

Sistemas de control de Versións

Despregamento de Aplicacións Web

IES de Teis
Marta Rey López
Curso 2023-2024

Traballo derivado por Marta Rey López con Licenza CC: BY-NC-SA (*) a partir dos documentos orixinais:

© Xunta de Galicia. Consellería de Cultura, Educación e Ordenación Universitaria.

Autores: María del Carmen Fernández Lameiro, Máximo Fernández López Andrés del Río Rodríguez. Licenza CC: BY-NC-SA (*)

Capítulo Git: Autor: Fernando Rodríguez Diéguez con Licenza Creative Commons BY-NC-SA (*)

(*) Licenza CC: BY-NC-SA: Creative Commons Reconecemento - Non Comercial - Compartir Igual. Para ver unha copia desta licenza, visitar a ligazón <http://creativecommons.org/licenses/by-nc-sa/3.0/es/>

Índice

1.	Sistemas de control de versións.	3
1.1	Introdución	3
1.2	Actividade	3
1.2.1	Introdución aos sistemas de control de versións	3
1.2.1.1	Características	3
1.2.2	Terminoloxía	4
1.2.3	Clasificación	4
1.2.3.1	Modelo cliente-servidor	4
1.2.3.2	Modelo distribuído	5
1.2.4	Software de control de versións	5
1.2.4.1	Software libre	5
	Modelo cliente-servidor	5
	Modelo distribuído	5
1.2.4.2	Software propietario	5
	Modelo cliente-servidor	5
	Modelo distribuído	5
	Diferenzas	6
2.	Git	6
2.1	Descrición	6
2.1.1	Terminoloxía	6
	As tres zonas	6
	Os tres estados	7
	Punteiros	8
	Boas prácticas: “Git Flow”, “GitHub Flow”, etc.	8
2.2	Instalación Git en Windows	10
2.3	Creando o repositorio	12
2.4	Operacións básicas	13
	Engadir arquivos	13
	Diferenzas entre arquivos	15
	Desfacendo cambios	15
2.5	Traballando con ramas	17
2.6	Resolución de erros e conflitos	19
	Ver diferenzas	19
2.7	Repositorios remotos	20
	Conexión remota	23
2.8	Outros comandos	26

1. Sistemas de control de versións.

1.1 Introducción

Na actividade que nos ocupa aprenderanse os seguintes conceptos e manexo de destrezas sobre os sistemas de control de versións:

- Instalación nun IDE.
- Creación e actualización de repositorios.
- Accesibilidade e seguridade.

1.2 Actividade

1.2.1 Introducción aos sistemas de control de versións

Chámase control de versións á xestión dos diversos cambios que se realizan sobre os elementos dalgún produto ou unha configuración do mesmo.

Unha versión, revisión ou edición dun produto, é o estado estable no que se atopa o devandito produto nun momento dado do seu desenvolvemento ou modificación.

Os programas de control de versións permiten traballar de forma conxunta a un grupo de persoas no desenvolvemento de proxectos normalmente a través de Internet.

O control de versións se leva a cabo principalmente na industria informática para controlar as distintas versións do código fonte pero tamén se aplica noutros ámbitos como documentación, imaxes, sitios web, e en xeral en calquera proxecto colaborativo que requira traballar con equipos de persoas de forma concorrente.

O control de versións de código está integrado no proceso de desenvolvemento de software de moitas empresas sobre todo se teñen máis dun programador traballando no mesmo proxecto.

1.2.1.1 Características

Os programas de control de versións realizan funcións indispensables durante a vida dun proxecto entre as que destacan:

- Permitir o control dos usuarios que traballarán en paralelo no proxecto:
 - Establecer os usuarios que terán acceso.
 - Asignarlles o tipo de acceso.
- Con relación aos ficheiros:
 - Permitir o almacenamento dos ficheiros.
 - Permitir realizar cambios sobre os ficheiros almacenados: modificar parcialmente un arquivo, borrarlo, cambiarlle o nome, movelo,...
 - Dispor dun histórico detallado (cambios, data, motivo, usuario...) das accións realizadas no tempo.
- Con relación ás versións:

- Etiquetar os arquivos nun punto determinado do desenvolvemento do proxecto para sinalar unha versión estable e así poder identificala posteriormente mediante esa etiqueta.
 - Dispor dun histórico detallado das versións (etiqueta, usuario responsable, data,...).
 - Permitir a recuperación de todos ou algún dos arquivos dunha versión.
 - Comparar versións tendo unha vista ou informe dos cambios entre elas.
- Con relación ao proxecto, permitir a creación de ramas, é dicir, bifurcar o proxecto en dous ou máis liñas que poden evolucionar paralelamente por separado. As ramas poden utilizarse para ensaiar novas características de forma independente sen perturbar a liña principal do desenvolvemento. Se as novas características son estables a rama de novo desenvolvemento pode ser fusionada coa rama principal ou tronco.

1.2.2 Terminoloxía

- **Repositorio.** Os sistemas de control de versións teñen un repositorio para cada proxecto, é dicir, un lugar no que se almacenan todos os arquivos do proxecto.
- **Importación (import).** Esta operación permite subir un proxecto non versionado existente no computador do usuario a un repositorio do servidor por primeira vez.
- **Despregamento (checkout).** Esta operación permite que un usuario poida crear un directorio de traballo no seu disco duro local cunha copia dunha versión dun repositorio, normalmente a última.
- **Publicar (commit ou check-in).** Esta operación permite actualizar o contido dun repositorio cos cambios do directorio de traballo local.
- **Sincronizar (update).** Esta operación permite actualizar o directorio de traballo (ten que existir) cos cambios (arquivos modificados, directorio novos, arquivos novos, directorios que quedan baleiros no directorio, ...) realizados no repositorio desde a última actualización.
- **Rotular (tag).** Esta operación permite etiquetar unha versión, é dicir, darlle un nome ao estado actual do proxecto, o que permite que nun determinado momento se volva a versións anteriores empregando este nome.
- **Abrir rama (branch).** Esta operación permite bifurcar o proxecto en ramas que levarán unha evolución paralela do código de tal maneira que en calquera momento se poida realizarse unha fusión de cambios entre elas.
- **Integración o fusión (merge).** Esta operación permite aplicar todos os cambios realizados nunha rama a outra rama calquera do repositorio. A operación **patch** realiza o mesmo que merge pero límitase a modificacións en ficheiros mentres que merge permite directorios e ficheiros.
- **Cambios (diff).** Antes de enviar os cambios ao repositorio, pode resultar interesante ver os cambios que se realizaron con relación ao repositorio.

1.2.3 Clasificación

1.2.3.1 Modelo cliente-servidor

Existe un repositorio centralizado de todo o código nun servidor, cun usuario responsable, ao que acceden os usuarios autorizados mediante un cliente. Facilítanse as tarefas administrativas

a cambio de reducir flexibilidade, pois todas as decisións fortes (como crear unha nova rama) necesitan a aprobación do responsable.

1.2.3.2 Modelo distribuído

Cada usuario traballa directamente co seu repositorio local que fai ao mesmo tempo de cliente e de servidor. Non é necesario tomar decisións de forma centralizada. Os distintos repositorios poden intercambiar e mesturar revisións entre eles.

1.2.4 Software de control de versións

1.2.4.1 Software libre

Modelo cliente-servidor

- CVS (Concurrent Version System <http://www.cvshome.org>): Desenvolvido por GNU. Distribúese baixo licenza GPL.
- SVN (Subversion <http://subversion.apache.org>): Hoxe é o máis popular. Foi creado para mellorar CVS sobre todo no manexo de arquivos binarios. Distribúese baixo licenza Apache/BSD.

Modelo distribuído

- Git (<http://git-scm.com>): Diseñado por Linus Torvalds, está baseado en BitKeeper e Monotone. Distribúese baixo licenza GNU GPL v2. É usado para o proxecto de programación do kernel de Linux.
- Mercurial (<http://mercurial.selenic.com/wiki>): Creado por Matt Mackall. Distribúese baixo licenza GNU GPL.
- Bazaar (<http://wiki.bazaar.canonical.com>): Desenvolvido por Canonical LTF e comunidade. Facilita proxectos de software libre e opensource. Distribúese baixo licenza GNU GPL v2.

