



Curso Git Básico 2016

Presentación del curso.

Recordar que si hay dudas preguntas.



- Qué es git?, vamos a ver en qué consiste GIT y qué nos soluciona.
- Instalación, cómo empezar a trabajar con GIT. Los programas en los que está implementado y las herramientas que tenemos para manejarlo.
- Uso, vamos a aprender cómo usar Git de forma sencilla.
- Filosofía, la política según la cual vamos a usar Git. Es lo que da sentido a usar GIT, casi tan importante como usarlo en sí mismo.



1. ¿Qué es?



- Qué es git?, vamos a ver en qué consiste GIT y qué nos soluciona.
- Instalación, cómo empezar a trabajar con GIT. Los programas en los que está implementado y las herramientas que tenemos para manejarlo.
- Uso, vamos a aprender cómo usar Git de forma sencilla.
- Filosofía, la política según la cual vamos a usar Git. Es lo que da sentido a usar GIT, casi tan importante como usarlo en sí mismo.



1. ¿Qué es?
2. Instalación



- Qué es git?, vamos a ver en qué consiste GIT y qué nos soluciona.
- Instalación, cómo empezar a trabajar con GIT. Los programas en los que está implementado y las herramientas que tenemos para manejarlo.
- Uso, vamos a aprender cómo usar Git de forma sencilla.
- Filosofía, la política según la cual vamos a usar Git. Es lo que da sentido a usar GIT, casi tan importante como usarlo en sí mismo.



1. ¿Qué es?
2. Instalación
3. Uso



- Qué es git?, vamos a ver en qué consiste GIT y qué nos soluciona.
- Instalación, cómo empezar a trabajar con GIT. Los programas en los que está implementado y las herramientas que tenemos para manejarlo.
- Uso, vamos a aprender cómo usar Git de forma sencilla.
- Filosofía, la política según la cual vamos a usar Git. Es lo que da sentido a usar GIT, casi tan importante como usarlo en sí mismo.



1. ¿Qué es?
2. Instalación
3. Uso
4. Filosofía



- Qué es git?, vamos a ver en qué consiste GIT y qué nos soluciona.
- Instalación, cómo empezar a trabajar con GIT. Los programas en los que está implementado y las herramientas que tenemos para manejarlo.
- Uso, vamos a aprender cómo usar Git de forma sencilla.
- Filosofía, la política según la cual vamos a usar Git. Es lo que da sentido a usar GIT, casi tan importante como usarlo en sí mismo.

¿Qué (dantres) soluciona?

3

- Primera imagen de las torres petronas, representa la fortaleza que da tener un sistema para el control de versiones, donde los desarrollos pueden realizarse de forma paralela y comunicarse entre sí. Por eso pongo los edificios en paralelo, que representan esta forma de construir el software.
- Segunda imagen, con TRUMP, todo queda documentado, desde usuarios hasta qué se hizo, qué tocó y cuándo fue. De esta forma podremos llevar la trazabilidad del proyecto tanto a la hora de desarrollarlo como a la hora de mantenerlo.
- Tercera imagen, el desastre que puede ser NO tener cuidado ni interés por ordenar tu proyecto. Lo que vamos a evitar con GIT.

¿Qué (dantres) soluciona?



3

- Primera imagen de las torres petronas, representa la fortaleza que da tener un sistema para el control de versiones, donde los desarrollos pueden realizarse de forma paralela y comunicarse entre sí. Por eso pongo los edificios en paralelo, que representan esta forma de construir el software.
- Segunda imagen, con TRUMP, todo queda documentado, desde usuarios hasta qué se hizo, qué tocó y cuándo fue. De esta forma podremos llevar la trazabilidad del proyecto tanto a la hora de desarrollarlo como a la hora de mantenerlo.
- Tercera imagen, el desastre que puede ser NO tener cuidado ni interés por ordenar tu proyecto. Lo que vamos a evitar con GIT.

¿Qué (dantres) soluciona?



3

- Primera imagen de las torres petronas, representa la fortaleza que da tener un sistema para el control de versiones, donde los desarrollos pueden realizarse de forma paralela y comunicarse entre sí. Por eso pongo los edificios en paralelo, que representan esta forma de construir el software.
- Segunda imagen, con TRUMP, todo queda documentado, desde usuarios hasta qué se hizo, qué tocó y cuándo fue. De esta forma podremos llevar la trazabilidad del proyecto tanto a la hora de desarrollarlo como a la hora de mantenerlo.
- Tercera imagen, el desastre que puede ser NO tener cuidado ni interés por ordenar tu proyecto. Lo que vamos a evitar con GIT.

¿Qué (dantres) soluciona?



3

- Primera imagen de las torres petronas, representa la fortaleza que da tener un sistema para el control de versiones, donde los desarrollos pueden realizarse de forma paralela y comunicarse entre sí. Por eso pongo los edificios en paralelo, que representan esta forma de construir el software.
- Segunda imagen, con TRUMP, todo queda documentado, desde usuarios hasta qué se hizo, qué tocó y cuándo fue. De esta forma podremos llevar la trazabilidad del proyecto tanto a la hora de desarrollarlo como a la hora de mantenerlo.
- Tercera imagen, el desastre que puede ser NO tener cuidado ni interés por ordenar tu proyecto. Lo que vamos a evitar con GIT.

Características

4

- Ligero, es extremadamente ligero, apenas intrusivo, es simplemente una carpeta oculta en el raíz del proyecto.

Es ideal para todo tipo de proyectos, desde proyectos pequeños minúsculos, hasta proyectos más grandes.

Los servidores Git tampoco requieren grandes requisitos, tampoco los equipos clientes requieren grandes requisitos.

- Rápido, es extremadamente rápido, hacer un branch o bajarte todo un proyecto apenas tarda lo que ocupe el proyecto. Además todas las acciones son muy rápidas, desde hace un commit hasta crear un branch, se hace en cuestión de segundos.

- Ampliamente soportado, más de la mitad de los programadores del mundo usan Git, además cuenta con una comunidad enorme como es GitHub para predicarlo. Incluso Microsoft ha mostrado su interés metiendo soporte nativo de Git a Visual Studio.

* Siguiente diapositiva representa la realidad actual, es microsoft el que más utiliza GitHub.

Características

- Ligero, no ocupa espacio en disco.

4

- Ligero, es extremadamente ligero, apenas intrusivo, es simplemente una carpeta oculta en el raíz del proyecto.

Es ideal para todo tipo de proyectos, desde proyectos pequeños minúsculos, hasta proyectos más grandes.

Los servidores Git tampoco requieren grandes requisitos, tampoco los equipos clientes requieren grandes requisitos.

- Rápido, es extremadamente rápido, hacer un branch o bajarte todo un proyecto apenas tarda lo que ocupe el proyecto. Además todas las acciones son muy rápidas, desde hace un commit hasta crear un branch, se hace en cuestión de segundos.

- Ampliamente soportado, más de la mitad de los programadores del mundo usan Git, además cuenta con una comunidad enorme como es GitHub para predicarlo. Incluso Microsoft ha mostrado su interés metiendo soporte nativo de Git a Visual Studio.

* Siguiente diapositiva representa la realidad actual, es microsoft el que más utiliza GitHub.

Características

- Ligero, no ocupa espacio en disco.
- Rápido.

4

- Ligero, es extremadamente ligero, apenas intrusivo, es simplemente una carpeta oculta en el raíz del proyecto.

Es ideal para todo tipo de proyectos, desde proyectos pequeños minúsculos, hasta proyectos más grandes.

Los servidores Git tampoco requieren grandes requisitos, tampoco los equipos clientes requieren grandes requisitos.

- Rápido, es extremadamente rápido, hacer un branch o bajarte todo un proyecto apenas tarda lo que ocupe el proyecto. Además todas las acciones son muy rápidas, desde hace un commit hasta crear un branch, se hace en cuestión de segundos.

- Ampliamente soportado, más de la mitad de los programadores del mundo usan Git, además cuenta con una comunidad enorme como es GitHub para predicarlo. Incluso Microsoft ha mostrado su interés metiendo soporte nativo de Git a Visual Studio.

* Siguiente diapositiva representa la realidad actual, es microsoft el que más utiliza GitHub.

Características

- Ligero, no ocupa espacio en disco.
- Rápido.
- Ampliamente soportado.

4

- Ligero, es extremadamente ligero, apenas intrusivo, es simplemente una carpeta oculta en el raíz del proyecto.

Es ideal para todo tipo de proyectos, desde proyectos pequeños minúsculos, hasta proyectos más grandes.

Los servidores Git tampoco requieren grandes requisitos, tampoco los equipos clientes requieren grandes requisitos.

- Rápido, es extremadamente rápido, hacer un branch o bajarte todo un proyecto apenas tarda lo que ocupe el proyecto. Además todas las acciones son muy rápidas, desde hace un commit hasta crear un branch, se hace en cuestión de segundos.

- Ampliamente soportado, más de la mitad de los programadores del mundo usan Git, además cuenta con una comunidad enorme como es GitHub para predicarlo. Incluso Microsoft ha mostrado su interés metiendo soporte nativo de Git a Visual Studio.

* Siguiente diapositiva representa la realidad actual, es microsoft el que más utiliza GitHub.



4

- Ligero, es extremadamente ligero, apenas intrusivo, es simplemente una carpeta oculta en el raíz del proyecto.

Es ideal para todo tipo de proyectos, desde proyectos pequeños minúsculos, hasta proyectos más grandes.

Los servidores Git tampoco requieren grandes requisitos, tampoco los equipos clientes requieren grandes requisitos.

- Rápido, es extremadamente rápido, hacer un branch o bajarte todo un proyecto apenas tarda lo que ocupe el proyecto. Además todas las acciones son muy rápidas, desde hace un commit hasta crear un branch, se hace en cuestión de segundos.

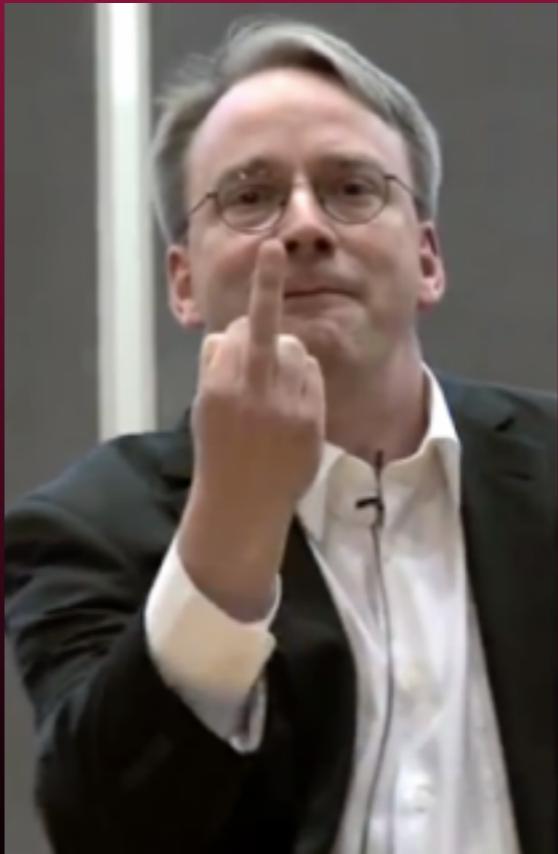
- Ampliamente soportado, más de la mitad de los programadores del mundo usan Git, además cuenta con una comunidad enorme como es GitHub para predicarlo. Incluso Microsoft ha mostrado su interés metiendo soporte nativo de Git a Visual Studio.

* Siguiente diapositiva representa la realidad actual, es microsoft el que más utiliza GitHub.

¿GIT?

Linus Torvald (1969 - 3201)

*“Soy un bastardo
egocéntrico y nombro a mis
proyectos pensando en mí.
Primero ‘Linux’, ahora ‘git’.”*



5

Git está creado por Linus Torvald en 2005, aunque actualmente no lleva personalmente el desarrollo.

Para entender el motivo que llevó a Linus a crear Git es ver su explicación:

— — — leer — — —

GIT en slang inglés se entiende por idiota.

Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información.

Hay que ser idiota para romper el git.

IDIOTA

6

GIT en slang inglés se entiende por idiota.

Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información.

Hay que ser idiota para romper el git.

¿Cómo funciona?



7

Cada vez que haces un cambio y lo confirmas él básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea.

Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información

¿Cómo funciona?



7

Cada vez que haces un cambio y lo confirmas él básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea.

Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información

¿Cómo funciona?



Cada vez que haces un cambio y lo confirmas él básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea.

Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información

¿Cómo funciona?



commit 1

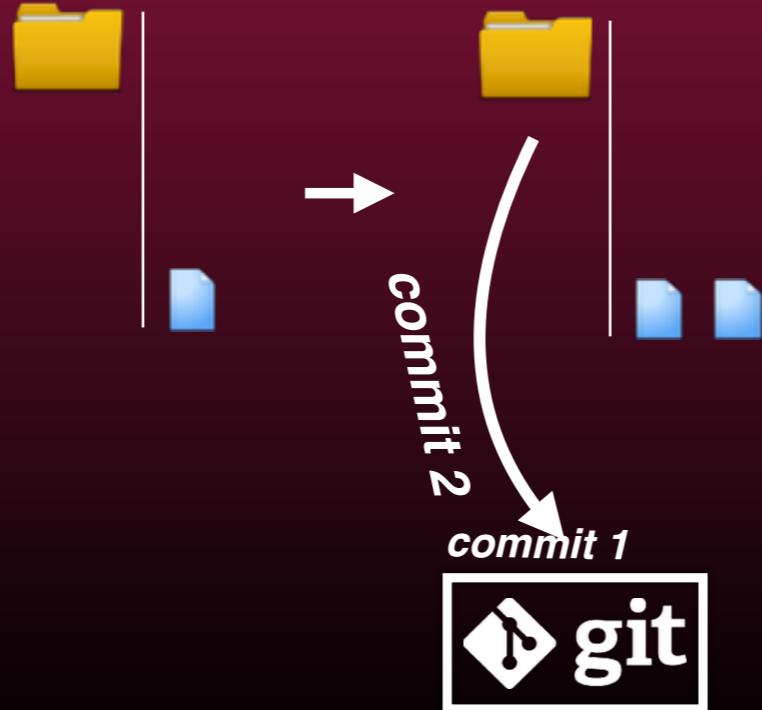


7

Cada vez que haces un cambio y lo confirmas él básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea.

Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información

¿Cómo funciona?

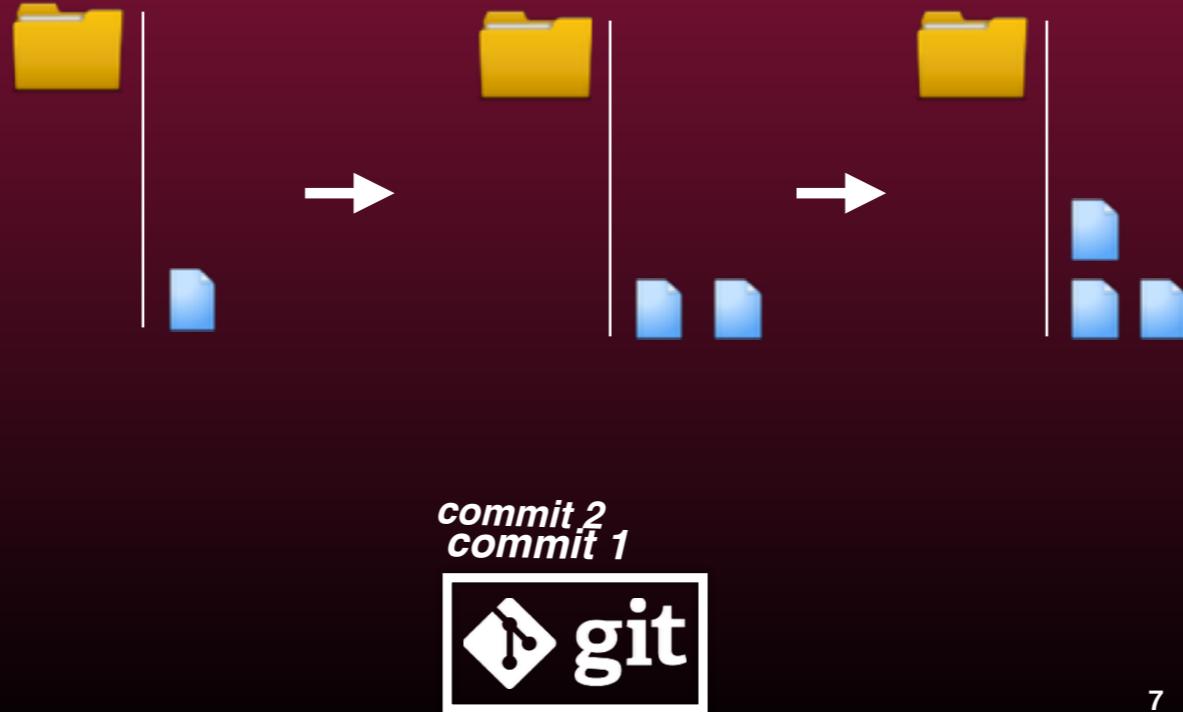


7

Cada vez que haces un cambio y lo confirmas él básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea.

Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información

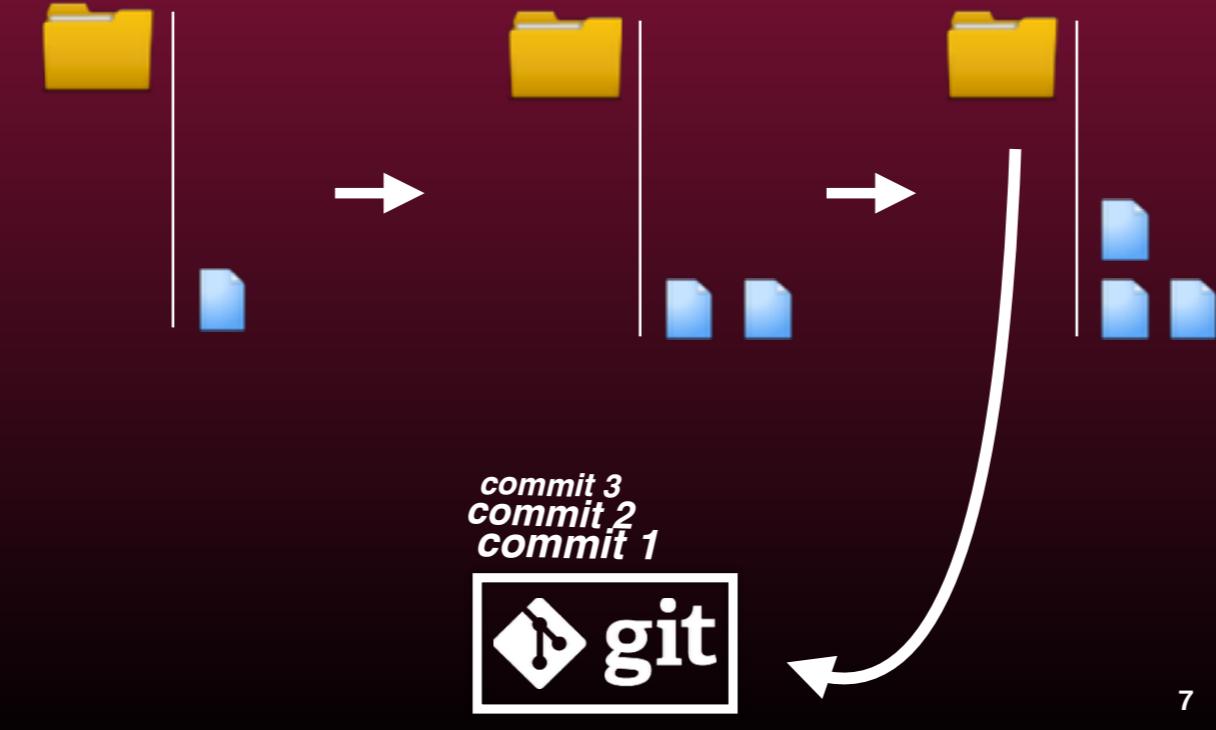
¿Cómo funciona?



Cada vez que haces un cambio y lo confirmas él básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea.

Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información

¿Cómo funciona?



7

Cada vez que haces un cambio y lo confirmas él básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea.

Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información

¿Cómo funciona?

8

Git funciona a nivel local. Esto es que, cuando tú decides implementar Git en tu repositorio éste crea una carpeta oculta en el raíz con su información. Cada vez que se hacen cambios éstos se almacenan en esta carpeta.

Ahora entra en juego el concepto de repositorios remotos, que son aquellos repositorios de los cuales tu repositorio local es un clon. Esto significa que cuando se haga algún cambio en local, estos cambios podrán "sincronizarse" con este repositorio remoto.

