

# Introducción a Git



## Introducción a Git

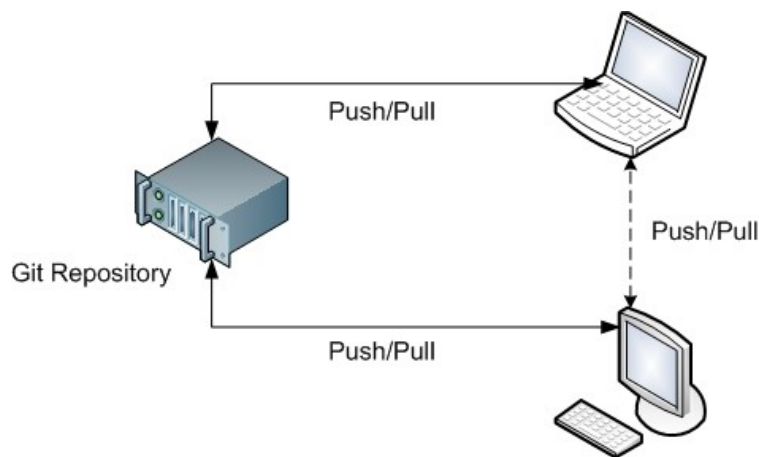
1 Introducción.....	3
2 Repositorios.....	3
3 Push.....	6
4 Pull y conflictos.....	10
5 Ramas (branches).....	15
6 Referencias.....	18

## 1 Introducción

Git es un sistema de control de versiones. El objetivo de un sistema como este es realizar un seguimiento de los cambios que se realizan en los archivos con código fuente de un proyecto cualquiera, de manera que ante un error, por ejemplo, se pueda saber qué nuevo código introducido es el que lo está produciendo, o poder volver a una versión del código antes de que se produjera determinado cambio.

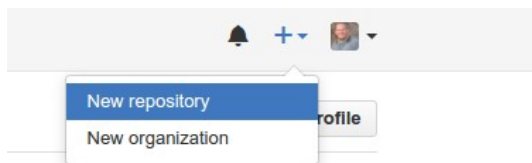
## 2 Repositorios

La forma de trabajar con Git consiste en que haya un repositorio central, y un repositorio local por cada desarrollador del proyecto. Cada desarrollador realiza cambios (*push*), y obtiene los cambios realizados por los otros desarrolladores del repositorio central (*pull*).



Existen múltiples formas de crear el repositorio central. Las más utilizadas, con diferencia, son la utilización de las aplicaciones web *GitHub.com* y *GitLab.com*, por ese orden. En este documento se utilizará la primera por ser la más común.

La primera tarea consiste en crear una cuenta de usuario, proporcionando cierta información básica además de un nombre de usuario y contraseña. Esto es similar a cualquier otra aplicación web, y los detalles no se discutirán aquí.



entornos de programación importantes proporcionan una integración con Git, y *MonoDevelop/XamarinStudio* no es una excepción.

Una vez creada la cuenta de usuario, para abordar un proyecto nuevo es necesario crear un repositorio que contendrá los archivos de código fuente de ese proyecto. Para este ejemplo, se creará una aplicación Hola, mundo, en C#. Como comprobaremos, todos los


### Create a new repository

A repository contains all the files for your project, including the revision history.

---

Owner

Repository name

 Baltasarq ▾

 / 

HolaMundo ✓

Great repository names are short and memorable. Need inspiration? How about [billous-woof](#).

Description (optional)

HolaMundo en C#

---

☒ Public

Anyone can see this repository. You choose who can commit.

☐ Private

You choose who can see and commit to this repository.

---

☒ Initialize this repository with a README

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None ▾

 | 

Add a license: MIT License ▾

 ⓘ

---

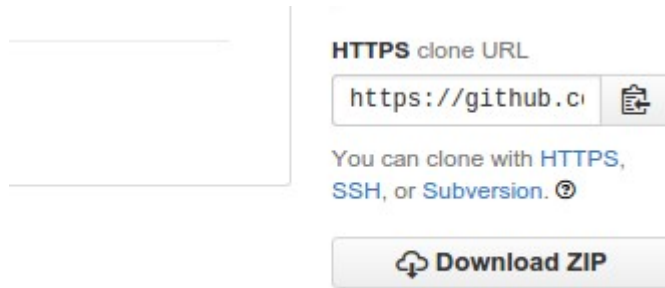
Create repository

Los campos a rellenar son: el nombre del proyecto (*HolaMundo*), una descripción opcional, crear un archivo *README* (léeme), un archivo *.gitignore*, y una licencia, que puede ser GPL, MIT...

