Using Mercurial and/or Git

Maxime Charlebois
Sherbrooke
December 2017

Contents

1 Introduction to mercurial and git

Most systems already have mercurial (hg) and git preinstalled. If not, you will need to install them. For example, on Ubuntu 16.04:

```
$ sudo apt install mercurial
$ sudo apt install git
```

or on MacOs:

```
$ brew install mercurial
$ brew install git
```

In this document, we use "$" sign to indicate the terminal prompt.

The goal of mercurial or git is the same: providing a way to structure the code development. The philosophy is to keep a working version, called changeset or commit, of the source code directory at every stage in the development.

Every changeset (or commit) is like a snapshot of the directory (and subdirectories) of the repository. At any point in time after a commit, you can come back to the state of the directory at the time of the commit. If done correctly, this means that you can compile or run any old version of your code. It is very useful for bug tracking, software development alone or in a team and to archive your progress. The advantages are multiple, but let just say that it is essential if you want to work in software development in the future.

2 Hello world

In this document, you can choose either left or right column to choose between the `hg` or `git` approach.

Both programs are pretty similar in the end. Only the most useful commands will be covered here. I suggest that you choose one of the two program and stick with it for the tutorial. You can check both program, but I suggest that you never use both in the same repository.

The first step is to create a new empty directory and move in this new directory. To create a new repository there, type:

```
$ hg init
```

```
$ git init
```

There you have it, a fresh new repository with nothing in it. Now create a new file named "`hello`" with any text editor (like `vi`), write the word "`world`" in it and save it. You can now add this new file to the repository:

```
$ hg add hello
```

```
$ git add hello
```

It means that this newly created file will now be tracked by your local repository. Note that adding a file is not a commit in itself. It just means that it will start to be tracked in the new commit. To see what are the changes to be committed, do:

```
$ hg status
```

```
$ git status
```

The information is similar. It should tell you which file have been added (modified or deleted) and which file are still not added. To commit the actual state of the directory to the repository, you now need to do a commit. But first, since you most likely never used mercurial or git, you need to configure the name and email, so that the changeset can identify who is the commiter:

```
$ hg config --edit
```

```
$ git config --global --edit
```

This will edit `.hgrc` or `.gitconfig` file in your home directory. Note that you can edit them directly with your prefered text editor. In these files, you need to provide at least your name and email using the following syntax:

```
[ui]
username = myName <my@email.com>
```

```
[user]
email = my@email.com
name = myName
```

Once this configuration detail is set, you can commit do your first commit:

```
$ hg commit -m "My first commit."
```

```
$ git commit -am "My first commit."
```

The option "`-m`" is needed to specify a message associated to the first commit. It is essential to write a message corresponding to the changeset you want to commit, in order to be able to find yourself afterward. It also implies to a certain extent that there is a good commit philosophy: you want to commit a program that run or compile ideally (unless specified in the commit message) and you want to commit every small change and not wait until you have a lot of them.

3 Growing Bigger

Now modify the committed file and create 2 new dummy files with a dummy text in it. Add them to the repository and commit the change. To add every files, you can simply do:

```
$ hg add
```

```
$ git add --all
```

Once you have more than one changeset, you can look at your history using

```
$ hg log -G
```

```
$ git log --graph --all --decorate
```

This will list every changeset with their information (number, name of the commiter, message). The option "`--graph`" will plot the tree of every commit and their relation on the left of the list. For now it is quite a simple tree: it is supposed to be a line. But it can be more complex and have multiple branches over time. Let us try to explore this.

4 Coming back

If you are not satisfied with your new changeset and want to go back to the first commit, type:

```
$ hg update <commit number>
```

```
$ git checkout <commit number>
```

The commit number can be obtained from the task "`log`", for example. Choose the commit number of the first commit and go back to this version. Doing so, you can see that you now only have one file in your directory: the one present at your first commit. The other files are not forgotten, you can come back to them if you update (checkout) to your most recent commit number.

But before coming back to the most recent commit: let us stay in the first commit. We want to create a new branch here. Type:

```
$ hg bookmark newBranch
```

```
$ git checkout -b newBranch
```

Create and add a new dummy file and commit this new changeset in the current branch. This will create a new branch. If you print the "`log`", you can see that there are now two branches with two heads. You can migrate from one branch to another using the commit number. Repeating this exercise, you can see that there can be an arbitrary number of branches.

5 Merge

Now that you have two branches, perhaps you would like to merge them. This happens a lot when collaborating with other developers. If you want to update the master branch with this newBranch, update (checkout) the repository to the master branch. Then, to merge, type:

```
$ hg merge newBranch
```

```
$ git merge newBranch
```

In mercurial, you will need to commit this merge as for git, the merge include a commit. Note that this merge is very simple to do because we created files with different names. Hence, the merge only corresponds to include (copy) all the files in the current branch.

In the case where the same file has been edited differently between two branches, you will need to merge the same file. This is possible to do in command line, with help of `vimdiff`, but this is out of the scope of the present document. The user can refer to:

https://www.mercurial-scm.org/wiki/MergingWithVim

http://www.rosipov.com/blog/use-vimdiff-as-git-mergetool/

for more information.

6 Ignore some files

Most of the time, you want to track plain text file only. Indeed, the "`diff`" tools and the changeset work best with ascii file. You could in principle track binaries or "`.pdf`" files, but as soon as you change them, the whole file will be different and the repository will conserve a whole copy of the file(s). The memory size of your repository will grow exponentially. This is to be avoided in general. So you must identify which file need or not to be tracked. Create for example the file "`a.out`" (a typical binary generated with `gcc`). If you look at the task "`status`", you see that the repository does not know what to do with the file in the next commit. If you want to ignore it completly in the future, edit the file "`.hgignore`" or "`.gitignore`" in the root directory of the repository. Write:

```
syntax: glob
a.out
*.o
```

```
a.out
*.o
```

Both "`ignore`" file has the same function here. It will always ignore a.out and any file that ends with "`.o`" (object files for example). This means that the task "`status`" will not complain or see them. So you will be able to commit new changeset without having to worry about them as they will not be tracked. It is a good practice to check any file to ignore before any call of the task "`add`".

7 Remote Repository

A good practice is to create a remote repository on a hosting facility like BitBucket for example. One can even host his own remote repository, as long as you have a valid address for your repository. If you go to BitBucket and follow the information there, you can create an account. Once this is done, you can create a new repository. Since we already have a repository here, we will upload it on BitBucket. Click on "I have an existing project" and follow the indication (copy the address it gives you). Then go in your local repository and type:

```
$ hg push <address of repository>
```

```
$ git push <address of repository>
```

If later you want to download the new changes pushed on this remote repository (if you work on more than one computer or with other programmer for example), you need to pull:

```
$ hg pull
```

```
$ git pull
```

Finally, if you want to import a remote project locally and you have nothing yet on your computer, you need to clone a repository.

```
$ hg clone <address of repository>
```

```
$ git clone <address of repository>
```

It is usually the first command that someone learn, but let us conclude on this.

8 Useful links

Only the most basic stuff have been covered by this tutorial. It is much more powerful, but with these command, you can deal with the most basic situation. Here is some link to different command translation between mercurial and git:

https://github.com/sympy/sympy/wiki/Git-hg-rosetta-stone

https://confluence.atlassian.com/get-started-with-bitbucket/git-and-mercurial-commands-860009656.html