Introducción a la Ingeniería del Software

Control de versiones con

Git

Grado en Ingeniería Informática
Grado en Ingeniería del Software
Grado en Ingeniería de Computadores

Índice de contenidos

- Sistemas de control de versiones
- Tipos de sistemas de control de versiones
- Caso de estudio: git

Conceptos generales

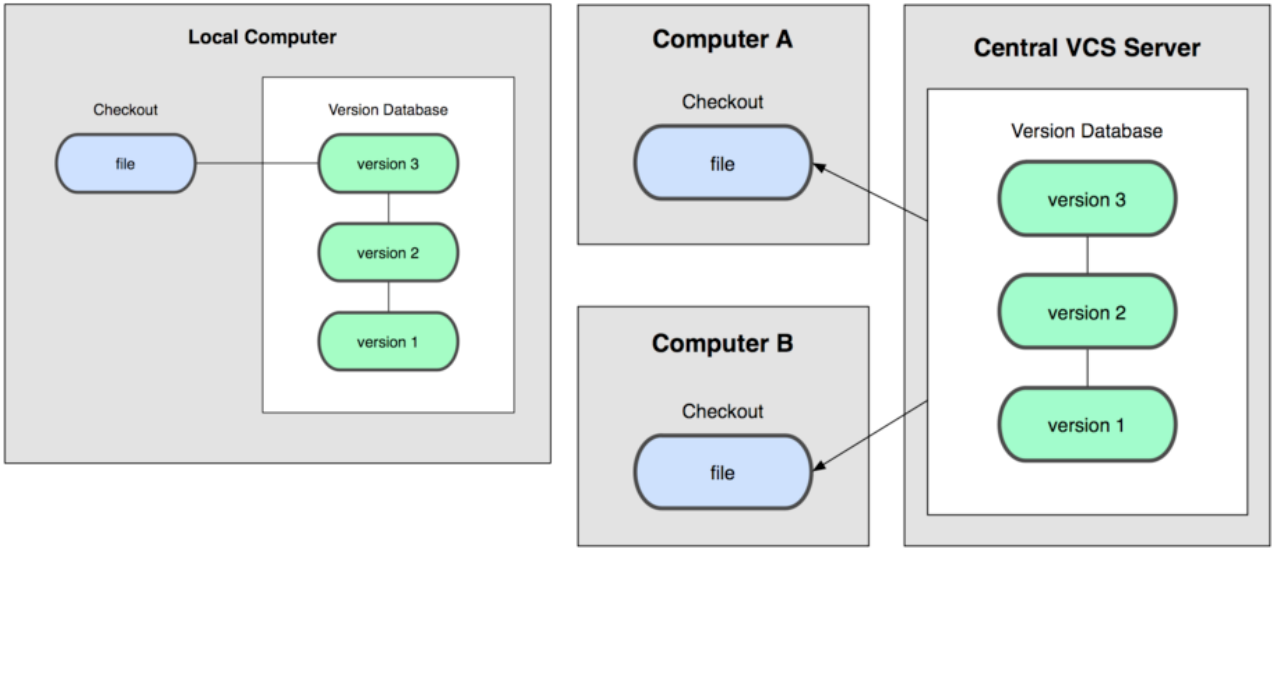
- ¿Qué es un sistema de control de versiones (SCV)?
  - Herramienta que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.
  - Cualquier tipo de archivo que encuentres en un ordenador puede ponerse bajo control de versiones.
- Propósito de un SCV
  - Permite revertir archivos a un estado anterior.
  - Comparar cambios a lo largo del tiempo.
  - Ver quién modificó por última vez algo que puede estar causando un problema.
  - Quién introdujo un error y cuándo, y mucho más.

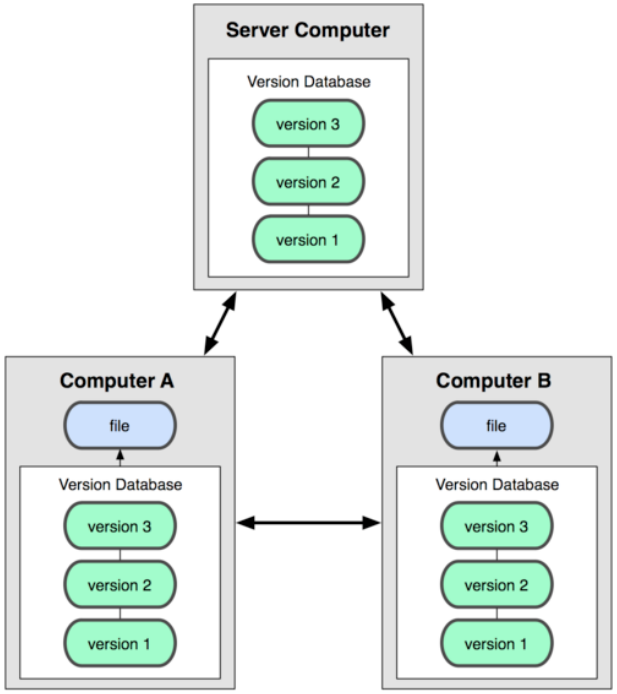
Tipos de sistemas de control de versiones

SCV Local (rcs)

SCV centralizado (cvs, subversion)

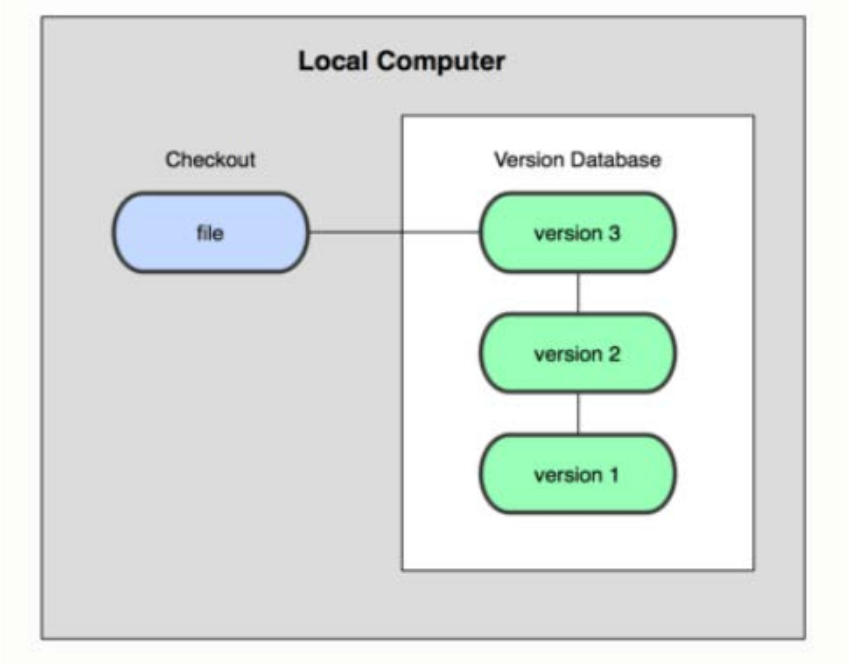
SCV distribuido (**git**, mercurial)





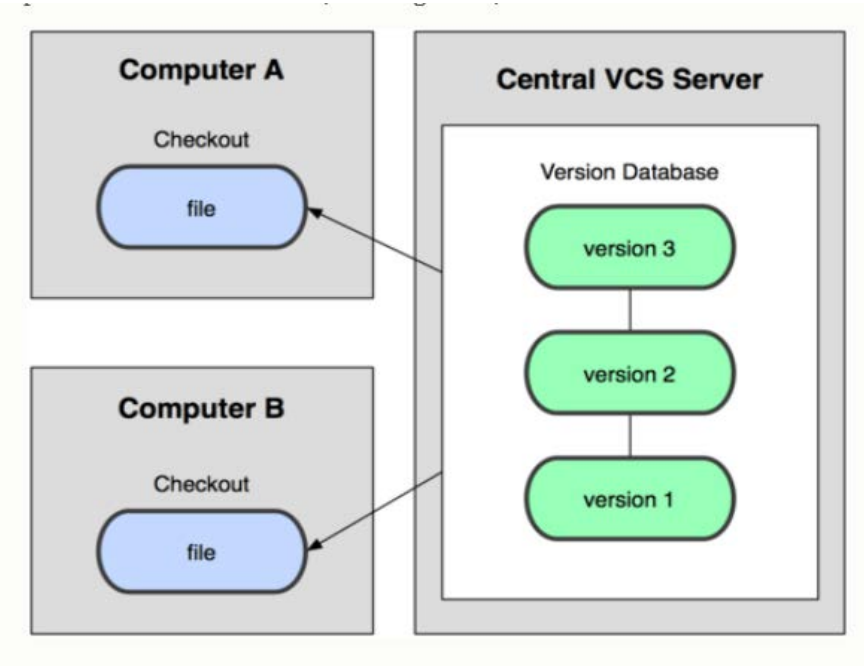
SCV Locales

- **Método más simple**: copiar los archivos a otro directorio, indicando la fecha y la hora.
- Método propenso a errores:
  - Es fácil olvidar en qué directorio te encuentras, y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías.
- **SCV locales**: tienen una base de datos en la que se registra todos los cambios realizados sobre los archivos.
- **Problema:** ¿cómo colaboran los desarrolladores?



SCV Centralizados

- **Arquitectura cliente-servidor**
  - Un único servidor que contiene todos los archivos versionados.
  - Los clientes descargan los archivos desde ese lugar central.
- Sistema más común para SCV durante muchos años.
- **Ejemplos**: CVS, Subversion, y Perforce,

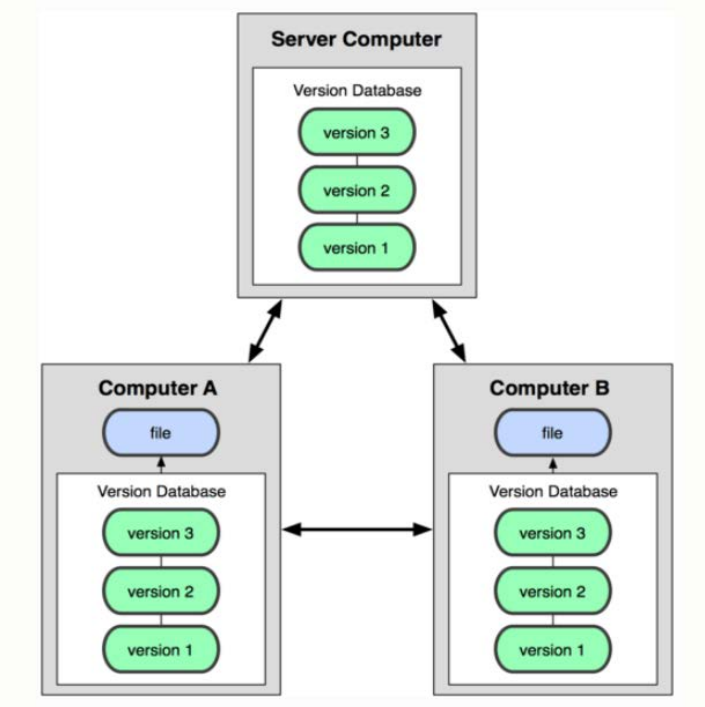


SCV Centralizado

- Ventajas respecto al SCV Local ☺
  - Todo el mundo puede saber en qué trabajan los otros colaboradores del proyecto.
  - Los administradores tienen control detallado de qué puede hacer cada uno.
  - Es más fácil administrar un SCV central que tener que lidiar con bases de datos locales en cada cliente.
- Desventajas ☹
  - El servidor es un punto crítico ante los fallos.

