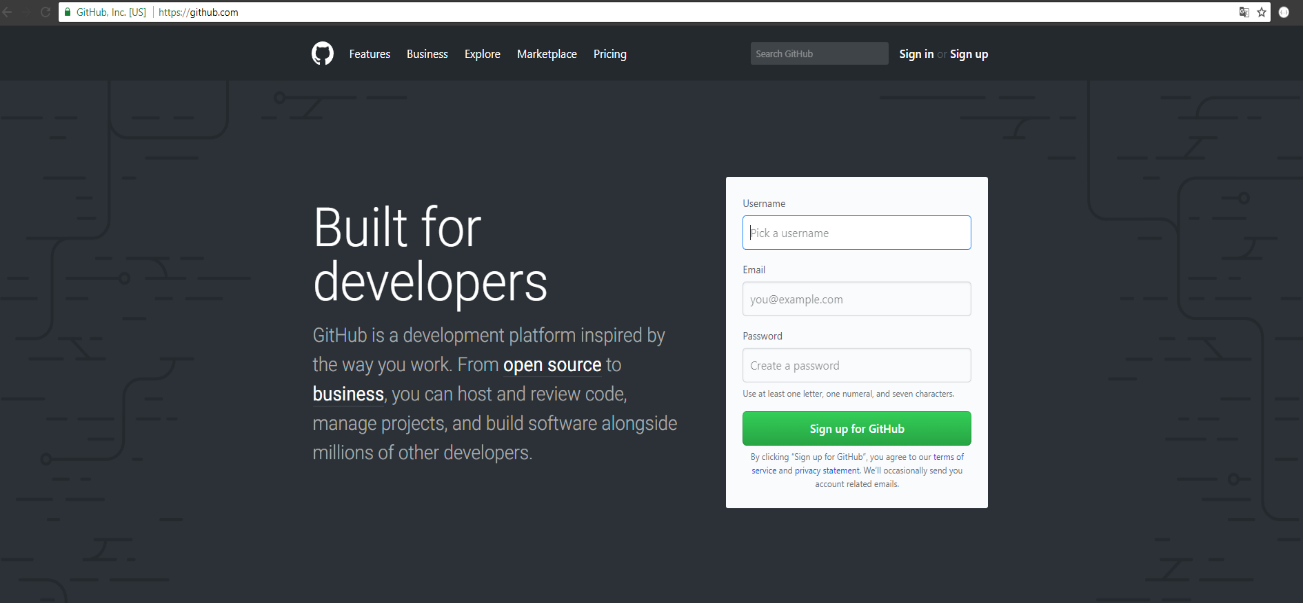
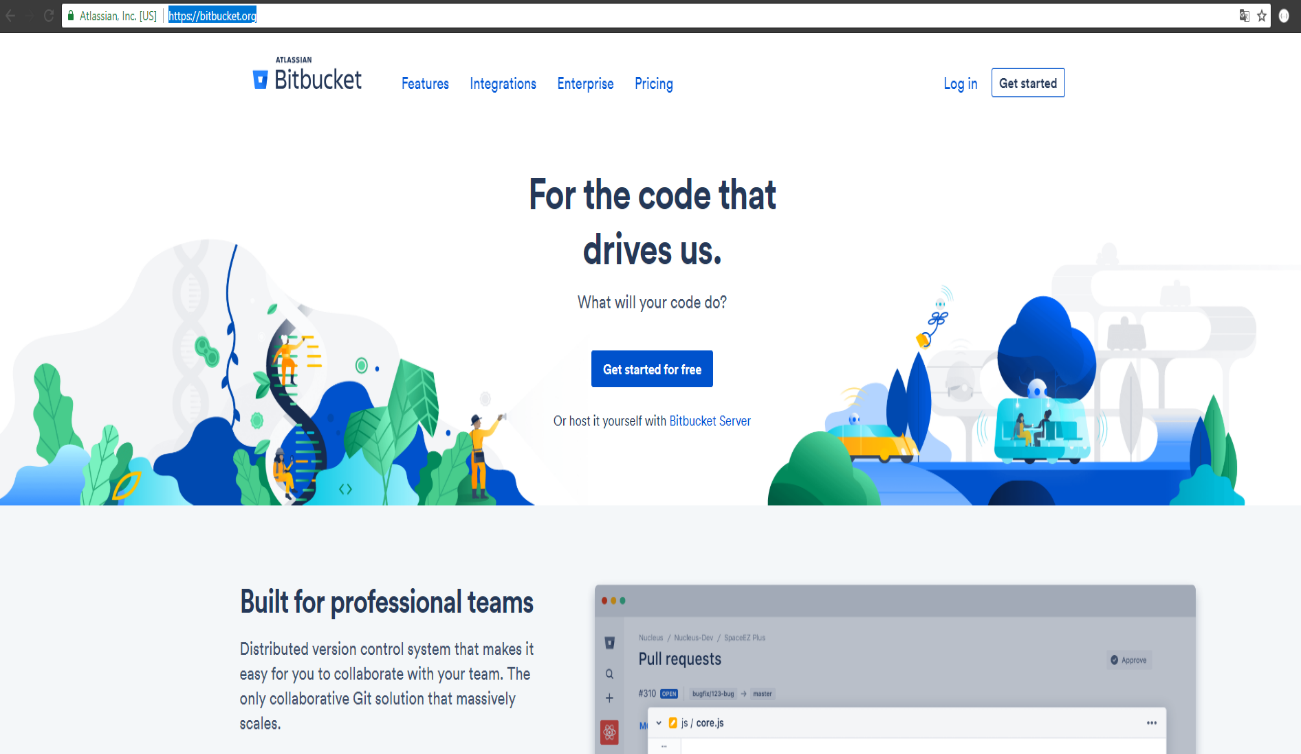
|  |  |
| --- | --- |
| Una imagen de una carretera de curvas con árboles  GIT  [Subtítulo del documento] | Descripción breve  [Capte la atención del lector con un resumen atractivo. Este resumen es una breve descripción del documento. Cuando esté listo para agregar contenido, haga clic aquí y empiece a escribir.]  Windows User  [Título del curso] |

