Git flow: una comparación con las estrategias desarrolladas por GitHub y GitLab

Hernández Rostrán Kevin, *Ingeniería en Computación, Instituto Tecnológico de Costa Rica*

***Resumen*—El sistema de control distribuido de versiones Git se ha convertido en una herramienta indispensable para manejar proyectos de software. Una de las mejores partes del desarrollo de software es la libertad que se tiene para explorar ideas y participar de manera colaborativa en proyectos de Código Abierto. Incursionar en estos proyectos es una excelente manera de evaluar y reflexionar sobre la experiencia y los métodos que tienen los desarrolladores para hacer las cosas. Últimamente, con las modernas prácticas de desarrollo como la Integración Continua, se busca automatizar estos procesos, pero para lograr este objetivo es necesario encontrar una estrategia de trabajo colaborativo. Este artículo se centra en motivar el uso de estas estrategias, explicar su implementación y compararlas.**

Palabras clave*—*Git, GitHub, GitLab, Integración Continua, Entrega Continua, Despliegue Continuo.

I. Introducción

Recientemente, he dedicado tiempo a estudiar una buena forma de administrar proyectos de software con Git. Esto debido al acercamiento que he tenido al colaborar en diferentes proyectos de Código Abierto (*Open Source*), del trabajo y personales. El control de versiones con Git hace que la ramificación (*branching*) y la fusión (*merging*) sean mucho más sencillas, y a su vez permite una amplia variedad de estrategias de ramificación y flujos de trabajo (*workflows*). Es cierto que no hay un flujo de trabajo milagroso que todos deberían seguir, ya que todos los modelos son subóptimos. Este artículo describe el flujo de trabajo con la metodología *Git flow* y dos estrategias que se pueden adoptar dependiendo del proceso de desarrollo (*development*) y despliegue (*deployment*).

II. Prácticas modernas de desarrollo

Para entender los patrones de uso de control de versiones, es importante conocer sobre las prácticas modernas de desarrollo [1], [2].

A. Integración Continua (IC)

La Integración Continua (*Continuous Integration*) es una práctica de desarrollo de software que requiere que los desarrolladores integren código en un repositorio compartido varias veces al día. Luego, cada cambio se verifica mediante una compilación automática, lo que permite que los equipos detecten problemas con anticipación, sin esperar al final del proyecto o del sprint para fusionar (*merge*) el trabajo de todos los desarrolladores.

Como lo señalan los colaboradores de ThoughtWorks [3], la integración continua involucra el siguiente proceso:

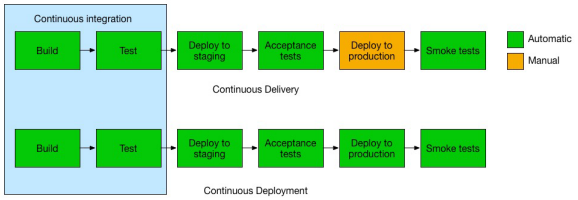

Fig. 1. Sten Pittet, *Tres prácticas modernas de desarrollo* [1]

- Los desarrolladores revisan el código en sus espacios de trabajo privados.
- Cuando terminan, envían los cambios al repositorio.
- El servidor de IC monitorea el repositorio y verifica los cambios que ocurren.
- El servidor de IC compila el sistema y ejecuta pruebas unitarias y de integración.
- El servidor de IC libera artefactos (código fuente compilado) desplegables para probar.
- El servidor de IC asigna una etiqueta de compilación a la versión del código que acaba de crear.
- El servidor de IC informa al equipo de la compilación exitosa.
- Si la compilación o las pruebas fallan, el servidor de IC alerta al equipo.
- El equipo soluciona el problema lo antes posible.
- Se repite el proceso continuamente a lo largo del proyecto.

B. Entrega Continua (EC)

La Entrega Continua (*Continuous Delivery*) refiere al lanzamiento (*release*) a producción, de software que pasa las pruebas automatizadas. Es una metodología de desarrollo de software donde el proceso de lanzamiento (*realease process*) es automático. Todos los cambios de software se crean, prueban y despliegan (*deploy*) automáticamente en producción. Antes del envío (*push*) final a producción, una persona, una prueba automatizada o una regla del negocio[1] (*business rule*) decide cuándo debe producirse el envío final. Aunque cada cambio de software exitoso se puede lanzar (*release*) inmediatamente a producción con entrega continua, no todos los cambios deben ser liberados de inmediato.