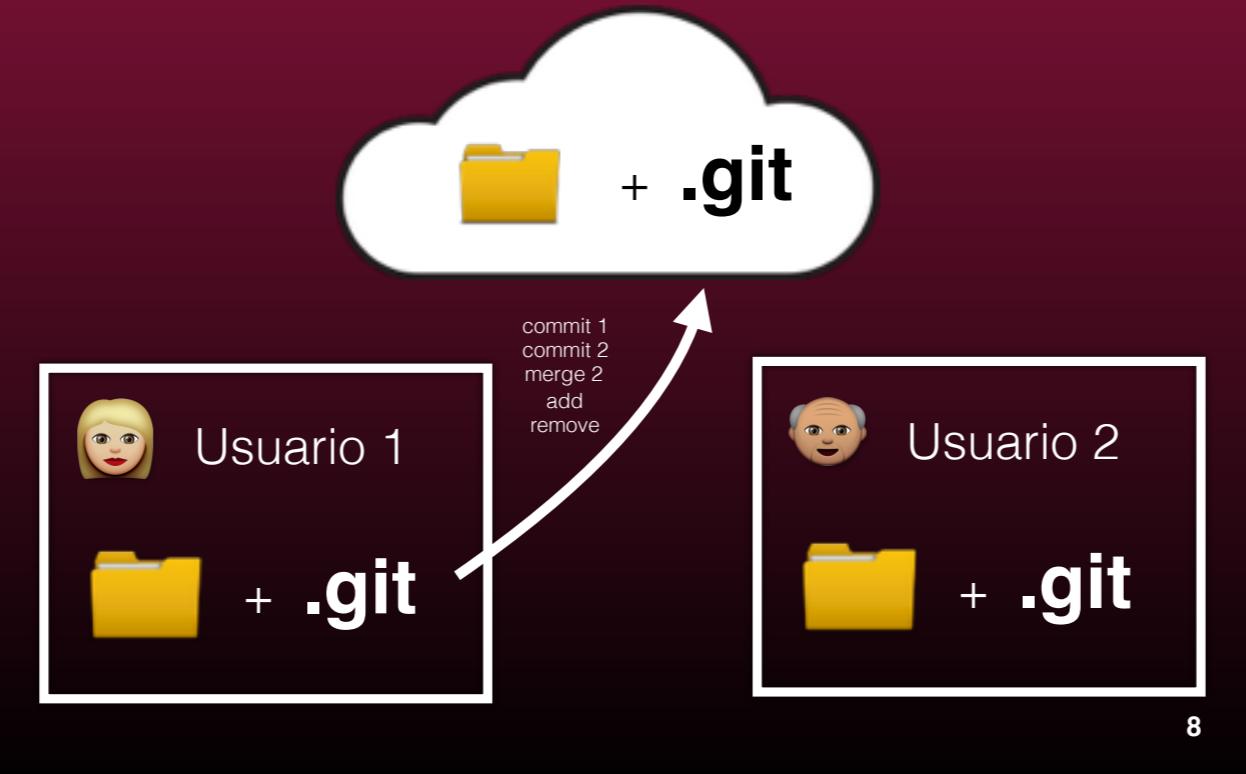
¿Cómo funciona?



Git funciona a nivel local. Esto es que, cuando tú decides implementar Git en tu repositorio éste crea una carpeta oculta en el raíz con su información. Cada vez que se hacen cambios éstos se almacenan en esta carpeta.

Ahora entra en juego el concepto de repositorios remotos, que son aquellos repositorios de los cuales tu repositorio local es un clon. Esto significa que cuando se haga algún cambio en local, estos cambios podrán "sincronizarse" con este repositorio remoto.

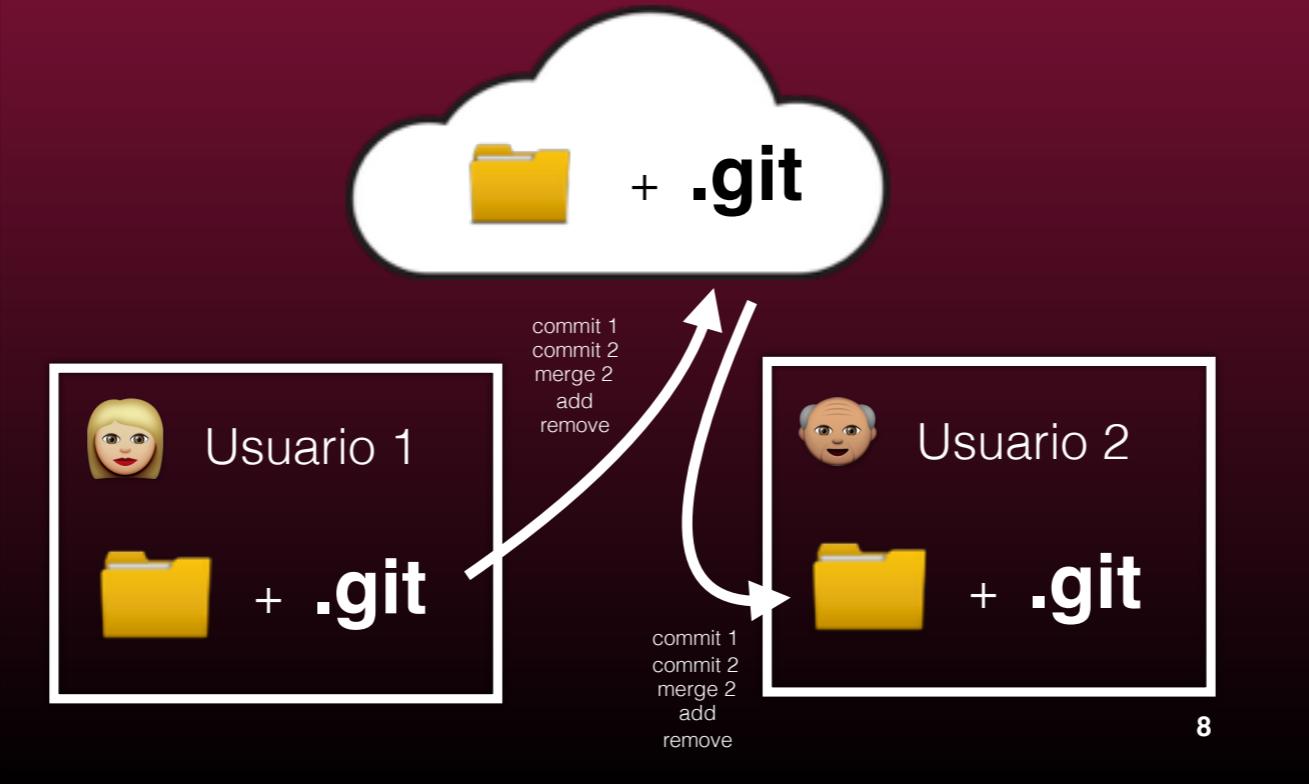
¿Cómo funciona?



Git funciona a nivel local. Esto es que, cuando tú decides implementar Git en tu repositorio éste crea una carpeta oculta en el raíz con su información. Cada vez que se hacen cambios éstos se almacenan en esta carpeta.

Ahora entra en juego el concepto de repositorios remotos, que son aquellos repositorios de los cuales tu repositorio local es un clon. Esto significa que cuando se haga algún cambio en local, estos cambios podrán "sincronizarse" con este repositorio remoto.

¿Cómo funciona?



Git funciona a nivel local. Esto es que, cuando tú decides implementar Git en tu repositorio éste crea una carpeta oculta en el raíz con su información. Cada vez que se hacen cambios éstos se almacenan en esta carpeta.

Ahora entra en juego el concepto de repositorios remotos, que son aquellos repositorios de los cuales tu repositorio local es un clon. Esto significa que cuando se haga algún cambio en local, estos cambios podrán "sincronizarse" con este repositorio remoto.

¿Como llevo el control de mi código?



9

Formas de gestionar el git.

Para administrar todo lo que se hace y se ha hecho, git provee de una interfaz por comandos.

—yoda —

La terminal de comandos es la forma por la que se hacen las acciones en el repositorio git. Todas las acciones disponibles para GIT se usan a través de la consola.

De forma paralela se puede gestionar estos comandos con una serie de aplicaciones pensadas para ello, que proveen al usuario de una interfaz gráfica.

Aclarar lo de que no se llama github, y decir lo que es gitub.

¿Como llevo el control de mi código?



9

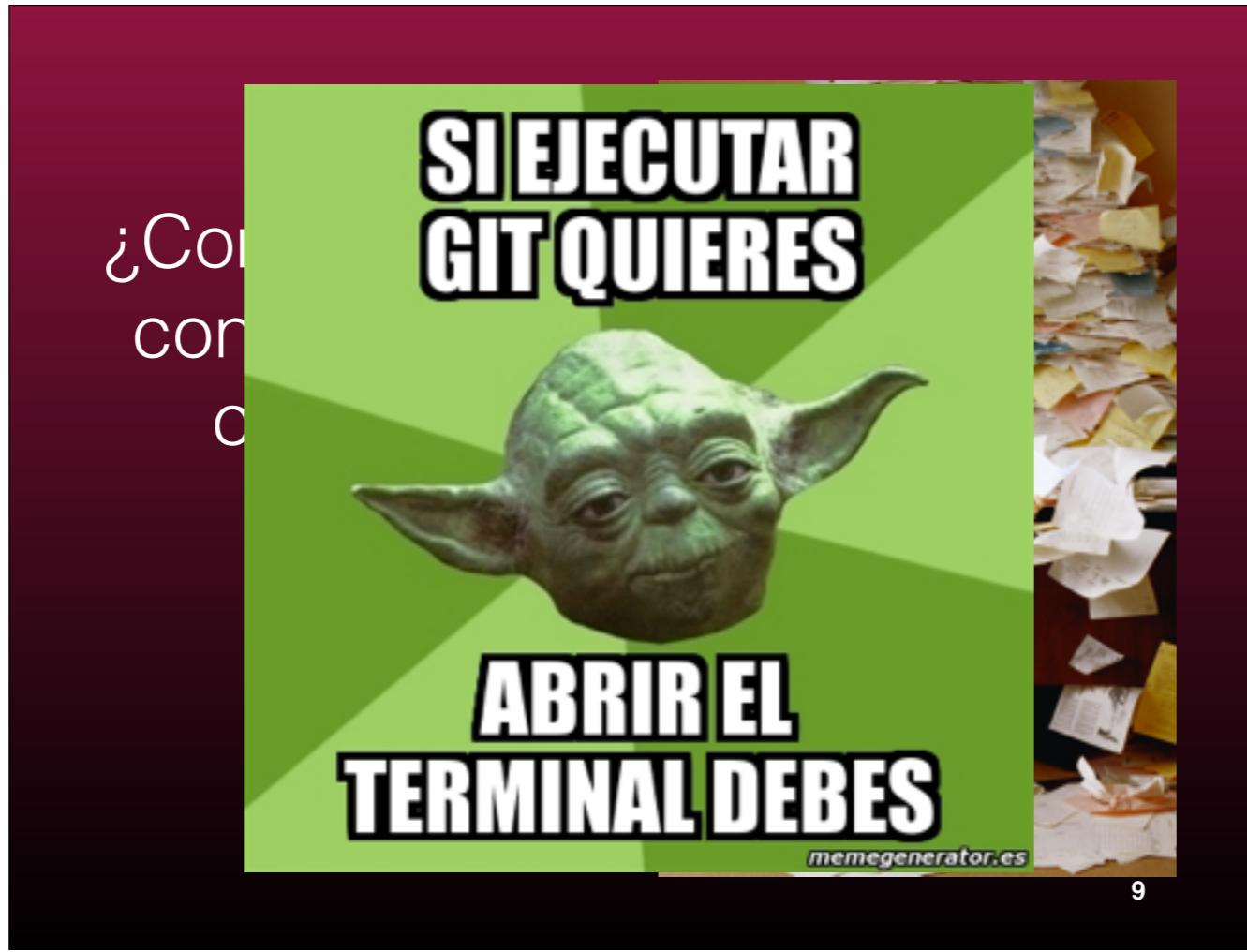
Formas de gestionar el git.

Para administrar todo lo que se hace y se ha hecho, git provee de una interfaz por comandos.

—yoda —

La terminal de comandos es la forma por la que se hacen las acciones en el repositorio git. Todas las acciones disponibles para GIT se usan a través de la consola. De forma paralela se puede gestionar estos comandos con una serie de aplicaciones pensadas para ello, que proveen al usuario de una interfaz gráfica.

Aclarar lo de que no se llama github, y decir lo que es gitub.



9

Formas de gestionar el git.

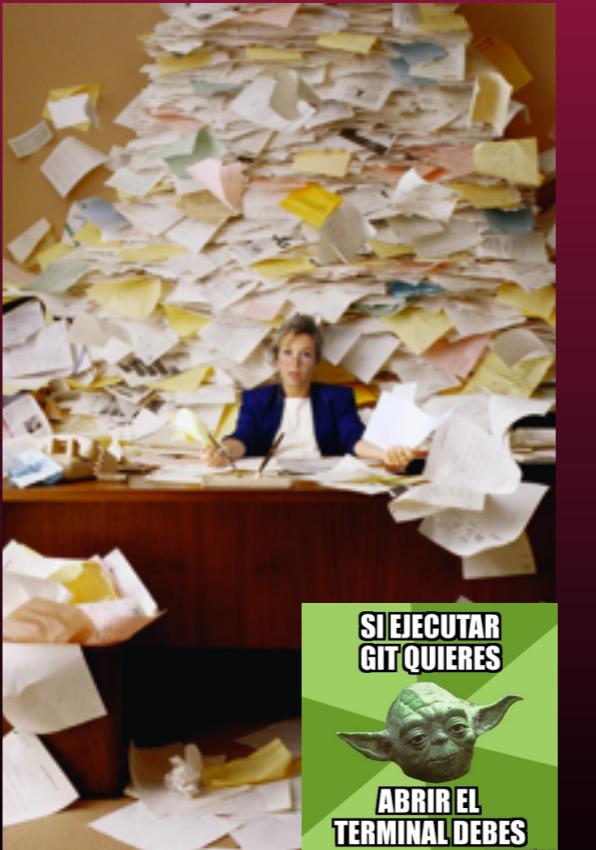
Para administrar todo lo que se hace y se ha hecho, git provee de una interfaz por comandos.

—yoda —

La terminal de comandos es la forma por la que se hacen las acciones en el repositorio git. Todas las acciones disponibles para GIT se usan a través de la consola. De forma paralela se puede gestionar estos comandos con una serie de aplicaciones pensadas para ello, que proveen al usuario de una interfaz gráfica.

Aclarar lo de que no se llama github, y decir lo que es gitub.

¿Como llevo el control de mi código?



9

Formas de gestionar el git.

Para administrar todo lo que se hace y se ha hecho, git provee de una interfaz por comandos.

—yoda —

La terminal de comandos es la forma por la que se hacen las acciones en el repositorio git. Todas las acciones disponibles para GIT se usan a través de la consola. De forma paralela se puede gestionar estos comandos con una serie de aplicaciones pensadas para ello, que proveen al usuario de una interfaz gráfica.

Aclarar lo de que no se llama github, y decir lo que es gitub.

Visual studio es compatible con Git desde la versión 2013 aproximadamente. Se comunica muy bien y permite trabajar con él. Incluso la nueva versión, la 2017 presentada ayer, ha hecho mejoras sobre GIT, de hecho la versión de Mac ya no soporta TFS.

Tiene problemas cuando un SLN no está dentro del GIT y administrando una app web con un proyecto git readdo desde fuera. No tiene los GIT ignore.



10

Visual studio es compatible con Git desde la versión 2013 aproximadamente. Se comunica muy bien y permite trabajar con él. Incluso la nueva versión, la 2017 presentada ayer, ha hecho mejoras sobre GIT, de hecho la versión de Mac ya no soporta TFS.

Tiene problemas cuando un SLN no está dentro del GIT y administrando una app web con un proyecto git readdo desde fuera. No tiene los GIT ignore.



Visual studio es compatible con Git desde la versión 2013 aproximadamente. Se comunica muy bien y permite trabajar con él. Incluso la nueva versión, la 2017 presentada ayer, ha hecho mejoras sobre GIT, de hecho la versión de Mac ya no soporta TFS.

Tiene problemas cuando un SLN no está dentro del GIT y administrando una app web con un proyecto git readdo desde fuera. No tiene los GIT ignore.



10

Visual studio es compatible con Git desde la versión 2013 aproximadamente. Se comunica muy bien y permite trabajar con él. Incluso la nueva versión, la 2017 presentada ayer, ha hecho mejoras sobre GIT, de hecho la versión de Mac ya no soporta TFS.

Tiene problemas cuando un SLN no está dentro del GIT y administrando una app web con un proyecto git reado desde fuera. No tiene los GIT ignore.

Conceptos básicos

commit

“guardar los cambios en el repositorio”

11

La acción más básica y significativa de GIT es el commit.

Commit significa guardar los cambios en el repositorio. Son todos esos puntos en las líneas que vemos.

Un commit se compone de una descripción (hay que ser concisos), un id formado mediante un checksum sha1 y el branch donde se realiza.

Hacer un commit implica un cambio, cualquier cambio, pero lo cierto es que un commit debería ser un cambio significativo y que tuviera sentido por sí solo.

Hablar de la importancia que tiene la descripción en el commit. Ahora vienen unos ejemplos de cómo no tiene que ser un commit.

Conceptos básicos

commit

“guardar los cambios en el repositorio”

```
[master 463dc4f] "ADD:Nuevo rayo de la muerte instalado"
```

11

La acción más básica y significativa de GIT es el commit.

Commit significa guardar los cambios en el repositorio. Son todos esos puntos en las líneas que vemos.

Un commit se compone de una descripción (hay que ser concisos), un id formado mediante un checksum sha1 y el branch donde se realiza.

Hacer un commit implica un cambio, cualquier cambio, pero lo cierto es que un commit debería ser un cambio significativo y que tuviera sentido por sí solo.

Hablar de la importancia que tiene la descripción en el commit. Ahora vienen unos ejemplos de cómo no tiene que ser un commit.

Conceptos básicos

commit

“guardar los cambios en el repositorio”

```
[master 463dc4f] “ADD:Nuevo rayo de la muerte instalado”
```

descripción

11

La acción más básica y significativa de GIT es el commit.

Commit significa guardar los cambios en el repositorio. Son todos esos puntos en las líneas que vemos.

Un commit se compone de una descripción (hay que ser concisos), un id formado mediante un checksum sha1 y el branch donde se realiza.

Hacer un commit implica un cambio, cualquier cambio, pero lo cierto es que un commit debería ser un cambio significativo y que tuviera sentido por sí solo.

Hablar de la importancia que tiene la descripción en el commit. Ahora vienen unos ejemplos de cómo no tiene que ser un commit.

Conceptos básicos

commit

“guardar los cambios en el repositorio”

```
[master 463dc4f] “ADD:Nuevo rayo de la muerte instalado”
```



11

La acción más básica y significativa de GIT es el commit.

Commit significa guardar los cambios en el repositorio. Son todos esos puntos en las líneas que vemos.

Un commit se compone de una descripción (hay que ser concisos), un id formado mediante un checksum sha1 y el branch donde se realiza.

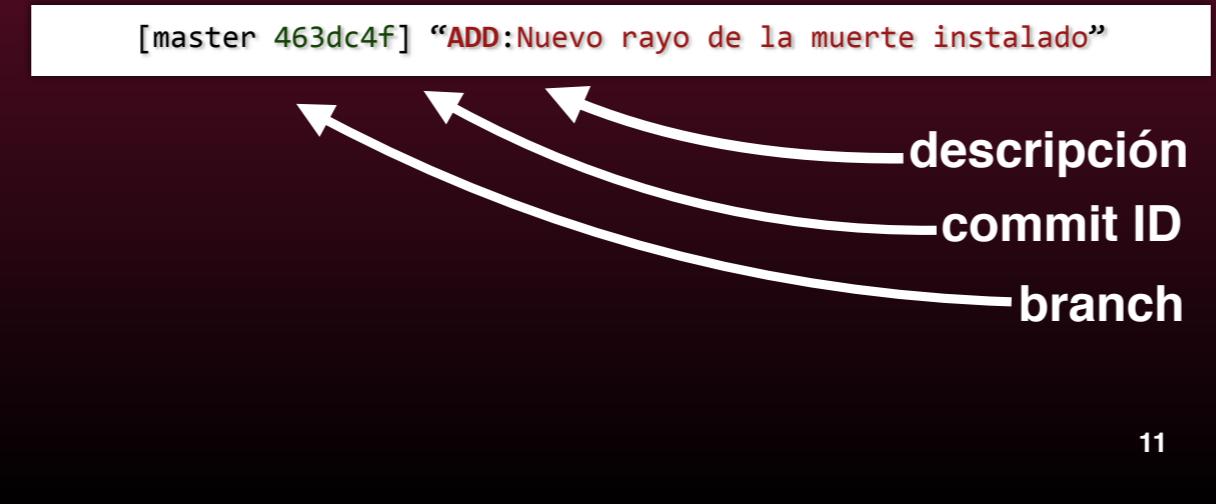
Hacer un commit implica un cambio, cualquier cambio, pero lo cierto es que un commit debería ser un cambio significativo y que tuviera sentido por sí solo.

Hablar de la importancia que tiene la descripción en el commit. Ahora vienen unos ejemplos de cómo no tiene que ser un commit.

Conceptos básicos

commit

“guardar los cambios en el repositorio”



11

La acción más básica y significativa de GIT es el commit.

Commit significa guardar los cambios en el repositorio. Son todos esos puntos en las líneas que vemos.

