

# Utiliser Mercurial in a Nutshell

Adrien Herubel

8 novembre 2007

## Table des matières

<b>I</b>	<b>Pour commencer</b>	<b>1</b>
1	Conventions de nommage	1
2	Installer Mercurial	1
2.1	Sur Linux . . . . .	1
2.2	sur MacOSX . . . . .	1
2.3	sur Windows . . . . .	1
3	Premières commandes	1
3.1	Tester l'installation . . . . .	2
3.2	L'aide intégrée . . . . .	2
3.3	Syntaxe des commandes . . . . .	2
4	Notions	2
4.1	Repository . . . . .	2
4.2	Changesets, Révisions . . . . .	2
4.3	Branches, Heads et Tip . . . . .	2
4.4	Clone . . . . .	3
4.5	Commit . . . . .	3
4.6	Pull, Push et Merge . . . . .	3
<b>II</b>	<b>Recettes</b>	<b>3</b>
5	Configurer son <i>.hgrc</i>	3
6	Commencer un nouveau projet avec Mercurial.	3
7	Développer un projet avec Mercurial	4
7.1	Gérer le tracking des fichiers . . . . .	4
7.2	Renommer les fichiers . . . . .	5
7.3	Committer ses modifications . . . . .	5
7.4	Consulter l'historique . . . . .	5
7.5	Revenir en arrière et corriger ses erreurs . . . . .	6
7.5.1	Annuler un changement pas encore commité . . . . .	6
7.5.2	Annuler un changement déjà commité . . . . .	6
7.5.3	Voyager dans le temps . . . . .	7

<b>8</b>	<b>Participer à un projet avec mainteneur central</b>	<b>7</b>
8.1	Récupérer le projet . . . . .	7
8.2	Mettre à jour son repository . . . . .	7
8.3	Partager ses modifications . . . . .	8
8.3.1	La méthode push . . . . .	8
8.3.2	Envoyer des patches . . . . .	8
<b>9</b>	<b>Maintenir un projet multi-développeurs</b>	<b>8</b>
9.1	Publier le projet en ssh . . . . .	8
9.2	Publier le projet en http . . . . .	8
9.3	Appliquer et tester un patch . . . . .	9
9.4	Définir des numéros de version . . . . .	9
9.5	Maintenir deux versions d'un projet . . . . .	9

## Première partie

# Pour commencer

## 1 Conventions de nommage

Les notions et mots clés propres à Mercurial seront notés en **gras** tandis que les noms de commandes seront notés en *italique*. Les résultats de commande seront présenté sous la forme :

```
$ commande
resultat
```

## 2 Installer Mercurial

### 2.1 Sur Linux

Il existe un paquet Mercurial pour la plupart des distributions donc sur Debian et Ubuntu un simple :

```
$ apt-get install mercurial
```

devrait suffire.

Les sources sont disponibles ici : <http://www.selenic.com/mercurial/download>.

### 2.2 sur MacOSX

Un paquet est disponible pour les macs PowerPC et Intel à l'adresse suivante : <http://mercurial.berkwood.com>. Il est nécessaire de disposer d'une installation de Python valide.

### 2.3 sur Windows

Un installateur indépendant est disponible à l'adresse suivante : <http://mercurial.berkwood.com>.

## 3 Premières commandes

Il n'existe qu'une seule commande pour gérer l'ensemble des fonctions de Mercurial : *hg* (symbole du mercure)

### 3.1 Tester l'installation

Si l'installation s'est bien passée la commande suivante devrait fonctionner :

```
$ hg version
Mercurial Distributed SCM (version 0.9.3)

Copyright (C) 2005, 2006 Matt Mackall <mpm@selenic.com>
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

### 3.2 L'aide intégrée

Mercurial possède un système d'aide intégrée. *hg help* fournit une liste de commande et une rapide description tandis que *hg help <commande>* fournit le manuel de la commande.

```
$ hg help init
hg init [-e CMD] [--remotecmd CMD] [DEST]

create a new repository in the given directory

    Initialize a new repository in the given directory.  If the given
    directory does not exist, it is created.

    If no directory is given, the current directory is used.

