# Gardez **le fil**.

Abonnez vous à notre flux RSS ou par email.

On vous rappelle !

## Adopter un modèle de versionnement efficace avec Git

07
02
2012

*posté par Nicolas Cavigneaux dans les catégories git*

Comme vous le savez peut-être, chez Synbioz, nous versionnons le code de nos projets à l'aide du DSCM Git dans le cadre de nos sessions de développement agile.

J'ai au départ été très critique vis-à-vis de Git, de sa relative complexité d'utilisation et de sa courbe d'apprentissage plutôt ardue dès qu'il s'agit de faire des choses un peu évoluées. En effet, je viens du monde de Mercurial que j'utilise depuis plusieurs années et que je trouve beaucoup plus simple d'accès et d'utilisation.

Je me suit donc beaucoup documenté à propos de Git, fais des essais, parcouru les manuels pour comprendre les commandes et leurs utilités puis j'ai cherché à trouver un modèle de travail en équipe satisfaisant, propre et surtout simple à suivre.

Effectivement, si votre équipe ne suis pas une méthodologie de travail pré-déterminée, on se retrouve rapidement avec un historique fouillis, des merges inutiles, ...

Récemment, nous avons découvert une méthodologie particulièrement adaptée à notre façon de travailler et qui tire parti des points forts de Git. Pour ne rien gâcher, cette méthologie est accompagnée d'un ensemble de scripts qui étend Git et permettent de respecter facilement les guidelines de cette méthodologie.

Avant de passer à la présentation de cette méthodologie, j'aimerai resituer le contexte. Nous travaillons à plusieurs, sur des projets qui peuvent durer plusieurs mois, voir plusieurs années. Des intervenants externes peuvent eux aussi contribuer au code. Nous avons besoin pour chaque projet d'une version de production, d'une version de développement et éventuellement de branches dédiées à de nouvelles fonctionnalités complexes qui doivent être isolées du reste du développement.

Il faut donc qu'aucun développeur ne fasse l'erreur de travailler dans master qui est

considéré comme un reflet du code en production, il faut également que chaque développeur gère correctement ses branches pour ne pas polluer la branche de développement. Si toute l'équipe respecte ces règles, on se retrouve avec un dépôt propre, un historique lisible et cohérent et donc un processus simplifié pour le déploiement en production ou la maintenance d'un serveur de pré-production tout en gardant la possibilité de faire évoluer le code très vite et dans diverses directions sans polluer le code stable.

## Le modèle adopté

Nos recherches nous ont mené à cet article qui décrit tout à fait la situation dans laquelle nous sommes et propose un workflow robuste.

Première constatation, le choix de Git n'est pas anodin puisqu'il est particulièrement adapté à un modèle à base de branches. Tout notre worflow se reposera donc sur un système de branches.

Lorsque nous commençons un projet, nous mettons toujours en place un dépôt central qui fait office de référence. Bien que Git soit un outil décentralisé, la mise en place d'un dépôt "maître" s'avère indispensable si vous travaillez à plusieurs. Chaque participant au projet va donc cloner le projet à partir de ce dépôt et c'est également sur ce dépôt que seront envoyées les modifications faîtes.

### Une branche de production saine et cloisonnée

Sur notre dépôt, nous avons besoin de pouvoir gérer le code en production ainsi que les développements en cours.

Il est donc pertinent de considérer que la branche "master" représente l'état du code en production.

Il nous suffit maintenant de créer une branche supplémentaire "develop" qui permet de gérer les développements pour les versions à venir, c'est ce qu'on appelle souvent la branche d'intégration :

```
$ git branch develop
$ git checkout develop
```

On peut donc dès maintenant contribuer au code et le commiter dans la branche "develop" sans que cela impacte notre code de production, voilà qui permet d'avancer en toute sérénité sans craindre que quelqu'un déploie du code non-testé ou validé.

### Une branche pour développer les fonctionnalités futures

Dans "develop" nous allons donc faire de multiples commits pour agrémenter notre logiciel de nouvelles fonctionnalités. Cette branche peut d'ailleurs être utilisée en pré-production pour que vos testeurs et clients puissent profiter des avancements du projet et vous faire des retours.