SCV Distribuido

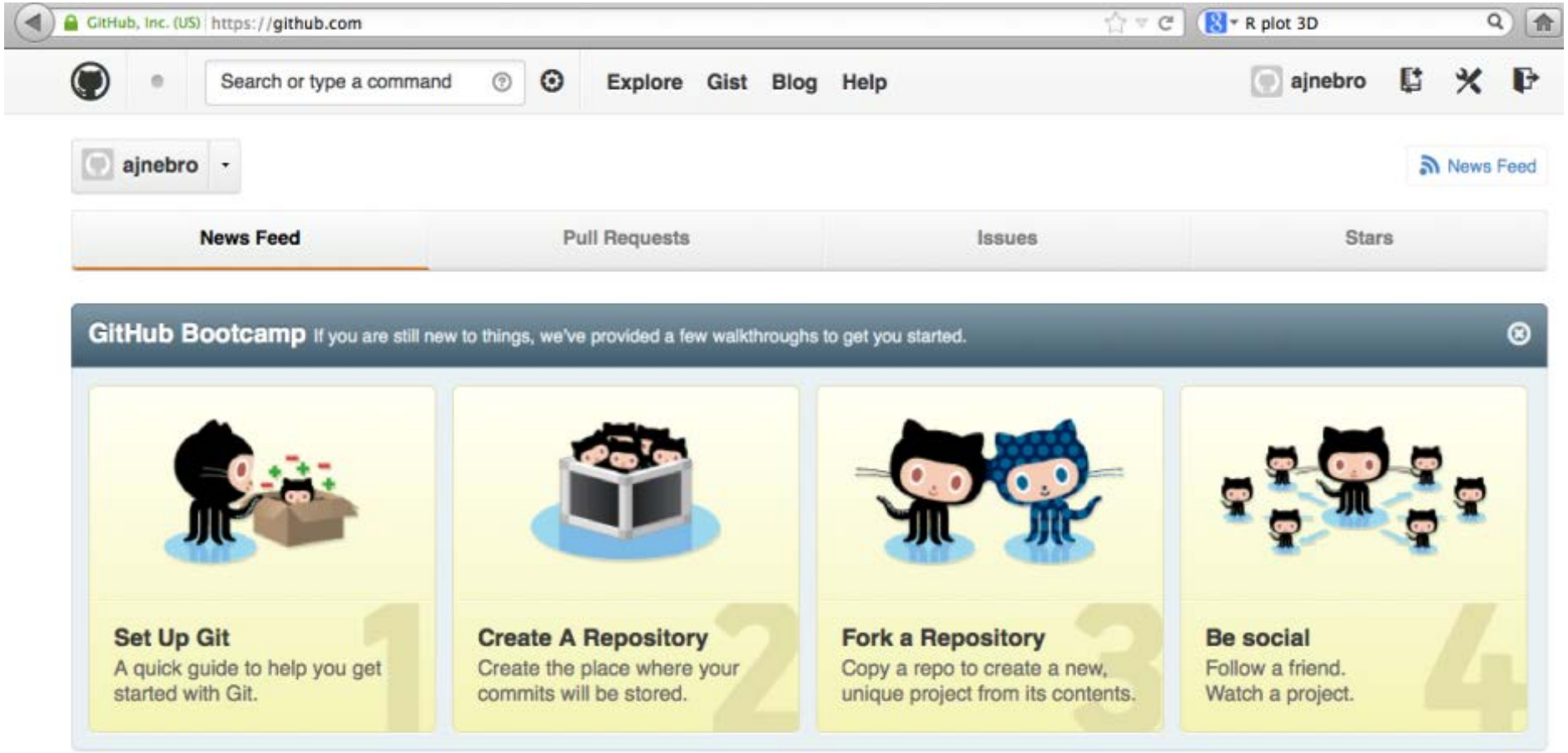
- **Arquitectura peer-to-peer**
  - Los clientes tienen una copia legítima del repositorio.
  - Pueden subir/bajar archivos al repositorio central.
  - Clientes pueden trabajar incluso cuando no hay conexión al servidor.
- **Ventajas** ☺
  - Si un servidor falla, cualquiera de los repositorios de los clientes puede copiarse en el nuevo servidor para restaurarlo.
  - Permiten varios repositorios de trabajo.
- **Ejemplos**: Git, Mercurial, Bazaar, Darcs



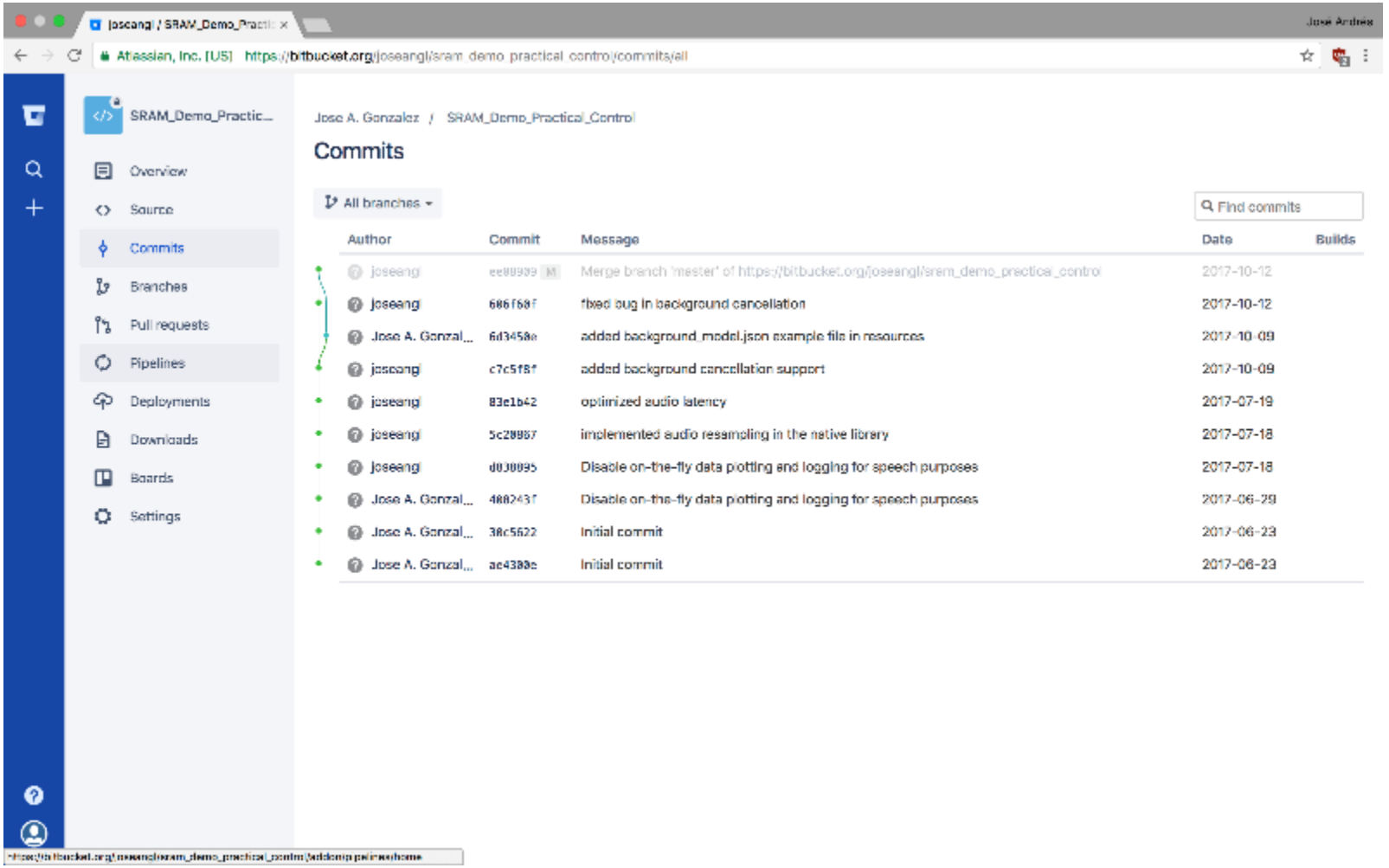
Caso de estudio: Git

- Ideado por Linus Torvalds para el desarrollo del kernel de Linux (1ª versión en abril de 2005).
- Características principales
  - SCV distribuido de código abierto
  - Existen numerosos clientes gráficos (GitKraken, SourceTree, TortoiseGit,…)
  - Existen *plugins* para muchos IDEs (Eclipse, NetBeans, …)
- Existen servicios de hosting gratuitos para alojar proyectos (repositorios) Git
  - GitHub (http://github.com)
  - Bitbucket (https://bitbucket.org/)

GitHub


GitHub, Inc. (US) https://github.com
R plot 3D
Search or type a command
Explore Gist Blog Help
ajnebro
ajnebro
News Feed
News Feed
Pull Requests
Issues
Stars
GitHub Bootcamp If you are still new to things, we've provided a few walkthroughs to get you started.
Set Up Git
A quick guide to help you get started with Git.
Create A Repository
Create the place where your commits will be stored.
Fork a Repository
Copy a repo to create a new, unique project from its contents.
Be social
Follow a friend.
Watch a project.

Bitbucket

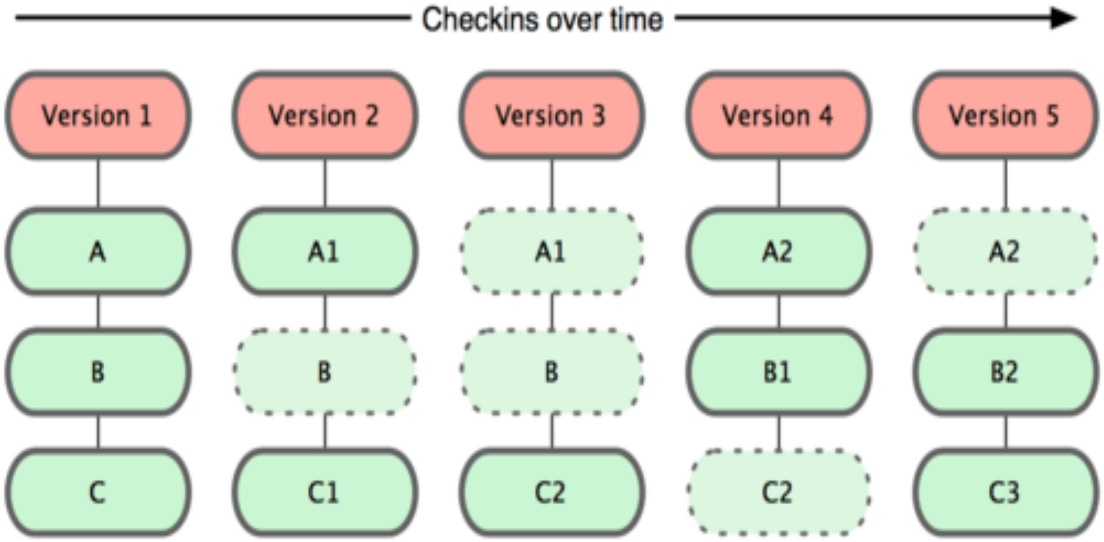

SRAM_Demo_Practic...
Overview
Source
Commits
Branches
Pull requests
Pipelines
Deployments
Downloads
Boards
Settings
Jose A. Gonzalez / SRAM_Demo_Practical_Control
Commits
All branches
Find commits
Author
Commit
Message
Date
Builds
joseangl
Merge branch 'master' of https://bitbucket.org/joseangl/sram_demo_practical_control
2017-10-12
joseangl
fixed bug in background cancellation
2017-10-12
Jose A. Gonzal...
added background_model.json example file in resources
2017-10-09
joseangl
added background cancellation support
2017-10-09
joseangl
optimized audio latency
2017-07-19
joseangl
implemented audio resampling in the native library
2017-07-18
joseangl
Disable on-the-fly data plotting and logging for speech purposes
2017-07-18
Jose A. Gonzal...
Disable on-the-fly data plotting and logging for speech purposes
2017-06-29
Jose A. Gonzal...
Initial commit
2017-06-23
Jose A. Gonzal...
Initial commit
2017-06-23

Instalación de Git

- Herramienta en línea de comandos:
  - **Linux/Mac**: ya suelen tener preinstalada una versión de git.
  - **Windows**: descargar desde https://git-scm.com/downloads
- Clientes gráficos
  - Ver: https://git-scm.com/downloads/guis
- Plugins para IDEs
  - Ya suelen venir preinstalados en los principales IDEs (Visual Studio, Android Studio, etc).

Fundamentos de Git

- **Git** modela los datos como un conjunto de instantáneas de un mini sistema de archivos.
- Cada vez que se confirma un cambio, Git “hace una foto” de los datos del proyecto y guarda una referencia a esa instantánea.
- Si los archivos no se han modificado, Git sólo almacena un enlace al archivo anterior idéntico que ya tiene almacenado.



Fundamentos de Git

- Operaciones en modo local
  - Sólo precisa archivos y recursos locales. Velocidad.
  - Si no estás conectado a una Red, puedes hacer casi todas las operaciones. No precisa un servidor.
  - Actúa como un SCV local.
- Integridad
  - Guarda el contenido no por el nombre, lo hace por el valor de hash.
  - Todas las comprobaciones se conocen como hash SHA-1. Se trata de una cadena de 40 caracteres hexadecimales y se calcula respecto a los contenidos del archivo o estructura de directorio.

