Learning Mercurial in Workflows

With Mercurial you can use a multitude of different workflows. This page shows some of them, including their use cases. It is intended to make it easy for beginners of version tracking to get going instantly and learn completely incrementally. It doesn't explain the concepts used, because there are already many other great resources doing that, for example the wiki and the hgbook.

If you want a more exhaustive tutorial with the basics, please have a look at the Tutorial in the Mercurial Wiki. For a really detailed and very nice to read description of Mercurial, please have a look at Mercurial: The Definitive Guide.

> Note:
>
> This guide doesn't require any prior knowledge of version control systems (though subversion users will likely feel at home quite quickly). Basic command line abilities are helpful, because we'll use the command line client.

Index

Basic workflows

We go from simple to more complex workflows. Those further down build on previous workflows.

Log keeping

Use Case

The first workflow is also the easiest one: You want to use Mercurial to be able to look back when you did which changes.

This workflow only requires an installed Mercurial and write access to some file storage (you almost definitely have that :)). It shows the basic techniques for more complex workflows.

Workflow

Prepare Mercurial

As first step, you should teach Mercurial your name. For that you open the file ~/.hgrc (or mercurial.ini in your home directory for Windows) with a text-editor and add the ui section (user interaction) with your username:

```
[ui]
username = Mr. Johnson <johnson@smith.com>
```

Initialize the project

Now you add a new folder in which you want to work:

```
$ hg init project
```

Add files and track them

Enter the project folder, create some files, then add and commit them.

```
$ cd project
$ echo 'print("Hello")' > hello.py
$ hg add
$ hg commit
(enter the commit message)
```

```
output of hg add:
adding hello.py
```

```
display of hg commit (with your message):
Initial commit.

HG: Enter commit message.  Lines beginning with
'HG:' are removed.
HG: Leave message empty to abort commit.
HG: --
HG: user: Mr. Johnson <johnson@smith.com>
HG: branch 'default'
HG: added hello.py
```

Note:

You can also go into an existing directory with files and init the repository there.

```
$ cd project
$ hg init
```

Also you can add only specific files instead of all files in the directory. Mercurial will then track only these files and won't know about the others. The following tells mercurial to track all files whose names begin with "file0" as well as file10, file11 and file12.

```
$ hg add file0* file10 file11 file12
```

Save changes

First do some changes:

```
$ echo 'print("Hello World") > hello.py
```

see which files changed, which have been added or removed, and which aren't tracked yet

```
$ hg status
```

```
output of hg status:
M hello.py
```

see the exact changes

```
$ hg diff
```

```
output of hg diff:
diff --git a/hello.py b/hello.py
--- a/hello.py
+++ b/hello.py
@@ -1,1 +1,1 @@
-print("Hello")
+print("Hello World")
```

commit the changes.

```
$ hg commit
```

now an editor pops up and asks you for a commit message. Upon saving and closing the editor, your changes have been stored by Mercurial.

```
display of hg commit (with your message):
Say Hello World, not just Hello.

HG: Enter commit message.  Lines beginning with
'HG:' are removed.
HG: Leave message empty to abort commit.
HG: --
HG: user: Mr. Johnson <johnson@smith.com>
HG: branch 'default'
HG: changed hello.py
```

> Note:
>
> You can also supply the commit message directly via *hg commit -m 'MESSAGE'*.

Copy and move files

When you copy or move files, you should tell Mercurial to do the copy or move for you, so it can track the relationship between the files.

Remember to *commit* after moving or copying. From the basic commands only *commit* creates a new revision

```
$ hg cp hello.py copy
$ hg mv hello.py target
$ hg diff # see the changes
$ hg commit
(enter the commit message)
```

Now you have two files, "copy" and "target", and Mercurial knows how they are related.

```
output of hg diff (before the commit):
diff --git a/hello.py b/copy
rename from hello.py
rename to copy
diff --git a/hello.py b/target
copy from hello.py
copy to target
```

Note:

Should you forget to do the explicit copy or move, you can still tell Mercurial to detect the changes via *hg addremove --similarity 100*. Just use *hg help addremove* for details.

