

INTRODUCCION A LA PROGRAMACION C#

3 – GIT COLABORATIVO: CREA TU PORTFOLIO

UNIDAD: 3

PRESENTACIÓN: En esta unidad se explicaran los conceptos de sistema de versionado de archivos GIT y como usarlo.

OBJETIVOS

Que los participantes logren: Comprender los fundamentos de GIT y saber usar un set de comandos que le permitan hacer un uso provechoso de esta herramienta.

TEMARIO

Introducción.....	4
Los tres estados: El flujo básico de trabajo en git	6
Instalando git	7
Bases de GIT para su utilización local	8
Configurando git.....	8
Chequeado la configuración	8
Creando un nuevo repositorio - git init.....	8
Verificando el estado - git status	9
Staging - git add.....	10
Confirmando nuestro trabajo en el repositorio local - git commit	11
Viendo el historial de commits - git log	11
Bases de GIT para su uso distribuido y colaborativo	12
Let's do it ! Conectando a un repositorio remoto - git remote add	12
Subiendo nuestro commit a un servidor - git push	12
Clonando un repo - git clone	13
Obteniendo los cambios del servidor - git pull.....	14
Branches.....	15
Creando nuevos branches - git branch.....	15
Moviéndonos entre branches - git checkout	16
Mergeando branches - git merge	16
Manejo avanzado de la estructura del repositorio	18
Chequeando diferencias entre commits.....	18
Inspeccionando un commit – git show	19
Diferencias entre dos commit – git diff.....	19
Volviendo a un commit previo	20
Resolviendo conflictos de merge.....	21
Configurando .gitignore.....	27

Introducción

Podemos definir a GIT en pocas palabras como un *sistema de control de versiones distribuido* que trabaja manteniendo instantáneas del estado de todo el repositorio.

Sistema de control de versiones:

El control de versiones es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.

Distribuido:

git le da a cada programador una copia local del historial del desarrollo entero. Su ventaja frente a los sistemas de control de versiones centralizados es que si un servidor muere, y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios de los clientes puede copiarse en el servidor para restaurarlo. Ya que con git, cada cliente replica una instantánea de *todo* el repositorio a un momento determinado en el tiempo.