24b9da6552252987aa493b52f8696cd6d3b00373

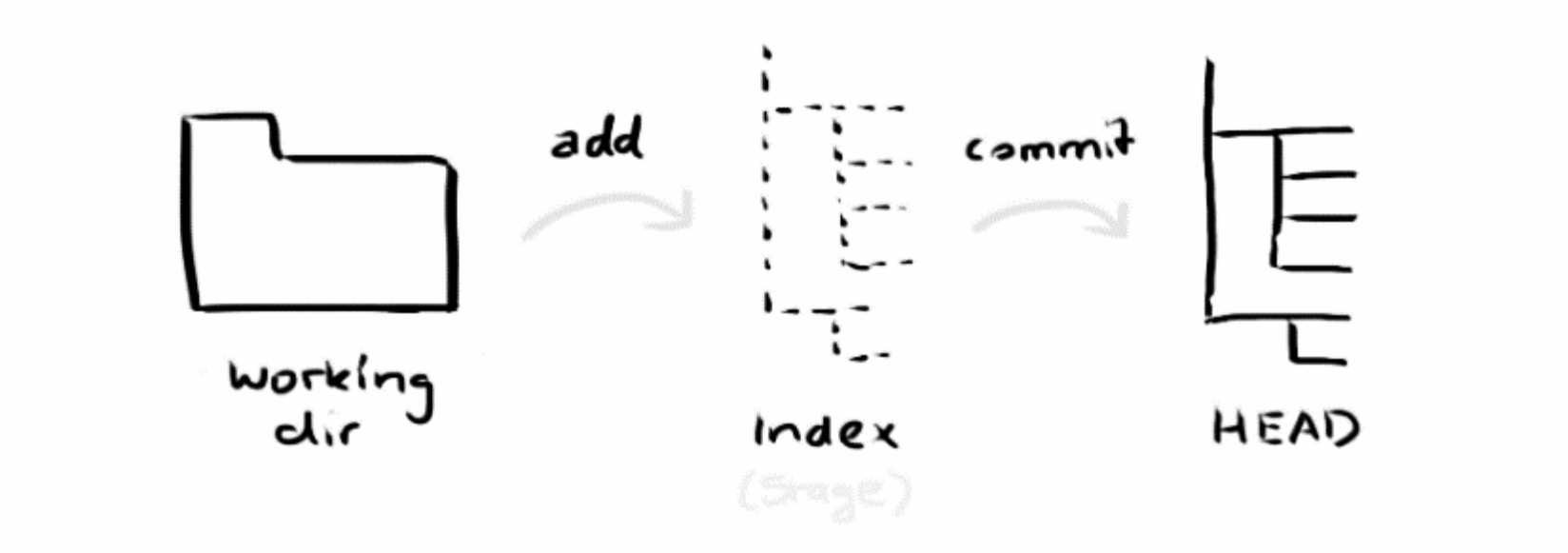
Fundamentos de Git

- Los cuatro estados de los archivos:
  - No seguido (untracked)
    - El archivo no está siendo seguido por Git (sus versiones no se controlan).
  - Modificado (modified).
    - Se ha modificado el archivo pero no ha sido confirmado en la base de datos.
  - Preparado (staged).
    - El archivo ha sido marcado en su versión actual para que vaya en la próxima confirmación.
  - Confirmado (committed).
    - Los datos están almacenados de manera segura en la base de datos local.

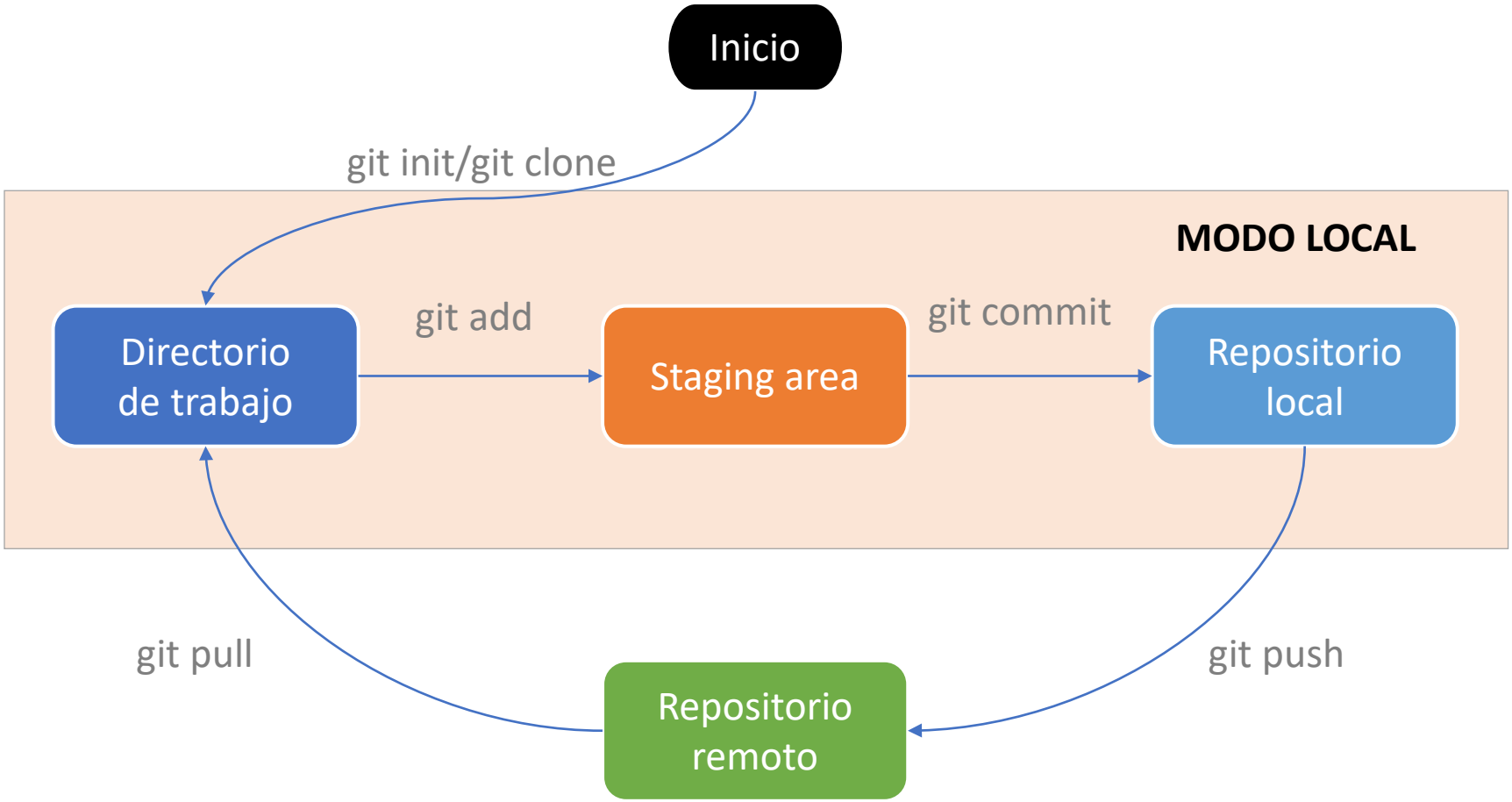
Fundamentos de Git

- Secciones de un proyecto de Git:
  - Directorio de trabajo (Working directory).
    - Es una copia de una versión del proyecto (no tiene por qué ser la última).
    - Los archivos se extraen de la base de datos del directorio de Git y se colocan en disco para usarlos.
  - Área de preparación (Staging area).
    - Almacena información sobre lo que va a ir en la próxima confirmación.
    - Es un área intermedia entre el Directorio de trabajo y el Directorio de Git.
  - Directorio Git (repositorio).
    - Es donde Git almacena los metadatos y la base de datos de objetos para el proyecto.
    - Es lo que se copia cuando se clona un repositorio.

Secciones de un proyecto



Flujo básico de trabajo en Git



Configuración inicial

- Configurar el nombre de usuario y dirección de correo electrónico
  - `git config --global user.name “jgonzalez"`
  - `git config --global user.email j.gonzalez@uma.es`

- Es importante porque las confirmaciones de cambios (commits) en Git usan esta información.
  - Si se usa --global esta opción siempre estará en el sistema.
  - Si se quiere reescribir esta información con otro nombre o dirección de correo para proyectos específicos se puede ejecutar el comando sin la opción --global cuando estemos en ese proyecto.

Creando un repositorio nuevo

- Crear un nuevo repositorio local desde cero
  - Nos situamos en el directorio desde donde cuelga el proyecto.
  - Ejecutamos

    ```
    git init
    ```

  - Se crea de forma automática una carpeta .git
  - A partir de ahora git llevará un control de las versiones de los archivos del proyecto.
- Clonando un repositorio existente
  - Para clonar un repositorio ejecutamos

    ```
    git clone <URL> [directorio]
    ```

  - El repositorio local es una copia del repositorio remoto.

Añadiendo archivos a la Staging Area

- Agregar un fichero a la *staging area*
  - Si el archivo se ha modificado o queremos que git empiece a llevar un control de sus versiones.

    ```
    git add <nombre del archivo>
    ```

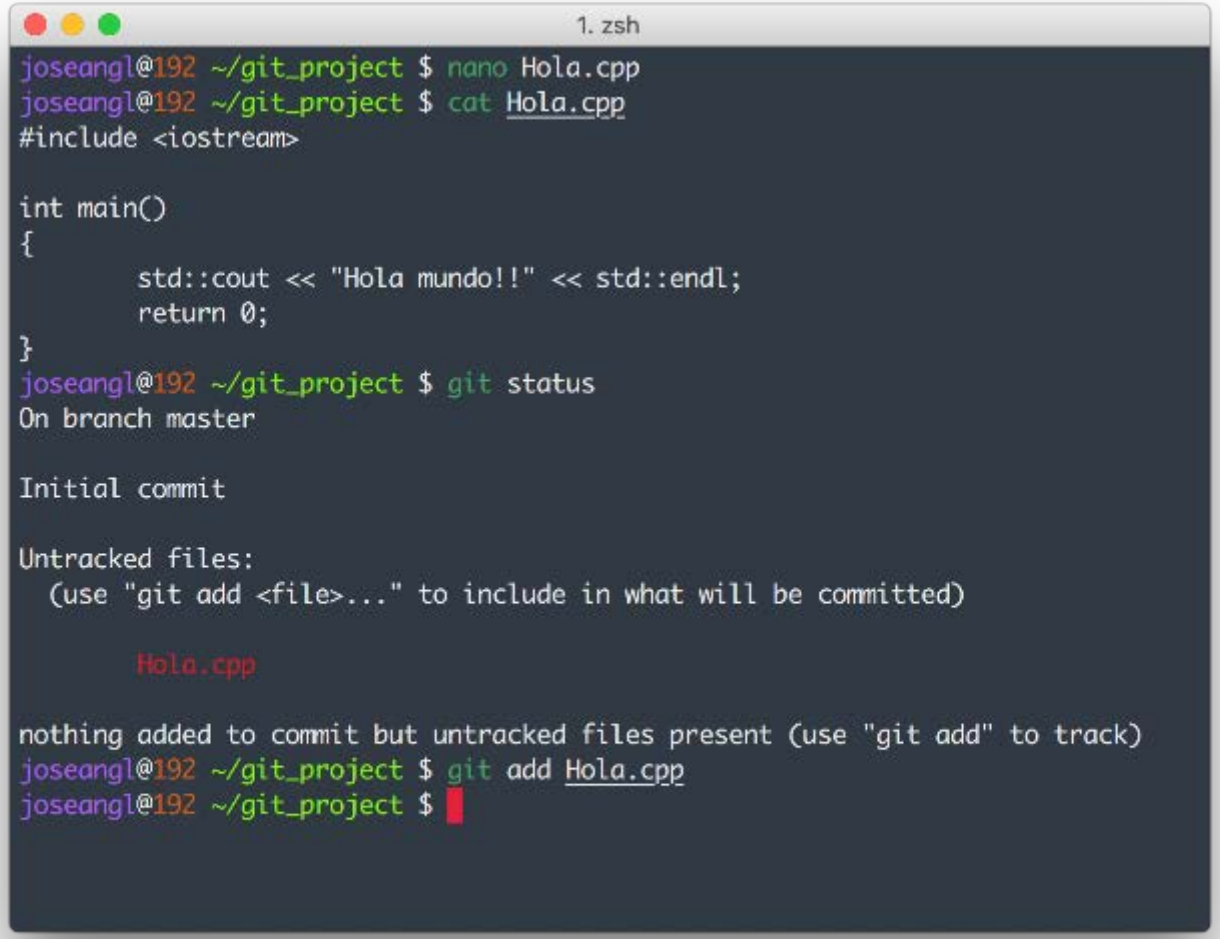
  - Esto añade el archivo a la *staging area* para que cuando hagamos un commit se confirmen sus cambios.
- Agregar todos los ficheros

  ```
  git add .
  ```

