Manual de Git

Introducción al Git

Git es un sistema de control de versiones. ¿Qué es un sistema de control de versiones (**VCS**)? Es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.

Git es un **sistema de control de versiones distribuido**. En este tipo de versiones, no nos bajamos la última versión de los archivos sino que replicamos, completamente, el repositorio.

Los conceptos fundamentales de Git son los siguientes:

- **Instantáneas, no diferencias**

  Git modela sus datos como un conjunto de **instantáneas**. Cada vez que se confirma un cambio, Git hace una **FOTO** del aspecto de **TODOS** los archivos y guarda una referencia a esa instantánea.

  En otros VCS lo que se hace es guardar las diferencias que tiene un archivo con respecto a la versión base.

- **Operaciones locales**

  Git es rápido, muy rápido. Como hemos dicho, cada uno de nosotros tiene una copia del repositorio en local, por lo tanto, la mayor parte de las operaciones se realizan sin necesidad de usar la red

- **Integridad**

  Cualquier almacenamiento en **Git** viene precedido por una suma de comprobación. Generación de hash. De esta manera, cada cambio es unívocamente identificable.

- **Añadir, sólo añadir**

  Casi todas las acciones que realizamos en Git añaden información a la base de datos del sistema. Es muy difícil, por lo tanto, perder información o hacer acciones que no se puedan deshacer.

- **Estados en Git**

  Git tiene tres estados en los que se pueden encontrar nuestros archivos. Esto es lo más importante a recordar para facilitar el proceso de aprendizaje. Los estados son:

  - **Modificado (modified)**. Hemos tocado el archivo pero, todavía, no lo hemos confirmado a nuestra base de datos local (no hemos hecho commit)
  - **Preparado (staged)**. Hemos marcado un archivo modificado para que vaya incluido en una próxima modificación (en un próximo commit)
  - **Confirmado (committed)**. El archivo está almacenado, de forma segura, en nuestra base de datos local.

En el siguiente enlace se encuentra un libro sobre **Git**.

http://git-scm.com/book/es/

Por favor, consultad el punto 1.3 para tener una visión más detallada sobre los fundamentos de **Git** antes de continuar:

http://git-scm.com/book/es/Empezando-Fundamentos-de-Git

Fundamentos de Git. Comandos principales

En este punto veremos los principales comandos de Git. Los vamos a ver desde el punto de vista de la consola, es decir, de línea de comandos. Animaros a utilizar la línea de comandos para realizar las operaciones de Git. Los diferentes GUIs y el plugin de Eclipse son opciones también válidas, pero, en ocasiones, ocultan lo que se está haciendo realmente. Si dominamos la consola, conoceremos la base de Git y sabremos lo que estamos haciendo. Por esta razón, en este punto explicaremos las cosas para línea de comandos.

En un punto posterior, explicaremos el manejo del plugin de Eclipse.

Este punto lo que recoge es un resumen de lo que se expone en el capítulo 2 del libro de Git. Su lectura es indispensable:

http://git-scm.com/book/es/Fundamentos-de-Git

en él encontraremos la manera de trabajar para deshacer cambios, cómo trabajar con repositorios remotos y algunos trucos y consejos útiles.

En el siguiente cuadro se recogen de manera resumida, los principales comandos **Git**:

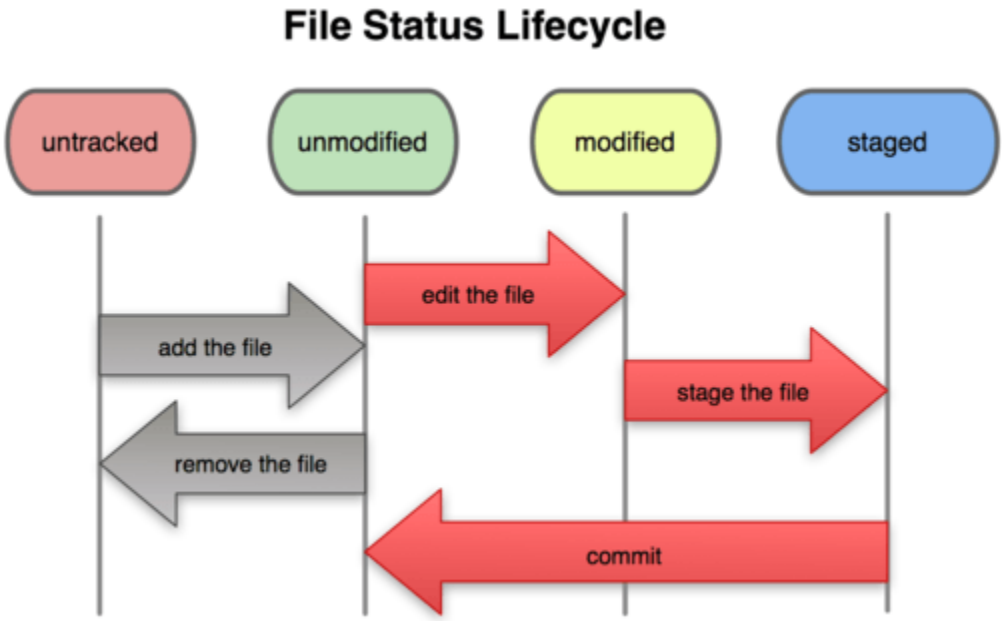
Comandos Git

$ git init	Inicializa un repositorio en un directorio existente
$ git clone <url>	Clonar un repositorio remoto situado en [url]
$ git comando --help	Abre el navegador con la página de ayuda. Si pones sólo **$ git** te da una lista con los comandos disponibles
$ git add <archivo>	Añade archivo al control de versiones También se usa para añadir un archivo al área de preparación, nombre que recibe el lugar en el que se sitúan los archivos que se conxolidarán con el siguiente commit.
$ git status	Nos dirá cual es el estado de nuestros archivos. Hay que acostumbrase a realizarlo de manera continuada.

$ git commit	Consolida los cambios en estado "staged". Si ponemos: **$ git commit -m "Commit inicial"** Nos asocia este comentario al commit Si no ponemos nada, se nos abrirá nuestro editor por defecto para introducir el comentario **$ git commit -a** Confirma todo archivo cambiado, incluso aquellos que no hayamos pasado a "staged" con add.
$ git diff	Nos indica lo que ha cambiado exactamente en los archivos. status te lista el nombre de los archivos cambiados, diff te indica qué partes de esos archivos han cambiado.
$ git rm	Prepara un archivo para eliminarlo
$ git mv	Renombrado de archivos
$ git log	Nos muestra el histórico de commits
$ git remote	En un directorio de repositorio local, nos mostrará los repositorios remotes que están configurados
$ git fetch <remote-name>	Nos bajamos del remoto las ramas
$ git pull	Recupera e intenta unir la rama remota con la rama en la que estás trabajando actualmente
$ git push <remoto> <rama-local>	Ya tenemos el proyecto en local en un estado que necesitamos compartir con otros usuarios, lo subimos al repositorio remoto con **push** **$ git push origin NEW_BRANCH**

$ git branch	Listado de los que tenemos en local
$ git branch -v	Listado con el último comentario
$ git branch --merged	Para ver las ramas fusionadas
$ git branch –no-merged	Para ver las no fusionadas
$ git branch –d NOMBRE_RAMA_LOCAL	Borra la rama
$ git checkout NOMBRE_RAMA_LOCAL	Apuntamos al que queramos
$ git checkout –b NEW_BRANCH	Crea una rama, copia del que apuntábamos, y apunta a ella.
$ git push origin :NEW_BRANCH	Borra del origin (nuestro remoto), el branch con ese nombre
$ git merge NEW_BRANCH	Estando posicionados sobre la OLD_BRANCH, este commando fusionará el OLD con el NEW.
$ git reset –merge	Cancelará los conflictos
$ git rebase NEW_BRANCH_II	Los cambios confirmados de una rama los mete en otra
$ git pull NEW_BRANCH	Fetch + Merge
EN EL REMOTO -> **git –-bare init**	Como el init en local

En este gráfico, se detalla el ciclo de vida de un archivo en Git:



Comparación Git vs Subversion

Git trabaja tanto de forma remota como en nuestra propia máquina. Así, estaremos hablando de repositorio remoto y de repositorio local.

Lo que hasta ahora con SVN han sido *Commits*, ahora serán *Commits* pero sólo a nivel LOCAL, es decir, el resto de desarrolladores no se enterarán de ellos.

La consolidación de nuestros Commits (locales) en el repositorio remoto los denominaremos PUSH.

Caso práctico. Importando el proyecto Producción

Utilizando la consola

Nos situamos en el directorio que queremos sea nuestro REPO_LOCAL, por ejemplo D:\GIT

$ git init

Añadimos el REPO_REMOTO

$ git remote add origin apps@guru-app.empresa:/opt/data/git/produccion/produccion.git

Ahora clonamos el REPO_REMOTO

$ git clone **apps@guru-app.empresa:/opt/data/git/produccion/produccion.git**

Nos crea el directorio produccion y, entrando en él, ya podemos ejecutar todos los comandos git que sean necesarios.

Estrategia de "branching" del proyecto Produccion. Git Flow

Todos sabemos lo que es un branch dentro de un proyecto de desarrollo. Los branches se crean para desarrollar funcionalidades aparte de la rama principal de desarrollo.

Uno de las características más potentes de Git es la facilidad que nos proporciona para crear ramas. Podemos trabajar con ramas, ramas y más ramas. Es más, debemos trabajar con ramas, ramas y más ramas. Para cualquier desarrollo que queramos hacer, lo mejor es que nos creemos una rama para realizarlo. Hay que perder el miedo a conceptos como branches y merging.

De todas formas, esta potencia para crear ramas "como churros", también puede ser un inconveniente si no establecemos unas reglas de juego que deben regir el trabajo de todo el equipo. Somos un equipo bastante grande y es muy importante que todos tengamos el mismo modo de hacer las cosas. Tampoco tienen que ser unas reglas estrictas, deben tener la suficiente flexibilidad para que podamos responder a situaciones especiales que pueden surgir en el día a día.

El modelo que vamos a utilizar es un modelo de flujo de trabajo creado por Vincent Driessen en 2010. Es un modelo que da mucha importancia a las ramas, de hecho, las crea de varios tipos, de forma temática, tal que cada tipo de rama es creada con un objetivo en concreto. Vamos a ver qué tipos de ramas tenemos y para que usaremos cada una de ellas. Por último, presentaremos una herramienta que nos ayudará a gestionar este modelo (**git-flow**), y aplicaremos este modelo a nuestro flujo de trabajo habitual, delimitando las responsabilidades y las acciones a realizar.

