Quick Intro to Version Control Systems and Project Hosting Services

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory,
[2]Deptartment of Informatics, University of Oslo

Jan 18, 2013

Version control systems allow you to record the history of files and share files among several computers and collaborators in a professional way. File changes on one computer are updated or merged with changes on another computer. Especially when working with programs or technical reports it is essential to have changes documented and to ensure that every computer and person involved in the project have the latest updates of the files. Greg Wilson' excellent Script for Introduction to Version Control provides a more detailed motivation why you will benefit greatly from using version control systems.

Projects that you want to share among several computers or project workers are today most conveniently stored at some web site "in the cloud" and updated through communication with that site. I strongly recommend you to use such sites for all serious programming and scientific writing work – and all other important files.

The simplest services for hosting project files is Dropbox. Simply go to `http://dropbox.com` and watch the video. It is very easy to get started with Dropbox, and it allows you to share files among laptops and mobile units.

When several people may edit files simultaneously, it can be difficult detect who did what when, roll back to previous versions, and to manually merge the edits when these are incompatible. Then one needs more sophisticated tools than Dropbox: project hosting services with true version control systems. The following text aims at providing you with the minimum information to started with such systems. Numerous other tutorials contain more comprehensive material and in-depth explanations of the concepts and tools.

The idea with project hosting services is that you have the files associated with a project in the cloud. Many people may share these files. Every time you want to work on the project you explicitly update your version of the files, edit the files as you like, and synchronize the files with the "master version" at the site where the project is hosted. If you at some point need to go back to a version of the files at some particular point in the past, this is an easy operation. You can also use tools to see what various people have done with the files in the various versions.

Four popular project hosting services are

- Bitbucket at `bitbucket.org`
- GitHub at `github.com`
- Googlecode at `code.google.com`
- Launchpad at `launchpad.net`

All these services are very similar. Below we describe how you get started with Bitbucket, GitHub, and Googlecode. Launchpad works very similarly to the latter three. All the project hosting services have excellent introductions available at their web sites, but the recipes below are much shorter and aim at getting you started even quicker by concentrating on the most important need-to-know commands. The tutorials at the cites contain more detailed information.

The mentioned services host all your files in a specific project in what is known as a *repository*, or *repo* for short. When a copy of the files are wanted on a certain computer, one clones the repository on that computer. This creates a local copy of the files. Now files can be edited, new ones can be added, and files can be deleted. These changes are then brought back to the repository. If users at different computers synchronize their files frequently with the repository, most modern version control systems will be able to merge changes in files that have been edited simultaneously on different computers. This is perhaps one of the most useful features of project hosting services. However, the merge functionality clearly works best for pure text files and less well for binary files, such as PDF files, MS Word or Excel documents, and OpenOffice documents.

For the examples below, assume that you have some folder tree `my-project` with files that you want to host at Bitbucket, GitHub, or Googlecode and bring under version control. The official name of the project is "My Project".

1 Installing version control systems

The various project hosting sites work with different version control systems:

- `bitbucket.org` offers Git (`git`) or Mercurial (`hg`),
- `github.com` offers Git (`git`),
- `code.google.com` offers Git (`git`), Mercurial (`hg`), or Subversion (`svn`),
- `launchpad.net` offers Bazaar (`bzr`).

All the version control systems are quite similar in the way users operate them, but Subversion is technically different from Git, Mercurial, and Bazaar. The latter three work very much in the same way for a beginner, but their more advanced commands and their nomenclature differ significantly. Which system to choose is mainly a matter of personal preference and experience. For some people the choice of project hosting site comes first, while others prefer a particular version control system and let this preference govern the choice of project hosting site.

It is possible to use Mercurial also with a GitHub project. This requires a plugin for Mercurial, described in `http://hg-git.github.com`. The plugin is easy to install (`easy-install hg-git` or `sudo apt-get install mercurial-git` on Ubuntu). You can also use Git to work with Subversion and Mercurial repositories.

The combination of GitHub and Git seems to have the strongest momentum and popularity in the software development community at present. However, my experience is that the learning curve of Git is steeper than that of Mercurial so the choice of Mercurial with Bitbucket or Googlecode will for many be the most convenient start.

1.1 Installing Mercurial

The Mercurial website has information on downloading Mercurial on different platforms. On Ubuntu and Debian-based Linux systems, you just perform

Terminal

```
sudo apt-get install mercurial
```

You can alternatively click in the Ubuntu Software Center application and choose the Mercurial package in the graphical interface.

Mercurial is pure Python code so it is trivial on any system with Python installed to download the Mercurial source code and perform the standard `sudo python setup.py install` command.

This tutorial emphasizes the command-line interface to Mercurial, but there is also a graphical interface called TortoiseHG, which can be installed by

Terminal

```
sudo apt-get install tortoisehg
```

on Ubuntu. TortoiseHG is the natural application to run Mercurial on a Windows machine.

We recommend that you create two files in your home folder: `.hgrc` for specifying the behavior of `hg` (Mercurial) and `.hgignore` for listing the type of files you in general do not want to have under version control. A simple `.hgrc` file can look like this:

```
[ui]
username = "Hans Petter Langtangen <hpl@simula.no>"
ignore=~/.hgignore
```

The `.hgignore` file lists the types of files that will be skipped when bringing new directories under Mercurial version control. Typically, the list contains file types that represent temporary information or just garbage, or files that are easily regenerated, such as object files (`*.o`) and libraries (`*.so`). The file types are normally specified using the Unix Shell Wildcard notation, also referred to as `glob` syntax (e.g., `*.o`, where `*` means any sequence of characters). An example of a `.hgignore` file may be

```
syntax: glob
# compiled files:
*.o
*.so
*.a
# temporary files:
*.bak
*.swp
*~
.*~
*.old
tmp*
temp*
# tex files:
*.log
*.dvi
*.aux
*.blg
*.idx
# eclipse files:
*.cproject
*.project
# misc:
.DS_Store
```

In addition to having an `.hgignore` file, you should always be careful with what kind of files you really need to add to your repo. Many computer tools produce a lot of big files that can easily be regenerated and that do not need to fill up repos with full information on previous versions.

