[**https://trailhead.salesforce.com/es-MX/users/strailhead/trailmixes/developer**](https://trailhead.salesforce.com/es-MX/users/strailhead/trailmixes/developer)

**GIT HUB DESARROLLO COLABORATIVO**

**Profe:** [**rapa.matias.e@gmail.com**](mailto:rapa.matias.e@gmail.com)

Git del profe

[https://github.com/rapaMatiase/CursoDeGitEIT/wiki/Presentaci%C3%B3n](https://github.com/rapaMatiase/CursoDeGitEIT/wiki/Presentación)

**CLASE 1**

**Git es un software de control de versiones. Su objetivo es llevar un registro de cambios de archivos locales y coordinar el trabajo entre varias personas. Nos permite gestsionar cada una de las etapas por las que va transitando el desarrollo de un proyecto.**

**Desarrollo colaborativo: Es un modelo de desarrollo de software cuyas bases son la disponibilidad publica del codigo y la comunicación via internet.**

**Repositorio:** Un repositorio es la carpeta **.git/** dentro de un proyecto y se encarga de rastrear todos los cambior realizados durante la vida del proyecto.

**Repositorios Locales:** Gestionan el control de verisiones localmente.

**Repositorios Distribuidos:** Gestionan el control de versiones almacenando datos solamente en un servidor.

**Repositorios Centralizados:** El control de versiones pude tenerse en cada dispositivo local, almacenando a la vez los cambios en un servidor.

**ESTADOS DE UN REPO:**

**1- Commited**: Los datos son almacenados de forma segura en la base de datos local.

**2-** **Modifies:** El archivo a cambiado pero los cambios no han sido confirmados en la base de datos local.

**3- Staged:** Se ha marcado un archivo modificado en su version actual, para pasar a su snapshot de comfirmacion.

Esto nos lleva a las **3** secciones principales de un proyecto git:

- el directorio .git/

- el working directory

- el Stage area

**COMANDOS GENERALES**

**git diff** Muestra los cambios realizados en el stage area

**git diff <commit\_1> <commit2>** Muestra las diferencias entre los cambios de dos commits.

**Git diff HEAD~1 HEAD** Muestra las diferencias entre el commit anterior a HEAD y el HEAD

**git diff <commit>** Si especifico un solo commit compara contra el HEAD

**GIT CONFIG / GIT INIT**

**git config –global user.name <valor>** configurar usuario

**git config –global user.email <valor>** configurar mail

**git config –list** mostrar configuraciones

**git config user.mail**  Consulta 1 sola variable

**git config –global core.editor** **<valor>** Configura el editor de texto

<Path del editor>

**git config –global init.defaulBranch <name>** Cambia el nombre de la rama x defecto

**git init** inicia un seguimiento de todo lo que contenga el directorio donde le damos inicio.

**git status** muestra el estado de archivos del directorio controlado

**git config –global commit.template ~/.mensajeCommit.txt** Crea un mensaje de commit predeterminado en un plantilla para añadir automaticamente a cada commit que se haga.

**GIT ADD**

**git add .** añade todos los archivos al stage area, sin estan sin seguir los añade y si tuvieron modificaciones también.

**git add <archivo>** añade un archivo especifico y/o (prepara archivos modificados para el commit, los pasa al stage )

**git add -u <valor>** añade solo los archivos modificados al stage.

**git add <regex>** añade los archivos modificados y los untracked al stage que cumplan la expresion reg

**git add -A ( --all)** añade todos los archivos (tracked y untracked) al stage.

**GIT COMMIT**

**git commit -m “message”** Confirma y añade todos los cambios y archivos del Stage al repositorio.

**git commit -a -m “msg”** Igual al commit -m pero añade todos los untracked y modified que esten o NO en el Stage. (comando simplificado)