Antes de empezar, una cuestión de convenio. Cuando nos referimos a ramas remotas, las llamaremos anteponiéndoles el nombre del repositorio. Así, origin/stable, será la rama stable situada en el repositorio remoto

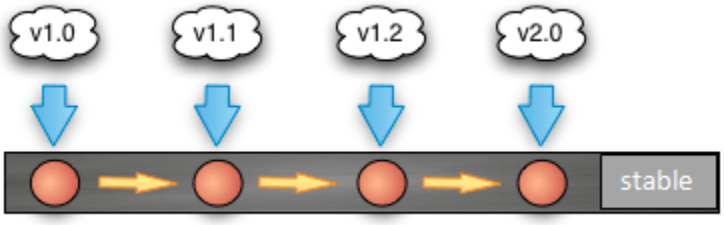
Tipos de ramas

El trabajo se organiza en dos ramas principales, la rama stable y la rama master (si consultáis cualquier tipo de documentación sobre el tema, tened cuidado porque a la stable se le llama master y a la master se le llama develop

Rama Stable

Esta rama (sincronizada en todo momento con **origin/stable**) tiene como objetivo ser el contenido del servidor de producción. Es decir, el **HEAD** de esta rama ha de apuntar en todo momento a la última versión de nuestro proyecto. Entendemos por HEAD el último commit existente en la rama de la que hablamos.

Cualquier commit que pongamos en esta rama debe estar preparado para subir a producción. Es decir, cada vez que se incorpora código a **stable,** tenemos una nueva versión para real.



Como podemos ver, cada commit (ilustrado por las pelotitas) es una nueva versión del proyecto, que a su vez está referenciada mediante un tag (representada por la nube).

No se va a desarrollar desde esta rama en ningún momento.

Sobre esta rama sólo podrán hacer commits los responsables de las subidas. En nuestro caso, los **responsables funcionales**.

Rama Master

Esta rama funciona paralelamente a la **stable**. Si la anterior contenía las versiones desplegadas en producción, esta (que también estará sincronizada con **origin/master**) contendrá el último estado de nuestro proyecto. Es decir, esta rama contiene todo el desarrollo del proyecto hasta el último commit realizado.

Cuando esta rama adquiera estabilidad y queramos lanzar una nueva versión, bastará con hacer un **merge** a la rama **stable** (no de forma directa, ya veremos como hacerlo) y asignarle un número de versión mediante un **tag**.

Esto será lo que cree una nueva versión de nuestro proyecto. Recordemos que queremos ser bastante estrictos en cuanto a que el **stable** solo alojará **commits** que supongan nuevas versiones. Nada más. Para ver el estado del desarrollo usaremos la rama **master**.

Esta rama será la que utilizaremos todos para subir nuestros desarrollos que tienen que ir en la siguiente subida semanal. Ojo, no se desarrolla directamente en la rama **master** que tenemos en local, lo que hacemos es crearnos un **branch** local a partir de **master**, desarrollar en él y, una vez que hayamos acabado, subir nuestros cambios a nuestro **master** (merge) y, posteriormente, al **origin/master** remoto. Veremos, posteriormente, cómo hacer esto.

Podemos pensar en esta rama como aquella que contiene lo que está preparado para subir a producción en la siguiente subida. Aquí nos surge la siguiente pregunta, ¿qué ocurre con los desarrollos que duran más tiempo que el que trascurre entre dos subidas semanales? No los podemos incluir en la rama master porque si no, al hacer el merge con stable, nos llevaríamos ese código que no está preparado. Para esto acuden a nuestro rescate otro tipo de ramas, las **ramas features**.

Ramas Feature

También denominadas ramas topic. Estas ramas se utilizan para desarrollar nuevas características de la aplicación que, una vez terminadas, se incorporan a la rama **master**. Además, estas características cumplen que su tiempo de desarrollo excede el tiempo que trascurre entre dos subidas semanales.

Estas ramas pueden tener cualquier nombre que no empiece por **release/**, **hotfix/**, ni **master** ni **stable**. Sería bueno adoptar la convención de que su nombre empiece por **feature/**. El nombre deberá reflejar el propósito de la rama (ej. **feature/reconew**).

NOMENCLATURA DE RAMAS

Estamos comentando que la nomenclatura a utilizar es anteponer el tipo de rama con un guión al nombre de la funcionalidad:

feature/reconew

En lugar de esta convención, y como explicamos en el apartado de git-flow (pulsa aquí), es mejor utilizar el enfoque de carpetas:

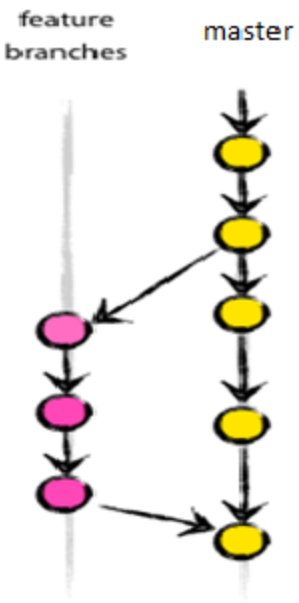
feature/reconew

Estas ramas pueden estar en local, si sólo es un desarrollador el que implemente la funcionalidad. Si el equipo que ataca el proyecto que implica la funcionalidad es de dos o más desarrolladores, se subirían al repositorio remoto. En dicho repositorio, serían referenciadas como **origin/feature/reconew**.

Estas ramas, tienen las siguientes características:

- Se originan a partir de la rama master.
- Se incorporan siempre a la rama master.

Gráficamente:



Cuando el desarrollo de la funcionalidad se alargue mucho en el tiempo, quizá sería bueno pensar en consolidar la rama master en estas ramas de features. El momento perfecto para hacerlo será cuando se realice el merge de las ramas de lanzamiento. Como comento, este será un tema para ver cuando se nos presente la ocasión.

Una vez finalizada la tarea, solo tendremos que integrar la rama creada dentro de **master**. Para esto usaremos un **merge** normal y corriente con un parámetro extra, **--no-ff**.

Este flag obliga a Git ha generar un commit para el merge. Con esto se evita que Git haga un **fast-forward** si es posible (es uno de los métodos para realizar merges, en los que se pierde la historia de la rama).

De esta manera, en todo momento en la historia del repositorio se tendrá constancia de que hubo una rama donde se desarrolló cierta funcionalidad, que **commits** contenía, y cuando se integró en el **trunk o master**.

Nuevamente, veremos más adelante lo que tenemos que hacer en cuanto a comandos.

Ramas Release

Como hemos dicho antes, el desarrollo del proyecto ha de realizarse en la rama **master**, para posteriormente lanzar una nueva versión desde la rama **stable**.

Esto no se hará directamente con un **merge** desde la rama **master** a **stable**, si no que se usarán las ramas de lanzamiento para este fin.

Este tipo de ramas sirve para poder liberar cuanto antes la rama **master** para continuar el desarrollo, y alojará todos aquellos **commits** que son de preparación para el lanzamiento de la versión: cambiar el número de la versión en los ficheros, compilar la documentación necesaria, empaquetar librerías (por ejemplo, una nueva versión de informa-xmlbeans), corregir los últimos bugs.

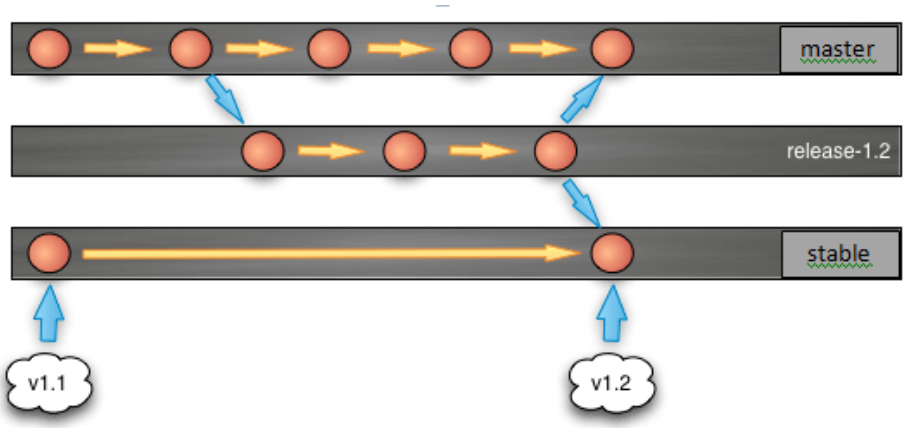
Es decir, todo aquello que no tiene que ver directamente con el desarrollo, si no con el lanzamiento de la siguiente versión del proyecto.

La creación de esta rama será análoga a las de otros tipos, teniendo en cuenta que tendrá que prefijarse con el nombre **release/**, seguido del número de versión que tendrá. Por ejemplo:

```
$ git checkout -b release/PRODUCCION_3_0_4 master
```

Una vez terminada la preparación, habrá que realizar un **merge** del **HEAD** de la rama **release** con el **stable**. De esta forma se creará una nueva versión.

Por último, se creará un tag en el commit del master para marcar la versión.



Como se puede ver en el gráfico, los cambios no sólo se integrarán en el **stable**, si no que también lo harán en la rama **master**, de esta manera no perderemos los cambios realizados en la rama **release**, y se integrarán en el desarrollo. Por último, y dado que ya no hace falta, podemos borrar la rama creada.

Como detalle adicional mencionar que en las ramas de lanzamiento no se puede añadir funcionalidad al producto. Es una rama cuyo objetivo es únicamente el realizar el trabajo necesario para lanzar una nueva release.

Si en este proceso se detectase un bug nuevo, se corregiría en la rama de lanzamiento. Al finalizarla, como el trabajo se integrará en la rama **master**, el commit que arregló dicho fallo se incluirá igualmente.

Veremos, posteriormente, los comandos a utilizar para gestionar esta operación.

En nuestro trabajo diario, la secuencia de trabajo será la siguiente:

- Las subidas semanales se hacen los Martes a las 15:30
- Los Martes, a primera hora, se generará la rama **release** a partir de **master.** De esta manera, todo el mundo podrá ya utilizar la rama **master** para seguir desarrollando funcionalidades que se integrarán en la siguiente subida semanal.
- Se realizará, en release, todo el trabajo para despliegue por parte del encargado de la subida.
- Se corregirán los últimos bugs, si surgen, sobre la rama **release**.
- El responsable de la subida hará el **merge** sobre **stable** y **master**, así como el **tag** de **stable**.

Ramas Hotfix

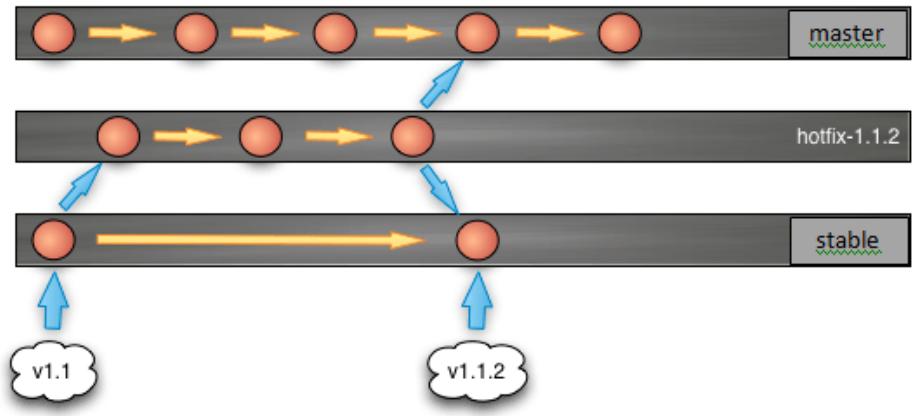
Esas ramas se utilizan para corregir errores y bugs en el código en producción. Funcionan de forma parecida a las Releases Branches, siendo la principal diferencia que los hotfixes no se planifican.