    It is possible to specify an ssh:// URL as the destination.
    Look at the help text for the pull command for important details
    about ssh:// URLs.

options:
  -e ssh          specify ssh command to use
  --remotecmd cmd specify hg command to run on the remote side
```

### 3.3 Syntaxe des commandes

Toutes les commandes mercurial sont préfixées par *hg*, dans ce manuel on se référera le plus souvent au suffixe, par exemple la commande *log* est en réalité la commande *hg log*. La plupart des commandes peuvent s'appliquer à un fichier en particulier s'il est précisé ou à l'ensemble du repository s'il est omis. De plus la majorité des commandes supportent les expressions régulières.

## 4 Notions

### 4.1 Repository

Un repository Mercurial contient deux éléments, le **répertoire de travail** dans lequel les fichiers sont modifiés et le **store** où se trouve l'historique complet du projet. Contrairement systèmes de gestion de version centralisés type CVS/Subversion, il n'existe pas de Repository central. Chaque développeur possède un clone du repository et donc de l'historique du projet. Cette approche permet le développement en parallèle.

### 4.2 Changesets, Révisions

Un **changeset** est un lot de modifications dans le répertoire de travail, enregistré par une opération de **commit**. Le changeset intégré s'appelle une **révision**. Chaque révision possède un numéro de révision. En cas de développement parallèle les numéros de révision peuvent entrer en conflit, Mercurial assigne donc à chaque révision un identifiant de changeset unique long de 40 chiffres qui peut s'abrèger sous la forme "e38487".

### 4.3 Branches, Heads et Tip

Une **branche** est une révision fille d'une autre révision. Une révision parente peut avoir plusieurs révisions filles en cas de développement parallèle. Les dernières révisions de chaque branche sont appelées des **heads**. Le

**tip** est la head avec le plus haut numéro de révision. Une révision possédant deux révisions parentes s'appelle un **merge**.

#### 4.4 Clone

Un **clone** est une copie complète d'un repository comprenant le repertoire de travail et le store. Le clone est l'opération qui permet à plusieurs développeurs de travailler sur le même projet ou au même développeur de travailler sur deux versions du projet (private/public, stable/unstable). L'opération de clone est aussi une façon simple de tester l'incidence d'un changement sur le projet en toute sécurité.

Plus de details :

```
$ hg help clone
```

#### 4.5 Commit

Lors d'une opération de **commit** l'état du repertoire de travail est enregistré en tant que nouvelle révision. Chaque commit enregistre un changeset et crée une nouvelle révision. Un commit donne lieu à un rapport de commit.

Plus de details :

```
$ hg help commit
```

#### 4.6 Pull, Push et Merge

L'opération **pull** permet de synchroniser un repository local avec un autre repository. Cette opération fonctionne aussi bien en local que par ssh, http ou https. Le **push** est l'opération inverse, on synchronise un repository distant avec son repository local. Si les deux repository ont subi des développements en parallèle, le pull/push créera une branche divergente à partir de la dernière révision commune. Il existera alors plusieurs heads dans le projet. Ces heads peuvent être mergées grâce à l'opération **merge**.

Plus de details :

```
$ hg help pull
$ hg help push
$ hg help merge
```

## Deuxième partie

# Recettes

## 5 Configurer son *.hgrc*

Le fichier de configuration *.hgrc* influe sur le comportement général de Mercurial et permet de configurer certaines fonctionnalités et extensions. Un fichier *.hgrc* doit se trouver dans  $\tilde{/}$  et comporter au minimum :

```
[ui]
username = "Adrien Herubel <herubel@gmail.com>"
```

Pour plus d'informations sur les options, consulter le man *hgrc* et le fichier */usr/share/doc/mercurial/examples/sample.hgrc* installé par le paquet *debian/ubuntu*.

## 6 Commencer un nouveau projet avec Mercurial.

Il suffit d'une seule commande pour transformer un repertoire en repository Mercurial. La création d'un nouveau projet utilisant Mercurial est particulièrement aisée :

```
$ mkdir myHelloWorld && cd myHelloWorld
$ hg init
```

La commande `hg init` permet d'initialiser le repository mercurial (voir `hg help init` pour les options). Elle construit le store :

```
$ls -la
drwxr-xr-x  3  ———  ———  4096 8888-88-88 88:88  .hg
```

La réciproque est aussi vraie, pour supprimer le support de Mercurial dans un repository il suffit d'effacer le store `.hg`. On peut ensuite créer ses premiers fichiers.