**git commit --amend** Permite editar el ultimo commit con cambios al stage area y

tambien modificar el mensaje del commit en VIM

**GIT LOG**

**git log** Muestra todos los commits realizados en la rama con detalle completo ordenados de ultimo a primero.

**git log -<N>** Muestral los primeros N (numero) de commits de la rama con detalle completo.

**git log --oneline** Muestra todos los commist en detalle reducido.

**Git log –all** Muestra los commits de todas las ramas

**git --graph** Muestra todos los commist en grafico ASCII.

**GIT GREP**

**git grep <expresion>** Busca el texto coincidente en todo el repositorio.

**git grep <expresion> <archivo>** Busca el texto en un archivo especifico.

**Git grep -n <expresion>** Busca y muestra la linea en que se encuentra la coincidencia.

**Git log -S <expresion>** Nos muestra un log de todos los commits donde se utilizo el criterio de busqueda.

Grep permite varios parametros de busqueda muy muy utiles, y ademas se puede combinar la expresion de busqueda con expresiones regulares para mejorar las coincidencias.

**GIT INGNORE**

Sirve para excluir del seguimiento de git archivos con informacion sensible.

Se crea un archivo **.gitignore** dentro del repositorio y todo lo que incluyamos dentro ( uno por fila ) sera ignorado por git. Pueden ser archivos o carpetas. Para evitar incluir archivos que cumplan la condicion del ignore pero si queremos incluirlos en el repo lo que podemos hacer es especificar mejor la ruta de los archivos incluidos en el git ignore.

Ej:

**/archivo** Toma los archivos que cuelgan de la raiz del proyecto

Para ver que tipos de archivos suelen incluirse en git ignore, **visitar github/gitignore**

**CLASE 2**

**CLIENTES DE GIT**

**GITHUB**

Es un host de almacenamiento de repositorios Git.

<https://git-scm.com/book/es/v2/Fundamentos-de-Git-Trabajar-con-Remotos>

<https://www.atlassian.com/es/git/tutorials/syncing/git-fetch>

**GIT REMOTE**

**git remote add origin <url\_repo>** Añade el repositorio origin (remoto) al local.

**Git remote -v** Lista las url de conexión

**git push origin <branch>** Pushea una rama particular al origin

**git push –all** Pushea todo al origin

**GIT REMOTO crear repo local y subirlo a remoto.**

1- Desde la consola en la carpeta que querramos **git inti <nombre\_repo>**

2- Con algun archivo ya creado, **git branch -M main**

3- Realizar algún commit inicial.

4- Crear en GitHub un repositorio vacío.

5- **git remote add origin <url\_del\_repo\_github>** agregamos nuestro repo al origin.

6- **git push –set-upstream origin main.** Mediante un push conecto el repo local con el remoto ( por única vez )

**GIT REMOTO añadir repo remoto.**

5- **git remote add <repo\_remoto> <url\_del\_repo\_github>** agregamos nuestro repo remoto al url.

6- **git push –set-upstream <repo\_remoto> main**. Mediante un push conecto el repo remoto con el otro remoto ( por única vez )

**GIT CLONE**

**git clone <url>** Traemos cada version de cada archivo del historial del proyecto

Crea un directorio con nombre igual al del cliente e inicializa un archivo .git en el mismo. Trae toda la información del mistmo y nos muestra la última version. Cuando el repo es clonado, git crea punteros para el seguimiento de cada branch y crea una rama inicial posicionandonos en ella.

**git clone <url> <nuevo\_nombre\_del\_directorio>**

Clona el repo pero en el nombre de directorio especificado.

Tener en cuenta que cuando creamos un repositorio clonado este **automaticamente** se **asocia al remoto**, generalmente con el nombre orgin salvo que lo especifiquemos.

**GIT FORK**

Un fork se utiliza para crear un clone de un repositorio publico en nuestra cuenta. De esta manera, podemos hacer cualquier cambio en el proyecto, y contribur al desarrollo cuando no tenemos permisos de escritura ( **push acces** ). Esto se hace mediante las “**Pull Request“**

Para crear un **fork** vamos a la pagina original del proyecto y en el margen superior derecho del sitio damos click en el **boton “FORK”.**

Trabajando con Fork:

1- Hacemos Fork de un proyecto

2- Creamos un nuevo **branch** desde el master.

