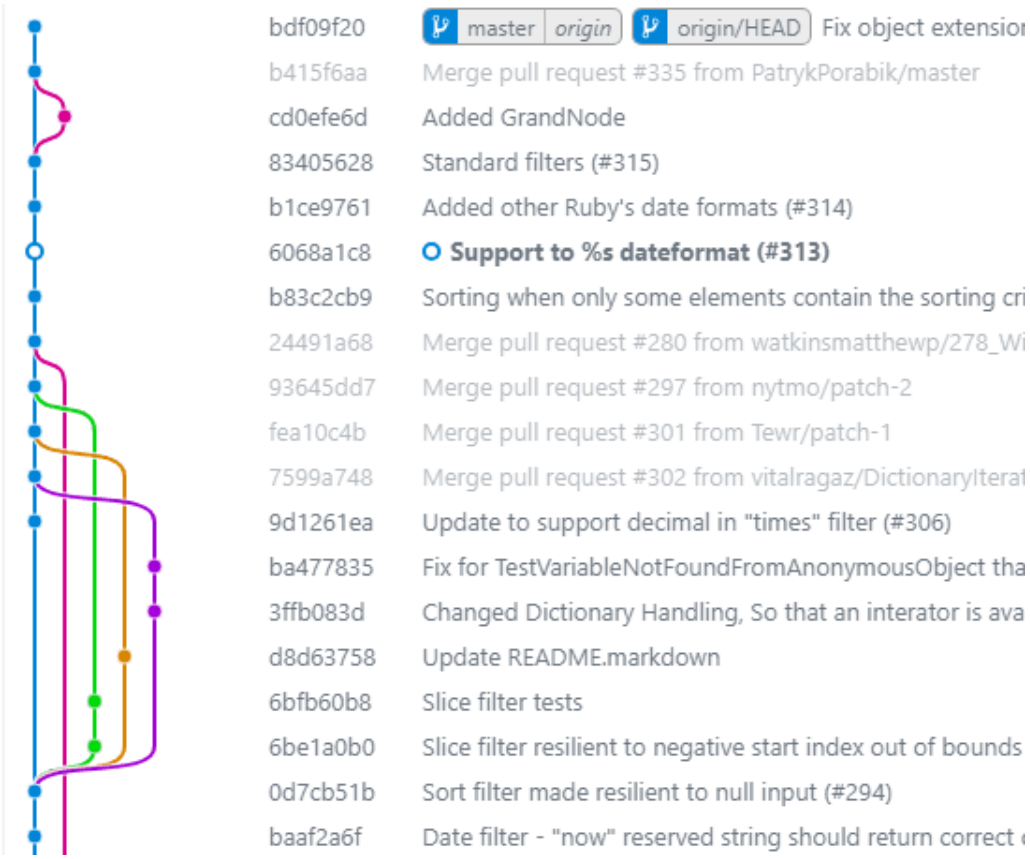

Figura 11. Movimiento a otra rama o a un commit anterior del Open Source (Alarcón, s.f.)

Aunque se trata de conceptos sencillos no siempre están claros para todo el mundo. De hecho, tanto main como origin son nombres sobre los que existe una convención en cuanto a su uso, pero no tienen por qué existir ya que la rama principal puede llamarse de otra manera cualquiera, y el origen principal también puede tener otro nombre (aunque estos dos sean los más habituales).

Sin embargo, HEAD es un nombre constante y un concepto que no tiene discusión, refiriéndose siempre al commit en el que tenemos posicionado el repositorio en el momento actual. Independientemente de la rama en la que estemos, de cómo se llame la rama principal o el repositorio principal remoto.

5. Ramas

Ramificaciones en Git

Cualquier sistema de control de versiones moderno tiene algún mecanismo para soportar el uso de ramas. Cuando hablamos de ramificaciones, significa que tú has tomado la rama principal de desarrollo (master) y a partir de ahí has continuado trabajando sin seguir la rama principal de desarrollo. En muchos sistemas de control de versiones este proceso es costoso, pues a menudo

requiere crear una nueva copia del código, lo cual puede tomar mucho tiempo cuando se trata de proyectos grandes.