1.2.4.2 Software propietario

Modelo cliente-servidor

- Visual SourceSafe : Ferramenta de control de versións de Microsoft orientada a equipos pequenos que se integra na contorna de traballo de Visual Studio e no resto de ferramentas de desenvolvemento de Microsoft. Actualmente está a ser substituído por Visual Studio Team Foundation Server(<http://www.visualstudio.com/es-es>).

Modelo distribuído

- BitKeeper (<http://www.bitkeeper.com/>) producido por Bitmover Inc.

Diferenzas

	SVN	Git
Modelo	Centralizado	Distribuído
Repositorio	Un repositorio central. É preciso ter conexión co repositorio.	Copias locais coas que se traballa directamente. Non é necesaria conexión para traballar.
Acceso	Depende da ruta de acceso	A todo o directorio.
Seguimento	Baseado en arquivos	Baseado en contido
Historial	So no repositorio completo	No repositorio central e no local
Conectividade	En cada acceso	So ao sincronizar

2. Git

2.1 Descrición

Git está pensado para que cada persoa teña o seu propio repositorio local e suban os cambios a un repositorio remoto de forma que todo o traballo sexa colaborativo e non sexa necesario obrigarse a traballar directamente sobre o repositorio remoto. O repositorio local de git chámase Git e un posible repositorio remoto chámase GitHub. A parte de GitHub hai máis repositorios remotos que implementan Git, como pode ser BitBucket. Cada un ofrece diferentes características.

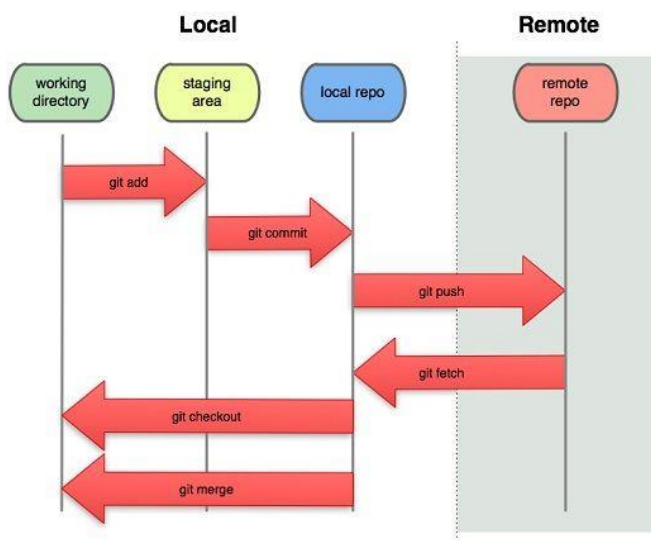
- A meirande parte das operacións son locais, o que fai que funcione moi rápido e que faga posible traballar sen conexión.
- Comproba todos os datos mediante SHA1, polo que destaca no mantemento da integridade da información.
- Xeralmente só engade datos. Case nunca borra nada, polo tanto, todo queda rexistrado e todo é recuperable.
- A diferenza doutros sistemas de control de versións, Git almacena instantáneas dos arquivos, e non as diferenzas que se van producindo neles.

2.1.1 Terminoloxía

As tres zonas

- Lugar de traballo (working directory), onde se levan a cabo as modificacións.
- Área intermedia (staging area), onde se atopan as modificacións que despois serán publicadas (commit). Git leva a cabo un seguimento dos arquivos antes de confirmar.

- Área confirmada (local repo). Unha vez que se levou a cabo a publicación (commit), temos no repositorio unha nova versión, polo que pasaremos automaticamente ao lugar de traballo para facer o seguinte ciclo de traballo.
- Repositorio remoto (remote repo). Pode ser outro repositorio Git que teñamos noutro PC ou un repositorio na nube como pode ser GitHub ou BitBucket. Aquí será onde se realicen os merge con outros usuarios que colaboren no proxecto.

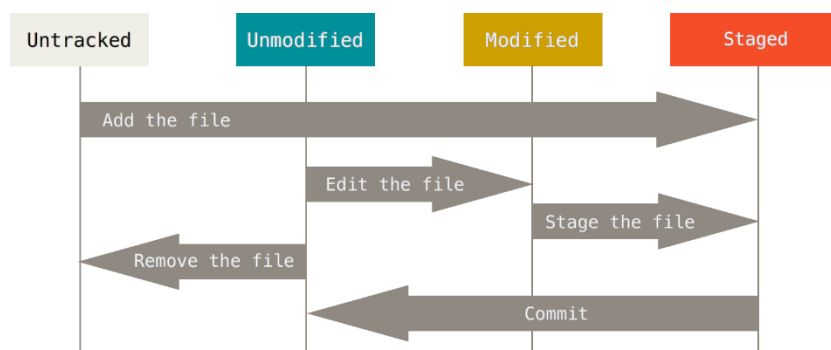


Os tres estados

Existen tres estados, un por cada zona:

- **Confirmado/non modificado** (committed), os datos están almacenados de forma segura no repositorio.
- **Modificado** (modified), modificouse o arquivo pero aínda non se confirmou.
- **Preparado** (staged), preparado para a publicación (commit). Unha vez feita a publicación o estado pasaría a ser confirmado.

Existe un estado adicional, o dos ficheiros que aínda non están engadidos ao repositorio: **untracked**.



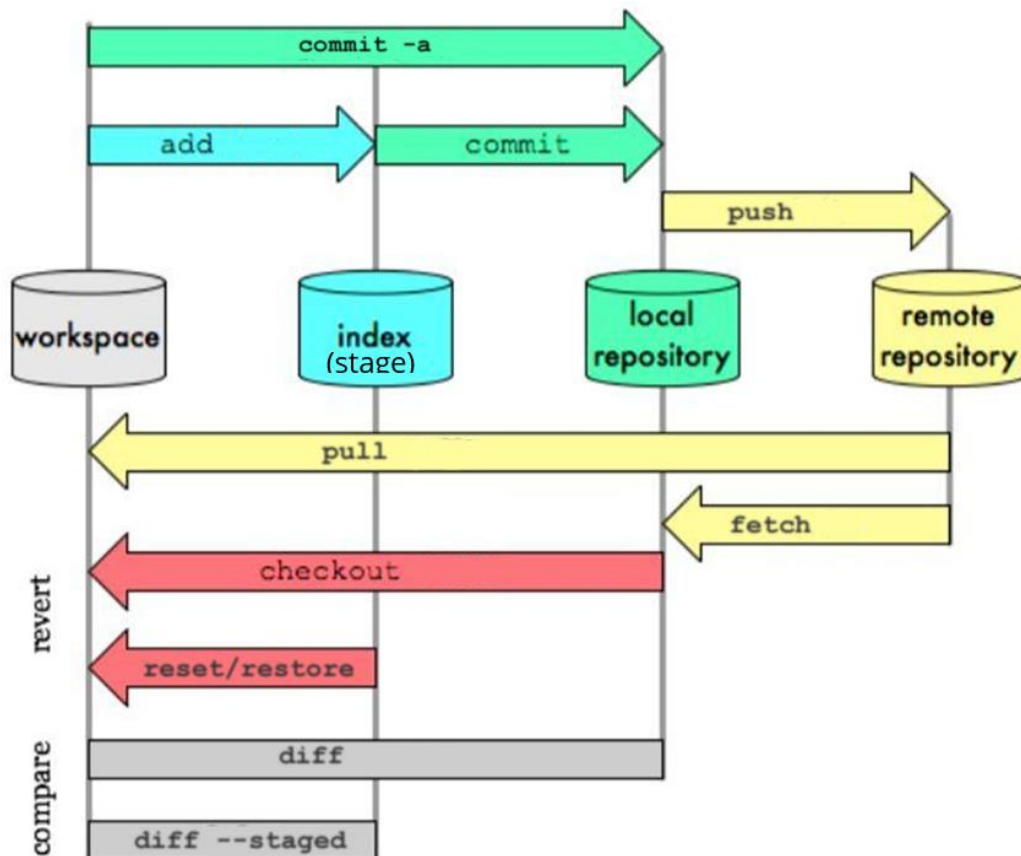
Os estados de Git <http://progit.org/book/ch2-2.html> (CC-BY-NC-SA 3.0)

Punteiros

Cada rama ten o seu propio punteiro que apunta ao último commit de dita rama. O punteiro HEAD apunta ao último commit da rama na que estamos situados.