Mais attention, “develop” n’est pas un fourre-tout et ne devrait être utilisé que pour les petites modifications simples ayant peu d’impact et ne demandant pas plus de 30 min de travail. En effet si vous travaillez directement sur “develop” pour une fonctionnalité estimée à 1 mois de développement, il y a fort à parier que vous aller ennuyer vos collégues avec vos commits qui s’incrustent entre 2 commits légitimes sans pour autant apporter une fonctionnalité finalisée.

Si vous êtes dans ce cas de figure, il vous faut créer une branche dédiée au développement de votre fonctionnalité.

## Des branches spécifiques pour les développements lourds

Ces branches de développement (feature) permettent de travailler de manière détachée du reste de l’équipe en onant vos modifications ce qui permet de ne les appliquer sur “develop” qu’une fois satisfait et par la même occasion de ne pas polluer les collègues.

On peut vouloir commencer à développer une fonctionnalité qui ne sera appliquée qu’à la release N+1, la branche de feature est donc dans ce cas l’unique solution puisque tout ce qui est dans develop est considéré comme faisant partie intégrante de la prochaine release.

Une branche de développement est créée à partir de “develop” et sera, une fois terminée, mergée dans “develop”. Ce type de branche reste généralement locale à la machine du développeur qui fini par la merger dans “develop” le moment voulu. Il reste toutefois des cas où ces branches de feature sont partagées sur le dépôt central pour une revue par les autres développeurs.

Voici comment procéder :

```
$ git checkout -b feature/foo develop
```

On a donc notre nouvelle branche “feature/foo” basée sur l’état actuel de “develop”. Nous pouvons donc coder et commiter autant de fois que nécessaire. Une fois fini, il faut intégrer cette branche dans “develop” :

```
$ git checkout develop
Switched to branch 'develop'

$ git merge --no-ff feature/foo
Updating ea1b82a..05e9557
(Summary of changes)

$ git branch -d feature/foo
Deleted branch myfeature (was 05e9557).

$ git push origin develop
```

On est donc retourné dans la branche “develop” dans laquelle on demande à git de merger notre branche de feature. L’option `--no-ff` permet de forcer la création d’un commit

même si les changements peuvent être intégrés en fast-forward. Ceci permet de garder une trace dans l'historique du développement dans une branche dédiée. Une fois mergée, nous supprimons la branche de feature devenue inutile et on push les modifications sur le serveur central.

## Passage en production

Lorsque "develop" atteint un état satisfaisant pour créer un nouvelle release et déployer, il faut merger "develop" dans "master", bumper la version, ajouter un tag de version, mettre à jour un README, ... Enfin le code peut être déployé en production.

On pourrait imaginer qu'à chaque commit dans "master", un hook soit déclenché pour déployer en production.

Nous allons donc, lors d'une release, créer une branche de support à la release. Les releases sont créées sur la base de la branche "develop" et doivent être mergées dans "master" et "develop".

Cette branche ne servira qu'à faire des modifications mineures liées à la création de la release :

```
$ git checkout -b release/v1.2 develop
Switched to a new branch "release/v1.2"
```

On peut maintenant bumper la version, mettre à jour le changelog, etc

```
$ git commit -a -m "Bumped version number to 1.2"
[release/v1.1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)
```

Il ne nous reste plus qu'à merger cette release dans master pour qu'elle prenne effet :

```
$ git checkout master
Switched to branch 'master'

$ git merge --no-ff release/v1.2
Merge made by recursive.
(Summary of changes)

$ git tag -a v1.2
```

On souhaite également récupérer ces informations de release dans "develop" :

```
$ git checkout develop
Switched to branch 'develop'
```

```
$ git merge --no-ff release/v1.2
Merge made by recursive.
(Summary of changes)
```

Notre release est donc intégrée en production (master) et en intégration (develop), on peut supprimer la branche :

```
$ git branch -d release/v1.2
Deleted branch release/v1.2 (was ff452fe).
```

## Dépanner les bugs critiques en production

Il vous arrivera certainement de faire face à un bug critique passé en production. Que faire dans ce cas ? Comment est-il géré dans le workflow ? Vous pensez peut-être qu'on doit nécessairement passer par une nouvelle release mais ce n'est pas le cas, ce n'est d'ailleurs même pas souhaitable puisque vous intégreriez au passage des fonctionnalités dans "develop" qui ne sont pas encore prête pour la production.