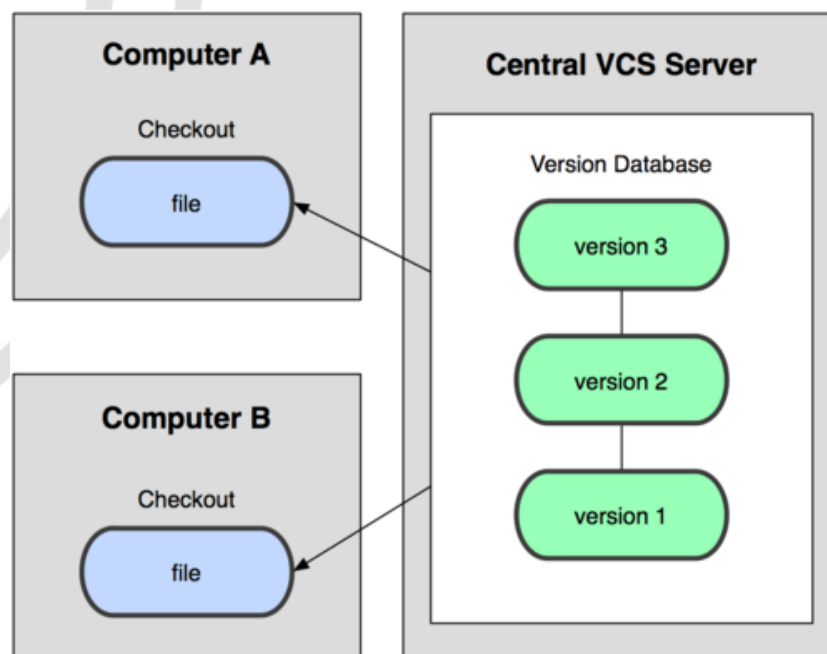


Figura - Versionado Centralizado

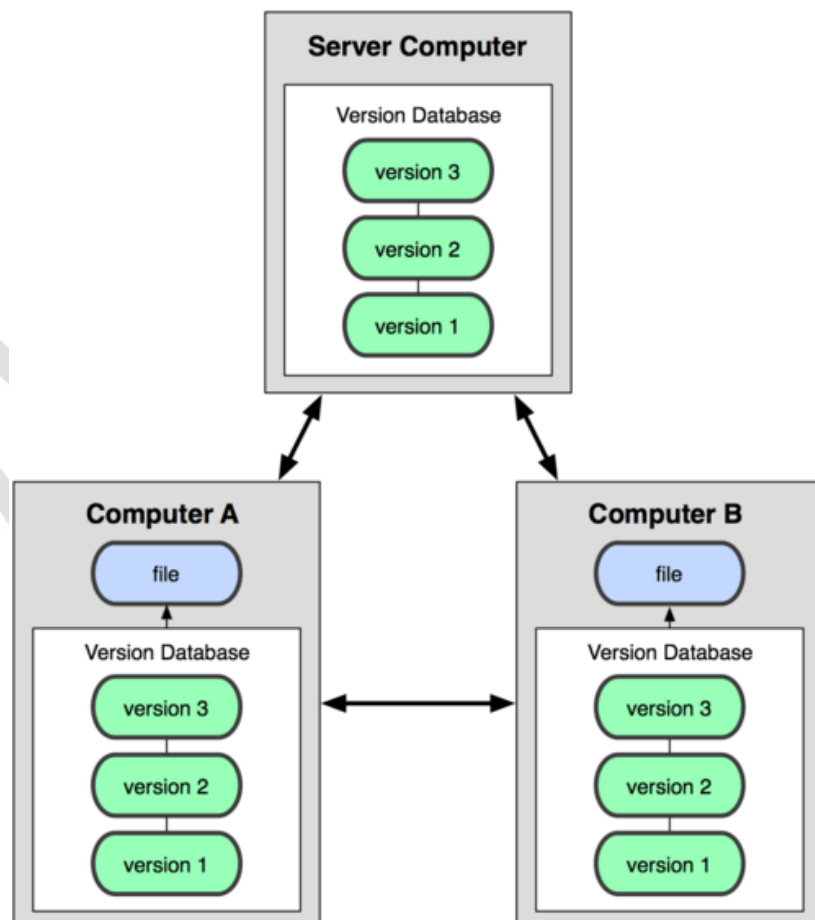


Figura - Versionado Distribuido

La integridad en git:

Todo cambio en Git se representa en un checksum antes de que se almacene, y luego se lo identifica con ese checksum.

Esto significa que es imposible cambiar el contenido de cualquier archivo o directorio sin que git lo sepa.

Esta funcionalidad está integrada en git al mas bajo nivel y es parte integral de su filosofía. No puede perder información en tránsito o modificar un archivo sin que git lo detecte.

El mecanismo que git usa para esta suma de comprobación(checksum) se llama hash SHA-1. Esta es una cadena de 40 caracteres compuesta de caracteres hexadecimales (0-9 y a-f) y calculada en función del contenido de un archivo o estructura de directorio en git.

Un hash SHA-1 se ve más o menos así:

24b9da6552252987aa493b52f8696cd6d3b00373

Y git almacena todo en su base de datos no por nombre de archivo sino por el valor hash de sus contenidos.

Si bien la mayoría del trabajo con git se hace local, existen repositorios online, que hacen las cosas más fáciles cuando tenemos que compartir nuestro código con otros colaboradores.

Los mas conocidos son: gitHub, BitBucket o gitLab.

Los tres estados: El flujo básico de trabajo en git

Preste atención ahora: aquí está lo principal que debe recordar acerca de Git si desea que el resto de su proceso de aprendizaje transcurra sin problemas.

*git tiene tres estados principales en los que pueden residir tus archivos: **committed**, **modified**, y **stage**.*

- Committed significa que los datos se almacenan de forma segura en su base de datos local.
- Modified significa que ha cambiado el archivo pero aún no lo ha comiteado con su base de datos.
- Stage significa que ha marcado un archivo modificado en su versión actual para ir a su próxima instantánea de commit.

Esto nos lleva a las tres secciones principales de un proyecto de Git: el directorio de Git, el directorio de trabajo y el área de stage.

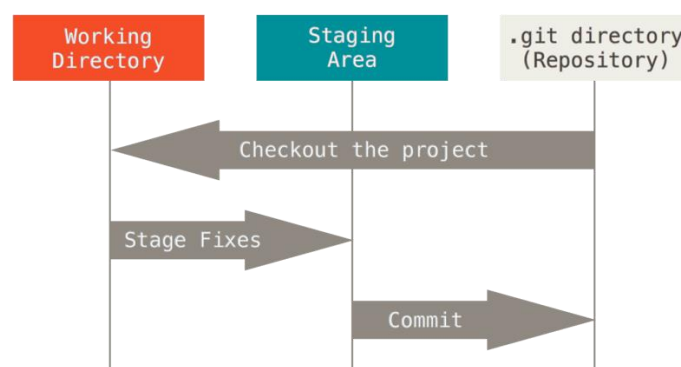


Figura – El flujo básico de git

Directorio de git: Es donde git almacena los metadatos y la base de datos de objetos para su proyecto. Esta es la parte más importante de git, y es lo que se copia cuando clonas un repositorio desde otra computadora.

Directorio de trabajo: Es la copia de una versión del proyecto. Estos archivos se extraen de la base de datos comprimida en el directorio de git y se colocan en el disco para que los use o modifique.

Stage área: Es un archivo, generalmente contenido en su directorio de git, que almacena información sobre lo que se incluirá en el próximo commit.

Instalando git

- En Windows descargar gitbash desde <https://gitforwindows.org/>
- En Linux - abrir una terminal e instalar git vía el administrador de paquetes. En Ubuntu el comando es: `sudo apt-get install git`.
- En OS X lo mas fácil es instalar [homebrew](https://brew.sh/), y ejecutar `brew install git` desde la terminal.