Check your history

```
$ hg log
```

This prints a list of changesets along with their date, the user who committed them (you) and their commit message.

```
output of hg log:
changeset:   2:70eb0ca9d264
tag:         tip
user:        Mr. Johnson
date:        Sun Nov 20 11:20:00 2011 +0100
summary:     Copy and move.

changeset:   1:487d7a20ccbc
user:        Mr. Johnson
date:        Sun Nov 20 11:11:00 2011 +0100
summary:     Say Hello World, not just Hello.

changeset:   0:a5ecbf5799c8
user:        Mr. Johnson
date:        Sun Nov 20 11:00:00 2011 +0100
summary:     Initial commit.
```

To see a certain revision, you can use the *-r* switch (--revision). To also see the diff of the displayed revisions, there's the *-p* switch (--patch)

```
$ hg log -p -r 3
```

Lone developer with nonlinear history

Use case

The second workflow is still very easy: You're a lone developer and you want to use Mercurial to keep track of your own changes.

It works just like the log keeping workflow, with the difference that you go back to earlier changes at times.

To start a new project, you initialize a repository, add your files and commit whenever you finished a part of your work.

Also you check your history from time to time, so see how you progressed.

Workflow

Basics from log keeping

Init your project, add files, see changes and commit them.

```
$ hg init project
$ cd project
```

```
$ (add files)
$ hg add # tell Mercurial to track all files
$ (do some changes)
$ hg diff # see changes
$ hg commit # save changes
$ hg cp # copy files or folders
$ hg mv # move files or folders
$ hg log # see history
```

Seeing an earlier revision

Different from the log keeping workflow, you'll want to go back in history at times and do some changes directly there, for example because an earlier change introduced a bug and you want to fix it where it occurred.

To look at a previous version of your code, you can use update. Let's assume that you want to see revision 1.

```
$ hg update 1
```

Now your code is back at revision 1, the second commit (Mercurial starts counting at 0). To check if you're really at that revision, you can use *identify -n*.

```
$ hg identify -n
```

```
output of hg identify -n:
1
```

```
output of ls:
hello.py
```

Note:

identify without options gives you the short form of a unique revision ID. That ID is what Mercurial uses internally. If you tell someone about the version you updated to, you should use that ID, since the numbers can be different for other people. If you want to know the reasons behind that, please read up Mercurials [basic concepts]. When you're at the most recent revision, *hg identify -n* will return "-1".

To update to the most recent revision, you can use "tip" as revision name.

```
$ hg update tip
```

Note:

If at any place any command complains, your best bet is to read what it tells you and follow that advice.

Note:

Instead of *hg update* you can also use the shorthand *hg up*. Similarly you can abbreviate *hg commit* to *hg ci*.

Note:

To get a revision devoid of files, just *update* to "null" via *hg update null*. That's the revision before any files were added.

Note:

If the output of *hg identify* ends in a “+”, your repository has uncommitted changes.

Fixing errors in earlier revisions

When you find a bug in some earlier revision you have two options: either you can fix it in the current code, or you can go back in history and fix the code exactly where you did it, which creates a cleaner history.

To do it the cleaner way, you first update to the old revision, fix the bug and commit it. Afterwards you merge this revision and commit the merge. Don't worry, though: Merging in mercurial is fast and painless, as you'll see in an instant.

Let's assume the bug was introduced in revision 1.

```
$ hg update 1
$ echo 'print("Hello Mercurial")' > hello.py
$ hg commit
```

```
output of hg commit (after entering the message):
created new head
```

Note:

> “created new head” means that there is now one more revision, which does not have children. Heads are current states of your project living side by side. It is good style to merge them together before propagating them.

Now the fix is already stored in history. We just need to merge it with the current version of your code.

```
$ hg merge
```

```
output of hg merge (here):
merging hello.py and copy to copy
merging hello.py and target to target
```

If there are conflicts use *hg resolve* - that's also what merge tells you to do in case of conflicts.

First list the files with conflicts

```
$ hg resolve --list
```

Then resolve them one by one. *resolve* attempts the merge again

```
$ hg resolve conflicting_file
(fix it by hand, if necessary)
```

> Note:
>
> For more details on resolving conflicts, see the wiki-page

TutorialConflict.