Dans ce cas de figure, il s'agit de faire ce qu'on appelle un "hotfix". Un hotfix est un patch qui va s'appliquer directement à la branche de production (master) et qui sera ensuite également appliqué sur la branche d'intégration (develop). On peut maintenant déployer la version de production corrigée mais aussi jouir de ces corrections dans "develop".

Pour se faire, comme toujours nous passerons par l'utilisation d'une branche pour avoir un historique sain :

```
$ git checkout -b hotfix/v1.2.1 master
Switched to a new branch "hotfix-1.2.1"
```

On crée une branche basée sur master et on bump la version du projet :

```
$ git commit -a -m "Bumped version number to 1.2.1"
[hotfix/v1.2.1 41e61bb] Bumped version number to 1.2.1
1 files changed, 1 insertions(+), 1 deletions(-)
```

On peut maintenant corriger le bug :

```
$ git commit -m "Fix huge production bug"
[hotfix/v1.2.1 abbe5d6] Fix huge production bug
5 files changed, 32 insertions(+), 17 deletions(-)
```

Et intégrer la correction en production :

```
$ git checkout master
Switched to branch 'master'
```

```
$ git merge --no-ff hotfix/v1.2.1
Merge made by recursive.

(Summary of changes)

$ git tag -a v1.2.1
```

mais également en intégration pour ne pas perdre le fix au passage :

```
$ git checkout develop
Switched to branch 'develop'

$ git merge --no-ff hotfix/v1.2.1
Merge made by recursive.
(Summary of changes)
```

Le patch étant intégré en production et en développement, on peut supprimer la branche de hotfix devenue inutile :

```
$ git branch -d hotfix/v1.2.1
Deleted branch hotfix/v1.2.1 (was abbe5d6).
```

Tous les cas d'usages typiques ont été couverts et on voit qu'en suivant ce workflow, on peut maintenir un dépôt sain avec un historique clair et parfaitement adapté à la manipulation si d'aventure on devait revenir en arrière ou faire sauter une fonctionnalité.

C'est propre et carré mais assez fastidieux n'est-ce pas ? Suivre ce modèle au quotidien demande concentration et discipline, pour tous les développeurs. Heureusement, une bonne âme nous simplifie le travail grâce à un ensemble d'extensions pour Git qui permettent de simplifier l'application de ce workflow.

## Git-flow

Git-flow est un ensemble d'extensions pour Git livré sous forme de shell-scripts très simples d'accès.

Git-flow va permettre de suivre le workflow présenté précédemment sans avoir à tout retenir, surtout si comme moi vous êtes un peu perdu avec toutes les commandes et options de Git.

### Initialisation

```
$ git flow init
```

va permettre de paramétrer votre dépôt pour une utilisation via git-flow. Notez que tout ceci est local ce qui veut dire que vous pouvez très bien utiliser git-flow alors que vos collègues ne l'utilisent pas.

Des questions sur la nomenclature vous seront posées, je vous conseille vivement d'utiliser les valeurs par défaut qui sont quasiment des conventions.

Une fois terminé, vous êtes automatiquement positionné sur la branche "develop", vous pouvez commencer à travailler.

## Créer une branche de feature

```
$ git flow feature start foo
```

Vous êtes désormais dans la branche "feature/foo" dans laquelle vous pouvez développer tranquillement votre fonctionnalité. Vous pouvez bien évidemment repartir dans d'autres branches, créer plusieurs branches de feature en parallèle, etc.

Une fois vos modifications terminées et prêtent à être intégrées dans "develop", vous pouvez finaliser cette branche :

```
$ git flow feature finish foo
```

Votre branche "feature/foo" va être mergée dans "develop" puis effacée et vous vous retrouverez à nouveau sur la branche "develop".