3- Hacemos algunos **commits** para mejorar el proyecto.

4- Abrimos un **pull request** en Github.

5- Generamos una discusipon de nuestro trabajo realizado y alternativamente seguimos haciendo mas commits.

6- Sincromizamos el branch master nuevamente a nuestro Fork

**GIT FETCH**

Cuando clonamos un repo git automaticamente crea un origin “remoto” en nuestro repo local. Cuando otros desarrolladores hacen cambios al proyecto remotos y queremos traer los cambios que otra persona haya hecho y no tengamos en nuestro repo local usamos:

**git fetch <alias\_remoto (origin)>**

Esto nos trae todas las referencias a objetos commit que no tengamos en nuestro repo local, para analizarlos y/o unirlos a nuestro trabajo actual. Si trabajamos con un repositorio clonado simplemente corremos **git fetch origin** ya que el clonado nos crea automaticamente un remoto local con el alias origin.

**Importante** notar que git fetch solo trae el trabajo nuevo a nuestro repo locar pero no lo une, esto hay que integrar ambas **ramas** manualmente con merge.

**GIT PULL**

**git pull** Trae los cambios que haya en el repo remoto y los fusiona.

**Git pull origin master** Actualiza desde otrigin (remoto) solo la rama master.

**Git pull –rebase** Cuando hacemos una fusion a tres bandas, el pull con rebase trata de ordenar los commits temporalmente para unirlos en el ultimo commit sin tener que hacer estrategia recursiva.

**PULL REQUEST: SUGERENCIAS DE CAMBIOS.**

1- **Generamos** **Fork** de un proyecto

2- **Clonamos** nuestro fork **localmente**.

3- Creamos una **rama** de trabajo **nueva**.

4- Realizamos **cambios** en el codigo.

5- **Verificamos** y **testeamos** los cambios..

6- **Confirmamos** los **cambios** con un **commit**.

7- Realizamos un **push** de dicha **rama** al **Fork** que tenemos en **Github**

8- En GitHub nos va a aparecer la nueva Rama con un boton **Verde** de **Pull** **Request**

9- Luego colocar un **buen** **titulo** y **descripcion** (para facilitar al dueño del proyecto)

10- Click sobre otro botón verde **“Create Pull Request”** ( El dueño del proyecto va a recibir una **notificación** de que alguien **sugiere** **cambio** y va a poder ver toda la info de cambios propuestos. ) El dueño del proyecto puede integrar los cambios, rechazarlos o realizar **comentarios** sobre cada linea en la vista de diferencias. )

Una vez comentado algún cambio, la persona que creo el pull reques sera notificada.

11- **Luego** de las **notificaciones** podemos realizar mas commits en el branch en cuestión y **volver** a hacer un **push**, y el **pull request** será **automáticamente** **actualizado** por Github.

Agregar commits a un pull request existente no genera notificaciones por lo que deberiamos dejar comentarios para informar al dueño del proyecto.

**GitHub verifica** que el pull request pueda ser integrado de manera **limpia** y sin **problemas**, y nos va a mostrar un boton de atajo para realizar esta operación.

**FLUJO DE TRABAJO: RAMAS**

Debemos recordar que **git no almacena sus datos en forma incremental** (cambios). Los alacena como copias exactas (snapshots).

En cada commit git almacena un punto de control que contiene:

- Un apuntador al staged area con una copia fiel de los contenidos

- Unos metadatos con el autor y mensaje explicativo.

- Y uno o varios apuntadores a los commits que sean que sean padres directos de estas.

**UN BRANCH** es un apuntador móvil apuntando a una de estas confirmacione. En el primer commit se creara la rama **main** y git apuntara a esta confirmacion. En cada commit la rama irá avanzando automaticamente para apuntar al último commit realizado.