El archivo *README.md* es un archivo con formato *markdown* (un pequeño formato de marcado tipo HTML, pero que es perfectamente legible como texto en claro<sup>1</sup>), que explica qué es lo que hace el proyecto. *GitHub* lo creará como un archivo conteniendo el título del proyecto y la descripción. También se creará el archivo *LICENSE* (licencia), que contendrá el texto de la licencia seleccionada (en este caso, MIT que es por otra parte la más permisiva).

El archivo *.gitignore* especifica qué archivos **no** se deben seguir. Lo típico en estos casos es no realizar seguimiento de archivos de *backups* (copias de respaldo), y especialmente, binarios, es decir, ejecutables y librerías. El archivo *.gitignore* lo crearemos nosotros mismos más adelante.

1 Por ejemplo, para remarcar algo se coloca entre asteriscos: Esto es *\*importante\**. Se interpreta fácilmente como negrita en muchos procesadores de texto (o conversores a HTML), y a la vez tiene sentido tal y como se escribe.



Es el momento de crear el repositorio en local, para lo cual utilizaremos el texto en la caja “*Https clone URL*”. Una vez copiado el texto en dicha caja, abrimos un terminal y nos movemos al directorio de donde queremos que cuelgue el proyecto, por ejemplo, *ejercicios/*. La orden para copiar el repositorio remoto en nuestro equipo local es *git clone* y la url copiada. Eso sí, antes de nada será necesario informar a Git de cuáles son nuestros datos. Esto puede hacerse mediante *git config --global user.name <nombre-usuario>* y

*git config --global user.email <email-usuario>*. Solo es necesario hacer esto la primera vez. Los datos introducidos pueden comprobarse tecleando las mismas órdenes sin los datos, es decir, *git config --global user.name* y *git config --global user.email*.

```
$ cd ejercicios
$ git config --global user.name <nombre-usuario>
$ git config --global user.email <email>
```

Ahora sí, realizamos el clonado del repositorio remoto.

```
$ git clone https://github.com/Baltasarq/HolaMundo.git
Cloning into 'HolaMundo'...
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), done.
Checking connectivity... done.
```

Podemos comprobar que tenemos los archivos que fueron creados por *GitHub*.

```
$ cd HolaMundo/
$ ls
LICENSE  README.md

$ ls -a
.  ..  .git  LICENSE  README.md
```

Git ha copiado los archivos en el repositorio, y además podemos comprobar que existe una carpeta *.git* que contiene ciertos archivos internos que necesita. Es el momento de crear el archivo *.gitignore*, que le indicará a **Git** qué archivos son los que debe seguir a base de especificar cuáles no debe tener en cuenta. Cada línea de este archivo es una especificación de máscara: cada archivo que cumpla esta máscara no será incluido. Si se especifica un directorio completo, este no será incluido. Además, una línea empezando por *#* es un comentario. Con cualquier editor de textos, es posible crear dicho archivo con el siguiente contenido, válido para C#:

```
$ cat .gitignore
# Backups
*~
```

```
*.bak

# Binarios
*.exe
*.dll
bin/
obj/

# Preferencias del usuario
*.userprefs
```

La última línea es específica para *MonoDevelop*, de tal forma que diferentes desarrolladores no estén continuamente pisándose sus preferencias de desarrollo personales.

### 3 Push

Es el momento de hacer el primer *push*, es decir pasar nuestras modificaciones al repositorio central (normalmente, se le conoce como repositorio remoto). Lo primero, es indicar qué archivos queremos que Git tenga en cuenta, después realizar un *commit*, y finalmente, *push*. Para saber en qué estado se encuentra el repositorio local, se utiliza la orden *git status*. Para añadir archivos de forma que Git los tenga en cuenta, *git add <nombre\_archivo>*, y finalmente *git commit -a -m "mensaje"* para formalizar el *commit*. Los cambios al repositorio remoto (los *commits*) se suben con *git push*.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    .gitignore

nothing added to commit but untracked files present (use "git add" to track)

$ git add .gitignore

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   .gitignore

$ git commit -a -m "Ignorar binarios"
[master 19d7605] Ignorar binarios
 1 file changed, 12 insertions(+)
 create mode 100644 .gitignore

$ git push
Username for 'https://github.com': baltasarq
Password for 'https://baltasarq@github.com':
Counting objects: 3, done.
```

## Introducción a Git

