

# Blog

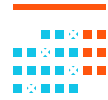
*Tecnología para Desarrollo*



Tecnología para Desarr...



Tecnología para Negocio



Eventos y seminarios

## Sácale más partido a tus proyectos con Git

por [Jorge Aguirre \(https://www.paradigmadigital.com/author/jaguirre/\)](https://www.paradigmadigital.com/author/jaguirre/)

Madrid, España

23 de octubre del 2017

Desarrollo

1 comentario

Ya hemos hablado de qué es exactamente Git, y las [ventajas que tiene](#) (<https://www.paradigmadigital.com/dev/primeros-pasos-git/>), así como las buenas prácticas para empezar a usarlo en un proyecto.

Si bien aporta un control de versiones muy bueno a un proyecto, hay veces, sobre todo en proyectos grandes, que si no nos ceñimos a las buenas prácticas de Git, podemos acabar con un versionado caótico y difícil de gestionar. Vamos a ver **algunas prácticas para poder exprimir Git al máximo**.



<https://www.paradigmadigital.com/wp-content/uploads/2017/10/portada-git.png>

Para navegar por el histórico de versiones y *commits*, y realizar las operaciones más habituales, a mí me ha parecido muy ventajoso el uso de programas con una interfaz gráfica, como pueden ser [SourceTree](https://www.sourcetreeapp.com/) (curiosamente no está disponible para Linux) o [GitKraken](https://www.gitkraken.com/) (disponible tanto en Windows, Linux, como Mac).



<https://www.paradigmadigital.com/wp-content/uploads/2017/10/1-1.png>

Las interfaces de usuario no son para todos, eso sí. Hay quien abre la consola y desde ahí ejecuta Git sin problema. [La regla del 80/20](https://es.wikipedia.org/wiki/Principio_de_Pareto) también se puede aplicar a Git, y con apenas 5 o 6 comandos de consola, se pueden hacer el 80% de las operaciones del día a día.

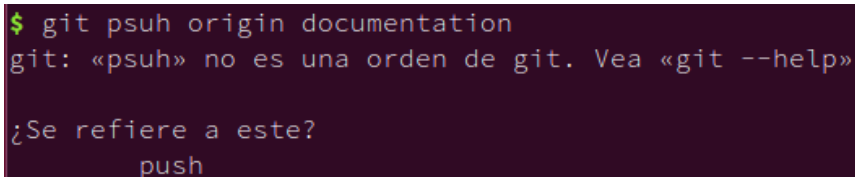
Sin embargo, tanto si eres de interfaz de usuario, como usuario de consola, hay veces que no queda más remedio que remangarse, poner las dos manos en el teclado y realizar alguna operación fuera de lo común desde la consola.

Así que presentamos aquí una serie de herramientas de Git que pueden hacer tu vida más fácil cuando toca trabajar con la consola, y además es posible que te ayuden a mantener tus *commits* y ramas más ordenados.

## Activar el autocorrector de Git

Un imprescindible si eres un manazas como yo. Intentas escribir demasiado deprisa y escribes alguna palabra mal. Evidentemente Git no lo reconoce como comando y te muestra un mensaje de error.

A veces, incluso te pregunta si no te referías a un comando casi igual al que has escrito, pero nada más, no hace ninguna operación.



```
$ git psuh origin documentation
git: «psuh» no es una orden de git. Vea «git --help»

¿Se refiere a este?
    push
```

<https://www.paradigmadigital.com/wp-content/uploads/2017/10/3.png>

Así pues, para evitar la molestia de tener que reescribir la línea entera de código, podemos usar el autocorrector. Es tan sencillo como ejecutar:

```
git config --global help.autocorrect N
```

Donde N son las décimas de segundo que ponemos de espera hasta que se active la corrección automática y ejecute el comando. Si pusiéramos 0, se desactiva la autocorrección y volveríamos al estado inicial, en que simplemente informa de que no entiende el comando (y este es, de hecho, el valor que viene por defecto).

```
17:33:31 ~
$ git config --global help.autocorrect 10

17:33:35 ~
$ git psuh
Aviso: ha llamado a la orden Git «psuh», pero no existe
Continuando asumiendo que quería decir «push»
en 1.0 segundos automáticamente...
```

<https://www.paradigmadigital.com/wp-content/uploads/2017/10/4.png>

En este caso hemos puesto un 10, por lo que se espera 10 décimas de segundo (1.0 segundos) y continúa automáticamente haciendo el push.