```
$ echo '
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    printf("Hello, World\n!");
    return (EXIT_SUCCESS);
}
' > main.c
```

On ajoute en suite les fichiers créés grâce à la commande `hg add`. On peut contrôler le résultat grâce à la commande `hg status` qui montre les changements du répertoire de travail par rapport à la dernière révision.

```
$ hg add main.c
$ hg status
A main.c
```

`hg status` nous indique que `main.c` a été ajouté. On utilise ensuite `hg commit` pour valider les changements et créer la première révision du projet.

```
$ hg commit -m 'Creation du projet'
```

## 7 Développer un projet avec Mercurial

Cette recette explique les commandes et les procédures d'usage quotidiennes de Mercurial : la gestion des fichiers du repository, la sauvegarde des modifications, l'utilisation de l'historique et la correction des erreurs.

### 7.1 Gérer le tracking des fichiers

Mercurial ne travaille pas avec les fichiers du repository tant qu'on ne lui a pas explicitement demandé de le faire. La commande `hg status` informe des fichiers du repository non traités en les marquant avec un `?`

```
$hg status
? main.c
```

Afin de spécifier à Mercurial de gérer un fichier il faut utiliser la commande `add` comme vu dans la recette précédente.

```
$hg add main.c
$hg status
A main.c
```

On remarque que `status` signale maintenant le fichier `main.c` comme ayant été ajouté.

Il est important de noter que Mercurial traite des fichiers et non des répertoires. Si l'on passe des répertoires et non des fichiers à la commande `add`, celle-ci traitera récursivement les fichiers inclus dans les répertoires. Si l'on veut que Mercurial traite un répertoire vide la seule méthode est d'ajouter un fichier invisible et de l'ajouter.

Si l'on veut que Mercurial arrête de gérer un fichier il faut utiliser la commande `remove`.

```
$hg remove main.c
$hg status
R main.c
```

Le fichier est alors marqué R par *status*. Les modifications du fichiers ne seront plus suivies par Mercurial, le fichier peut être effacé et recréé puis ajouté à nouveau il ne sera pas lié à l'ancienne version. Si un fichier toujours géré par Mercurial est effacé, il est marqué comme manquant.

```
$rm main.c
$hg status
! main.c
```

Ces commandes n'ont aucune incidence sur l'historique du fichier. En cas de retour en arrière dans les versions du projet, le fichier sera présent.

## 7.2 Renommer les fichiers

L'opération de renommage de Mercurial s'appuie sur deux autres opérations, la suppression et la copie.

```
$ls
main.c
$hg rename main.c myHelloWorld.c
$ls
myHelloWorld.c
$hg status
A myHelloWorld.c
R main.c
$hg status -C
A myHelloWorld.c
  main.c
R main.c
```

Lors de l'utilisation de l'opération *rename* Mercurial a fait une copie du fichier *main.c* et l'a appelé *myHelloWorld.c*. Il a ensuite marqué *main.c* comme effacé et *myHelloWorld.c* comme ajouté. Mercurial garde une trace de la source d'une copie. L'option *-C* de la commande *status* permet de trouver la source d'un fichier copié. Ainsi *myHelloWorld.c* possède son historique propre et celui de sa source.

## 7.3 Commiter ses modifications

La commande *status* permet de lister tous les fichiers modifiés depuis la dernière révision : elle permet de voir les fichiers ajoutés, supprimés, manquants et modifiés. Le groupe de modifications consultable par la commande *status* est un changeset. Ce changeset doit être commité pour devenir une nouvelle révision. La commande *commit* permet d'enregistrer le changeset. Cette commande s'accompagne par la rédaction d'un message de commit. Seule la première ligne d'un message de commit est affichée par défaut, le message rédigé doit donc comprendre une première ligne de résumé indépendante du reste du message. Le message est rédigé avec l'éditeur décrit dans le fichier *.hgrc* ou bien dans la variable d'environnement *EDITOR* ou dans *vi* par défaut. Enfin l'option *-m* permet de passer directement le message de commit en ligne de commande.

