**Diferencias entre GIT y CVS**

* En GIT es más fácil de configurar para commit; en CVS tienes que configurar CVSROOT.
* En GIT los cambios son atómicos, o se realizan todos o no se realiza nada: en CVS si se interrumpe al subir cambios pueden quedar inconsistencias.
* En GIT los cambios siempre se referencian a todo el proyecto, haciendo más fácil revertir cambios; en CVS los cambios son por archivo.
* En GIT se pueden renombrar los archivos, en CVS no hay soporte para ello.
* CVS tiene un soporte muy limitado para archivos binarios, el usuario tiene que marcar un archivo como binario; GIT lo detecta automáticamente.

**Comandos:**

Configuración:

* **git version**
* **git config --global** 
  + **user.name “<nombre de usuario>”**
  + **user.email “<email@servidor.com>”**
  + **http.proxy <http://direccion\_ip\_proxy:puerto> 🡪** En caso de necesitar configurar proxy.
  + **https.proxy <http://direccion\_ip\_proxy:puerto> 🡪** En caso de necesitar configurar proxy.
  + **user.alias.<nombre-del-alias> “<comando que quieres que se guarde>”**
    - Ejemplo:

git config --global alias.superlog "log --graph --abbrev-commit --decorate --date=relative --format=format:'%C(bold blue)%h%C(reset) - %C(bold green)(%ar)%C(reset) %C(white)%s%C(reset) %C(dim white) - %an%C(reset)%C(bold yellow)%d%C(reset)' --all"

* **git help <comando>**

Flujo de trabajo

* **git init** 🡪 Inicia un repositorio.
* **git status** 🡪 Status del repositorio.
* **git add** 
  + **.** 🡪 Agrega todo al stage.
  + **<file or directory>** 🡪 Solo agrega el archivo o directorio especificado.
* **git commit** 
  + **-m “<mensaje>”** 🡪 Se realiza commit con mensaje.
  + **-am “<mensaje>”** 🡪 Se realiza add \* y commit a la vez.
  + **-[a]m “<mensaje>”** **--amend** 🡪 Modifica el último commit.
* **git tag**
  + **-a <versión> -m “<mensaje>”** 🡪 Crea etiqueta con mensaje al último commit.
  + **-l “<version>”** 🡪 Nos indica si existe la versión.
  + **show <version>** 🡪 Nos muestra la información (autor, fecha, diferencias) del commit.
  + **-a <versión> <ID commit> -m “<mensaje>”** 🡪 Etiqueta un commit en específico.
* **git checkout <ID commit o tag>** 🡪 Pone el HEAD en el commit o tag que queramos.
* **git diff <ID commit>** 🡪 Muestra diferencias entre el commit donde estamos y el commit que especificamos.
* **git log** 🡪 Muestra el historial de commit realizados.
  + **--oneline** 🡪 Solo muestra el código de commit y la descripción.
  + **--oneline --graph** 🡪 Muestra el árbol de commits, útil en fusiones.
  + **--decorate** 🡪 Muestra en que commit se encuentra el HEAD.
  + **--stat** 🡪 Muestra el número de cambios que hubo en cada commit.
  + **-p** 🡪 Muestra todos los cambios de cada commit.
  + **shortlog** 🡪 Que commits ha realizado cada usuario con todo y los títulos de cada commit.
  + **pretty=format:"%cn hizo un commit %h el dia %cd"** 🡪 Muestra mensajes perzonalizados de los commits.
  + **-3**🡪 Limitamos el número de commits.
  + **--after="2016-1-2"**
  + **--after="today"**
  + **--after="2016-1-2" --before="today"** 🡪 Commits para localizar por fechas.
  + **--author="Santiago Iván"** 🡪 Commits por autor que cumplan con las letras.
  + **--grep="FOOTER"** 🡪 Busca los commits que cumplan tal cual está escrito entre las comillas.
  + **--grep="FOOTER" -i** 🡪 Busca los commits que cumplan sin importar mayúsculas o minúsculas.
  + **-- index.html** 🡪 Busca los commits en un archivo en específico.
  + **-S"Por contenido"** 🡪 Buscar los commits con el contenido dentro del archivo.