Esta rama se creará a partir de **stable** ya que únicamente queremos resolver el fallo (sin incluir nada del nuevo desarrollo realizado en la rama **master**). Su nombre deberá prefijarse mediante **hotfix/** y una vez creada la rama simplemente se corregirá el fallo desde la misma.

```
$ git checkout -b hotfix/PRODUCCION_3_0_4_1 stable
```

Una vez corregido el fallo tendremos que integrar el arreglo en el **stable**. Para ello, y como siempre, se hará un merge y se creará un tag para indicar que se ha creado una nueva versión del proyecto.

También se tendrá que integrar la corrección del bug en la rama **master**, ya que no queremos perder el fix en el desarrollo de futuras versiones.



Los comandos necesarios para llevar a cabo la finalización de la rama hotfix los veremos posteriormente.

Hay que tener en cuenta una pequeña excepción, el caso en el que exista una rama de lanzamiento cuando creamos la rama hotfix.

En este caso, se hará todo exactamente igual con una salvedad. Y es que a la hora de integrar los commits de la rama hotfix, por un lado lo haremos al stabley por el otro, en vez de a la rama master, lo haremos a la rama de lanzamiento.

De esta manera, el bug se verá corregido en la nueva versión que está a punto de lanzarse, y, cuando esta se integre en **develop** (recordad que las ramas de lanzamiento se integran en el **stable** y en **master** al finalizarse), se verá corregido en el resto del desarrollo.

Cambiar nombre de la rama

¡¡¡Me he confundido a la hora de crear el nombre de mi rama!!!

¿Si después de lo visto anteriormente resulta que hemos creado nuestra rama con distinta nomenclatura y ya estamos operando con ella tenemos que borrarla?

¡No!

Podemos cambiar su nombre con el siguiente comando: **git branch -m <*nombre-rama-anterior*> <*nombre-rama-nuevo*>**

Git flow

Es una herramienta creada para facilitar el seguir este flujo de trabajo. Como habréis visto, las formas de crear y finalizar las ramas son siempre prácticamente iguales, y, aparte, en el caso de la integración de los cambios, hay que pasar el flag no-ff siempre al comando. git-flow nació

para evitar errores en estos aspectos y para facilitar el día a día al seguir este flujo (ejecutar un solo comando para finalizar una rama, en vez de cinco por ejemplo).

Bring This To Life

Bueno, pues ya conocemos la teoría. Ya sabemos lo que son las ramas, sabemos que son nuestras amigas y sabemos que no sólo podemos, sino que debemos crear ramas para todo. Las ramas no son mogwais, no importa que tengamos cientos, miles de ellas (eso sí, no les deis de comer después de media noche). Así que, una vez que tenemos la teoría, expliquemos la práctica, como dijo el doctor Frankenstein "Bring it to life!"

Podríamos explicar la práctica de muchísimas maneras, pero creo que lo mejor es que dibujemos las situaciones con las que nos podemos encontrar en el día a día del proyecto de Producción. Para cada una de estas situaciones, expondremos las acciones a realizar, los comandos mediante los que se realizan dichas acciones y los actores que intervienen en estas pequeñas representaciones.

Antes de empezar, grabaos una palabra en la cabeza CONSOLA (no le añadáis sufijos, guarros). Tenemos que acostumbrarnos a trabajar con la consola Git Bash, no es tan bonito como el plugin de Eclipse, pero es más seguro y, sobre todo, sabemos que estaremos haciendo lo que queremos hacer.

Desarrollos para la subida semanal. Estilo Juan Palomo

Esta es la más común de las situaciones.

Andrés tiene que incluir el desarrollo necesario para una petición (por supuesto, urgente), que le han hecho desde balances. Estudiamos la petición y vemos que es abordable para incluirla en la futura subida semanal, ¿cómo debería actuar Andrés?

Crear un branch a partir del master y situarse sobre él, previamente nos traemos todo lo que haya en el master

```
$ git pull origin master
```

> Si sólo hacemos git pull, nos bajamos todo lo que hay en el repositorio, no sólo el master, e igual no interesa

```
$ git checkout -b feature/its_up_to_u master
```

Es conveniente nombrarlo feature/ y algo representativo de lo que la funcionalidad incluye.

Desarrollaría su funcionalidad sobre este branch, realizando los commits que fueran necesario y realizando sus pruebas en local, lugar en el que suele funcionar. Es muy recomendable ir viendo el estado de nuestro desarrollo bajo Git, usar, siempre que podáis,

```
$ git status
```

Nos dirá qué hemos hecho, qué hemos tocado, qué tenemos pendiente de commit, qué tenemos pendiente de añadir, etc..

```
$ git commit -a
```

La opción -a nos evita tener que añadir, individualmente al estado "staged", todos los ficheros que toquemos

Una vez que Andrés está seguro de que funciona, lo siguiente que hacemos es realizar el merge con el master. Es un proceso en dos partes, hacemos un pull del master para traernos todos los cambios y un merge con nuestra rama de feature.

```
$ git checkout master
$ git pull origin master
```

Si sólo hacemos git pull, nos bajamos todo lo que hay en el repositorio, no sólo el master, e igual no interesa

```
$ git merge --no-ff feature/its_up_to_u
```

> ⚠ OJO
>
> Siempre opción **--no-ff**, esto obliga a GIT a crear un commit para el merge y nos asegura que se respeta la historia de la rama feature.

Y, por último, subimos los cambios al repositorio con un push y borramos la rama feature

```
$ git push origin master
```

> Si sólo hacemos git push, subimos todo lo que tenemos en local al repositorio, incluyendo la rama que hemos usado para desarrollar, y que bien podríamos llamado feature/minabo. Ahora os explicáis algunas cosas extrañas, ¿no?

```
$ git branch -d feature/its_up_to_u
```

Y nada más, funcionalidad implementada. Un Juan Palomo en toda regla. Un solo desarrollador se lo guisa y se lo come, no es necesario subir el branch al remote porque no tenemos que hacer desarrollo en equipo.

Ahora, sobre el master, podemos generar el WAR para tango y decirle a los usuarios que nos validen la funcionalidad. Esto nos abre dos escenarios nuevos:

- Andrés es un monstruo del desarrollo y las pruebas se desarrollan sin incidencias. Caso de uso Juan Palomo cerrado
- Andrés es un zoquete, en local funcionaba o "me faltan especificaciones". Iteraremos, de nuevo, el caso de uso Juan Palomo, nuevo branch desde el master, nuevos commits, nuevo merge, nuevos push y nueva versión. Y, así, hasta que la funcionalidad esté validada. Como veréis, si fallamos cinco veces, tendremos cinco ramas feature creadas y borradas en local. ¿Podemos usar la misma y borrarla sólo cuando me validen mi desarrollo? Sí, podemos hacerlo.

Desarrollos para la subida semanal. Estilo Make a Team

Trabajo en equipo para funcionalidades de la subida semanal

Nacho tiene que incluir el desarrollo necesario para cambiar la generación de solicitudes de evaluación desde Producción. Este desarrollo afecta a entorno, módulo en el que hay que hacer cambios, por lo que Diego participará, activamente, en él. Estudiamos la petición y vemos que es abordable para incluirla en la futura subida semanal, ¿qué harán Nacho y Diego?

Uno de ellos, creará un branch a partir del master y se situará sobre él, previamente nos traemos todo lo que haya en el master

```
$ git pull origin master
```

> Si sólo hacemos git pull, nos bajamos todo lo que hay en el repositorio, no sólo el master, e igual no interesa

```
$ git checkout -b feature/evaluation_gen master
```

Es conveniente nombrarlo feature/ y algo representativo de lo que la funcionalidad incluye.

Como va a trabajar en equipo con Diego, subirá este branch al repositorio remoto:

```
$ git push origin feature/evaluation_gen
```

> Ahora el branch ya está disponible en el remote y Diego puede acceder a él

Diego se crea un branch a partir del que ha subido Nacho

```
$ git checkout -b feature/evaluation_gen origin/feature/evaluation_gen
```

Ahora, los dos hacen su desarrollo, sus pruebas, sus commits. La diferencia es que ahora no trabajamos solos, por lo que es necesario bajarnos el desarrollo del remote justo antes de hacer nuestros commits. Sigue siendo muy recomendable ir viendo el estado de nuestro desarrollo bajo Git, usar, siempre que podáis,

```
$ git status
$ git pull origin feature/evaluation_gen
$ git commit -a
```

> La opción -a nos evita tener que añadir, individualmente al estado "staged", todos los ficheros que toquemos

Como vimos, la potencia de Git está en su facilidad para gestionar ramas. Nadie os va a mirar raro si os creáis una rama local, a partir de feature/evaluation_gen, para hacer vuestros cambios. Si optáis por esto, el escenario sería igual al comentado en el caso de uso "Juan palomo", cambiando el master por feature/evaluation_gen.

Una vez que el dúo dinámico Nacho y Diego están seguros de que funciona todo, se aseguran de que está todo subido y que ambos tienen en su branch local todo el desarrollo realizado. Lo siguiente que hacemos es realizar el merge con el master. Esto lo haría uno de ellos, por ejemplo, Diego.

```
$ git pull origin feature/evaluation_gen
```

> Se asegura de tenerlo todo.

```
$ git checkout master
$ git pull origin master
```

> Si sólo hacemos git pull, nos bajamos todo lo que hay en el repositorio, no sólo el master, e igual no interesa

```
$ git merge --no-ff feature/evaluation_gen
```

⚠ OJO

Siempre opción **--no-ff**, esto obliga a GIT a crear un commit para el merge y nos asegura que se respeta la historia de la rama feature.

Y, por último, subimos los cambios al repositorio con un push. No vamos a borrar la rama feature hasta que nos vcaliden la funcionalidad, la gestión en desarrollos en equipo, habéis visto que es un pelín más compleja y borramos la rama feature

```
$ git push origin master
```

⚠ OJO

Si sólo hacemos **git push**, subimos todo lo que tenemos en local al repositorio, incluyendo la rama que hemos usado para desarrollar, y que bien podríamos llamado feature/novaleparanada. Ahora os explicáis algunas cosas extrañas, ¿no?

Y nada más, funcionalidad implementada.

Ahora, sobre el master, podemos generar el WAR para tango y decirle a los usuarios que nos validen la funcionalidad. Esto nos abre dos escenarios nuevos:

- Nacho y Diego son un equipo completo, equipo Comansi y las pruebas se desarrollan sin incidencias. Caso de uso cerrado
- Nos hemos olvidado de apuntar al sitio correcto, error en las pruebas. Se retoma el desarrollo en la rama y se itera este proceso hasta que todo esté OK.