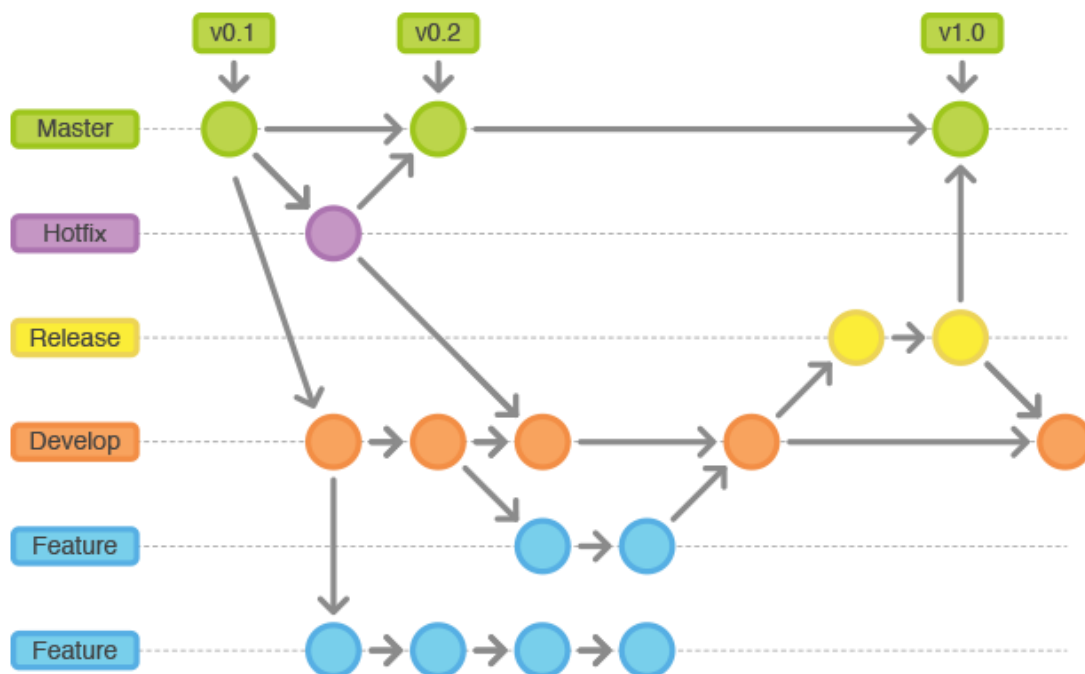


Nesta ligazón podes atopar un tutorial de introdución a Git por liña de comandos moi sinxelo <http://rogerdudler.github.io/git-guide/index.es.html>



Boas prácticas: “Git Flow”, “GitHub Flow”, etc...

Existen diferentes modelos de desenvolvemento de versións. Un típico é o “Git Flow” que se basea nas liñas que se ven na seguinte figura:



Cada desenvolvedor ou equipo de desenvolvemento pode facer uso de Git da forma que lle pareza máis adecuada. Porén, unha boa práctica é a seguinte:

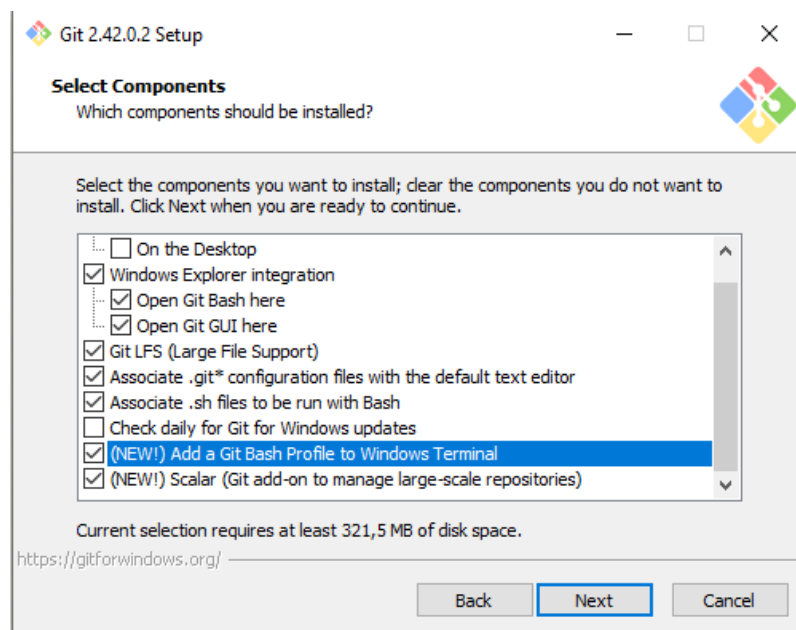
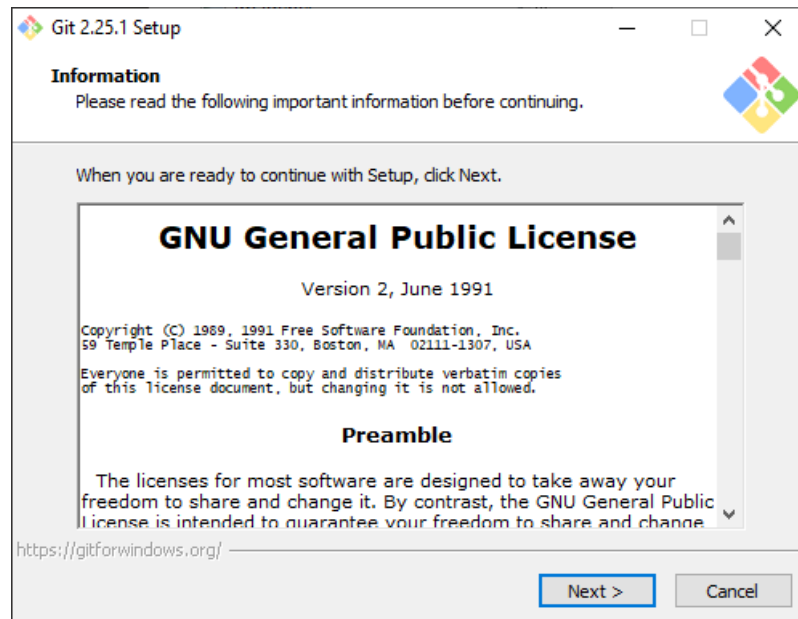
Débense utilizar 4 tipos de ramas: Master, Development, Features, y Hotfix.

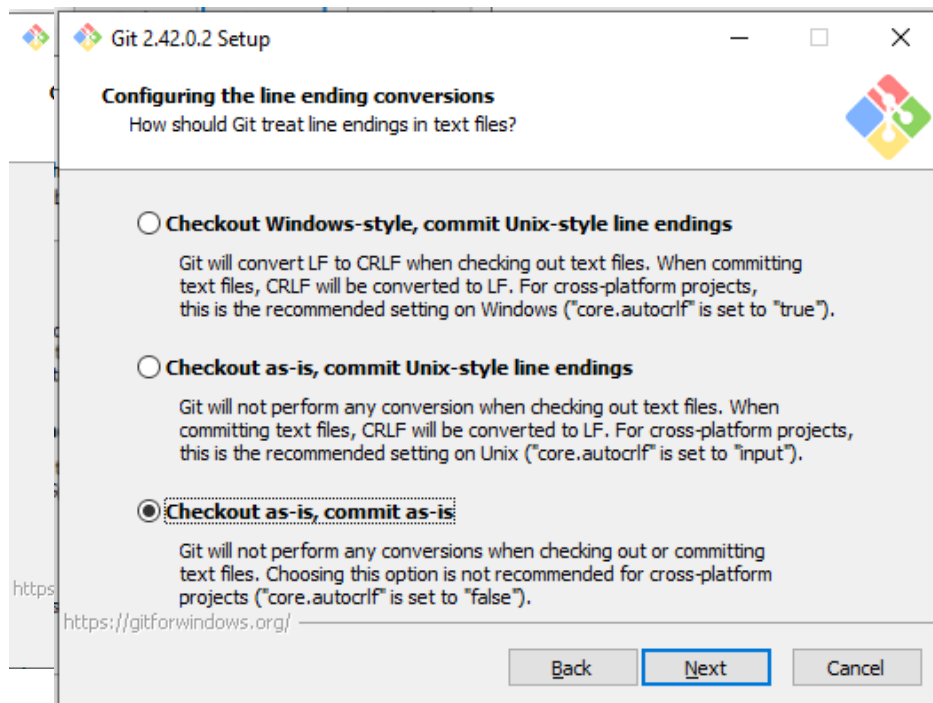
- *Master*: É a rama principal. Contén o repositorio que se atopa publicado en produción, polo que debe estar sempre estable.
- *Development*: É unha rama sacada de ‘master’. É a rama de integración, todas as novas funcionalidades débense integrar nesta rama. Logo de que se realice a integración e se corrixan los erros (no caso de haber algún), é dicir, que a rama se atope estable, pódese facer un *merge* de ‘development’ sobre a rama ‘master’.
- *Features*: Para cada nova funcionalidade debe realizarse nunha rama nova, específica para esa funcionalidade. Estas débense sacar de *development*. Unha vez que a funcionalidade estea feita, faise un *merge* da rama sobre *development*, onde se integrará coas demais funcionalidades.
- *Hotfix*: Son bugs que xorden en produción, polo que se deben amañar e publicar de forma urxente. É por iso, que son ramas sacadas de *master*. Unha vez corrixido o erro, debese facer un *merge* da rama sobre *master*. Ao final, para que non quede desactualizada, debese realizar o *merge* de *master* sobre *development*.