```
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 391 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/Baltasarq/HolaMundo.git
0fa5fa5..19d7605 master -> master
```

Pero en realidad todavía no hemos hecho nada de código. Desde *MonoDevelop*, crearemos un proyecto de consola en C#, que tendrá el mismo nombre “HolaMundo”, y cuya ubicación será la misma carpeta *ejercicios/*. Es necesario desmarcar la opción “crear el proyecto dentro del directorio de la solución”, ya que esto crearía el proyecto en la carpeta *HolaMundo* dentro de *HolaMundo*, es decir *ejercicios/HolaMundo/HolaMundo*. También es necesario desmarcar la opción “utilizar git para control de versiones”, ya que el trabajo que realizaría *MonoDevelop* ya ha sido llevado a cabo por nosotros, y no haría más que borrarlo.

Al crear el proyecto, se quejará sobre la carpeta de destino, que ya existe. Ignoramos este mensaje (es decir, continuamos), y ejecutamos la aplicación de plantilla, que ya incorpora un pequeño *Hola, Mundo!* en inglés. Volvemos a la línea de comando:

### **\$ git status**

```
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

```
    HolaMundo.csproj
    HolaMundo.sln
    Program.cs
    Properties/
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

### **\$ git add --all**

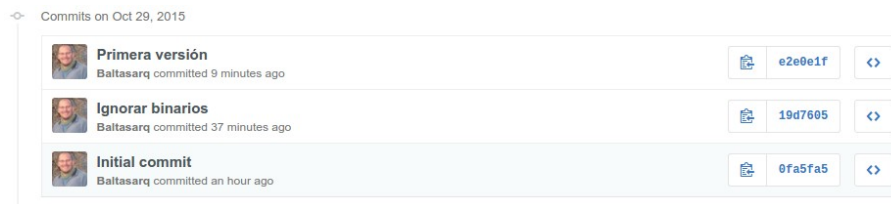
### **\$ git commit -a -m"Primera versión"**

```
[master e2e0e1f] Primera versión
 4 files changed, 94 insertions(+)
 create mode 100644 HolaMundo.csproj
 create mode 100644 HolaMundo.sln
 create mode 100644 Program.cs
 create mode 100644 Properties/AssemblyInfo.cs
baltasarq@PC-baltasarq:~/tmp/ejercicios/HolaMundo$ 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 directory clean
baltasarq@PC-baltasarq:~/tmp/ejercicios/HolaMundo$ git push
Counting objects: 7, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (7/7), 1.99 KiB | 0 bytes/s, done.
Total 7 (delta 0), reused 0 (delta 0)
To https://github.com/Baltasarq/HolaMundo.git
```

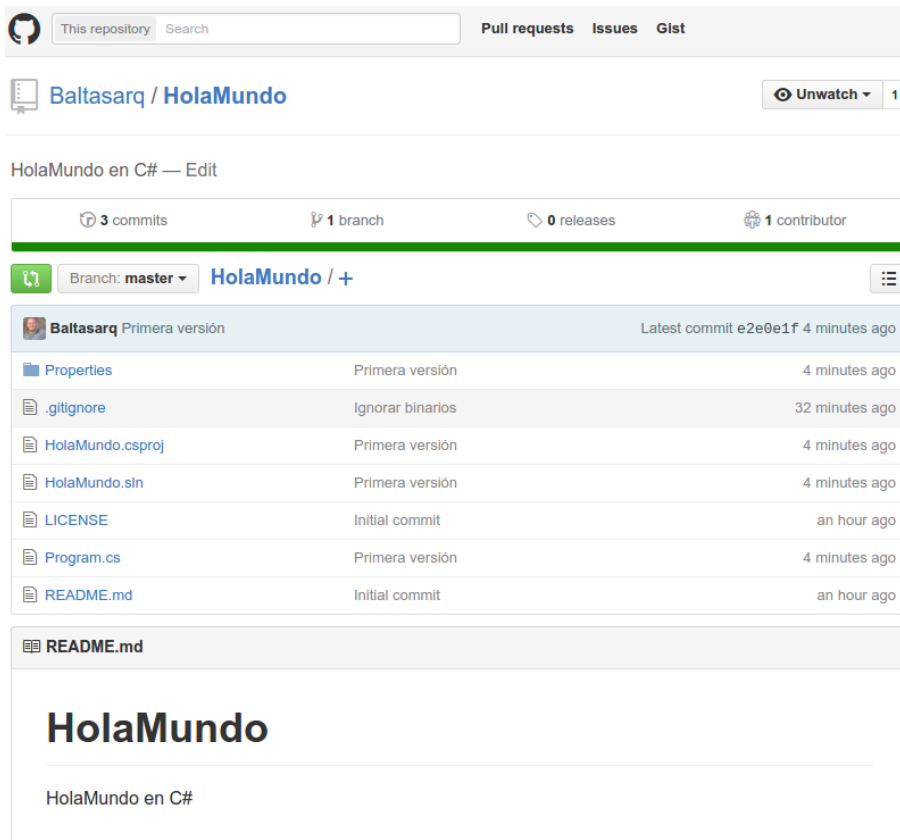
## Introducción a Git