**Cada rama se creará a partir del ultimo commit de la rama en la que estemos parados, con sus archivos y su contenido.**

**Git branch** Muestra las ramas del repo local

**git branch <nueva\_rama>** Crea una nueva rama

**gtit branch -r -l**  lista las ramas remotas

**git branch -a** lista todas las ramas, locales y remotas

**git branch -m <nombre\_viejo> <nombre\_nuevo>** Renombrar rama existente.

Este comando **crea** una nueva rama, con un apuntador móvil para que lo puedas manejar libremente. Cuando se crea una nueva rama, esta sigue apuntando al último commit realizado. Mediante el archivo HEAD git identifica en que rama estas parado actuamente.

**git checkout <rama>**  Me muevo entre ramas

**git switch <rama>** Me muevo entre ramas ( nuevo comando )

**git checkout -b <nueva\_rama>** (creo la rama y me muevo a ella autmaticamente.)

**git switch -c <nueva\_rama>** Igual a checkout -b ( nuevo comando )

**git switch –orphan** Crea una rama en blanco. Sin Commits ni nada. Se suelen usar para documentacion. ( nuevo comando )

**git switch –discard-changes <rama>** Cambia de Rama y descarta los cambios del working directory. ( nuevo comando )

Este comando **mueve** el apuntador head a la rama que requieras. Cuando hacemos un commit en la nueva rama, el head apuntara al ultimo commit, pero la rama main, u otra rama que no sea la actual, se mantendra apuntando al ultimo commit realizado en ella.

**Las nuevas ramas se suelen utilizar para realizar modificaciones al proyecto, dejando la rama main para el codigo base, estable y testeado.**

**Git branch -d <rama>** ( elimino la rama ) Forzar la eliminación con **D**.

**ESTRATEGIA DE UNIFICACION: INTEGRANDO CAMBIOS.**

Estrategias para unificar el trabajo de ramas ( modificado, testeado y comprobado ) al proyecto general. Para esto contamos con varios metodos o estrategias.

**MERGE**

**git merge <branch>** Situados en la rama de destino hacemos merge refiriendonos a la rama con los cambios a traer.

**Git merge - - abort** En el estado de resolucion de conflictos del merge podemos abortar la fusion en curso.

**Merge fast-forward**

Si hacemos merge de una rama fusionando un commit con otro commit que sigue el historial de la primera ( padre directo en el main ) Git avanza el puntero hacia adelante y hace la fusion **sin conflictos** ( **fast forward** ), como si siguieramos avanzando en la rama **main.** Git avanza el Head hacia el commit de la rama con cambios y ubica el main/master en ese commit.

**Merge no lineal:**

Si el **branch** a fusionar con main **no es lineal**, o sea no es ancestro directo de nuestra rama main si no que su ancestro común se ha bifurcado en un commit anterior, git realizará una **fusión a tres bandas ( recursiva )**, utilizando el ultimo commit de cada branch y el ancestro en común (commit donde se bifurca el proyecto). En lugar de avanzar el apuntador de la rama git crea un nuevo commit “fusion confirmada” al cual queda apuntando y se situa en la rama main.

**CONFLICTOS DE MERGE.**

Si hay **modificaciones** en el **mismo sector del mismo archivo** de **dos ramas distintas a fusionar**, git no podra resolverlo y enviará un mensaje de **auto-merge fallido y merge conflict**. Pondrá en **pausa** el proceso y esperará a que resolvamos los conflictos **manualmente**.

Podemos acudir a **git status** para ver los archivos sin fusionar.

Los archivos se marcaran como **unmerged** y git añadira unos marcadores especiales >>>>>> y <<<<<< para guiarnos cuando los editemos manualmente.

Una vez **solucionados**  los conflictos, usamos git add para añadir el archivo conflictivos al stage area y asi indicarle a git que los conflictos han sido resueltos. Luego commit.