Bases de GIT para su utilización local

Configurando git

```
osvaldo@guille-PC MINGW64 ~
$ git config --global user.name "Oswaldo Olmos"
$ git config --global user.email oolmos@xsidesolutions.com
```

Chequeado la configuración

```
osvaldo@guille-PC MINGW64 ~
$ git config --list
core.symlinks=false
core.autocrlf=true
core.fscache=true
color.diff=auto
color.status=auto
color.branch=auto
color.interactive=true
help.format=html
rebase.autosquash=true
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
http.sslbackend=openssl
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
credential.helper=manager
user.name=Oswaldo Olmos
user.email=oolmos@xsidesolutions.com
pull.rebase=true
alias.lga=log --graph --oneline --all --decorate
```

Creando un nuevo repositorio - git init

Como mencionamos anteriormente, git almacena sus archivos e historial directamente como una carpeta en el proyecto.

Para configurar un nuevo repositorio, necesitamos abrir un terminal, navegar a nuestro directorio de proyectos y ejecutar git init. Esto habilitará a git para esta carpeta en particular y creará un directorio oculto .git donde se almacenará el historial y la configuración del repositorio.



```
osvaldo@guille-PC MINGW64 ~  
$ pwd  
/c:/Users/osvaldo  
osvaldo@guille-PC MINGW64 ~  
$ cd source/repos/  
osvaldo@guille-PC MINGW64 ~/source/repos  
$ mkdir demogit  
osvaldo@guille-PC MINGW64 ~/source/repos  
$ cd demogit  
osvaldo@guille-PC MINGW64 ~/source/repos/demogit  
$ git init  
Initialized empty Git repository in C:/Users/osvaldo/source/repos/demogit/.git/  
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)  
$
```

Verificando el estado - git status

git status es otro comando imprescindible que devuelve información sobre el estado actual del repositorio:

- Si está todo actualizado
- Qué hay de nuevo
- Qué ha cambiado
- Etc.

Creemos un nuevo archivo: hola.txt y luego ejecutamos git status

```
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)  
$ vim hola.txt  
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)  
$ git status  
On branch master  
  
No commits yet  
  
Untracked files:  
  (use "git add <file>..." to include in what will be committed)  
  
    hola.txt  
  
nothing added to commit but untracked files present (use "git add" to track)  
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)  
$
```

El mensaje devuelto indica que hola.txt no está trackeado por git.

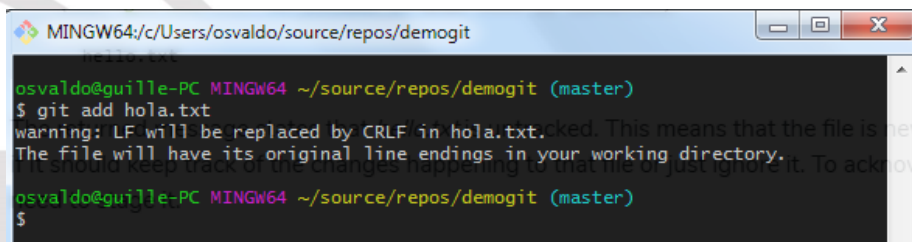
Esto significa que el archivo es nuevo y git aún no sabe si debe hacer un seguimiento de los cambios que suceden en ese archivo o simplemente ignorarlo. Para reconocer el nuevo archivo, debemos "stagearlo".

Staging - git add

git tiene el concepto de un "área de preparación". Lo podemos ver como un espacio en blanco, que contiene los cambios que luego podemos decidir comitear.

Comienza vacío, con el comando git add voy agregando archivos a este área de "preparación", y finalmente para enviar todo (crear una instantánea en el repositorio) uso el commit de git.

En nuestro caso, solo tenemos un archivo, así que agreguemos eso:



```
MINGW64:/c/Users/osvaldo/source/repos/demogit
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$ git add hola.txt
warning: LF will be replaced by CRLF in hola.txt.
The file will have its original line endings in your working directory.
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$
```

Hay otros modificadores del add, por ejemplo:

- git add -A agrega todos los archivos del árbol donde estas ubicado.
- git add *.cs agrega todos los archivos que matcheen con la expresión "*.cs"

Para conocer mas sobre el uso del add, consultar el help o la documentación online.



```
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$ git status
On branch master

No commits yet

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

    new file:   hola.txt
```

Confirmando nuestro trabajo en el repositorio local - git commit

Un commit representa el estado de nuestro repositorio en un punto determinado en el tiempo. Es como una instantánea a la que podemos volver y ver cual era el estado cuando sacamos esa foto.

Para crear un nuevo commit, necesitamos tener al menos un cambio agregado al área de staging (lo hicimos con git add) y ejecutar lo siguiente:

```
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$ git commit -m "Initial commit."
[master (root-commit) f94a883] Initial commit.
1 file changed, 1 insertion(+)
 create mode 100644 hola.txt

osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$
```

Esto creará un nuevo commit con todos los cambios desde el área de staging (agregando hello.txt). Lo indicado después de modificador "m", En este caso la frase :“Initial commit.”, es una descripción personalizada escrita por el usuario que resume los cambios realizados en ese commit.