```
19d7605..e2e0e1f master -> master
```

Efectivamente, hemos incluido todos los archivos generados por *MonoDevelop* para el proyecto con *git add -all*, y realizar el commit como siempre (*git commit -a -m "Primera versión"*). Los cambios se suben con *git push*, y se pueden comprobar desde *GitHub.com*.



Todos los archivos del proyecto figuran ya en *GitHub.com*. Se puede observar el proceso seguido para llegar al estado actual. Por ejemplo, *README.md* se creó con el commit “Initial Commit”, que fue establecido por el propio *GitHub.com* al crear el repositorio. El archivo *.gitignore* se creó con el commit “Ignorar binarios”, mientras que el resto de archivos ya pertenece al último commit: “Primera versión”. Así, en la parte superior se observa la leyenda “3 commits”. Al pulsar sobre ella, se abre un historial de versiones. El botón <> a la derecha de cada una de esas versiones, permite hojear los archivos de ese repositorio tal y como estaban en ese momento.

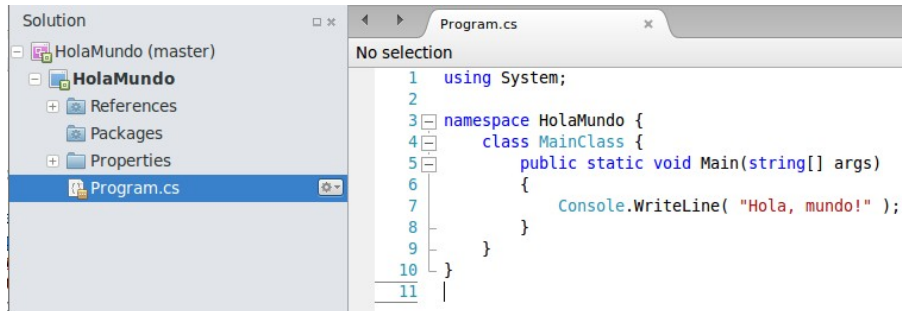


*MonoDevelop* ofrece integración con Git como control de versiones, si bien en Linux será necesario instalarlo aparte con el paquete *monodevelop-versioncontrol*. Por ejemplo, desde cualquier distribución basada en Debian: *sudo apt install monodevelop-versioncontrol*, desde el terminal.

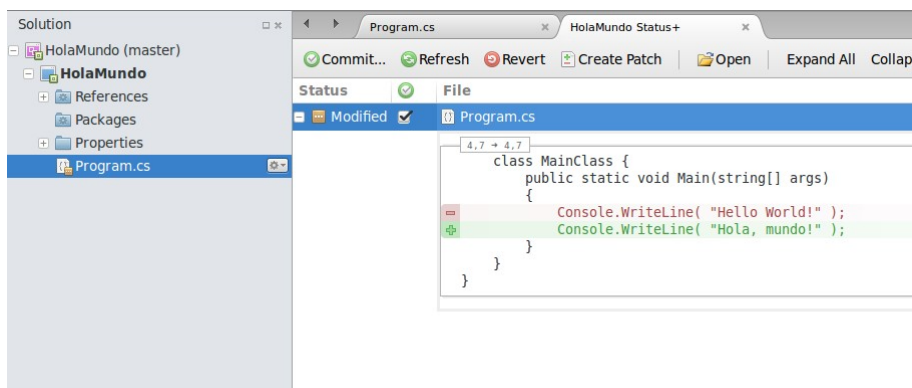
Para comprobar cómo funciona desde *MonoDevelop* el control de versiones, modificaremos el mensaje que se visualiza por pantalla, es decir, modificaremos el método *Main()* de la clase



**MainClass.** En lugar de “Hello, World!”, teclearemos “Hola, mundo!”, y guardamos el archivo. Inmediatamente, MonoDevelop añade un símbolo al archivo en el árbol de la solución que indica que el archivo ha cambiado.



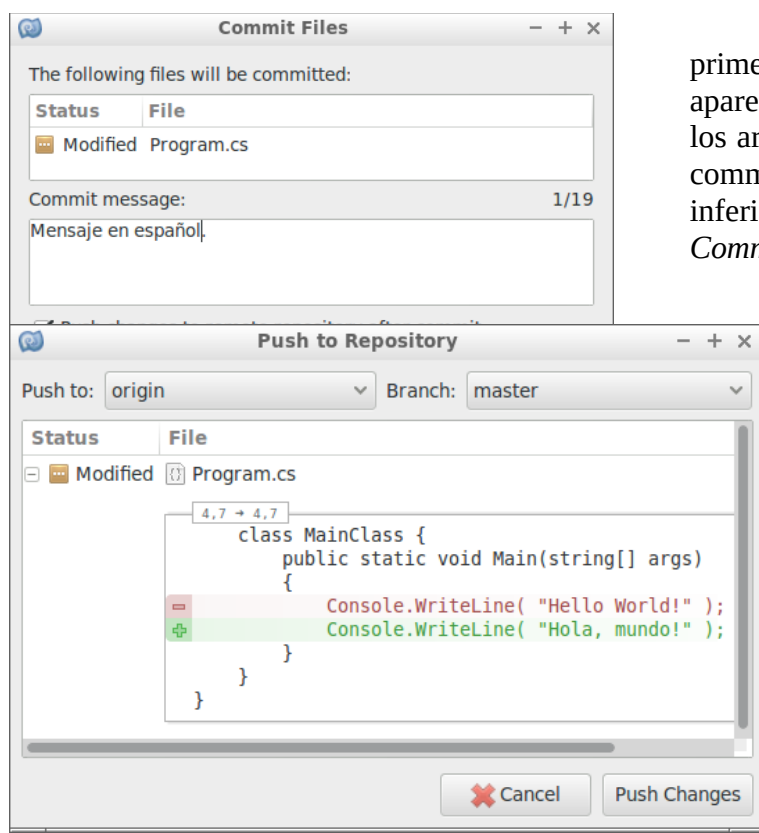