**MERGEANDO CON BUENAS PRACTICAS**

EJ: Merge de rama secundaria a la principal:

**Mala practica**

- git checkout main

- git merge second

- gti log -2 ( para chequear los cambios )

**Buena practica**

- git checkout second

- git merge main

- git status ( confirmar si se creo el nuevo commit en la rama secundaria )

- git checkout main

- git checkout second

**DESACIENDO CAMBIOS Y CORRIGIENDO ERRORES**

**git checkout - - <archivo.xxx>** Estando en el **working area** revertimos los ultimos cambios hechos en el archivo.

**Git restore <archivo>** Deshace los cambios del archivo igual que checkout – archivo. ( Comando nuevo )

**git restore .** Deshace todos los cambios hechos. ( Comando nuevo )

**git restore - - staged <archivo>** Restore desde el **Stage** area ( Comando nuevo )

**git restore –staged –worktree <archivo>** Deshace los cambios del Stage y del Working area en un solo paso.( Comando nuevo )

**git restore –source=<tag> <archivo>** Trae el archivo al working tree desde un lugar especifico.

**DESACIENDO CAMBIOS – RESET**

**Git reset HEAD <archivo>** Saca el archivo <archivo> del stage area y lo retrae al working area.

**Git reset <hash>** Lleva los cambios al estado del commit indicado. Saca los archivos del stage area y deja las diferencias del working area.

**Git reset - -soft HEAD~** Lleva el HEAD al commit seleccionado pero deja los cambios introducidos en el commit del cual venimos en el stage area, como si recien se hubiesen aplicado. Estos cambios tambien van a poder ser visibles en los archivos en los cuales fueron hechos.

**Git reset - -mixed HEAD~** Igual que el reset soft pero los cambios en vez de permaneces en el stage area pasan al working area. Los cambios se mantienen en los archovs pero es como si no hubiesemos hecho git add.

**Git reset - -hard HEAD~** Lleva los cambios a ese commit pero a la vez elimina todos los cambios ( stage y working area ), y los cambios realizados en los archivos desaparecen. Limpia el status.

**\* LOS COMANDOS RESET PUEDEN DEJAR DESPAREJOS LOS COMMITS CUANDO SE TRABAJA EN REMOTO !**

**DESACIENDO CAMBIOS – REVERT ( commits )**

**Git revert <hash>**  Este comando selecciona un commit y envía el estado previo a este commit seleccionado al final del branch, **creando un nuevo commi ( con el estado previo al commit seleccionado)**

**Git revert –no-commit HEAD** Revierte el cambio en HEAD **==>**

**git revert –no-commit HEAD~1** Revierte el cambio anterior a HEAD **==>**

y así sucesivamente hasta que querramos, y para finalizar,

**git revert –continue** Esto finaliza el proceso de varios revert dejando un solo commit de revert al final en vez de varios.

**GIT ALIAS**

**git config –global alias.nombre ‘Comando completo’**

Crea un alias de comandos. Importante, los alias pueden seguir recibiendo parametros de comandos detrás.

**Git config –global –get-regex alias** Muestra los alias configurados.

**Git config –global –unset alias.nombre** Elimina el alias corresp. Al nombre

**GIT TAGS – Puntos de versionado.**

Son etiquetas que utilizamos para crear **Alias** a los **commits,** **basicamente son referencias al hash de un com**mit. Se suelen utilizar para marcar versiones, commits especiales o cambios importantes en el proyeco. Por intermedio de estos TAGS me puede referir al commit en cualquier momento.

**Git tag <nombre>** Crea el tag soblre el commit del HEAD

**Git tag <nombre> nro\_comit** Crea un tag sobre un commit especifico

Con checkout y el mobre del TAG me puede mover a ese commit anterior y por ejemplo, crear una rama nueva.