Algunas personas resaltan que uno de los puntos más fuertes de Git es su sistema de ramificaciones y lo cierto es que esto le hace resaltar sobre los otros sistemas de control de versiones. ¿Por qué esto es tan importante? La forma en la que Git maneja las ramificaciones es increíblemente rápida, haciendo así de las operaciones de ramificación algo casi instantáneo, al igual que el avance o el retroceso entre distintas ramas, lo cual también es tremendamente rápido. A diferencia de otros sistemas de control de versiones, Git promueve un ciclo de desarrollo donde las ramas se crean y se unen ramas entre sí, incluso varias veces en el mismo día. Entender y manejar esta opción te proporciona una poderosa y exclusiva herramienta que puede, literalmente, cambiar la forma en la que desarrollas.

¿Qué es una rama?

Para entender realmente cómo ramifica Git, previamente hemos de examinar la forma en que almacena sus datos.

Git no los almacena de forma incremental (guardando solo diferencias), sino que los almacena como una serie de instantáneas (copias puntuales de los archivos completos, tal y como se encuentran en ese momento).

En cada confirmación de cambios (commit), Git almacena una instantánea de tu trabajo preparado. Dicha instantánea contiene además unos metadatos con el autor y el mensaje explicativo, y uno o varios apuntadores a las confirmaciones (commit) que sean padres directos de esta (un padre en los casos de confirmación normal, y múltiples padres en los casos de estar confirmando una fusión (merge) de dos o más ramas).

Para ilustrar esto, vamos a suponer, por ejemplo, que tienes una carpeta con tres archivos, que preparas (stage) todos ellos y los confirmas (commit). Al preparar los archivos, Git realiza una suma de control de cada uno de ellos (un resumen SHA-1), almacena una copia de cada uno en el repositorio (estas copias se denominan "blobs"), y guarda cada suma de control en el área de preparación (staging area):

```
$ git add README test.rb LICENSE

$ git commit -m 'initial commit of my project'
```

Cuando creas una confirmación con el comando git commit, Git realiza sumas de control de cada subdirectorio (en el ejemplo, solamente tenemos el directorio principal del proyecto), y las guarda como objetos árbol en el repositorio Git. Después, Git crea un objeto de confirmación con los metadatos pertinentes y un apuntador al objeto árbol raíz del proyecto.

En este momento, el repositorio de Git contendrá cinco objetos: un "blob" para cada uno de los tres archivos, un árbol con la lista de contenidos del directorio (más sus respectivas relaciones

con los "blobs"), y una confirmación de cambios (commit) apuntando a la raíz de ese árbol y conteniendo el resto de metadatos pertinentes *(Fig. 12).*

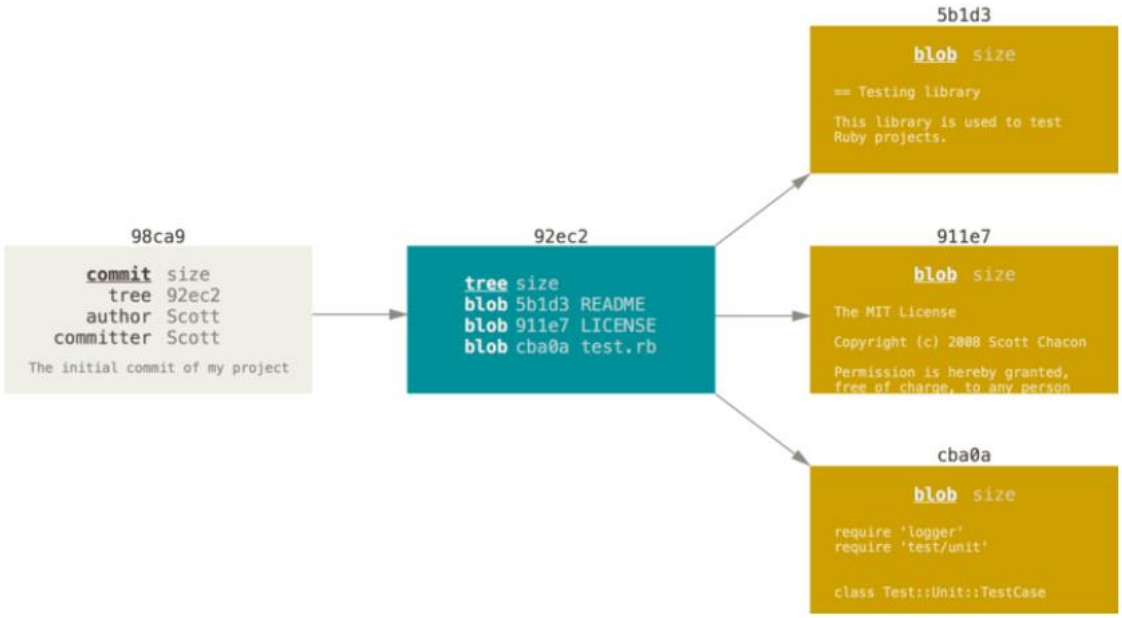

Figura 12. Una confirmación y sus árboles. (Libro Oficial de Git, 2005)

Si haces más cambios y vuelves a confirmar, la siguiente confirmación guardará un apuntador a su confirmación precedente (Fig. 13).