1.2 Installing Git

The installation of Git on various systems is described on the Git website under the *Download* section. On Ubuntu the relevant commands are

Terminal

```
sudo apt-get install git gitk git-doc
```

Git involves compiled code so it is most convenient to download a precompiled binary version of the software on Windows, Mac and other Linux computers.

Make a file `.gitconfig` in your home folder with information on your full name, email address, your favorite emacs editor, and the name of a file which defines the file types that Git should omit when bringing new directories under version control. Here is a simplified version of the author's `.gitconfig` file:

```
[user]
name = Hans Petter Langtangen
email = hpl@simula.no
editor = emacs

[core]
excludesfile = ~/.gitignore
```

The `.gitignore` file should list, using the Unix Shell Wildcard notation, the type of files that you do not need to have under version control, because they represent garbage or temporary information, or they can easily be regenerated from some other source files. A suggested `.gitignore` file looks like

```
# compiled files:
*.o
*.so
*.a
# temporary files:
*.bak
*.swp
*~
.*~
*.old
tmp*
temp*
# tex files:
*.log
*.dvi
*.aux
*.blg
*.idx
# eclipse files:
*.cproject
*.project
# misc:
.DS_Store
```

Be critical to what kind of files you really need a full history of. For example, you do not want to populate the repository with big graphics files of the type that can easily be regenerated by some program.

2 Bitbucket

To start using Bitbucket, go to `bitbucket.org` and create an account. The communication channel with Bitbucket repositories is either through SSH or HTTPS. To use SSH, you must upload your SSH key, typically the contents of the file `id_rsa.pub` or `id_dsa.pub` in the `.ssh` subfolder of your home folder. Go to the page for your account, choose *SSH keys*, and upload one of these files. The essence of the SSH keys is that they allow you to communicate with Bitbucket without using a password, which is very convenient.

There are links to extensive help pages if you do not have such keys or if you are unfamiliar with SSH. Follow these steps on Mac or Linux machines to generate keys: 1) check that you have `ssh` installed; 2) create a `.ssh` folder in your home folder; and 3) run `ssh-keygen` in the `.ssh` folder (you are prompted for a passphrase - just write something). On Windows one applies the PuTTY and the TortoiseHG programs to generate and register keys, see the help pages on Bitbucket. Once the keys are generated, you can continue using them on any current and future computer.

2.1 Create a new project

Click at *Repositories* and at *Create repository*. You can now

- fill in the name of the project, here `my-project`,
- decide whether the project is private or public (the number of private repos is unlimited for yourself, but you have to pay to invite more than five users in total to share your private repos),

- choose between the Git or Mercurial version control system,
- click whether you want issue tracking for reporting errors, suggesting improvements, etc.,
- click whether you want a wiki page associated with the project,
- fill in a brief description,
- click on *Create repository*.

While doing this you may also want to have the Bitbucket 101 guide available (open the link in another tab or window in your browser).

It is now time to copy the project to your laptop, or *clone* it in Git and Mercurial terminology. Go to the project page and find the *Clone this repository* line. Click on *SSH*, copy the *clone* line and run this command in a terminal:

Terminal

```
hg clone ssh://hg@bitbucket.org/username/my-project
# or
git clone git@bitbucket.org:usernamet/my-project.git
```

Only the `hg` or the `git` clone command is displayed on Bitbucket since your project was initialized with either Merucial or Git as version control system (this cannot be changed later). In the above command you must replace `username` with your own user name at Bitbucket.

The first time you do the clone command you may be prompted by cryptic Unix output ending with "Are you sure you want to continue connecting (yes/no)?". Just answer yes.

The next step is to collect files and directories that should make up the project and put them in the `my-project` folder.

Then all files should be brought under version control, with the exception of the file types listed in `.hgignore`. Issue the following command in the `my-project` folder:

Terminal

```
hg add
# or
git add .
```

Thereafter, the changes to the repository (adding of files) must be registered, or *committed* if we use standard version control system terminology. The command reads

Terminal

```
hg commit   -m 'Initial import of files.'
# or
git commit -am 'Initial import of files.'
```

The `-m` option is a required description of the changes that have taken place. This description does not matter much for this initial import of files into the repository, but is of importance for future commit commands so that you can easily track the history of your project files.

The final step is to push the local changes to the master repo at Bitbucket: perform

Terminal

```
hg push
# or
git push -u origin master
```

You must be connected to the Internet for the `push` command to work since it sends file information to the `bitbucket.org` site.

Further work with the files must always follow the pull, edit, commit, and push steps explained in Section 5 for Mercurial and Section 6 for Git.

2.2 User web pages

Bitbucket can host web pages associated with a Bitbucket account. Say your account/user name is `username`. Make a new repository on Bitbucket called `username.bitbucket.org`. Clone it, fill it with a file, and push back, e.g.,

Terminal

```
git clone git@bitbucket.org:username/username.bitbucket.org.git
cd username.bitbucket.org
echo "Welcome to my web pages!" > test.html
git add test.html
git commit -am 'First web page'
git push origin master
```

You can now load the URL `http://username.bitbucket.org/test.html` into a web browser and view the page. By creating various subfolders you can host web pages for a series of projects in this repo.