## Arreglar un commit con amend

Llevas un rato trabajando, has cambiado unos cuantos archivos. Terminas de desarrollar un *feature* o resolver un *bug*. Haces el *commit* con los archivos modificados y, de repente, te das cuenta de que te has dejado algo. Ya sea un test unitario, algo de documentación, CSS o cualquier otra cosa.

Con suerte, no afecta mucho, con algo menos de suerte rompes todo el *build* con ese *commit* incompleto. No pasa nada, haces otro *commit* justo después y todo arreglado.

Sin embargo, es una mala filosofía tener *commits* incoherentes, incompletos o incluso que rompan el *build*, ya que si alguien elige retroceder a algún estado anterior del proyecto, no debería encontrarse con un *commit* que haga que las cosas no funcionen como deberían.

Si en vez de hacer un nuevo *commit*, ejecutas:

```
git commit --amend --no-edit
```

Actualizas el último *commit* que hayas hecho con los cambios que hayas añadido a Git. Así lo dejas todo empaquetado en un solo *commit* y mantienes el Git limpio.

## Copiar los cambios introducidos por un commit

Esto es especialmente útil si se tiene que hacer *backporting* (modificar o parchear una versión de software más antigua que la existente), aunque hay más casos de uso.

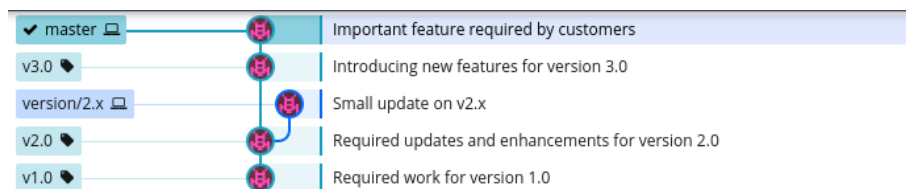
Supongamos que ya ha salido la versión 2.0 de tu producto y el equipo lleva un tiempo trabajando en la nueva release: la versión 3.0, que incorpora las nuevas funcionalidades que el mercado tanto demanda.

Pero los usuarios del producto, que al final son los que importan y a los que hay que cuidar, te piden que incorpores una de esas nuevas funcionalidades en la versión 2.1, para que la puedan ir utilizando ya.

O a lo mejor un competidor lanza esa funcionalidad al mercado, empiezas a perder usuarios y resulta imprescindible actualizar el producto ofreciendo la misma funcionalidad para retener a los usuarios actuales.

En cualquier caso, tienes que traerte una serie de *commits* de la versión 3.0 a la versión 2.0. Mi primera idea fue: "Bueno, pues con Git puedo ver los cambios que ha introducido cada *commit*, los voy replicando con cuidado en la versión a actualizar".

Efectivamente, puedes hacer eso, pero también, gracias al comando **cherry-pick** de Git, puedes hacer de forma inmediata todo este proceso.

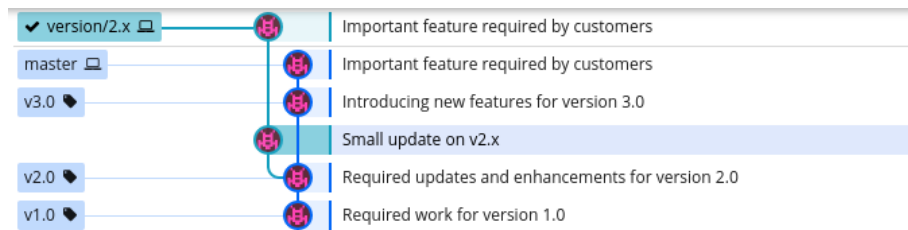


Estado del proyecto antes de hacer cherry-pick

Para hacer un cherry-pick básico hay que hacer checkout a la rama en que queremos aplicar los cambios, y ejecutar:

```
git cherry-pick <commit-hash>
```

En este caso, vamos a migrar el último *commit* de la rama master a la rama 2.x. En el repositorio aparecerá un nuevo *commit* en la rama a la que hayamos hecho el *cherry-pick* (en nuestro caso, la rama 2.x)