Un commit se compone de una descripción (hay que ser concisos), un id formado mediante un checksum sha1 y el branch donde se realiza.

Hacer un commit implica un cambio, cualquier cambio, pero lo cierto es que un commit debería ser un cambio significativo y que tuviera sentido por sí solo.

Hablar de la importancia que tiene la descripción en el commit. Ahora vienen unos ejemplos de cómo no tiene que ser un commit.



11

La acción más básica y significativa de GIT es el commit.

Commit significa guardar los cambios en el repositorio. Son todos esos puntos en las líneas que vemos.

Un commit se compone de una descripción (hay que ser concisos), un id formado mediante un checksum sha1 y el branch donde se realiza.

Hacer un commit implica un cambio, cualquier cambio, pero lo cierto es que un commit debería ser un cambio significativo y que tuviera sentido por sí solo.

Hablar de la importancia que tiene la descripción en el commit. Ahora vienen unos ejemplos de cómo no tiene que ser un commit.







14

Aún todavía podría ser más conciso, entre 30 y 50 caracteres.

Conceptos básicos

branch

“líneas paralelas de desarrollo”

15

Explicación de branch.

- En nuestra línea principal existen una serie de commits que sirven de precedente para montar una determinada funcionalidad de la aplicación.
- en un momento dado se tiene lo suficiente para montar una nueva funcionalidad (o necesidad) es entonces cuando se crea un branch nuevo sobre el que se trabaja, de forma paralela a la que se trabaja en el master.
- Un branch tiene un nexo de unión entre su padre.
- Se trabajará de forma paralela en el branch padre.
- Llegará un momento en el que estos dos branch tengan que converger, tal como lo hicieron al principio.

No significa la muerte del branch, pero es recomendable, lo explicaré más tarde.

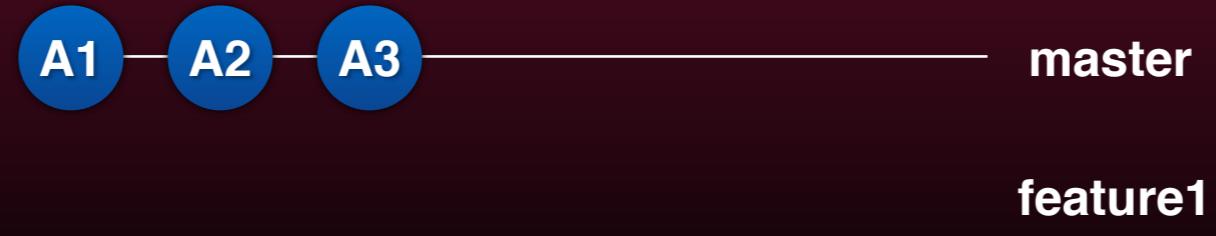
Se pueden crear todos los branch que sean necesarios, no ocupan espacio de más.

Un branch puede tener un “mirror” en remoto, pero también puede quedarse estrictamente en local.

Conceptos básicos

branch

“líneas paralelas de desarrollo”



15

Explicación de branch.

- En nuestra línea principal existen una serie de commits que sirven de precedente para montar una determinada funcionalidad de la aplicación.
- en un momento dado se tiene lo suficiente para montar una nueva funcionalidad (o necesidad) es entonces cuando se crea un branch nuevo sobre el que se trabaja, de forma paralela a la que se trabaja en el master.
- Un branch tiene un nexo de unión entre su padre.
- Se trabajará de forma paralela en el branch padre.
- Llegará un momento en el que estos dos branch tengan que converger, tal como lo hicieron al principio.

No significa la muerte del branch, pero es recomendable, lo explicaré más tarde.

Se pueden crear todos los branch que sean necesarios, no ocupan espacio de más.

Un branch puede tener un “mirror” en remoto, pero también puede quedarse estrictamente en local.

Conceptos básicos

branch

“líneas paralelas de desarrollo”



15

Explicación de branch.

- En nuestra línea principal existen una serie de commits que sirven de precedente para montar una determinada funcionalidad de la aplicación.
- en un momento dado se tiene lo suficiente para montar una nueva funcionalidad (o necesidad) es entonces cuando se crea un branch nuevo sobre el que se trabaja, de forma paralela a la que se trabaja en el master.
- Un branch tiene un nexo de unión entre su padre.
- Se trabajará de forma paralela en el branch padre.
- Llegará un momento en el que estos dos branch tengan que converger, tal como lo hicieron al principio.

No significa la muerte del branch, pero es recomendable, lo explicaré más tarde.

Se pueden crear todos los branch que sean necesarios, no ocupan espacio de más.

Un branch puede tener un “mirror” en remoto, pero también puede quedarse estrictamente en local.

Conceptos básicos

branch

“líneas paralelas de desarrollo”



15

Explicación de branch.

- En nuestra línea principal existen una serie de commits que sirven de precedente para montar una determinada funcionalidad de la aplicación.
- en un momento dado se tiene lo suficiente para montar una nueva funcionalidad (o necesidad) es entonces cuando se crea un branch nuevo sobre el que se trabaja, de forma paralela a la que se trabaja en el master.
- Un branch tiene un nexo de unión entre su padre.
- Se trabajará de forma paralela en el branch padre.
- Llegará un momento en el que estos dos branch tengan que converger, tal como lo hicieron al principio.

No significa la muerte del branch, pero es recomendable, lo explicaré más tarde.

Se pueden crear todos los branch que sean necesarios, no ocupan espacio de más.

Un branch puede tener un “mirror” en remoto, pero también puede quedarse estrictamente en local.

Conceptos básicos

branch

“líneas paralelas de desarrollo”



15

Explicación de branch.

- En nuestra línea principal existen una serie de commits que sirven de precedente para montar una determinada funcionalidad de la aplicación.
- en un momento dado se tiene lo suficiente para montar una nueva funcionalidad (o necesidad) es entonces cuando se crea un branch nuevo sobre el que se trabaja, de forma paralela a la que se trabaja en el master.
- Un branch tiene un nexo de unión entre su padre.
- Se trabajará de forma paralela en el branch padre.
- Llegará un momento en el que estos dos branch tengan que converger, tal como lo hicieron al principio.

No significa la muerte del branch, pero es recomendable, lo explicaré más tarde.

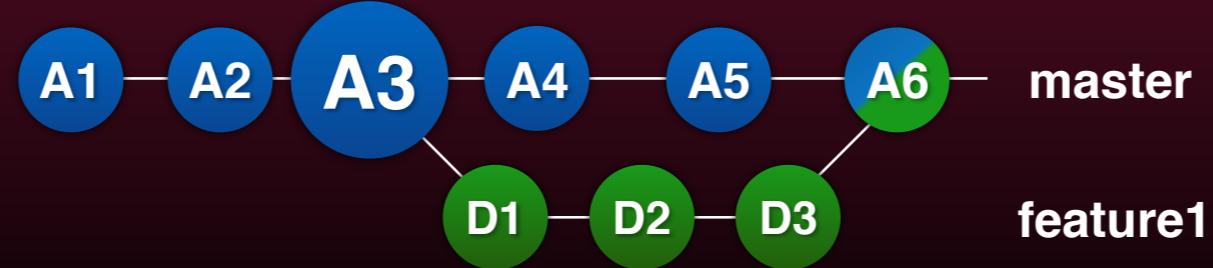
Se pueden crear todos los branch que sean necesarios, no ocupan espacio de más.

Un branch puede tener un “mirror” en remoto, pero también puede quedarse estrictamente en local.

Conceptos básicos

branch

“Líneas paralelas de desarrollo”



15

Explicación de branch.

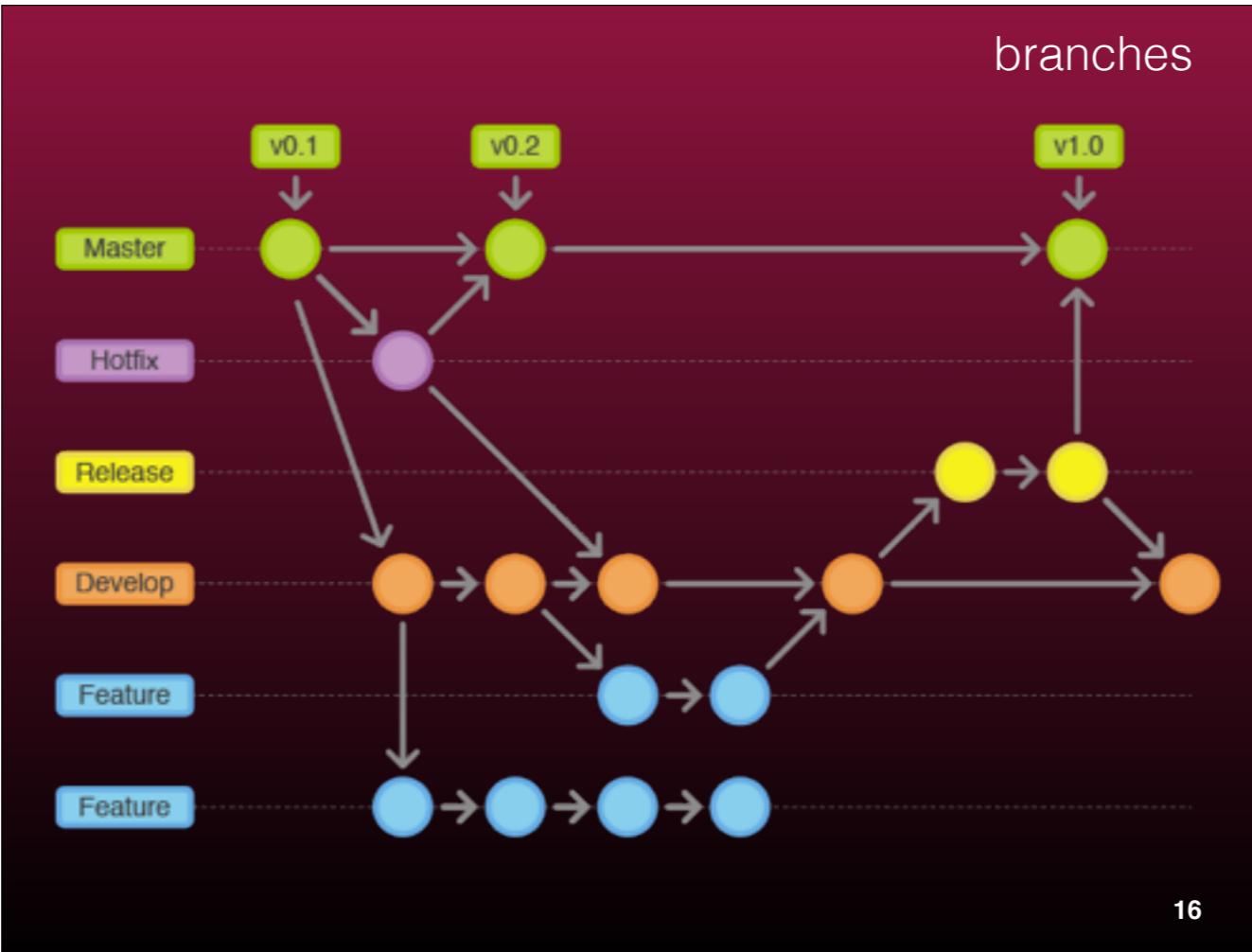
- En nuestra línea principal existen una serie de commits que sirven de precedente para montar una determinada funcionalidad de la aplicación.
- en un momento dado se tiene lo suficiente para montar una nueva funcionalidad (o necesidad) es entonces cuando se crea un branch nuevo sobre el que se trabaja, de forma paralela a la que se trabaja en el master.
- Un branch tiene un nexo de unión entre su padre.
- Se trabajará de forma paralela en el branch padre.
- Llegará un momento en el que estos dos branch tengan que converger, tal como lo hicieron al principio.

No significa la muerte del branch, pero es recomendable, lo explicaré más tarde.

Se pueden crear todos los branch que sean necesarios, no ocupan espacio de más.

Un branch puede tener un “mirror” en remoto, pero también puede quedarse estrictamente en local.

branches



16

Esto es un esquema real de desarrollo. Lo desarrollaremos más adelante.

Conceptos básicos

merge

“introduce cambios de un branch en otro distinto”

17

El merge es una acción por la cual un branch se une a otro por algún motivo.

Normalmente lo utilizaremos para incorporar una funcionalidad a nuestra rama de desarrollo.

El concepto de merge es bastante más amplio, significa crear un punto de unión entre dos ramas, independientemente de cuales sean.



- Para explicar como funciona un merge por dentro es necesario ver cómo están los dos branches antes del merge. desarrollándose de forma paralela con sus respectivos commits (adds, merges, cualquier cosa), los branches no se comunican entre sí, uno no sabe lo que hace el otro.
- Después del merge, se crea un nuevo commit en el branch de destino y los commits realizados en el branch feature se incorporarán tal cual estaban pero esta vez dentro del nuevo branch de destino.
- Para comprender cómo GIT junta el código de estos archivos es necesario entender que cogerá el estado previo al merge de la línea destino y añadirá **en bloque** los nuevos commit del branch feature. De esta forma sólo se tendrán que resolver los conflictos de una vez, al final en el commit de merge.

El orden en el que seincorporan estos cambios es irrelevante, pues los dos branch estaban incomunicados y lo que se hiciera en ellos no ha tenido influencia en el otro.

No es necesario que el branch muera tras un merge. Pero para evitar conflictos esto es necesario.

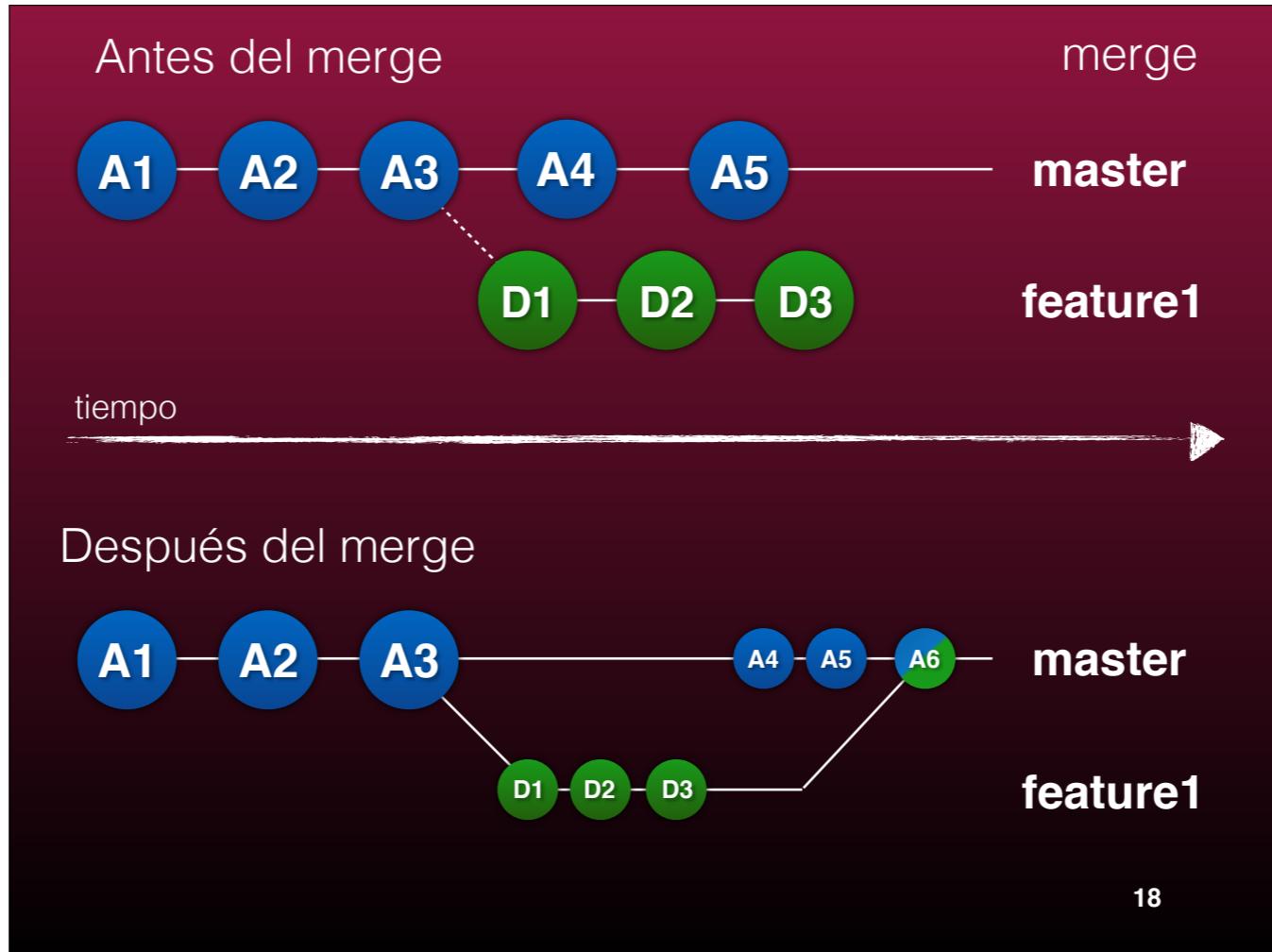


18

- Para explicar como funciona un merge por dentro es necesario ver cómo están los dos branches antes del merge. desarrollándose de forma paralela con sus respectivos commits (adds, merges, cualquier cosa), los branches no se comunican entre sí, uno no sabe lo que hace el otro.
- Después del merge, se crea un nuevo commit en el branch de destino y los commits realizados en el branch feature se incorporarán tal cual estaban pero esta vez dentro del nuevo branch de destino.
- Para comprender cómo GIT junta el código de estos archivos es necesario entender que cogerá el estado previo al merge de la línea destino y añadirá **en bloque** los nuevos commit del branch feature. De esta forma sólo se tendrán que resolver los conflictos de una vez, al final en el commit de merge.

El orden en el que seincorporan estos cambios es irrelevante, pues los dos branch estaban incomunicados y lo que se hiciera en ellos no ha tenido influencia en el otro.

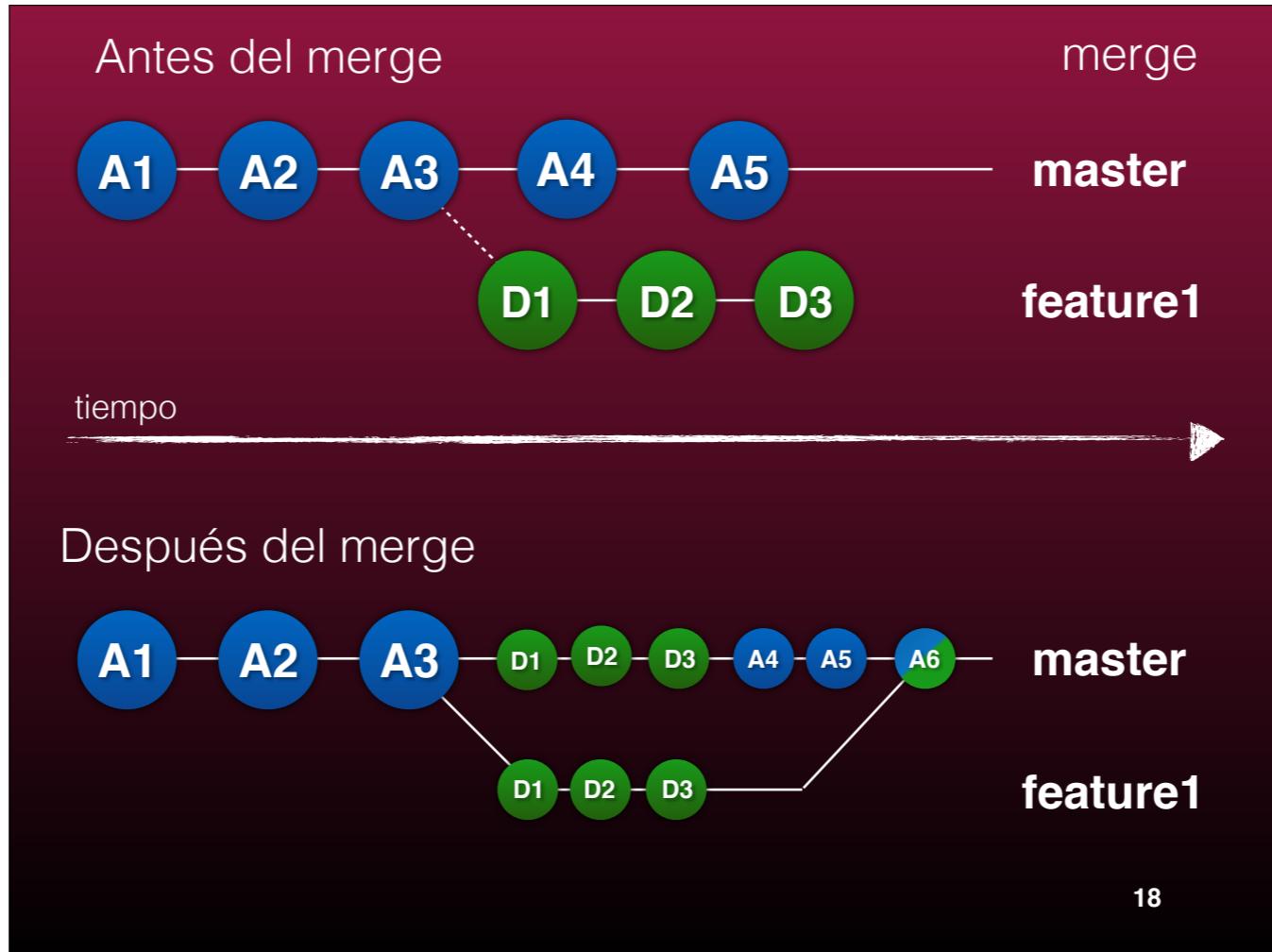
No es necesario que el branch muera tras un merge. Pero para evitar conflictos esto es necesario.