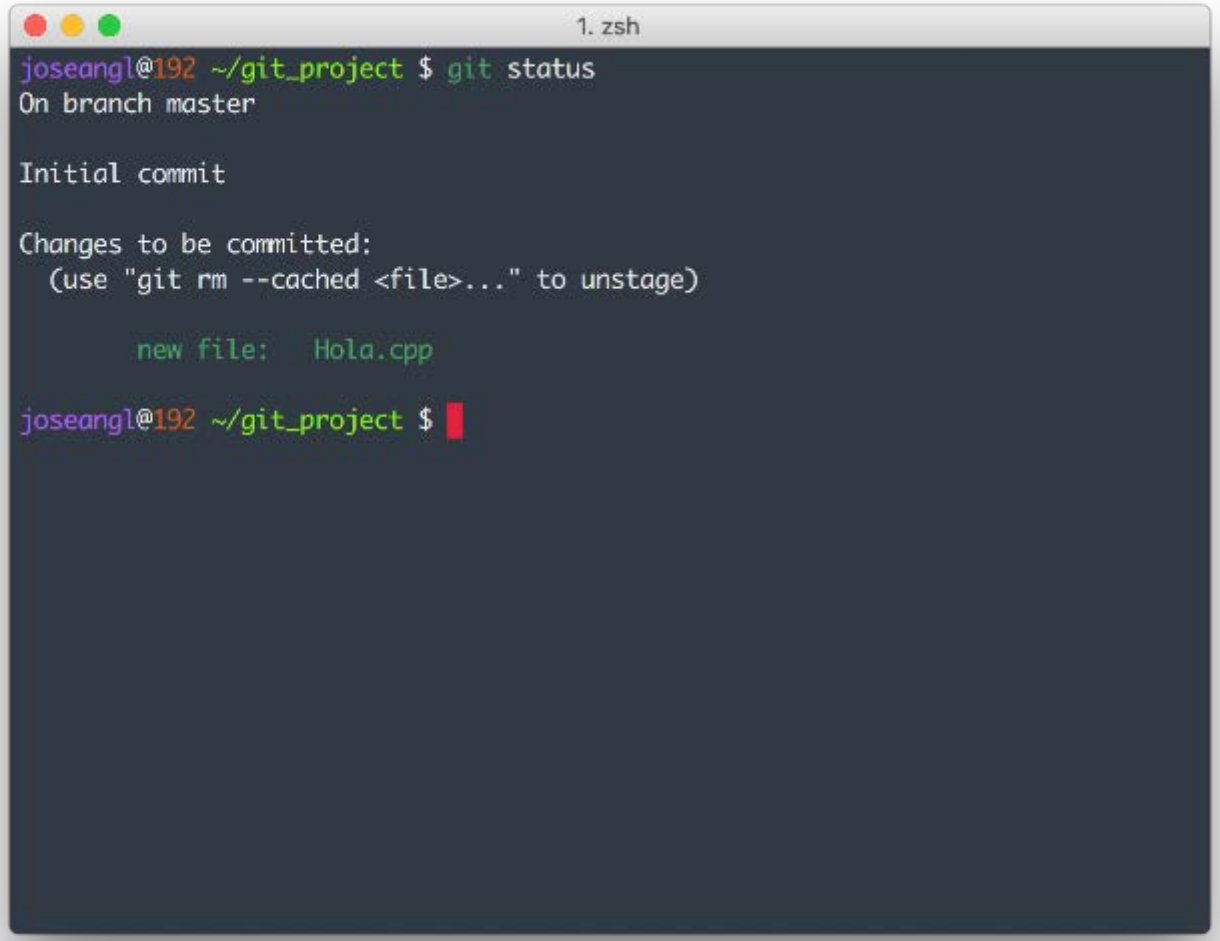
- Ver el estado de los ficheros

  ```
  git status
  ```

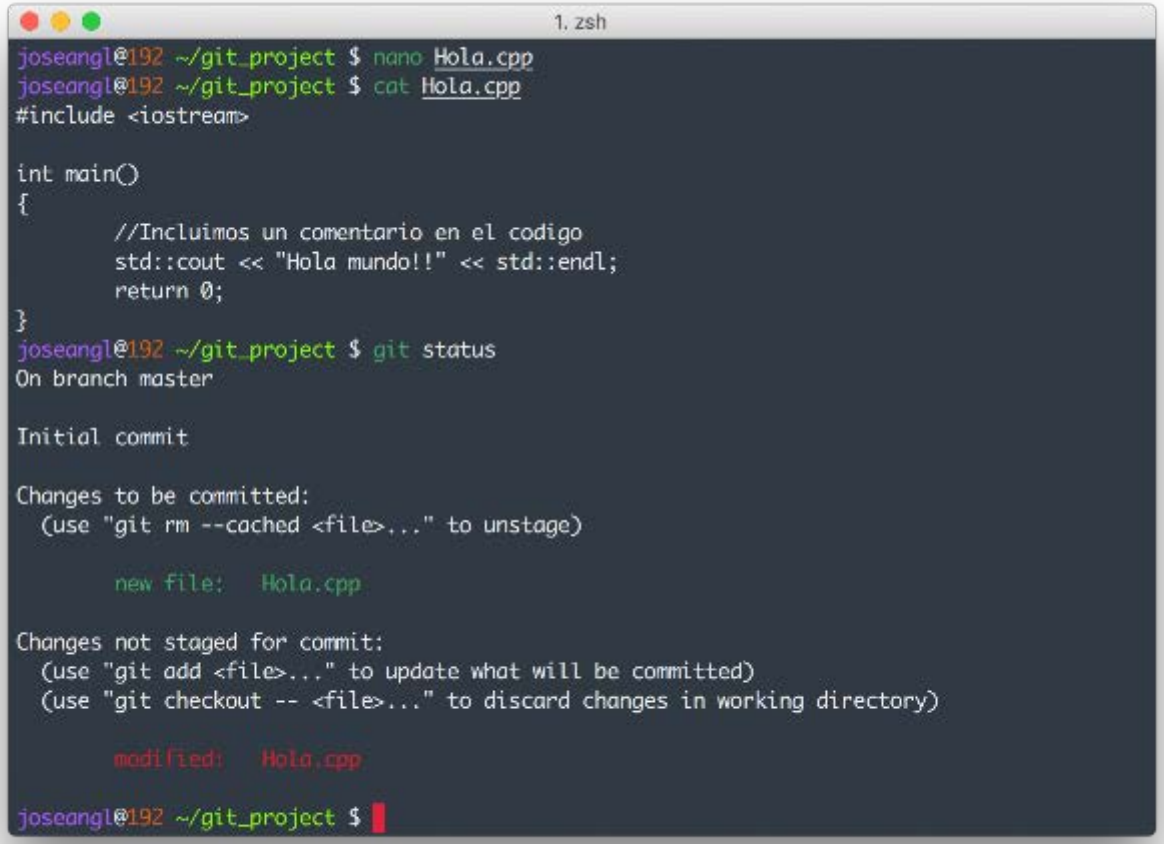
Ejemplo


1. zsh
joseangl@192 ~/git_project $ nano Hola.cpp
joseangl@192 ~/git_project $ cat Hola.cpp
#include <iostream>
int main()
{
std::cout << "Hola mundo!!" << std::endl;
return 0;
}
joseangl@192 ~/git_project $ git status
On branch master
Initial commit
Untracked files:
(use "git add <file>..." to include in what will be committed)
Hola.cpp
nothing added to commit but untracked files present (use "git add" to track)
joseangl@192 ~/git_project $ git add Hola.cpp
joseangl@192 ~/git_project $

Ejemplo


1. zsh
joseangl@192 ~/git_project $ git status
On branch master
Initial commit
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
new file: Hola.cpp
joseangl@192 ~/git_project $

Ejemplo


1. zsh
joseangl@192 ~/git_project $ nano Hola.cpp
joseangl@192 ~/git_project $ cat Hola.cpp
#include <iostream>
int main()
{
//Incluimos un comentario en el codigo
std::cout << "Hola mundo!!" << std::endl;
return 0;
}
joseangl@192 ~/git_project $ git status
On branch master
Initial commit
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
new file: Hola.cpp
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: Hola.cpp
joseangl@192 ~/git_project $

Consolidar los cambios

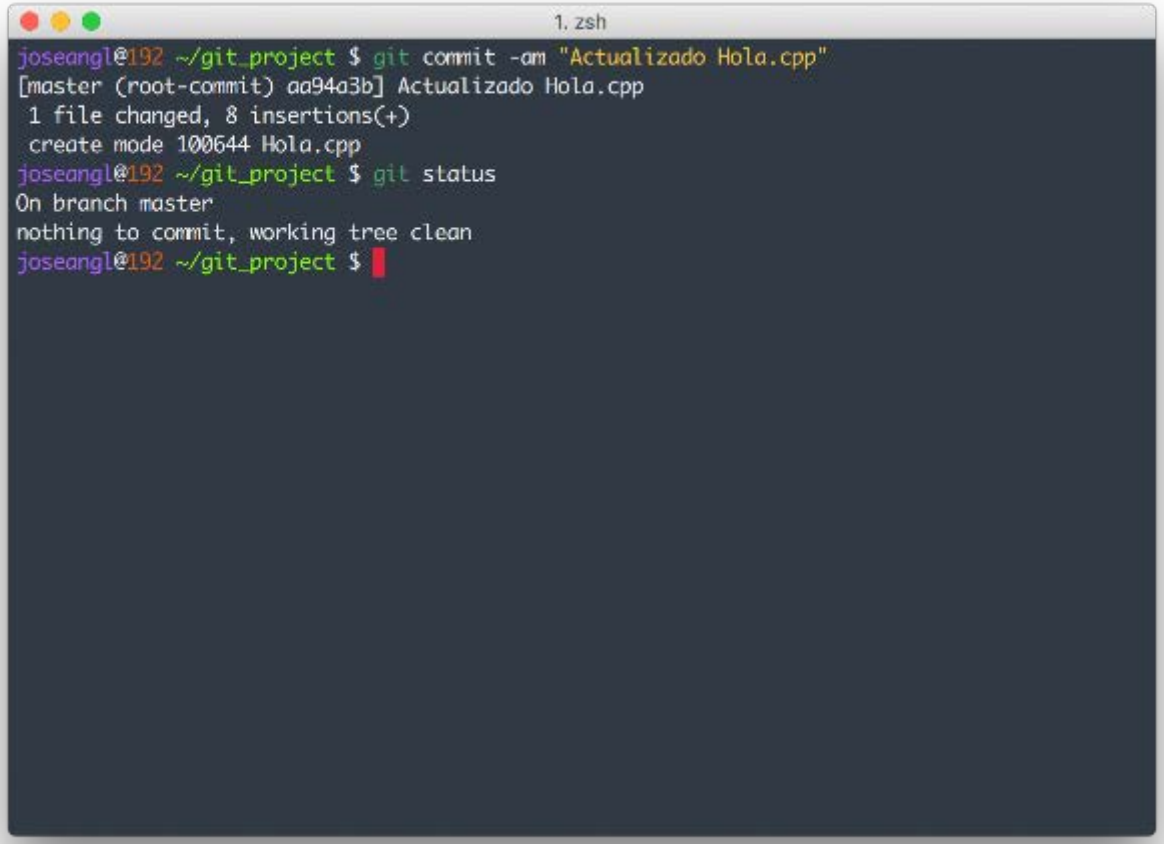
- Los cambios hay que confirmarlos para añadirlos al directorio de git.

```
git commit –m “comentario”
```

- El comentario da información de lo que hace un commit.
  - ¿Por qué has hecho este cambio?
  - ¿Qué cambios has hecho?
  - ¿Qué consecuencias esperas que tenga tu cambio?
- Añadir al *staging area* y commit a la vez:

```
git commit –a –m “comentario”
```

Ejemplo


1. zsh
joseangl@192 ~/git_project $ git commit -am "Actualizado Hola.cpp"
[master (root-commit) aa94a3b] Actualizado Hola.cpp
1 file changed, 8 insertions(+)
create mode 100644 Hola.cpp
joseangl@192 ~/git_project $ git status
On branch master
nothing to commit, working tree clean
joseangl@192 ~/git_project $

Viendo los cambios

- Se pueden ver los cambios de un fichero en el directorio de trabajo respecto a su última versión confirmada (commited).
- Para ello se usa la orden:

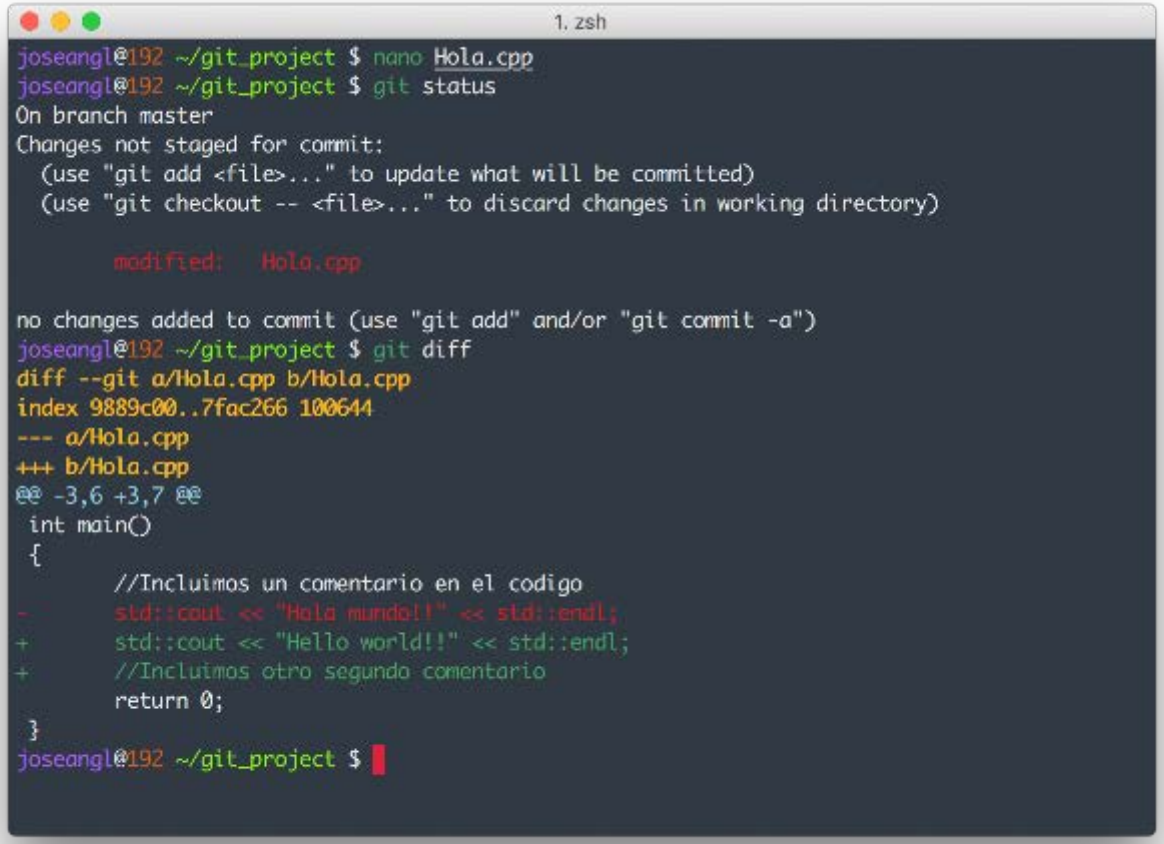
```
git diff [archivo]
```

- Salida del comando:

  Líneas borradas en rojo (con un - delante)

  Líneas añadidas en verde (con un + delante)

Ejemplo


1. zsh
joseangl@192 ~/git_project $ nano Hola.cpp
joseangl@192 ~/git_project $ git status
On branch 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: Hola.cpp
no changes added to commit (use "git add" and/or "git commit -a")
joseangl@192 ~/git_project $ git diff
diff --git a/Hola.cpp b/Hola.cpp
index 9889c00..7fac266 100644
--- a/Hola.cpp
+++ b/Hola.cpp
@@ -3,6 +3,7 @@
 int main()
 {
 //Incluimos un comentario en el codigo
- std::cout << "Hola mundo!!" << std::endl;
+ std::cout << "Hello world!!" << std::endl;
+ //Incluimos otro segundo comentario
 return 0;
 }
joseangl@192 ~/git_project $

Otras órdenes útiles

- Reemplazar cambios locales

```
git checkout -- <nombre del archivo>
```

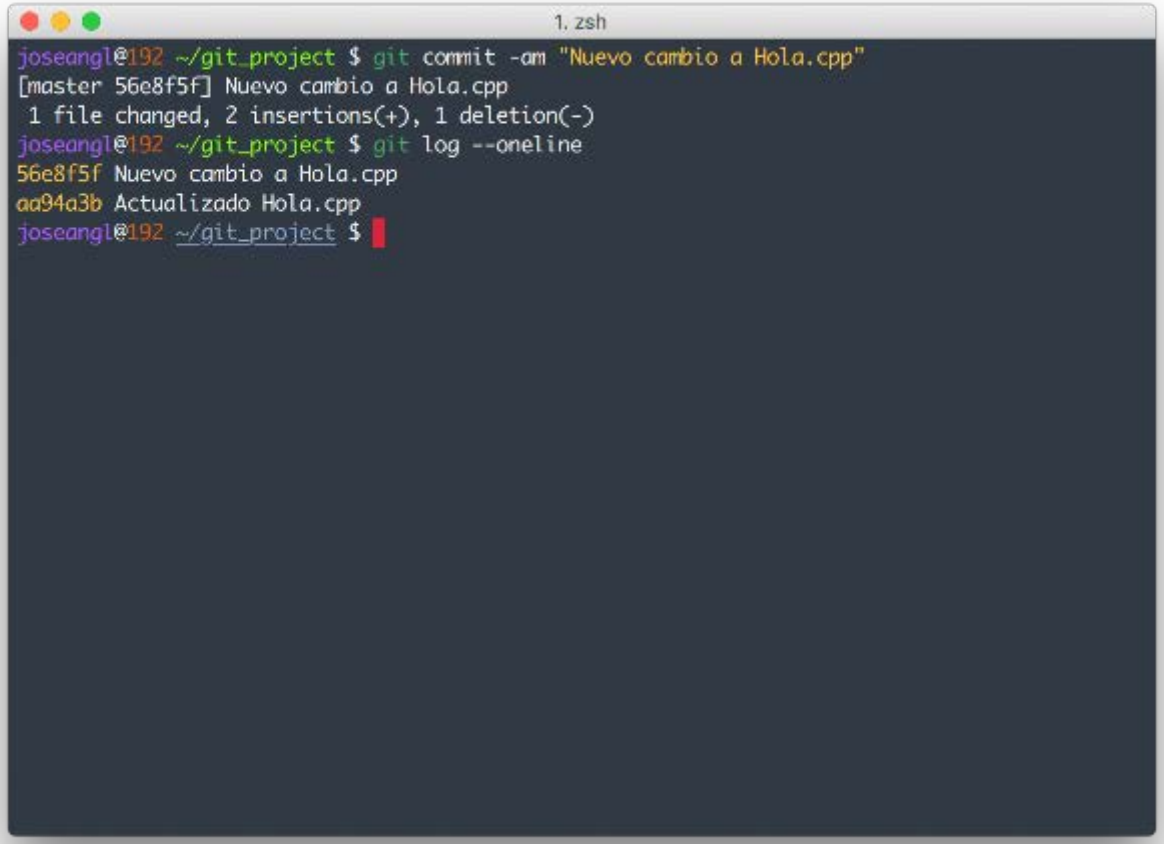
  - Esto reemplaza el archivo con su última versión confirmada.
- Para quitar un archivo del control de Git (dejar de seguirlo)

```
git rm <nombre del archivo>
```

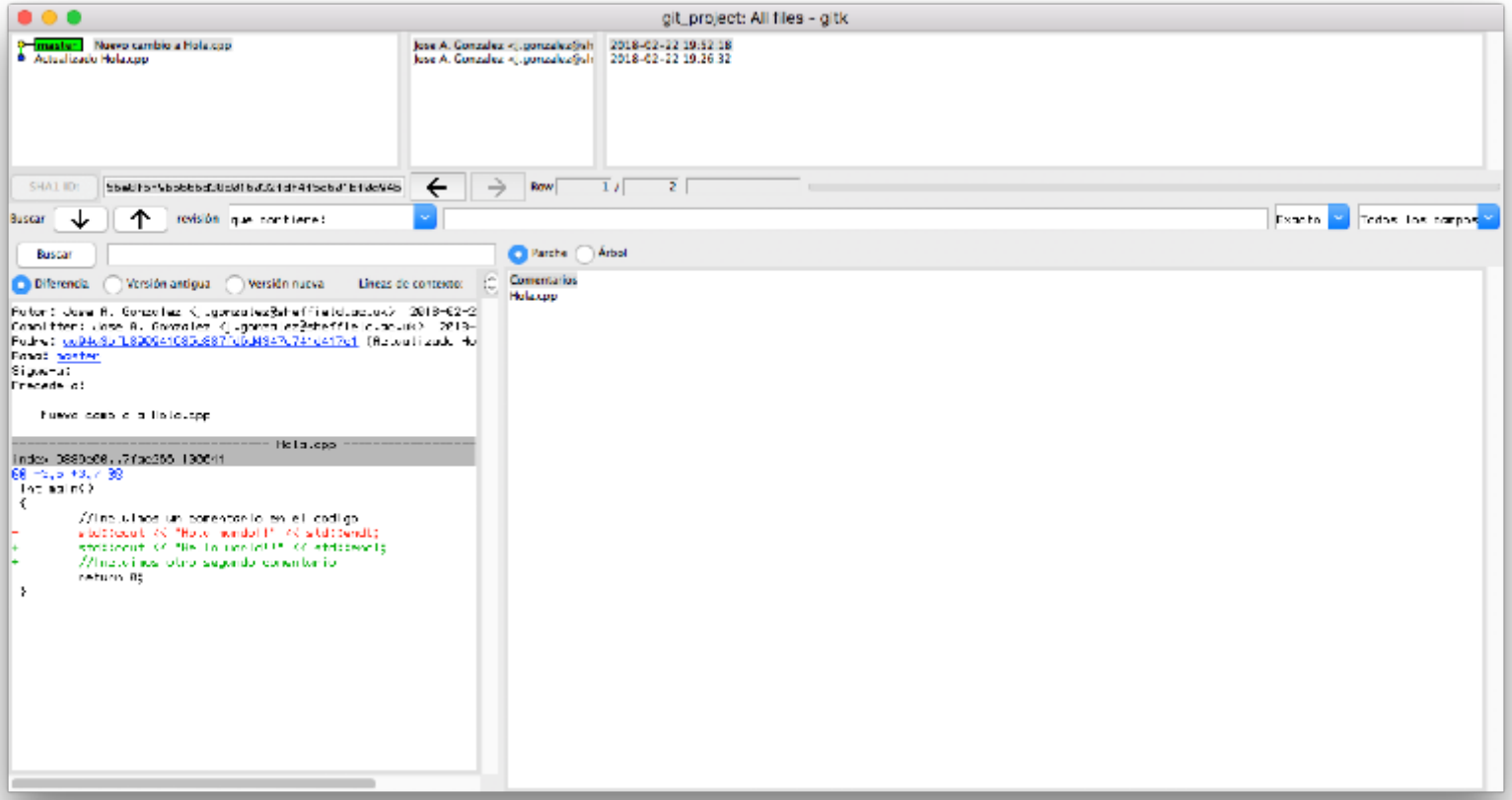
- Para no controlar un archivo pero dejarlo en el directorio de trabajo usamos el fichero
  - .gitignore
  - Se pueden especificar extensiones (p.ej. .exe, .o, .jar)
- Ver historial y los commits que se han realizado

```
git log --oneline
```

Ejemplo


1. zsh
joseangl@192 ~/git_project $ git commit -am "Nuevo cambio a Hola.cpp"
[master 56e8f5f] Nuevo cambio a Hola.cpp
1 file changed, 2 insertions(+), 1 deletion(-)
joseangl@192 ~/git_project $ git log --oneline
56e8f5f Nuevo cambio a Hola.cpp
aa94a3b Actualizado Hola.cpp
joseangl@192 ~/git_project $

Gitk


git_project: All files - gitk
master
Nuevo cambio a Hola.cpp
Actualizado Hola.cpp
Buscar
Diferencia
Versión antigua
Versión nueva
Parche
Árbol
Comentarios
Hola.cpp

Examinar commits previos

- Volver a un commit anterior.

```
git checkout <hash del commit>
```

  - Ejemplo: `git checkout f704f4a`
- Esta orden reemplaza los archivos del directorio de trabajo por la copia almacenada en el commit.
- Para volver al commit donde estuviésemos:

```
git checkout <nombre de la rama>
```

  - Ejemplo: `git checkout master`

Ejemplo

1. zsh

```
joseangl@192 ~/git_project $ nano Readme.md
joseangl@192 ~/git_project $ git add .; git commit -m "Añadido Readme.md"
[development 82c96e0] Añadido Readme.md
 1 file changed, 5 insertions(+)
 create mode 100644 Readme.md
joseangl@192 ~/git_project $ ls
Hola.cpp  Readme.md
joseangl@192 ~/git_project $ git checkout aa94a3b
Note: checking out 'aa94a3b'.

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 aa94a3b... Actualizado Hola.cpp
joseangl@192 ~/git_project $ ls
Hola.cpp
joseangl@192 ~/git_project $
```