Bitbucket does not yet offer ordinary repos to host and display web pages.

3 Googlecode

To use Googlecode you need a general account on Google, which allows you to use Gmail, Google Docs, and other products.

3.1 Create a new project

Go to `http://code.google.com/hosting` and click on *Create a new project*. Fill out *all* the fields. For now the project name is `my-project`. You have to choose between Git, Mercurial, or Subversion as version control system for your

project, and this choice cannot be changed. Git and Mercurial are preferred over Subversion nowadays. If you choose Git, you must create a file `.netrc` in your home folder containing the line

```
machine code.google.com login uname password pw
```

Here, `uname` is your user name for the Google account and `pw` is the Googlecode password which is generated for you on the `code.google.com/p/my-project`. The `.netrc` file avoids typing or pasting in your long and complicated password every time you push changes to the repository on `code.google.com`.

The next step is to clone the empty repository on your local machine so that you can add files:

Terminal

```
git clone https://username@code.google.com/p/my-project/
```

Now you can go to the `my-project` folder and add files. Perform

Terminal

```
git add .
git commit -am 'First import of files.'
git push origin master
```

Click on *Source* and *Browse* on the project's web page, and observe that the added files are visible on the project page.

If you use Mercurial as version control system on Googlecode and you want to avoid typing your password when you push changes to the repository, you should add the following section to your `.hgrc` file:

```
[auth]
my-project.prefix = https://my-project.googlecode.com/hg/
my-project.username = uname
my-project.password = pw
```

where `uname` and `pw` must be replaced by your account name and the special Googlecode password. Other projects on Googlecode using Mercurial will need similar lines.

A very strong and useful feature with Googlecode, in my opinion, is that one can reach the repository files directly through an URL. That means that one can place documentation of the project in the repository and find an URL to the HTML or PDF files of the documentation, which will then be displayed correctly. All other project hosting sites demands either wiki pages or special web areas for locating documentation. The URL to your files is

```
https://my-project.googlecode.com/git/
```

When using other version control systems, `git` is simply replaced by `hg` or `svn`. For example, if we have HTML documentation of our project in the folder `doc/html`, we can point users to

```
https://my-project.googlecode.com/git/doc/html/index.html
```

The HTML will be rendered correctly as opposed to when you load the similar file into the web browser from the repository,

```
http://code.google.com/p/my-project/source/browse/doc/html/index.html
```

Now you can only see the HTML source code of this file, as is usual on other project hosting sites. We remark that MathJax mathematics within the HTML code is *not* rendered correctly. At the time of this writing, GitHub is the only service that offers full MathJax support when you need web pages with mathematics.

You can click on *Project Home* and then on *Administer* to edit the main page of the project. This is a wiki, using Google's wiki syntax, but it is quite easy to add links to your documentation, e.g.,

```
Browse the
[https://my-project.googlecode.com/git/doc/API/html/index.html
API documentation].
```

It is easy to allow others to push their changes to the repository: click on *Sharing* and then on *Administer*. The Google account names of people you allow write access can be listed under each other in the *Project committers* field.

3.2 Wiki pages

Wiki pages can intuitively be made directly in the browser. However, it is often more convenient to have them locally on your computer. Click on *Source* and choose *wiki* on the *Repository* pull down menu. The proper clone command to get a copy of the wiki repository then appears.

Googlecode applies their own Google wiki format. My preference is to write documentation in the neutral Doconce format and transform the document to LaTeX, Sphinx, and Google wiki. The wiki can then be copied from the project directories to the wiki folder and then pushed to the repository. This ensures that there is only one source of the documentation (despite the need for many formats) and that the wiki pages are frequently updated.

4 GitHub

Go to `github.com` and create an account. Then go to your account settings, choose *SSH Keys*, and provide your SSH key. Everyone who is supposed to use your repository must provide their SSH key. There is a help that explains what this is all about. Often, it is just a matter of pasting the contents of `id_rsa.pub` or `id_dsa.pub` in the `.ssh` subfolder of your home folder into the *Key* box in the web page. Make sure to cut and paste the text from, e.g., `id_dsa.pub` without any extra whitespaces or other text. How to generate these files is briefly described in the introduction to Bitbucket above.

4.1 Create a new project

Click on *New repository* on the main page and fill out a project name, here *My Project*, click the check button *Initialize this repository with a README*, and click on *Create repository*. Unless you pay, all repos are public, but students and teachers can request free, private repos.

The next step is to clone the project on your personal computer. Click on the *Git Read-Only* button to see the correct address of the project, paste this address into a terminal window, after `git clone`:

Terminal

```
git clone git://github.com:username/My-Project.git
```

Make sure you substitute `username` by your own user name on GitHub.

The result of the `git clone` command is a new folder `My-Project`. It contains the file `.git`, which shows that it is a Git repository. It also contains a default `README.md` file with the project name and description. The extension `.md` signifies a file written in the Markdown format. You may use the reStructuredText format as an alternative (`README.rst`), or simply write a plain text file (`README`).

You can now add files and directories into the `My-Project` folder. When your initial file collection has the desired form, you must run

Terminal

```
git add .
git commit -am 'First set of files.'
git push -u origin master
```

To give others permissions to push their edits of files to the repository, you click on *Admin*, centered at the top of the project's page, then click on *Collaborators* on the left, and fill in the name of a collaborator (her or his user name on GitHub). Many find it convenient to be notified in email when others have pushed a new version of the files to the repo. Click on *Service Hooks* (in the project's *Admin* menu), choose *Email*, and fill in up to two whitespace-separated email addresses. One can send to more addresses by making a mailing list.