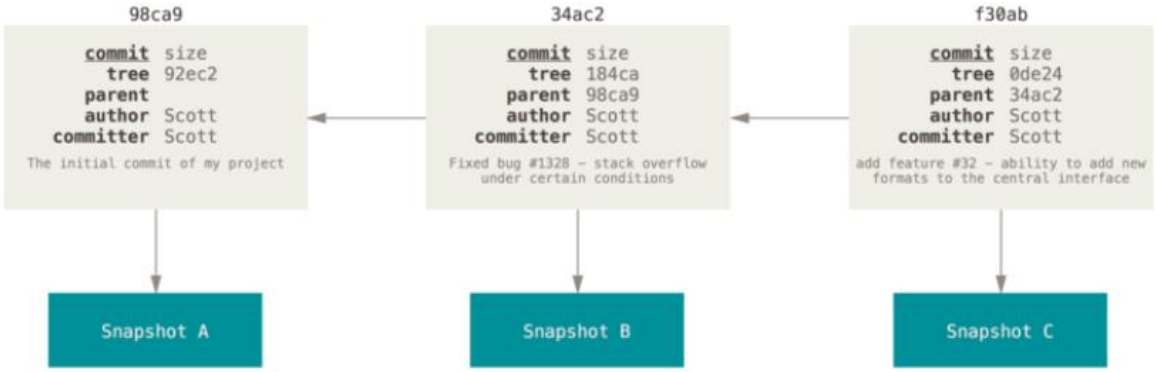

Figura 13. Confirmaciones y sus predecesoras. (Libro Oficial de Git, 2005)

Una rama Git es simplemente un apuntador móvil apuntando a una de esas confirmaciones. 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.

Crear una Rama Nueva

¿Qué sucede cuando creas una nueva rama? Bueno…, simplemente se crea un nuevo apuntador para que lo puedas mover libremente. Por ejemplo, supongamos que quieres crear una rama nueva denominada "testing". Para ello, usarás el comando git branch: *$ git branch testing*.

Esto creará un nuevo apuntador apuntando a la misma confirmación donde estés actualmente.

Y, ¿cómo sabe Git en qué rama estás en este momento? Pues…, mediante un apuntador especial denominado HEAD que nos referimos anteriormente.

En Git, es simplemente el apuntador a la rama local en la que tú estés en ese momento, en este caso la rama master; pues el comando git Branch solamente crea una nueva rama, pero no salta a dicha rama.

Esto puedes verlo fácilmente al ejecutar el comando git log para que te muestre a dónde apunta cada rama. Esta opción se llama --decorate.

Cambiar de Rama

Para saltar de una rama a otra, tienes que utilizar el comando git checkout. Hagamos una prueba, saltando a la rama testing recién creada: *$ git checkout testing.* Esto mueve el apuntador HEAD a la rama testing *(Fig. 14)*.

Figura 14. Dos ramas apuntando al mismo grupo de confirmaciones. (Libro Oficial de Git, 2005)

Observamos algo interesante: la rama testing avanza, mientras que la rama master permanece en la confirmación donde estaba cuando lanzaste el comando *git checkout* para saltar. Volvamos ahora a la rama master: *$ git checkout master.*

Este comando realiza dos acciones: Mueve el apuntador HEAD de nuevo a la rama master, y revierte los archivos de tu directorio de trabajo; dejándolos tal y como estaban en la última instantánea confirmada en dicha rama master. Esto supone que los cambios que hagas desde este momento en adelante, divergirán de la antigua versión del proyecto. Básicamente, lo que se está haciendo es rebobinar el trabajo que habías hecho temporalmente en la rama testing; de tal forma que puedas avanzar en otra dirección diferente.

Procedimientos Básicos para Ramificar y Fusionar

Vamos a presentar un ejemplo simple de ramificar y de fusionar, con un flujo de trabajo que se podría presentar en la realidad. Imagina que sigues los siguientes pasos:

1. Trabajas en un sitio web.

2. Creas una rama para un nuevo tema sobre el que quieres trabajar.

3. Realizas algo de trabajo en esa rama.

En este momento, recibes una llamada avisándote de un problema crítico que has de resolver.

