Licenciatura en Sistemas - Orientación a Objetos II – 2021

Git & Github

Git:

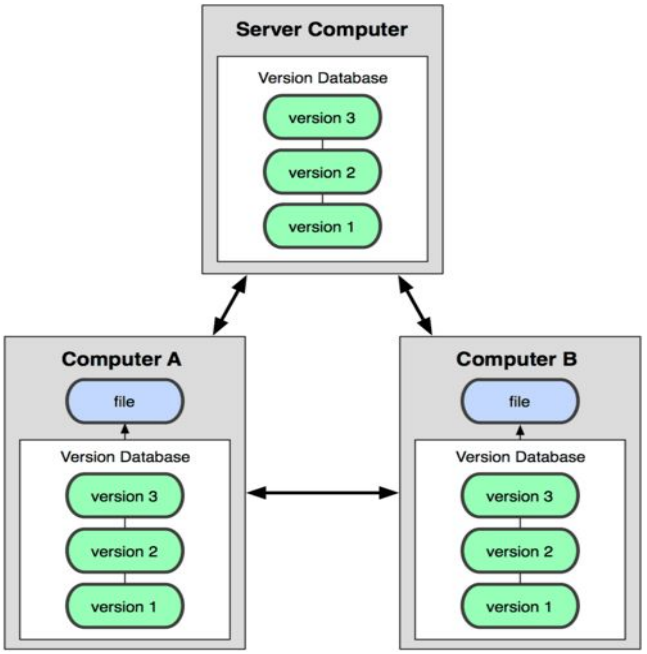
Git es un sistema distribuido de control de versiones, el cual nace cuando se trabaja en el kernel de Linux, el cual claramente era un proyecto de software de código abierto con un gran alcance. En ese entonces para el proyecto se usó inicialmente un control de versiones distribuido con BitKeeper, pero en un momento la tool deja de ser ofrecida gratuitamente para este desarrollo dejando sin soporte a la comunidad.
Ante este problema y la falta de una herramienta de control de versiones gratuita y de calidad, se llevó adelante por parte de una gran porción la comunidad de Linux, con Linus Torvalds (https://www.ecured.cu/Linus_Torvalds) a la cabeza, el proyecto para crear Git, un nuevo software de este tipo y open source que cumpliera con las siguientes características:

- Velocidad
- Diseño sencillo
- Gran soporte para desarrollo no lineal (miles de ramas paralelas)
- Completamente distribuido
- Capaz de manejar grandes proyectos (como el kernel de Linux) eficientemente (velocidad y tamaño de los datos)

Fuente: "Una breve historia de Git": https://git-scm.com/book/es/v2/Inicio---Sobre-el-Control-de-Versiones-Una-breve-historia-de-Git).

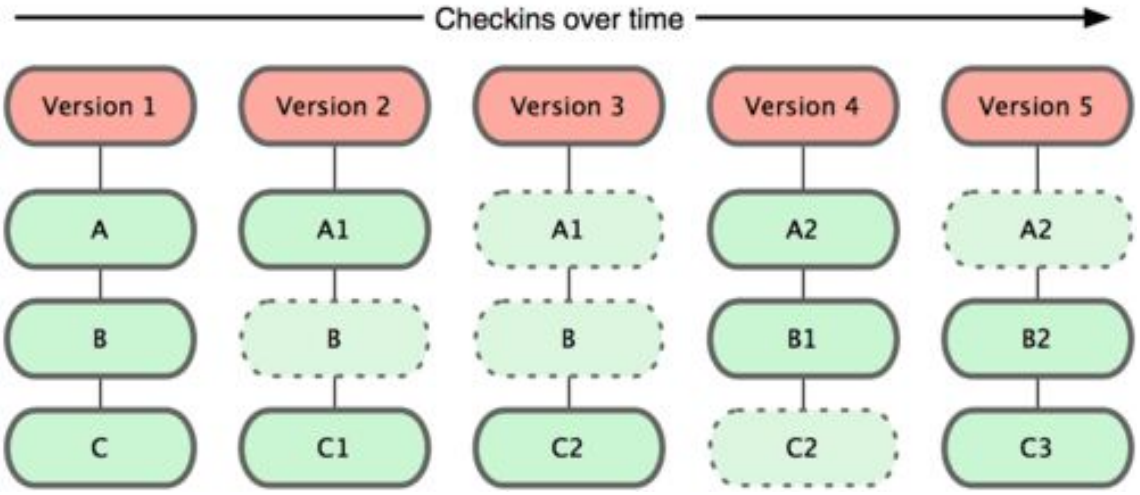
Sistema distribuido de control de versiones:

- Control de versiones
- Histórico del proyecto
- Repositorios
- Organización distribuida



Histórico del proyecto:

Git almacena los datos como instantáneas del proyecto a través del tiempo. Cada vez que se genera una copia nueva del proyecto, la copia es una foto del estado anterior, agregando los cambios nuevos. Para poder ser eficaz, Git no guarda en si un archivo nuevo de uno cuando este no se modificó, sino que guarda una referencia a la última copia estable.



En la imagen podemos ver que las versiones 2 y 3 del proyecto tienen la referencia al archivo B, guardando un acceso a la última copia estable. Ya para la versión 4 el contenido del archivo B cambia y se guarda con las modificaciones y con una bandera que indica que es una nueva versión del archivo (B1 en el caso del ejemplo).

Comandos básicos:

- git init: Si corremos este comando en una carpeta, vamos a estar indicando que dentro de esa carpeta se va a poder trabajar con Git, se debería hacer al nivel de la carpeta raíz del proyecto.
- git status: Indica el estado actual del repositorio local en la pc de trabajo (estado del Workspace).
- git add: Nos permite agregar nuevos archivos a estado de Index.

  corriendo **git add src/ejemplo/Clase1.java**

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   src/ejemplo/Clase1.java

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .classpath
        .project
        .settings/
        bin/
        src/ejemplo/Clase2.java
```

  Se puede usar **git add .** para subir todos los archivos que estén en Workspace a Index.

- git commit: Nos permite agregar nuevos archivos al estado de cambios del Repositorio Local. Los agrega todos juntos en un “paquete”, con una marca de tiempo e identificación de dicho paquete.
  Si corremos **git commit -m "Agregando la 1er clase"**

```
[master (root-commit) 486e8f9] Agregando la 1er clase
 1 file changed, 5 insertions(+)
 create mode 100644 src/ejemplo/Clase1.java
```

- git push: Nos permite subir los commits del Repositorio Local al Repositorio Remoto.
  Si corremos **git push origin master**