Mark the fixed file as resolved

```
$ hg resolve --mark conflicting_file
```

Commit the merge, as soon as you resolved all conflicts. This step is also necessary when there were no conflicts!

```
$ hg commit
```

At this point, your fix is merged with all your other work, and you can just go on coding. Additionally the history shows clearly where you fixed the bug, so you'll always be able to check where the bug was.

```
output of hg log (after the final commit):
changeset:   4:3b06bba7c1a9
tag:         tip
parent:      3:7ff5cd572d80
parent:      2:70eb0ca9d264
user:        Mr. Johnson
date:        Sun Nov 20 20:11:00 2011 +0100
summary:     merge greeting and copy+move.

changeset:   3:7ff5cd572d80
parent:      1:487d7a20ccbc
user:        Mr. Johnson
date:        Sun Nov 20 20:00:00 2011 +0100
summary:     Greet Mercurial
```

```
changeset:   2:70eb0ca9d264
user:        Mr. Johnson
date:        Sun Nov 20 11:20:00 2011 +0100
summary:     Copy and move.

changeset:   1:487d7a20ccbc
user:        Mr. Johnson
date:        Sun Nov 20 11:11:00 2011 +0100
summary:     Say Hello World, not just Hello.

changeset:   0:a5ecbf5799c8
user:        Mr. Johnson
date:        Sun Nov 20 11:00:00 2011 +0100
summary:     Initial commit.
```

> Note:
>
> Most merges will just work. You only need *resolve*, when *merge* complains.

So now you can initialize repositories, save changes, update to previous changes and develop in a nonlinear history by committing in earlier changesets and merging the changes into the current code.

> Note:
>
> If you fix a bug in an earlier revision, and some later revision

copied or moved that file, the fix will be propagated to the target file(s) when you merge. This is the main reason why you should always use *hg cp* and *hg mv*.

Separate features

Use Case

At times you'll be working on several features in parallel. If you want to avoid mixing incomplete code versions, you can create clones of your local repository and work on each feature in its own code directory.

After finishing your feature you then *pull* it back into your main directory and *merge* the changes.

Workflow

Work in different clones

First create the feature clone and do some changes

```
$ hg clone project feature1
$ cd feature1
$ hg update 3
$ echo 'print("Hello feature1")' > hello.py
$ hg commit -m "Greet feature1"
```

Now check what will come in when you *pull* from feature1, just like you can use *diff* before committing. The respective command for

pulling is *incoming*

```
$ cd ../project
$ hg incoming ../feature1
```

```
output of hg incoming ../feature1:
comparing with ../feature1
searching for changes
changeset:   5:3eb7b39fcf57
tag:         tip
parent:      3:7ff5cd572d80
user:        Arne Babenhauserheide
date:        Sun Nov 20 20:11:11 2011 +0100
summary:     Greet feature1
```

> Note:
>
> If you want to see the diffs, you can use *hg incoming --patch* just as you can do with *hg log --patch* for the changes in the repository.

If you like the changes, you pull them into the project

```
$ hg pull ../feature1
```

Now you have the history of feature1 inside your project, but the changes aren't yet visible. Instead they are only stored inside a ".hg" directory of the project (more information on the store).

```
output of hg pull:
pulling from ../feature1
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1
heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
```

> Note:
>
> From now on we'll use the name "repository" for a directory which has a .hg directory with Mercurial history.

If you didn't do any changes in the project, while you were working on feature1, you can just update to tip (*hg update tip*), but it is more likely that you'll have done some other changes in between changes. In that case, it's time for merging.

Merge feature1 into the project code

```
$ hg merge
```

If there are conflicts use *hg resolve* - that's also what merge tells you to do in case of conflicts. After you *merge*, you have to *commit* explicitly to make your *merge* final

```
$ hg commit
(enter commit message, for example "merged feature1")
```

```
output of hg log -r -1:-3 (the last 3 changesets):
changeset:   6:e8a33691171a
tag:         tip
parent:      4:3b06bba7c1a9
parent:      5:3eb7b39fcf57
user:        Mr. Johnson
date:        Sun Nov 20 20:20:00 2011 +0100
summary:     merged feature1

changeset:   5:3eb7b39fcf57
parent:      3:7ff5cd572d80
user:        Arne Babenhauserheide
date:        Sun Nov 20 20:11:11 2011 +0100
summary:     Greet feature1

changeset:   4:3b06bba7c1a9
parent:      3:7ff5cd572d80
parent:      2:70eb0ca9d264
user:        Mr. Johnson
date:        Sun Nov 20 20:11:00 2011 +0100
summary:     merge greeting and copy+move.
```

Note:

Mercurial offers powerful ways specify revisions. Too see them all, use *hg help revsets*.