Borrar commits

* **git reset**
  + **\*** 🡪 Borra todo lo que se encuentra en el staging area.
  + **<file or directory>** 🡪 Borra archivo o directorio específico del staging area.
  + **--hard <ID del commit base>** 🡪 Borra todos los commit hacia arriba. Aunque se hayan borrado algún commit, si se tiene el ID se puede recuperar con el mismo comando, aunque seguirá borrando del ID hacia arriba.
  + **--mixed <ID del commit base>** 🡪 Se borran los commits hacia arriba pero no borra el working área, por lo que los cambios que se tengan se tendrán que hacer add y commit. Entonces sirve para encapsular muchos commit en uno solo. Al igual que con --hard, se puede recuperar los commits borrados con el mismo comando pero poniendo el ID del último commit borrado y crea de nuevo la rama como estaba.
  + **--soft <ID del commit base>** 🡪 Igual que el --mixed pero realiza un add a los archivos que están cambiados en el working área.
* **git checkout <ID del commit base>** 🡪 Para regresar en el tiempo. Comúnmente se utiliza para a partir de ahí crear una rama.

Ramas

* **git checkout -b <branch>** 🡪 Crea una nueva rama y el HEAD se mueve a esa rama.
* **git branch** 
  + **-a** 🡪 Muestra las ramas que se tienen en el proyecto.
  + **<branch>** 🡪 Crean una nueva rama.
  + **-d <branch>** 🡪 Borra una rama, solo si se ha realizado merge.
* **git merge <branch que queremos fusionar al branch actual> -m <mensaje>** 🡪 Para fusionar ramas en un nuevo commit.
* **git rebase <branch que queremos agregar los commits al branch actual>** 🡪 Para fusionar ramas pero respetando los commits de la rama que se fusiona. Quedará como último commit el de la rama donde se encuentra en ese momento pero la rama fusionada quedará en su último commit. Para hacer que se emparejen se tiene que hacer merge. La mejor forma de hacerlo es posicionarse en la rama que queremos desaparecer y hacer rebase hacia master.

Stash

* **git stash** 🡪 Guarda cambios temporalmente sin hacer commit y regresa los archivos hasta el último commit realizado.
  + **list** 🡪 Muestra si hay algo guardado en el stash y sobre que commit.
  + **apply** 🡪 Regresa los cambios guardados en el stash.

Ignorar archivos en commits

* Se crea archivo .gitignore dentro de la carpeta del proyecto.
* Dentro del archivo especificar archivos que se tienen que ignorar en los commit como \*.txt o abc.txt

Repositorios remotos

*Crear llave SSH:*

* **ssh-keygen -t rsa -b 4096 -C “jahazielmartinez@gmail.com”** 🡪 Crea una llave SSH, este comando es de Linux, por lo que se deberá hacer con el programa Git Bash y la llave se genera en el directorio de usuario, carpeta .ssh (en Windows en C:\Users\everardo\_martinez\.ssh). El correo debe ser el registrado en GitHub.
* **Borrar cache con llaves ssh anteriores (en ~/.ssh)**
  + **eval $(ssh-agent)** 🡪 Para que reconozca el agente.
  + **ssh-add -D** 🡪 Borrar cache de llaves ssh.
* **Agregar llave ssh (en ~/.ssh)**
  + **ssh-add <llave\_rsa>**
* **Probar una cuenta configurada en el config**
  + **ssh -T cuenta1**
* **Agregar llave SSH.pub en GitHub desde Settings/SSH Keys.**

*Personales*

* **git clone <dirección https del remoto>** 🡪 Descarga la carpeta del proyecto desde GitHub, no es necesario inicializar git para usar clone.
* **git remote** 
  + **add origin <dirección HTTPS o SSH>** --> Agregar un repositorio remoto llamado origin (se recomienda usar ese nombre por defecto).
  + **-v** 🡪 Para ver las conexiones que tenemos en el proyecto.
  + **remove origin** 🡪 Borrar conexión origin.
* **git fetch origin** 🡪 Para traer cambios del repositorio remoto a origin/master. Siempre se debe hacer esto junto con el merge antes de push cuando se está en un repositorio donde hay muchos colaboradores.
* **git merge origin/master** 🡪 Para fusionar los cambios de origin/master (que es un espejo de origin una vez hecho el fetch) a nuestra rama donde estamos trabajando.
* **git push origin <rama> <opciones>**🡪 Se envía al repositorio remoto origin la rama especificada.
  + Opciones:
    - --tags - Envía también los tags que se tienen del proyecto.

**NOTA: En caso de tener problema al realizar fetch o push por proxy o por permisos, se debe agregar el proxy y desactivar sslVerify (así lo solucioné en FG):**