Una vez que está todo OK, borramos el branch remoto y luego el local:

```
$ git push origin :feature/evaluation_gen
$ git branch -d feature/evaluation_gen
```

Y, hasta aquí, las situaciones normales en el desarrollo semanal. Todavía nos pueden pasar más cosas, las veremos incluidas en los siguientes puntos.

Subida semanal. Última llamada a los pasajeros del vuelo PRODUCCION_X_Y_Z

Actualmente, los Martes es el día en el que se realiza la subida semanal. Ese día, a primera hora (8:45), se acaba el tiempo para realizar subidas al master. ¿Significa esto que nadie puede consolidar ningún desarrollo en el master, ni siquiera los que no se incluyen en la subida semanal?. No, el tiempo de cierre del master no supera el minuto. Veamos cómo funciona esto.

El responsable de hacer la subida semanal es cualquiera de los responsables funcionales. Supongamos que se va a subir la versión 3.0.7 de Producción y lo va a hacer Carlos, ¿que acciones se realizarían? Entran en juego las ramas release.

Carlos pone un mensaje en Teambox indicando que a las 8:45 se acaba el plazo para subir desarrollos al master que deben entrar en la subida semanal. Ojo, siempre debemos cumplir la premisa de que no se puede subir al master nada que no queremos que vaya en la subida.

A las 8:45, Carlos genera la rama release/3.0.7 a partir del master, previamente nos traemos todo lo que haya en el master

```
$ git pull origin master
```

> Si sólo hacemos git pull, nos bajamos todo lo que hay en el repositorio, no sólo el master, e igual no interesa

```
$ git checkout -b release/3.0.7 master
```

> A partir de este momento, el master queda de nuevo abierto para consolidar desarrollos que vayan en la siguiente subida.

Se nos pueden plantear, en este momento, dos posibles escenarios

Subida semanal. Fina, fina, como la seda

Todo va bien, la mañana trascurre sin incidencias en la versión que hay en UAT en Tango 7001, por lo que, a un poco antes de las 15:30, Carlos procede a realizar las tareas para la subida.

Si es necesario cambiar alguna librería para su versión definitva de explotación (por ejemplo, informa-xmlbeans) o algún fichero de configuración, se hará el cambio en release/3.0.7 y los commits correspondientes:

```
$ git commit -a
```

La opción -a nos evita tener que añadir, individualmente al estado "staged", todos los ficheros que toquemos

Lo más habitual es que no tengamos que tocar nada sobre el branch release/3.0.7.

Lo siguiente que hará Carlos, es consolidar los cambios sobre la rama stable que, como vimos, es a partir de la cual se generan las versiones para Producción. Los únicos commits en esta rama, corresponden con código desde el que generamos la versión. Entonces:

```
$ git checkout stable
$ git pull origin stable
```

> Si sólo hacemos git pull, nos bajamos todo lo que hay en el repositorio, no sólo el stable, e igual no interesa

```
$ git merge --no-ff release/3.0.7
```

⚠ OJO

Siempre opción **--no-ff**, esto obliga a GIT a crear un commit para el merge y nos asegura que se respeta la historia de la rama feature.

Con esto tenemos nuestro stable local preparado. Carlos generaría la versión para real a partir de él. En ese momento, se sube stable al remote y, también, se consolida release/3.0.7 contra master, para llevarnos los posibles cambios que hayamos introducido.

```
$ git push origin stable
$ git checkout master
$ git pull origin master
```

> Si sólo hacemos git pull, nos bajamos todo lo que hay en el repositorio, no sólo el master, e igual no interesa

```
$ git merge --no-ff release/3.0.7
```

> ⚠ OJO
>
> Siempre opción **--no-ff**, esto obliga a GIT a crear un commit para el merge y nos asegura que se respeta la historia de la rama feature.

Por último, generamos el tag de la subida y borramos la rama release:

```
$ git checkout stable
$ git tag -a PRODUCCION_3_0_7 -m "Versión semanal 3.0.7"
$ git push origin PRODUCCION_3_0_7
$ git branch -d release/3.0.7
```

Subida semanal. ¡Dios, se habían olvidado de mirar en un sitio!

Y han encontrado una cuchara (Monty Python, gracias)

Tena ha entrado a darse una vuelta por Tango 7001 y ha descubierto un fallo de última hora. ¡Qué desastre! ¿Paren las máquinas? No, que no cunda el pánico, no es necesario. Realizaremos el desarrollo necesario en la rama release. No podemos hacerlo sobre master puesto que puede contener cosas que van en la siguiente subida. La operativa es similar a la enunciada en el punto anterior, con sus commits a release y demás. Hay una pequeña diferencia, si el bug debe arreglarlo una persona que no es Carlos, por ejemplo, Iván, aquel deberá compartir el branch en el repositorio remoto. Recordad:

```
$ git push origin release/3.0.7
```

Iván deberá crearse un branch a partir de él y hacer allí sus commits.

```
$ git checkout -b release/3.0.7 origin/release/3.0.7
```

Y así, Carlos e Iván harán sus commits sobre sus branchs locales, haciendo pull para bajarse los cambios del otro y usando status de manera prolija. Ver caso de uso Make a Team

```
$ git pull origin release/3.0.7
$ git commit -a
```

> La opción -a nos evita tener que añadir, individualmente al estado "staged", todos los ficheros que toquemos

```
$ git push origin release/3.0.7
```

Ya tenemos la versión para subir validada. Lo siguiente que hará Carlos, es consolidar los cambios sobre la rama stable que, como vimos, es a partir de la cual se generan las versiones para Producción. Los únicos commits en esta rama, corresponden con código desde el que generamos la versión. Entonces:

```
$ git checkout stable
$ git pull origin stable
```

> Si sólo hacemos git pull, nos bajamos todo lo que hay en el repositorio, no sólo el stable, e igual no interesa

```
$ git merge --no-ff release/3.0.7
```

> ⚠ OJO
>
> Siempre opción **--no-ff**, esto obliga a GIT a crear un commit para el merge y nos asegura que se respeta la historia de la rama feature.

Con esto tenemos nuestro stable local preparado. Carlos generaría la versión para real a partir de él. En ese momento, se sube stable al remote y, también, se consolida release/3.0.7 contra master, para llevarnos los posibles cambios que hayamos introducido.

```
$ git push origin stable
$ git checkout master
$ git pull origin master
```

> Si sólo hacemos git pull, nos bajamos todo lo que hay en el repositorio, no sólo el master, e igual no interesa

```
$ git merge --no-ff release/3.0.7
```

> ⚠ OJO
>
> Siempre opción **--no-ff**, esto obliga a GIT a crear un commit para el merge y nos asegura que se respeta la historia de la rama feature.

Por último, generamos el tag de la subida y borramos la rama release, de local y de remoto:

```
$ git checkout stable
$ git tag -a PRODUCCION_3_0_7 -m "Versión semanal 3.0.7"
$ git push origin PRODUCCION_3_0_7
$ git branch -d release/3.0.7
$ git push origin :release/3.0.7
```

Con estos casos de uso terminamos las situaciones que pueden producirse para una subida semanal. Probablemente, exista alguna más o, de repente, nos surja alguna necesidad no contemplada. En el momento en el que surja, veremos como lidiar con ella. Sigamos, por lo tanto, con otras situaciones ajenas al proceso de subida semanal.

Un mundo feliz... ¿estáis seguros?

Tenemos una versión en Producción que funciona perfectamente pero, un buen día, alguien detecta un funcionamiento erróneo que se debe corregir lo antes posible. Lo normal es que sean errores que contiene una reciente subida semanal (las famosas subidas de emergencia X.Y.Z.N), pero también pueden, de vez en cuando, aparecer viejos fantasmas del pasado, y salir a la luz bugs que llevan mucho tiempo durmientes.

¿Qué es lo que tenemos que hacer en este caso? Tenemos que generar una versión as soon as posible, pero no podemos hacerlo desde el master pues, con toda probabilidad, tendremos desarrollos en estado de validación. La respuesta es simple, recordad que en la rama stable, todos los commits son versiones del proyecto que están subidas a producción, por lo tanto, utilizaremos stable para crear una rama hotfix, en la que desarrollaremos la solución al bug.

Como entra stable en juego, la persona que creará la rama será uno de los responsables funcionales (encargados de los despliegues). Si es el propio creador de la rama el que soluciona el bug, no hará falta que la suba al remote. Vamos a estudiar el caso, más común, en el que es

otra persona la que soluciona el bug, por ejemplo, Ismael es el encargado de solucionar el bug. Por lo tanto:

Felicidad se encarga de crear la rama hotfix a partir de stable, previamente nos traemos todo lo que haya en el stable

```
$ git pull origin stable
```

Si sólo hacemos git pull, nos bajamos todo lo que hay en el repositorio, no sólo el stable, e igual no interesa

```
$ git checkout -b hotfix/bug_en_denos_extincion stable
```

A continuación, sube esta rama al repositorio para que Ismael acceda a ella:

```
$ git push origin hotfix/bug_en_denos_extincion
```

Ismael deberá crearse un branch a partir de él y hacer allí sus commits.

```
$ git checkout -b hotfix/bug_en_denos_extincion
origin/hotfix/bug_en_denos_extincion
$ git commit -a
```

> La opción -a nos evita tener que añadir, individualmente al estado "staged", todos los ficheros que toquemos

Ismael generará versión para probar en tango 7001 y, una vez que esté todo validado, Ismael subirá la rama

```
$ git push origin hotfix/bug_en_denos_extincion
```

Felicidad deberá bajarse la rama hotfix, y realizar el merge sobre el stable, para generar la versión a subir a producción. Después, deberá consolidar, también, en el master y crear el correspondiente tag. Por último, Felicidad borrará la rama remota y local e Ismael la local.

```
$ git checkout stable
$ git pull origin stable
```

> Si sólo hacemos git pull, nos bajamos todo lo que hay en el repositorio, no sólo el stable, e igual no interesa

```
$ git merge --no-ff hotfix/bug_en_denos_extincion
```

> ⚠ OJO
>
> Siempre opción **--no-ff**, esto obliga a GIT a crear un commit para el merge y nos asegura que se respeta la historia de la rama feature.

```
$ git push origin stable
$ git checkout master
$ git pull origin master
```

> Si sólo hacemos git pull, nos bajamos todo lo que hay en el repositorio, no sólo el master, e igual no interesa

```
$ git merge --no-ff hotfix/bug_en_denos_extincion
```

> ⚠ OJO
>
> Siempre opción **--no-ff**, esto obliga a GIT a crear un commit para el merge y nos asegura que se respeta la historia de la rama feature.

$ git checkout stable

```
$ git tag -a PRODUCCION_3_0_7_1 -m "3.0.7.1 Bugfix en denos en el proceso de extinción"
$ git push origin PRODUCCION_3_0_7_1
$ git branch -d hotfix/bug_en_denos_extincion
$ git push origin :hotfix/big_en_denos_extincion
```

Puede producirse una pequeña variante de este proceso, y es que el bug aparezca cuando estamos en fase de lanzamiento del branch semanal. Entonces, pueden darse dos situaciones:

- Si el arreglo puede esperar e ir incluido en la subida semanal, se incluirá el desarrollo en la rama release.
- Si es un arreglo muy urgente y no puede esperar, generaríamos rama hotfix a partir del stable, subiríamos versión consolidando hotfix en stable y nos llevaríamos el cambio de hotfix a release y a master.