You can create an arbitrary number of clones and also carry them around on USB sticks. Also you can use them to synchronize your files at home and at work, or between your desktop and your laptop.

> Note:
>
> You also have to commit after a merge when there are no conflicts, because merging creates new history and you might want to attach a specific message to the merge (like "merge feature1").

Rollback mistakes

Now you can work on different features in parallel, but from time to time a bad commit might sneak in. Naturally you could then just go back one revision and merge the stray error, keeping all mistakes out of the merged revision. However, there's an easier way, if you realize your error before you do another *commit* or *pull*: *rollback*.

Rolling back means undoing the last operation which added something to your history.

Imagine you just realized that you did a bad commit - for example you didn't see a spelling error in a commit message. To fix it you would use

```
$ hg rollback
```

```
output of hg rollback:
repository tip rolled back to revision 5 (undo
```

```
commit)
working directory now based on revisions 4 and 5
```

And then redo the commit

```
$ hg commit -m "Merged Feature 1"
```

```
output of hg log -r -1:-2 (after rollback and
commit):
changeset:   6:3f549b33c7ef
tag:         tip
parent:      4:3b06bba7c1a9
parent:      5:3eb7b39fcf57
user:        Mr. Johnson
date:        Sun Nov 20 20:20:11 2011 +1100
summary:     Merged Feature 1

changeset:   5:3eb7b39fcf57
parent:      3:7ff5cd572d80
user:        Arne Babenhauserheide
date:        Sun Nov 20 20:11:11 2011 +0100
summary:     Greet feature1
```

If you can use the command history of your shell and you added the previous message via *commit -m "message"*, that following commit just means two clicks on the arrow-key "up" and one click on "enter".

Though it changes your history, rolling back doesn't change your files. It only undoes the last addition to your history.

But beware, that a rollback itself can't be undone. If you *rollback* and then forget to commit, you can't just say "give me my old commit back". You have to create a new commit.

> Note:
>
> Rollback is possible, because Mercurial uses transactions when recording changes, and you can use the transaction record to undo the last transaction. This means that you can also use *rollback* to undo your last *pull*, if you didn't yet commit anything new.

Sharing changes

Use Case

Now we go one step further: You are no longer alone, and you want to share your changes with others and include their changes.

The basic requirement for that is that you have to be able to see the changes of others.

Mercurial allows you to do that very easily by including a simple webserver from which you can *pull* changes just as you can pull changes from local clones.

> Note:
>
> There are a few other ways to share changes, though. Instead of

using the builtin webserver, you can also send the changes by email or setup a shared repository, to where you *push* changes instead of pulling them. We'll cover one of those later.

Workflow

Using the builtin webserver

This is the easiest way to quickly share changes.

First the one who wants to share his changes creates the webserver

```
$ hg serve
```

Now all others can point their browsers to his IP address (for example 192.168.178.100) at port 8000. They will then see all his history there and can decide if they want to pull his changes.

```
$ firefox http://192.168.178.100:8000
```

If they decide to include the changes, they just pull from the same URL

```
$ hg pull http://192.168.178.100:8000
```

At this point you all can work as if you had pulled from a local repository. All the data is now in your individual repositories and you can merge the changes and work with them without needing any connection to the served repository.

Sending changes by email

Often you won't have direct access to the repository of someone else, be it because he's behind a restrictive firewall, or because you live in different timezones. You might also want to keep your changes confidential and prefer internal email (if you want additional protection, you can also encrypt the emails, for example with GnuPG).

In that case, you can easily export your changes as patches and send them by email.

Another reason to send them by email can be that your policy requires manual review of the changes when the other developers are used to reading diffs in emails. I'm sure you can think of more reasons.

Sending the changes via email is pretty straightforward with Mercurial. You just *export* your changes and attach (or copy paste) it in your email. Your colleagues can then just *import* them.