GitHub Flow cambia lixeiramente o modo de traballo. Non hai rama “Development” senón que as novas *features* parten directamente da rama *master*. O *deploy* tamén é diferente porque se fai a nivel *feature*, non a nivel *master*.

2.2 Instalación Git en Windows

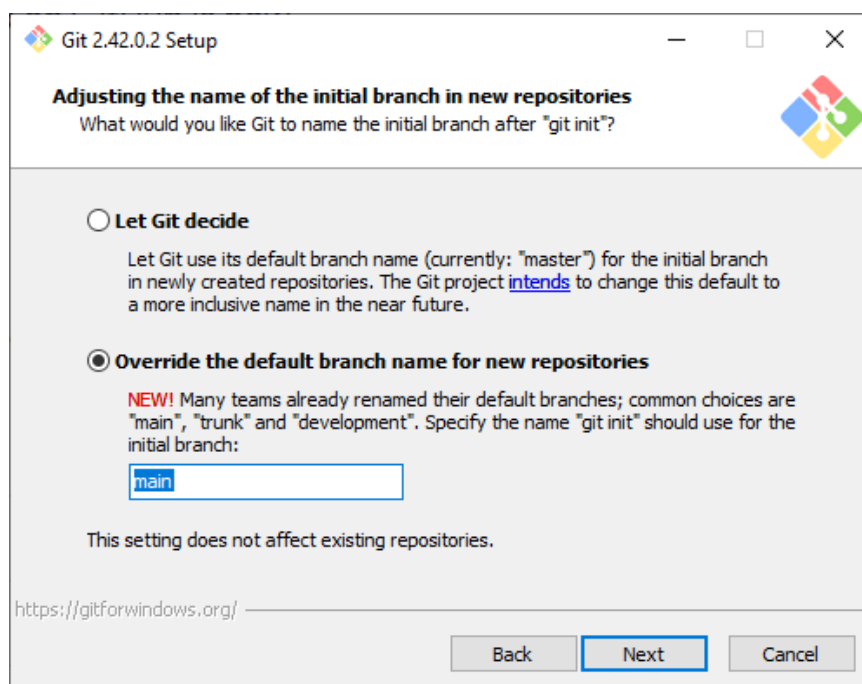
Descargaremos a ferramenta oficial *Git for Windows* desde <https://git-scm.com/download/win>. Esta intégrase no explorador de arquivos.



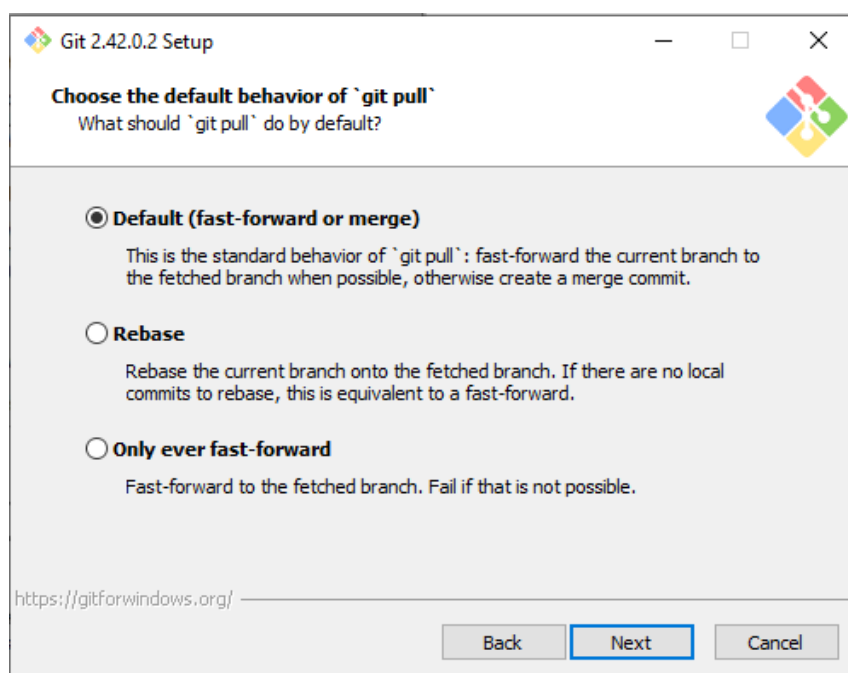
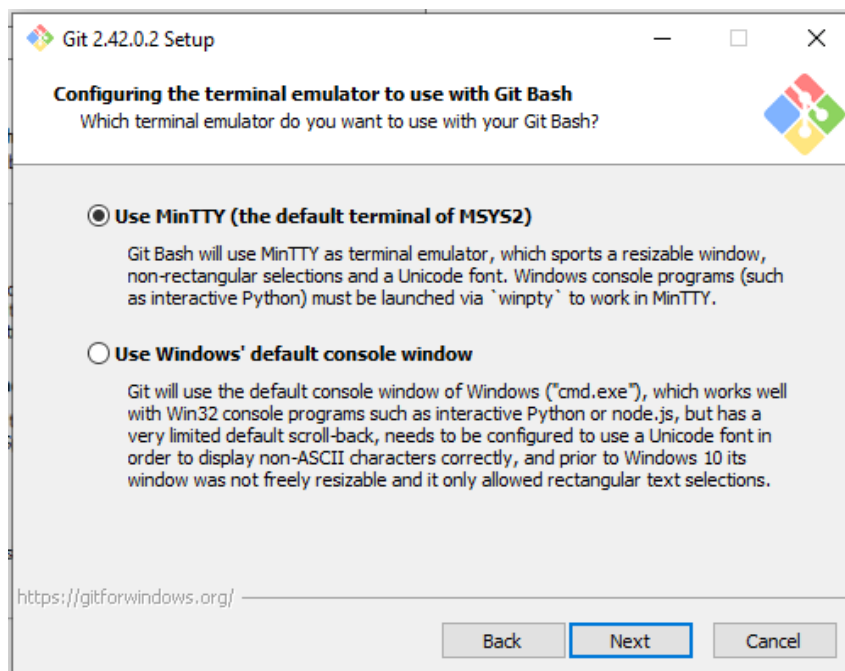


Se non marcamos esta opción, podemos modificala máis tarde, asociando VSCode á aplicación co comando:

```
git config --global core.editor "code --wait"
```



E finalmente na que indica o tratamento de fin de liña (estilo Windows ou Linux) marcamos a terceira opción: checkout as-is, commit as-is xa que traballaremos soamente nun sistema operativo.



2.3 Creando o repositorio

O primeiro paso será crear un novo cartafol con nome, por exemplo, de *misproyectos*, e con botón dereito: *Git Bash Here*

Na consola que aparece, creamos un novo cartafol para o noso primeiro proxecto: `mkdir proyecto1` Na consola dispoñemos de comandos estilo linux como `pwd`, `cd`, `ls`, `rm -rf`.