TIP: Se considera una buena práctica commitear cada vez que tengo algo nuevo codeado y siempre escribir mensajes de commit que digan lo que se está comitteando de forma clara.

Viendo el historial de commits - git log

Con este comando puedo ver el historial de los commits hechos.
Como nosotros hemos hecho uno solo hasta acá, la salida de consola es algo así:

```
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$ git log
commit f94a883a614a5f8c08190a1a9355cc091ec15270 (HEAD -> master)
Author: 000 <oolmos@xsidesolutions.com>
Date:   Wed Jul 25 20:16:06 2018 -0300

    Initial commit.

osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$
```

Bases de GIT para su uso distribuido y colaborativo

Hasta acá, nuestro commit es local; solo existe en la carpeta .git. Aunque un repositorio local es útil en sí mismo, en la mayoría de los casos deberíamos compartir nuestro trabajo y desplegarlo en un servidor o en un servicio de git.

Para este tutorial, vamos a crear una cuenta en github: <https://github.com/>

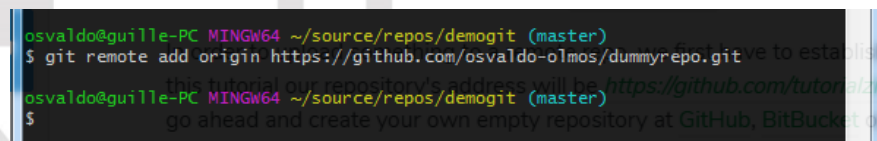
Let's do it !

Conectando a un repositorio remoto - git remote add

Una vez que tenemos nuestra cuenta en github creada, para subir algo a un repositorio remoto, primero debemos establecer una conexión con él.

Para vincular nuestro repositorio local con el de github, ejecutamos la siguiente línea en la terminal:

`git remote add origin <url_del_repositorio>`



```
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$ git remote add origin https://github.com/osvaldo-olmos/dummyrepo.git
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$
```

Un proyecto puede tener muchos repositorios remotos al mismo tiempo. Para poder distinguirlos, les damos diferentes nombres. Tradicionalmente, el repositorio remoto principal en git se llama *origin*.

Subiendo nuestro commit a un servidor - git push

Ahora es el momento de transferir nuestros commits locales al servidor. Este proceso se llama push y se realiza cada vez que queremos actualizar el repositorio remoto.

El comando git para hacer esto es git push y toma dos parámetros:

- El nombre del repositorio remoto (en nuestro caso: origen)
- El branch destino donde vamos a enviar nuestros commits (master es el branch default de todo repositorio).

```
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$ git push origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 224 bytes | 224.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/osvaldo-olmos/dummyrepo.git
 * [new branch] master -> master
```

Dependiendo del servicio que estemos utilizando, tendremos que autenticarnos para hacer el push. Si todo salió bien, cuando browseemos el repositorio remoto creado anteriormente, hello.txt debería estar disponible allí.

Clonando un repo - git clone

Ahora, otros desarrolladores pueden ver y navegar a través de su repositorio remoto en github. Pueden descargarlo localmente y tener una copia completamente funcional de su proyecto con el comando git clone:

```
osvaldo@guille-PC MINGW64 ~/source2/repos2
$ git clone https://github.com/osvaldo-olmos/dummyrepo.git
Cloning into 'dummyrepo'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
```

Se crea automáticamente un nuevo repositorio local, con la versión del repositorio alojada en github configurada como remota.

Obteniendo los cambios del servidor - git pull

Se pueden descargar las actualizaciones hechas a un repositorio con el comando - git pull:

```
osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (master)
$ git pull
Already up to date.
Current branch master is up to date.
osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (master)
$
```

Si no hay cambios para recuperar, la salida de la consola va a ser como la de arriba.

Branches

Al desarrollar una nueva feature, se considera una buena práctica trabajar en una copia del proyecto original, a cada una de estas copias las llamamos branch.

Los branches tienen su propio historial y aíslan sus cambios el uno del otro, hasta que decide mergearlos nuevamente. Esto se hace por algunas razones:

- Una versión que ya funciona y estable del código no se romperá. Esto es primordial para cualquier desarrollador.
- Muchos features pueden desarrollarse de forma segura a la vez por diferentes devs.
- Los desarrolladores pueden trabajar en su propio branch, sin el riesgo de que su código cambie debido al trabajo de otro developer.
- Cuando no está seguro de qué es lo mejor, se pueden desarrollar múltiples versiones del mismo feature en branches separados y luego compararlas.

Creando nuevos branches - git branch

El branch predeterminado de cada repositorio se llama master. Para crear otros branches, use el comando `git branch <nombre_del_branch>`:

```
osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (master)
$ git branch feature1
```

Esto solo crea el nuevo branch, que en este punto es exactamente igual que nuestro master.