First check which changes you want to export

```
$ cd project
$ hg log
```

We assume that you want to export changeset 3 and 4

```
$ hg export 3 > change3.diff
$ hg export 4 > change4.diff
```

Now attach them to an email and your colleagues can just run *import*

on both diffs to get your full changes, including your user information.

To be careful, they first *clone* their repository to have an integration directory as sandbox

```
$ hg clone project integration
$ cd integration
$ hg import change3.diff
$ hg import change4.diff
```

That's it. They can now test your changes in feature clones. If they accept them, they *pull* the changes into the main repository

```
$ cd ../project
$ hg pull ../integration
```

Note:

The *patchbomb* extension automates the email-sending, but you don't need it for this workflow.

Note:

You can also send around bundles, which are snippets of your actual history. Just create them via
`$ hg bundle --base FIRST_REVISION_TO_BUNDLE changes.bundle`

Others can then get your changes by simply pulling them, as if your bundle were an actual repository
`$ hg pull path/to/changes.bundle`

Using a shared repository

Sending changes by email might be the easiest way to reach people when you aren't yet part of the regular development team, but it creates additional workload: You have to *bundle* the changes, send mails and then *import* the bundles manually. Luckily there's an easier way which works quite well: The shared push repository.

Till now we transferred all changes either via email or via *pull*. Now we use another option: pushing. As the name suggests it's just the opposite of pulling: You *push* your changes into another repository.

But to make use of it, we first need something we can push to.

By default *hg serve* doesn't allow pushing, since that would be a major security hole. You can allow pushing in the server, but that's no solution when you live in different timezones, so we'll go with another approach here: Using a shared repository, either on an existing shared server or on a service like BitBucket. Doing so has a bit higher starting cost and takes a bit longer to explain, but it's well worth the effort spent.

If you want to use an existing shared server, you can use *serve* there and allow pushing. Also there are some other nice ways to allow pushing to a Mercurial repository, including simple access via SSH.

Otherwise you first need to setup a BitBucket Account. Just signup at BitBucket. After signing up (and login) hover your mouse over "Repositories". There click the item at the bottom of the opening

dialog which say "Create new".

Give it a name and a description. If you want to keep it hidden from the public, select "private"

```
$ firefox http://bitbucket.org
```

Now your repository is created and you see instructions for *push*ing to it. For that you'll use a command similar to the following (just with a different URL):

```
$ hg push https://bitbucket.org/ArneBab/hello/
```

(Replace the URL with the URL of your created repository. If your username is "Foo" and your repository is named "bar", the URL will be https://bitbucket.org/Foo/bar/)

Mercurial will ask for your BitBucket name and password, then *push* your code.

Voilà, your code is online.

To see what you would get if you would push, you can use outgoing. It works with local repositories in the same way as with shared ones, so you can test it with a local one:

```
$ hg outgoing ../feature1
```

```
output of hg outgoing ../feature1 (our feature
seperation repo):
```

```
comparing with ../feature1
searching for changes
changeset:   6:3f549b33c7ef
tag:         tip
parent:      4:3b06bba7c1a9
parent:      5:3eb7b39fcf57
user:        Mr. Johnson
date:        Sun Nov 20 20:20:11 2011 +1100
summary:     Merged Feature 1
```

Note:

You can also use SSH for pushing to BitBucket.

Now it's time to tell all your colleagues to sign up at BitBucket, too.

After that you can click the "Admin" tab of your created repository and add the usernames of your colleagues on the right side under "Permission: Writers". Now they are allowed to *push* code to the repository.

(If you chose to make the repository private, you'll need to add them to "Permission: Readers", too)

If one of you now wants to publish changes, he'll simply *push* them to the repository, and all others get them by *pull*ing.

Publish your changes

```
$ hg push https://bitbucket.org/ArneBab/hello/
```

Pull others changes into your local repository

```
$ hg pull https://bitbucket.org/ArneBab/hello/
```

People who join you in development can also just *clone* this repository, as if one of you were using *hg serve*

```
$ hg clone https://bitbucket.org/ArneBab/hello/ hello
```

That local repository will automatically be configured to pull/push from/to the online repository, so new contributors can just use *hg push* and *hg pull* without an URL.