## Mise en production

Lorsque "develop" représente l'état souhaité en production, vous pouvez passer à la release :

```
$ git flow release start v1.0.0
```

Une branche "release/v1.0.0" est créée, vous pouvez donc bumper la version de l'appli et faire les dernières modifications avant release. Une fois paré, vous pouvez finaliser la release :

```
$ git flow release finish v1.0.0
```

Ceci aura pour effet de merger "develop" dans "master", de taguer la release, puis de back-merger ces modifications dans "develop". La branche de release sera ensuite supprimée et vous retournerez sur la branche "develop".

## Hotfix

Si vous avez besoin de patcher urgemment la production, vous utiliserez la commande dédiée aux hotfixes :

```
$ git flow hotfix start typo
```

Une branche “hotfix/typo” est créée et vous pouvez commencer à patcher. Une fois fini, il suffira de finaliser la branche :

```
$ git flow hotfix finish typo
```

Ceci aura pour effet d’appliquer votre fix sur master et develop, d’ajouter un tag puis de supprimer la branche de hotfix. À ce propos, n’oubliez pas de bumper la version avant de finaliser le hotfix.

J’espère avoir pu vous éclairer un peu sur la façon dont nous gérons le code source dans nos équipes et que celà vous donnera envie d’essayer ce worflow qui selon moi devrait permettre à vos dépôts de rester cohérents, de pouvoir gérer indépendamment les avancées fonctionnelles, les patchs et releases sans finir la journée avec un mal de tête !

L’équipe Synbioz.

Libres d’être ensemble.

Le kit du bon développeur Rails, quelques gems à connaitre, partie 2

HockeyApp - A great service that lets you test your mobile or mac application.

## Partage sociaux

## Articles connexes

13
09
2011

### Optimiser sa productivité en console

git shell zsh bundler gem github fr

J’ai l’intime conviction que tous les développeurs, dès que cela est possible, devraient travailler sous un environnement dérivé d’Unix (Linux, *BSD, Mac OS X) ce qui assure d’avoir à disposition tou…

Lire la suite

## Commentaires (25)

### BOB

*08/02/2012 à 13:01*

Super article. Je ne me suis jamais frotté au workflow nvie même si le .png trône, imprimé, sur le coté de mon UC au boulot. Par souci d'inspiration sans doute.
Toujours est-il que je m'y frotterais surement jamais si git-flow n'existait pas. Sacré discipline sans cet outil je trouve...

Il y a débat sur les branches de feature, concernant le fait ou non de récuppérer les commits de la branche develop parente au fil de l'eau ou non, au risque d'avoir un merge difficile si la feature traîne dans le temps.

C'est bien entendu dépendant de l'architecture du projet et/ou du nombre de développeurs. J'imagine que dans une équipe comme celle de Synbioz cela ne doit pas poser de problème, mais quid dans un projet avec beaucoup de contributeurs ?

### FUSE

*09/02/2012 à 09:31*

A mon sens celui qui développe sa feature ne devrait pas s'occuper de develop, c'est plus le boulot du merge master s'il y en a un, autrement j'aime autant le faire à la fin.

Mais c'est valable parce qu'on ne fait pas de branche de feature qui dure 3 mois. Dans le cas contraire ça peut avoir du sens je trouve.
Ce qui m'a perturbé avec git flow c'est le fait que le flow ne soit pas imposé aux devs.

Potentiellement un dev peut utiliser une branche foo pour gérer develop en local. A mon sens la configuration du flow devrait être dans le projet.

Mais c'est un autre sujet...

### NICOLAS CAVIGNEAUX

*09/02/2012 à 09:59*

@bob Concernant le débat sur les branches de feature j'applique une règle très simple. Je rebase master dans ma branche de feature juste avant de la merger dans master. Ceci n'est applicable que si la branche de feature n'est pas partagée.

Si la branche a été partagée, le rebase devient impossible dans ce cas je passe donc par un merge toujours à la toute fin de la vie de la branche, j'évite de merger entre 2 pour ne pas pourrir l'historique.