Creando tags

- **Tags (etiquetas)**: sirven para crear alias a commits concretos con un nombre más *amigable*.
  - En lugar de referirnos a un commit con su hash lo hacemos usando su tag.
  - Se suelen usar para nombrar las diferentes versiones del proyecto.
- Creación:

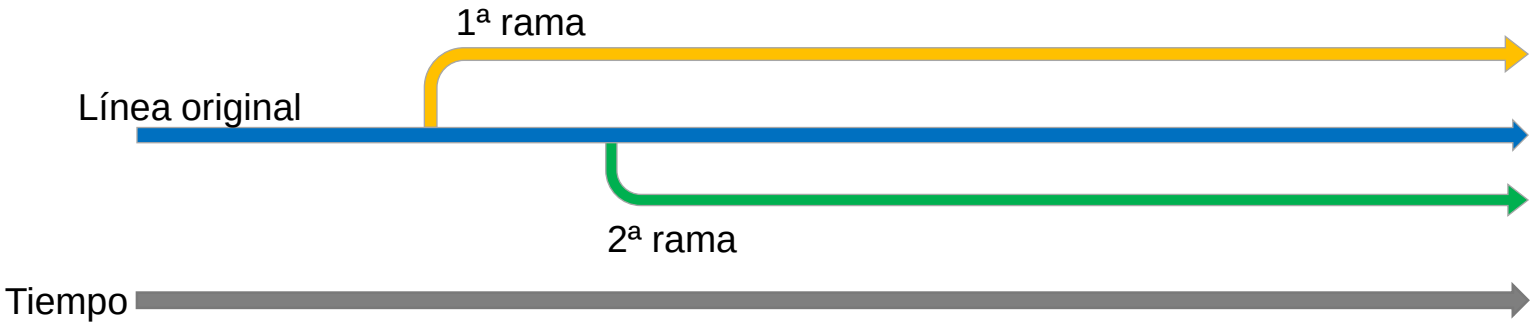
```
git tag <alias> <código del commit>
```

  - Ejemplo (commit actual): `git tag v1.0`
  - Ejemplo (commit antiguo): `git tag v1.0 f704f43`
- Ver la lista de tags:

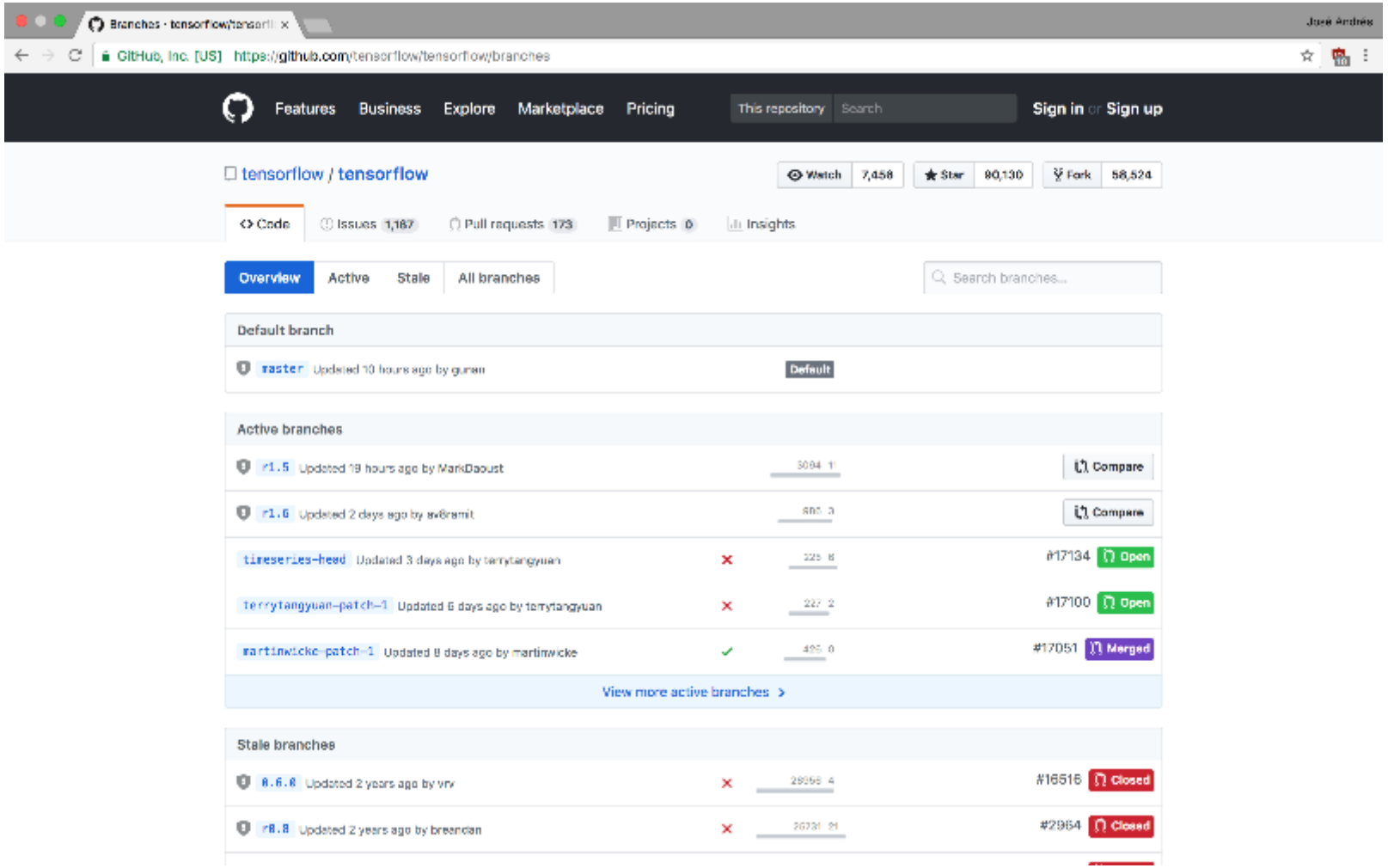
```
git tag
```

Ramas

- Una rama (*branch*) es una línea de desarrollo que existe de forma independiente de una anterior
  - Pero que parte de ella
  - Y posiblemente se fusionen posteriormente
- Se suelen usar para probar nuevas funcionalidades en el proyecto sin modificar la rama estable (master)



Ramas de TensorFlow en GitHub

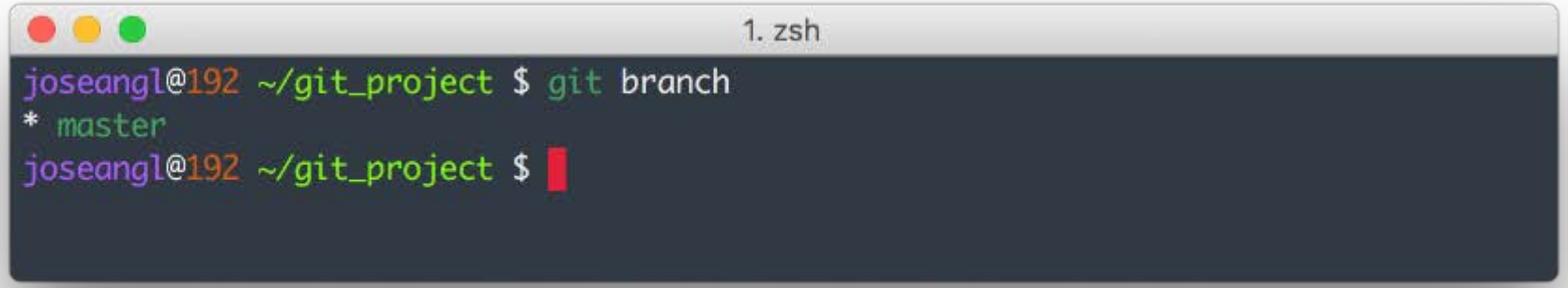


Ramas

- La rama principal en git se denomina por defecto `master` (se crea al crear el repositorio)
- Las ramas se muestran con el comando

```
git branch
```

- La rama activa se marca con un asterisco



Creación de nuevas ramas

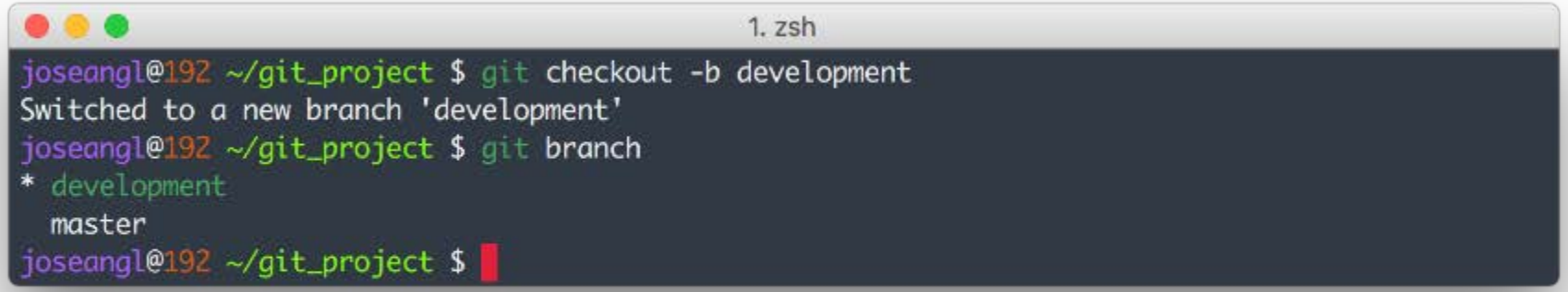
- Creación de una rama:

```
git branch <nombre de la rama>
```

- Cambiar a una rama:

```
git checkout <nombre de la rama>
```

  - Los siguientes *commits* afectarán a esa rama y no a otras.



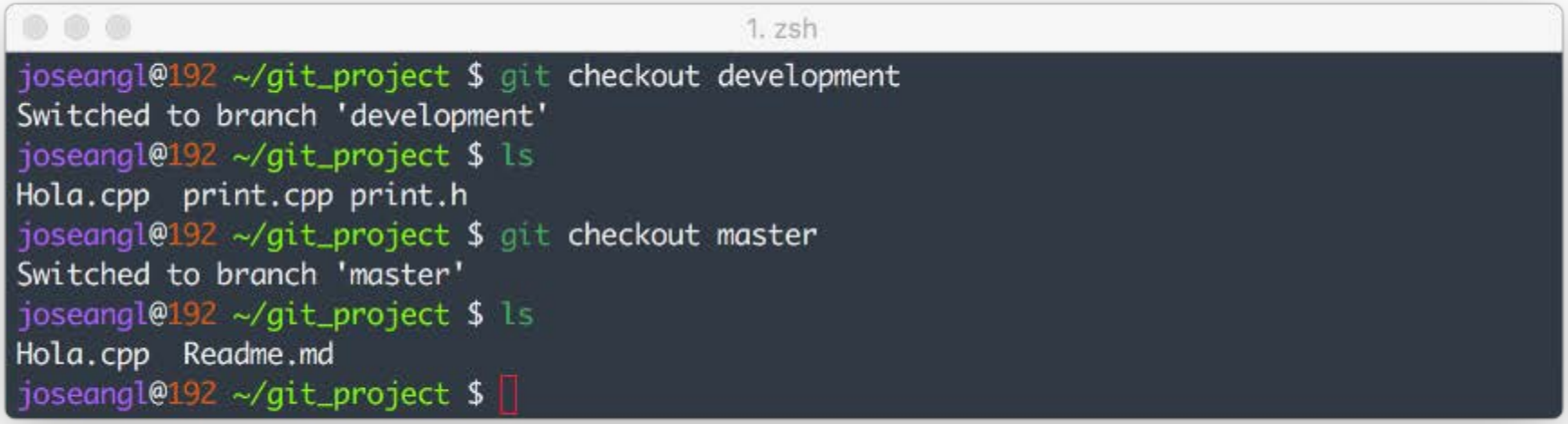
Añadiendo archivos a *development*

- Creamos dos nuevos archivos en *development*:
  - `print.h`
  - `print.cpp`
- El código de `Hola.cpp` queda de la siguiente forma:

```
#include "print.h”

int main()
{
	//Llamada a la función que imprime "Hello world!!"
	print_hello_world();

	return 0;
}
```