- Para explicar como funciona un merge por dentro es necesario ver cómo están los dos branches antes del merge. Desarrollándose de forma paralela con sus respectivos commits (adds, merges, cualquier cosa), los branches no se comunican entre sí, uno no sabe lo que hace el otro.
- Después del merge, se crea un nuevo commit en el branch de destino y los commits realizados en el branch feature se incorporarán tal cual estaban pero esta vez dentro del nuevo branch de destino.
- Para comprender cómo GIT junta el código de estos archivos es necesario entender que cogerá el estado previo al merge de la línea destino y añadirá **en bloque** los nuevos commit del branch feature. De esta forma sólo se tendrán que resolver los conflictos de una vez, al final en el commit de merge.

El orden en el que seincorporan estos cambios es irrelevante, pues los dos branch estaban incomunicados y lo que se hiciera en ellos no ha tenido influencia en el otro.

No es necesario que el branch muera tras un merge. Pero para evitar conflictos esto es necesario.



- Para explicar como funciona un merge por dentro es necesario ver cómo están los dos branches antes del merge. Desarrollándose de forma paralela con sus respectivos commits (adds, merges, cualquier cosa), los branches no se comunican entre sí, uno no sabe lo que hace el otro.
- Después del merge, se crea un nuevo commit en el branch de destino y los commits realizados en el branch feature se incorporarán tal cual estaban pero esta vez dentro del nuevo branch de destino.
- Para comprender cómo GIT junta el código de estos archivos es necesario entender que cogerá el estado previo al merge de la línea destino y añadirá **en bloque** los nuevos commit del branch feature. De esta forma sólo se tendrán que resolver los conflictos de una vez, al final en el commit de merge.

El orden en el que seincorporan estos cambios es irrelevante, pues los dos branch estaban incomunicados y lo que se hiciera en ellos no ha tenido influencia en el otro.

No es necesario que el branch muera tras un merge. Pero para evitar conflictos esto es necesario.

Possibles escenarios

19

Los posibles escenarios que nos podemos encontrar tras un merge son los siguientes.

- 1.No hay conflictos, la mayor parte del tiempo será así.
- 2.No hay conflictos, pero debería, porque un compañero ha estado trabajando en los mismos archivos. Comprobar si:
 - 1.Estoy en el branch que debería.
 - 2.Si yo he recibido sus cambios previamente.
 - 3.Si mi compañero ha subido los cambios.

Conflictos, conflictos por todas partes. Hora de lidiar con ellos. Más adelante veremos que VS estudio tiene una herramienta muy potente para ello.

Aclarar que GIT ya resuelve él de forma interna los conflictos, por lo que cuando pide al usuario que resuelva los conflictos es porque hay algo raro de verdad.

Posibles escenarios



19

Los posibles escenarios que nos podemos encontrar tras un merge son los siguientes.

1. No hay conflictos, la mayor parte del tiempo será así.
2. No hay conflictos, pero debería, porque un compañero ha estado trabajando en los mismos archivos. Comprobar si:
 1. Estoy en el branch que debería.
 2. Si yo he recibido sus cambios previamente.
 3. Si mi compañero ha subido los cambios.

Conflictos, conflictos por todas partes. Hora de lidiar con ellos. Más adelante veremos que VS estudio tiene una herramienta muy potente para ello.

Aclarar que GIT ya resuelve él de forma interna los conflictos, por lo que cuando pide al usuario que resuelva los conflictos es porque hay algo raro de verdad.

Possibles escenarios



19

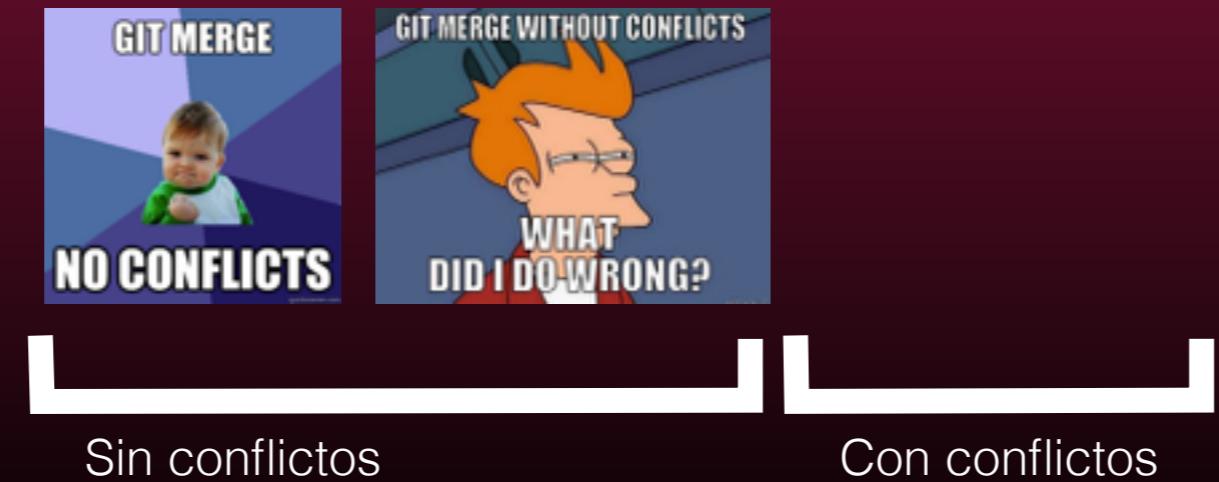
Los posibles escenarios que nos podemos encontrar tras un merge son los siguientes.

1. No hay conflictos, la mayor parte del tiempo será así.
2. No hay conflictos, pero debería, porque un compañero ha estado trabajando en los mismos archivos. Comprobar si:
 1. Estoy en el branch que debería.
 2. Si yo he recibido sus cambios previamente.
 3. Si mi compañero ha subido los cambios.

Conflictos, conflictos por todas partes. Hora de lidiar con ellos. Más adelante veremos que VS estudio tiene una herramienta muy potente para ello.

Aclarar que GIT ya resuelve él de forma interna los conflictos, por lo que cuando pide al usuario que resuelva los conflictos es porque hay algo raro de verdad.

Posibles escenarios



Los posibles escenarios que nos podemos encontrar tras un merge son los siguientes.

- 1.No hay conflictos, la mayor parte del tiempo será así.
 - 2.No hay conflictos, pero debería, porque un compañero ha estado trabajando en los mismos archivos. Comprobar si:
 - 1.Estoy en el branch que debería.
 - 2.Si yo he recibido sus cambios previamente.
 - 3.Si mi compañero ha subido los cambios.

Conflictos, conflictos por todas partes. Hora de lidiar con ellos. Más adelante veremos que VS estudio tiene una herramienta muy potente para ello.

Aclarar que GIT ya resuelve él de forma interna los conflictos, por lo que cuando pide al usuario que resuelva los conflictos es porque hay algo raro de verdad.

Possibles escenarios



19

Los posibles escenarios que nos podemos encontrar tras un merge son los siguientes.

1. No hay conflictos, la mayor parte del tiempo será así.
2. No hay conflictos, pero debería, porque un compañero ha estado trabajando en los mismos archivos. Comprobar si:
 1. Estoy en el branch que debería.
 2. Si yo he recibido sus cambios previamente.
 3. Si mi compañero ha subido los cambios.

Conflictos, conflictos por todas partes. Hora de lidiar con ellos. Más adelante veremos que VS estudio tiene una herramienta muy potente para ello.

Aclarar que GIT ya resuelve él de forma interna los conflictos, por lo que cuando pide al usuario que resuelva los conflictos es porque hay algo raro de verdad.

Conceptos básicos

pull

“traerse cambios del branch remoto”

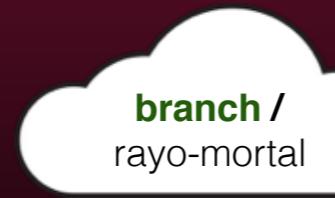
20

Pull, es esa acción por la que nos traemos los cambios del branch remoto a nuestro branch local.

Conceptos básicos

pull

“traerse cambios del branch remoto”



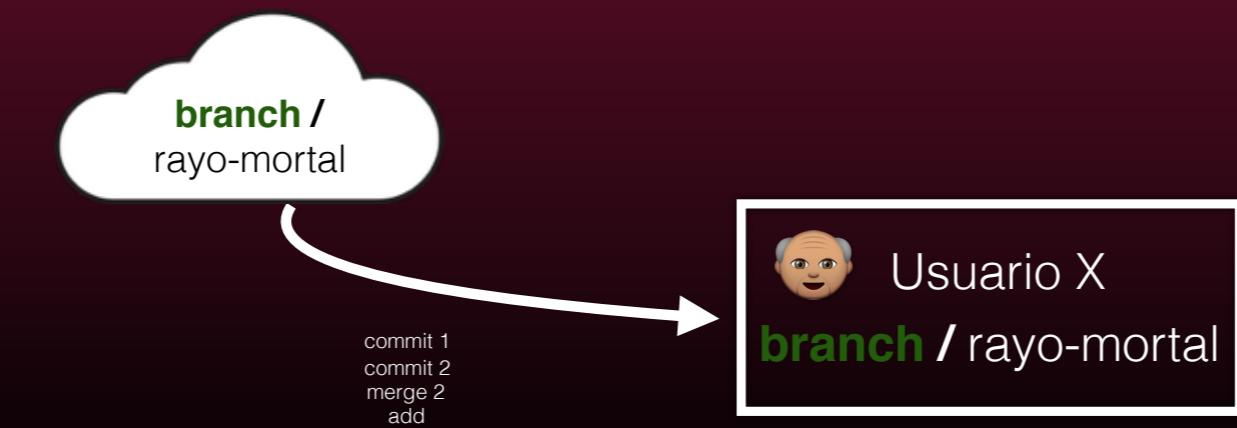
20

Pull, es esa acción por la que nos traemos los cambios del branch remoto a nuestro branch local.

Conceptos básicos

pull

“traerse cambios del branch remoto”



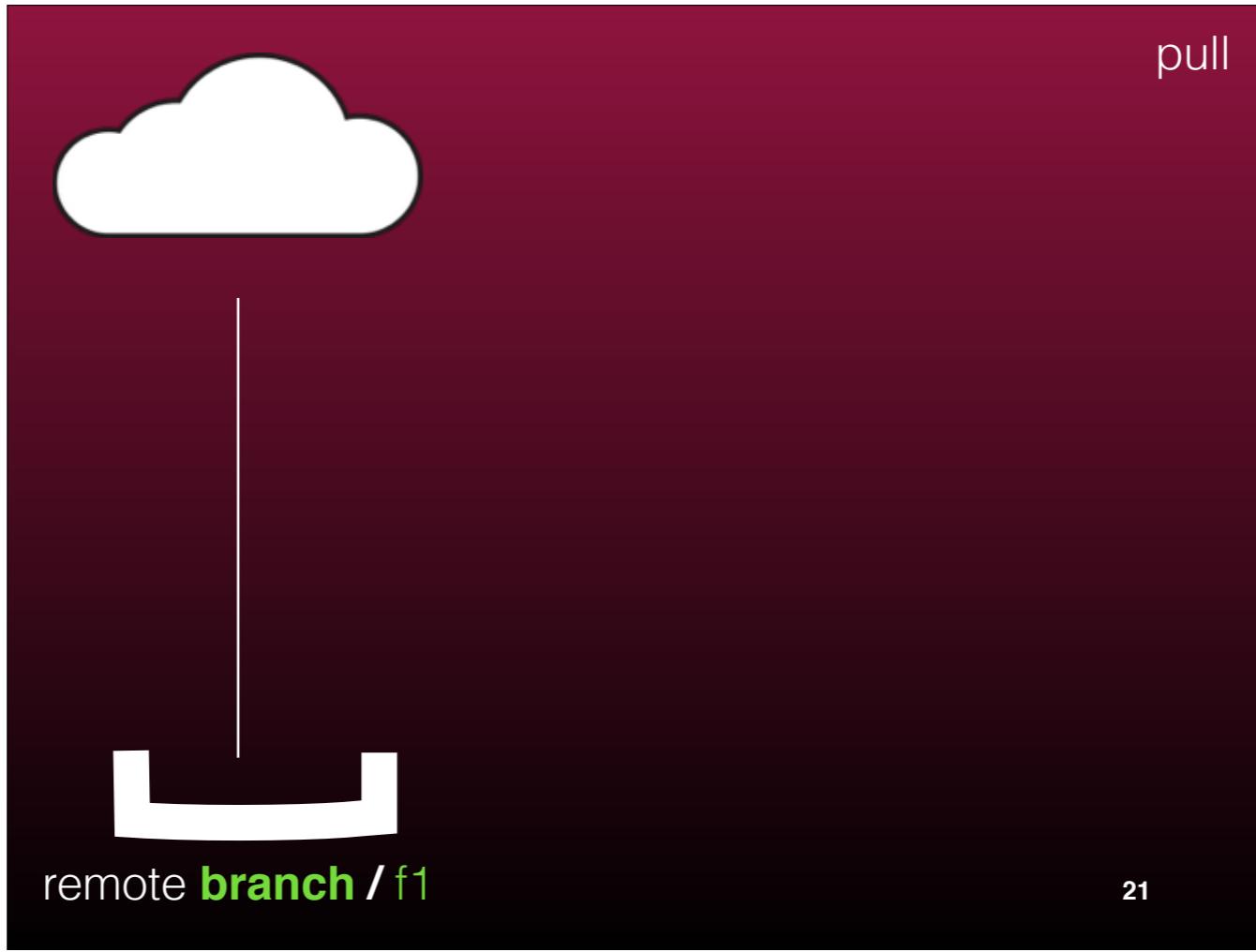
20

Pull, es esa acción por la que nos traemos los cambios del branch remoto a nuestro branch local.

- Tenemos dos partes, la remota.
- La parte local, con su repositorio y los archivos reales de trabajo.

Un pull se compone en realidad de dos acciones:

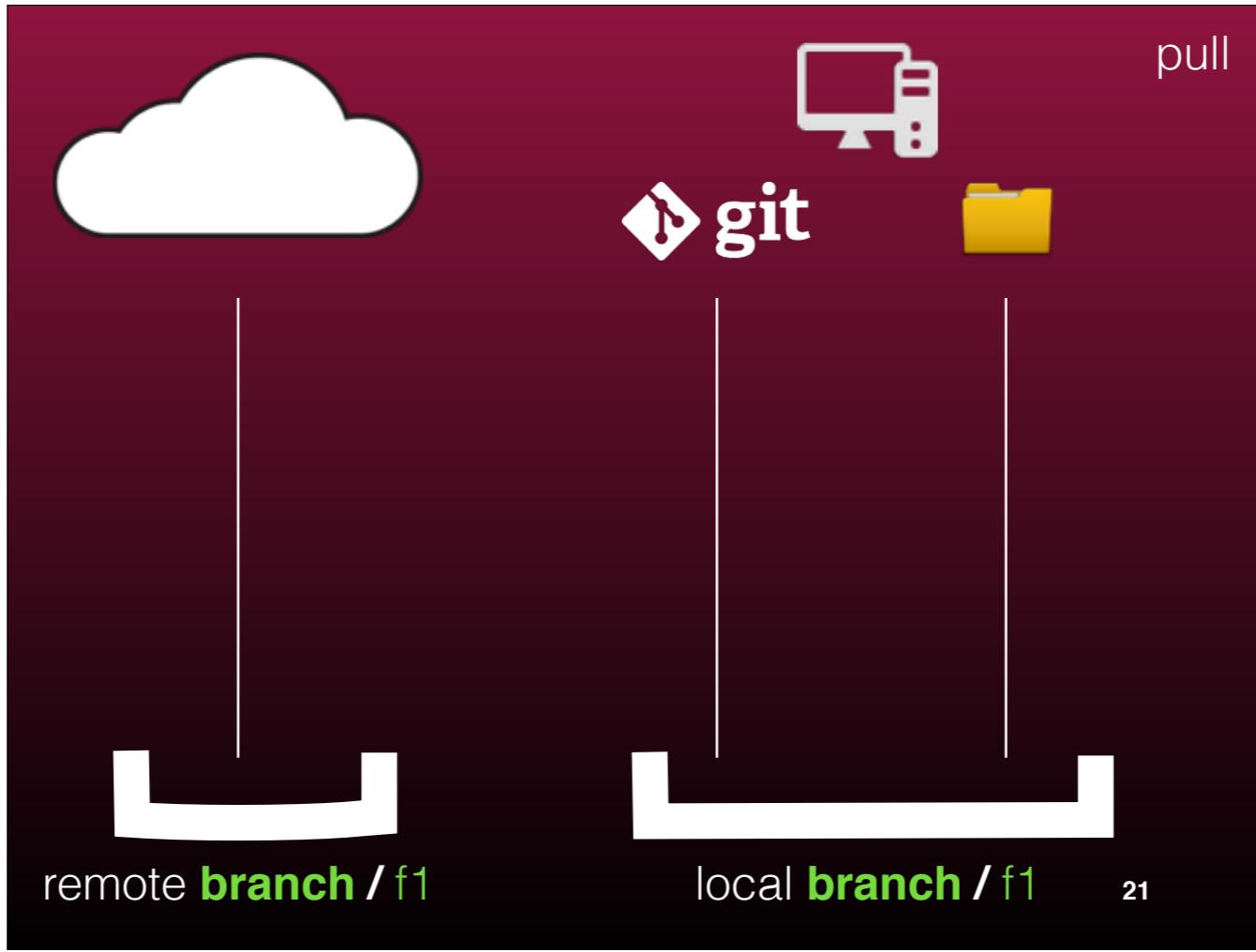
- Fetch por un lado lo que hace es reflejar los cambios del remoto dentro de nuestro repositorio local, de nuestra “base de datos” de cambios. Este es el momento en el que podemos hacer una copia de seguridad de nuestro código sabiendo qué cambios se van a hacer, antes de hacerlos.
- Merge, por el cual los cambios reflejados en nuestra base de datos se “materializan” dentro de nuestros archivos de trabajo (workspace).



- Tenemos dos partes, la remota.
- La parte local, con su repositorio y los archivos reales de trabajo.

Un pull se compone en realidad de dos acciones:

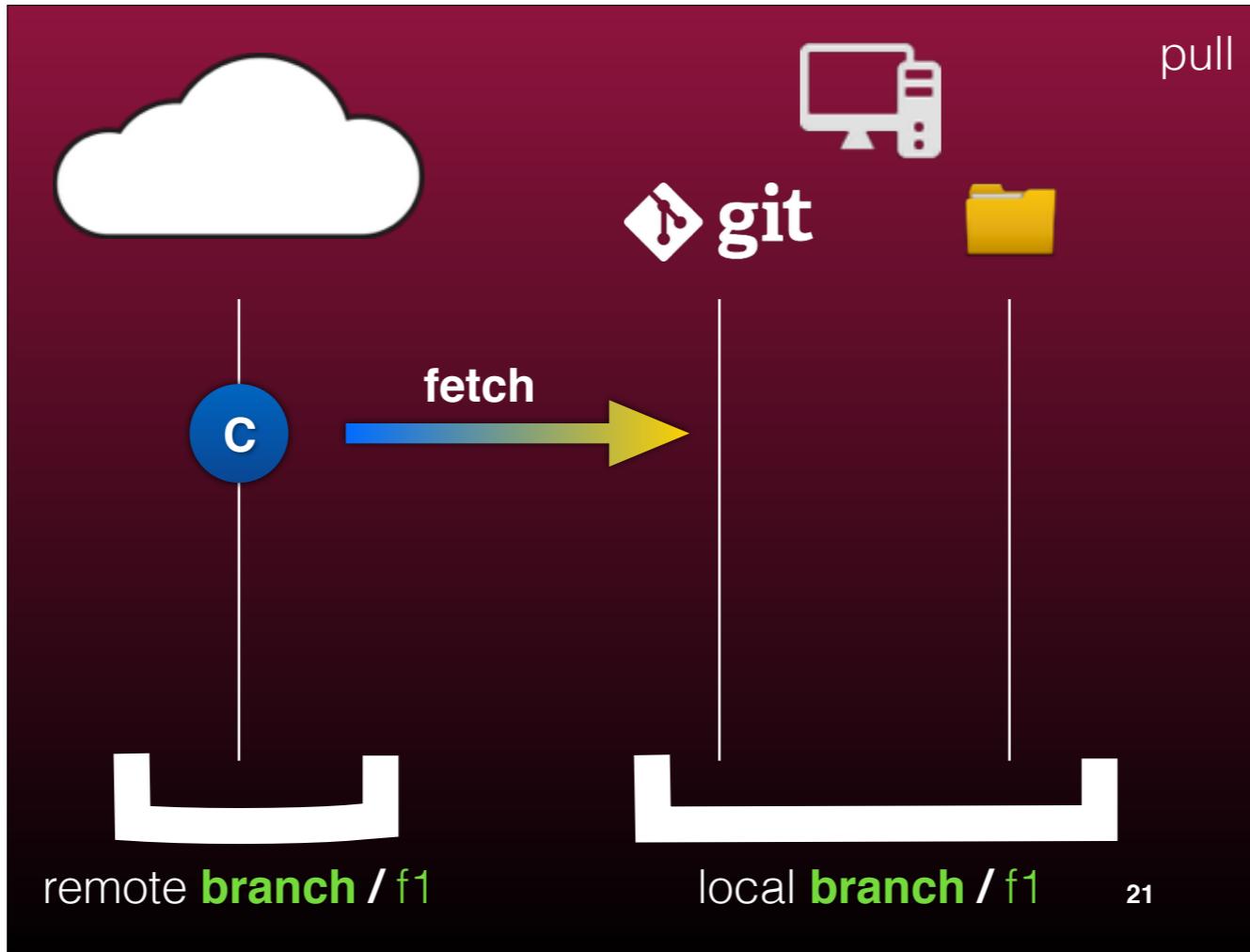
- Fetch por un lado lo que hace es reflejar los cambios del remoto dentro de nuestro repositorio local, de nuestra “base de datos” de cambios. Este es el momento en el que podemos hacer una copia de seguridad de nuestro código sabiendo qué cambios se van a hacer, antes de hacerlos.
- Merge, por el cual los cambios reflejados en nuestra base de datos se “materializan” dentro de nuestros archivos de trabajo (workspace).