Y sigues los siguientes pasos:

1. Vuelves a la rama de producción original.

2. Creas una nueva rama para el problema crítico y lo resuelves trabajando en ella.

3. Tras las pertinentes pruebas, fusionas (merge) esa rama y la envías (push) a la rama

de producción.

4. Vuelves a la rama del tema en que andabas antes de la llamada y continúas tu

trabajo.

6. Commits

¿Sabes que son commits? Los commits son la base principal del trabajo de Git, ya que es el comando más usado para guardar cualquier cambio en esta herramienta. Si te preguntas qué es un commit y como agregar un commit en git, te puedes hacer una idea al entenderlo como una captura de pantalla del trabajo que haces cada segundo en Git, allí se crea una versión del proyecto en lo que sería el repositorio local.

¿Qué es un Commit en Git?

En el proceso de entender qué es un Commit en Git, debes saber que los commits tienen dos características muy importantes las cuales sirven para comprender mejor su funcionamiento.

Como usuario, puedes recordar los cambios que fueron aplicados en cualquier versión con fechas anteriores, incluso logrando revertir el progreso del proyecto a esa última versión.

Si se editan varios commits en diferentes partes del proyecto, estos cambios no se sobrescribirán entre sí, aunque los autores no tengan conexión alguna. Esto da peso en la balanza a favor de Git en comparación con otras herramientas.

Funcionamiento de un commit

Sintetizando en lo que se puede definir que es un Commit en Git, este se puede entender como un sistema de control de versiones en el cual se crea lo que se entiende como confirmaciones, estas confirmaciones son las unidades básicas más importantes dentro del cronograma del proyecto de Git, las cuales se presentan como instantáneas o hitos dentro del proyecto de Git.

Para tener acceso a estas confirmaciones, es necesario el comando git commit que es para capturar exactamente el estado del proyecto en un momento concreto, después de realizar la captura, las confirmaciones instantáneas son almacenadas en el repositorio local de Git.

Esta forma de almacenar las confirmaciones resulta ser opuesta al sistema SVN git commit - amend , ya que este almacena los datos o lo que serían confirmaciones en un repositorio central el cual necesita obligatoriamente la interacción con este sistema centralizado para realizar cambios y ajustes dentro del proyecto, al final esto da ventaja a Git, ya que no obliga a interactuar con el repositorio central hasta que sea netamente necesario para el desarrollador, permitiéndole trabajar siempre desde su repositorio local.

¿En qué se diferencia un Git commit de un SVN commit?

Pese a que cada comando tenga ciertas similitudes, entre esas su nombre, es importante saber en qué se diferencia cada uno, ya que esto puede terminar siendo confuso para los desarrolladores que comienzan sus primeros pasos con Git o, por el contrario, los que ya han tenido encuentros con SVN.

Es por esto que se debe aclarar la diferencia entre estos dos en el proceso de entender qué es un Commit en Git. Uno es modelo de aplicaciones centralizadas (SVN), el otro es un modelo de aplicaciones distribuidas (Git).

En SVN se envía una confirmación por parte del usuario local a un repositorio compartido el cual es centralizado y remoto, solo con el acceso a este repositorio remoto se pueden ejecutar los cambios o ajustes de versiones.

Por el contrario, en Git, los repositorios se encuentran distribuidos, como se explicó anteriormente, las confirmaciones se realizan en el repositorio local y no se requiere ninguna interacción con otros repositorios de Git que sea dependiente a algún servidor. Las confirmaciones de Git se pueden enviar en cualquier momento a cualquier repositorio remoto.

Esta ventaja del comando git commit o git commit -s resulta en una mayor eficiencia en las operaciones ya que una versión particular de un archivo no tiene que ser «montada» a partir de sus diferencias y es por eso que la revisión completa de cada archivo está disponible de forma inmediata en la base de datos interna de Git, eso sí, teniendo en cuenta la diferencia que puede haber en las versiones de los proyectos que estén almacenadas en cada repositorio local de los desarrolladores.

Conclusiones

Debido a su capacidad para gestionar el historial de cambios y facilitar la colaboración entre desarrolladores, Git se ha consolidado como una herramienta esencial en el desarrollo de software moderno. Los equipos pueden trabajar de manera eficiente y autónoma sin depender de un servidor central gracias a su modelo de datos basado en instantáneas y su arquitectura distribuida.

