

Mon tableau git vs mercurial

Introduction

Quel est le meilleur gestionnaire de version : *git* ou *mercurial* ?

git	mercurial	commentaire
concepts sous-jacents simples	concepts sous-jacents nombreux	surtout en introduisant de nouveaux concepts avec <i>mq</i> (mercurial queues)
toutes les commandes font un <i>less</i> implicite si nécessaire	pas de <i>less</i> implicite, surtout dans le cas des <i>logs</i> ou d'un <i>diff</i> - un peu chiant	Mais, mercurial introduit une extension appelée ???
tout est dedans d'origine (ou presque)	beaucoup d'extensions qu'il faut ajouter au fur et à mesure	un peu chiant certaines fois de devoir ajouter des choses pratiques (pour mercurial) : <i>less</i> , <i>mq</i> , couleur, etc.
status bien plus lisible et aidant	hg status n'est pas très utile, je trouve	
couleurs aisément ajoutables	il faut ajouter une extension pour en avoir	pas une grande différence, finalement
outils statistiques ?	existence d'outils statistiques dans des extensions	
gestion des tags dans des objets propres, internes à git	gestion des tags dans un fichier à part, versionné lui-même	Je n'aime pas l'approche de mercurial sur le sujet des tags qui introduisent, quand on les utilise, des <i>commits</i> supplémentaires qui me semblent un peu artificiels. Des problèmes surtout quand on revient en arrière.

Bref, il y a des choses bien des deux côtés mais je trouve que les outils sont plus abondants, d'emblée, dans *git*. Le modèle très simple du fonctionnement interne de *git* rend la manipulation des dépôts performante avec peu de complexité même si les concepts peuvent être peu familiers. J'ai l'impression que pour faire la même chose, *mercurial* a besoin d'introduire de nouveaux concepts (comme les *mercurial queues*) qui introduisent de nouvelles commandes.

Pour autant, les numéros de version plutôt que les hashes, c'est bien pratique et il y a un certain confort, une sorte d'élévation dans les commandes *mercurial* que l'on ne retrouve pas dans *git*. *Git*, c'est plus les mains dans l' cambouis.

Liens

Bonnes pratiques d'utilisation de ces logiciels

Git

- <http://tech.libresoft.es/doku.php/git> - un petit guide des meilleures pratiques pour *git*. Intéressant à suivre. Notamment :
 - garder des *commits* atomiques n'impliquant que des fonctionnalités descriptibles en une ligne. Pas de multiplicité dans les fonctionnalités ou les changements effectués.
 - Ne pas utiliser la branche maître (*master*) pour des développements.
 - Avant de peupler avec les nouvelles choses la branche maître, il faut que tout ait été testé et fonctionne.
- <http://tbaggery.com/2008/04/13/best-practices-for-contributing-to-rails-with-git.html> - Plus un guide de style pour les commentaires de *commit*. À suivre. Important.
- <http://www.kernel.org/pub/software/scm/git/docs/gitworkflows.html> - le *man* de *git* nous explique un *workflow* conseillé. Certains éléments sont certainement de trop si l'équipe est constituée de 4 personnes au max.
 - « As a general rule, you should try to split your changes into small logical steps, and commit each of them. They should be consistent, working independently of any later commits, pass the test suite, etc. »
 - Existence de plusieurs branches "officielle" (en plus de toutes celles créées par les utilisateurs) :
 - *maint* - la version qui maintient la version courante du code et sur laquelle on apporte des correctifs de bugs par exemple.
 - *master* - la version stable suivante du code sur laquelle se font les développements courants.
 - *next* - la version de test pour les choses devant aller dans le *master*.
 - *pu* - branche d'intégration pour des fonctionnalités nouvelles qui ne sont pas encore acceptées pour inclusion dans une version suivante.
 - Remarque : nom à franciser et ajouter des alias courts.
 - Utiliser les branches comme des sortes d'actions ou de (petites) tâches à faire. Un peu comme des "demandes" dans *redmine*. Ces branches d'une durée courte seront rapidement réintégrées dans les branches plus "officielles". Elles permettent aussi de préparer le terrain à un historique des modifications (*commits*) plus lisible (en « réécrivant l'histoire »).
 - fusionner des versions les plus expérimentales vers les plus établies (stables).
 - fusionner dans le sens contraire (stables -> expérimentales ou *maint* -> *pu*) rarement et pour des raisons précises (cf. doc *git*).
 - pour tester l'intégration d'une fonctionnalité (ou de plusieurs) ensemble avec une branche principale, utiliser une branche à mettre à la poubelle une fois les tests terminés. Ne pas se baser sur cette branche mais ne l'utiliser que pour faire des tests. Elle sera réceptrice des fusions des fonctionnalités devant être testées.
 - Une fois la nouvelle version prête pour devenir la nouvelle version maintenue, il faut la nommer avec un *tag*. Quelques autres manœuvres administratives juste avant (cf. man page de *gitworkflows*).
- <http://lwn.net/Articles/328436/> - **Rebasing and merging: some git best practices** - Quelques bonnes pratiques compilées par un publiciste des mailing lists du noyau Linux.
- <http://lwn.net/Articles/328438/> - Un mail de Linus sur la manière qu'il souhaite voir géré les sous-projets qu'il intègre dans son noyau Linux. En gros : historique à soi propre et ne pas écraser l'historique de ceux depuis lesquels on reprend du code.

Guides et tutoriaux

- http://www.sourcemap.org/Git_Guide - Bien : informé et assez complet.
- <http://gitref.org/> - la référence *git*.
- <http://progit.org/book/> - un livre sur *git* : Pro Git.
- <http://www-cs-students.stanford.edu/~blynn/gitmagic/index.html> - un autre livre sur *git* - Git Magic.