- Tenemos dos partes, la remota.
- La parte local, con su repositorio y los archivos reales de trabajo.

Un pull se compone en realidad de dos acciones:

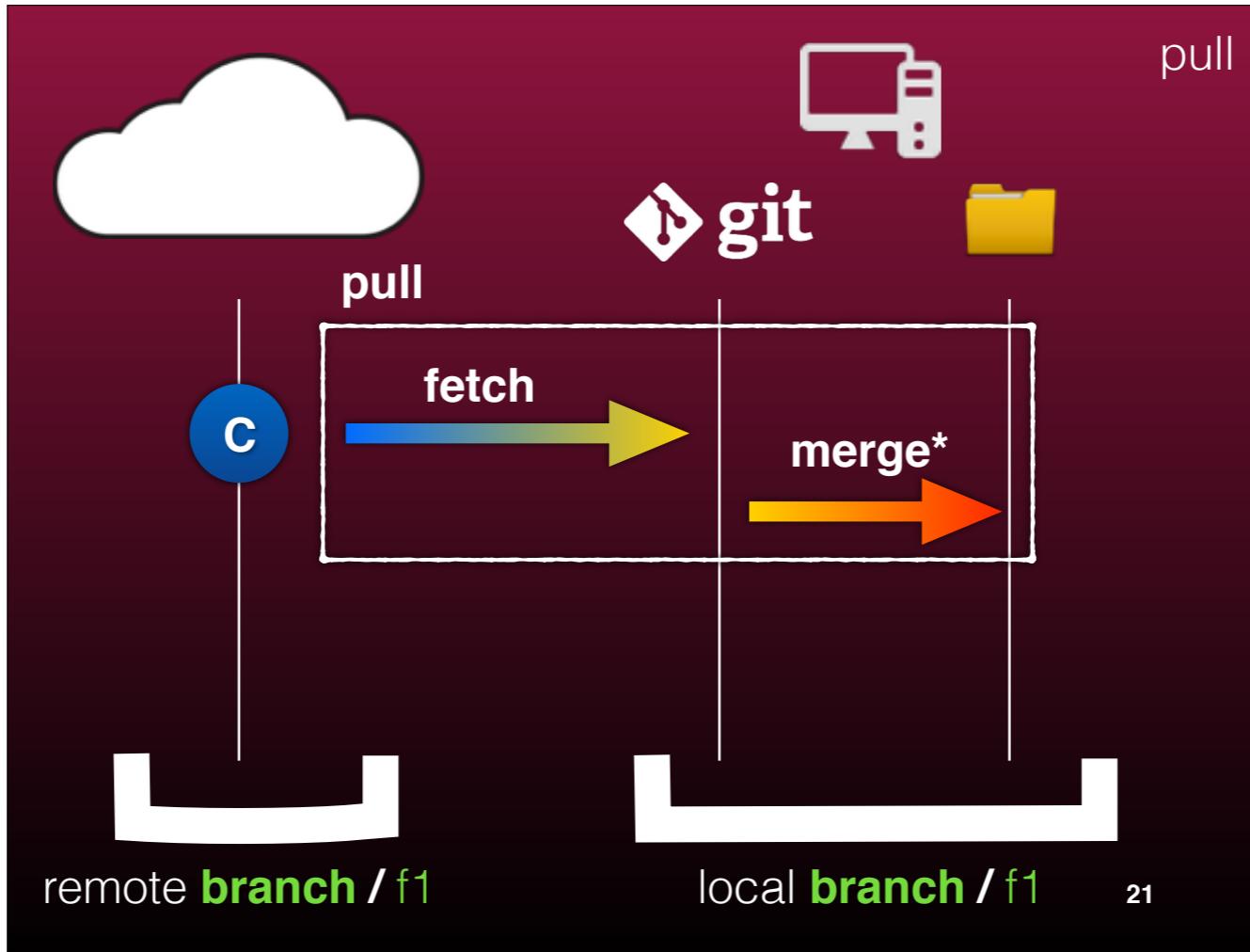
- Fetch por un lado lo que hace es reflejar los cambios del remoto dentro de nuestro repositorio local, de nuestra “base de datos” de cambios.
- Este es el momento en el que podemos hacer una copia de seguridad de nuestro código sabiendo qué cambios se van a hacer, antes de hacerlos.
- Merge, por el cual los cambios reflejados en nuestra base de datos se “materializan” dentro de nuestros archivos de trabajo (workspace).



- Tenemos dos partes, la remota.
- La parte local, con su repositorio y los archivos reales de trabajo.

Un pull se compone en realidad de dos acciones:

- Fetch por un lado lo que hace es reflejar los cambios del remoto dentro de nuestro repositorio local, de nuestra “base de datos” de cambios.
- Este es el momento en el que podemos hacer una copia de seguridad de nuestro código sabiendo qué cambios se van a hacer, antes de hacerlos.
- Merge, por el cual los cambios reflejados en nuestra base de datos se “materializan” dentro de nuestros archivos de trabajo (workspace).



- Tenemos dos partes, la remota.
- La parte local, con su repositorio y los archivos reales de trabajo.

Un pull se compone en realidad de dos acciones:

- Fetch por un lado lo que hace es reflejar los cambios del remoto dentro de nuestro repositorio local, de nuestra “base de datos” de cambios. Este es el momento en el que podemos hacer una copia de seguridad de nuestro código sabiendo qué cambios se van a hacer, antes de hacerlos.
- Merge, por el cual los cambios reflejados en nuestra base de datos se “materializan” dentro de nuestros archivos de trabajo (workspace).

Conceptos básicos

push

“llevar cambios al remoto”

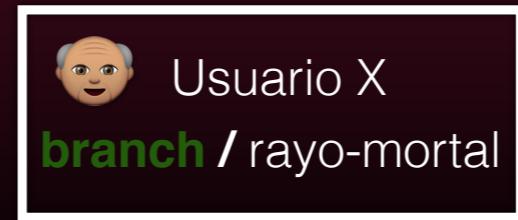
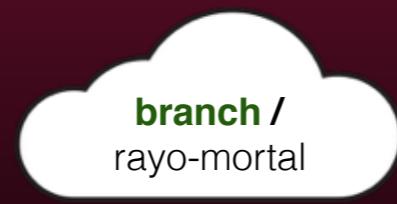
22

Push, es la acción por la cual nuestros commits (CAMBIOS YA REALIZADOS) se reflejan en el repositorio remoto.

Conceptos básicos

push

“llevar cambios al remoto”



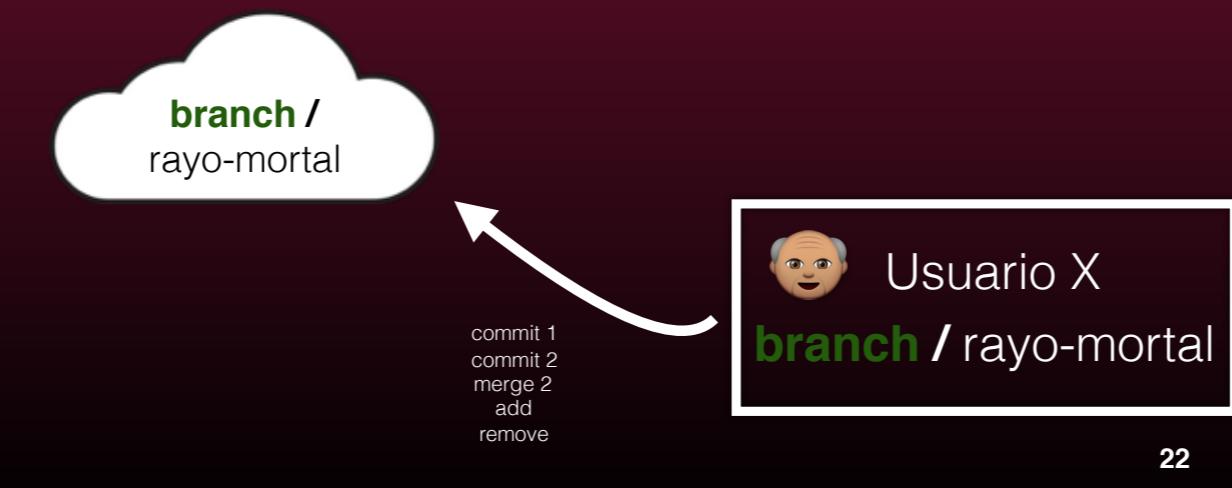
22

Push, es la acción por la cual nuestros commits (CAMBIOS YA REALIZADOS) se reflejan en el repositorio remoto.

Conceptos básicos

push

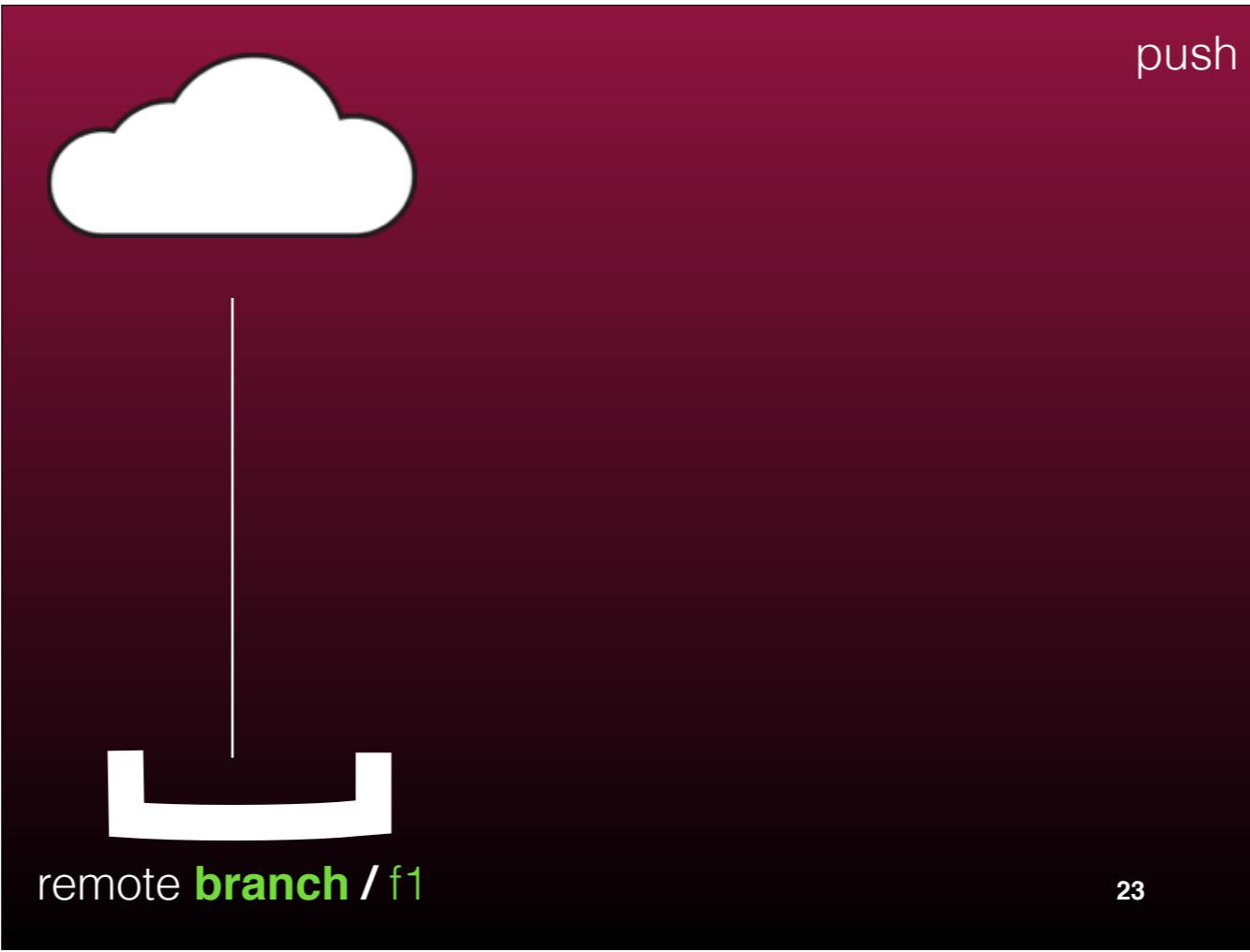
“llevar cambios al remoto”



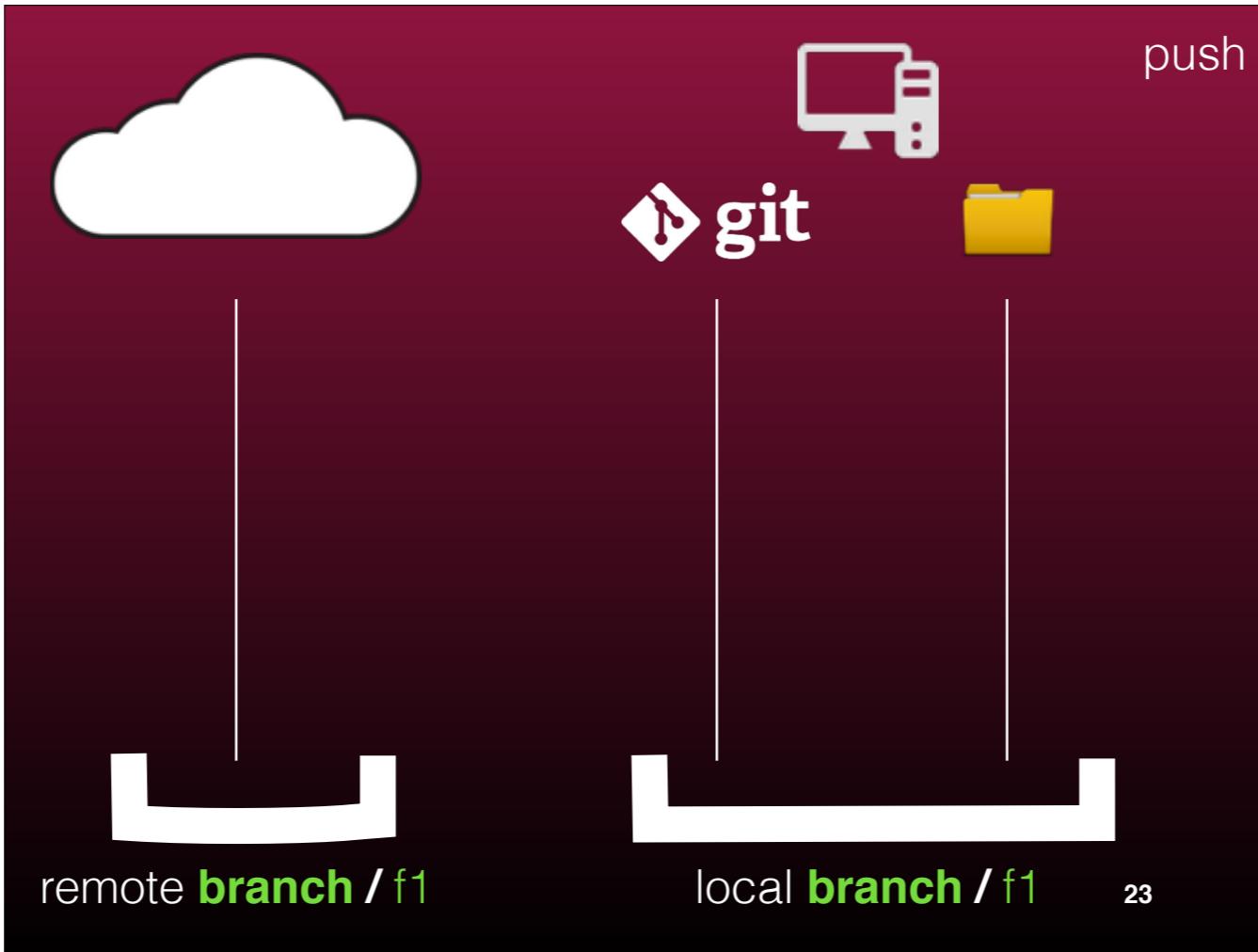
22

Push, es la acción por la cual nuestros commits (CAMBIOS YA REALIZADOS) se reflejan en el repositorio remoto.

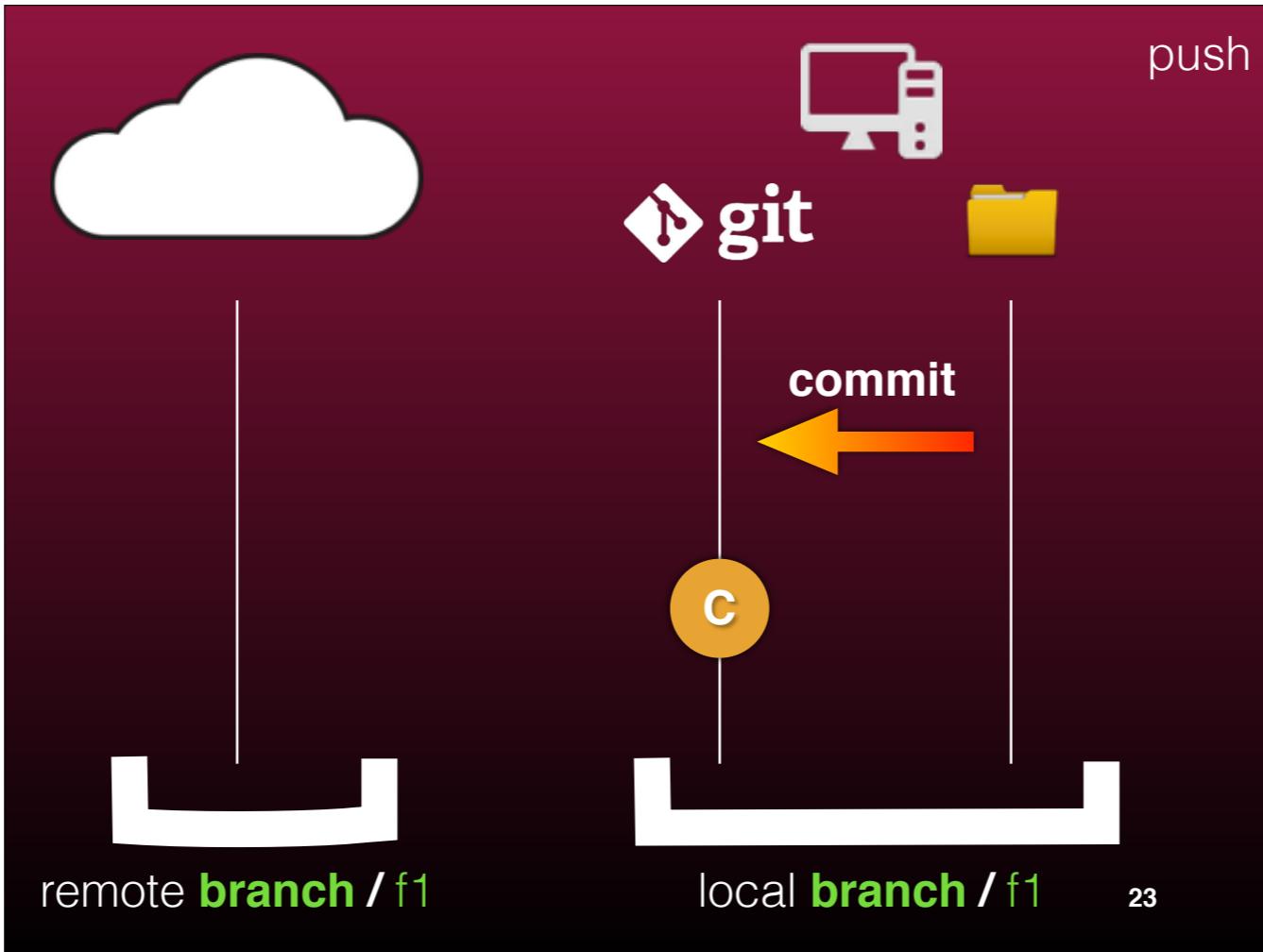
- Para realizar un push, primero tenemos que tener los commit hechos en nuestro repositorio LOCAL.
- Los commit realizados en nuestro repositorio local se reflejarán en el repositorio remoto.
- Es la forma que tenemos de guardar nuestro trabajo en la nube y no perderlo.