Para crear o repositorio git, executamos: `git init` creándose un cartafol oculto `.git` coa información necesaria para a xestión da ferramenta. Podes comprobalo con `ls -lart`

.git/ (l: detalles, a: amosa os ocultos, r: orde inversa, t: ordenar por data de última modificación).

```
Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/misproyectos
$ git init
Initialized empty Git repository in C:/Users/Usuario/Desktop/misproyectos/.git/

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/misproyectos (master)
$ ls -lart .git/
total 11
drwxr-xr-x 1 Usuario 197121  0 feb. 27 16:27 ../
-rw-r--r-- 1 Usuario 197121 73 feb. 27 16:27 description
drwxr-xr-x 1 Usuario 197121  0 feb. 27 16:27 hooks/
drwxr-xr-x 1 Usuario 197121  0 feb. 27 16:27 info/
drwxr-xr-x 1 Usuario 197121  0 feb. 27 16:27 refs/
-rw-r--r-- 1 Usuario 197121 23 feb. 27 16:27 HEAD
-rw-r--r-- 1 Usuario 197121 130 feb. 27 16:27 config
drwxr-xr-x 1 Usuario 197121  0 feb. 27 16:27 ./
drwxr-xr-x 1 Usuario 197121  0 feb. 27 16:27 objects/
```

É preciso tamén indicarlle dous parámetros, o correo electrónico e o nome do noso usuario:

```
git config --global user.name "Fernando RD"
```

```
git config --global user.email "rdf@fernandowirtz.com"
```

Esta configuración almacénase en ./gitconfig

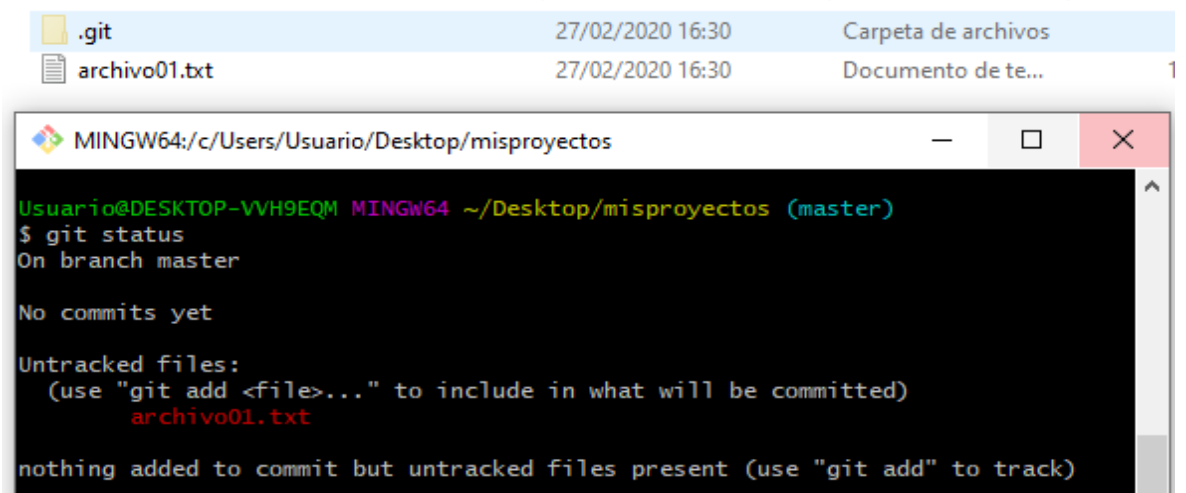
2.4 Operacións básicas

Ollo: Revisar o gráfico do principio deste tema co esquema de operacións git entre working directory / staged (ou index) / repositorio local / repositorio remoto.

Engadir arquivos

Se creamos agora un arquivo nese cartafol (por exemplo o código da nosa aplicación) por agora está fóra de git, non ten constancia, non está rexistrado. Podemos comprobalo con **git status**:

```
git status
```



The image shows a file explorer window at the top with the following contents:

Icon	Name	Date/Time	Type
Folder icon	.git	27/02/2020 16:30	Carpeta de archivos
File icon	archivo01.txt	27/02/2020 16:30	Documento de te...

Below the file explorer is a terminal window titled "MINGW64:/c/Users/Usuario/Desktop/misproyectos". The terminal output is as follows:

```
Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/misproyectos (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        archivo01.txt

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

Os arquivos non rexistrados amósanse en cor vermella. O primeiro paso será introducilo no sistema, é dicir, que git o rexistre como un arquivo sobre o que xestionar versións. Para iso executamos o comando **git add**:

<code>git add arquivo.txt</code>	→ engade o arquivo
<code>git add carpeta</code>	→ engade toda a carpeta

Para engadir máis arquivos nunha soa vez, podemos usar o *punto* para indicar todos os arquivos do directorio actual: `git add .` ou con `-A` ou `--all` ou empregar máscaras: `git add *.txt`

Agora repetimos `git status` para comprobar a situación. Comprobamos que está nesa primeira situación chamada **staged**, é dicir, listo para ser enviado ao repositorio.

```
Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/misproyectos (master)
$ git add archivo01.txt

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/misproyectos (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   archivo01.txt
```

Estes arquivos aparecen en verde, preparados para aprobar.

Agora se facemos **git commit**, o arquivo xa quedaría rexistrado, nunha “foto” ou revisión coa que poderemos traballar ao longo do tempo. Non hai que indicar o nome do arquivo, faise commit de todos os arquivos que estean na situación de *staged* (os que aparecen en verde). Executamos co parametro `-m` para indicar cun texto o obxecto desta revisión ou versión

```
git commit -m "primer commit"
```

`git commit -a -m "segundo commit"` Fai commit de todos os arquivos modificados ou borrados pero non arquivos novos (**aos novos é preciso facerlle add unha primeira vez**)

Se agora facemos `git status` veremos que non hai nada pendente e con: `git log` vemos o historial de cambios.

```
Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/misproyectos (master)
$ git commit -m "primer commit"
[master (root-commit) 88c6a88] primer commit
1 file changed, 4 insertions(+)
create mode 100644 archivo01.txt

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/misproyectos (master)
$ git status
On branch master
nothing to commit, working tree clean

Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/misproyectos (master)
$ git log --oneline
88c6a88 (HEAD -> master) primer commit
```

Son interesantes estas opcións de `git log`: `git log --oneline --decorate --graph --all`

Diferenzas entre arquivos

Se modificamos un arquivo do que xa fixemos un *commit*, git detectará a diferenza entre os dous (o da versión do *working directory* e que está na área de *commit*). Para seguir co exemplo, modificamos o arquivo *archivo01.txt* inserindo unha liña.

```
Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/misproyectos (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   archivo01.txt

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

Para ver as diferenzas **git diff**:

`git diff`

```
Usuario@DESKTOP-VVH9EQM MINGW64 ~/Desktop/misproyectos (master)
$ git diff
diff --git a/archivo01.txt b/archivo01.txt
index 9359161..7f12cdb 100644
--- a/archivo01.txt
+++ b/archivo01.txt
@@ -1,3 +1,4 @@
  linea A
  linea B
+linea nueva entre B y C
  linea C
```

Borrar e Renomear arquivos

Farémolo sempre con **git rm** e **git mv** respectivamente para que *git* rexistre o cambio. Despois de calquera das dúas operacións deberemos facer *commit*, e dicir, confirmar o cambios no HEAD. Se **facemos borrados o renomeados por fóra de git** estarán *untracked*, e dicir, non rexistrados en git.

Desfacendo cambios

checkout

Agora imos tratar casos no sentido oposto, se con *add* e *commit* pasabamos arquivos do *working directory* a *stage* e *commit* respectivamente, agora veremos o contrario, como pasar cambios xa aprobados ao noso directorio de traballo.

Para pasar da sección *commit* ao directorio de traballo: **git checkout**. Isto é útil para desfacer cambios que teñamos no *working directory*.

`git checkout <arquivo>` (ou arquivos empregando máscaras ou *punto*.)

`git diff` comparan diferenzas de arquivos entre os que están no *working directory* e no repositorio (*commit*), pero non nos que xa están no *staging*, é dicir non ten en conta sobre os que xa fixemos *add*. Para comparar os *staged* cos que xa están na área de *commit* engadimos o parámetro `--staged` ou `--cached`

```
git checkout idRevision (pode ser HEAD, como vemos no exemplo seguinte)
git checkout HEAD <arquivo> (recupera todo na súa última versión, moi frecuente)
```

HEAD é a última situación da rama, a situación actual.

reset

Para pasar da sección *staging* ao directorio de traballo: **git reset**. Isto é útil cando cremos que xa temos os cambios listos para pasar ao repositorio pero queremos modificar algo de novo.