Note:

To make this workflow more scalable, each one of you can have his own BitBucket repository and you can simply *pull* from the others repositories. That way you can easily establish workflows in which certain people act as integrators and finally *push* checked code to a shared pull repository from which all others pull.

Note:

You can also use this workflow with a shared server instead of BitBucket, either via SSH or via a shared directory. An example for an SSH URL with Mercurial is be ssh://user@example.com/path/to/repo. When using a shared directory you just push as if the repository in the shared directory

were on your local drive.

Summary

Now let's take a step back and look where we are.

With the commands you already know, a bit reading of *hg help <command>* and some evil script-fu you can already do almost everything you'll ever need to do when working with source code history. So from now on almost everything is convenience, and that's a good thing.

First this is good, because it means, that you can now use most of the concepts which are utilized in more complex workflows.

Second it aids you, because convenience lets you focus on your task instead of focusing on your tool. It helps you concentrate on the coding itself. Still you can always go back to the basics, if you want to.

A short summary of what you can do which can also act as a short check, if you still remember the meaning of the commands.

create a project

```
$ hg init project
$ cd project
$ (add some files)
$ hg add
$ hg commit
```

```
(enter the commit message)
```

do nonlinear development

```
$ (do some changes)
$ hg commit
(enter the commit message)
$ hg update 0
$ (do some changes)
$ hg commit
(enter the commit message)
$ hg merge
$ (optionally hg resolve)
$ hg commit
(enter the commit message)
```

use feature clones

```
$ cd ..
$ hg clone project feature1
$ cd feature1
$ (do some changes)
$ hg commit
(enter the commit message)
$ cd ../project
$ hg pull ../feature1
```

share your repository via the integrated webserver

```
$ hg serve &
$ cd ..
$ hg clone http://127.0.0.1:8000 project-clone
```

export changes to files

```
$ cd project-clone
$ (do some changes)
$ hg commit
(enter the commit message)
$ hg export tip > ../changes.diff
```

import changes from files

```
$ cd ../project
$ hg import ../changes.diff
```

pull changes from a served repository (hg serve still runs on project)

```
$ cd ../feature1
$ hg pull http://127.0.0.1:8000
```

Use shared repositories on BitBucket

```
$ (setup bitbucket repo)
```

```
$ hg push https://bitbucket.org/USER/REPO
(enter name and password in the prompt)
$ hg pull https://bitbucket.org/USER/REPO
```

Let's move on towards useful features and a bit more advanced workflows.

Advanced workflows

Backing out bad revisions

Use Case

When you routinely pull code from others, it can happen that you overlook some bad change. As soon as others pull that change from you, you have little chance to get completely rid of it.

To resolve that problem, Mercurial offers you the *backout* command. Backing out a change means, that you tell Mercurial to create a commit which reverses the bad change. That way you don't get rid of the bad code in history, but you can remove it from new revisions.

> Note:
>
> The basic commands don't directly rewrite history. If you want to do that, you need to activate some of the extensions which are shipped with mercurial. We'll come to that later on.

Workflow

Let's assume the bad change was revision 3, and you already have one more revision in your repository. To remove the bad code, you can just *backout* of it. This creates a new change which reverses the bad change. After backing out, you can then merge that new change into the current code.

```
$ hg backout 3
$ hg merge
(potentially resolve conflicts)
$ hg commit
(enter commit message. For example: "merged backout")
```

That's it. You reversed the bad change. It's still recorded that it was once there (following the principle "don't rewrite history, if it's not really necessary"), but it doesn't affect future code anymore.

Collaborative feature development

Now that you can share changes and reverse them if necessary, you can go one step further: Using Mercurial to help in coordinating the coding.

The first part is an easy way to develop features together, without requiring every developer to keep track of several feature clones.

Use Case

When you want to split your development into several features, you need to keep track of who works on which feature and where to get which changes.

Mercurial makes this easy for you by providing named branches. They are a part of the main repository, so they are available to everyone involved. At the same time, changes committed on a certain branch don't get mixed with the changes in the default branch, so features are kept separate, until they get merged into the default branch.

Note:

Cloning a repository always puts you onto the default branch at first.

Workflow

When someone in your group wants to start coding on a feature without disturbing the others, he can create a named branch and commit there. When someone else wants to join in, he just updates to the branch and commits away. As soon as the feature is finished, someone merges the named branch into the default branch.