Añadiendo archivos a *development*

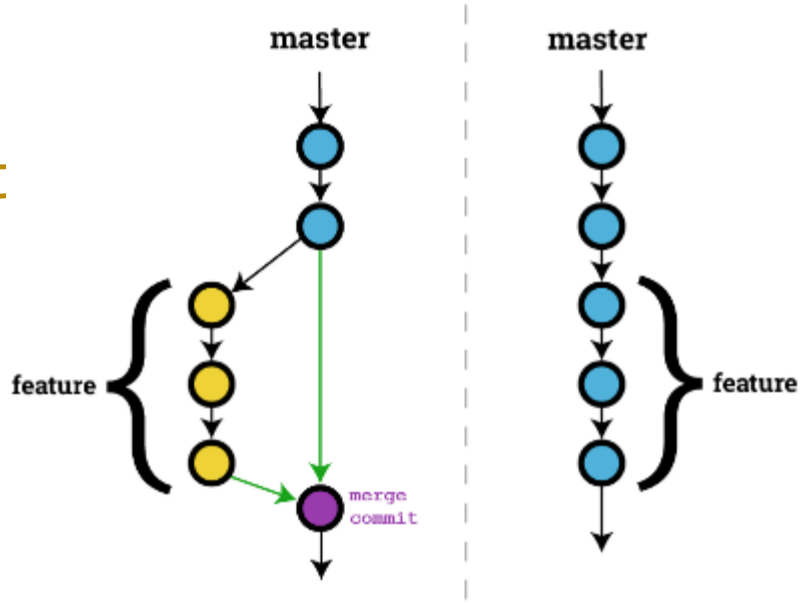
- Comprobamos que los nuevos ficheros sólo están en la rama *development*:



Fusionar ramas

- Cuando fusionamos dos ramas los cambios de una rama se añaden a la otra rama.
- **Ejemplo**: fusionar `development` y `master`
- La forma más sencilla de hacerlo es:

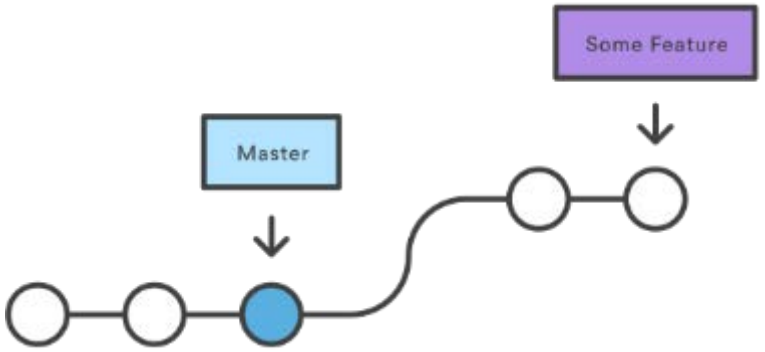
```
git checkout master
git merge development
```



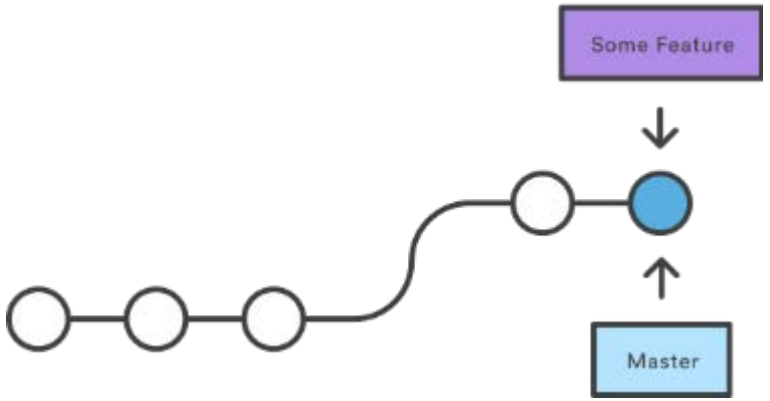
Conflictos al fusionar ramas

- Situación más simple (*fast-forward merge*)

Antes del merge



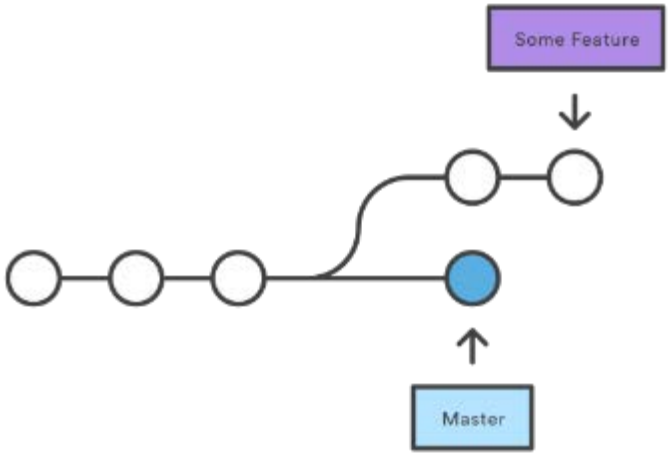
Después del merge



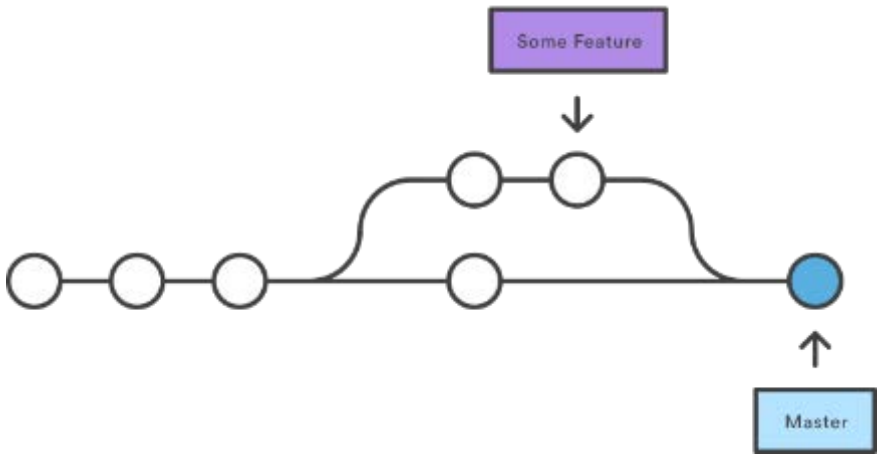
Conflictos al fusionar ramas

- Situación más compleja (*3-way merge*):

Antes del merge



Después del merge



Conflictos al fusionar ramas

- Varias situaciones posibles:
  - Merge de una rama en otra sin haber tocado los mismos archivos
    - Los cambios se hacen bien.
  - Se ha tocado el mismo archivo en ambas ramas pero editando partes separadas
    - Git detecta que no has tocado la misma región y se suelen integrar bien.
  - Se ha modificado el mismo archivo en ambas ramas y se ha modificado la misma región
    - Git genera un conflicto porque no sabe con qué modificación quedarse.
    - Git nos indica que reparemos estos problemas manualmente.
    - Una vez arreglados se vuelve a enviar.

Ejemplo

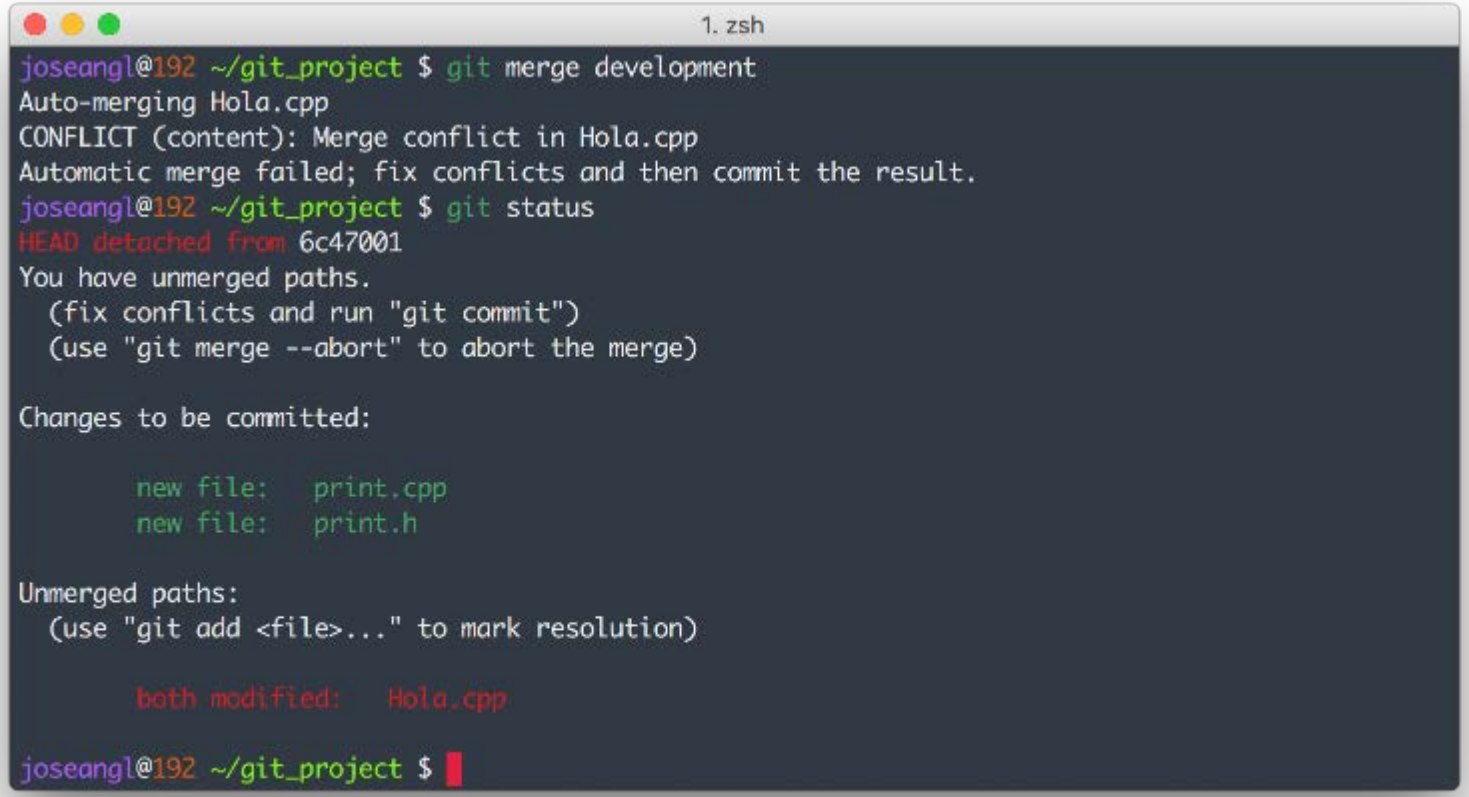
- Modificamos `Hola.cpp` en la rama `master`:

```
#include <iostream>

void print ()
{
          std::cout << "Hello world!!" << std::endl;
}

int main()
{
        print();
        return 0;
}
```

- Fusionamos las ramas:

```
git checkout master
git merge development
```

Ejemplo


1. zsh
joseangl@192 ~/git_project $ git merge development
Auto-merging Hola.cpp
CONFLICT (content): Merge conflict in Hola.cpp
Automatic merge failed; fix conflicts and then commit the result.
joseangl@192 ~/git_project $ git status
HEAD detached from 6c47001
You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)
Changes to be committed:
new file: print.cpp
new file: print.h
Unmerged paths:
(use "git add <file>..." to mark resolution)
both modified: Hola.cpp
joseangl@192 ~/git_project $

Conflictos al fusionar ramas

- Git edita el contenido de los archivos con conflictos con las siguientes marcas:
  - `<<<<<<<`
  - `=======`
  - `>>>>>>>`
- El contenido antes de `=======` se suele referir a la rama que recibe los cambios (`master`).
- **Resolución del conflicto**: editar manualmente el fichero con conflictos y ejecutar `git add <fichero>`.
- Finalmente se ejecuta un `git commit` para mezclar las ramas.