Il est vrai que du coup on peut se retrouver face à des merge compliqués avec des conflits à résoudre mais je n'ai pas trouvé d'autre solution à ce jour.

### NICOLAS CAVIGNEAUX

*09/02/2012 à 10:06*

@fuse l'idée de l'auteur de git flow est de ne rien imposer à l'équipe et de permettre à un membre d'utiliser git flow alors que d'autres ne l'utilisent pas. La config est donc locale mais c'est vrai qu'il pourrait être intéressant d'avoir une config par projet qui impose les noms de branches etc.

Sinon effectivement je pense comme toi, tu laisses le boulot au merge master ou alors tu rebase à la toute fin.

**NICOLAB**
*15/02/2012 à 12:27*

Super article. Par contre comment vous vous organisez au niveau de votre IDE ?

Car il faut jongler d'une branche à une autre et donc par exemple sous eclipse créer autant de projet / sous-projet. Je suis parti pour faire projet (branche develop) / sous-projet (autre branche).

Il y a peut être mieux comme organisation ?

**NICOLAS CAVIGNEAUX**
*15/02/2012 à 12:55*

@Nicolab À vrai dire, nous n'utilisons pas d'IDE, on a pour habitude d'avoir un éditeur (Textmate ou Vim) et une console.

On peut donc passer d'une branche à l'autre avec une simple commande dans le terminal ou directement depuis l'éditeur (plugin Git).

Il n'y a pas de notion de projet et en fait que se passe t'il dans Eclipse si tu fais un "git checkout ta_branche" ? Il est perdu ? En ce qui nous concerne, on a juste le code qui s'update dans notre éditeur donc aucun souci.

**NICOLAB**
*15/02/2012 à 12:56*

"Ceci aura pour effet d'appliquer votre fix sur master et develop, d'ajouter un tag puis de supprimer la branche de hotfix. À ce propos, n'oubliez pas de bumper la version avant de finaliser le hotfix."

Comment ça ? Pas sûr d'avoir compris ce que tu voulais dire, je préfère demander au cas où ...

**NICOLAB**
*15/02/2012 à 13:02*

@Nicolas : Ok merci pour ta réponse.

"Il n'y a pas de notion de projet et en fait que se passe t'il dans Eclipse si tu fais un "git checkout ta_branche" ? Il est perdu ?"
Bein justement, je ne sais pas car pour l'instant j'utilise GIT pour sécuriser mes devs (backup donc) et je n'ai pas de notion de branche (pour l'instant).

Mais là ce n'est pas trop tenable niveau organisation, je m'oriente donc vers GIT flow pour plus de souplesse et je me demande aussi comment organiser ça dans Eclipse pour éviter de planter le projet en cours avec des checkouts ou autre

**NICOLAS CAVIGNEAUX**
*15/02/2012 à 13:20*

@Nicolab Je pense vraiment que ça ne va poser aucun problème à Eclipse de passer d'une branche à l'autre, et pas besoin de créer de sous-projets. Ça rendrait Eclipse inutilisable (en tout cas très peu pratique) avec Git ou tout autre système de versionnement à base de branches. De plus, Git est intégré à Eclipse si je ne m'abuse. Essais de créer plusieurs branche dans un projet et de passer de l'une à l'autre et voir ce

que ça donne. Ça devrait être transparent et sans souci.

**NICOLAB**
*15/02/2012 à 14:19*

Je vais tester et je reviens poster mon retour, ça peut servir.

Merci d'avoir pris le temps de me répondre ;)

**NICOLAB**
*18/02/2012 à 09:00*

Bonjour,

je reviens faire mon petit retour concernant Eclipse et Git flow. C'est un peu ce que je craignais, le changement de branche fait qu'eclipse doit régénérer le projet à chaque changement de branche (sur un projet Symfony2 c'est 2 minutes d'attente) et l'historique d'eclipse est faussé.

Pas très pratique.