Moviéndonos entre branches - git checkout

Ahora, cuando ejecutemos git branch, veremos que hay dos opciones disponibles:

```
osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (master)
$ git branch
  feature1
* master
```

Cómo estamos parados sobre el master, este está marcado con un asterisco. Sin embargo, queremos trabajar en feature1, por lo que debemos cambiar al otro branch. Esto se hace con el comando git checkout, esperando un parámetro: el branch al que cambiar.

```
osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (master)
$ git checkout feature1
Switched to branch 'feature1'

osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (feature1)
$ git branch
* feature1
  master
```

Si hacemos un nuevo git branch, ahora el marcado con asterisco es feature1

Mergeando branches - git merge

Para este ejemplo, vamos a suponer que nuestra nueva funcionalidad a desarrollar queda ejemplificada creando simplemente otro archivo de texto llamado feature1.txt. Lo crearemos, lo agregaremos al área de stage y lo comitearemos.

```
osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (feature1)
$ git add feature1.txt
warning: LF will be replaced by CRLF in feature1.txt.
The file will have its original line endings in your working directory.

osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (feature1)
$ git commit -m "feature1 added"
[feature1 fe10c67] feature1 added
1 file changed, 1 insertion(+)
 create mode 100644 feature1.txt

osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (feature1)
$ git checkout master
Switched to branch 'master'
```

El nuevo feature está completo. Ahora volvemos al branch principal.


```
osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (feature1)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (master)
$
```

Ahora, si abrimos nuestro proyecto en el buscador de archivos, notaremos que feature1.txt ha desaparecido. Esto sucede porque estamos de vuelta en el master, y aquí feature1.txt nunca se creó.

Para incorporar este cambio al master, debemos “mergear”.

```
osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (master)
$ git merge feature1
Updating f94a883..fe10c67
Fast-forward
 feature1.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 feature1.txt
osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (master)
$ ls
feature1.txt  hola.txt
osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (master)
$
```

El master ya está actualizado. Si el branch feature1 ya no es necesario y se puede eliminar, podemos hacerlo ejecutando:

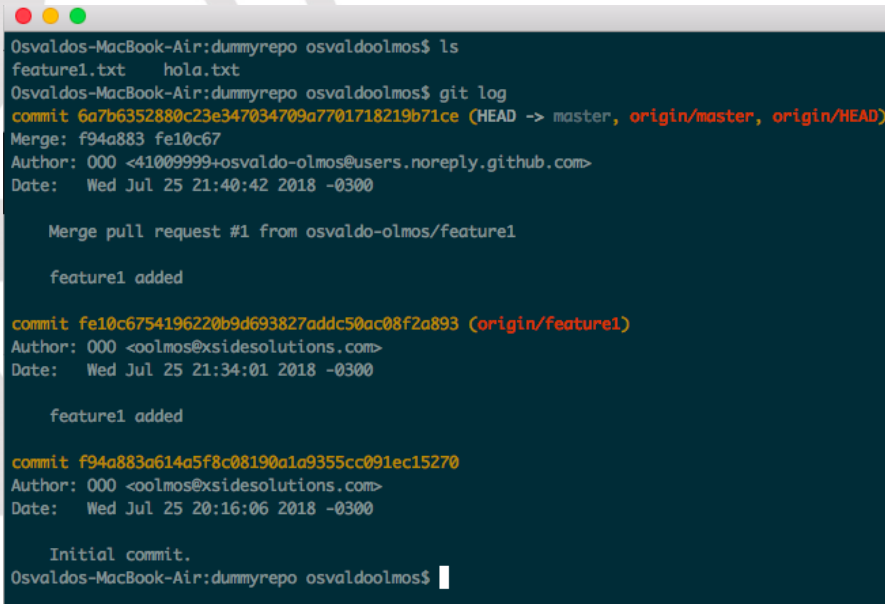
```
osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (master)
$ git branch -d feature1
Deleted branch feature1 (was fe10c67).
osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (master)
$
```

Manejo avanzado de la estructura del repositorio

En esta sección vamos a echar un vistazo a algunas técnicas que nos van a ser útiles en el día a día de un proyecto de desarrollo de software.

Chequeando diferencias entre commits

Como vimos en la introducción, cada commit tiene su identificación única en forma de hash. Para ver una lista de todos los commits y sus ids, podemos usar git log:



```
Osvaldos-MacBook-Air:dummyrepo osvaldoolmos$ ls
feature1.txt  hola.txt
Osvaldos-MacBook-Air:dummyrepo osvaldoolmos$ git log
commit 6a7b6352880c23e347034709a7701718219b71ce (HEAD -> master, origin/master, origin/HEAD)
Merge: f94a883 fe10c67
Author: 000 <41009999+osvaldo-olmos@users.noreply.github.com>
Date:   Wed Jul 25 21:40:42 2018 -0300

    Merge pull request #1 from osvaldo-olmos/feature1

    feature1 added

commit fe10c6754196220b9d693827addc50ac08f2a893 (origin/feature1)
Author: 000 <ooolmos@xsidesolutions.com>
Date:   Wed Jul 25 21:34:01 2018 -0300

    feature1 added

commit f94a883a614a5f8c08190a1a9355cc091ec15270
Author: 000 <ooolmos@xsidesolutions.com>
Date:   Wed Jul 25 20:16:06 2018 -0300

    Initial commit.
Osvaldos-MacBook-Air:dummyrepo osvaldoolmos$
```

Si bien los ids son largos, cuando queremos referenciarlos, podemos usar solo los primeros caracteres del id.