```
$ hg commit -m 'Creation du projet'
```

## 7.4 Consulter l'historique

La commande *status* permet de consulter l'état des fichiers du répertoire de travail. Après un commit le répertoire de travail est identique à la dernière révision et la commande *status* ne montre plus rien. C'est la commande *log* qui va permettre de consulter l'historique du projet. Par défaut elle se contente d'afficher l'ensemble des révisions et pour chaque révision, l'id de changeset, l'utilisateur, la date du commit ainsi que la première ligne du message de commit.

```
$ hg log
changeset: 1:1e9ce0993cc1
tag: tip
user: "Adrien Herubel <herubel@gmail.com>"
date: Wed Jul 25 11:54:16 2007 +0200
summary: Correction syntaxique

changeset: 0:a42bff8275ac
user: "Adrien Herubel <herubel@gmail.com>"
date: Tue Jul 24 18:37:25 2007 +0200
```

```
summary:      Ajout du fichier
```

La commande *log* accepte de nombreuses options, l'option *-r <revision number>* permet de consulter une ou plusieurs révisions précises. L'option *-v* ou *-verbose* donne plus d'informations sur le commit et l'option *-p* ou *-patch* affiche le diff unifié entre les deux dernières révisions.

```
$ hg log -v -r1 -p
changeset:   1:1e9ce0993cc1
tag:         tip
user:        "Adrien Herubel <herubel@gmail.com>"
date:        Wed Jul 25 11:54:16 2007 +0200
files:       main.c
description:
Correction syntaxique

diff -r a42bff8275ac -r 1e9ce0993cc1 main.c
--- a/main.c  Tue Jul 24 18:37:25 2007 +0200
+++ b/main.c  Wed Jul 25 11:54:16 2007 +0200
@@ -1,8 +1,8 @@

-#include <stdio . h>
-#include <stdlib . h>
+#include <stdio.h>
+#include <stdlib.h>
  int main( int argc , char **argv ) {
-printf ( Hello , World\n ! ) ;
-return (EXIT_SUCCESS) ;
+printf ( "Hello , World!\n" );
+return (EXIT_SUCCESS);
  }
```

La commande *hg diff* permet d'afficher un fichier diff en réglant plus finement les paramètres d'affichage et de révision. La commande *tip* fonctionne de la même manière que *log* sauf qu'elle ne s'applique qu'au *tip* c'est à dire la dernière révision, cette commande est particulièrement utile après un commit.

## 7.5 Revenir en arrière et corriger ses erreurs

### 7.5.1 Annuler un changement pas encore commité

Si un changement dans le répertoire de travail doit être annulé et que celui-ci n'a pas encore été commité , il est possible de faire revenir le repository, ou un seul fichier, à l'état de la dernière révision grâce à la commande *revert*. Par exemple si un fichier a été ajouté avec *add*, modifié ou effacé par mégarde :

```
$ ls
myHelloWorld.c
$ rm myHelloWorld.c
$ hg revert myHelloWorld.c
$ ls
myHelloWorld.c
```

L'option *-all* permet de replacer l'ensemble du répertoire de travail dans l'état de la dernière révision. Attention il est impossible d'annuler un *revert*.

### 7.5.2 Annuler un changement déjà commité

Mercurial fournit une commande simple permettant d'annuler un commit accidentel. Par exemple, s'il l'on crée un fichier et qu'on commit les changements en ayant oublier d'utiliser *add* sur le fichier, il est possible d'annuler le commit grâce à la commande *rollback*. Cette commande peut annuler un et un seul *commit*, *pull* ou *import*. Attention il est impossible d'annuler un *rollback*. Mercurial fournit de nombreuses commandes de retour et d'annulation de changements. La commande *revert* précédemment vue permet aussi de remonter dans l'historique des révision grâce à l'option *-r*. La commande *backout* est extrêmement efficace. Soient trois changements a, b et c. La commande *backout* va permettre d'annuler le changement b sans annuler les changements a et c. Elle va en fait créer un autre changement inverse à b qui va annuler ses effets.

### 7.5.3 Voyager dans le temps

Il est souvent utile de faire revenir le répertoire de travail à l'état ou il était à telle ou telle version, notamment pour la recherche d'anciens bugs. Mercurial fournit la commande *update* qui permet de faire revenir le répertoire de travail à la révision souhaitée. Cette opération est non destructrice, elle ne modifie à aucun moment l'historique.