* **git config --global http.sslVerify false**

*Forked (para colaborar en proyectos de terceros)*

* **Buscar proyecto en GitHub y hacer clic en botón Fork para clonarlo a nuestro repositorio.**
* **git remote** 
  + **add origin <dirección HTTPS o SSH del proyecto forked>**
  + **add upstream <dirección HTTPS o SSH del proyecto a colaborar>**
* **git clone <dirección https del remoto> o git pull upstream master**
* **git fetch origin** 🡪 Para descargar los cambios del clon del proyecto de nuestro repositorio a origin/master en caso de que haya más colaboradores apuntando a nuestro clon.
* **git merge origin/master** 🡪 Fusionamos los cambios del proyecto clon de nuestro repositorio a nuestra rama de trabajo.
* **git fetch upstream** 🡪 Para descargar los cambios del proyecto principal a upstream/master.
* **git merge upstream/master** 🡪 Fusionamos los cambios del proyecto forked a nuestra rama de trabajo.
* **Hacemos los cambios en local.**
* **git fetch upstream** 🡪 Para descargar los cambios del proyecto principal.
* **git merge upstream/master** 🡪 Fusionamos los cambios del proyecto forked a nuestra rama de trabajo.
* **git fetch origin** 🡪 Para descargar los cambios del clon del proyecto de nuestro repositorio a origin/master en caso de que haya más colaboradores apuntando a nuestro clon.
* **git merge origin/master** 🡪 Fusionamos los cambios del proyecto clon de nuestro repositorio a nuestra rama de trabajo.
* **git push origin <rama>** 🡪 Se envía al repositorio remoto forked la rama especificada.
* **En GitHub presionamos el botón “New pull request”.**
* **Elegimos la rama que queremos enviar y la rama destino y presionamos “Create pull request”.**
* **En el proyecto principal, el autor acepta la propuesta presionando “Acept and merge”.**

**GitHub como gestor de proyectos:**

Tareas:

Al crear un proyecto se pueden crear tareas y los integrantes que estarán encargados de realizarlas. Lo primero que se debe hacer es crear las tareas específicas, esto se hace en el apartado de Issues-“Milestones”-“New milestone” ingresando el título de la tarea, su descripción y la fecha de fin (opcional). Una vez teniendo todas las tareas se crean los Issues o los problemas que queremos desarrollar/corregir con las tareas que ya están declaradas, esto se hace en el apartado Issues-“New issue” y se le asigna colaborador, etiqueta y tarea que soluciona el problema. Este mismo apartado lo puede utilizar el área de pruebas para ingresar comentarios sobre cierto Issue donde encontraron errores o crear un nuevo Issue reportando un error nuevo, pueden utilizar la etiqueta bug.

Documentación:

En el apartado de Wiki se pueden agregar páginas para explicar el status actual de proyecto, documentar el funcionamiento, manual de instalación, etc.

ZenHub:

Se instala como plugin de Chrome y cuando estamos en GitHub agrega más opciones que ayudarán a gestionar mejor los milestones e issues (pestaña “Boards”) o avance como gráfica (Boardown)

**Utilizar GitHub para hacer deployment a producción:**

Comandos a realizar en el servidor de producción:

* **git clone <dirección https del remoto>** 🡪 Descarga la carpeta del proyecto desde GitHub, no es necesario inicializar git para usar clone. Una vez descargado el proyecto, para actualizarlo se deberá hacer git pull.
* **git remote add origin git@github.com:jahazielmtz/proyecto.git** --> Agregar un repositorio remoto llamado origin apuntando al proyecto en GitHub por HTTPS o SSH.
* **git pull origin master** 🡪 Es un fetch y merge en uno. Sirve para pasar los cambios de origin al servidor.

**Utilizar Bare Repositories en lugar de GitHub**

Comandos:

* EN SERVER: **git init --bare <repositorio.git>** 🡪 Se realiza en el bare repositorie.
* EN LOCAL: **git clone <ruta\_servidor:carpeta/repositorio.git>** 🡪 Se clona el bare repositorie, se incluye la conexión origin al mismo repositorio.
* EN LOCAL: **Realizamos cambios al proyecto local.**
* EN LOCAL: **git push origin master** 🡪 Enviamos cambios al bare repositorie.
* EN SERVER: **git clone <repositorio.git> <carpeta para deploy>** 🡪 Clonamos el bare respositorie en la carpeta para deploy y ahí estarán los archivos.
* EN LOCAL: **Realizamos cambios al proyecto local.**
* EN SERVER: **git fetch origin** 🡪 Descargamos cambios de local a nuestro servidor.
* EN SERVER: **git merge origin/master** 🡪 Fusionamos origin/master a la rama general.

**Utilizar GitLab en el servidor propio.**

Se instala en un servidor propio y se maneja casi igual que GitHub pero de manera privada.

**Utilizar GitHub Pages (solo para aplicaciones front como blogs, portafolios y documentación)**

* Se crea un nuevo repositorio en GitHub y lo bajamos a local.
* Editamos el repositorio y le ponemos la URL: http:jahazielmtz.github.io/<repositorio>
* Creamos una rama llamada gh-pages.
* Desarrollamos blog en esa rama y hacemos push a GitHub.

**Bitbucket** - Investigar