****

****

[**https://github.com/**](https://github.com/)

****

[**https://bitbucket.org/**](https://bitbucket.org/)

Contenido

[¿Qué necesitas? 3](#_Toc114235932)

[**Comandos básicos de GIT** 3](#_Toc114235933)

[GIT HELP 3](#_Toc114235934)

[GIT CONFIG 3](#_Toc114235935)

[GIT INIT 4](#_Toc114235936)

[GIT ADD 4](#_Toc114235937)

[GIT CLONE 4](#_Toc114235938)

[GIT COMMIT 4](#_Toc114235939)

[GIT STATUS 5](#_Toc114235940)

[GIT PUSH 5](#_Toc114235941)

[GIT PULL 5](#_Toc114235942)

[GIT REMOTE 5](#_Toc114235943)

[GIT BRANCH 6](#_Toc114235944)

[GIT CHECKOUT 6](#_Toc114235945)

[GIT MERGE 6](#_Toc114235946)

[GIT LOG 8](#_Toc114235947)

[Deshacer archivo agregado (GIT ADD . ) 8](#_Toc114235948)

[Deshacer un Archivo Modificado 9](#_Toc114235949)

[GIT RESET 9](#_Toc114235950)

[FORZAR GIT PULL 10](#_Toc114235951)

[**Explicación:** 10](#_Toc114235952)

[FORZAR GIT PUSH 10](#_Toc114235953)

[TAGS 10](#_Toc114235954)

[Git Flows 10](#_Toc114235955)

[Flujo Git 10](#_Toc114235956)

[Ventajas 11](#_Toc114235957)

[Desventajas 11](#_Toc114235958)

[Git Lab Flow 12](#_Toc114235959)

[Ventajas 13](#_Toc114235960)