```
$ ls
myHelloWorld.c
$ hg update -r 0
$ ls
main.c
$ hg update -r tip
$ ls
myHelloWorld.c
```

A noter qu'il est possible de modifier les fichiers alors qu'on est en update dans une ancienne révision. Le commit d'un tel changeset créera une nouvelle head sur laquelle pourront se greffer d'autres développements. Les deux heads pourront bien sûr être mergées.

## 8 Participer à un projet avec mainteneur central

L'avantage des systèmes de gestion de versions distribués, c'est qu'ils laissent au développeur le soin de décider eux-même de l'architecture du développement. On s'intéressera dans cette recette au projet avec mainteneur central. Un développeur met à disposition la version officielle du projet, quelques développeurs ont accès au repository, les autres envoient des patches au mainteneur central.

### 8.1 Récupérer le projet

La récupération d'un repository existant s'effectue à travers la commande *clone*. Les clones peuvent se faire en local, auquel cas Mercurial utilisera des liens en dur pour effectuer un clonage rapide et peu coûteux en espace disque (équivalent à *cp -al*), au travers d'ssh, par le mini-serveur http *hg serve* ou encore par cgi avec Apache et *hgweb.cgi*.

```
$ hg clone myHelloWorld myHelloWorld-dev
$ hg clone ssh://login@myhelloworld.org:22/repository/myhelloworld
$ hg clone http://hg.myhelloworld.com/myhelloworld
```

### 8.2 Mettre à jour son repository

Lorsque d'un développeur travaille sur son propre clone du repository central, il est nécessaire d'effectuer régulièrement des *pull* du repository central afin d'inclure les mises à jour dans sa propre version. Cela facilite grandement le travail de mainteneur central, car la plupart des conflits lors des merge sont ainsi résolus par les développeurs. La commande *pull* est semblable par la syntaxe à la commande *clone* et fonctionne sur les mêmes supports.

```
$ hg pull ../myHelloWorld-dev
pulling from ../myHelloWorld-dev
searching for changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 2 changes to 2 files
(run 'hg update' to get a working copy)
```

À noter que la commande *pull* ne modifie pas elle-même le répertoire de travail, notamment pour éviter des problèmes liés à un répertoire se trouvant dans un état antérieur suite à une commande *update*. Afin d'appliquer les changements au répertoire de travail, il faut utiliser la commande *hg update* qui remplacera le répertoire dans l'état du tip. Si des développements ont eu lieu en parallèle, le *pull* créera une nouvelle head. Les deux heads pourront être mergées et les conflits résolus grâce à la commande *hg merge*. La commande *pull* peut être annulée grâce à *rollback*.

## 8.3 Partager ses modifications

### 8.3.1 La méthode push

La commande *push* est la réciproque de *pull* : elle permet d'envoyer ses changements vers un repository distant. Dans notre exemple d'architecture de développement, quelques développeurs privilégiés ont les droits d'accès en ssh sur le repository central et effectuent des *pushs* directement. Par défaut, la commande *push* interdit la création de nouvelles head dans le repository distant (ce qui est plutôt une bonne chose), car Mercurial préconise une approche où le développeur résout les conflits locaux avant d'envoyer ses modifications. Attention la commande *push* ne peut pas être annulée par un *rollback*.

### 8.3.2 Envoyer des patches

Pour le reste du monde, l'accès ssh n'est pas disponible. Seule reste la possibilité d'envoyer des patches par mail au mainteneur ou à l'un des développeurs privilégiés. La commande *export* est utilisée pour produire des patches. Elle prend en argument la ou les révisions qui vont constituer le patch.