Pulsando con el botón derecho en el árbol de la solución, y seleccionando *Version Control >> Review and Commit* (o su equivalente *Control de Versiones >> Revisar y proteger* si está traducido), se accede a la pantalla de *commit*.



Para realizar el *commit*, pulsamos en el primer botón, “Commit”, o “Proteger”. Entonces aparece un cuadro de diálogo que permite revisar los archivos con cambios, y añadir el mensaje del commit. Debe marcarse la opción en la parte inferior “push changes ...” para que tras pulsar en *Commit* se abra el diálogo relativo al *push*.

En la siguiente ventana, se puede escoger la rama en la que hacer el *push* (el concepto de rama se explicará más adelante). Finalmente, pulsar en el botón “Push changes” repercutirá los cambios en el repositorio remoto, pudiendo comprobarlos desde *GitHub.com*.

Evidentemente, el proceso es mucho

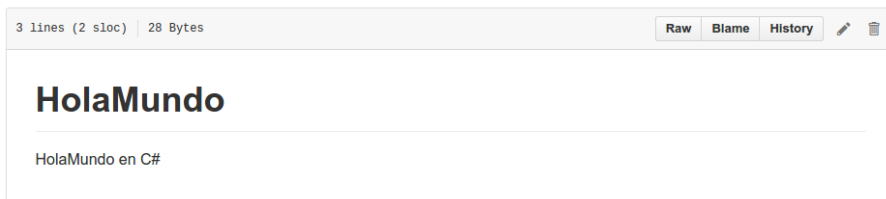


más sencillo y visual que el realizado desde el terminal, aunque como es obvio el resultado es exactamente el mismo. Nótese que las secciones *diff* permiten visualizar los cambios (las líneas precedidas con + son adiciones, y las precedidas con – son eliminaciones). Además, las secciones *blame* permiten saber quién hizo cada modificación en un archivo determinado.

## 4 Pull y conflictos

La operación *pull* es la contraria a *push*: permite traer los cambios que se hayan realizado en el repositorio desde la última vez que se trabajó en el proyecto. En *MonoDevelop*, esto se puede realizar desde la opción “Update solution” o “Actualizar solución”. La orden desde el terminal es simplemente *git pull*.