Inspeccionando un commit – git show

Para ver que conforma un commit, podemos ejecutar `git show [commit]`:

```
Osvaldos-MacBook-Air:dummyrepo osvaldoolmos$ git show fe10c67
commit fe10c6754196220b9d693827addc50ac08f2a893 (origin/feature1)
Author: 000 <ooolmos@xsidesolutions.com>
Date:   Wed Jul 25 21:34:01 2018 -0300

    feature1 added

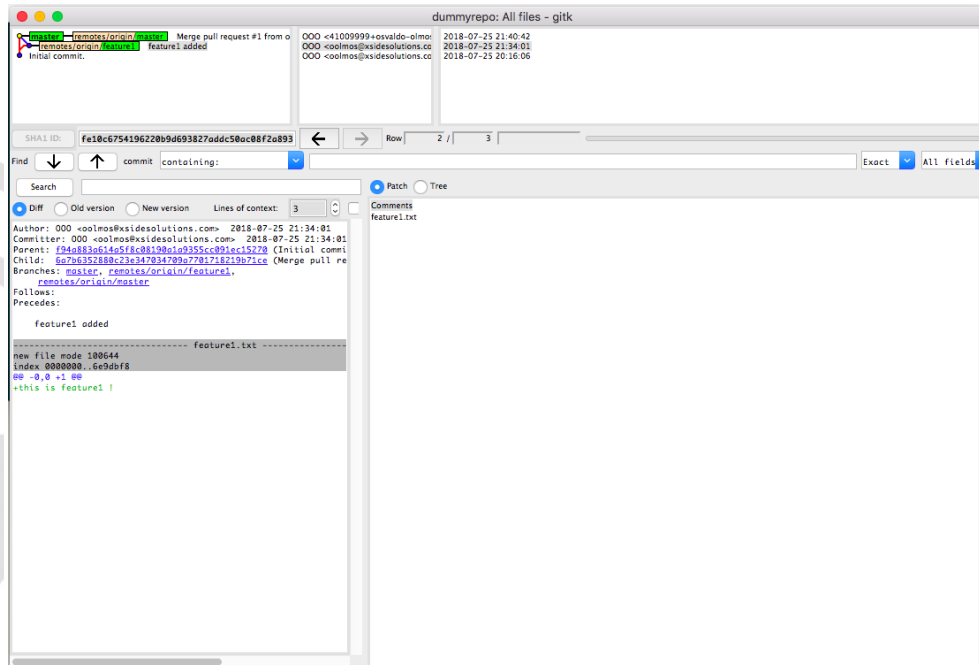
diff --git a/feature1.txt b/feature1.txt
new file mode 100644
index 0000000..6e9dbf8
--- /dev/null
+++ b/feature1.txt
@@ -0,0 +1 @@
+this is feature1 !
Osvaldos-MacBook-Air:dummyrepo osvaldoolmos$
```

Diferencias entre dos commit – git diff

Para ver la diferencia entre dos commits, podemos usar `git diff` con la sintaxis `[commit-from] .. [commit-to]`:

```
Osvaldos-MacBook-Air:dummyrepo osvaldoolmos$ git diff 6a7b6352..f94a883
diff --git a/feature1.txt b/feature1.txt
deleted file mode 100644
index 6e9dbf8..0000000
--- a/feature1.txt
+++ /dev/null
@@ -1,0 +0,0 @@
-this is feature1 !
Osvaldos-MacBook-Air:dummyrepo osvaldoolmos$ gitk
```

O bien podemos usar otras herramientas de interfaz visual, por ejemplo gitk



En ambos casos nos muestra que se agregó el archivo feature1.txt y su contenido (en verde)

Volviendo a un commit previo

Si queremos ver que tenía un commit previo completo, ya sea para hacer pruebas o buscar donde podemos haber introducido un bug, podemos volver con git checkout

```
Osvaldos-MacBook-Air:dummyrepo osvaldoolmos$ git checkout f94a883
Note: checking out 'f94a883'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at f94a883... Initial commit.
Osvaldos-MacBook-Air:dummyrepo osvaldoolmos$ ls
hola.txt
Osvaldos-MacBook-Air:dummyrepo osvaldoolmos$
```