```
$ hg export 17 18 19 > mypatch.patch
```

La même règle s'applique que pour les push : un patch ne doit pas créer de nouvelles head sur le repository du mainteneur. Le développeur qui fournit le patch résout les conflits locaux avant la soumission.

## 9 Maintenir un projet multi-développeurs

### 9.1 Publier le projet en ssh

La publication par ssh permet de publier le projet en lecture et écriture pour les développeurs et mainteneurs principaux du projet. Chaque développeur doit bénéficier d'un compte sur la machine hébergeant le serveur ssh, ainsi qu'un accès en lecture et écriture sur le repository. Il est conseillé de générer une paire de clé d'identification afin d'éviter de devoir retaper un mot de passe à chaque *pull*. La syntaxe des URL ssh est la suivante :

```
ssh://login@server:port/path
```

### 9.2 Publier le projet en http

La publication par http permet de fournir rapidement sur un protocole standard un accès en lecture au repository et une interface web de consultation. Mercurial fournit un petit serveur http basique pour publier un repository :

```
$ hg serve -p 8000
```

Pour un usage plus soutenu, il est conseillé d'utiliser *hgweb.cgi* (*/usr/share/doc/mercurial/examples/hgweb.cgi*) un script CGI qui s'utilise avec Apache. Il est pour cela nécessaire de disposer d'un serveur web type Apache ou lighttpd supportant le CGI python.

Une troisième méthode basée sur le protocole static-http permet de publier le repository sans disposer d'un serveur supportant les CGI, il suffit de placer le repository en accès sur le serveur web. Les commandes de *clone*, *push* et *pull* utiliseront alors la syntaxe suivante

```
$ hg clone static-http://server/repository
```

Cette méthode est cependant assez lente.

Enfin une dernière méthode, la plus simple, mais qui ne permettra pas les opérations de *clone*, *push* et *pull* est de diffuser une archive du repository. Attention, la commande *hg archive* ne convient pas pour cette méthode car elle crée une archive sans le store.

### 9.3 Appliquer et tester un patch

Les patches proviennent d'utilisateurs n'ayant pas d'accès ssh au repository. Les patches peuvent potentiellement être nocifs et doivent être étudiés avant application. Il existe de nombreux programmes pour lire les patches de façon graphique. La manière la plus sûre d'appliquer un patch est de cloner le repository en local et d'appliquer le patch sur le clone. On peut ensuite tester le patch puis en cas de validation faire un push dans le repository officiel.

```
$ hg clone myHelloWorld myHelloWorld-experimental
$ cd myHelloWorld-experimental
$ hg import ../patchexperimental.patch
**faire les tests**
$ hg push ../myHelloWorld
$ cd ../myHelloWorld
$ hg update
```

### 9.4 Définir des numéros de version

Il peut parfois être utile de marquer une révision de façon plus spécifique, par exemple pour la réalisation d'une branche stable ou d'une publication particulière. Mercurial fournit le mécanisme des tags conjointement aux numéros de révisions. Les tags s'utilisent de façon transparente dans les commandes Mercurial au même titre que les numéros de version. La commande *tag* permet de donner une étiquette à la révision courante et la commande *tags* permet de lister les tags du repository.

```
$hg tag v0.22
$ hg tags
tip                20:cc37a4823319
v0.22             19:4d4c066c7aae
v0.2              13:640a581737cd
v0.11            11:b7b9a085b543
v0.1              6:572f1e9be549
$ cd ../myHelloWorld-stable/
$ hg pull -rv0.22 ../myHelloWorld/
pulling from ../myHelloWorld/
searching for changes
adding changesets
adding manifests
adding file changes
added 5 changesets with 5 changes to 3 files
(run 'hg update' to get a working copy)
$ hg update
4 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Dans cette exemple la version de travail myHelloWorld est étiquetée v0.22, et la version stable récupère les modifications effectuée jusqu'à cette version.

### 9.5 Maintenir deux versions d'un projet

Il est parfois nécessaire de maintenir deux versions concurrentes d'un même projet, par exemple une version publique et une version privée contenant le code de la version publique ainsi que du code sensible. Une des approches possible est de créer un clone du repository public qui deviendra le repository privé et d'y ajouter le code sensible. Le repository public sera accessible en http tandis que l'accès ssh permettra d'accéder à la fois au repository public et au repository privé. Les patches concernant le code sensible sont uniquement appliqués au repository privé, et le mainteneur effectue régulièrement des pulls du repository public vers le privé pour récupérer les modifications concernant le code. Attention à ne pas réaliser l'opération inverse, qui ajouterait l'ensemble du code sensible au repository public.

## Références

- [1] Bryan O'Sullivan, *Distributed revision control with Mercurial* <http://hgbook.red-bean.com/>.