**Git tag** Lista todos los tag

**Git tag -d <tag>** Elimina el tag seleccionado

**git tag -a <nombre\_tag>** Crea un tag con informacion para introducir con Vim

**git show <nombre\_tag>** Muestra toda la info de un tag ( show sirve para muchos comandos)

TAG ANOTADOS

Los tags son etiquetas simples, cuya información es un número ( puntero ) que apunta a un commit especifico. Con tags ANOTADOS podemos añadir mucha mas informacion a un tag que solo un puntero de referencia. Ej versiones, autor, descripcion, fimra de autenticidad, fecha y hora. Etc. Esto, a diferencia del tag simple, crea un objeto en la base de git y pueden ser subidos al repositorio.

**git tag -a <nombre> nr\_commit o head** Crea un tag con mensaje de TAG

**git tag -a <nombre> <hash>** Crea tag y mensaje directamente

**git show <nombre\_tag>** Muestra el tag anotado con detalle de la info.

Los tags de git **no se suben de manera automatica** como los commits cuando hacemos push. Para subirlos

**git push origin <nombre del tag>** Suber el tag al remoto

**git push origin –tags** Sube todos los tags juntos.

**GIT STASH**

Cuando queremos cambiar de rama y tenenmos añadidos archivos al stage area y cambios sin confirmar en el working directory, pero no podemos realizar el commit, usamos el comando **stash** para guardar los cambios sin comitear.

**git stash** Guarda los cambios hechos sin commitear en el stage area

**git stash save “<mensaje>”** Guarda el stash con un mensaje

**git stash list** Muestra los stash guardados.

**Git stash apply** Recupera lo guardado en stash al working area

**git stash apply <stash@{n}>** Carga los cambios de un stash especifico.

**Git stash drop** Quita el ultimo stash de la pila.

**Git stash clear** Limpia el stash

**git stash branch <nueva\_rama> <stash\_id>** Crea una nueva rama con el stash indicado por el id.

Los **stash** se comportan como una **pila**. Si uso el comando apply ira aplicando los cambios desde el ultimo al primero. Apply devuelve es stash guardado al working area y/o al stage area, pero luego de add y commit, el stash ya no se puede aplicar. Sin embargo queda registrado en la pila de stashes. Los apply del stash se pueden aplicar en cualquier branch.

**Git stash pop**  Realiza un apply y un drop al mismo tiempo.

**GIT REBASE**

Git **rebase** nos permite permite llevarnos los cambios confirmados de un branch y aplicarlos sobre otra rama, en vez de fusionar a tres bandas como merge, rebase rebovina hasta encontrar un ancestro común, y trata de aplicar los cambiors en orden cronologico para integrarlos en el ultimmo commit realizado en la rama en que se aplicartan los cambios.

Aplicacion: desde la rama de la cual queremos fuisionar los cambios, indicamos a que rama hacer el rebase.

Ej: **git cheout secondary\_branch**

**git rebase master** (Esto fusiona secondary desp del ultimocommit en master)

Cuando se ejecuta rebase git va al ancestro comun de la rama en que estas y a la que quieres reorganizar, saca las diferencias aplicadas en cada commit de la rama en que estas, reincia la rama actual y la lleva a la misma confirmacion en la rama que quieres fusionar, y luego aplica los cambios ordenadamente.

\* Nunca reorganizar commits y ramas que ya se hayan pusheado.

\* Rebase es util para emprolijar el historia, pero esa es la contra, se pierde el historial

**Rebase interactivo**

El rebase interactivo, rebase -i, nos permite manipular el historial de commits.

- Cambira el orden de los commits

- Cambiar el mensaje de un commit previo.

- Aplastar commits en un unico commit.

**Git rebase -i HEAD~4**  Ej: Toma los ultimos 4 commits del Head en que estemos y abre un editor VIM para editar los cambios. Un vez realizados los cambios guardamos en Vim :wq, y git automaticamente ejecuta el rebase. Tener en cuenta que esto cambia los hashes de los commits, OJO !