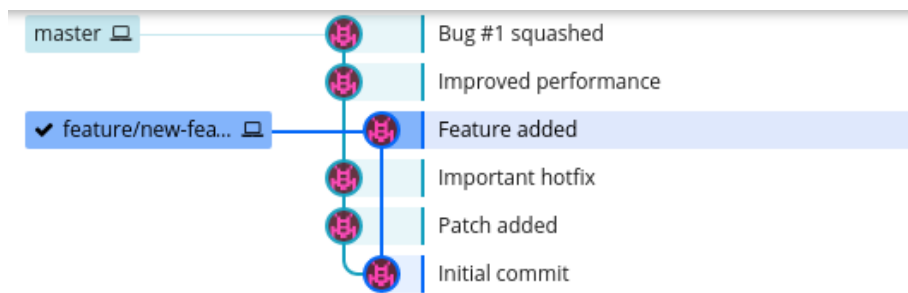
Estado del proyecto tras acabar el cherry-pick

Puedes ver más detalles de esta opción en el [manual de Git \(https://git-scm.com/docs/git-cherry-pick\)](https://git-scm.com/docs/git-cherry-pick), donde además se explican detalladamente todos los flags y opciones que tiene el comando. Desde **GitKraken** y **SourceTree** existe la opción de hacer *cherry-pick*; aunque, por simplicidad, ofrece menos opciones de las que el comando realmente permite.

## Mergear (adecuadamente) las ramas con git-rebase

Estás trabajando en un desarrollo tranquilamente, en tu propia rama local, sin molestar a nadie. Por las circunstancias que sean, ha pasado algo de tiempo desde que abriste la rama hasta que la terminas.

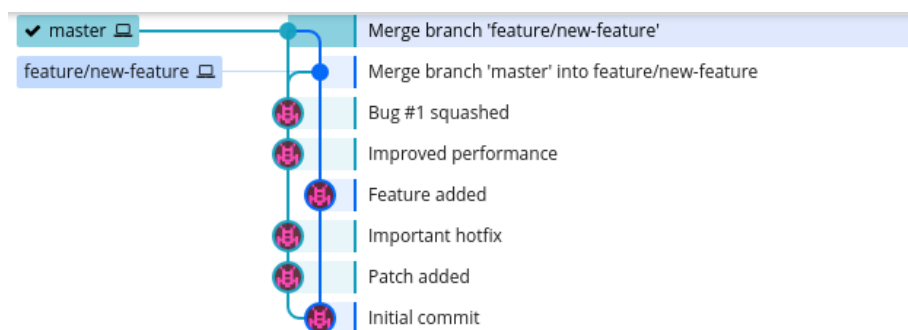
Puede que el desarrollo fuese más largo de lo esperado, o que tuvieras que atender una incidencia urgente, o simplemente que te fueras de vacaciones unos días. Por fin la terminas, estás listo para juntarla con la rama principal de desarrollo y, de repente, ves que te has quedado varios *commits* por detrás.



(<https://www.paradigmadigital.com/wp-content/uploads/2017/10/7.png>)

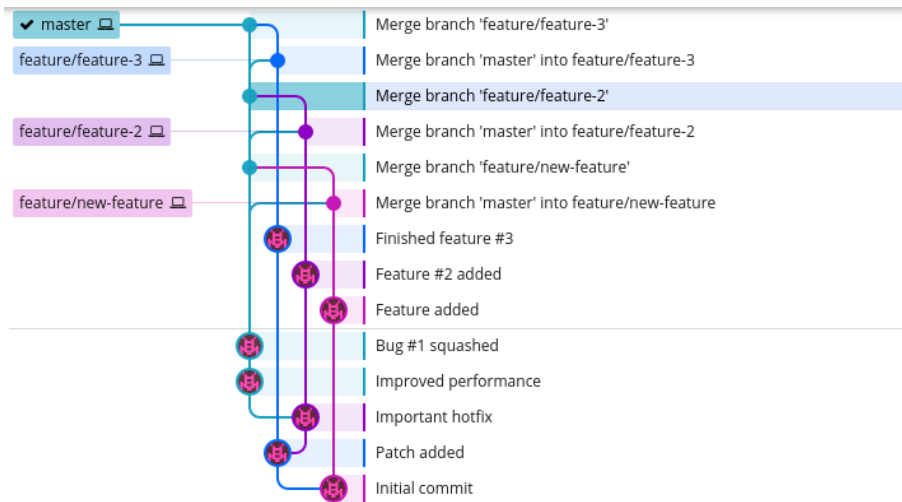
Para realizar el merge se nos presenta la posibilidad de hacer primero un pull de la rama principal a nuestra rama, para luego crear el Merge Request a la rama principal.

Con el *pull* inicial se actualizan todos los ficheros de nuestra rama (y si hay conflictos, aparecerán aquí) y ya. La solicitud de integración que se crea es con nuestra rama actualizada y los posibles conflictos resueltos. Todo en orden.