Los tres estados principales de Git, confirmado, modificado y preparado, junto con las diferentes secciones del proyecto, como el directorio de trabajo y el área de preparación, permiten un control detallado y preciso sobre el desarrollo del proyecto. Los diversos flujos de trabajo distribuidos, desde el flujo centralizado hasta el modelo de dictador y tenientes, demuestran la versatilidad de Git para adaptarse a diversos tamaños de equipo y estilos de desarrollo. La funcionalidad de Git, que incluye la capacidad de crear ramas, fusionar cambios y resolver conflictos, proporciona herramientas poderosas para gestionar la evolución del código y mantener la integridad del proyecto. Gracias a su diseño distribuido, Git no solo mejora la eficiencia del trabajo en equipo, sino que también facilita la experimentación y la recuperación de versiones anteriores del código. –Carlos Jaén—

Los repositorios y el directorio de trabajo son los dos componentes principales de Git, que brindan una infraestructura sólida para la gestión del código fuente. Los repositorios, tanto locales como remotos, son la base de la gestión de versiones, lo que permite a los desarrolladores almacenar y compartir de manera efectiva el historial de cambios de un proyecto. El repositorio remoto facilita la colaboración y la sincronización entre varios miembros del equipo, mientras que el repositorio local sirve como el entorno de trabajo privado del desarrollador.

Sin embargo, el directorio de trabajo es donde se realizan las modificaciones diarias en los archivos del proyecto. Los archivos en este directorio pueden estar modificados, en etapa de preparación o confirmados. Esta estructura de estados contribuye a mantener un flujo de trabajo organizado, permitiendo a los desarrolladores gestionar cambios de manera efectiva antes de que estos se integren en el historial del proyecto. –David Ortega—

Debido a su capacidad para gestionar versiones de manera eficiente y flexible, Git es una herramienta fundamental en el desarrollo de software moderno. El área de planificación, el HEAD, las ramas y los commits son componentes clave de Git, cada uno de los cuales desempeña un papel importante en la organización y control del código. El área de preparación sirve como puente entre el repositorio y los cambios locales, lo que permite revisar y preparar las modificaciones antes de ser confirmadas. El concepto HEAD facilita el intercambio de versiones, mientras que las ramas permiten el desarrollo simultáneo de múltiples características sin obstáculos. Sin embargo, los commits son capturas de estos cambios que permiten un seguimiento detallado del progreso del proyecto. Estos componentes se convierten en Git en conjunto. –-Luis Ramírez—

Webgrafía

- Danielnavarroymas.com. (2019, mayo). https://www.danielnavarroymas.com/gestion-de-proyectos-web-con-git/. Obtenido de Danielnavarroymas.com

- Universidad de Córdoba. (noviembre, 2015). https://www.uco.es/aulasoftwarelibre/wp-content/uploads/2015/11/git-cosfera-dia-1.pdf. Obtenido de Uco.es

- Libro Oficial de Git. (abril, 2005). https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Fundamentos-de-Git

- ¿Qué son los repositorios? - Explicación sobre los repositorios - AWS. (s. f.). Amazon Web Services, Inc. https://aws.amazon.com/es/what-is/repo/

- Acerca de los repositorios - Documentación de GitHub. (s. f.). GitHub Docs. https://docs.github.com/es/repositories/creating-and-managing-repositories/about-repositories

- Git. (s. f.). Desarrollo Web. https://desarrolloweb.com/home/git

- Git - Fundamentos de Git. (s. f.-b). https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Fundamentos-de-Git#:~:text=El%20directorio%20de%20trabajo%20es,los%20puedas%20usar%20o%20modificar

- Moisset, D. (s. f.). Estados de los archivos cuando trabajamos con Git. https://www.tutorialesprogramacionya.com/herramientas/gitya/detalleconcepto.php?punto=5&codigo=15&inicio=0

- KeepCoding, R. (2023, marzo 4). ¿Qué es Staging Area y para qué sirve? KeepCoding Bootcamps. https://keepcoding.io/blog/que-es-staging-area-y-para-que-sirve/

- Git - Acerca del control de versiones. (s. f.). https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Acerca-del-Control-de-Versiones

- Alarcón, J. M. (s. f.). Git: los conceptos de «main», «origin» y «HEAD» - campusMVP.es. campusMVP.es. https://www.campusmvp.es/recursos/post/git-los-conceptos-de-master-origin-y-head.aspx

- KeepCoding, R. (2023b, noviembre 29). ¿Qué es un Commit en Git? | KeepCoding Bootcamps. KeepCoding Bootcamps. https://keepcoding.io/blog/que-es-un-commit-en-git/