```
$ git pull
Already up-to-date.
```



Git indica que todo está actualizado, lo cual no es extraño teniendo en cuenta que en este ejemplo solamente hay un desarrollador trabajando contra el repositorio central. Sin embargo, es

posible cambiar algo desde *GitHub.com*, por ejemplo el archivo *README.md*. Solo es necesario seleccionarlo y pulsar sobre el lápiz para realizar cualquier modificación, por ejemplo, añadirle la línea “Ejemplo con Git”. Una vez realizado el cambio, y al volver al terminal, podemos utilizar *git pull* para sincronizar los repositorios. Es interesante, sin embargo, comprobar qué pasa cuando el repositorio remoto tiene cambios que no han sido reflejados en local. Para ello, cambiaremos también el mensaje “Hola, mundo!”, por “Hola, nuevo mundo!” en el *Program.cs* local. Git se negará a realizar el *push*, debido a esos cambios pendientes en remoto.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
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)

        modified:   Program.cs
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

```
$ git commit -a -m"Nuevo mensaje"
[master d2c2000] Nuevo mensaje
1 file changed, 1 insertion(+), 1 deletion(-)
```

```
$ git push
To https://github.com/Baltasarq/HolaMundo.git
! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/Baltasarq/HolaMundo.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
```

## Introducción a Git

hint: (e.g., 'git pull ...') before pushing again.  
hint: See the 'Note about fast-forwards' in 'git push --help' for details.

### **\$ git pull**

```
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.
From https://github.com/Baltasarq/HolaMundo
   15fbad5..b5fc0bb  master    -> origin/master
Merge made by the 'recursive' strategy.
 README.md | 2 ++
 1 file changed, 2 insertions(+)
```

### **\$ git push**

```
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 560 bytes | 0 bytes/s, done.
Total 5 (delta 3), reused 0 (delta 0)
To https://github.com/Baltasarq/HolaMundo.git
   b5fc0bb..6fc8a88  master -> master
```

### **\$ git status**

```
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

En este caso, el conflicto se ha resuelto de forma sencilla (simplemente, anteponiendo un *git pull* antes de volver a hacer *git push*), debido a que los cambios se habían hecho en distintos archivos. En el caso en el que los cambios en local y en remoto afecten al mismo archivo, será necesaria la intervención del programador, que debe manualmente discriminar qué es lo que no desea conservar. En este caso, modificaremos el archivo Program.cs en local para utilizar una variable intermedia, y en remoto para utilizar el método *Console.WriteLine()* en lugar de *Console.WriteLine()*.

El archivo Program.cs modificado desde MonoDevelop quedaría:

```
using System;

namespace HolaMundo {
    class MainClass {
        public static void Main(string[] args)
        {
            string msg = "Hola, nuevo mundo!";
            Console.WriteLine( msg );
        }
    }
}
```

Mientras que desde *GitHub.com* lo modificaremos de la siguiente forma:

```
using System;

namespace HolaMundo {
    class MainClass {
        public static void Main(string[] args)
        {
            Console.Write( "Hola, nuevo mundo!\n" );
        }
    }
}
```

Al intentar hacer un push desde el terminal, o incluso desde MonoDevelop, la respuesta será la misma: “The update is a non-fast-forward update. Merge the remote changes before pushing again.”, es decir, que es necesario mezclar los cambios antes de poder hacer nada.

```
$ git commit -a -m"Variable intermedia"
```

```
$ git push
```

```
To https://github.com/Baltasarq/HolaMundo.git
! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/Baltasarq/HolaMundo.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.
```

```
$ git pull
```

```
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/Baltasarq/HolaMundo
 6fc8a88..13be8e1 master    -> origin/master
Auto-merging Program.cs
CONFLICT (content): Merge conflict in Program.cs
Automatic merge failed; fix conflicts and then commit the result.
```

Git informa de que ha hecho el cambio a medias: el punto final debemos resolverlo nosotros, pues no puede saber qué cambios tienen prioridad en un mismo archivo. Si volvemos a MonoDevelop (o incluso desde cualquier editor de texto), el archivo tendrá el siguiente aspecto.

```
using System;

namespace HolaMundo {
    class MainClass {
        public static void Main(string[] args)
        {
<<<<<<< HEAD
                string msg = "Hola, nuevo mundo!";
                Console.WriteLine( msg );

=====
                Console.Write( "Hola, nuevo mundo!\n" );
>>>>>>> 13be8e1c774b3be715d205463b05ba06a36ee3b1
        }
    }
}
```

La parte encabezada por <<< HEAD es el cambio en el repositorio local. La parte terminada por >>> y un código SHA5 es la residente en remoto. Supongamos que elegimos realizar una mezcla de ambas soluciones, editándolo el manualmente, y guardándolo.

```
using System;

namespace HolaMundo {
    class MainClass {
        public static void Main(string[] args)
        {
            string msg = "Hola, nuevo mundo!";
            Console.Write( msg + "\n" );
        }
    }
}
```

Ahora solamente es necesario realizar un *commit* y un *push*, para que los cambios sean mezclados.