(<https://www.paradigmadigital.com/wp-content/uploads/2017/10/8.png>)

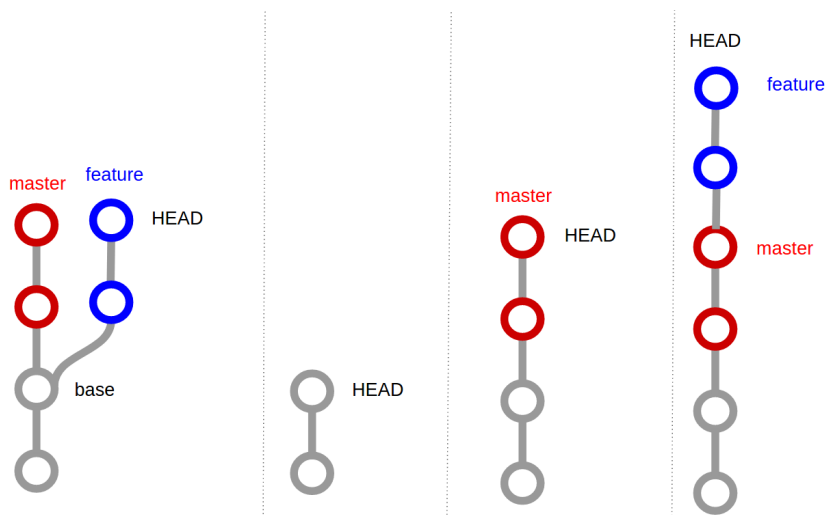
¿El problema? Que las ramas se van entrelazando. Con una sola rama no parece muy grave, pero si hay varios desarrollos en paralelo, la relación entre las ramas se va complicando y el flujo de desarrollo se va haciendo cada vez más complejo de seguir.



(<https://www.paradigmadigital.com/wp-content/uploads/2017/10/9.png>)

Y entonces llega [Git Rebase](https://git-scm.com/docs/git-rebase) (<https://git-scm.com/docs/git-rebase>). Este sencillo comando nos permite mover nuestros *commits*. La rama en la que empezamos a trabajar, partía de un *commit* "base" de la rama principal. En su momento era el *commit* más reciente, pero ya no lo es.

Al ejecutar el *rebase*, Git moverá los *commits* de nuestra rama al *commit* que le indiquemos de la rama principal. Así, cambiamos la "base" de la que partía. Lo que hace Git es lo siguiente:

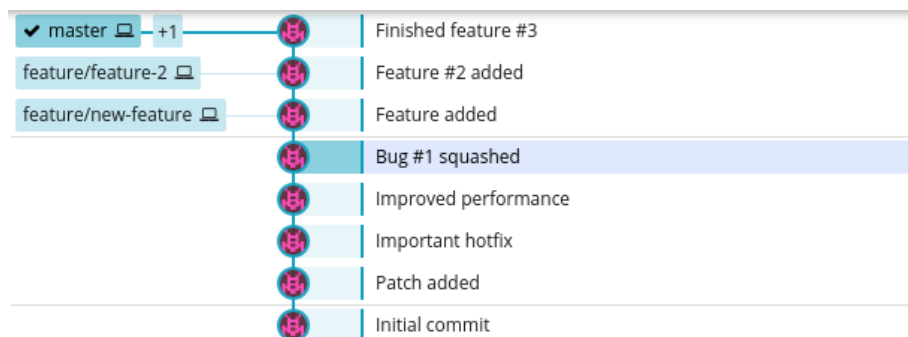


(<https://www.paradigmadigital.com/wp-content/uploads/2017/10/10.png>)

Si surgen conflictos durante el *rebase*, Git te ofrece la posibilidad de solucionarlos y continuar, o bien abortar el proceso de *rebase* y dejar los *commits* como estaban antes.



Tras acabar el *rebase*, solucionando los conflictos que pudieran surgir, la rama estará en la posición adecuada para hacer el Merge Request directamente. El resultado final es el mismo, el código de nuestra rama ha quedado correctamente mergeado con la rama principal, pero el histórico de Git queda mucho más claro y fácil de seguir.



(<https://www.paradigmadigital.com/wp-content/uploads/2017/10/11.png>)

## Encontrar una aguja en un pajar con git bisect

Esta utilidad de Git es, quizás, la que menos se utilice en el día a día, pero puede ahorrarnos mucho tiempo e investigación detectivesca.