Agunos comandos de rebase interactivo:

**- p, pick**  Mover commits de lugar

**- r, reword** Reescribir el mensaje del commit en modo editor

**- e, edit** nos permite pausar el rebase en ese commit en particular.

**- s, squash**  Nos permite fusionar ese commit particular con el commit anterior.

**- f, fixup** Igual a squash pero descarta todos los mensajes de los commits a fusionar

Esto se aplica cambiando los prefijos de los commits a modificar y luego saliendo de VIM. Cuando salimos de VIM gurdantdo los cambios, rebase va a continuar aplicando los cambios uno x uno y revisando que no haya conflictos, en cuyo caso va a pausarse para poder corregir el problema.

**GIT CHERRY-PICK**

Git **cherry-pick** nos permite permite llevarnos los cambios de un commit previo de cualquier parte del historial del repo, a diferencia de merge, rebase o pull que traen todos los commits existentes, cherry-pick solo trae uno especifico y crea una nueva version de ese commit (nuevo commit con distinto hash).

**Git cherry-pick <hash>** Siempre en la rama en la que lo queremos aplicar.

Ojo al utilizar cherry pick, porque puede provocar desfazajes temporales en los commits, suele usarse frecuentemente para **hot fixes** donde queremos traeer un commit puntual, generalmente con algun fix, pero no abusar.

**DEPURACION DE REPOSITORIOS - GIT BISECT**

El comando bisect, no permitira realizar tecnicas de **depuracion** o **debbuging** de nuestro codigo utiilzando la búsqueda binaria.

Al iniciar esta busqueda con bisect, nos permitira ir probando codigo de testeo en ciertos commits, e ira reduciendo automaticamente el rango de busqueda mediante los parametros “good o bad” que le vayamos dando.

Pasos a seguir para aplicar bisect:

1- **git bisect start** Entra en modo BISECT

2- **git bisect bad** Indicamos un commit en el cual el programa no funciona. generalmente es el commit actual, o sea el ultimo. Solo con ejecutar git bisect bad, git marca ese commit como malo.

3.A- **git bisect good** Para esto tenemos que buscar un commit donde el programa funcione bien. Tenemos 2 opciones, nos movemos a un commit ( git switch <hash> ) que creemos que funciona y ejeuctamos el testeo, si es malo indicaremos “git bisect bad” y repetiremos con otro commit. Si el resultado del testeo es bueno, indicamos “git bisect good”. Entonces git ya tendra los 2 parametros good y bad. Y automaticamente nos cambiara el HEAD a otro commit donde volvemos a ejecutar el testeo. Entonces devolveremos el parametro “git bisect bad / good” depende como salga. Y nuevamente nos cambiara a otro commit para realizar el mismo proceso hasta que git encuente el commit con el bug.

3.B- **( opcional ) git bisect good <hash>** Si conocemos el commit donde el programa funcionaba correctamente, le damos el ok de de “good” de manera directa con el comando seguido del hash del commit que funcionaba.

4- **git bisect reset** Git sale del modo depuracion BISECT

**DEPURACION DE REPOSITORIOS – GIT BLAME – File anotation**

**git blame** Nos permite saber el ultimo commit en el cual fue introducida una linea especifica de un archivo.

**Git blame -L 10,20 <archivo>** Esto nos muesta: **commit, autor, y fecha** de las lineas especificadas en el archivo especificado. El parametro - L 10,20 indica desde cual hasta cual linea analizar..

**GIT REFLOG**

El **log de referencia** o **reglog**, es el lugar donde quedan almacenadas las puntas de branch y otras referencias del repo local, y cubre todas las acciones recientes. Switch, commit, reset, etc..

A medida que avanzamos git guarda cada movimiento realizado por el HEAD y actualiza el reflog en cada accion que realizamos.

**Git reflog** Muestra todas las acciones y moviemientos del head HEAD@{n}

Luego con git checkout nos podemos mover al HEAD@{} que querramos, y entraremos en un **detached HEAD,** o sea estaremos en un momento historico de nuestro HEAD, el cual puede extistir o no en nuestro arbol de trabajo.