Anyone who participates in a project (has write access) or watches a project (having clicked the *watch* button) can monitor the development of the activity through email. Go to *Account Settings* and choose *Notification Center*. There you see two sections, *Participating* and *Watching*, for those participating in the project (granted write access) and those watching the project (having clicked the *watch* button), respectively. Clicking the *Email* button turns on email notifications to those participating and/or watching.

The daily file operations are explained in Section bitgit:git. There you also find information on how to create wiki pages and web pages associated with the project.

4.2 Wiki pages

With every GitHub project there is an option to create wiki pages. Click on the *Wiki* button right under the line with the project name of the main page of the project. Click on *New Page* to create a new page. The wiki pages can be written in different markup languages. Markdown is the default choice, but you can alternatively use reStructuredText (unfortunately not the Sphinx extension with nice LaTeX mathematics) or MediaWiki (the markup used for Wikipedia).

The wiki pages can be written and maintained through the web browser interface, but it is usually more convenient to clone them on your computer as this makes it easy to add figures and other documents you may want to link to. It also makes it straightforward to edit the wiki text in your favorite text editor. The wiki pages are stored in a separate repo: click on *Git Access* and then on *Git Read-Only* to see the Git address for a standard `git clone` command, typically

Terminal

```
git clone git://github.com/username/My-Project.wiki.git
```

This command makes a local copy of the pages in the folder `My-Project.wiki`, which you may prefer to have at the same level as the project folder itself in your folder tree.

Each wiki page has its own file, where the extension reflects the markup language used, e.g., `.md` for Markdown and `.rest` for reStructuredText. The wiki files are handled as other files in a GitHub project, i.e., you need to pull before editing and then perform commit and push. After the push you can reload the page in the web browser to monitor the effect.

You may consider having the original text in `doconce` format and generate the wiki in reStructuredText format.

Do changes, commit the usual way, and push by

Terminal

```
git push git@github.com:username/My-project.wiki.git
```

The address can be stored as `url` in `.git/config` so that just a `git push` works.

Project web pages

GitHub can also host a set of web pages for your project where you can store various types of documentation. Here is a simple recipe for creating a set of project web pages.

1. Go to the project page on `github.com` and click *Admin*.
2. Check the *GitHub Pages* check box.
3. Click on the *Automatic GitHub Page Generator* button in the pop up window.

4. Proceed with choosing a design of the `index.html` page and click *Publish*.

5. Go to the root folder of the project, `My-Project` and run `git fetch origin`.

6. Run `git checkout gh-pages`

You have now a *new branch* called `gh-pages` of your project containing an `index.html` file in the root folder (and in fact all files *not* contained in the *master branch*, typically redundant files not to be stored in the version control system). You can populate this folder and subdirectories you create with HTML and other files as you like.

The `index.html` page is invoked by the web address `http://username.github.com/My-Project/index.ht` where `username` is the GitHub user name and `My-Project` is the project name.

The web pages and project files are now in two different branches. To see the branches, type `git branch` and the one you are in will be marked with a star in the output. Switching to the master branch is done by `git checkout master`. Similarly, `git checkout gh-pages` switches to the gh-pages branch.

When branches diverge you can merge them by

Terminal

```
git merge otherbranch
```

where `otherbranch` is the name of another branch. If there is a conflict in the merge, there will be markers in problematic files. Run `git diff` to show the problems. After a manual edit, do `git commit -a`. Then you may run `gitk` to see a graphical illustration of what has happened.

My personal preference is to have the master and gh-pages synchronized, at least in projects where I need the source code files for generating API documentation in the gh-pages branch. Sometimes I also update files in the gh-pages branch without remembering to switch to the master branch. Merging the branches is then an easy way out of such potential trouble.

Start with merging the gh-pages branch with the master branch and push the complete file collection to the gh-pages branch. Then switch to the master branch and merge with gh-pages so you get the autogenerated `index.html` file and associated files and directories for the web design in the root folder of the master branch as well:

Terminal

```
git merge master
touch .nojekyll
git push origin gh-pages
git checkout master
git merge gh-pages
```

You *must* add an empty file `.nojekyll` in the top folder of the project pages if you want to use Sphinx-generated HTML pages (or other pages using javascripts,

style sheets, and images in subdirectories whose names start with an underscore).

You can now add the documentation to the project files and maintain them in the master branch. Before publishing documents online, make sure to update the gh-pages branch by

Terminal

```
git commit -am 'Ensure commit of everything in master branch'
git push origin master
git checkout gh-pages
git merge master
git push origin gh-pages
git checkout master
```

Personally, I like to move the generated `index.html` file and all associated scripts, stylesheets, and images from the root folder to some more isolated place, say `doc/web/git`:

Terminal

```
git mv index.html params.json stylesheets/ images/ \
       javascripts/ doc/web/git/
```

The `index.html` then has the URL

```
http://username.github.com/My-Project/doc/web/git/index.html
```

Linking to source code files or other files in the project is easy: just find the file in GitHub's web interface, choose which version of the file you want to link to (nicely HTML formatted version or the raw file), right-click on the link, choose *Copy Link*, and paste the link into the document you want. You can test that the link works by the Unix command `curl -O <link>`. Note that the link to a file is quite different from the source file's path in the repository. Typically, a source file `mydir/myfile.py` in project `prj` is reached through

```
https://github.com/username/prj/blob/master/mydir/myfile.py?raw=true
```

4.3 User web pages

GitHub also allows you to create user pages and organization pages not tied to any specific project. Your personal site has address `http://username.github.com`. Go to your home page on `github.com` and click *New repository*, and give it the project name `username.github.com`. Then follow the instructions that come up:

Terminal

```
mkdir username.github.com
cd username.github.com
git init
```

```
# make an index.html file with some test text
git add index.html
git commit -m 'First commit'
git remote add origin git@github.com:username/username.github.com.git
git push -u origin master
```

Go to `http://username.github.com` and see how the `index.html` is rendered. You can now add various contents as in any ordinary Git repository. If you want to use Sphinx generated HTML pages, recall to add an empty file `.nojekyll`. Different projects can be hosted in different subfolders.