[1]Describen las políticas, normas, operaciones, definiciones y restricciones presentes en una organización y que son de vital importancia para alcanzar sus objetivos.

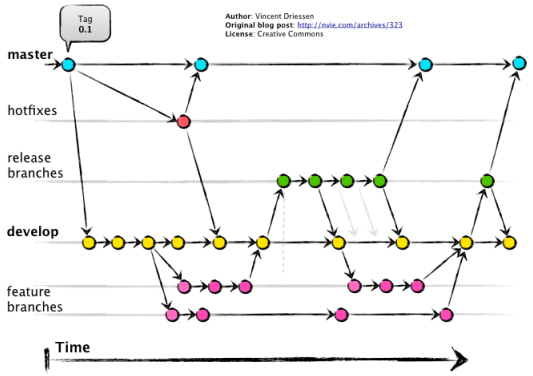

Fig. 2. Vincent Driessen, *Git flow*, 2010 [4]

C. Despliegue Continuo (DC)

El Despliegue Continuo (*Continuous Deployment*) es la Entrega Continua, pero sin la intervención humana. Estas dos prácticas son similares, pero difieren en sus enfoques de puesta (*deploy*) en producción. En el despliegue continuo, la puesta en producción se realiza automáticamente para cada cambio que pasa el conjunto de pruebas automatizadas.

III. Principales estrategias

A. Git flow

El modelo de Git flow fue dado a conocer en 2010 por Vincent Driessen en su famoso artículo "*A successful Git branching model*", acompañado de algunas extensiones de Git para gestionar ese flujo [4]. La idea general detrás de Git Flow es tener varias ramas separadas que siempre existen, cada una para un propósito diferente: *master*, *develop*, *feature*, *release* y *hotfix*. El proceso de desarrollo de funciones o errores fluye de una rama a otra antes de que finalmente se lance a producción.

Se tienen dos ramas principales:

- La rama **master** contiene la última versión en producción que ha sido desplegada, y versionada con etiquetas apropiadas por cada lanzamiento.
- La rama **develop** se crea a partir de master y contiene el código para la siguiente característica (*feature*) que ha sido desarrollada.

Además de las ramas principales, existen las denominadas ramas de apoyo:

- La rama **feature** se usará para desarrollar nuevas funcionalidades, se crearán a partir de la rama **develop** y al terminar con la funcionalidad, se tiene que fusionar (*merge*) otra vez con **develop**.
- La rama **release** se usará para lanzar (*release*) una nueva versión de nuestro proyecto. Estas ramas son creadas a partir de **develop**. En este punto, cualquier corrección de errores/regresión se crea en **hotfix** y se fusiona (*merge*) directamente en la rama de **release**. Al momento de poner (*deploy*) en producción, la rama de **release** se fusiona (*merge*) de nuevo en **develop** y **master**, y se crea un **tag**. Este tag se nombra con Versionamiento Semántico[2]. Cada nuevo tag se convierte en la última versión del proyecto.
- La rama **hotfix** servirá para cualquier error/regresión que se encuentre después de un lanzamiento (*release*). Son creadas a partir de **master**, se prueban y se fusionan (*merge*) de la misma manera que las ramas de **release**.

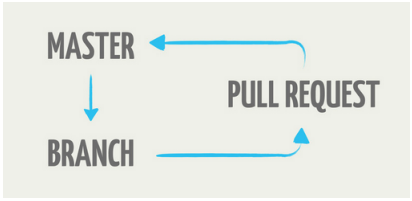

Fig. 3. Mateusz Mistecki, *GitHub flow*, 2017 [6]

B. GitHub flow

Entonces se podría decir que ¿en GitHub se trabaja con Git flow? Definitivamente, no. Como se logra ver en el esquema de Git flow (ver Fig. 2), no es una buena decisión elegir este modelo cuando se trabaja en ambientes de Entrega Continua o Despliegue Continuo por ser un proceso complicado. GitHub trabaja con un entorno de Despliegue Continuo [5] y cada vez que terminan una nueva característica (*feature*) la publican inmediatamente (después de toda la cadena de automatización, ver Fig. 1).