```
Counting objects: 5, done.
Writing objects: 100% (5/5), 352 bytes | 0 bytes/s, done.
Total 5 (delta 0), reused 0 (delta 0)
To github.com:gussiciliano/Git.git
 * [new branch]      master -> master
```

De todas formas ¿**quién es el Repositorio Remoto**?
Este repositorio debe existir en la nube en algún proveedor de alojamiento de código, soportando el uso del sistema de control de versiones Git. Cuando este repositorio existe en la nube, debe tener una ruta que habilite el acceso por HTTP o por SSH, permitiendo ejecutar el siguiente comando:

- git remote add origin Ruta_HTTP_o_SSH

Un ejemplo podría ser: **git remote add origin git@github.com:cuenta/proyecto.git** (usando Github)
Esto hace que nuestro Repositorio Local quede vinculado con el Repositorio Remoto, permitiendo que se suban cambios desde nuestra computadora a la nube (o al lugar donde esté alojado el código).

¿**Qué significa la palabra master**?
Dentro de Git se trabaja siempre dentro de una branch (rama). La idea es que cada programador trabaje en una branch relacionada con la funcionalidad que tiene que desarrollar. Esto se hace para que entre colegas no se pisen los cambios de otro desarrollo.
Para revisar las branches que tenemos, crear y realizar diferentes operaciones se usan los siguientes comandos:

- git branch: Para ver las branches de trabajo en el workspace (con **git branch -d nombreBranch** se puede borrar una rama).
- git checkout nombreBranch: Para movernos entre las branches (con **git checkout -b nuevaBrach** se puede crear una branch nueva)

¿**Qué pasa si me sumo al equipo y quiero descargar los cambios de una branch**?

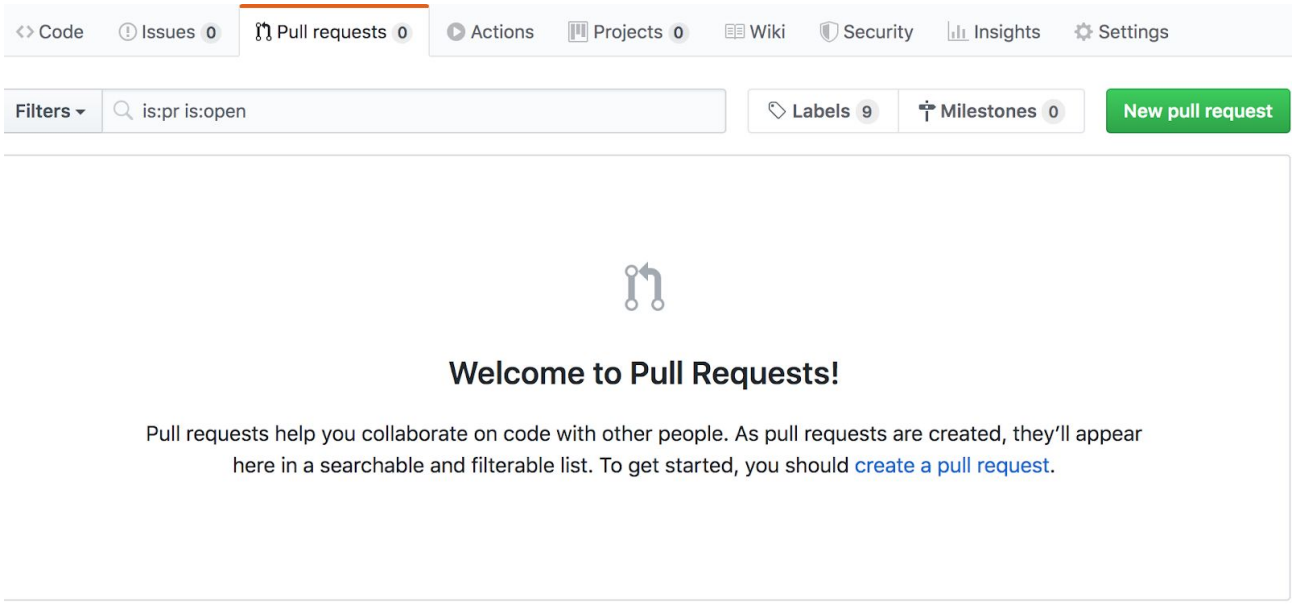
- git pull origin nombreBranch: Con este comando podemos traer los últimos cambios del Repositorio Remoto de la branch que estamos consultando (en este caso la branch nombreBranch).

¿**Qué pasa si me sumo a un equipo que ya viene trabajando?**

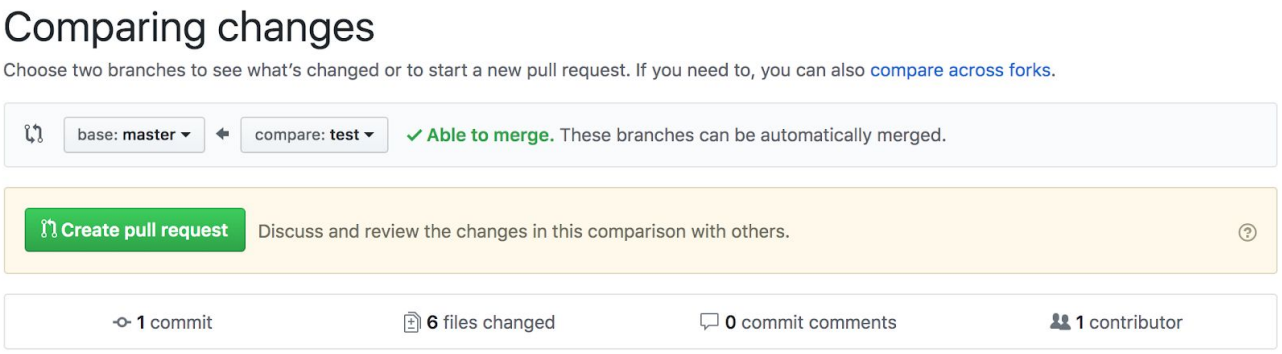
- git clone Ruta_HTTP_o_SSH: con este comando se puede “clonar” un repositorio remoto en el repositorio local.

Github:

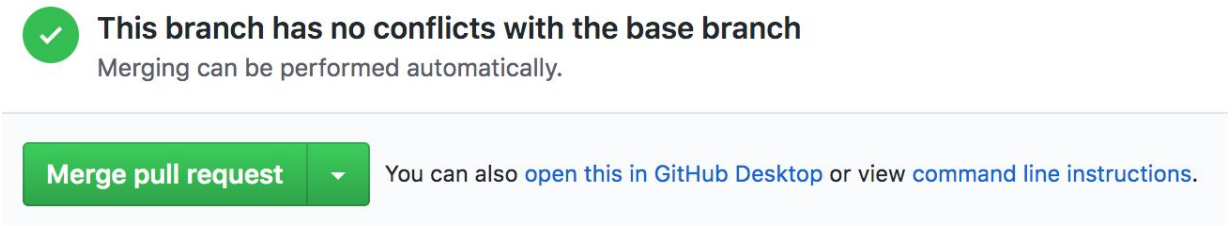
GitHub es una plataforma para alojar proyectos utilizando el sistema de control de versiones Git. Existen muchas otras como Gitlab, Bitbucket, etc.