A opción **--hard** de *reset* elimina commits previos. É perigoso porque perderemos o histórico desde commit do noso historial (se por exemplo fixemos push a un servidor remoto previamente, pode crear confusión que haxa uns commits no remoto que no existen no local). Por exemplo:

```
git reset --hard HEAD~1
git reset --hard idRevision
```

O til de nasalidade despois do HEAD indica o número de revisión, se HEAD é a última, HEAD~1 é a penúltima (no teclado, o til de nasalidade escríbese con AltGr + 4 ou Alt + 12 no teclado numérico).

restore

Outra ferramenta para desfacer cambios é **restore**, que reverte cambios dos que aínda non fixemos commit, cambios non aprobados tanto no working directory como en staged, son revertidos desde a copia que hai na zona de commit.

Este comando está aínda en versión experimental, polo que pode cambiar o seu comportamento.

```
git restore fich (se está no working directory)
```

```
git restore --staged fich e logo: git restore fich (se está staged)
```

revert

Para rematar este apartado, un último comando: **revert** que permite reverter a situacións anteriores indicando as modificacións que queremos eliminar. Exemplo:

```
git revert HEAD --no-edit //reverter o último commit
git revert 899001f --no-edit //reverter o commit 899001f
git revert HEAD...HEAD~2 --no-edit //reverter os 3 ult.commits
```


Diferentes situacións dos arquivos á hora de borrar /restaurar

Borrados fóra de git (p.e. borrado desde o explorador de arquivos)

* Se creo un arquivo e o borro (fóra de git)

Cun git status non vemos nada. git non sabe nada del e, polo tanto, non é capaz de recuperalo con restore.

Supoñemos que nunca se fixo commit sobre ese arquivo.

* Se creo un arquivo, fago add e o borro (fóra de git)

Cun git status vemos o delete en vermello (un borrado non rexistrado para facer commit dese borrado)

Se facemos un restore, git coñéceo (tívoo na área de stage un tempo) así que si é capaz de recuperalo.

* Se creo un arquivo, fago add+commit e o borro (fóra de git)

Igual que no caso anterior. Cun git status vemos o delete en vermello (un borrado non rexistrado para facer commit dese borrado).

Se facemos un restore, git coñéceo (teno no HEAD) así que si é capaz de recuperalo.

Borrados con "git rm"

A diferenza fundamental co borrado por fóra, é que pasa ese borrado á área de stage. É dicir, é un borrado rexistrado para facerlle un commit a ese borrado e, polo tanto, borrar o arquivo que estivese en HEAD (porque lle fixemos un commit anteriormente).

* Se creo un arquivo e o borro con git rm

git non sabe nada del e non nos deixa facer git rm (e, polo tanto, non o borra)

* Se creo un arquivo, lle fago add e o borro con git rm

Non deixa facer git rm directamente, porque git rm borra do working directory e este xa está na área de preparación (stage). Ten medo a que borremos algo que temos listo para facer commit.

Habería que facer **git rm -f** (f de force) e nese caso elimínalo totalmente ou ben **git rm --cached**, o que elimina o paso add. En calquera caso, volveríamos ao caso inicial, non é capaz de recuperalo, porque o ficheiro non está rexistrado.

* Se creo un arquivo, fágolle add+commit e o borro con git rm

Este é o caso máis habitual. É un arquivo que xa está no HEAD, xa fixemos commit previamente.

Ao facer git rm borra o arquivo do working directory, e deixa o borrado en staged (delete en verde).

2.5 Traballando con ramas

Unha rama (branch) permite traballar nun mesmo repositorio con versións diferentes dun mesmo conxunto de arquivos. Normalmente unha rama xurde a partir dunha situación (por exemplo da rama MASTER) e a partir dese momento ten “vida propia” e evoluciona de forma independente.

Exemplos típicos de ramas son as correspondentes á fase de desenvolvemento de novas funcionalidades, ou a parches que debemos desenvolver de forma rápida separada doutras liñas de traballo no código, como xa vimos na sección: *Boas prácticas: “Git Flow”, “GitHub Flow”, etc.*

Poderemos “movernos” entre as distintas ramas sen movernos de directorio. Ao cambiar de rama, git cambia os contidos do *working directory*, pero mantendo todas as ramas actualizadas. Veremos como despois podemos fusionar distintas “ramas” resolvendo os conflitos que se produzan.

O comando para crear unha nova rama é **branch**:

<code>git branch novarama</code>	(copia da rama actual)
<code>git branch novarama MASTER</code>	(copia de MASTER ou rama principal)
<code>git branch novarama <u>ramaorixe</u></code>	(copia de <i>ramaorixe</i>)

Para cambiar dunha rama a outra, facemos **checkout**: `git checkout rama`

Agora podemos traballar na nova rama, os cambios que fagamos afectan soamente a esa rama, permanecendo intactas o resto de ramas.

Se queremos crear unha rama e movernos a ela, en vez de facer o dous comandos anteriores podemos facelo con **checkout -b** e a crea co contido da rama na que esteamos nese momento.

```
git checkout -b novarama
```

Para ver o conxunto de ramas que temos no proxecto: **branch** sen parámetros.

```
git branch
```

 (con opción `-a` mostra as ocultas e con `-v` informa do último commit)

Para facer a fusión de rama, situámonos na rama destino e **merge**:

```
git merge ramaOrixe
```

Este é un dos pasos máis importantes e críticos, xa que se fusionarán os cambios da rama indicada coa rama actual, quedando un código único. A fusión faise liña a liña e pode haber conflitos se dúas ramas distintas fixeron cambios sobre as mesmas liñas, como veremos no seguinte apartado.

fast forward é un termo que se aplica ao *merge* cando á rama orixe é tal cual a última situación en todos os aspectos. É dicir, non necesitamos ningunha liña da rama actual, collendo a rama orixe dos cambios temos a situación final que buscamos. Neste caso, git é intelixente, e en vez de facer un merge “normal” xuntando liñas dunha e outra rama, o que fai é que o HEAD apunte á última situación da rama orixe en vez de facer os cambios e commit sobre a rama actual.

Para que se dea esta situación, a rama actual non debeu ter ningún cambio desde que creamos a rama orixe ata o momento do merge.

Existe outra forma de facer a fusión de ramas, co comando **rebase**. O resultado final é o mesmo que con *merge* pero a diferenza é que neste caso non se fai a fusión da situación “final” de cada rama, senón que se van aplicando todos os commits que tivera esa rama ao longo da súa historia sobre a outra rama. Esta opción é máis propensa a ter conflitos.

Exemplo:



`git rebase` ao ser aplicado, NON mantén a salvo a historia da rama secundaria, senón que "re-escribe" a historia da rama principal integrando os *commits* da rama secundaria na rama principal, non crea un *commit* de unión adicional.

Para rematar co tema das ramas, quedaríanos por ver como eliminar unha rama (por exemplo unha vez fusionada con outro contido xa non a precisamos). O facemos co comando: **branch -d**

```
git branch -d rama
```

2.6 Resolución de erros e conflitos

Ver diferenzas

Xa vimos que o comando *diff* é unha boa ferramenta para comprobar as diferenzas entre arquivos que están nas distintas áreas do repositorio (working directory, staged, commit) pero tamén serve para comprobar as diferenzas dun arquivo en diferentes momentos (en diferentes commits). Por exemplo para comparar a situación actual coa de dous commits atrás:

```
git diff HEAD HEAD~2
```

Conflitos nas modificacións das ramas

Como vimos na sección anterior, ‘merge’ xunta ramas (a indicadas sobre a rama na que nos atopamos) pero podemos atoparnos con conflitos, é dicir, que en distintas ramas fan cambios sobre o mesmo código ao mesmo tempo.

Situación 1:

1. Creamos desde MASTER unha rama na que modificamos un arquivo nunha liña
2. Creamos desde MASTER outra rama na que modificamos o mesmo arquivo pero noutra liña.
3. Facemos *merge* da rama do punto 1 en MASTER.
4. Facemos *merge* da rama do punto 2 en MASTER

Que ocorre?