El mundo del desarrollo fuera de las subidas semanales

No todos los desarrollos pueden ajustarse al marco temporal de las subidas semanales. Existen proyectos cuyo desarrollo se extiende durante semanas, meses o, incluso, años. Para este tipo de proyectos, los casos de uso explicados no cuadran. ¿Cómo tenemos que trabajar para estos proyectos?

Supongamos que Juanqui, Daniel, Ángeles, Alberto, Araceli y Gerardo van a trabajar en el proyecto "Eliminación de la BDO". Es un proyecto que se estima que dure 5 meses y, por lo tanto, durante ese tiempo, los desarrollos no pueden estar en el master pero sí deben estar compartidos en una rama.

El enfoque que debemos darle a estos desarrollos es el siguiente. Se debe crear una rama y, para la gente que está trabajando en este proyecto, esta rama tendrá la misma categoría y consideración que la rama master para los desarrollos semanales. Es decir:

- No se desarrolla directamente sobre esta rama.
- Los componentes del proyecto se crean ramas de desarrollo a partir de ella. Tan autocontenidas como sean posibles. Funcionalidades cortas, ramas cortas.
- Cuando el desarrollador está conforme con su código, se realiza el merge sobre la rama del proyecto

Así, Juanqui, como responsable del proyecto, creará la rama para el mismo. Lo haremos a partir del stable, rama con las versiones en producción de nuestro repositorio. Como el desarrollo es compartido, subirá la rama al repositorio.

```
$ git pull origin stable
```

Si sólo hacemos git pull, nos bajamos todo lo que hay en el repositorio, no sólo el stable, e igual no interesa

```
$ git checkout -b feature/eliminar_bdo stable
$ git push origin feature/eliminar_bdo
```

Todos los desarrolladores se crean una rama local apuntando a esta rama remota:

```
$ git checkout -b feature/eliminar_bdo origin/feature/eliminar_bdo
```

Esta rama va a tener, para todos, y como hemos comentado, la misma consideración que el master. Por lo tanto, para desarrollar, cualquiera se creará una rama a partir de ella y hará allí sus desarrollo, sus commits y demás. Las ramas de trabajo local deben ser de duración mínima, un día de trabajo, dos días a lo sumo:

```
$ git checkout -b mi_desarrollo feature/eliminar_bdo
$ git commit -a
```

> La opción -a nos evita tener que añadir, individualmente al estado "staged", todos los ficheros que toquemos

Cuando tenemos el minidesarrollo consolidado en nuestra rama temporal, lo consolidamos en la rama del proyecto:

```
$ git pull origin feature/eliminar_bdo
```

> Si sólo hacemos git pull, nos bajamos todo lo que hay en el repositorio, no sólo la rama, e igual no interesa

```
$ git merge --no-ff mi_desarrollo
```

⚠ OJO

Siempre opción **--no-ff**, esto obliga a GIT a crear un commit para el merge y nos asegura que se respeta la historia de la rama local.

Y borramos la rama:

```
$ git branch -d mi_desarrollo
```

Con el siguiente desarrollo, iteramos de nuevo todo el proceso, y, así, hasta que finalicemos todo el desarrollo necesario

Desde el punto de vista de los desarrolladores, esto es todo lo que hay que tener en cuenta. El responsable del proyecto tiene dos tareas más que atacar:

- Realizar consolidaciones regulares desde el stable a la rama del proyecto. Esto es opcional, se puede optar por no consolidar nada en la rama del proyecto y, cuando haya que subirlo, hacer el merge de todo el desarrollo con el master, pero igual los problemas son grandes porque los códigos de master y proyecto serán muy diferentes, La experiencia nos dirá lo que es mejor auqneu yo recomiendo consolidaciones regulares. Es más, recomiendo consolidar stable con la rama del proeycto cuando se genere una versión para producción.
- Al finalizar el desarrollo, realizar el merge de la rama del proyecto con el master, para que se incluya, todo, en la siguiente subida a Producción.

Vamos a ver estas cómo realizaría Juanqui estas dos acciones

Proyectos largos. Consolidaciones "Poco a poco, suavecito"

Optamos por consolidar en la rama del proyecto el stable cada vez que haya subida a Producción. Juanqui debe bajarse stable, bajarse todo lo existente en la rama y realizar el merge de la rama a stable:

```
$ git checkout stable
$ git pull origin stable
```

> Si sólo hacemos git pull, nos bajamos todo lo que hay en el repositorio, no sólo el stable, e igual no interesa

```
$ git checkout feature/eliminar_bdo
$ git pull origin feature/eliminar_bdo
$ git merge --no-ff stable
```

⚠ OJO

Siempre opción **--no-ff**, esto obliga a GIT a crear un commit para el merge y nos asegura que se respeta la historia de la rama release.

```
$ git push origin feature/eliminar_bdo
```

Tras hacer esto, todos los desarrolladores se actualizarán su rama

```
$ git checkout feature/eliminar_bdo
$ git pull origin feature/eliminar_bdo
```

Si sólo hacemos git pull, nos bajamos todo lo que hay en el repositorio, no sólo el stable, e igual no interesa

Y seguiremos trabajando como hasta el momento. OJO. No es necesario hacer el merge de la rama de desarrollo feature/eliminar_bdo con la rama temporal mi_desarrollo. Los merges siempre hacia arriba, hacia las ramas de mayor nivel de desarrollo.

Proyectos largos. The end of the long and winding road

El proyecto está finalizado, las UAT del mismo se han acabado y sólo nos queda consolidarlo en el master para incluir el desarrollo en el código de la futura subida semanal o especial. Es responsabilidad de Juanqui realizar esta acción, y su forma de proceder será más de lo mismo de siempre:

```
$ git checkout feature/eliminar_bdo

$ git pull origin feature/eliminar_bdo

$ git checkout master
$ git pull origin master
```

Si sólo hacemos git pull, nos bajamos todo lo que hay en el repositorio, no sólo el stable, e igual no interesa

```
$ git merge --no-ff feature/eliminar_bdo
```

⚠ OJO

Siempre opción **--no-ff**, esto obliga a GIT a crear un commit para el merge y nos asegura que se respeta la historia de la rama release.

```
$ git push origin master
```

Posteriormente a que las pruebas finales se hayan validado, se consolidará en stable mediante una rama de lanzamiento como hemos visto antes. Y, una vez que el proyecto finalice, borraremos las ramas remotas y locales de todos los participantes.

Algunos enlaces útiles en los que me he basado para este tutorial:

http://sysvar.net/es/entendiendo-git-flow/

http://aprendegit.com/que-es-git-flow/

http://jesuslc.com/2012/11/24/una-buena-manera-de-afrontar-la-ramificacion-branching-en-git/

Git Flow

Existe una herramienta para facilitarnos la gestión de las ramas con la estrategia de branching que vamos a aplicar. Para explicar **git-flow** es mejor preguntarnos qué no es **git-flow**, que preguntarnos qué es.

git-flow no es un sistema de control de versiones.

git-flow no es algo distinto a GIT.

git-flow es una capa que se sitúa por encima de Git y que nos proporciona comandos que agrupan varias de las acciones que se deben hacer, siempre, para gestionar las ramas.

Podéis ver, fácilmente, cual es la potencia de **git-flow**, simplificar la manera de hacer las cosas y estandarizar los nombres de las ramas para que no dependan de la creatividad o el tino al darle a las teclas de cada uno.

Vamos a enfocar este apartado de una manera un poco diferente a lo que es habitual. En lugar de empezar comentando cómo hacer la instalación y, después, explicar la herramienta, vamos a empezar enumerando las características de git-flow y, una vez que os haya convencido de su utilidad, explicamos como instalarlo.

Antes de nada, y cómo veréis que he modificado en la teoría de la estrategia de branching, vamos a tomar, como convención, que las ramas se nombran como tipo_rama/nombre_funcionalidad. Por ejemplo, hotfix/bug_aig_nocensadas, feature/lo_que_sea, release/3.0.7, etc...

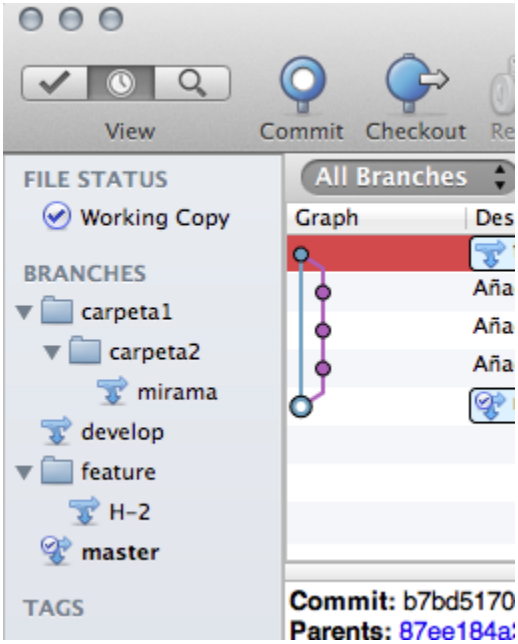
En este enlace explica, muy claramente, las razones de hacerlo así. De hecho, al empezar a usar git-flow en un proyecto, te pide el nombre con el que distinguir las ramas, y, por defecto, la opción que da es esa (release/ hotfix/ y feature/)

http://aprendegit.com/organizando-las-ramas-en-carpetas/

Básicamente, es por un tema de organización. Poniendo los nombres de las ramas como si fueran carpetas, se nos visualizan de forma más clara en los GUI o, incluso, en la consola. Podemos usar la potendia de las ramas para indicar, en el nombre, sobre qué módulo versa el desarrollo que contienen. Por ejemplo:

feature/borme/nuevo_acto_cierre_hoja

En el GUI que introduciremos más adelante (Source Tree), una rama carpeta1/carpeta2/mirama se vería así:



Y así en la consola:

```
$ git branch -av
   carpeta1/carpeta2/mirama 4604003 Introduciendo el texto definitivo
```

¡A jugaaaaaaaaar!

En este apartado vamos a ver cómo trabajar con git-flow. Utilizaremos un caso de uso, tal y cómo hicimos con los escenarios de nuestro día a día.