5 Using Mercurial

Mac and Linux users will normally work with Mercurial through commands in a terminal window, which is the emphasized interface described here. Windows users will prefer to use TortoiseHG. This graphical interface is mostly self-explanatory, but a brief description appears below.

5.1 Basic use of TortoiseHG on a Windows machine

When you have an account on Bitbucket and have downloaded and installed TortoiseHG, you are ready to work with Mercurial repositories the following way.

- Start *Hg workbench*
- Register your username as your email address: choose *File - Settings* and then *Commit*.
- Clone a repository: choose *File - clone repository*. Paste the URL of the Bitbucket repo as *Source* and set *Destination* as some preferred folder on your machine. You will be prompted for a password which is the password for your Bitbucket account.
- When a file is changed and you want to commit the changes, right-click in the Explorer window showing the Mercurial folder and choose *Commit*. Write a comment in the text field to the right to document the changes.
- To synchronize your local files with the repository (i.e., to do a `push` command), right-click in the Explorer window, choose *TortoiseHG - Synchronize*. You need to provide the password for the Bitbucket account.

Many prefer to work in Visual Studio with their program files on Windows. This is easy also when you adopt Bitbucket as repo. Just start VisualStudio, make a solution and a project, create the desired files, then one can just right-click with the mouse on the solution, the project, or the file one wants to commit and choose *Commit*.

5.2 Basic Mercurial commands

Cloning. You get started with your project on a new machine or another user can get started with the project by the command

Terminal

```
hg clone ssh://hg@bitbucket.org/username/my-project
```

The pull-change-push cycle. Your typical working style with the `my-project` project goes as follows. First you go to the desired folder where this project is stored on your local computer and make sure you have the latest versions of the files:

Terminal

```
hg pull
hg update
```

These commands download the latest versions of the files from your `bitbucket.org` repository and make them ready for changes on your computer. The `pull` command requires a functioning Internet connection.

You can now edit some files. Maybe you also add, remove, and move some files:

Terminal

```
hg add filename
hg remove filename
hg rename oldfilename ../somedir/newfilename
```

The removal of a file is physically performed when you do a `hg commit`. The file is never removed from the repository, only hidden, so it is easy to get the file and its entire history back at a later stage.

After some changes, you have to commit and push the files to the repository at `bitbucket.org`:

Terminal

```
hg commit -m 'Description of changes.'
hg push
```

Note that you cannot easily redo the description so be careful with the wording. You can run `hg diff` to get a listing of all the changes you have made since the last commit or pull command.

The command

Terminal

```
hg stat
```

shows the status of the individual files (`M` for modified, `A` for added, `R` for removed), and you should pay attention to files with a question mark because these are not tracked in the repository. It is very easy to forget adding new files so `hg stat` is a useful command to ensure that all files you want to track have been added to the repository.

Viewing the history of files. The power of `hg` and your file repository `bitbucket.org` is that you can work with the project files on several computers and others can also contribute to this project. The history of each file is recorded and anyone can roll back to previous versions, if needed. You can easily see who did what with the various files.

For example,

Terminal

```
hg annotate -aun myfile
```

lists the various lines in the file `myfile` annotated with the revision number of the latest change of the line and the name of the user who performed the change.

Another command,

Terminal

```
hg log -p filename | less
```

lists the history of `filename` in more detail so that you can track the evolution of this file. Adding the `--follow` option will list the history also when `filename` had other names. The history of all files in a folder is listed by specifying the folder name(s). No name specification gives the history of the whole repo.

More compact output from `hg log`, without line differences, is triggered by the `--stat` option:

Terminal

```
hg log --stat filename | less   # changes in a specific file
hg log --stat | less            # changes in the whole repo
```

The combination `-p --stat` equips the line differences with an overview of which files that were changed in each revision.

If you wonder what has recently happened to a file after updating your local repository, just run `hg log -p --stat`: the last changes appear in the beginning of the output.

Retrieving old files. To restore an old version of a file with name `filename`, check the file history with `hg log -p filename` and find the revision number corresponding to the version of the file you want. Say this revision number is 152. The command

Terminal

```
hg revert -r 152 filename
```

replaces the current version of the file by the old one from revision 152. Sometimes it is better to store the old version in a separate file:

Terminal

```
hg cat -r 152 filename > somefile
```

The entire local repo can also be set back:

Terminal

```
hg update -r 152
hg update            # go back to the latest version again
```

Comments on two-level repositories. Mercurial has a two-level type of repository: there is global repository for all users of the `my-project` tree at `bitbucket.org`, but each user also has a local repository (automatically made). To update files, one must first pull new files from the global repository (`pull`) and then update the local repository (`update`), before any file can be edited. The `hg commit` command saves files in the local repository. One may run many `hg commit` commands to make a sequence of corresponding versions of files in the local repository. Finally, when the local versions are ready to be pushed to the global repository, one runs the `hg push` command.

Older version control systems, including Subversion and CVS, have only a global repository and no local ones. The advantage of the two-level repository is that you can change your files locally and keep track of the changes (by doing `hg commit`) without affecting other users of the files. This feature allows you to commit changes to the global repository only when you feel comfortable with the state of the files. Nevertheless, if others work actively on the same files as you, it is generally recommended to push often in order to exchange the latest version of the files. This strategy may avoid problems with future merging of files.