Entre las cosas que ofrece Github además de alojar el código en la nube, tiene una buena interfaz para el trabajo colaborativo por intermedio de Pull Requests.
Cuando se sube funcionalidad de varios programadores en sus branches de trabajo, eventualmente este código se tiene que “mergear”, para hacer esto de forma segura se puede crear un nuevo Pull Request, solicitando desde que branch viene el código nuevo, a que branch debe ir a parar.



Por ejemplo si vamos a la sección para crear un Pull Request podemos nuestra branch de trabajo es **test** vemos que si queremos subir esos cambios en la branch **master** tenemos 6 archivos que se cambiaron, por intermedio de un commit.

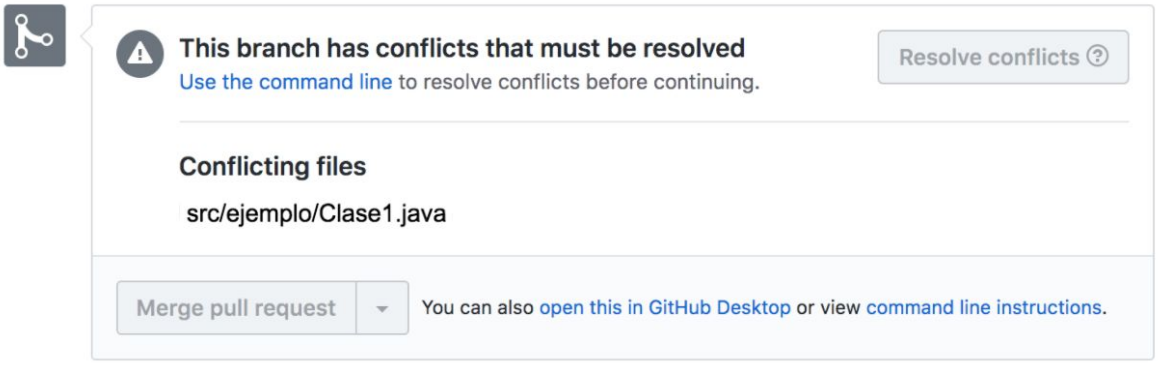


Si creamos el Pull Request vamos a ver lo una sección con los archivos cambiados y vamos a poder “mergear” los cambios en la branch master



una vez subido se puede borrar la branch de trabajo.

¿Qué pasa si un Pull Request tiene conflictos para “mergearse”?
Por ejemplo, desde la branch **master** salen dos branches **alta-usuario** y **baja-usuario**. Ambas branches de trabajo hacen cambios sobre un mismo archivo, la branch alta-usuario hace un Pull Request y se “mergea” a master. En ese caso cuando la baja-usuario se quiera “mergear” a master, va a tener un problema, porque los archivos originales de los cuales partió, cambiaron y Git no sabe cuál es el cambio válido, si el cambio que trata de introducir la alta-usuario o el cambio que trata de introducir la baja-usuario.



En ese caso se deben seguir los siguientes pasos:

1. Ir a la branch a la cual se quiere subir nuestro código, la cual sufrió cambios luego que nosotros partimos de ahí. En el ejemplo, deberíamos ejecutar **git checkout master**.
2. Una vez que estamos en la branch sobre la que queremos mergear nuestro código, es necesario actualizarla, ya que es probable que no contemos con los cambios del Repositorio Remoto. En el ejemplo, deberíamos ejecutar **git pull origin master**.
3. Luego de la actualización volvemos a nuestra branch de trabajo. En el ejemplo, deberíamos ejecutar **git checkout baja-usuario**.