### **\$ git status**

```
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commit each, respectively.
(use "git pull" to merge the remote branch into yours)
You have unmerged paths.
(fix conflicts and run "git commit")
```

```
Unmerged paths:
(use "git add <file>..." to mark resolution)
```

```
    both modified:   Program.cs
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

```
$ git commit -a -m"Modificaciones mezcladas"
```

```
[master 4dea53d] Modificaciones mezcladas
```

```
$ git push
```

```
Counting objects: 6, done.
```

```
Delta compression using up to 8 threads.
```

```
Compressing objects: 100% (6/6), done.
```

```
Writing objects: 100% (6/6), 692 bytes | 0 bytes/s, done.
```

```
Total 6 (delta 3), reused 0 (delta 0)
```

```
To https://github.com/Baltasarq/HolaMundo.git
```

```
13be8e1..4dea53d master -> master
```

```
$ git status
```

```
On branch master
```

```
Your branch is up-to-date with 'origin/master'.
```

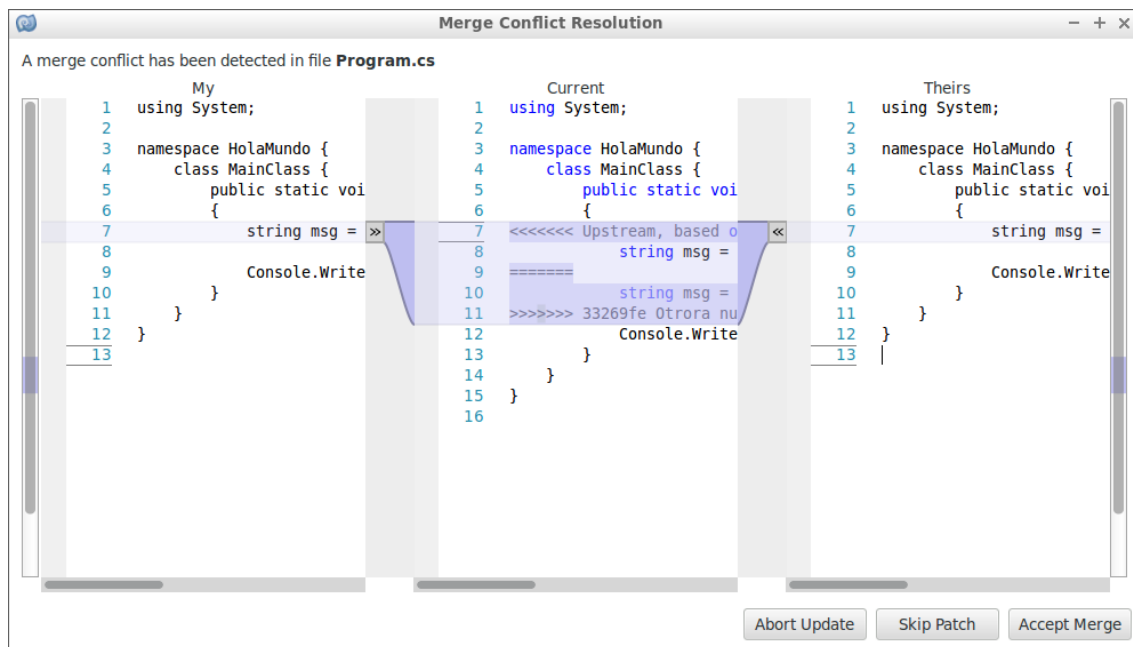
```
nothing to commit, working directory clean
```

```
$ git pull
```

```
Already up-to-date.
```

Ahora los cambios han sido mezclados correctamente, y el repositorio local y el repositorio remoto están sincronizados.

Si, en cambio, este proceso se realiza desde *MonoDevelop* este se comporta de manera similar, pues una vez que *push* falla, si se realiza un *pull* (*Version Control >> Project update*), entonces por cada archivo con cambios que no se puedan automatizar se abre una ventana similar a la siguiente, donde se pueden mezclar ambas versiones. Tras pulsar en “Accept merge”, el cambio queda almacenado de manera que en el siguiente *push* se sincronicen ambos repositorios.



## 5 Ramas (branches)

Hasta ahora, se ha trabajado sobre la misma rama continuamente (*master*). Sin embargo, esto no es lo más habitual. Lo normal es que cada desarrollador trabaje en una nueva característica distinta, de manera que el software quede sin modificar en su versión “estable” anterior. Para ello, cada desarrollador debe trabajar en una rama distinta. La opción *branch* es la que se encarga de informar de las ramas presentes en el desarrollo, la actual precedida de un asterisco. La rama por defecto siempre es *master*. Para saber las ramas que hay, *git branch*. Para crear una nueva rama, se utiliza *git checkout -b <nombre-rama>*. La nueva rama siempre es igual a la rama actual. Para cambiar entre ramas, se utiliza *git checkout <nombre-rama>*.

En el siguiente ejemplo, se modifica *Program.cs* para crear una característica nueva, que es una mejorada forma de mostrar información por pantalla.