**GUILLAUME**
*27/03/2012 à 08:14*

Un article très clair et bien écrit, il me reste cependant des questions pour bien comprendre : master reflète la production mais est-ce la production ?
Je ne cerne pas bien encore l'organisation complète allant du poste de travail d'un dev jusqu'au serveur web de production... Faut-il nécessairement passer par un serveur de dev faisant l'intermédiaire ?

Autre question : voilà en gros mon arborescence typique web :
- application
--- controllers
--- models
--- views
--- helpers
- www
--- index.php
--- assets (js, css)
--- media

Dans git vous mettez quoi ? Juste application ? Tout sauf 'media' ? (média peut peser des 100aine de Go)

Merci pour votre aide ! :)

**FUSE**
*27/03/2012 à 10:19*

Bonjour Guillaume,

Effectivement dans cette organisation master doit refléter la prod.
Il n'y a aucune obligation d'avoir un serveur de dev intermédiaire, même si c'est ce que nous faisons (staging) afin que nos cllients puissent voir l'avancée des travaux.

Le staging est lui calé sur develop. L'intermédiaire c'est d'avoir un outil de déploiement, je te conseille nos articles sur capistrano.

Enfin, on versionne tout sauf les contenus uploadés (media) et les fichiers de configurations.

### GUILLAUME

*27/03/2012 à 10:23*

Merci pour ces réponses !

Du coup je peux installer et initialiser git sur mon serveur de prod, puis créer les branches et développer sur ces dernières.
Ou alors passer par un outil de déploiement, je m'en vais de ce pas lire vos articles sur capistrano ! ;)

### JONATHAN-DAVID

*24/07/2012 à 18:15*

Bonjour fuse,
Merci pour votre article.
Qu'entendez-vous par "bumper une version" ? Avez-vous un fichier VERSION avec un numéro dedans dans votre working copy ?
Y a-t-il un automatisme existant permettant de synchroniser version fichier<=>version du nom de branche<=>version de tag ?
Merci beaucoup !

### NICOLAS CAVIGNEAUX

*26/07/2012 à 14:03*

@Jonathan-David: Tu as souvent dans ton code, un endroit où tu références la version en cours que ce soit une constante dans ton code ou un endroit dans le readme ou encore la liste des modifications apportées depuis la dernière release dans les release notes.

C'est cette info qu'il faut bumper. Il ne faut donc pas oublier de mettre à jour ta constante, ton README et / ou tes releases notes avant de releaser ton code. Ça va de paire.

### JONATHAN-DAVID

*26/07/2012 à 14:21*

@Nicolas, merci de ta réponse !
Je m'interrogeais sur l'existence d'outil permettant de ne pas à avoir à inscrire le numéro de version dans 3 endroits : fichier de code, commit log, tag... Sur SVN, il y avait un string $SvnId$ rempli au moment du commit par le numéro de version, l'auteur et la date...
Ce que cherchais est codé en Ruby ici par exemple :
http://code418.com/blog/2012/07/23/github-dashboard-number-8/
"I also enhanced the :release task of the Rakefile so it automatically deploys the current version of the dashboard, tags the current code base and bumps the version."

Certes, en faisant tout à la main (surtout, comme tu le soulignes, que la documentation style README doit être mis à jour pour les nouvelles versions), on voit mieux ce qu'on fait.

Merci pour ce beau blog ! Joie !

**MAKA**

*03/09/2012 à 10:35*

Bonjour et merci pour cet article très enrichissant.

Je m'intéresse depuis peu au système de versionnement GIT et particulièrement à GITFLOW qui semble très bien pensé.

Néanmoins une chose m'échappe. Je développe sur NetBeans et je me pose un peu les mêmes questions que Nicolab.

Pendant le développement (peu importe la branche developp, feature, release ..) comment tester notre application "branche par branche". Est ce que le simple fait de faire un "checkout "puis d'accéder à notre application suffit à refléter l'état de la branche en cours ?

Est ce possible, à partir d'un seul dépôt et d'un seul serveur d'accéder via navigateur (pour test) à chaque branche pendant que les utilisateurs finaux de l'application continuent tranquillement à accéder à la branche "master" (production) ???