Chacon [5], nos describe los pasos a seguir en el flujo de GitHub:

- A cualquier cosa en la rama master se le puede hacer un despliegue (*deploy*).
- Para trabajar en algo nuevo, se crea una rama con un nombre descriptivo que se origina a partir de master (e.g., new-oauth2-scopes).
- Las confirmaciones de cambio (*commits*) se realizan de manera local y regularmente se envía (*push*) su trabajo a la rama con el mismo nombre en el servidor.
- Cuando se necesita retroalimentación o ayuda, o cree que la rama está lista para fusionarse (*merge*), se crea una solicitud de recuperación/integración (*pull request*).
- Después que alguien más ha revisado y aprobado la característica, se puede fusionar (*merge*) de nuevo a la rama master.
- Una vez que se haya fusionado (*merge*) y enviado (*push*) a master, se puede y se debe poner (*deploy*) en producción inmediatamente.

C. GitLab Flow

El flujo de GitHub es muy simple y limpio. GitLab ofrece una alternativa similar e introduce las ramas de despliegue (*deployment branches*). Donde las herramientas de IC/EC/DC deben correr, antes de ir a producción. La idea es proteger la versión en producción ya que el proyecto sobre el que ocurren los cambios es utilizado por millones de personas, un

[2]http://semver.org/lang/es/

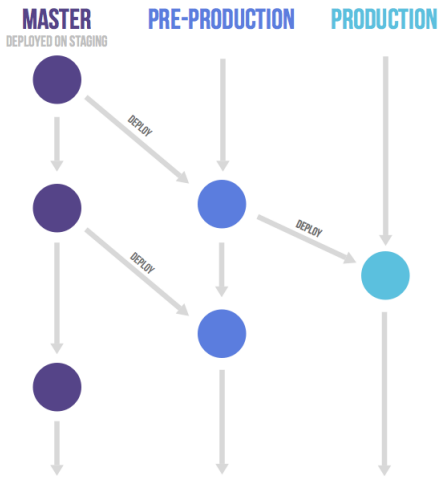

Fig. 4. Sytse Sijbrandij, *Ramas de ambiente*, 2014 [7]

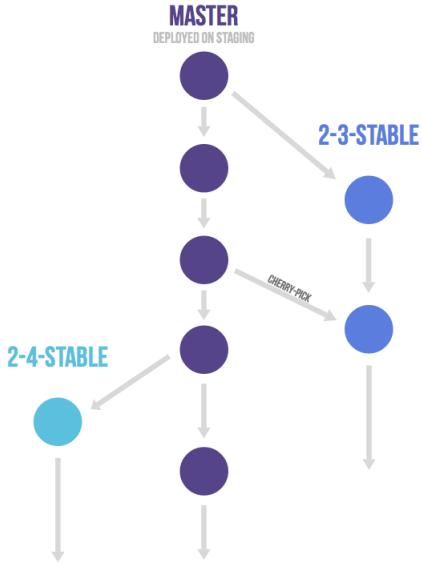

Fig. 5. Sytse Sijbrandij, *Ramas de lanzamiento*, 2014 [7]

ejemplo de estos son los que utilizan el modelo de Software como un Servicio (ScuS). Es por ello que GitLab propone dos alternativas.

Si se trabaja en aplicaciones ScuS, utilizamos la propuesta de ramas de ambiente (*environment branches*) — e.g., **pre-production** y **production** (ver Fig. 4). Estas ramas se originan a partir de la rama master cuando estamos listos para desplegar (*deploy*) nuestra aplicación. Al tener diferentes ramas por ambiente se nos permite configurar las herramientas de IC/EC/DC para desplegar automaticamente la aplicación a la hora de hacer una confirmación de cambios (*commit*) en estas. Si hay un problema crítico lo podemos solucionar en la rama de característica (*feature*) o en la master y luego fusionar (*merge*) los cambios a las ramas de ambiente.

El otro caso es el de las aplicaciones que requieren lanzamiento (*release*) (como las aplicaciones móviles o de escritorio). GitLab propone el uso de ramas **release**. Se sigue el mismo flujo de GitHub, pero solo en caso de que necesite lanzar el software a producción, se trabaja sobre las ramas **release**. En la Fig. 5, se muestra un ejemplo de ramas que contienen cambios menores de una versión estable (2-3-stable, 2-4-stable), note que se originan de master. Después que una rama **release** es lanzada, solamente correcciones de errores serios son incluidos en la rama **release**. Estas correcciones se deben fusionar primero en **master** y luego se entresaca (*cherry-picked*) en la rama de lanzamiento. De esta forma nos aseguramos de no encontrar el mismo error en **master** para futuros lanzamientos. "A esto se lo conoce como una política 'primero en la cadena' (*upstream first*) que también practica Google[3] y Red Hat[4] " [7]. Cada vez que se incluye una corrección de errores en una rama de lanzamiento, se genera la versión de parche (*patch*) (para cumplir con las versiones semánticas) creando una nueva etiqueta.