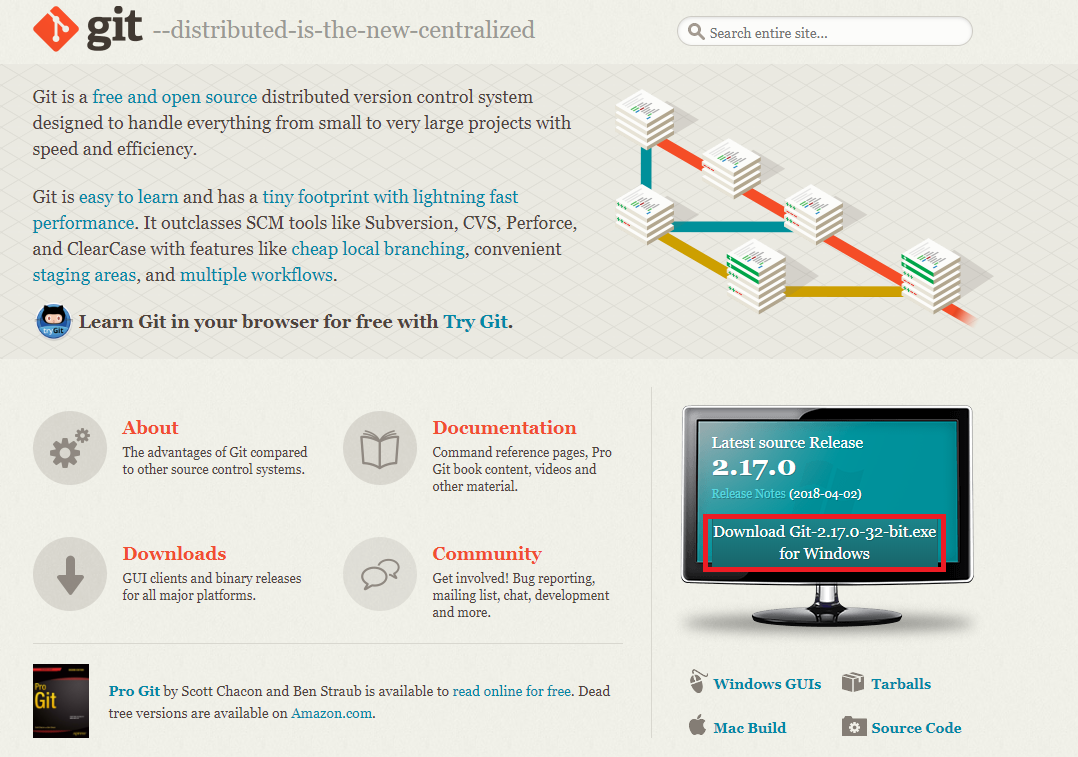
[Desventajas 13](#_Toc114235961)

[Release Flow (by Microsoft) 13](#_Toc114235962)

# ¿Qué necesitas?

* Solamente tener GIT instalado en tu sistema.

<https://git-scm.com/>



**Comandos básicos de GIT**

# GIT HELP

* git help
* git help <verbo> x ejemplo add

# GIT CONFIG

Es usado para establecer una configuración específica de usuario, como sería el caso del email, un algoritmo preferido para diff, nombre de usuario y tipo de formato, etc

* git config --global user.name "fname lname"
* git config --global user.email [example@gmail.com](mailto:example@gmail.com)
* git config --global user.password "secret"

Por base de repositorio, también puede editar .git/config manualmente en su lugar.

* git config --list:Ejecute para verificar el nombre de usuario y el correo electrónico actuales en su repositorio local.

**Nota:** Si la consola queda en “:” al final, salir con “q”

# GIT INIT

Este comando se usa para crear un nuevo repertorio vacío de GIT

1. git init

2. git config user.name ealbarracindev

3. git config user.email ealbarracindev@gmail.com

4. git config user.password key\_privada

5. git add origin <https://github.com/username/repo.git> este siguente punto solo si no esta sincronizado todavia con el remoto

GIT ADD

Este comando puede ser usado para agregar archivos al index

* git add .

# GIT CLONE

Permite clonar un repositorio local desde el servidor de GIT

* git clone <http://10.90.1.27:3000/GENDARMERIANACIONAL/GLOSARIO.git>

# GIT COMMIT

El comando commit es usado para cambiar a la cabecera. Ten en cuenta que cualquier cambio comprometido no afectara al repertorio remoto ES TODO LOCAL.

* git commit –m “Mensaje del commit”

GIT STATUS

Este comando muestra la lista de los archivos que se han cambiado junto con los archivos que están por ser añadidos o comprometidos.

* git status

# GIT PUSH

Este es uno de los comandos más básicos. Un simple push envía los cambios que se han hecho en la rama principal de los repertorios remotos que están asociados con el directorio que está trabajando. Por ejemplo:

* git push origin master

# GIT PULL

Para poder fusionar todos los cambios que estaban en el server en el repositorio local que estamos trabajando trabajando.

* git pull

# GIT REMOTE

El comando git se usa para conectar a un repositorio remoto.

* git remote add [nombre] [url]

Ejemplo:

git remote add origin <http://10.90.1.27:3000/aplicacion-name/repo-name.git>

El siguiente comando muestra los repositorios remotos que están configurados actualmente:

* git remote –v

# GIT BRANCH

Este comando para el manejo de branchs:

* git branch (listar todas)
* git branch featureA (solo crea la rama)
* git checkout -b feature (crea la rama y se posiciona en ella)
* git branch -d <branch-name> (elimina la rama)

# GIT CHECKOUT

El comando checkout se puede usar para crear ramas o cambiar entre ellas. Por ejemplo, el siguiente comando crea una nueva y se cambia a ella:

* git checkout -b <banch-name>

Para cambiar de una rama a otra solo usa:

* git checkout <branch-name>

# GIT MERGE

Este comando se usa para fusionar una rama con otra rama activa:

Nos situamos en la rama que queremos actualizar por ejemplo la rama master

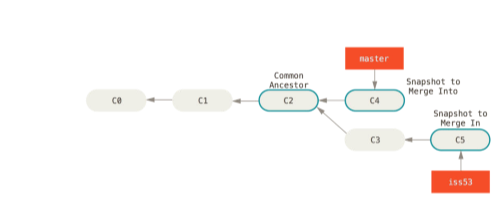
1. git checkout master

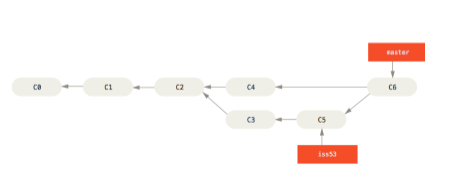
Desde master traemos todos los cambios de la rama hotfix

1. git merge hotfix

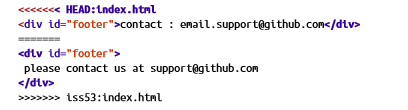


Antes del Merge (master en C4) y iss53(hotfix) (C5)



Luego del merge junta ambas ramas en (C6)

Pueden surgir conflictos en las fusiones y se verán de la siguiente manera

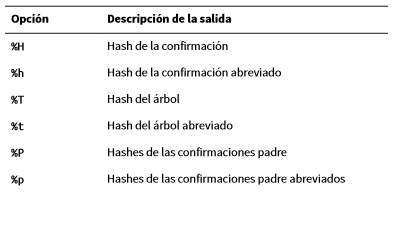


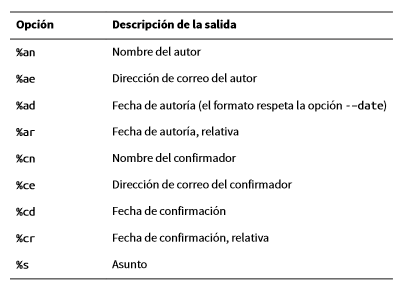
Como estamos parados en master nuestro HEAD es esa rama y la otra linea es lo que esta en la rama iss53 y nosotros deberemos decider que version debe quedar en la rama C6.

# GIT LOG

Ejecutar este comando muestra una lista de commits en una rama junto con todos los detalles. Por ejemplo:

* git log
* git log --stat
* git log --pretty=oneline
* git log --pretty=format:"%h, %ar : %s"





# GIT DIFF

Este comando permite ver los cambios realizados en mi local con respecto al remoto

* git diff

# GIT COMMIT –amend

Este comando nos permite agregar un ultimo cambio al ultimo “git add .”

# Deshacer archivo agregado (GIT ADD . )



git reset HEAD CONTRIBUTING.md

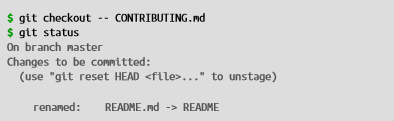


# Deshacer un Archivo Modificado

Queremos volver atrás esos cambios de un archive modificado y no preparado.



Si ponemos git status vemos README



## Git Checkout -- .

Permite recuperar los cambios realizados o archivos borrados al ultimo commit que teniamos

Este comando nos permite volver a la versión inicial del ultimo pull o clone de la rama en la que estamos

# GIT RESET

Tener en cuenta que este comando es un comando destructivo, esto implica que si reseteamos un commit nos puede causar problemas cuando trabajamos en equipo con respecto a los commits de nuestro equipo.

Para resetear el index y el directorio que está trabajando al último estado comprometido se usa este comando:

* git reset --hard HEAD

Ejemplo

git reset --hard d545833fa849d41a4586cd14579d58a28822e182

git reset --soft HEAD@{1}

git commit -m “Revertir el commit d545833fa849d41a4586cd14579d58a28822e182”

**--soft => Deja los archivos modificados con el rollback.**

# GIT REVERT

Este comando nos permite revertir cambios especificando la cantidad de commits que queremos volver en el tiempo. NO es un comando destructivo con lo es RESET.

Recordar que para ver los commits que tenemos podemos utilizar *git log –oneline --decorate*

* git revert hash-commit-a-revertir

También se puede utilizar HEAD~1 (HEAD es el índice actual), para decir que queremos volver al commit anterior, quedaría de la siguiente manera:

* git revert HEAD~1

Lo anterior haciendo que se incorporen commits al stage. Para volver mas un commit a la vez:

* git revert HEAD~1

# FORZAR GIT PULL

* git fetch -–all

Si es la master

* git reset --hard origin/master

Si es alguna rama

* git reset --hard origin/<branch\_name>

### **Explicación:**

git fetch descarga lo último desde el control remoto sin intentar fusionar o volver a generar nada.