Entonces podemos crear un **nuevo branch**, y automaticamente el nuevo branch contiene el trabajo que tiene dentro el HEAD sobre el cual estamos parados.

**EXPIRACION DE REFLOG**

Por defecto expira en **30 dias para unreachable** y **90 dias para casos normales**.

Para reconfigurar su expiracion:

**> git config gc.reflogExpire  
> git config gc.reflogExpireUnreachable**

**FLUJOS DE TRABAJO.**

**LONG-RUNNING BRANCH**

En esta metodologia de trabajo, se suele tener solamente codigo estable y testeado en la rama main, generalmente codigo ya lanzado o a lanzarse. A la vez se trabaja con una rama paralela ej: **desarrollo**, que generalmente se usa para trabajar o probar la estabilidad del codigo. A esta rama paralela, a medida que logra un nivel de estabilidad, se le van agregando (pull) otras ramas (ya testeadas), generalmente de corta duracion, y luego, una vez alcanzado nuevamente un nivel de estabilidad se fusionan a main. Basicamente el head se mueve hacia adelante a medida que las ramas avanzan y el codigo mas estable va quedando atrás del codigo nuevo. Estas ramas se pueden bifurcar cuantas veces consideremos necesario.

**TOPIC BRANCH**

En este contexto de workflow, se van creando (topic) ramas independientes de corta duracion creadas para un proposito muy particular, pueden ser tantas como se vayan necesitando, y un vez que la rama tópico se resuelve o finaliza, puede mergearse con main o guardarse para ser aplicada mas tarde, no importa cuando, pues su proposito es tan particular que no deberia interferir con el proyecto general.

**DICTADOR TENIENTE**

Se utiliza en grandes proyectos con cientos de colaboradores.

En este tipo de workflow, el dictador hace push del proyecto al “**blessed repository”**, los desarrolladores o **colaboradores publicos** trabajan en su rama tematica y hacen **pull del blessed repository y luego un rebase** con las implementaciones que desarrollan.

Los **tenientes** tambien hacen pull del blessed repository y luego **fusionan** las ramas tematicas de los **colaboradores** en su rama master.

El **dictador** **fusiona** la rama master del **teniente** y luego hace un **push al blessed repository** para que los desarrolladores publicos puedan ajustarse a los cambios nuevamente.

**GIT FLOW**

Es un modelo de **ramificacion predeterminado** y diseñado en torno al lanzamiento de un proyecto. Es ideal para proyectos con un ciclo de lanzamiento predeterminado.

En este tipo de work flow, el historial de trabajo del proyecto se registra en dos ramas.

- 1 **Master**, que solo contiene las versiones de **lanzamiento oficial y los hotfix** (versiones de correcction). Se suelen etiquetar los commits con el nro de version.

- 2 La rama de **desarrollo** que sirve para la integracion de **caracteristicas**.

Cada nuevo **Feature** del proyecto debe residir en su **propio branch**, que se bifurca de la rama desarrollo. Cuando una rama Feature es completada se fusiona nuevamente con desarrollo.

Una vez que **desarrollo alcanza un punto de lanzamiento, se bifurca una rama de release** (lanzamiento) fuera de la rama desarrollo, la cual inicia el siguente ciclo de lanzamiento y no se pueden agregar nuevas caracteristicas al proyecto. Solo se agregan correcctiones de errores y documentación orientada a la version. Una vez finalizada la rama **release** se **fusiona a master** y se etiqueta con el numero de **version**. Tambien es fusionada a la rama desarrollo que puede haber avanzado hacia la proxima version del proyecto.

Las ramas de **hotfix** son la **unicas** que se **bifurcan** de la rama **master** y se usan para hacer correcciones rapidas de errores graves y se vuelven a fusionar a master con un nuevo numero de version, y tambien a desarrollo. Esto permite al equipo corregir errores sin interrumpir el desarrollo en curso.