Podemos utilizar esta herramienta cuando tenemos una incidencia en el proyecto y no sabemos a ciencia cierta cuándo se introdujo el error, y no vemos de manera inmediata el origen del fallo.

Supongamos, por ejemplo, que nos llega una incidencia con el proyecto en el que estamos trabajando. Por ejemplo, un endpoint de nuestra API está devolviendo un error. La API ha funcionado en algún momento, porque si no, no se hubiese desplegado y el endpoint no estaría activo.

Así que, en algún momento, alguien ha cambiado alguna cosa del código y ha empezado a dar el error. Claro que, desde ese momento, pueden haber pasado varias decenas de *commits*. Encontrar qué *commit* introdujo el error puede ser especialmente tedioso, sobre todo en proyectos grandes.



<https://www.paradigmadigital.com/wp-content/uploads/2017/10/12.png>

Ahí es donde entra [Git Bisect \(https://git-scm.com/docs/git-bisect\)](https://git-scm.com/docs/git-bisect), que nos va a permitir localizar de forma más sencilla el *commit* que introdujo el error. Para empezar, hay que indicarle 1 *commit* que sepamos a ciencia cierta que funcione correctamente (un "good" *commit*), así como el *commit* en que estamos viendo que funciona erróneamente (un "bad" *commit*). Git seleccionará un commit situado entre ambos *commits* (el "good" y el "bad") haciendo una bisección (de ahí el nombre).

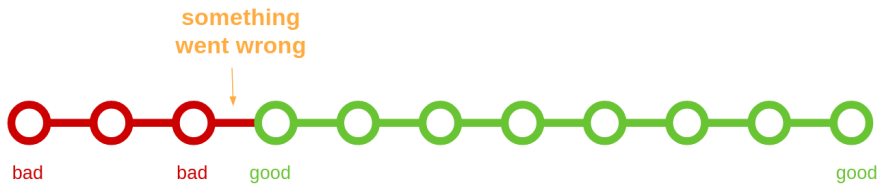


Paso inicial: git-bisect selecciona el commit intermedio, y nos pregunta si ese commit es correcto, o no.

Tras seleccionarlo, nos preguntará si ese *commit* es correcto o incorrecto y realizará, con esta nueva información, una nueva bisección. El proceso se repetirá hasta encontrar el *commit* que pasa de "good" a "bad", con lo que el proceso finaliza y solo tendremos que revisar un *commit* a ver qué ha causado el error.



Siguientes pasos: Git va preguntándote por otros commits, acotando el commit conflictivo.



Resultado: Encontramos el commit en el que se tuercen las cosas

En la consola de comandos, el proceso completo sería así (he utilizado un repositorio de prueba con 7 *commits* en el que el error se introdujo en el 5º *commit*):

```
17:43:43 ~ /workspace/git-test master ✓
$ git bisect start

17:43:49 ~ /workspace/git-test master ✓
$ git bisect bad HEAD

17:43:53 ~ /workspace/git-test master ✓
$ git bisect good 935fa227928887657ca0769c077dedbbca476361
Bisecting: 2 revisions left to test after this (roughly 2 steps)
[1f3cf0415472420fa06fdfada0aa518e724ea6ba] Version 4

17:44:00 ~ /workspace/git-test 1f3cf04 ✓
$ git bisect good
Bisecting: 0 revisions left to test after this (roughly 1 step)
[156ccaeba8fa7969de4209f5c2a4455814cc4b81] Version 6

17:44:25 ~ /workspace/git-test 156ccae ✓
$ git bisect bad
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[4539236543bb764157ae502e24827f765a66342b] Version 5

17:44:40 ~ /workspace/git-test 4539236 ✓
$ git bisect bad
4539236543bb764157ae502e24827f765a66342b is the first bad commit
commit 4539236543bb764157ae502e24827f765a66342b
```

<https://www.paradigmadigital.com/wp-content/uploads/2017/10/16.png>

## Conclusiones (y advertencia)

Git es una herramienta muy completa para el versionado del código y en el día a día apenas arañamos la superficie de sus posibilidades. Gracias a comandos como los que he detallado (y alguno que sin duda, habré olvidado o desconozca) **se puede tener un control mayor sobre el estado del proyecto y las versiones y ramas que pueda tener.**

Además, es muy importante adoptar una serie de buenas prácticas con el control de versiones. Esto se debe a que Git suele ser parte de un proceso mayor (normalmente, los *commits* van acompañados de un proceso de integración continua, en el que se ejecuta una batería de tests) y también porque, para corregir errores y evitar regresiones, es necesario tener el histórico de Git ordenado.