Inicializando el repositorio

Lo primero que tenemos que hacer para usar git-flow, es preparar el repositorio del proyecto para ello. Lo hacemos, por consola, con el comando:

Inicializar git-flow

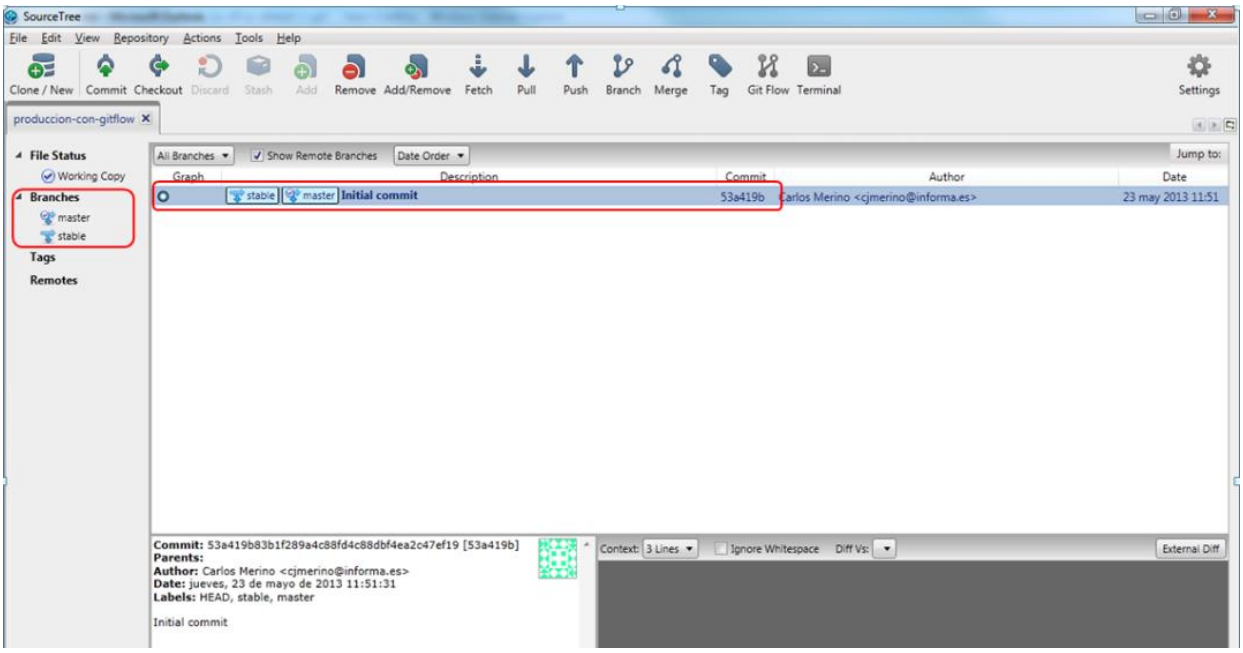
```
$ cd produccion
$ git-flow init

No branches exist yet. Base branches must be created now.
Branch name for production releases: [master]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? [] v
```

Cómo veis, al ejecutar el init de git-flow nos sale un diálogo interactivo en el que nombrar los tipos de rama:

- Primera pregunta (production releases), ponemos **stable**.
- Segunda pregunta (next release), ponemos **master**.
- Dejamos los valores por defecto para el resto las preguntas.

Posteriormente, de todas formas, se podrían cambiar estos valores.

En Source Tree, veríamos esto:



En consola:

Branches iniciales

```
$ git branch -av
* master 53a419b Initial commit
  stable 53a419b Initial commit
```

Gestionando ramas con git-flow

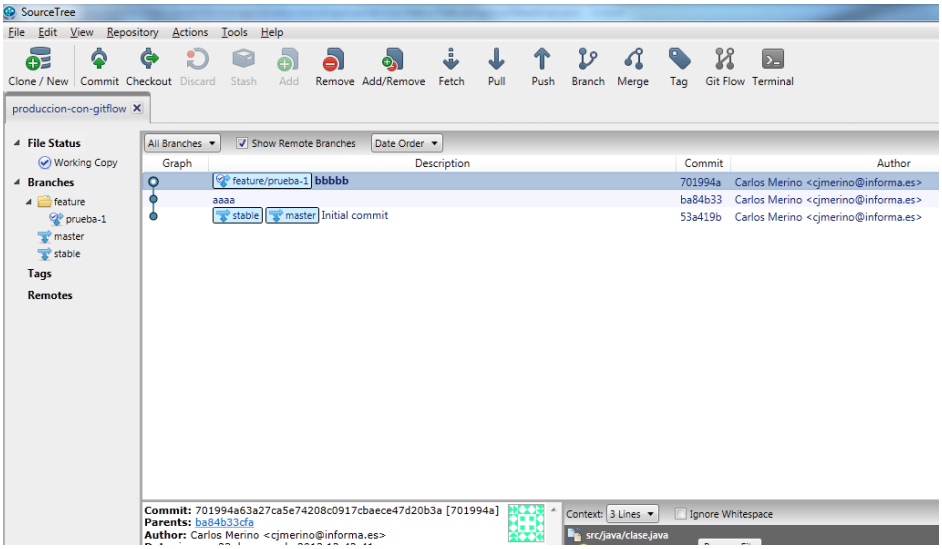
Desde consola

Si recordáis, cuando queremos crear una rama feature lo que hacemos, por consola es

Branches iniciales

```
cjmerino@INF_CMERINO /d/GIT/produccion-con-gitflow (master)
$ git checkout -b feature/prueba-1 master
Switched to a new branch 'feature/prueba-1'
cjmerino@INF_CMERINO /d/GIT/produccion-con-gitflow (feature/prueba-1)
$ git branch -av
* feature/prueba-1 53a419b Initial commit
  master           53a419b Initial commit
  stable           53a419b Initial commit
```

Luego, hacemos una serie de commits para el desarrollo. Esto se verá así en el GUI:

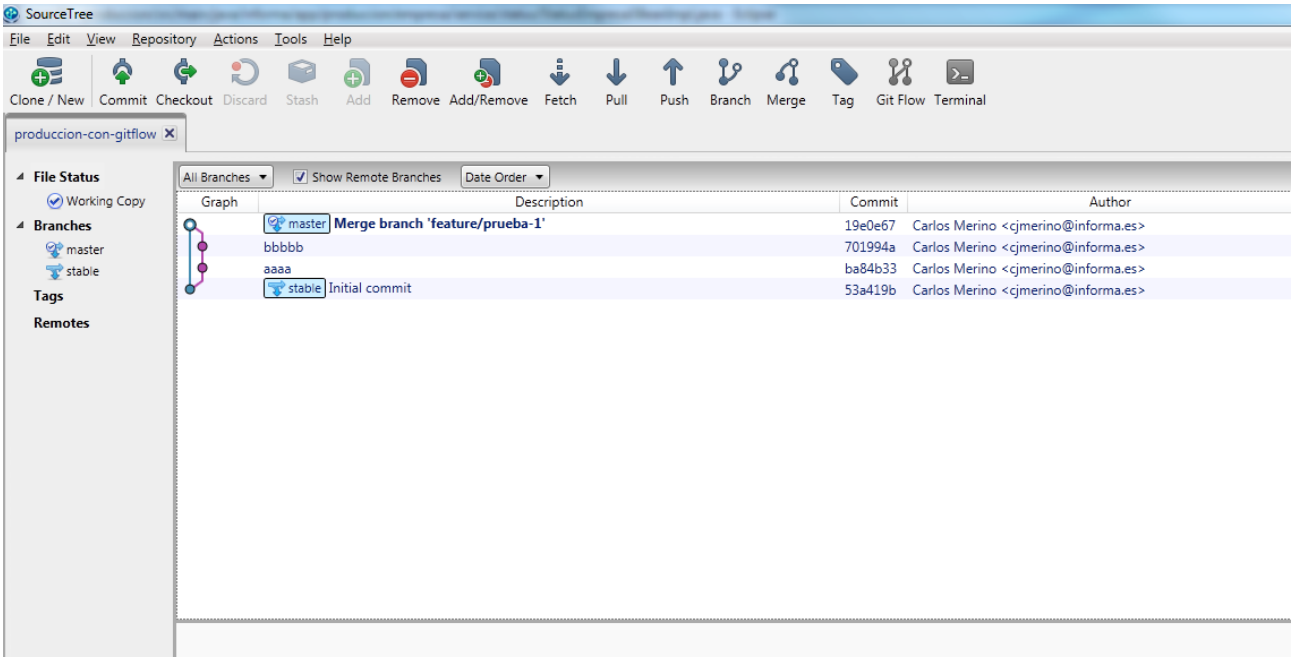


Y acabamos el desarrollo. Ahora hacemos el merge con la rama master:

Merge con master

```
$ git checkout master
$ git merge --no-ff feature/prueba-1
```

Y ahora, tenemos:



Gestión con git-flow

Hacemos ahora el mismo proceso con git-flow:

git-flow. Comienzo de branch

```
$ git-flow feature start prueba-2
Switched to a new branch 'feature/prueba-2'


Summary of actions:
- A new branch 'feature/prueba-2' was created, based on 'master'
- You are now on branch 'feature/prueba-2'


Now, start committing on your feature. When done, use:
     git flow feature finish prueba-2
```

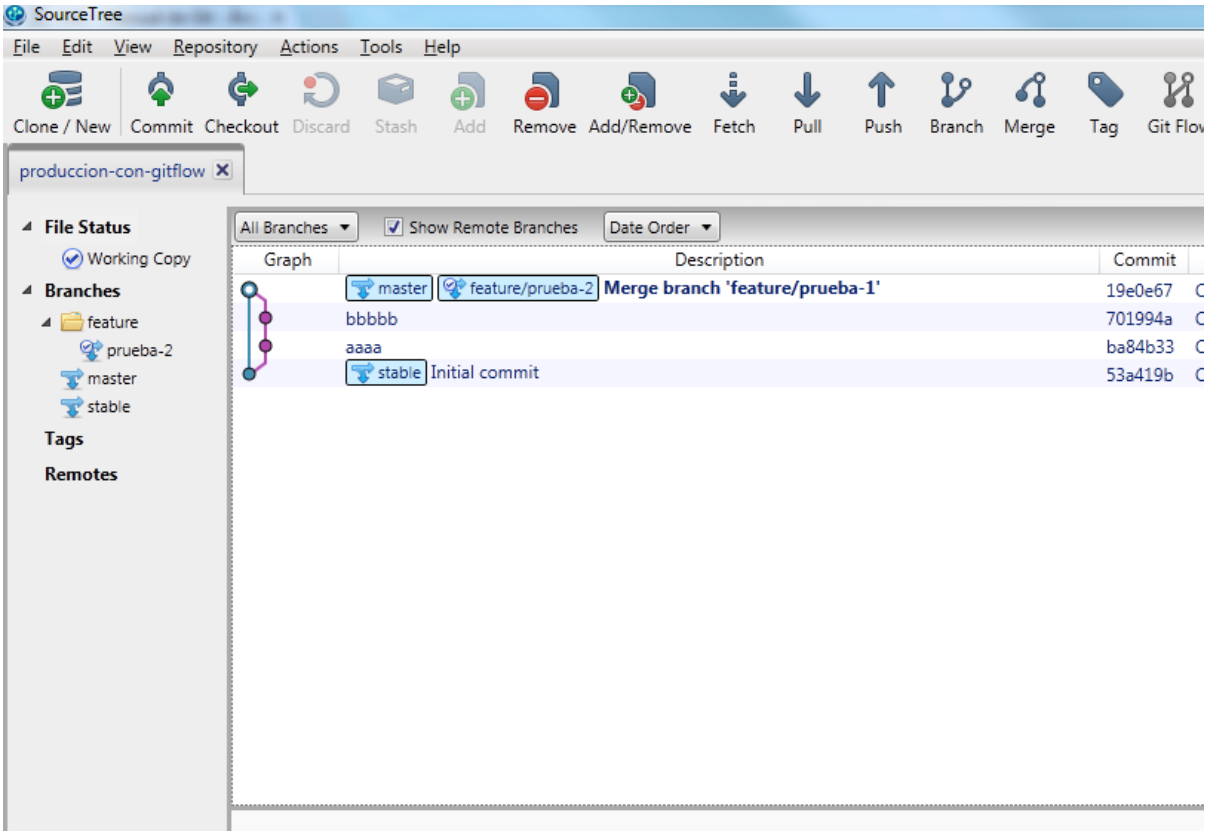
Como veis, le indicamos:

- El tipo de rama a crear (feature)
- La operación que queremos hacer (start - creación de rama)

- Nombre de rama. Sin el prefijo feature/, eso lo añade él

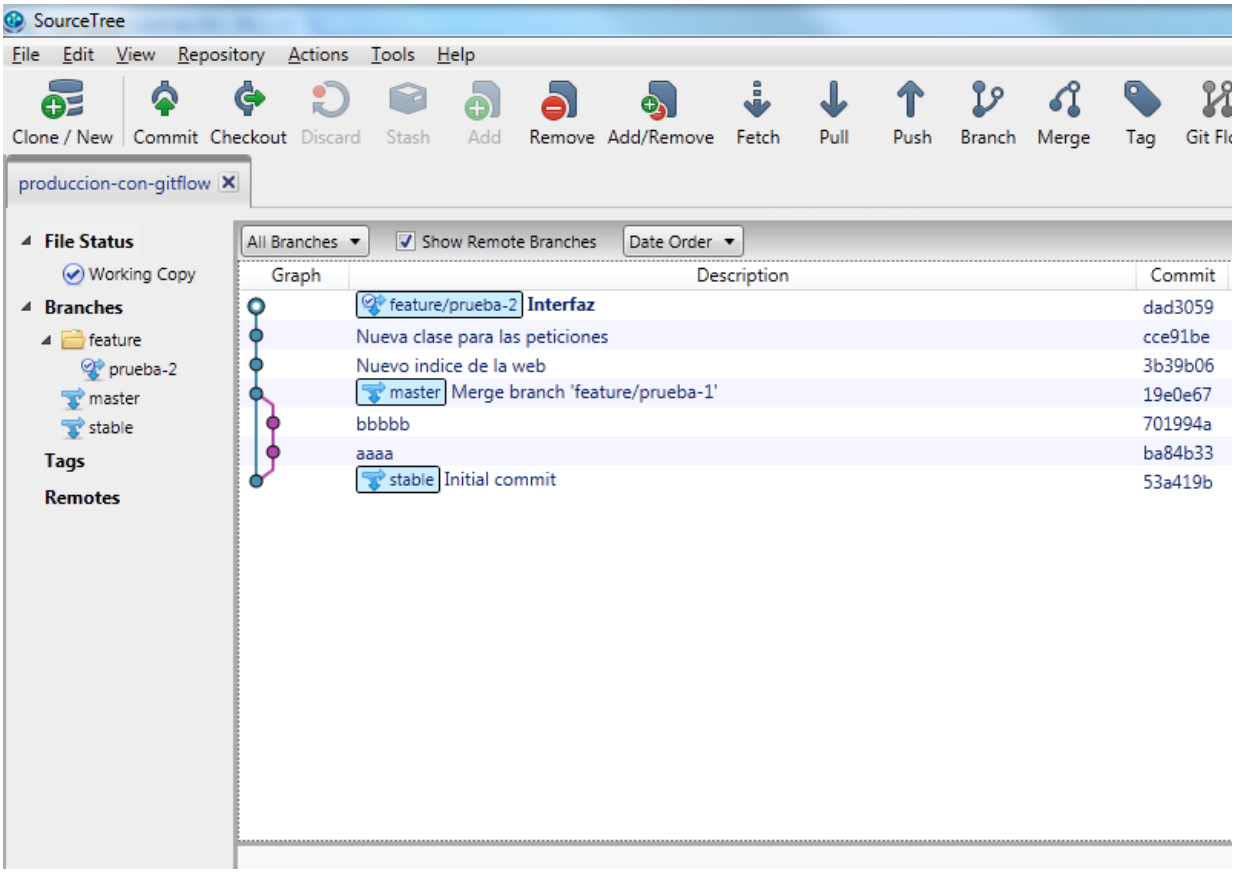
Y él nos dice, excatamente, qué acciones ha hecho.

En Source Tree:



Ahora, igual que sin git-flow, trabajaremos en el proyecto, haciendo nuestros commits. Esto no varía.

En Source Tree:



Si consultamos las ramas desde la línea de comandos:

git-flow. Estado ramas

```
$ git branch -av
* feature/prueba-2 dad3059 Interfaz
  master           19e0e67 Merge branch 'feature/prueba-1'
  stable           53a419b Initial commit
```

Si queremos ver las ramas feature gestionadas por git-flow:

git-flow. Estado ramas

```
$ git flow feature
* prueba-2
```

De igual manera, haríamos para las ramas hotfix y release

Una vez que nuestro código esté testado y validado, cerramos el desarrollo con git-flow. Para ello:

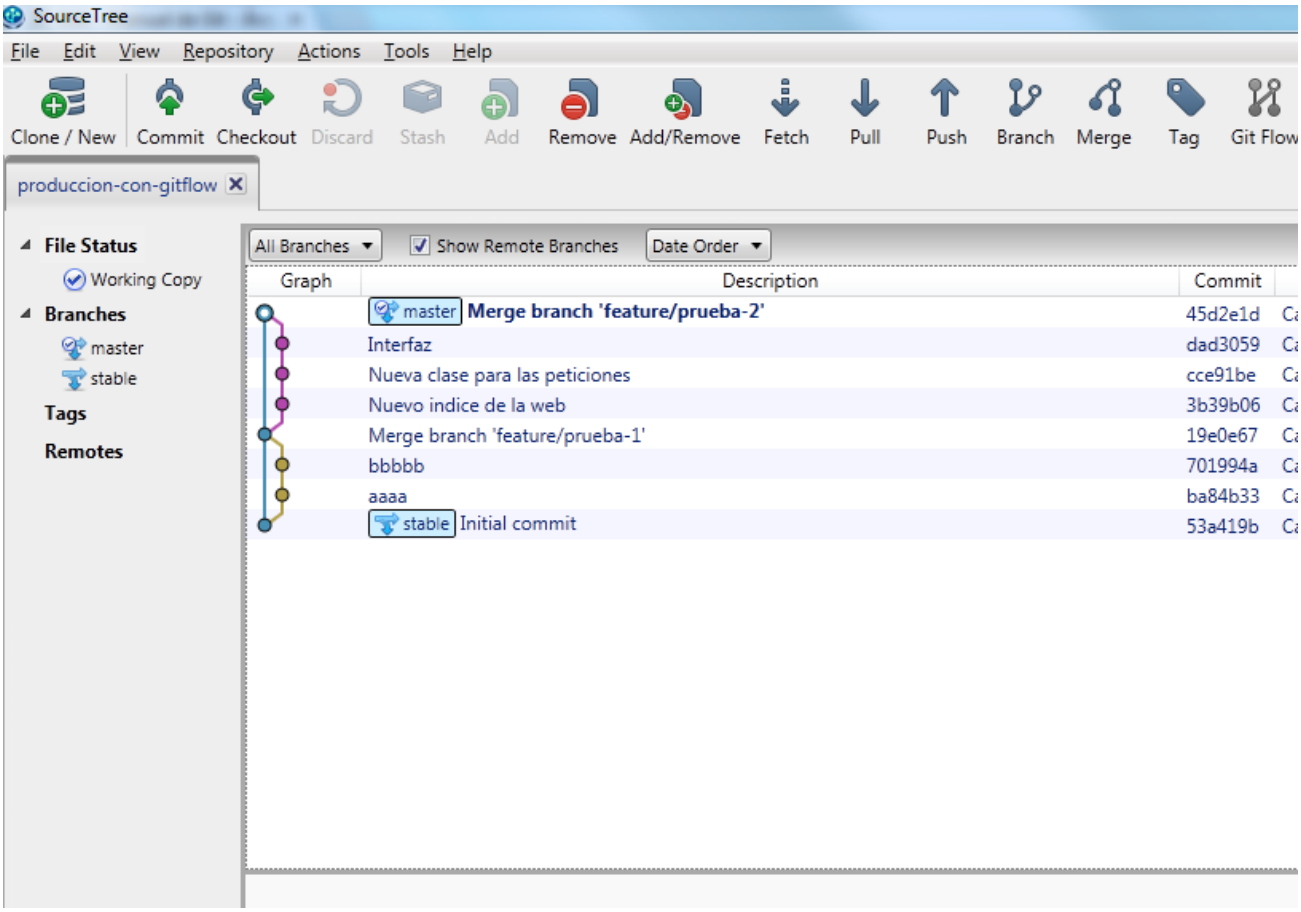
git-flow. Cerrando desarrollo

```
cjmerino@INF_CMERINO /d/GIT/produccion-con-gitflow (feature/prueba-2)
$ git flow feature finish prueba-2


Switched to branch 'master'


Merge made by the 'recursive' strategy.
 index-prueba2.html             | 1 +
 src/java/clase-prueba2.java    | 1 +
 src/java/interfaz-prueba2.java | 1 +
 3 files changed, 3 insertions(+)
 create mode 100644 index-prueba2.html
 create mode 100644 src/java/clase-prueba2.java
 create mode 100644 src/java/interfaz-prueba2.java


Deleted branch feature/prueba-2 (was dad3059).


Summary of actions:
- The feature branch 'feature/prueba-2' was merged into 'master'
- Feature branch 'feature/prueba-2' has been removed
- You are now on branch 'master'
```

Como veis, con un simple comando hemos hecho varias acciones que antes nos llevaban varios comandos. Es importante fijarse que estábamos posicionados en feature/prueba-2. Hemos usado finish para indicar que es el cierre de la rama. Lo que se ha hecho es:

- Posicionar en master (git checkout master)
- Realizar el merge, con --no-ff (git merge --no-ff feature/prueba-2)
- Borrar la rama de feature (git branch -d feature/prueba-2)

Y en la siguiente captura de Source Tree, vemos la situación final:



Veis, en la captura, como la estrategia de branching seguida queda reflejada, en la historia del repositorio, de forma limpia y clara, indicando, en todo momento, cuando comenzó y finalizó el desarrollo en cada una de las ramas feature.

Este proceso es similar para las ramas de release y hotfix. Consultad los siguientes enlaces:

http://aprendegit.com/git-flow-release-branches/

http://aprendegit.com/git-flow-hotfix-branches/

Recordad, al leer estos enlaces, que, en ellos, nuestra rama stable se llama master y nuestra rama master se llama develop.