IV. Comparación de la principales estrategias

Como se mencionó, es cierto que no hay un flujo de trabajo milagroso que todos deberían seguir, ya que todos los modelos son subóptimos. En busca de seleccionar un modelo de trabajo en función de la pieza de software a desarrollar podemos realizar la siguiente comparación:

1) Según Vincent Driessen Git flow es un estándar bien definido que presenta de manera elegante un modelo de ramificación y lanzamiento [4]. Pero su complejidad presenta varios problemas. Muchas personas se quejan de lo tedioso que puede ser el cambio entre las ramas master y develop o la manera en que se manejan los errores. Y coincido con Scott Chacon[5] en que la configuración del entorno de desarrollo para utilizar el flujo es muy grande y complicada, los desarrolladores difieren al utilizar Git GUI o Git Bash (esto puede significar un grado de complejidad mayor) y la curva de aprendizaje puede ser bastante alta para los nuevos integrantes que no conocen el proceso [5].
   En fin, la recomendación es usar Git flow para productos donde se necesita dar soporte de manera continua a previas versiones en producción, mientras se desarrolla la siguiente versión. La recomendación de Chacon es:
   > Para los equipos que tienen que hacer lanzamientos formales en un intervalo de períodos largos (de algunas semanas a algunos meses entre lanzamientos), y deben hacer correcciones urgentes y ramas de mantenimiento y otras cosas que surgen con poca frecuencia del lanzamiento, git-flow tiene sentido y yo recomendaría su uso.
2) Este gurú de Git continúa diciendo:

[3] http://www.chromium.org/chromium-os/chromiumos-design-docs/upstream-first

[4] http://www.redhat.com/about/news/archive/2013/5/a-community-for-using-openstack-with-red-hat-rdo

[5] Scott Chacon es desarrollador de software en GitHub. Autor del libro Pro Git (https://git-scm.com/book/en/v2) y otras publicaciones importantes relacionadas a Git. Más información: http://scottchacon.com/about.html

Para los equipos que han establecido una cultura de Despliegue Continuo, que lanzan a producción todos los días, que están constantemente probando e implementando, se recomienda algo más simple como GitHub Flow.

También GitHub flow facilita el revertir cambios para funcionalidades individuales, alienta la investigación exahustiva de las solicitudes de recuperación/integración (*pull request*), esto permite que exista mayor control en las revisiones de código.

3) Para los productos de software más complejos como Facebook y Gmail, es necesario el uso de ramas de despliegue donde las herramientas de IC/EC/DC entran en función antes de pasar a producción. Como se mencionó anteriormente la idea es proteger la versión en producción.

V. Conclusión

Es interesante comprender las diferentes filosofías que han surgido a través del uso de Git y cómo los equipos de desarrollo han elegido adaptarlas a su flujo de trabajo. Por supuesto, todas ellas son opcionales y puede que las estrategias de desarrollo cambien a medida que el proyecto avance y el equipo madure. Hacer uso de cualquiera de las estrategias existentes es un gran paso para optimizar el proceso de desarrollo dentro del equipo. Cada modelo que se ha explicado tiene su atractivo, sus pro y sus contras. Queda a criterio propio la elección de alguno de ellos, o ninguno. Existen muchos otros esquemas y es de gran importancia tomar un tiempo para analizar y elegir el uso de alguno de ellos (los miembros del equipo lo van a agradecer).

Referencias

[1] S. Pittet, "Continuous integration vs. continuous delivery vs. continuous deployment | continuous delivery," 2017. [Online]. Available: https://www.atlassian.com/continuous-delivery/ci-vs-ci-vs-cd

[2] A. W. Services, "Aws codepipeline concepts." [Online]. Available: http://docs.aws.amazon.com/codepipeline/latest/userguide/concepts.html

[3] ThoughtWorks, "Continuous integration." [Online]. Available: https://www.thoughtworks.com/continuous-integration

[4] V. Driessen, "A successful git branching model." [Online]. Available: http://nvie.com/posts/a-successful-git-branching-model/

[5] S. Chacon, "Github flow." [Online]. Available: http://scottchacon.com/2011/08/31/github-flow.html

[6] M. Mistecki, "Github flow," Mar 2017. [Online]. Available: http://mateuszmistecki.pl/2017/03/27/github-flow/

[7] S. Sijbrandij, "Gitlab flow," Sep 2014. [Online]. Available: https://about.gitlab.com/2014/09/29/gitlab-flow/