A continuación, git reset restablece la rama maestra a lo que acaba de recuperar. La –-hard opción cambia todos los archivos en su árbol de trabajo para que coincida con los archivos en origin/master

# FORZAR GIT PUSH

* git push --force origin B:C

# TAGS

# Git Flows

## Flujo Git

Git Flow es el flujo de trabajo más conocido de esta lista. Fue creado por [Vincent Driessen en 2010](http://nvie.com/posts/a-successful-git-branching-model/) y se basa en dos ramas principales con una vida útil infinita:

* master— esta rama contiene código de producción. Todo el código de desarrollo se fusiona masteren algún momento.
* develop— esta rama contiene código de preproducción. Cuando las características están terminadas, se fusionan en desarrollo.

Durante el ciclo de desarrollo, se utilizan una variedad de ramas de apoyo:

* feature-\*— Las ramas de funciones se utilizan para desarrollar nuevas funciones para los próximos lanzamientos. Puede ramificarse developy debe fusionarse con develop.
* hotfix-\*— Las ramas de hotfix son necesarias para actuar inmediatamente sobre un estado no deseado de master. Puede ramificarse de mastery debe fusionarse con mastery develop.
* release-\*— las ramas de versión admiten la preparación de una nueva versión de producción. Permiten corregir muchos errores menores y preparar metadatos para un lanzamiento. Puede ramificarse de developy debe fusionarse con mastery develop.

## Ventajas

* Asegura un estado limpio de las sucursales en cualquier momento del ciclo de vida del proyecto
* La denominación de las ramas sigue un patrón sistemático que facilita su comprensión.
* [Tiene extensiones](https://github.com/nvie/gitflow) y soporte en las herramientas git más utilizadas.
* Es ideal cuando se necesita una versión múltiple en producción.

## Desventajas

* El historial de Git se vuelve ilegible
* La división master/develop se considera redundante y dificulta la entrega continua y la integración continua.
* No se recomienda cuando se necesita mantener una sola versión en producción.

## Git Lab Flow

GitLab Flow es un flujo de trabajo creado por [GitLab en 2014](https://about.gitlab.com/2014/09/29/gitlab-flow/" \t "_blank) . Combina [el desarrollo basado en](https://en.wikipedia.org/wiki/Feature-driven_development) funciones y las ramas de funciones con el seguimiento de problemas. La mayor diferencia entre GitLab Flow y GitHub Flow son las ramas de entorno que tienen en GitLab Flow (p. ej . stagingy production) porque habrá un proyecto que no podrá implementarse en producción cada vez que combine una rama de función (p. ej., aplicaciones SaaS y aplicaciones móviles )

El GitLab Flow se basa en 11 reglas:

1. Use ramas de características, sin compromisos directos enmaster
2. Prueba todas las confirmaciones, no solo las que están enmaster
3. Ejecute todas las pruebas en todas las confirmaciones (si sus pruebas duran más de 5 minutos, haga que se ejecuten en paralelo).
4. Realice revisiones de código antes de fusionarse con master, no después.
5. Las implementaciones son automáticas, basadas en sucursales o etiquetas.
6. Las etiquetas las establece el usuario, no CI.
7. Los lanzamientos se basan en etiquetas.
8. Las confirmaciones enviadas nunca se reorganizan.
9. Todos comienzan desde mastery apuntan al maestro.
10. Corrija los errores en masterprimer lugar y libere las ramas en segundo lugar.
11. Los mensajes de confirmación reflejan la intención.

## Ventajas

* Define cómo hacer la Integración Continua y la Entrega Continua
* El historial de git será más limpio, menos desordenado y más legible (vea [por qué los desarrolladores prefieren squash and merge, en lugar de solo fusionar, en este artículo](https://softwareengineering.stackexchange.com/questions/263164/why-squash-git-commits-for-pull-requests) )
* Es ideal cuando se necesita una sola versión en producción.

## Desventajas

* Es más complejo que el GitHub Flow
* Puede volverse complejo como Git Flow cuando necesita mantener múltiples versiones en producción

# Release Flow (by Microsoft)

* Siempre se deployan reléase branches
* Si hay un fallo se hace rama hotfix y Cherry-pick de los cambios
* Luego se llevan también a las releases branches
* Una reléase por sprint