Ejemplo

- Fichero `Hola.cpp` después del `git merge`:

```
#include "print.h”

<<<<<<< HEAD
void print ()
{
        std::cout << "Hello world!!" << std::endl;
}

int main()
{
        print();
=======
        //Llamada a la funcion que imprime "Hello world!!"
        print_hello_world();
>>>>>>> development
        return 0;
}
```

Ejemplo

- Editamos el fichero

```
#include "print.h”

int main()
{
        //Llamada a la funcion que imprime "Hello world!!"
        print_hello_world();
        return 0;
}
```

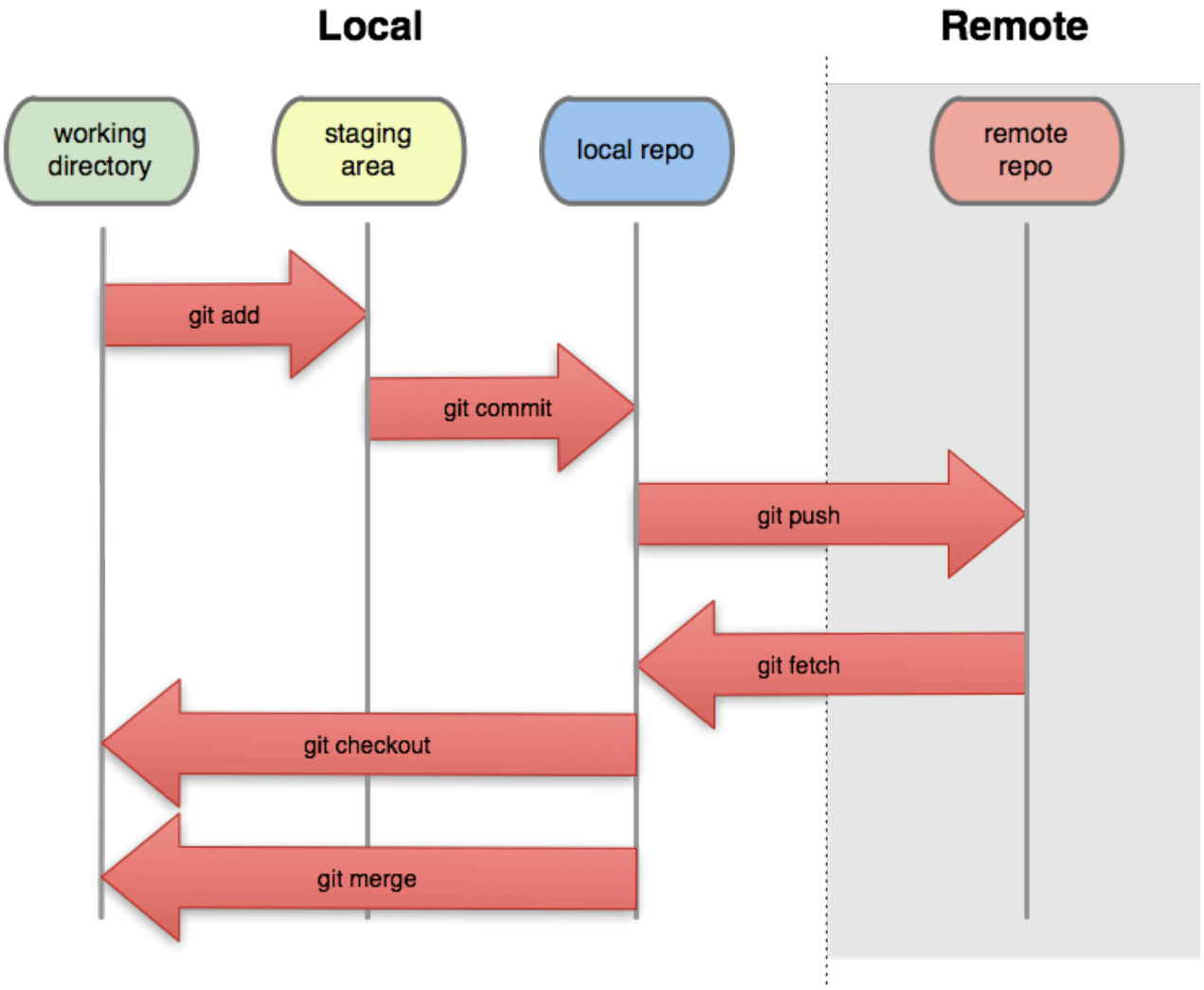
- Resolvemos el conflicto con:

```
git add Hola.ccp
git commit –m “Merge de master y development”
```

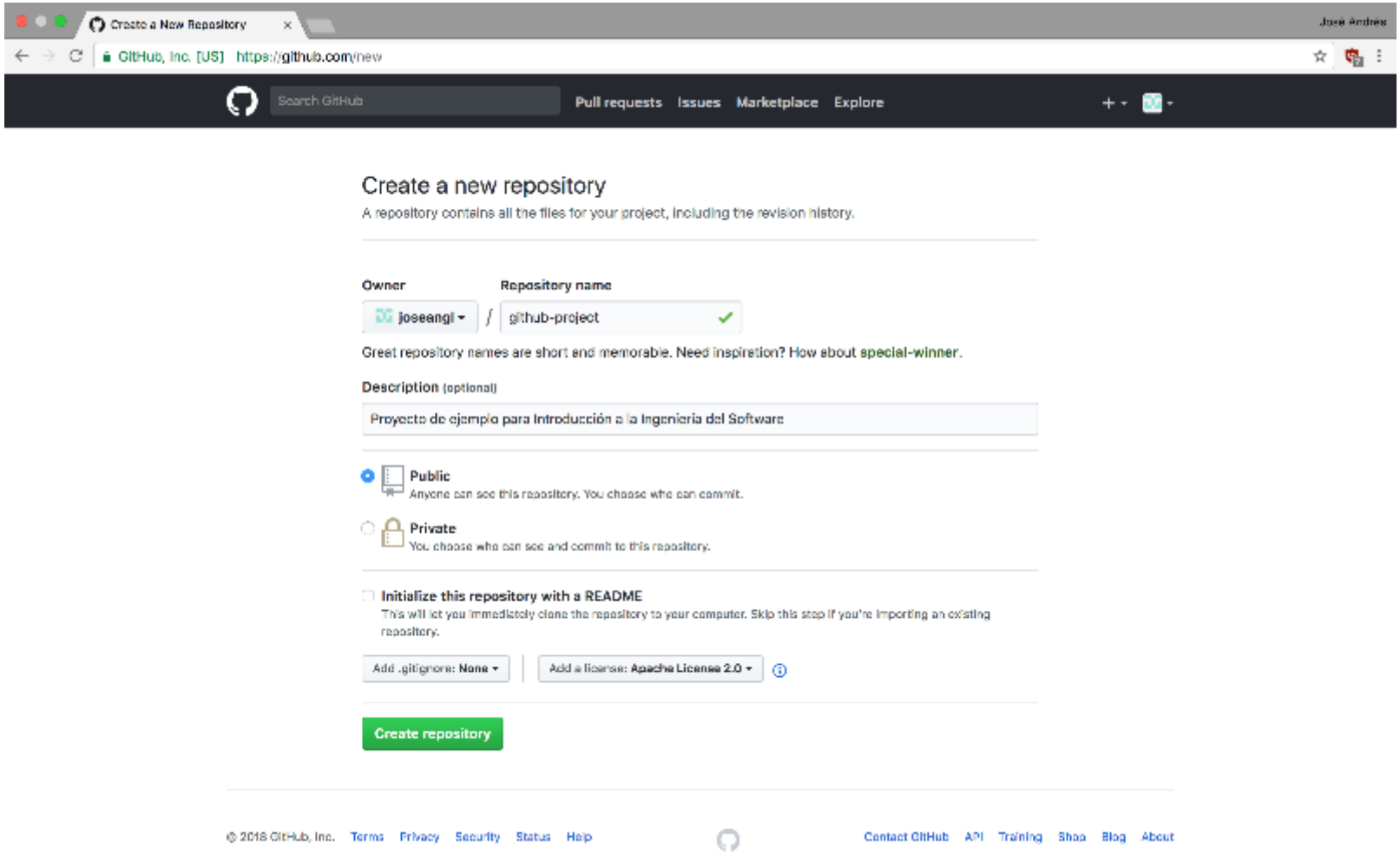
- Para eliminar la rama development:

```
git branch –d development
```

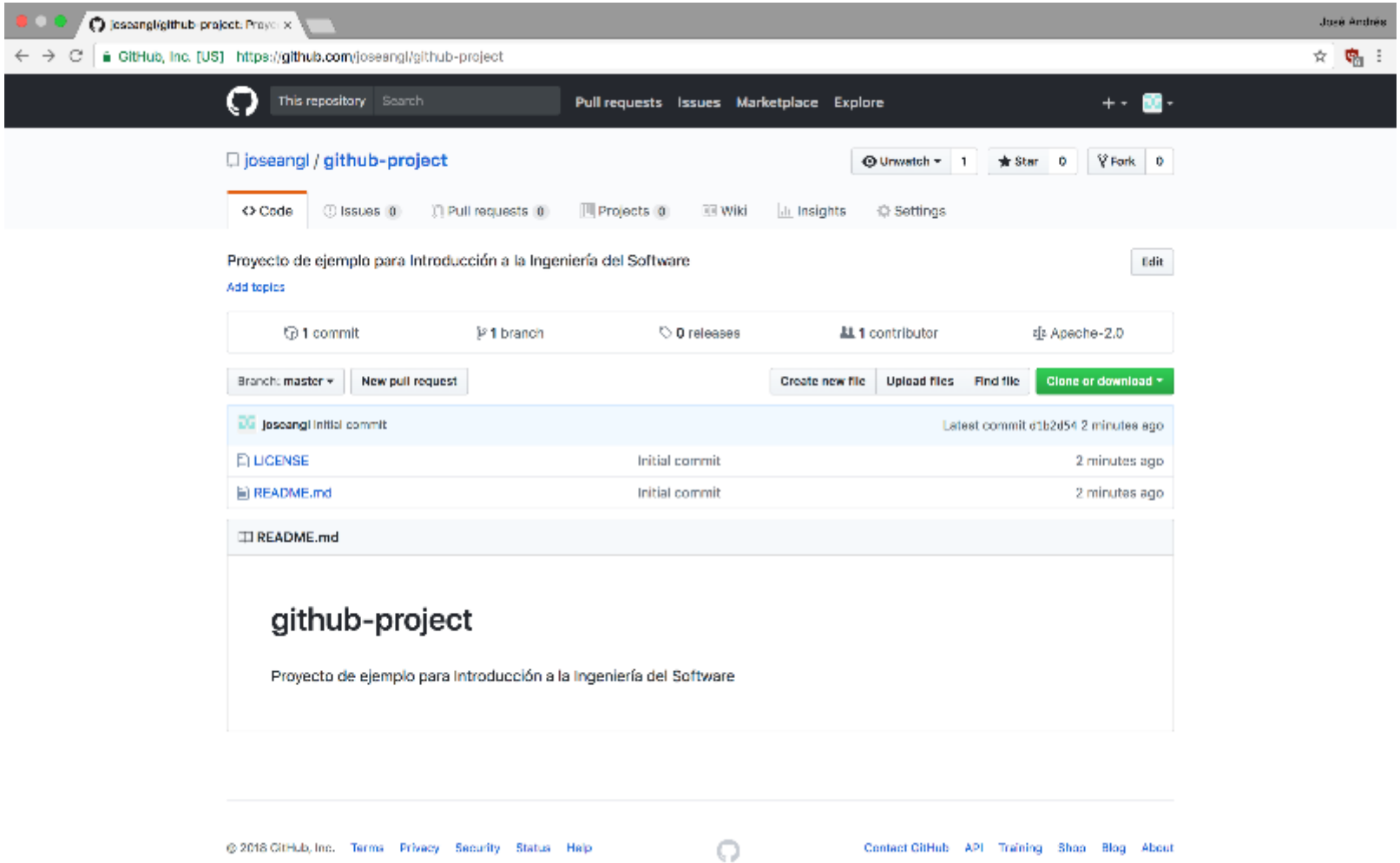
Git en modo remoto



Creando el repositorio en GitHub



Contenido del repositorio



Añadiendo GitHub como repositorio remoto

- En el directorio de trabajo ejecutamos:

```
git remote add <nombre remoto> <URL GitHub>
```

  - Al repositorio remoto se le suele denominar `origin`.
  - La URL nos la da GitHub en el botón Clone or download.
- Subir datos al repositorio remoto:

```
git push <nombre remoto> <nombre rama>
```

  - Ejemplo: `git push –u origin master`
- Descargar datos del repositorio remoto:

```
git pull <nombre remoto> <nombre rama>
```

  - Ejemplo: `git pull origin master`
  - Esta orden también fusiona los cambios del repositorio remoto en el local (merge) ➔ puede generar conflictos.

Ejemplo


1. zsh
joseangl@192 ~/git_project $ git remote add origin https://github.com/joseangl/practica-git.git
joseangl@192 ~/git_project $ git push -u origin master
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 641 bytes | 0 bytes/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://github.com/joseangl/practica-git.git
 * [new branch]   master -> master
Branch master set up to track remote branch master from origin.
joseangl@192 ~/git_project $

Otras órdenes útiles

- Sobreescribir el repositorio local con los datos del remoto:

```
git fetch origin
git reset –hard origin/master
```