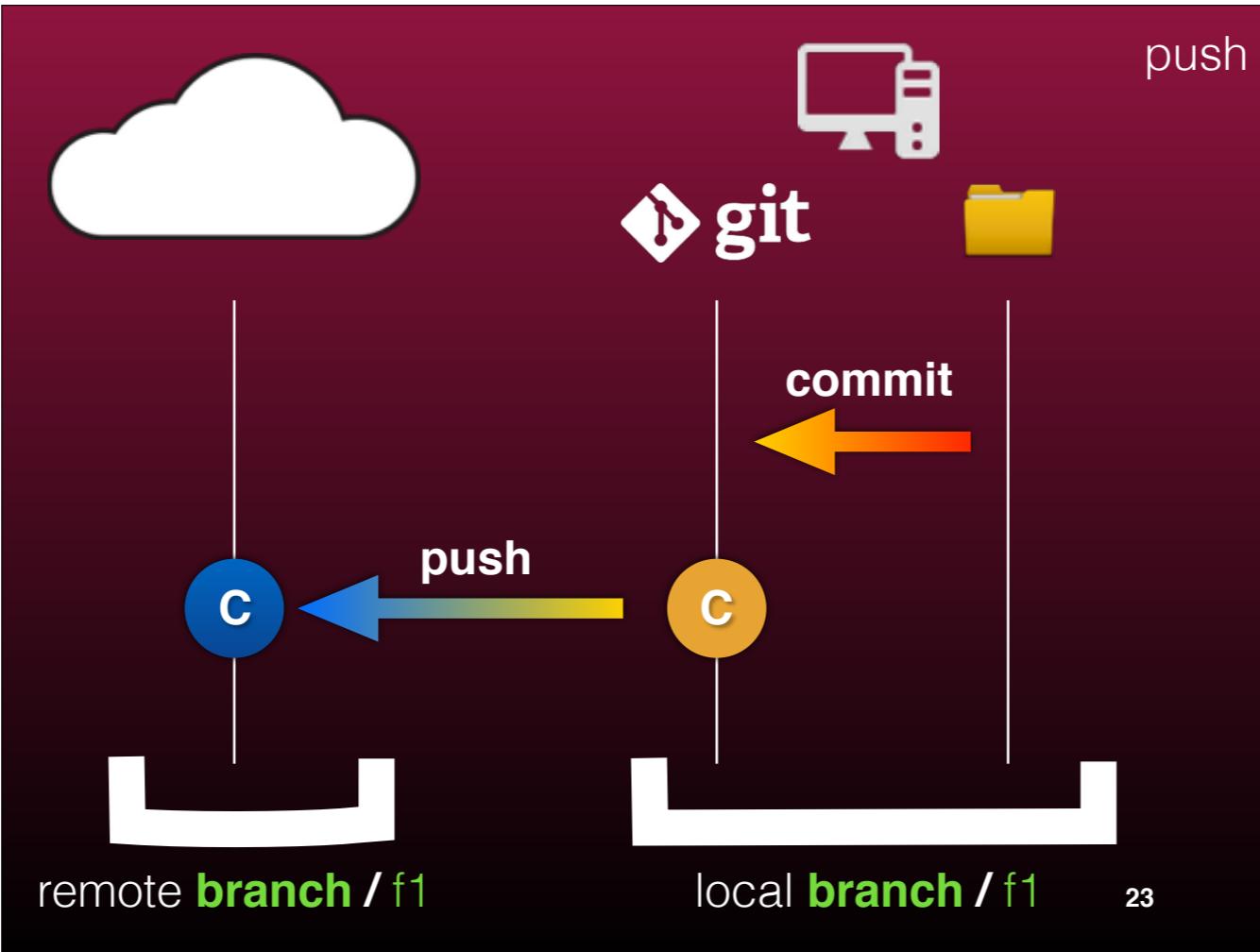
- Para realizar un push, primero tenemos que tener los commit hechos en nuestro repositorio LOCAL.
- Los commit realizados en nuestro repositorio local se reflejarán en el repositorio remoto.
- Es la forma que tenemos de guardar nuestro trabajo en la nube y no perderlo.



- Para realizar un push, primero tenemos que tener los commit hechos en nuestro repositorio LOCAL.
- Los commit realizados en nuestro repositorio local se reflejarán en el repositorio remoto.
- Es la forma que tenemos de guardar nuestro trabajo en la nube y no perderlo.



- Para realizar un push, primero tenemos que tener los commit hechos en nuestro repositorio LOCAL.
- Los commit realizados en nuestro repositorio local se reflejarán en el repositorio remoto.
- Es la forma que tenemos de guardar nuestro trabajo en la nube y no perderlo.



- Para realizar un push, primero tenemos que tener los commit hechos en nuestro repositorio LOCAL.
- Los commit realizados en nuestro repositorio local se reflejarán en el repositorio remoto.
- Es la forma que tenemos de guardar nuestro trabajo en la nube y no perderlo.

In case of fire 🔥



1. git commit



2. git push



3. leave building

23

- Para realizar un push, primero tenemos que tener los commit hechos en nuestro repositorio LOCAL.
- Los commit realizados en nuestro repositorio local se reflejarán en el repositorio remoto.
- Es la forma que tenemos de guardar nuestro trabajo en la nube y no perderlo.

**“Recuerda hacer pull,
antes de hacer push”**

~ Paulo Coelho



Φ www.paulocoelho.com

24

- Una práctica NECESARIA para hacer push es la de primero hacer pull.

Esto es porque si existiera algún conflicto grande, primero se resolvería en local y no se expandiría el conflicto a través del remoto.

Incluso, si se hace **fetch** primero, se podrá ver qué se ha cambiado desde la última vez que actualizamos el branch y ya poder prever si esto va a dar algún conflicto con nuestro código actual, de tal forma que haciendo **FETCH** primero, aún el cambio no se ha aplicado a nuestro “Workspace” y podemos guardarlo en un lugar seguro antes de que se apliquen los cambios.

Conceptos básicos

clone

“clonar repositorio remoto en local”



25

Mediante clonar un repositorio lo que hacemos es hacer una copia del repositorio remoto en local y ENLAZA este al repositorio remoto.

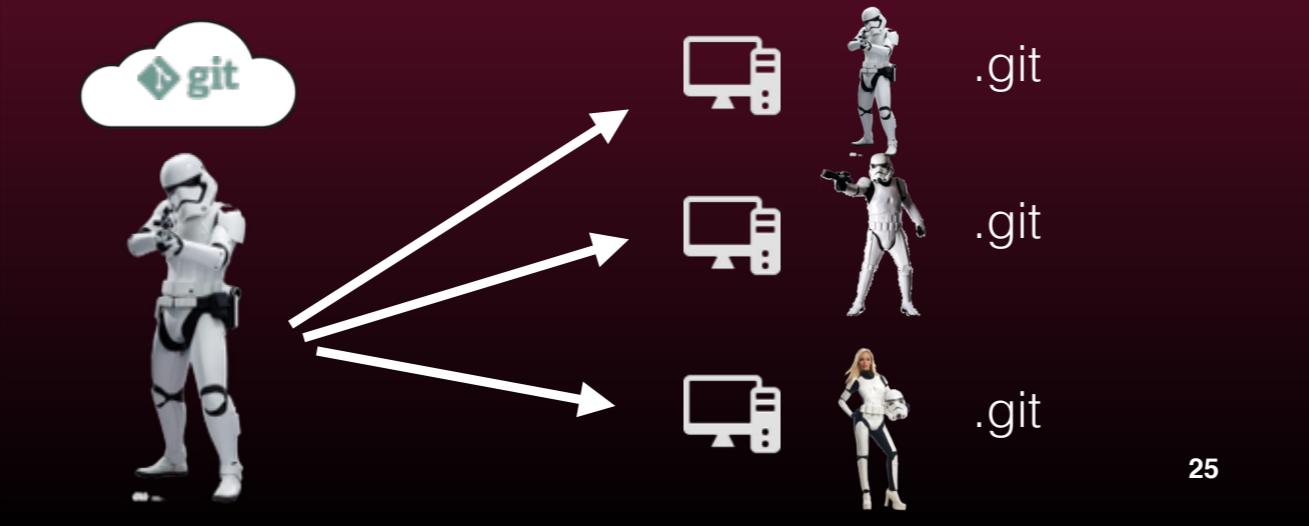
Es la forma que tenemos de empezar a trabajar en un proyecto cuyo repositorio ya ha sido creado en remoto.

Es importante clonar un repositorio ya que podríamos iniciar un repositorio vacío y enlazarlo después, pero entonces podríamos no tener exactamente el mismo proyecto ya que por ejemplo VS crea una configuración para omitir del git una serie de archivos, como por ejemplo los de compilación

Conceptos básicos

clone

“clonar repositorio remoto en local”



25

Mediante clonar un repositorio lo que hacemos es hacer una copia del repositorio remoto en local y ENLAZA este al repositorio remoto.

Es la forma que tenemos de empezar a trabajar en un proyecto cuyo repositorio ya ha sido creado en remoto.

Es importante clonar un repositorio ya que podríamos iniciar un repositorio vacío y enlazarlo después, pero entonces podríamos no tener exactamente el mismo proyecto ya que por ejemplo VS crea una configuración para omitir del git una serie de archivos, como por ejemplo los de compilación

Conceptos (**no**) básicos

- ✓ **init** : iniciar un repositorio vacío en una carpeta.
- ✓ **stage** : preparar un archivo para hacer commit.
- ✓ **rebase** : similar a merge, añade los commits en la línea sobre la que se hace rebase. NUNCA EN PÚBLICO.
- ✓ **sync** : (solo en VS) hace push + pull.
- ✓ **checkout** : seleccionar un branch y trabajar con el.
- ✓ **reset** : deshace cambios que no han sido publicados en remoto.
- ✓ **revert** : revierte un commit (pero no lo elimina).
- ✓ **pull request** : sugerir un cambio al administrador de otro branch. Alguien deberá supervisar el cambio.
- ✓ **tag** : etiqueta que se ponen a commit para documentarlos.

26

Hablar de que hay controversia entre rebase y merge, que nosotros hemos usado merge para hacer el flow más sencillo.

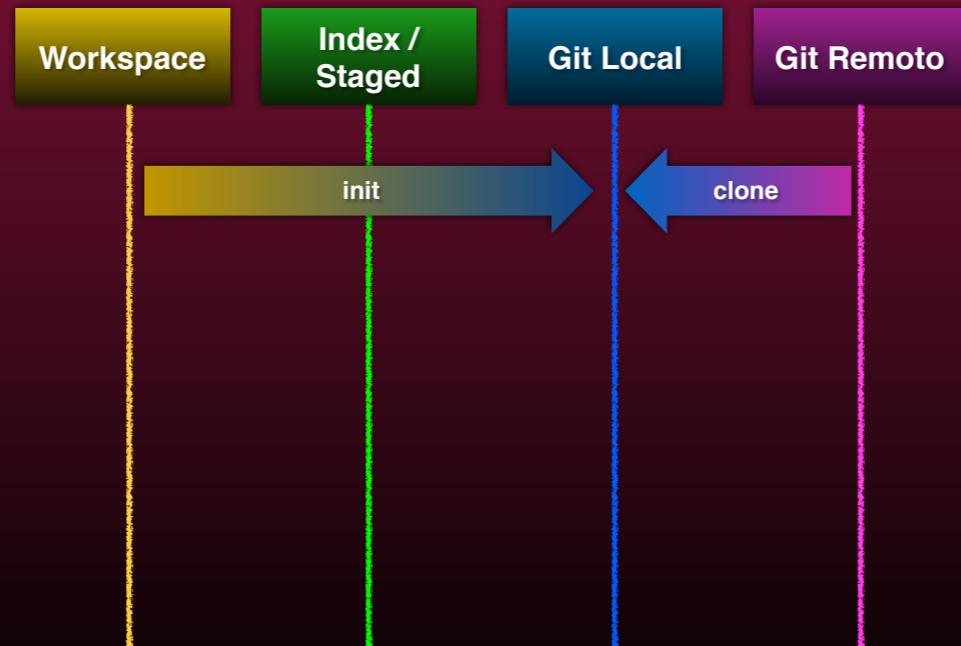
Resumen



27

- Primero los comandos para iniciar a trabajar con git.
- Segundo aquellos por los que realizamos cambios en el repositorio.
- Tercero Aquellos por los que traemos cambios del servidor.
- Checkout sirve para cambiar entre branchs.

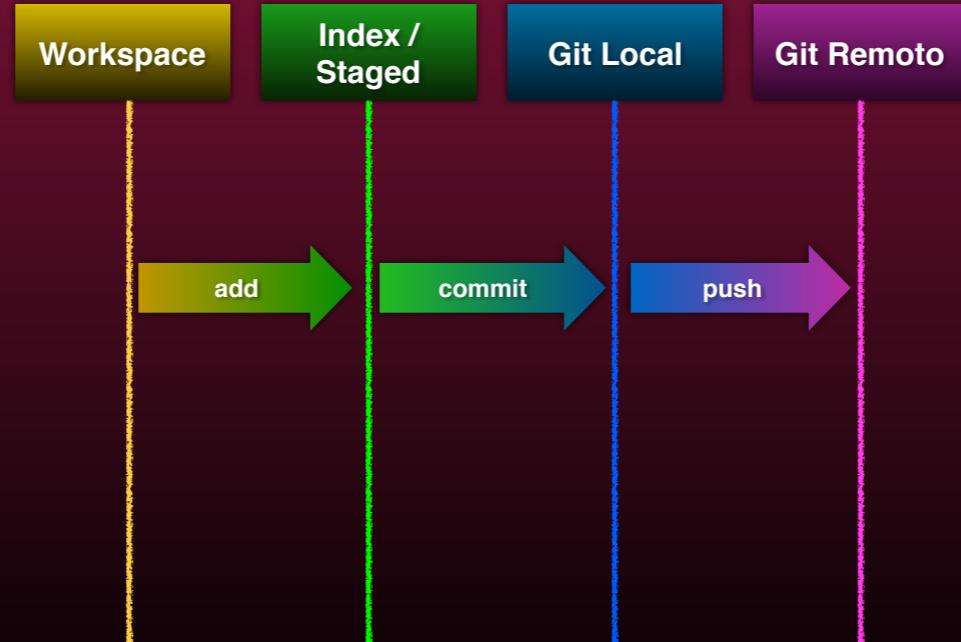
Resumen



27

- Primero los comandos para iniciar a trabajar con git.
- Segundo aquellos por los que realizamos cambios en el repositorio.
- Tercero Aquellos por los que traemos cambios del servidor.
- Checkout sirve para cambiar entre branchs.

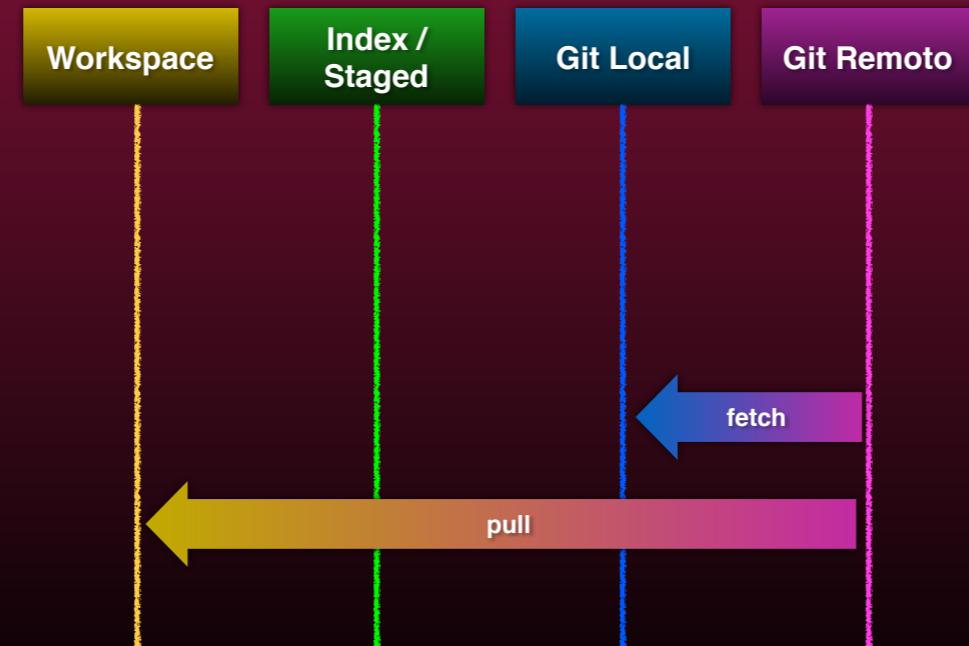
Resumen



27

- Primero los comandos para iniciar a trabajar con git.
- Segundo aquellos por los que realizamos cambios en el repositorio.
- Tercero Aquellos por los que traemos cambios del servidor.
- Checkout sirve para cambiar entre branchs.

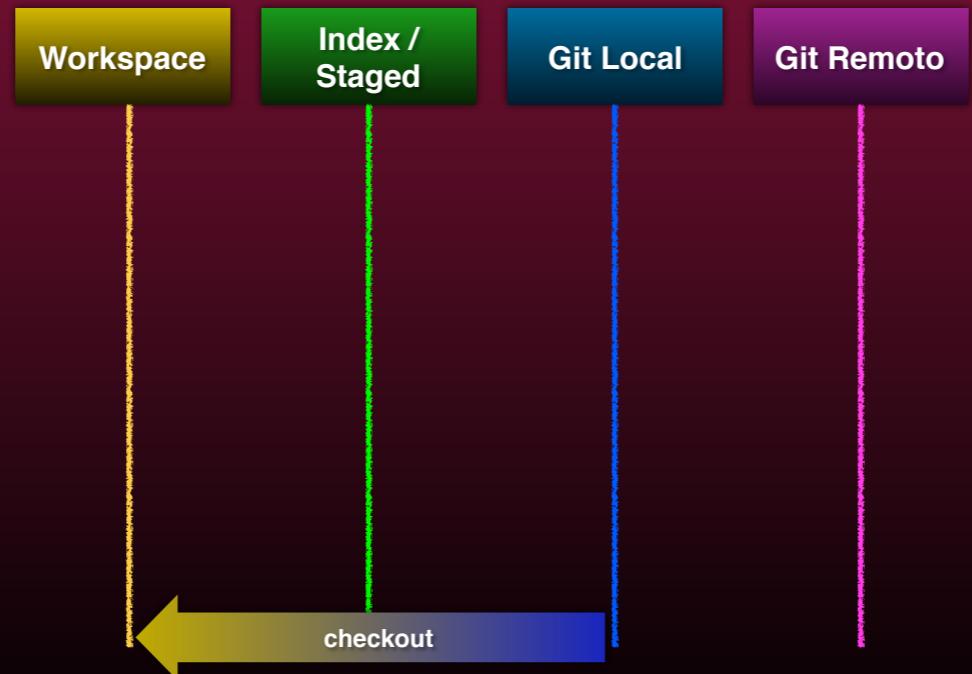
Resumen



27

- Primero los comandos para iniciar a trabajar con git.
- Segundo aquellos por los que realizamos cambios en el repositorio.
- Tercero Aquellos por los que traemos cambios del servidor.
- Checkout sirve para cambiar entre branchs.