Neste caso non habería problema. Git fai a fusión de ramas liña a liña.

Situación 2:

1. Creamos desde MASTER unha rama na que modificamos un arquivo nunha liña
2. Creamos desde MASTER outra rama na que modificamos o mesmo arquivo na mesma liña.
3. Facemos *merge* da rama do punto 1 en MASTER.
4. Facemos *merge* da rama do punto 2 en MASTER

Que ocorre?

- Neste caso o paso 3 non dá problemas porque hai aínda non hai conflito.
- No paso 4 atópase que o MASTER actual (despois do paso3) non é o mesmo do que partiu, e isto é un conflito que git non sabe resolver.

Solucións:

- A solución é modificar o arquivo coas liñas que consideremos correctas (as da primeira rama, as da segunda ou o que queiramos) e facer **commit** a ese arquivo. Podemos ver as diferenzas con **git diff rama1 rama2** (ou **git diff tool**)
- Se hai moitos conflitos e non é sinxelo resolvelo modificando os arquivos involucrados podemos desfacer o último merge: **git merge --abort**

2.7 Detached HEAD

HEAD apunta normalmente a unha rama con nome (por exemplo, *main*). Á súa vez, cada rama apunta a un *commit* específico. Vexamos un repositorio con tres *commits*, un deles etiquetado, e coa rama `master` seleccionada:

```
HEAD (refers to branch 'master')
|
v
a---b---c branch 'master' (refers to commit 'c')
^
|
tag 'v2.0' (refers to commit 'b')
```

Cando se crea un *commit* neste estado, a rama se actualiza para apuntar ao novo *commit*. HEAD aínda apunta á rama `master`, e, polo tanto, indirectamente, ao *commit* d:

```
$ edit; git add; git commit

HEAD (refers to branch 'master')
|
v
a---b---c---d branch 'master' (refers to commit 'd')
^
|
tag 'v2.0' (refers to commit 'b')
```

Ás veces é útil poder saltar a un *commit* que non está na punta de ningunha rama con nome, ou mesmo crear un novo *commit* que non sexa apuntado por unha rama con nome. Aquí facemos un `checkout` ao *commit* b:

```
$ git checkout v2.0 # or
$ git checkout master^^

HEAD (refers to commit 'b')
```

```

      |
      v
a---b---c---d  branch 'master' (refers to commit 'd')
      ^
      |
tag 'v2.0' (refers to commit 'b')

```

HEAD agora apunta directamente ao *commit* b. Isto coñécese como *detached* HEAD. Significa simplemente que HEAD apunta a un *commit* específico, en oposición a apuntar a unha rama con nome. Vexamos o que pasa cando creamos un *commit*:

```

$ edit; git add; git commit

      HEAD (refers to commit 'e')
      |
      v
      e
      /
a---b---c---d  branch 'master' (refers to commit 'd')
      ^
      |
tag 'v2.0' (refers to commit 'b')

```

Temos un novo *commit* que só é apuntado por HEAD, e incluso podemos crear máis *commits*:

```

$ edit; git add; git commit

      HEAD (refers to commit 'f')
      |
      v
e---f
      /

```

```

a---b---c---d  branch 'master' (refers to commit 'd')
      ^
      |
tag 'v2.0' (refers to commit 'b')

```

E executar todas as operacións de git, pero no momento que volvemos a facer `checkout` a `master`:

```

$ git checkout master

      HEAD (refers to branch 'master')
      e---f      |
      /          v
a---b---c---d  branch 'master' (refers to commit 'd')
      ^
      |
tag 'v2.0' (refers to commit 'b')

```

Xa nada referencia o *commit* `f`, durante a rutina de recolección de lixo de Git vaino borrar. Se non queremos perdelo, debemos facer o seguinte mentres conservamos a referencia (antes de saltar á rama `master`):

```

$ git checkout -b foo # or "git switch -c foo"  (1)
$ git branch foo      (2)
$ git tag foo         (3)

```

Crea unha nova rama chamada `foo`, que se apunta ao *commit* `f`, e logo actualiza `HEAD` para apuntar á rama `foo`. Noutras palabras, xa non estaremos en estado de *detached* `HEAD` despois deste comando.

2.8 Repositorios remotos

Os repositorios remotos son versións do teu proxecto que están hospedados en Internet ou na intranet da túa empresa. Podes ter varios, dalgúns deles serás o propietario, doutros un colaborador e de moitos deles só terás permisos de lectura.

A forma de traballar con eles, a nivel xeral, consiste e traer os seus datos a local (`push`), modificalos e subir os cambios (`pull`).

Dispoñemos de dúas ferramentas moi coñecidas para xestionar as nosas versións de código en **remoto** e compartilas con outros usuarios ou co público en xeral: GitHub e GitLab. Teñen ambas unhas características comúns.

- Teñen interface web e como plataforma social para compartir coñecemento e traballo.
- Podemos ter un número ilimitado de repositorios públicos (vista, non commit) pero tamén privados.
- Hai contas para empresa (de pago).
- Permite ramas e comunicarnos cos usuarios (pull request).

A maior diferenza é que GitLab podémolo instalar no noso servidor e o primeiro sempre hai que úsalo a través da web github.com. Por outra banda, nas versións web gratuitas, GitLab ofrece mellores condicións de espazo e número de colaboradores por proxecto.

Conexión remota

Para conectarnos cun repositorio, primeiro crearemos o noso repositorio local con *git init*. Logo crearemos o repositorio vacío en GitHub /GitLab:



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere?

[Import a repository.](#)

Owner

Repository name *


 fernando / myweb 

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

Description (optional)

Probas


☐  **Public**
Anyone can see this repository. You choose who can commit.

☒  **Private**
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

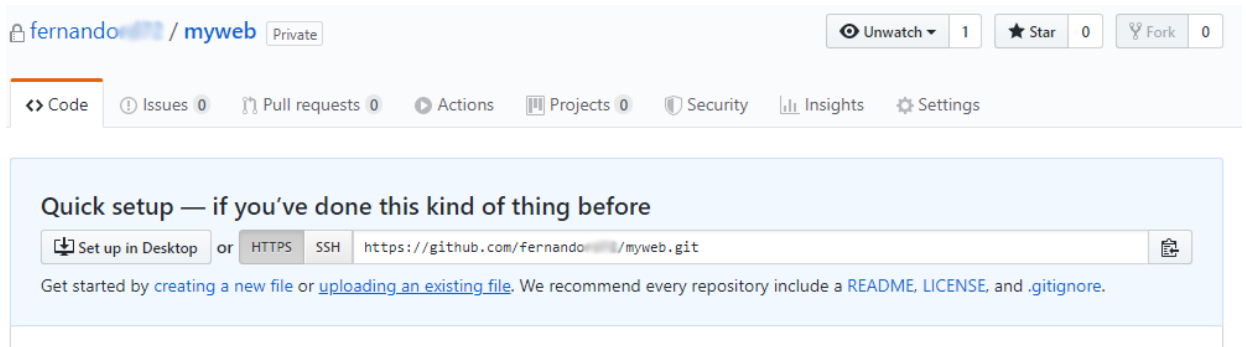
☐ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer.

Add .gitignore: **None** ▼

Add a license: **None** ▼ 

Create repository

Obtendo a URL de conexión:



Despois o vinculamos con remoto con **remote add origin url**:

```
git remote add origin https://github.com/fernando-xxxx/myweb.git
```

“*origin*” non é unha palabra reservada, pódese poñer calquera nome, pero é un estándar chamarlle así ao noso repositorio principal. De feito, se temos varios repositorios propios, terán distintos nomes. E claro, os repositorios dos que só somos colaboradores tamén lle asignaremos outros nomes.

Para ver os repositorios remotos que temos configurados empregamos simplemente **remote**, coa opción **-v** para ter máis información.

```
git remote -v
```

A partires dese momento, nos comandos traballaremos con este nome, e non coa url real.