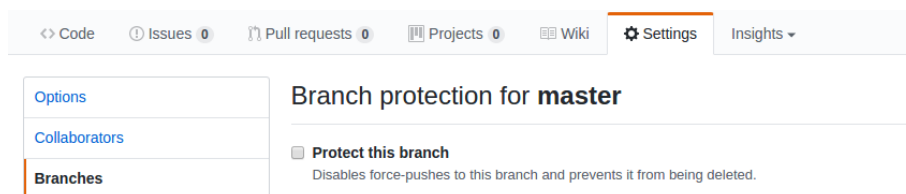
Hilando con la parte de integración continua llega una advertencia. Al modificar los datos de los *commits* (bien a través de un *amend*, o bien a través de un *rebase*), hacemos lo que se llama "reescribir la historia".

Se llama así porque modificamos el histórico de Git, moviendo *commits* de un lado a otro, modificando la información de los mismos, o incluso borrándolos.

Si los *commits* que reescribimos no se habían subido a un remoto, entonces no hay problema, ya que solo afecta a información de nuestro equipo.

Sin embargo, si modificamos la historia de *commits* que ya están disponibles en remoto, podemos ocasionar problemas a un compañero que estuviera trabajando también con nuestro proyecto, o afectar a los procesos de integración continua y despliegues, ya que estos podrían tener referencias a *commits* inexistentes.

Esto es lo suficientemente importante para que las empresas de hosting de repositorios nos ofrezcan la posibilidad de proteger nuestras ramas frente a reescrituras.



Protección de ramas en GitHub, al evitar force-push eliminamos la posibilidad de borrar commits

#### Branch permissions

Add a branch permission to specify write and merge access to a specific branch. [Learn more](#)

Branch	Access type	Users and groups	Custom settings
🔓 master	Write access	 <a href="#">Jorge Aguirre</a>	Deleting this branch is not allowed
	Merge via pull request	Everybody	
🔓 production	Write access	 <a href="#">Jorge Aguirre</a>	Deleting this branch is not allowed
	Merge via pull request	<small>*includes users and groups with write access</small>	Rewriting branch history is not allowed

Protección de ramas en Bitbucket, se indica que no se permite reescribir la historia de una rama

Sin duda, **Git ofrece unas herramientas que, aunque seguramente no usemos en el día a día, más de una vez pueden resultar ventajosas en el ciclo de vida de un proyecto de software.**

Claro que, un uso inadecuado de las mismas puede causar problemas, así que, antes de ponerte manos a la obra, recuerda que **un gran poder conlleva una gran responsabilidad.**

Compartir en Twitter

Share in LinkedIn



## Jorge Aguirre

Elegí ser desarrollador de software porque, por lo visto, superhéroe no es una trayectoria profesional válida. Soy un apasionado de las nuevas tecnologías en desarrollo y el potencial que presentan para cambiar la sociedad tal y como la conocemos. Actualmente formo parte del equipo de desarrollo de Paradigma Digital.

[Ver toda la actividad de Jorge Aguirre](#)

# Comentarios



**Miguel Sesma**

23 octubre, 2017 a las 13:18

Por lo del autocorrector tienes una caña pagada :)

Responder

## ESCRIBE UN COMENTARIO

Nombre \*

Mail (no será publicado) \*

Página web

Comentario:

Enviar comentario

En Twitter ○ ○ ○ ○ ○

**@paradigmate**

**#JHipster** (<https://twitter.com/search?q=%23JHipster&src=hash>) es un generador de

código que nos permite, a través de Yeoman, generar una aplicación spring-boot con un...

<https://t.co/xzkpp6GVe6> (<https://t.co/xzkpp6GVe6>)

En nuestro blog

**Spring Cloud Consul (2/2): Configuración centralizada**

En la primera parte de Spring Cloud Consul vimos qué es Consul, cómo se puede utilizar para descubrir microservicios y cómo montar un sistema entre Consul y ...

{ paradigma

91 352 59 42

Vía de las dos Castillas, 33 - Ática 4,  
2ª Planta 28224

Pozuelo de Alarcón (Madrid)

Copyright Paradigma Digital 2017

[contacto](#)   [download vcard](#)

[legal](#)

newsletter

Suscribirse

*bimestral*

*acepto [términos](#)*

*[legales](#)*