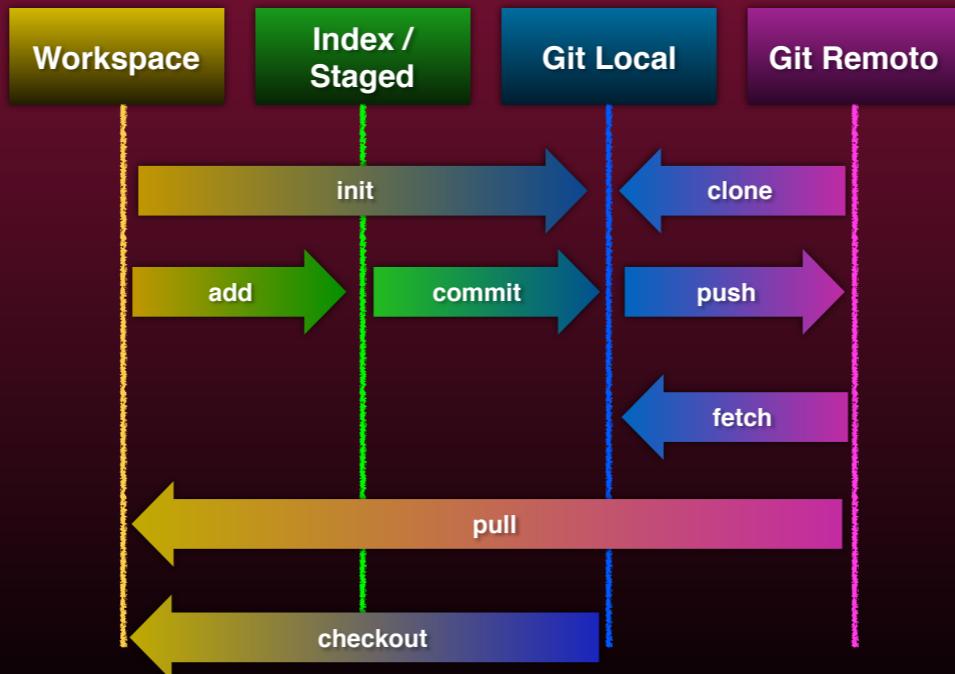
Resumen



27

- Primero los comandos para iniciar a trabajar con git.
- Segundo aquellos por los que realizamos cambios en el repositorio.
- Tercero Aquellos por los que traemos cambios del servidor.
- Checkout sirve para cambiar entre branchs.

Resumen



28

Todos otra vez para verlos en conjunto y tomar notas

Uso: empezar

Uso: empezar

- Clonar repositorio.

Uso: empezar

- Clonar repositorio.
- Crear repositorio desde proyecto existente.

Uso: empezar

- Clonar repositorio.
- Crear repositorio desde proyecto existente.
- Publicar repositorio local en remoto.

Uso: empezar

- Clonar repositorio.
- Crear repositorio desde proyecto existente.
- Publicar repositorio local en remoto.

Uso: hacer cambios

30

Uso: hacer cambios

- Commit.

Uso: hacer cambios

Commit.

Stage.

Uso: hacer cambios

- Commit.
- Stage.
- Push.

Uso: hacer cambios

- Commit.
- Stage.
- Push.

Uso: branch

Uso: branch

- Crear branch.

Uso: branch

Crear branch.

Merge.

Uso: branch

- Crear branch.
- Merge.
- Resolución conflictos.

Uso: branch

- Crear branch.
- Merge.
- Resolución conflictos.

Uso: gestión

Uso: gestión

- Git GUI.

Uso: gestión

- Git GUI.
- Visual Studio: Cambios.

Uso: gestión

- Git GUI.
- Visual Studio: Cambios.
- Source Tree.

Uso: gestión

- Git GUI.
- Visual Studio: Cambios.
- Source Tree.

Filosofía

33

Esta es la filosofía con la que usamos git, es importante tenerla clara para trabajar de forma correcta y eficiente.

Se llama branch por funcionalidad, cada funcionalidad va a tener su propio branch independiente de todas las demás.

Es necesario que la función sea independiente de las demás, puesto que no es aconsejable estar actualizando un branch (como el de desarrollo o máster) cada vez que alguna otra rama cambie.

Cuando se termine de trabajar sobre el branch, se incorpora este branch al branch de desarrollo. Se conoce como un merge, el nuevo branch muere.

Resumiendo:

- Cada característica del programa solo tendrá responsabilidad sobre sí misma, esto es que, para funcionar no necesitará recibir cambios de otras características. Esto evita que tengamos que actualizar con el develop la rama constantemente.
- Cuando se termina una característica su correspondiente branch será eliminado. Si se trabajara más adelante podría correr el peligro de trabajar sobre una versión de todo lo demás muy desactualizada, lo que daría una gran cantidad de fallos más adelante.
- Cuando se necesite arreglar un fallo de una característica, no se creará un branch nuevo ni se hará sobre el branch de la susodicha característica (en el caso que siga existiendo). Un hotfix será realizado en la release más antigua y éste se heredará por todas las releases posteriores siendo su último destino la rama de desarrollo.

Filosofía

Branch per feature:

- Cada característica solo tiene responsabilidad sobre sí misma, distinto branch por cada característica.

33

Esta es la filosofía con la que usamos git, es importante tenerla clara para trabajar de forma correcta y eficiente.

Se llama branch por funcionalidad, cada funcionalidad va a tener su propio branch independiente de todas las demás.

Es necesario que la función sea independiente de las demás, puesto que no es aconsejable estar actualizando un branch (como el de desarrollo o máster) cada vez que alguna otra rama cambie.

Cuando se termine de trabajar sobre el branch, se incorpora este branch al branch de desarrollo. Se conoce como un merge, el nuevo branch muere.

Resumiendo:

- Cada característica del programa solo tendrá responsabilidad sobre sí misma, esto es que, para funcionar no necesitará recibir cambios de otras características. Esto evita que tengamos que actualizar con el develop la rama constantemente.
- Cuando se termina una característica su correspondiente branch será eliminado. Si se trabajara más adelante podría correr el peligro de trabajar sobre una versión de todo lo demás muy desactualizada, lo que daría una gran cantidad de fallos más adelante.
- Cuando se necesite arreglar un fallo de una característica, no se creará un branch nuevo ni se hará sobre el branch de la susodicha característica (en el caso que siga existiendo). Un hotfix será realizado en la release más antigua y éste se heredará por todas las releases posteriores siendo su último destino la rama de desarrollo.

Filosofía

Branch per feature:

- Cada característica solo tiene responsabilidad sobre sí misma, distinto branch por cada característica.
- Característica terminada es branch eliminado.

33

Esta es la filosofía con la que usamos git, es importante tenerla clara para trabajar de forma correcta y eficiente.

Se llama branch por funcionalidad, cada funcionalidad va a tener su propio branch independiente de todas las demás.

Es necesario que la función sea independiente de las demás, puesto que no es aconsejable estar actualizando un branch (como el de desarrollo o máster) cada vez que alguna otra rama cambie.

Cuando se termine de trabajar sobre el branch, se incorpora este branch al branch de desarrollo. Se conoce como un merge, el nuevo branch muere.

Resumiendo:

- Cada característica del programa solo tendrá responsabilidad sobre sí misma, esto es que, para funcionar no necesitará recibir cambios de otras características. Esto evita que tengamos que actualizar con el develop la rama constantemente.
- Cuando se termina una característica su correspondiente branch será eliminado. Si se trabajara más adelante podría correr el peligro de trabajar sobre una versión de todo lo demás muy desactualizada, lo que daría una gran cantidad de fallos más adelante.
- Cuando se necesite arreglar un fallo de una característica, no se creará un branch nuevo ni se hará sobre el branch de la susodicha característica (en el caso que siga existiendo). Un hotfix será realizado en la release más antigua y éste se heredará por todas las releases posteriores siendo su último destino la rama de desarrollo.

Filosofía

Branch per feature:

- Cada característica solo tiene responsabilidad sobre sí misma, distinto branch por cada característica.
- Característica terminada es branch eliminado.
- Hotfix se hará en la release más antigua y será heredado.

33

Esta es la filosofía con la que usamos git, es importante tenerla clara para trabajar de forma correcta y eficiente.

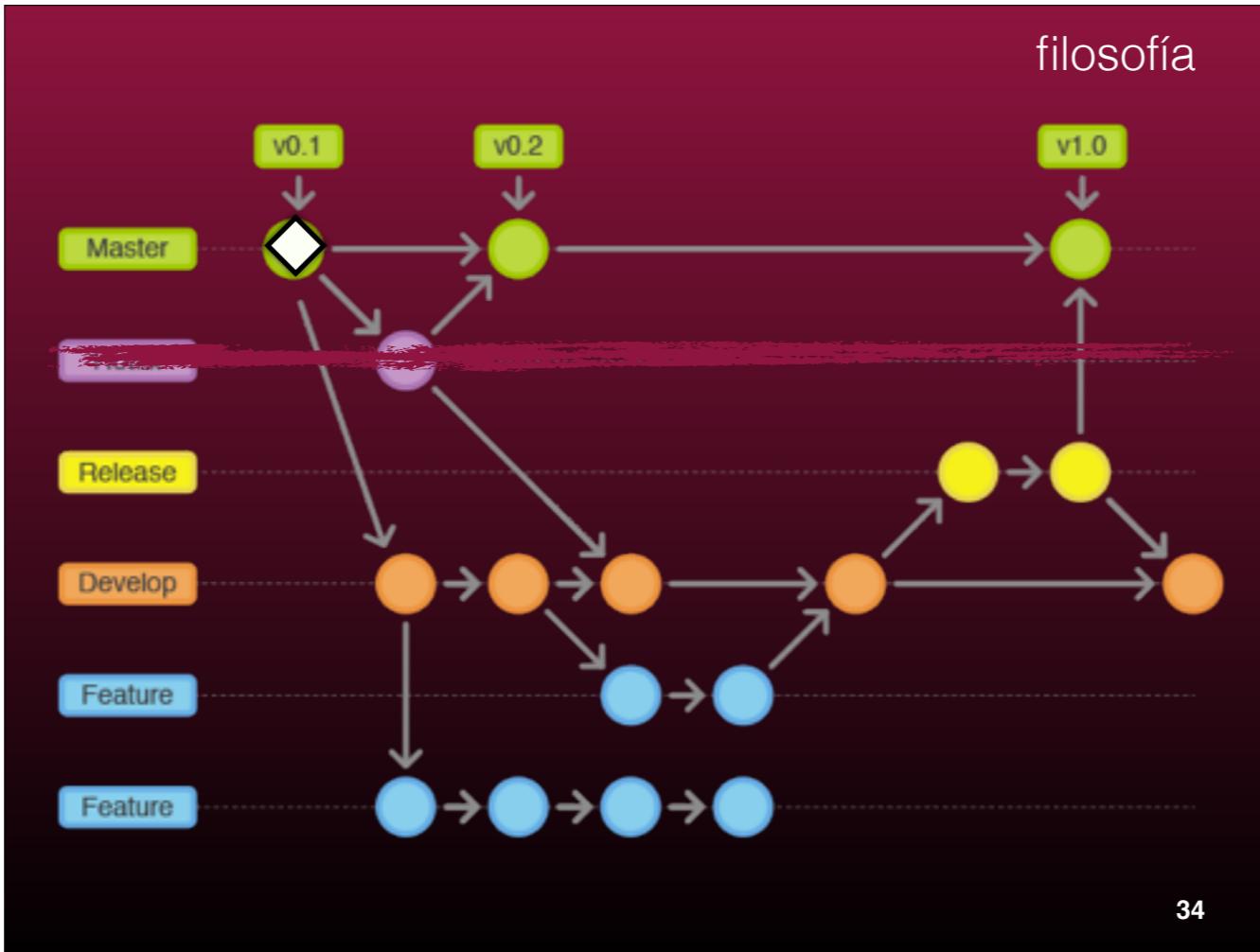
Se llama branch por funcionalidad, cada funcionalidad va a tener su propio branch independiente de todas las demás.

Es necesario que la función sea independiente de las demás, puesto que no es aconsejable estar actualizando un branch (como el de desarrollo o máster) cada vez que alguna otra rama cambie.

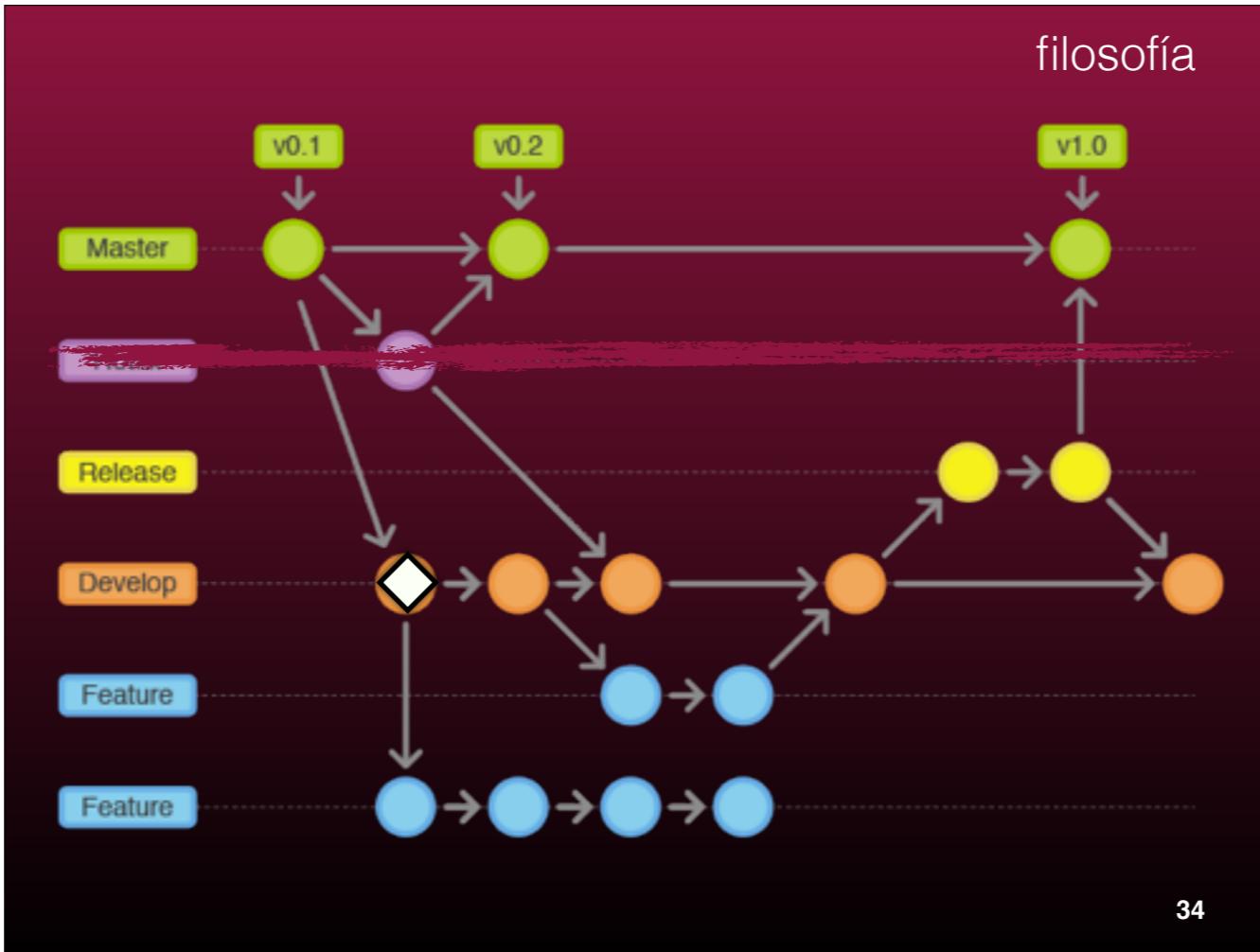
Cuando se termine de trabajar sobre el branch, se incorpora este branch al branch de desarrollo. Se conoce como un merge, el nuevo branch muere.

Resumiendo:

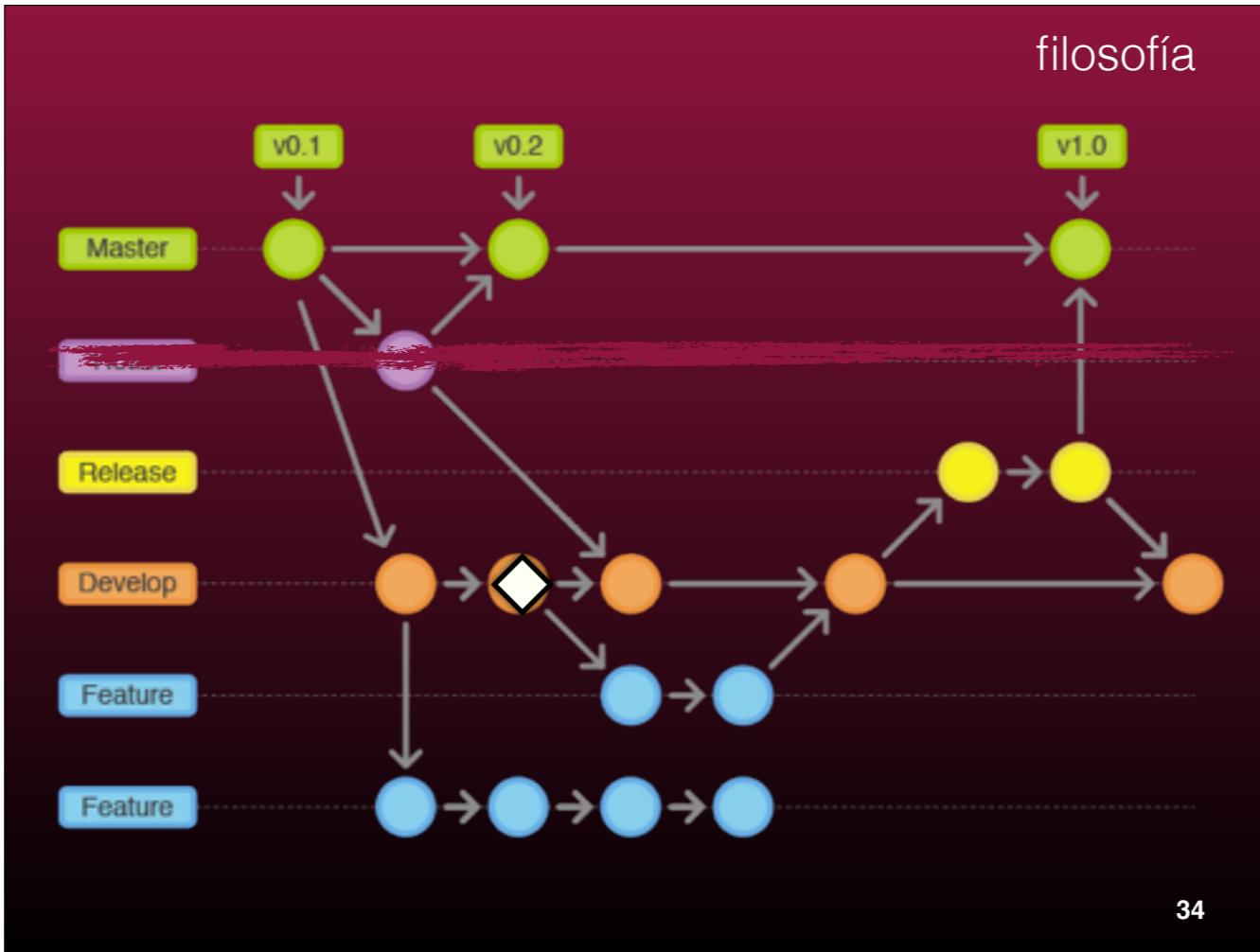
- Cada característica del programa solo tendrá responsabilidad sobre sí misma, esto es que, para funcionar no necesitará recibir cambios de otras características. Esto evita que tengamos que actualizar con el develop la rama constantemente.
- Cuando se termina una característica su correspondiente branch será eliminado. Si se trabajara más adelante podría correr el peligro de trabajar sobre una versión de todo lo demás muy desactualizada, lo que daría una gran cantidad de fallos más adelante.
- Cuando se necesite arreglar un fallo de una característica, no se creará un branch nuevo ni se hará sobre el branch de la susodicha característica (en el caso que siga existiendo). Un hotfix será realizado en la release más antigua y éste se heredará por todas las releases posteriores siendo su último destino la rama de desarrollo.



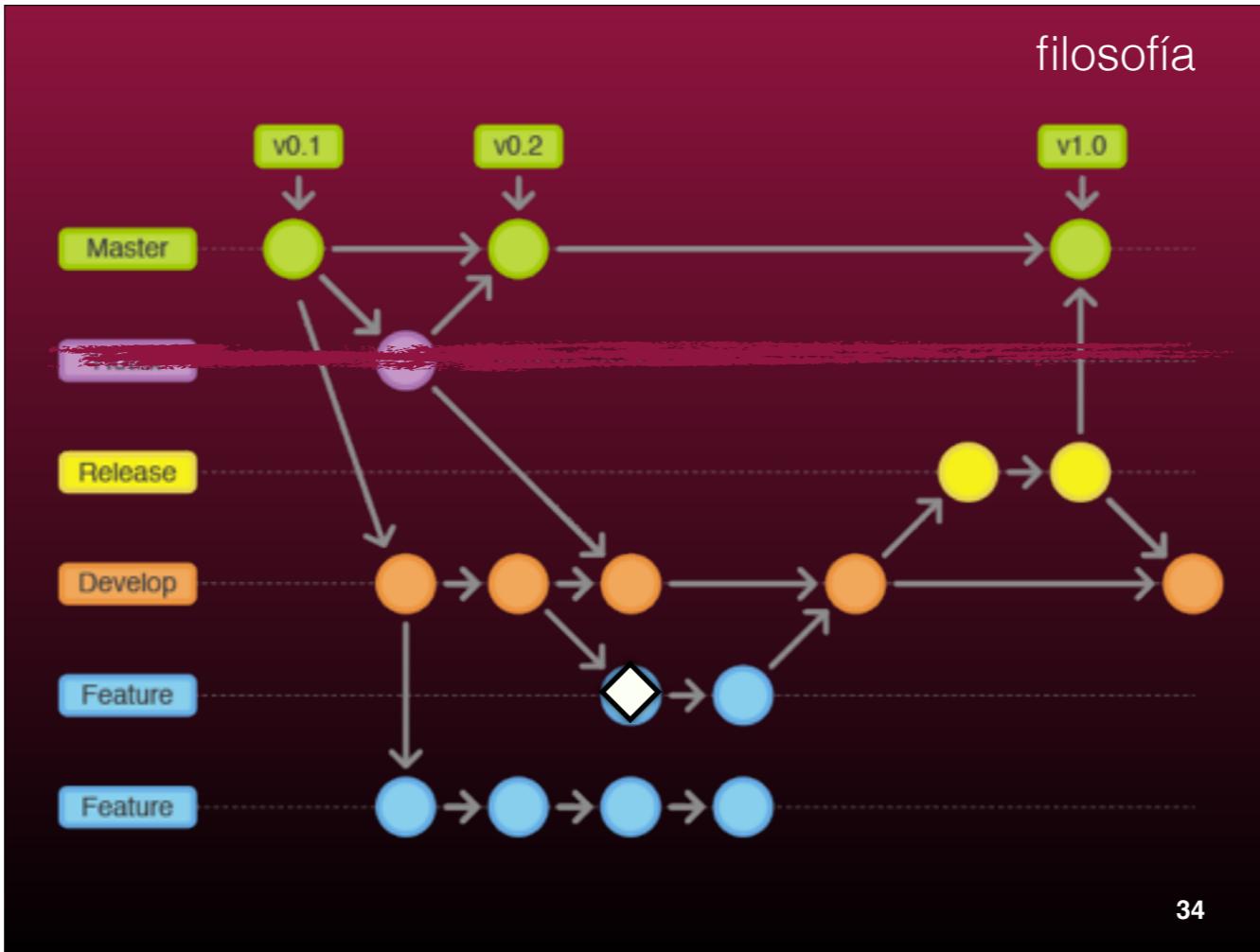
Master, hablar sobre la última versión estable y que es importante diferenciarlo del develop.