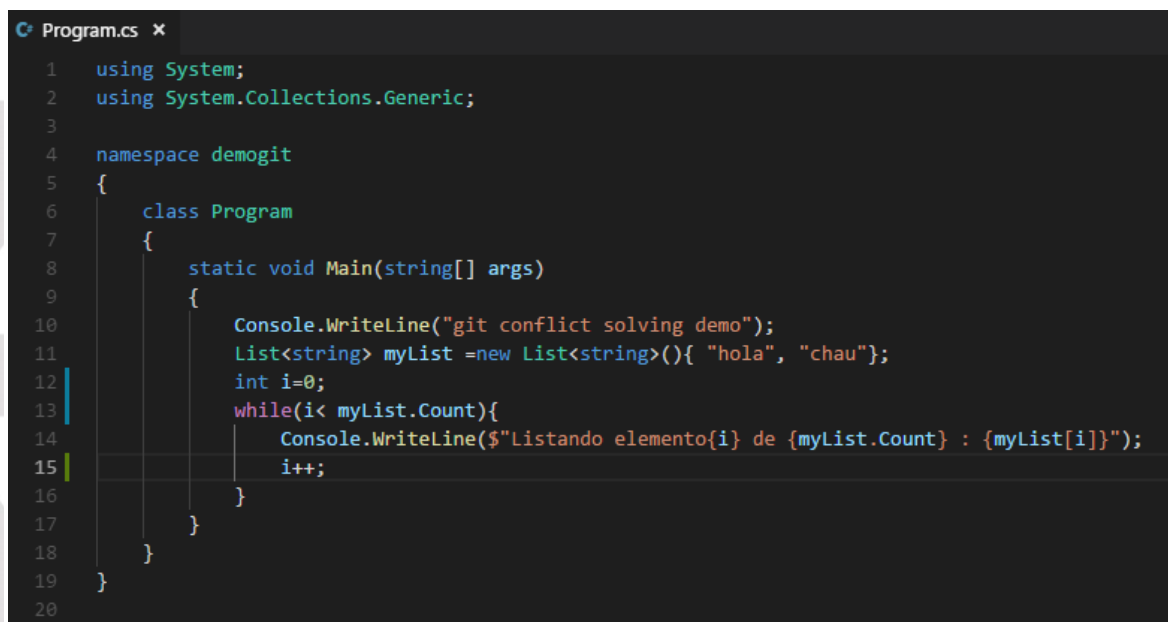
Podemos ver que ahora nuestra versión del código no tiene el archivo feature1.txt

Resolviendo conflictos de merge

Los conflictos usualmente aparecen al mergear branches. Generalmente git puede hacer los merges automáticamente, pero cuando la misma línea está modificada en ambos branches, ahí necesitamos la intervención de los desarrolladores.

Supongamos el siguiente escenario:

Está el desarrollador dev1 que se corta un branch desde master para codear. Llama a su branch feature1. Hace un cambio a la iteración principal dejándola como a él le gusta.



```
1  using System;
2  using System.Collections.Generic;
3
4  namespace demogit
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             Console.WriteLine("git conflict solving demo");
11             List<string> myList = new List<string>(){ "hola", "chau"};
12             int i=0;
13             while(i< myList.Count){
14                 Console.WriteLine($"Listando elemento{i} de {myList.Count} : {myList[i]}");
15                 i++;
16             }
17         }
18     }
19 }
20
```

Luego realiza la siguiente actividad para enviar sus cambios del branch feature1 al repositorio remoto y finalmente mergearlo en el master de su copia local.

```
MINGW64:/c/Users/osvaldo/source2/repos2/dummyrepo
no changes added to commit (use "git add" and/or "git commit -a")

osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (feature1)
$ git status
On branch feature1
Your branch is up to date with 'origin/feature1'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

[] arg.modified:  .gitignore
   modified:      Program.cs

no changes added to commit (use "git add" and/or "git commit -a")
osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (feature1)
$ git commit -a
[feature1 b678536] change message on iteration
 2 files changed, 5 insertions(+), 2 deletions(-) (myList[i]);

osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (feature1)
$ git push
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 489 bytes | 489.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/osvaldo-olmos/dummyrepo.git
 792574e..b678536  feature1 -> feature1

osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (feature1)
```

```
osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (feature1)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (master)
$ git pull
Already up to date.
Current branch master is up to date.

osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (master)
$ git merge feature1
Merge made by the 'recursive' strategy.
 .gitignore | 3 ++-
 Program.cs | 4 +++-
 2 files changed, 5 insertions(+), 2 deletions(-)

osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (master)
$
```



Por otro lado está el desarrollador dev2 que se cortó también su branch desde master. Lo llamó feature2. Y también hace cambios a la iteración principal, dejándola de la siguiente manera:

```
C# Program.cs x
1  using System;
2  using System.Collections.Generic;
3
4  namespace demogit
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10              Console.WriteLine("git conflict solving demo");
11              List<string> myList = new List<string>(){ "hola", "chau"};
12              foreach(string element in myList){
13                  Console.WriteLine($"{element}");
14              }
15          }
16      }
17  }
18
```

Luego en git ejecuta los siguientes comandos, a fin de llevar sus cambios al master del repositorio remoto.

```
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (feature2)
$ git add Program.cs

osvaldo@guille-PC MINGW64 ~/source/repos/demogit (feature2)
$ git status
On branch feature2
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   Program.cs

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

  feature1.txt
  bin/
  hola.txt

osvaldo@guille-PC MINGW64 ~/source/repos/demogit (feature2)
$ git commit -m "changed iteration to use a for each"
[feature2 07bc2ff] changed iteration to use a for each
1 file changed, 2 insertions(+), 2 deletions(-)

osvaldo@guille-PC MINGW64 ~/source/repos/demogit (feature2)
```




```
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (feature2)
$ git push origin feature2
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 342 bytes | 342.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/osvaldo-olmos/dummyrepo.git
 * [new branch] feature2 -> feature2

osvaldo@guille-PC MINGW64 ~/source/repos/demogit (feature2)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$ git pull
Already up to date.
Current branch master is up to date.

osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$ |
```

dev 2 mergea su branch feature2 a master local...

```
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$ git merge feature2
Updating 1f54f9c..07bc2ff
Fast-forward
 Program.cs | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)
```