Para inspeccionar a información en profundidade: **remote show origin** ou `remote show nomeRepo`

“Subir” información ao repositorio remoto

A operación contraria, sería “subir” ao repositorio remoto os cambios feitos en local (os cambios fariámolos co proceso xa visto: modificar o arquivo + add + commit). O comando subilo ao repositorio remoto é **push**:

```
git push origin nomeRama
```

solicitando as credenciais en GitHub

origin é o nome co que nos referimos á conexión remota, e no nome da rama poñeremos a rama que desexamos subir, en moitas ocasións será MASTER

```
git push origin master
```

“Baixar” información do repositorio remoto

O paso seguinte será “ver” a estrutura do repositorio remoto, as distintas “ramas”. Veremos máis adiante que as ramas son as distintas versións do programa (a de produción, as que están desenvolvendo novas funcionalidades, outra pode estar arranxando algún bug, etc). Sempre haberá unha rama principal, que é a que está en produción e chámase *master*. Descargaremos con **fetch origin** ou ben **fetch nomeRepoRemoto**:

```
git fetch origin
```


Unha vez feito isto, temos descargados os arquivos pero nunha rama local oculta (chámase `origin/master`) Agora temos que pasar o descargado á nosa rama `master` no repositorio local. Iso facémolo con **`merge origin/master`**:

```
git merge origin/master
```

Como estas dúas operacións fanse habitualmente xuntas, temos un comando que fai as dúas seguidas. É o comando **`pull`** e é máis empregado que os dous anteriores.

```
git pull origin master
```

Tamén temos un comando git que fai dunha sola vez o `git init` e o `pull`, e chámase **`clone`**.

```
git clone https://github.com/nomerepositorio .
```

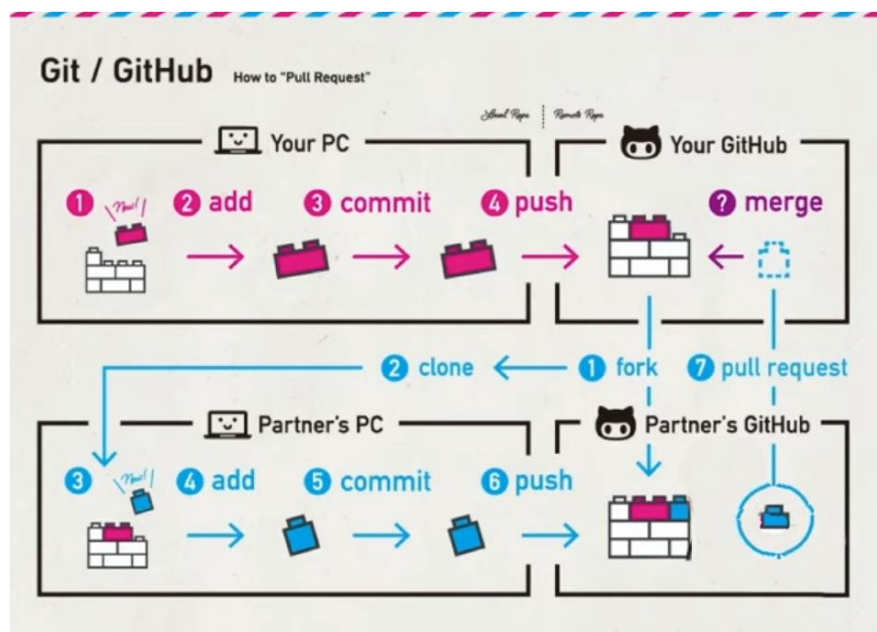
o punto indica que o copie no cartafol actual

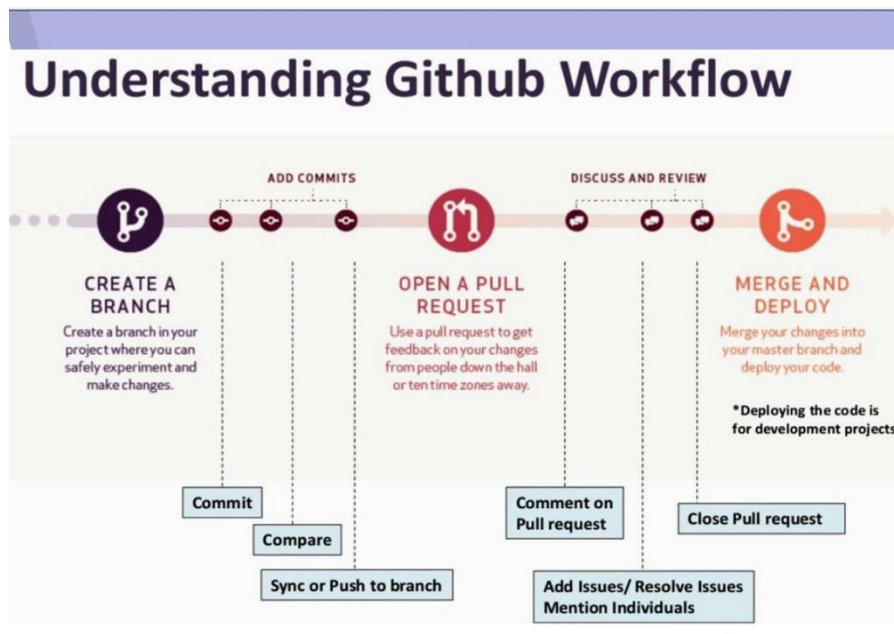
Temos tamén unha opción no checkout para traballar con ramas remotas: **`checkout -t origin/nomeRama`**: Se existe unha rama remota de nome “nomeRama”, ao executar este comando crease unha rama local co mesmo nome para facer un seguimento da rama remota co mesmo nome.

pull request

Son as solicitudes que facemos para modificar proxecto en GitHub que non son nosos, por exemplo para facer contribucións en proxectos de software libre. O propietario é o que vai facer os cambios, polo que o noso traballo será obter una copia (*fork*) do proxecto, traelo a local (*clone*), modifícalo (*commit*), subilo á nosa conta (*push*) e facer a petición de integración (*pull request*)

O propietario será o encargado de facer a fusión (*merge*).





2.9 Outros comandos

blame

Cando traballamos en grupo pode ser interesante saber que fixo algún cambio determinado, por exemplo para acordar coordinar os novos cambios (ou se é erróneo, para botarlle a culpa...). Para iso temos o comando **blame**.

```
git blame nomearquivo
```

podemos indicar quen modificou un arquivo pero nunhas liñas concretas.

```
git blame -L 12,14 src/Main.java
```

Cambiar o nome do repositorio remoto

Para cambiar o nome que lle demos a un repositorio ao mapealo con “add origin” empregaríamos **remote rename** nomeRepo e para eliminalo (por exemplo se non colaboras máis nese proxecto) **remote rm** nomeRepo

Tags

Unha etiqueta ou **tag** é un nome que lle podemos asignar a unha determinada rama nun determinado momento (isto é, a un commit determinado). Poderíamos usalo para nomear versións do noso software, ou para marcar logros.

```
git tag -a v1 -m 'Release v0.1'
```

Este comando nomea o último commit. Se queremos etiquetar un commit anterior, engadiremos o id do commit. Logo será sinxelo voltar e esas versións (moi útil en proxectos *agile*, por exemplo para etiquetar o final de cada sprint semanal). Farémolo do mesmo xeito que nos cambiamos dunha rama a outra, co comando checkout.

```
git checkout v1
```

stash

Se estamos traballando no noso código e temos que cambiar a outra rama por calquera motivo pero non queremos facer un *commit* do noso traballo porque se atopa nun estado parcial podemos facer *stashing* como comando **stash**, isto é, “conxelar” os nosos cambios desde o último *commit* e almacenalo de forma temporal, quedando o directorio de traballo limpo, sen ningún *commit* pendente.

```
git stash
```

Agora poderíamos pasar á nova tarefa, e cando desexásemos retomar o traballo anterior:

```
git stash list
```

mostaría a lista de “stashs” e seleccionaríamos o que queiramos:

```
git stash apply stash@{x}
```

Sendo **x** o número mostrado na lista.

rebase

Este comando, que xa vimos para o *merge* de ramas, tamén permite “refundir” varios commits do historial nun so. Imaxinemos que nun proxecto grande fixemos o ano pasado 100 commits e este ano outros 100. Pódenos interesar, por limpeza, xuntar os commits do ano pasado, por que xa non nos interesa telos por separado, nun so.

```
git rebase --interactive --root
```

A opción *interactive* permítenos seleccionar “a man” o rango de commits mediante unha pantalla e a opción *root* quere dicir que mostre desde o comezo dos tempos.

Esta perda de historial (perda de commits), pode ser perigoso, xa que nunca poderemos voltar á situación dos *commit* eliminados.