4. Una vez en la branch de trabajo, se debe hacer “un rastreo sobre los commits de diferencia entre ambas ramas”. Esto significa que se va a comprar la branch baja-usuario con la branch master, commit a commit (ósea, paquete de código a paquete de código). Y se va a producir una actualización entre las diferencias. Cada vez que la diferencia sea un conflicto, la consola nos va a avisar para hacer los cambios en los archivos a mano (ya que el/la developer sabe cuál es el correcto porque conoce la lógica de negocio de la aplicación, cosa que es imposible de conocer por git). Cuando se resuelve el conflicto se vuelve a correr la operación de rastreo y comparación de commits, si se cuenta un conflicto se tiene que volver a cambiar y así sucesivamente. Eventualmente este proceso va a terminar. El proceso consta de los siguientes comandos:

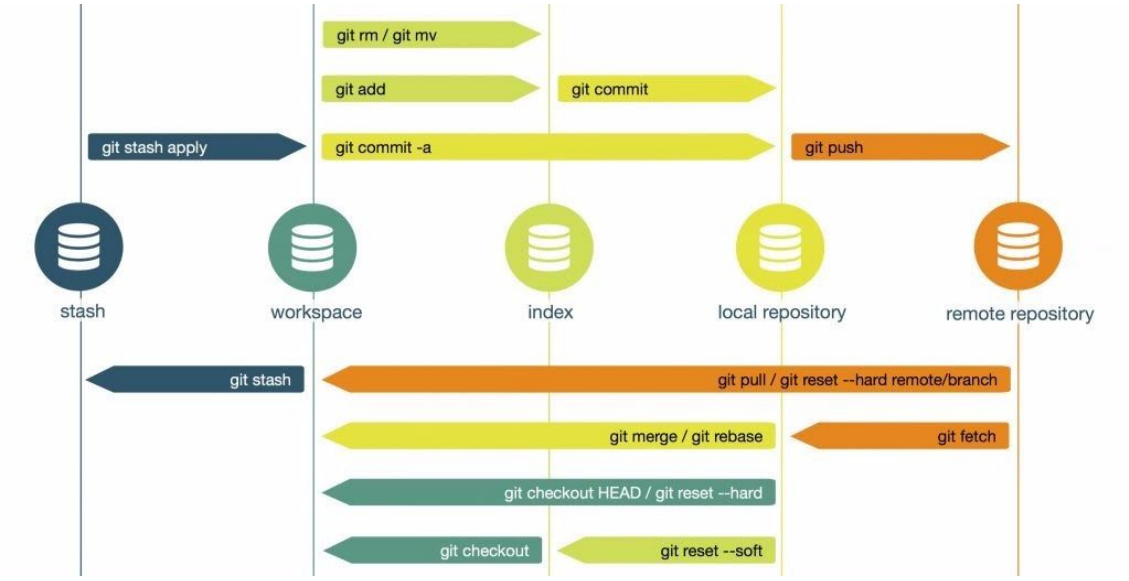
(según el ejemplo)

a. **git rebase master**
b. **git add .** (en caso de tener conflictos y de cambiarlos a mano)
c. **git rebase --continue** (para volver a ejecutar el primer rebase, si hay conflictos nuevamente se corre el punto **b**, seguido del **c**. Vamos a saber que terminamos cuando la consola nos diga **Applying: baja-usuario**)
d. **git push -f origin baja-usuario** (para subir los cambios al repositorio remoto)

Hecho esto, si vamos a Github nuevamente vamos a ver que el Pull Request ya se puede mergear con la habilitación del botón verde de merge.

Nota: Todo este proceso de PR y Merge se puede hacer por comandos también, pero normalmente se usa la interfaz del proveedor del alojamiento del código.

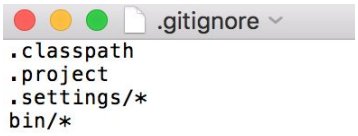
Para cerrar este concepto podemos ver el flujo de archivos entre los diferentes estados de trabajo y los comandos a utilizar para cada movimiento



Nota: Como se puede apreciar, existen más comandos (y de hecho hay muchos comando más), pero esta guía solo hace un repaso por los básicos.

Git Ignore:

Para evitar subir archivos de configuración, credenciales o info del contexto del IDE de desarrollo del developer se usar un archivo llamado .gitignore. Es un archivo oculto que lista cuáles son los archivos y carpetas que Git nunca debe subir al Repositorio Remoto. En nuestro podríamos tener un .gitignore como el siguiente ejemplo:



Branching Model:

Para finalizar es importante que siempre se trabaje con un branching model. Normalmente se trabaja con el siguiente modelo de branches:

1. Branch **master** (branch estable, que tiene entregables finales del producto al cliente).
2. Branch **development** (branch donde se tiene el código de todo el equipo de desarrollo, a medida que se van produciendo entregables internos en el equipo).