5.3 Merging files with Mercurial

It might have happened that others and you have edited the same files at the same time. How should the edits then be combined? Often the `hg update` command is clever enough to merge the changes made by different users automatically. If not, you have to run an explicit `hg merge` command. This command tries to use some merge program on your computer system to automatically merge files. If this fails, `hg merge` invokes some graphical tool to help you resolve the conflict between files. Example on popular merge programs for this purpose are `meld`, `xxdiff`, `kdiff3`, and `diffuse`. You can specify the merge program, say `xxdiff`, by `hg merge --tool xxdiff`. How you now proceed is

dependent on the particular program. Usually, for each change you must choose either your new local version of the text, or your old local version of the text, or the version of the text pulled from the global repository. After using the merge program successfully you must save the merged file and perform an `hg commit` command on it, and perhaps do an `hg push` to also update the global repository with the merged file(s).

Suppose you do not manage to merge using the merge tool. Then you have to invoke the file, say `myfile` in an editor and do the merge manually. There will often be lines starting with '¡¡¡¡', '=====', and '¿¿¿¿' to mark conflicting texts (*merge markers*). After removing these lines and editing the text manually, you must register the conflict as resolved:

Terminal

```
hg resolve -m merge myfile
hg commit -m merge myfile
```

You can read more about merging files in the hgbook. We shall now illustrate the merge problems through an example. Suppose we have a global repository `tmp_repo`, and two (cloned) directories `tmp1` and `tmp2`, corresponding to two users, each with their copy of the global repository. There is only one file `myfile` in the repository. We then simulate the two users and perform edits in parallel. The shell script below simulates the two users and illustrates the importance of pulling before editing and the need of merge.

```
#!/bin/sh
# Demo script for exemplifying hg merge

rm -rf tmp1 tmp2 tmp_repo   # Clean up previous runs

mkdir tmp_repo   # Global repository for testing
cd tmp_repo
cat > myfile <<EOF
This is a little
test file for
exemplifying merge
of files in different
hg directories.
EOF
hg init   # Make hg global repository out of this directory
hg add    # Add all files not mentioned in ~/.hgignore
hg commit -m 'first commit'
cd ..

# Make a new hg repositories tmp1 and tmp2 (two users)
hg clone tmp_repo tmp1
hg clone tmp_repo tmp2

# Change myfile in the directory tmp1
cd tmp1
# Edit myfile: insert a new second line
perl -pi -e 's/a little\n/a little\ntmp1-add1\n/g' myfile
# Register change in local repository
hg commit -m 'Inserted a new second line in myfile.'
# Look at changes in this clone
```

```
hg log -p
# Register change in global repository tmp_repo
hg push
cd ..

# Change myfile in the directory tmp2 "in parallel"
cd tmp2
# Edit myfile: add a line at the end
cat >> myfile <<EOF
tmp2-add1
EOF
# Register change locally
hg commit -m 'Added a new line at the end'
# Register change globally
hg push
# Error message: global repository has changed,
# we need to pull those changes to local repository first
# and see if all files are compatible before we can update
# our own changes to the global repository.
# hg writes
# abort: push creates new remote head d0a2f8e6b9d9!
# (you should pull and merge or use push -f to force)

hg pull
# hg writes:
# added 1 changesets with 1 changes to 1 files (+1 heads)
# (run 'hg heads' to see heads, 'hg merge' to merge)
hg merge
# Successful merge!
cat myfile
hg commit -m merge
hg push
cd ..

# Perform new changes in parallel in tmp1 and tmp2,
# this time causing hg merge to fail

# Change myfile in the directory tmp1
cd tmp1
# Do it all right by pulling and updating first
hg pull
hg update
# Edit myfile: insert "just" in first line.
perl -pi -e 's/a little/tmp1-add2 a little/g' myfile
# Register change in local repository
hg commit -m 'Inserted "just" in first line.'
# Register change in global repository tmp_repo
hg push
cd ..

# Change myfile in the directory tmp2 "in parallel"
cd tmp2
# Edit myfile: replace little by modest
perl -pi -e 's/a little/a tmp2-replace1\ntmp2-add2\n/g' myfile
# Register change locally
hg commit -m 'Replaced "little" by "modest"'
# Register change globally
hg push
# Not possible: need to pull changes in the global repository
hg pull
hg update
```

```
# hg update aborts: we have to run hg merge
diff myfile ../tmp_repo/myfile
echo 'Now you must do hg merge manually'
```

Try to run this file named `hg_merge.sh` by `sh -x hg_merge.sh`. To resolve the resulting merge conflict you need to go to the `tmp2` folder, run `hg merge --tool meld` and use the `meld` tool to select which text snippets that should make up the final, merged version. Save and quit `meld` and perform `hg commit -m merge`. You can now do `hg push` successfully to update the global repository.

5.4 More documentation on Mercurial

- Mercurial Quick Start (for the impatient)
- A Tour of Mercurial - The Basics
- Mercurial FAQ
- Mercurial Tutorial
- Mercurial: The Definitive Guide (online or printed book)

6 Using Git

Most Mac and Linux users prefer to work with Git via a commands in a terminal window. Windows users prefer a graphical user interface (GUI). There are many options in this respect. There are also GUIs for Mac users. Most GUIs are easy to use once the basic concepts of Git and version control are understood.

6.1 Basic Git commands

Cloning. You get started with your project on a new machine, or another user can get started with the project, by running

Terminal

```
git clone git@github.com:username/My-Project.git
cd My-Project
ls
```

Recall to replace `username` by your real user name and `My-Project` by the actual project name.