Pero en esos momentos, dev1 pusha su merge al repositorio remoto...

```
osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (master)
$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 282 bytes | 282.00 KiB/s, done.
Total 2 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/osvaldo-olmos/dummyrepo.git
 1f54f9c..485f54b master -> master

osvaldo@guille-PC MINGW64 ~/source2/repos2/dummyrepo (master)
$ |
```

Luego dev2 intenta pushar su copia local de master al repo remoto



```
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$ git push
To https://github.com/osvaldo-olmos/dummyrepo.git
! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/osvaldo-olmos/dummyrepo.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.

osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$
```

Como indica el mensaje en amarillo, hay cambios en la copia de master que está en el repositorio remoto (los cambios que pusheó dev1) que son mas reciente que la última copia que dev2 tiene localmente.

Entonces dev1 hace un git pull para traerse los últimos cambios

```
MINGW64/c/Users/osvaldo/source/repos/demogit
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$ ls
bin/  demogit.csproj  feature1.txt  hola.txt  obj/  Program.cs

osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1), pack-reused 0
Unpacking objects: 100% (2/2), done.
From https://github.com/osvaldo-olmos/dummyrepo
 1f54f9c..485f54b  master      -> origin/master
First, rewinding head to replay your work on top of it...
Applying: changed iteration to use a for each
Using index info to reconstruct a base tree...
M   Program.cs
Falling back to patching base and 3-way merge...
Auto-merging Program.cs
CONFLICT (content): Merge conflict in Program.cs
error: Failed to merge in the changes.
Patch failed at 0001 changed iteration to use a for each
Use 'git am --show-current-patch' to see the failed patch

Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort"
.

osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master|REBASE 1/1)
$
```

Y ve que hay conflictos...



Si vemos el archivo en conflicto con VSCode vemos lo siguiente:

```
Program.cs x
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             Console.WriteLine("git conflict solving demo");
11             List<string> myList = new List<string>(){ "hola", "chau"};
12             <<<<<< HEAD (Current Change)
13                 int i=0;
14                 while(i< myList.Count){
15                     Console.WriteLine($"Listando elemento{i} de {myList.Count} : {myList[i]}");
16                     i++;
17             =====
18                 foreach(string element in myList){
19                     Console.WriteLine($"element");
20             >>>>>> changed iteration to use a for each (Incoming Change)
21                 }
22             }
23         }
24     }
25 }
```

HEAD indica el cambio local nuestro (o el de dev2 en este caso). Incoming change es el cambio que viene desde el repositorio remoto.

Resolvemos los conflictos con el editor de texto del VSCode (u otro editor), finalmente staggeamos el archivo (git -add <filename>) o bien con el + de staging del plugin para repositorios de VSCode).

Luego continuamos con las instrucciones indicadas para terminar el merge:

```
MINGW64/c:/Users/osvaldo/source/repos/demogit
osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master|REBASE 1/1)
$ git status
rebase in progress; onto 485f54b
You are currently rebasing branch 'master' on '485f54b'. Color
  (all conflicts fixed: run "git rebase --continue")

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   Program.cs

osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master|REBASE 1/1)
$ git rebase --continue
Applying: changed iteration to use a for each

osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 356 bytes | 356.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/osvaldo-olmos/dummyrepo.git
  485f54b..4db4b93 master -> master

osvaldo@guille-PC MINGW64 ~/source/repos/demogit (master)
$
```



Listo ! CONFLICTO RESUELTO

Como puede ver, este proceso es bastante tedioso y puede ser extremadamente difícil de tratar en proyectos grandes. La mayoría de los desarrolladores prefieren resolver conflictos con la ayuda de un cliente GUI, lo que hace las cosas mucho más fáciles. Si no quiere usar el plugin de VSCode, puede usar otro cliente gráfico, como ser: git mergetool.

Configurando .gitignore

En la mayoría de los proyectos, hay archivos o carpetas completas que no queremos comitear. Podemos asegurarnos de que no se incluyan accidentalmente en nuestro git add -A al crear un archivo .gitignore:

- Cree manualmente un archivo de texto llamado .gitignore y guárdelo en el directorio de su proyecto.
- En el interior, enumere los nombres de los archivos / directorios que se ignorarán, cada uno en una nueva línea.

El .gitignore en sí tiene que ser stageado, commiteado y pusheado (si estamos trabajando con un repositorio remoto), ya que es como cualquier otro archivo en el proyecto.

Generalmente, se ignoran aquellos archivos o directorios que contienen información local referida a la IDE que se está usando para desarrollar y los binarios que son resultados del buildeo del proyecto.

REFERENCIAS

<https://git-scm.com/book/es/v1/Empezando>
<https://learngitbranching.js.org/>

Fecha	Versión	Observaciones
24/07/2018	1.0	Versión Original