3. Branch de trabajo (son las branches de trabajo de cada desarrollador, normalmente tienen el nombre de la funcionalidad a desarrollar, como **alta-usuario**, y se mergear sobre development).
4. Branch **topic-name** o **epic-name** (es opcional, nace desde development, se suele usar para agrupar funcionalidad y no mergear directo en master, por ejemplo un topic o epic puede ser **topic-abm-usuario**, dentro, N developers harían branches de trabajo para hacer esta funcionalidad. Una vez terminada se mergea en développement).
5. Branch **stage**, **qa** o **test** (es una branch con un entregable de una funcionalidad completa que se coloca en esta branch para su testeo y eventual entrega parcial o final al cliente. La idea es que mientras se producen pruebas sobre esta branch, el resto del equipo sigue trabajando sobre development).
6. Branch **hotfix-nombre** (es una branch se usa cuando se va a subir un cambio urgente de código, normalmente se produce cuando se encuentra un error crítico y se tiene que solucionar rápido, sin pasar por el flujo común del branching model y “mergeandose” con prioridad a la branch que corresponda).
7. Branch **release-AAAA-MM-DD** (es un branch que se corresponde con un entregable de código para subir a un ambiente de prueba. La idea es tener este orden para poder seguir un error, en caso de haber, en diferentes entregables que se produzcan sobre el ambiente de prueba).
8. Branch **production** (branch estable con el código a subir a producción, puede tener una branch de release para un mayor control. Si es la entrega final del producto, su contenido se agrega además en master).

Instalación y otras herramientas:

Para instalar Git se puede ir a la página oficial en https://git-scm.com/. Además, actualmente se cuenta con una serie de tools que ayudan al developer a abstraerse de la consola a través de una interfaz. Por ejemploGithub Desktop (https://desktop.github.com/), Source Tree (https://www.sourcetreeapp.com/), Git Extensions (http://gitextensions.github.io/), etc. Además de diferentes plugins que se pueden agregar en los IDEs de desarrollo como Eclipse, Netbeans, etc. Hoy en día sigue siendo recomendable conocer primero el uso con comandos y una vez dominado eso apoyarse en el soporte de una a GUI.