Working in a named branch

Create the branch

```
$ hg branch feature1
(do some changes)
```

```
$ hg commit
(write commit message)
```

Update into the branch and work in it

```
$ hg update feature1
(do some changes)
$ hg commit
(write commit message)
```

Now you can *commit*, *pull*, *push* and *merge* (and anything else) as if you were working in a separate repository. If the history of the named branch is linear and you call "hg merge", Mercurial asks you to specify an explicit revision, since the branch in which you work doesn't have anything to merge.

Merge the named branch

When you finished the feature, you *merge* the branch back into the default branch.

```
$ hg update default
$ hg merge feature1
$ hg commit
(write commit message)
```

To see the active branches with *hg branches*. To mark a branch as inactive, for example because you finished implementing the feature, you can close it. Closed branches are hidden in *hg branches* as well as in *hg heads*.

```
$ hg update feature1
$ hg commit --close-branch -m "finished feature1"
```

And that's it. Now you can easily keep features separate without unnecessary bookkeeping.

Note:

Named branches stay in history as permanent record after you finished your work. If you don't like having that record in your history, please have a look at some of the advanced workflows.

Tagging revisions

Use Case

Since you can now code separate features more easily, you might want to mark certain revisions as fit for consumption (or similar). For example you might want to mark releases, or just mark off revisions as reviewed.

For this Mercurial offers tags. Tags add a name to a revision and are part of the history. You can tag a change years after it was committed. The tag includes the information when it was added, and tags can be pulled, pushed and merged just like any other committed change.

Note:

A tag must not contain the char ":", since that char is used for specifying multiple revisions - see "hg help revisions".

Note:

To securely mark a revision, you can use the gpg extension to sign the tag.

Workflow

Let's assume you want to give revision 3 the name "v0.1".

Add the tag

```
$ hg tag -r 3 v0.1
```

See all tags

```
$ hg tags
```

When you look at the log you'll now see a line in changeset 3 which marks the Tag. If someone wants to *update* to the tagged revision, he can just use the name of your tag

```
$ hg update v0.1
```

Now he'll be at the tagged revision and can work from there.

Removing history

Use Case

At times you will have changes in your repository, which you really don't want in it.

There are many advanced options for removing these, and most of them use great extensions (Mercurial Queues is the most often used one), but in this basic guide, we'll solve the problem with just the commands we already learned. But we'll use an option to clone which we didn't yet need.

This workflow becomes inconvenient when you need to remove changes, which are buried below many new changes. If you spot the bad changes early enough, you can get rid of them without too much effort, though.

Workflow

Let's assume you want to get rid of revision 2 and the highest revision is 3.

The first step is to use the "--rev" option to *clone*: Create a clone which only contains the changes up to the specified revision. Since you want to keep revision 1, you only clone up to that

```
$ hg clone -r 1 project stripped
```

Now you can *export* the change 3 from the original repository (project) and *import* it into the stripped one

```
$ cd project
$ hg export 3 > ../changes.diff
$ cd ../stripped
$ hg import ../changes.diff
```

If a part of the changes couldn't be applied, you'll see that part in *.rej files. If you have *.rej files, you'll have to include or discard changes by hand

```
$ cat *.rej
(apply changes by hand)
$ hg commit
(write commit message)
```

That's it. *hg export* also includes the commit message, date, committer and similar metadata, so you are already done.

> Note:
>
> removing history will change the revision IDs of revisions after the removed one, and if you pull from someone else who still has the revision you removed, you will pull the removed parts again. That's why rewriting history should most times only be done for changes which you didn't yet publicise.

Summary

So now you can work with Mercurial in private, and also share your changes in a multitude of ways.

Additionally you can remove bad changes, either by creating a change in the repository which reverses the original change, or by really rewriting history, so it looks like the change never occurred.

And you can separate the work on features in a single repository by using named branches and add tags to revisions which are visible markers for others and can be used to update to the tagged revisions.

With this we can conclude our practical guide.

More Complex Workflows

If you now want to check some more complex workflows, please have a look at the general workflows wikipage and the list of extensions.

To deepen your understanding, you should also check the basic concept overview.

Have fun with Mercurial!