The pull-change-push cycle. The typical work flow with the "My Project" project starts with updating the local repository by going to the `My-Project` folder and writing

Terminal

```
git pull origin master
```

After editing files, adding new files, deleting files, or renaming files,

Terminal

```
git add file1 file2.* dir1 dir2
git rm filename
git rm -r dirname
git mv oldname newname
git mv oldname ../newdir
```

it is time to commit your changes (note the `-am` option):

Terminal

```
git commit -am 'Description of changes.'
```

If typos or errors enter the message, the `git commit --amend` command can be used to reformulate the message. Running `git diff` makes it easier to formulate descriptive commit messages since this command gives a listing of all the changes you have made to the files since the last commit or pull command.

You may perform many commits, to keep track of small changes, before you push your changes to the global repository:

Terminal

```
git push origin master
```

To see the status of files, run

Terminal

```
git status -s
```

Files are marked with different symbols, e.g., `A` for added, `M` for modified, `R` for renamed, and `??` for not being registered in the repo.

Viewing the history of files. A nice graphical tool allows you to view all changes, or just the latest ones:

Terminal

```
gitk --all
gitk --since="2 weeks ago"
```

You can also view changes to all files or some selected ones in the terminal window:

Terminal

```
git log -p                          # all changes to all files
git log -p filename                 # all changes to a specific file
git log --stat --summary            # compact summary of changed files
git log --stat --summary  subdir    # only files in subdir folder
```

Adding `--follow` will print the history of file versions before the file got its present name.

To show the author who is responsible for the last modification of each line in the file, use `git blame`:

Terminal

```
git blame filename
git blame --since="1 week" filename
```

A useful command to see the history of who did what, where individual edits of words are highlighted (`--word-diff`), is

Terminal

```
git log -p --stat --word-diff filename
```

Removed words appear in brackets and added words in curly braces.

Looking for when a particular piece of text entered or left the file, say the text `def myfunc`, one can run

Terminal

```
git log -p --word-diff --stat -S'def myfunc' filename
```

This is useful to track down particular changes in the files to see when they occurred and who introduced them. One can also search for regular expressions instead of exact text: just replace `-S` by `-G`.

Retrieving old files. Occasionally you need to go back to an earlier version of a file, say its name is `myfile`. Start with viewing the history:

Terminal

```
git log myfile
```

Find a commit candidate from the list that you will compare the present version to and copy the commit hash (string like `c7673487...`) and run

Terminal

```
git diff c7673487763ec2bb374758fb8e7efefa12f16dea myfile
```

where the long string is the relevant commit hash. You can now view the differences between the most recent version and the one in the commit you picked. If you want to restore the old file, write

Terminal

```
git checkout c7673487763ec2bb374758fb8e7efefa12f16dea myfile
```

To go back to another version (the most recent one, for instance), find the commit hash with `git log myfile`, and do `get checkout <commit hash> myfile`. At any time you do `git commit` the current version of `myfile` will enter the most recent commit of the current branch.

If `myfile` changed name from `yourfile` at some point and you want `yourfile` back, run `git log --follow myfile` to find the commit when `yourfile` existed, and do a `git checkout <commit hash> yourfile`.

The Git equivalent to `hg cat` for just viewing old versions of files is `git show`, but the command requires the full path from the root git folder:

Terminal

```
git show c7673487763ec2bb374758fb8e7efefa12f16dea:dir1/dir2/myfile
```

6.2 Merging files with Git

The `git pull` command fetches new files from the repository and tries to perform an automatic merge if there are conflicts between the local files and the files in the repository. Git will write a message if the merge is unsuccessful for one or more files. These files will have to be edited manually. Merge markers of the type '¿¿¿¿¿', '======', and '¡¡¡¡¡' have been inserted by Git to mark sections of a file where the version in the repository differ from the local version. You must decide which lines that are to appear in the final, merged version. When done, perform `git commit` and the conflicts are resolved.

Graphical merge tools may ease the process of merging text files. You can run `git mergetool --tool=meld` to open the merge tool `meld` for every file that needs to be merged (or specify the name of a particular file). Other popular merge tools supported by Git are `araxis`, `bc3`, `diffuse`, `ecmerge`, `emerge`, `gvimdiff`, `kdiff3`, `opendiff`, `p4merge`, `tkdiff`, `tortoisemerge`, `vimdiff`, and `xxdiff`.