En el artículo sobre hotfix, se dan unas nociones sobre un concepto importante en Git, el Stash. Este mecanismo es útil para "guardar" tu trabajo cuando cambias de rama. Iván, that's for you.

Yo soy mucho más moderno que eso que me vendes: GUIs

Como hemos comentado en alguna ocasión, se recomienda trabajar, siempre, desde consola Git Bash, abusando del comando git status. No se recomienda, para nada, usar EGit, el plugin de Git para Eclipse, principalmente porque, bajo mi punto de vista, da la impresión de que no controlas lo que se está haciendo.

De todas formas, si sufres alguna especie de alergia primaveral o tienes alguna contraindicación médica que te impida usar la consola, existen diversos GUIs (Graphical user Interface) para trabajar con Git. Por ejemplo SmartGit o Source Tree.

GIT. Conceptos avanzados

Subir proyecto local a remoto

1. Primero crear el proyecto java. Bien creándolo desde cero, bien clonando uno existente para servirnos de su estructura y cambiando el nombre del proyecto. En el caso de **clonar** debemos tener en cuenta **borrar la carpeta "/.git".**
2. Consola git: Ejecutar la instrucción **git init**
3. Añadir el archivo "**.gitignore**" al proyecto. (Si clonamos ya nos lo traeremos. Si no copiarlo de otro proyecto).
4. Ejecutar:**git status**
5. Ejecutar:**git add** /directorios/. Con esto añadimos recursivamente la estructura de nuestro proyecto al índice. (Se repite con cada directorio).
6. Una vez añadida la estructura hacemos el commit. Ejecutamos:**git commit**
7. Estamos situados en nuestro proyecto. **Subir una carpeta para colocarnos en el raiz de git**. Ej. D:/git
8. Después clonarl nuestro proyecto local y convertirlo en proyecto git: **git clone --bare my_project my_project.git**
9. Seguidamente copiar el proyecto de git a remoto: **scp -r my_project.git apps@guru-app.empresa:/opt/data/git/produccion/my_project.git**
10. Finalmente importamos desde eclipse el proyecto desde local (si nos dice que el proyecto ya existe en local lo borramos ya que lo descargaremos de remoto).

Subir rama a remoto

Partimos del supuesto de crear nuestra rama a partir, generalmente, del master.

La nomenclatura de nuestra rama atenderá a los criterios comentados en puntos anteriores.

Desde la línea de comandos: **git push origin <nombre-rama>**

SOCORRO

¡AYUDA! La he liado parda.

Nos ha pasado a todos y nos seguirá pasando: **Deshacer un *commit* en local y en remoto**.

Si nadie se ha hecho un *pull* todavía bastaría con cambiar el HEAD (el puntero al último *commit*) y forzar un *push* al repositorio remoto:

```
git reset --hard HEAD^ git push -f
```

- `HEAD^`: 1 commit atrás
- `HEAD^^`: 2 commits atrás
- `HEAD~10`: 10 commits atrás

> Que pasa si alguien ya se ha hecho un pull del repositorio? que hago entonces?

La solución es el uso de **revert**:

- **`git revert`** deshace tu último *commit* en local creando un nuevo commit que revierte lo que hizo el *commit* "erróneo"
- push el cambio generado por `git revert`.

Cambiar editor por defecto

- Si no nos gusta el editor de texto de la consola se puede cambiar ejecutando:

```
git config --global core.editor C:\prog\git\npp.bat
```

donde npp.bat es un fichero con la ruta de nuestro editor (en mi caso el Notepad++)

```
#!/bin/sh
"D:/Archivos de Programa/Notepad++/notepad++.exe" -multiInst "$*"
```

También podemos editar directamente el fichero .gitconfig añadiendo esto:

```
[core]
editor = D:/npp.bat
```

Cambiar herramienta para gestionar conflictos

Para cuando tengamos conflictos y queramos realizar `git difftool` y `git mergetool` desde herramientas externas, por ejemplo el WinMerge, el DiffMerge o el KDiff3.

Lo más fácil, KDiff3:

Nos instalamos el programa y después añadimos al *.gitconfig* de la siguiente manera:

```
[diff]
    tool = kdiff3

[merge]
    tool = kdiff3

[mergetool "kdiff3"]
    path = D:/Program Files/KDiff3/kdiff3.exe
    keepBackup = false
```

```
    trustExitCode = false
```

Desde consola ya podremos ejecutar la herramienta de comparación, ya sea archivo a archivo, o directamente comparando todo nuestro proyecto con otra versión, ya sea local o remota. Para esto último el comando sería:

```
$git difftool --dir-diff origin/master
```

Para usar otros programas que no sean KDiff3:

Nos definimos otro archivo del tipo .bat para lanzar la herramienta que queramos:

```
 @echo off
 REM This is winmerge.bat
 "D:\Archivos de Programa\WinMerge\WinMergeU.exe" %2 %5
```

Para los conflictos el Winmerge no es la mejor solución porque no permite las diferencias a 3 bandas (local, padre y remota):

http://manual.winmerge.org/Faq.html

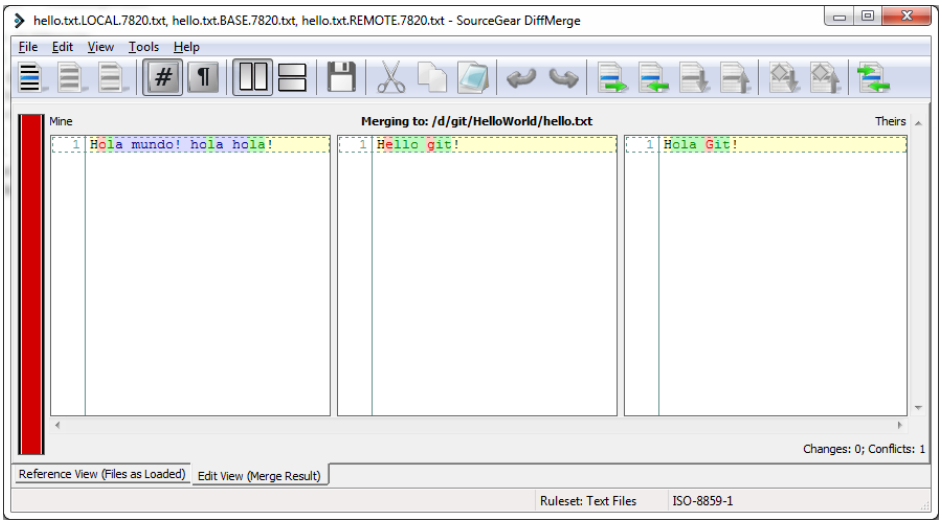
Por lo tanto igual es mejor opción usar el DiffMerge. Creamos un archivo .sh y lo dejamos en Git/cmd

```
 #!/bin/sh
 # Pasamos los parametros al mergetool: local base remote merge_result
 "C:/Program Files/SourceGear/Common/DiffMerge/sgdm.exe" "$1" "$2" "$3" --
result="$4" --title1="Mine" --title2="Merging to: $4" --title3="Theirs"
```

Cambiamos nuestro .gitconfig añadiendo:

```
   [merge]
     tool = diffmerge
   [mergetool "diffmerge"]
     cmd = git-merge-diffmerge.sh "$PWD/$LOCAL" "$PWD/$BASE" "$PWD/$REMOTE"
"$PWD/$MERGED"
```

```
trustExitCode = false
keepBackup = false
```



Git Ignore

Ignores globales

Es importante tener configurado el archivo .gitignore para no compartir las clases compiladas y demás ficheros que no sirven para nada en el repositorio. Como este tipo de archivos son comunes a todos nuestros proyectos, podemos definir un gitignore global que nos sirve para cualquier proyecto. Tendríamos que crear un fichero y luego setear la variable global **core.excludesfile**

```
git config --global core.excludesfile 'D:/git/.gitignore_global'
```

Un fichero de ejemplo para nuestro caso puede ser: .gitignore_global

Ignorando ficheros que ya hemos versionado

Suponed que estamos metiendo en el fichero .gitignore el .classpath o ese directorio src/main/java/es que contiene las clases que genera JAXB. No queremos que nos diga que se ha modificado en cada momento que compilemos, ni que nos dé problemas de conflictos. Pero, ¿qué pasa? por mucho que lo ponemos en el .gitignore como:

```
.classpath
src/main/java/es/
```

No nos hace ni caso, siguen dándonos por el, emmmmmm, siguen dándonos problemas.

Lo más probable es que ya se hayan añadido al índice de GIT y, por eso, no se tiene en cuenta lo del gitignore.

Para hacer que los ignore de verdad, debemos sacarlos del control de GIT. Lo haremos así:

```
$ git rm --cached .classpath
$ git rm -r --cached src/main/java/es/
```

> Ojo al -r, le indica que excluya, recursivamente, todo lo contenido en el directorio.

Veréis que ahora ignora lo que tiene que ignorar. Enlace útiles:

https://help.github.com/articles/ignoring-files

http://stackoverflow.com/questions/343646/ignoring-directories-in-git-repos-on-windows/343734#343734

No Fast Forward como opción por defecto

Podemos "ahorrarnos" el tener que escribir --no --ff cada vez que hagamos un merge, para ello seteamos la variable de configuración **`merge.ff`** a `false`:

```
git config --global merge.ff false
```

(`Sin --global` solo afectaría al proyecto actual)

http://git-scm.com/docs/git-config

Stash. Guardado rápido provisional

En algunas ocasiones, nos hemos encontrado que tenemos, en nuestra rama de trabajo cambios que aun no queremos subir (hacer commit) porque es un desarrollo a medias o, simplemente, porque no nos apetece.

Si tenemos que hacer checkout a otra rama, esos cambios sin consolidar nos darán problemas.

Una forma para solucionar esta situación es usar **stash**, Stash te guarda el estado del proyecto y el índice git en una pila para poder recuperarlo a posteriori.

```
$ git stash
```

- Hacemos un guardado rápido

```
$ git stash list
```

- `Listamos todos los guardados rápidos hechos`

```
$ git stash apply
```

- Recuperamos el último guardado rápido (y lo dejamos en la pila, no se borra)

```
$ git stash drop
```

- Para borrar el último guardado rápido de la pila

```
$ git stash pop
```

- Aplica cambios de un guardado y lo retira inmediatamente de la pila. (apply + drop)

Este enlace es muy ilustrativo:

http://git-scm.com/book/es/Las-herramientas-de-Git-Guardado-r%C3%A1pido-provisional

Lentitud desesperante en git

Una de las caracteristicas de las que presume git es de ser rápido. Pero por experiencia, en repositorios grandes, con mucho historial, y especialmente con el paso del tiempo git empieza a funcionar un poco más lento cada vez. Incluso un git status se hace muy lento.

Si os ocurre esto, la forma de solucionarlo es haciendo un

```
$ git gc --aggressive
```

Git se toma un tiempo para actualizar y optimizar sus indices, siendo el resultado una mejora considerable en los tiempos de respuesta.