```
$ git branch
```

```
* master
```

```
$ git checkout -b feature-output
```

```
Switched to a new branch 'feature-output'
```

```
$ git branch
```

```
* feature-output  
master
```

```
$ ls
```

```
HolaMundo.csproj  HolaMundo.sln  LICENSE  Program.cs  Properties  README.md
```

En este punto, se modifica el archivo *Program.cs* para que incorpore la nueva característica.

```
$ cat Program.cs
```

```
using System;
```

```
namespace HolaMundo {  
    class MainClass {  
        public static void Main(string[] args)  
        {  
            string msg = "Hola, otrora nuevo mundo!";  
  
            Output( msg );  
        }  
  
        public static void Output(string msg) {  
            Console.Write( msg + "\n" );  
        }  
    }  
}
```

```
$ git status
```

```
En la rama feature-output
```

```
Cambios no preparados para el commit:
```

```
(use «git add <archivo>...» para actualizar lo que se ejecutará)
```

```
(use «git checkout -- <archivo>...» para descartar cambios en le directorio de
```



## Introducción a Git

trabajo)

```
modified: Program.cs
```

no hay cambios agregados al commit (use «git add» o «git commit -a»)

```
$ git commit -a -m"Output implementado"
[feature-output 3d26c4c] Output implementado
1 file changed, 5 insertions(+)
```

```
$ git push
```

```
fatal: The current branch feature-output has no upstream branch.
To push the current branch and set the remote as upstream, use
```

```
git push --set-upstream origin feature-output
```

Efectivamente, el problema es que la rama creada en local, *feature-output*, no existe en el repositorio remoto. Es necesario por tanto hacer un *push* especial, de manera que cree la rama y realice los cambios.

```
$ git push --set-upstream origin feature-output
Username for 'https://github.com': baltasarq
Password for 'https://baltasarq@github.com':
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 370 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To https://github.com/Baltasarq/HolaMundo.git
 * [new branch] feature-output -> feature-output
Branch feature-output set up to track remote branch feature-output from origin.
```

En este punto, la rama *feature-output* ya tiene todos los cambios que deseábamos, y todo funciona correctamente.

```
$ git checkout master
```

```
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

```
$ git branch
```

```
feature-output
* master
```

```
$ cat Program.cs
```

```
using System;

namespace HolaMundo {
    class MainClass {
        public static void Main(string[] args)
        {
            string msg = "Hola, otrora nuevo mundo!";
            Console.Write( msg + "\n" );
        }
    }
}
```

```
}  
}
```

La rama ha sido creada y el *commit* realizado, y al volver a la rama *máster* podemos comprobar que el contenido de *Program.cs* sigue siendo el que era antes del cambio. Ahora que la nueva característica ha sido creada y probada, es necesario mezclar ambas ramas para traer la nueva característica a *master*. Para ello, se utiliza *git merge <nombre-de-rama>* desde *master*.

```
$ git branch
```

```
feature-output  
* master
```

```
$ git merge feature-output
```

```
Updating f6177ab..3d26c4c
```

```
Fast-forward
```

```
Program.cs | 5 +++++
```

```
1 file changed, 5 insertions(+)
```

```
jbgarcia@ESEI:~/Documentos/DIA/HolaMundo$ git status
```

```
En la rama master
```

```
Su rama está delante de «origin/master» para 1 commit.
```

```
(use "git push" to publish your local commits)
```

```
nothing to commit, working directory clean
```

```
$ git push
```

```
Username for 'https://github.com': baltasarq
```

```
Password for 'https://baltasarq@github.com':
```

```
Total 0 (delta 0), reused 0 (delta 0)
```

```
To https://github.com/Baltasarq/HolaMundo.git
```

```
f6177ab..3d26c4c master -> master
```

## 6 Referencias

- Git:  
<http://www.git-scm.com/>
- Markdown:  
<http://es.wikipedia.org/wiki/Markdown>