Sinon, quelle architecture faudrait il mettre en place pour permette ce fonctionnement ? (plusieurs dépôts/serveurs ? )

J'espère que quelqu'un comprendra ma requête malgré mes idées embrouillées :)

merci par avance .

**NICOLAS CAVIGNEAUX**

*24/09/2012 à 10:13*

@ MaKa: Oui un simple checkout te permet de sauter d'une branche à l'autre et donc d'avoir une arborescence qui refléte la branche en cours.

Concernant la possibilité de laisser les utilisateurs sur master alors que tu travaille sur develop ou autre, c'est tout l'intérêt du système. Laisser les utilisateurs utiliser une version stable pendant que tu continues tes développement et les testes.

Un seul serveur / dépôt te suffit mais il faudra faire tourner 2 instances de ton app, une en master et une en develop. Tu as ainsi un DNS qui pointe sur la version de production et est utilisée par les utilisateurs finaux et un DNS qui pointe sur ton instance en "develop" qui te permet de tester tes développement.

En espérant t'avoir éclairé.

**PACHEIKH**

*17/12/2012 à 14:02*

Slt.

Merci pour cet excellent article.
Moi mon problème reste les restrictions liées aux branches. Si vous devez travailler à plusieurs sur un dépôt, êtes vous parvenus à restreindre l'accès à certaines branches pour juste quelques développeurs?

Merci encore.

**NICOLAS CAVIGNEAUX**

*17/12/2012 à 14:51*

@Pacheikh Salut, il ne me semble pas que git donne la possibilité de gérer les droits par branche non. Le plus simple reste encore de pusher les branches privées sur un autre dépôt auquel n'auront accès que les personnes autorisées. Un fork en gros.

**PACHEIKH**
*17/12/2012 à 15:32*

Tu as raison Nicolas.

Merci en tout cas pour la réponse.

J'ai un cas de figure où j'envisageais de créer 3 environnements pour un projet (dev, staging et prod). Je ne veux pas par exemple qu'un testeur puisse avoir accès à dev mais plutot à staging après qu'un développeur qui bosse sur dev ait fini d'y pusher son travail.
Cependant j'avais voulu, au lieu de créer 1 dépôt pour chaque environnement, créer juste 3 branches et restreindre l'accès à la branche dev aux testeurs, etc.
Tu ne verrais pas une façon plus souple de faire ce genre de restrictions.

Merci d'avance.

**BOB**
*17/12/2012 à 15:39*

Il faut utiliser une surcouche à git pour faire du contrôle d'accès à un dépôt. Dans ce domaine, il y a gitolite(*) qui fait ça très bien à l'aide de clé SSH et une meta configuration à base de git évidemment. :)

(*)http://sitaramc.github.com/gitolite/

À noter, le projet gitlab (gitlabhq.com) utilise gitolite en backend.

**PACHEIKH**
*17/12/2012 à 15:46*

Merci Bob, je crois que dans ce cas gitolite est mon ami. Je vais de ce pas me documenter là dessus.
Encore merci à vous tous ;)

**NICOLAS CAVIGNEAUX**
*17/12/2012 à 16:01*

@Bob Intéressant, merci pour le lien !

## Ajouter un commentaire

Nom *

Votre nom

Email *

Votre Email

Site web

Votre site web

Message *

Votre message

Suivre les commentaires

Envoyer

## Recevoir nos articles

Votre email S'inscrire

## Rechercher un article

Votre recherche

## Tags de l'article

git fr git-flow workflow dscm

## Les derniers articles

Le 16/10/2013
Générer des PDF depuis votre application Rails

Le 09/10/2013
Communication réseau avec Go

Le 07/10/2013
Créer un nouvel utilisateur PostgreSQL

Le 24/09/2013
RubyMotion - les tableviews

Le 18/09/2013
respond_to et Ruby on Rails

## Newsletter

Votre email S'abonner

Besoin de compétence rails ?

## Nos dernières nouvelles

*Nos derniers tweets*

*5 rue de douai 75009* ***Paris***

*2 rue hegel*
*ZAC Euratechnologies*
*Batiment Canal 59160* ***Lille***

***Troyes***