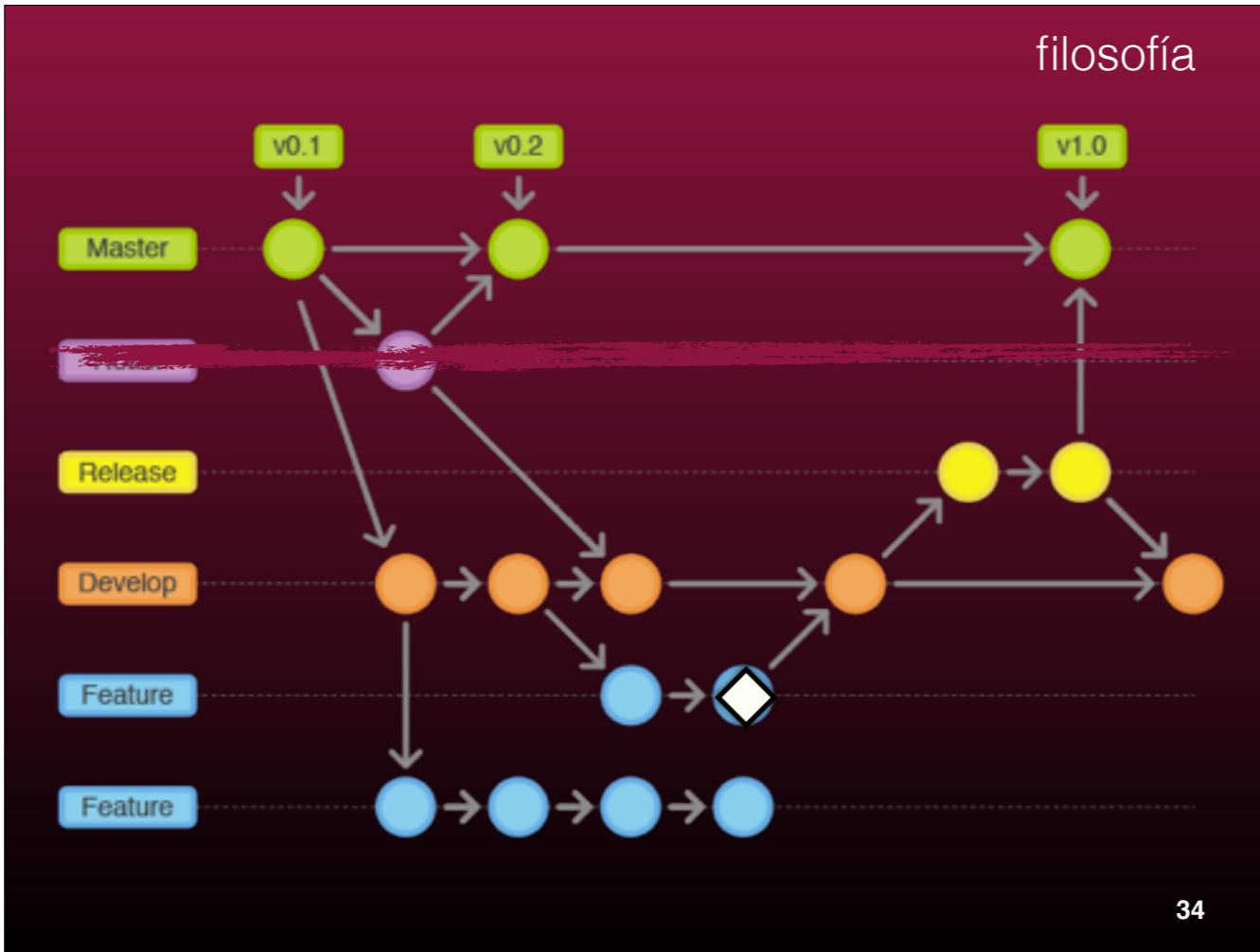
Master, hablar sobre la última versión estable y que es importante diferenciarlo del develop.



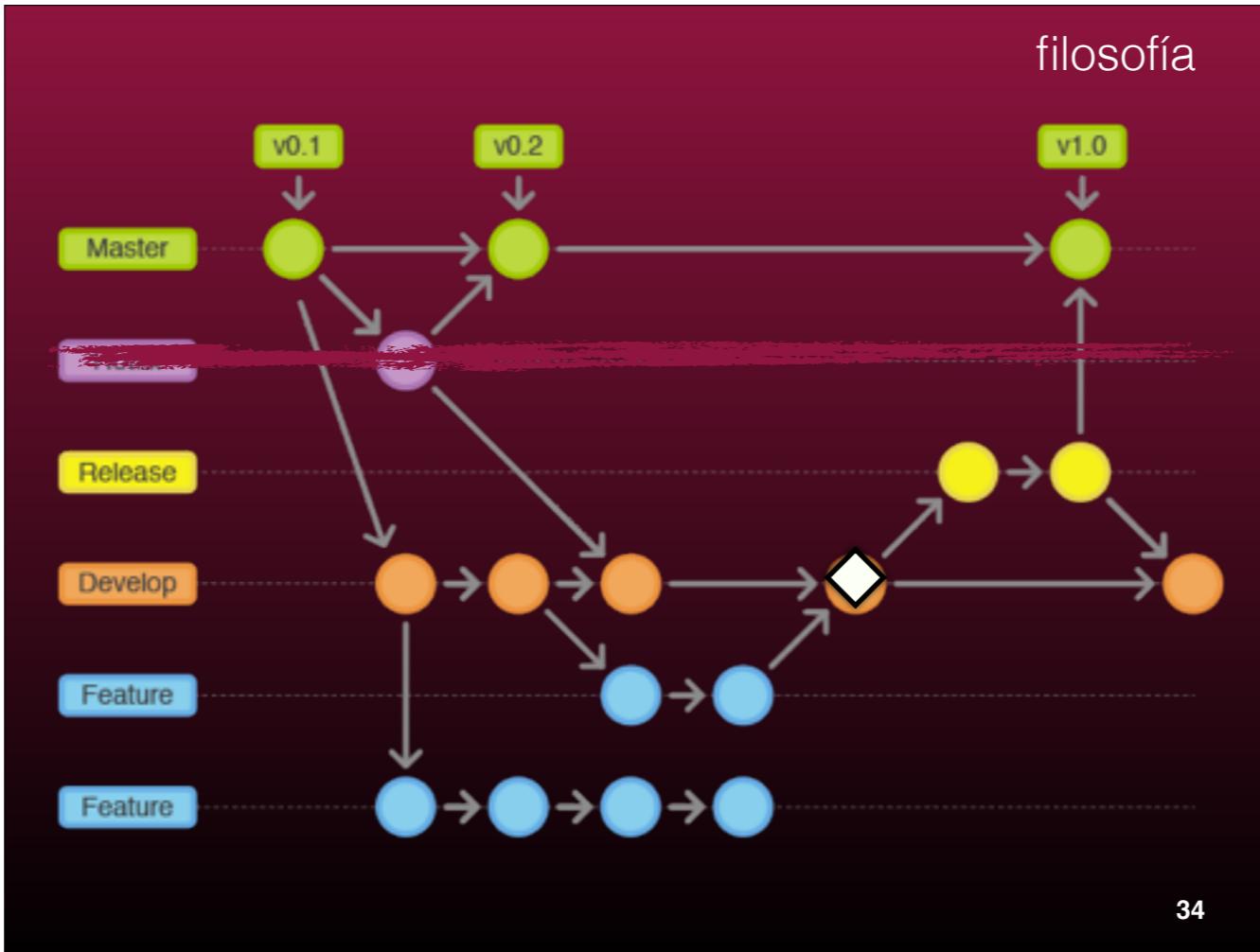
Master, hablar sobre la última versión estable y que es importante diferenciarlo del develop.



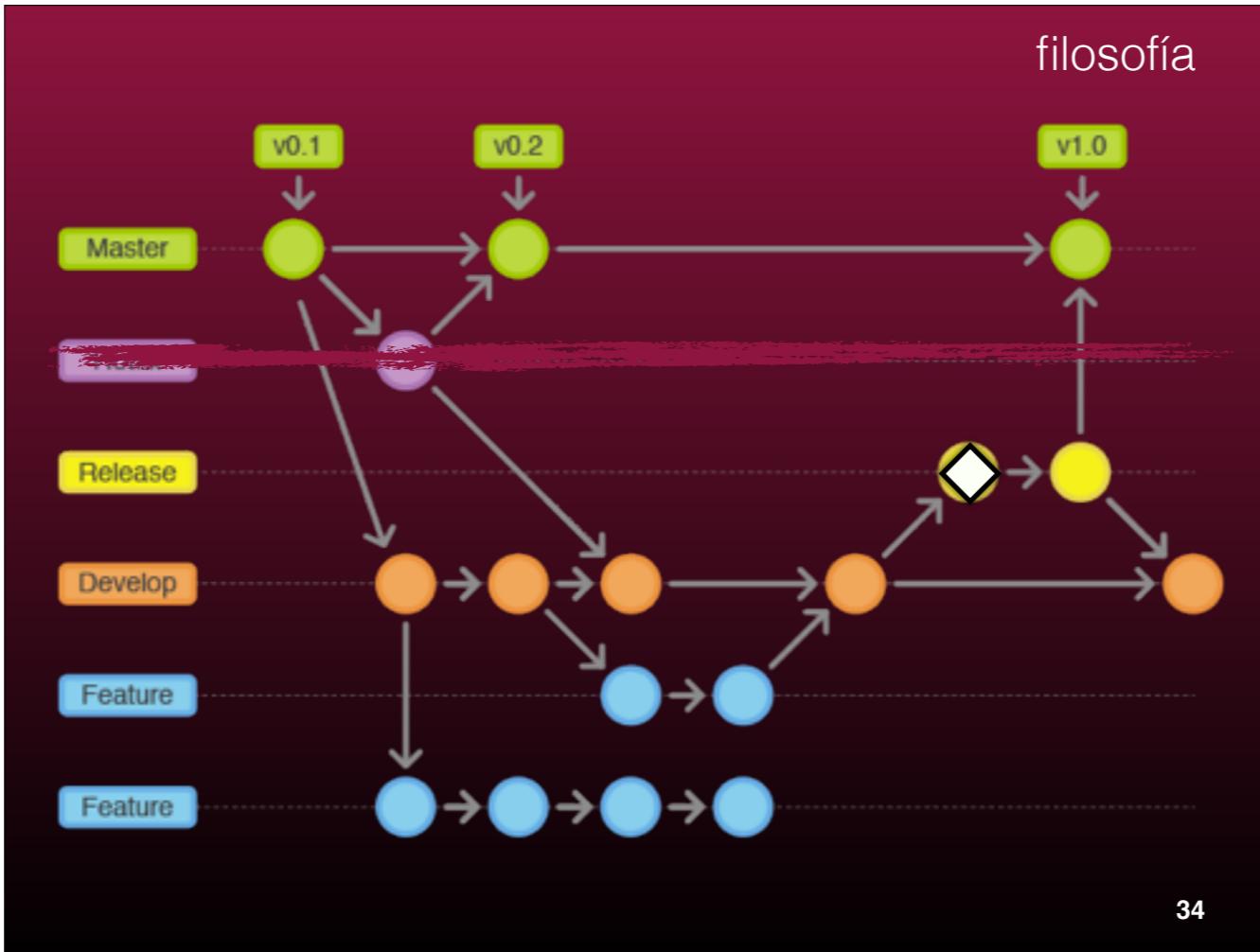
Master, hablar sobre la última versión estable y que es importante diferenciarlo del develop.



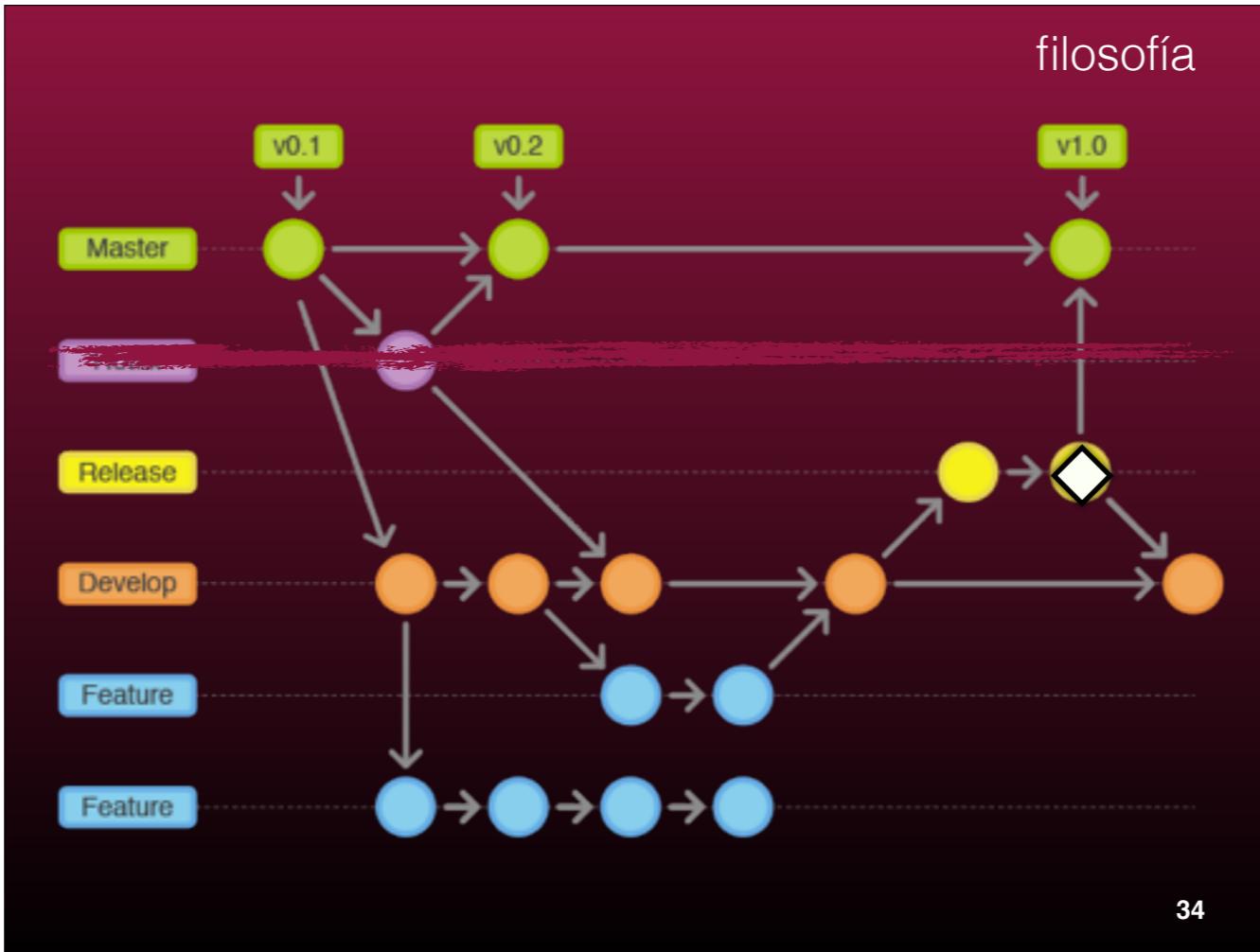
Master, hablar sobre la última versión estable y que es importante diferenciarlo del develop.



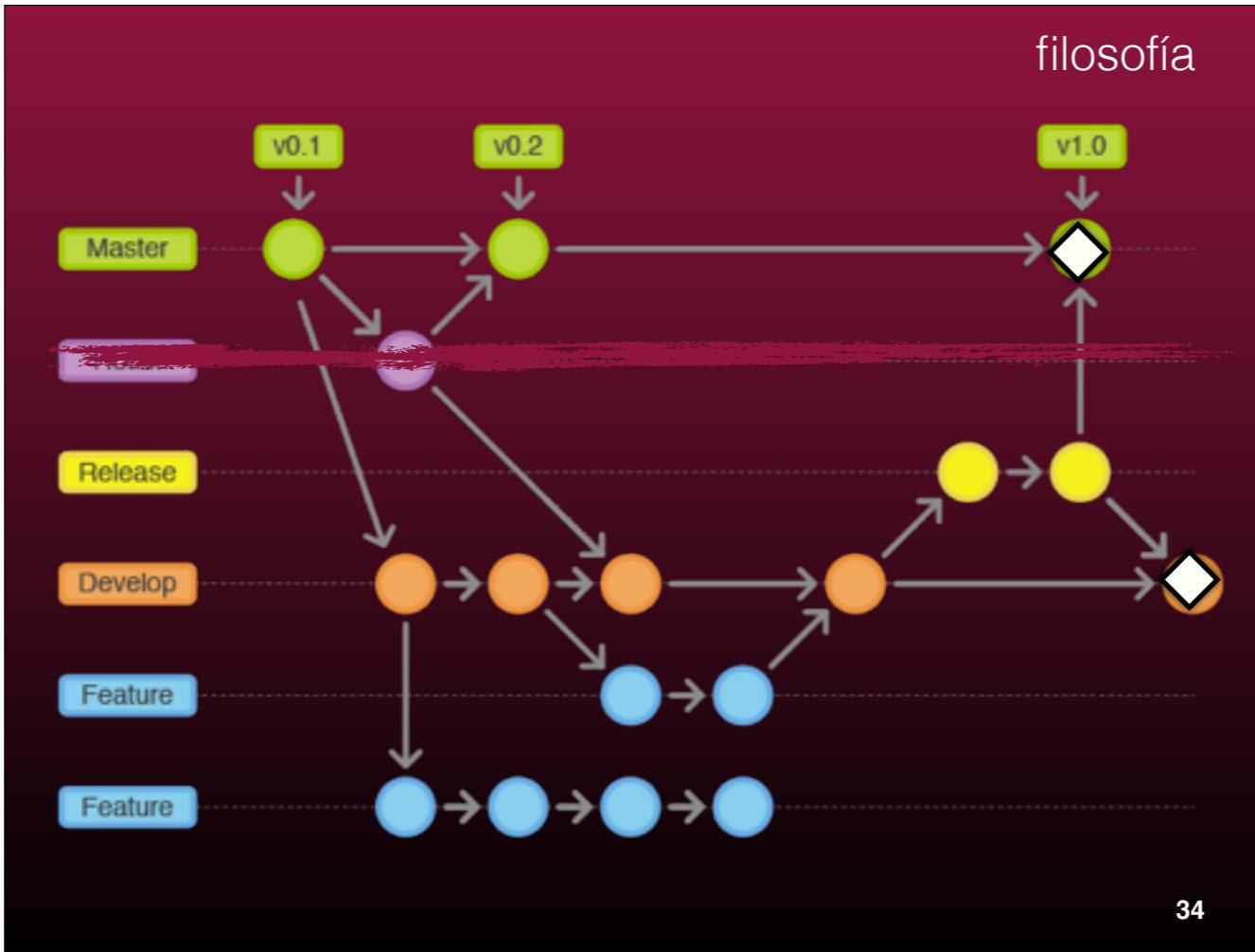
Master, hablar sobre la última versión estable y que es importante diferenciarlo del develop.



Master, hablar sobre la última versión estable y que es importante diferenciarlo del develop.



Master, hablar sobre la última versión estable y que es importante diferenciarlo del develop.



Master, hablar sobre la última versión estable y que es importante diferenciarlo del develop.