Below is a Unix shell script illustrating how to make a global repository in Git, and how two users clone this repository and perform edits in parallel. There is one file `myfile` in the repository.

```
#!/bin/sh
# Demo script for exemplifying git and merge

rm -rf tmp1 tmp2 tmp_repo   # Clean up previous runs

mkdir tmp_repo   # Global repository for testing
cd tmp_repo
```

```
git --bare init --shared
cd ..

# Make a repo that can be pushed to tmp_repo
mkdir _tmp
cd _tmp
cat > myfile <<EOF
This is a little
test file for
exemplifying merge
of files in different
git directories.
EOF
git init
git add .   # Add all files not mentioned in ~/.gitignore
git commit -am 'first commit'
git push ../tmp_repo master
cd ..
rm -rf _tmp

# Make a new hg repositories tmp1 and tmp2 (two users)
git clone tmp_repo tmp1
git clone tmp_repo tmp2
# Change myfile in the directory tmp1
cd tmp1
# Edit myfile: insert a new second line
perl -pi -e 's/a little\n/a little\ntmp1-add1\n/g' myfile
# Register change in local repository
git commit -am 'Inserted a new second line in myfile.'
# Look at changes in this clone
git log -p
# or a more compact summary
git log --stat --summary
# or graphically
#gitk
# Register change in global repository tmp_repo
git push origin master
cd ..

# Change myfile in the directory tmp2 "in parallel"
cd tmp2
# Edit myfile: add a line at the end
cat >> myfile <<EOF
tmp2-add1
EOF
# Register change locally
git commit -am 'Added a new line at the end'
# Register change globally
git push origin master
# Error message: global repository has changed,
# we need to pull those changes to local repository first
# and see if all files are compatible before we can update
# our own changes to the global repository.
# git writes
#To /home/hpl/vc/scripting/manu/py/bitgit/src-bitgit/tmp_repo
# ! [rejected]        master -> master (non-fast-forward)
#error: failed to push some refs to ...

git pull origin master
# git writes:
#Auto-merging myfile
```

```
#Merge made by recursive.
# myfile |    1 +
# 1 files changed, 1 insertions(+), 0 deletions(-)
cat myfile  # successful merge!
git commit -am merge
git push origin master
cd ..

# Perform new changes in parallel in tmp1 and tmp2,
# this time causing hg merge to fail

# Change myfile in the directory tmp1
cd tmp1
# Do it all right by pulling and updating first
git pull origin master
# Edit myfile: insert "just" in first line.
perl -pi -e 's/a little/tmp1-add2 a little/g' myfile
# Register change in local repository
git commit -am 'Inserted "just" in first line.'
# Register change in global repository tmp_repo
git push origin master
cd ..

# Change myfile in the directory tmp2 "in parallel"
cd tmp2
# Edit myfile: replace little by modest
perl -pi -e 's/a little/a tmp2-replace1\ntmp2-add2\n/g' myfile
# Register change locally
git commit -am 'Replaced "little" by "modest"'
# Register change globally
git push origin master
# Not possible: need to pull changes in the global repository
git pull origin master
# git writes
#CONFLICT (content): Merge conflict in myfile
#Automatic merge failed; fix conflicts and then commit the result.
# we have to do a manual merge
cat myfile
echo 'Now you must edit myfile manually'
```

You may run this file named `git_merge.sh` by `sh -x git_merge.sh`. At the end, the versions of `myfile` in the repository and the `tmp2` folder are in conflict. Git tried to merge the two versions, but failed. Merge markers are left in `tmp2/myfile`:

```
<<<<<<< HEAD
This is a tmp2-replace1
tmp2-add2

=======
This is tmp1-add2 a little
>>>>>>> ad9b9f631c4cc586ea951390d9415ac83bcc9c01
tmp1-add1
test file for
exemplifying merge
of files in different
git directories.
tmp2-add1
```

Launch a text editor and edit the file, or use `git mergetool`, so that the file becomes correct. Then run `git commit -am merge` to finalize the merge.

6.3 Git working style with branching and stashing

Branching and stashing are nice features of Git that allow you to try out new things without affecting the stable version of your files. Usually, you extend and modify files quite often and perform a `git commit` every time you want to record the changes in your local repository. Imagine that you want to correct a set of errors in some files and push these corrections immediately. The problem is that such a push will also include the latest, unfinished files that you have committed.

A better organization of your work would be to keep the latest, ongoing developments separate from the more official and stable version of the files. This is easily achieved by creating a separate branch where new developments takes place:

Terminal

```
git branch newstuff      # create and switch to new branch
# extend and modify files
git commit -am 'Modified ... and added a file on ...'

git checkout master      # swith back to master
# correct errors
git push origin master

git checkout newstuff    # switch to other branch
# continue development work
git commit -am 'More modifications of ...'  # record changes
```

At some point, your new developments are mature enough to be incorporated in the master branch. This is done via a merge:

Terminal

```
git checkout master
git merge newstuff
```

You no longer need the `newstuff` branch and can delete it:

Terminal

```
git branch -d newstuff
```

It is not possible to switch branches unless you have committed the files in the current branch. If your work on some files is in a mess, and you want to change to another branch or commit other files, a "global" commit might be immature. Then the `gitstash` command is handy. It records the state of your files and sets you back to the state of the last commit in the current branch. With `git stash apply` you get the files in this branch back to the state when you did the last `git stash`.

Stashing is ideal if you want to perform incremental, experimental work, especially with programming. Suppose an experimental task consists of subtasks

A, B, C, and D. You create a new branch and start working on A. Halfway into that work, you are interrupted and need to do correct some other files. Instead of committing messy files, you stash the changes, switch to the master branch and address the other files. Later, you can go back to the stashed files. Here is a typical session:

Terminal

```
git stash
git checkout master
# fix a thing in master
git commit -am 'Fixed ...'
git checkout newstuff
git apply stash
# finalize subtask A
git stash  # record files after subtask A and set back to last commit
git stash apply  # set back to files before last git stash
```

You can now continue with tasks B, C, and D and do `git stash` whenever you want a piece of work to be saved. Maybe it turns out that the experimental task was not a good idea. With `git stash` you are back to the state where you were prior to working on A. On the other hand, if A and B were successful, but not C and D, you may roll back to the state after B was finished, if you did a `git stash` at that stage (see the chapter about stashing in the Pro Git for details). If A, B, C, and D were all successful, you just perform a commit to get a new "official" version of your files. At some later stage, you can switch back to the master branch and merge with the new, experimental branch.

Sometimes you have not planned your new work well enough to start a new branch, but suddenly you just realize that you are the process of bringing some files into a mess. It is easy to take all the new developments since the last commit out in a new branch by

Terminal

```
git stash
git stash branch newstuff
```

The real power of branching and stashing becomes clear when you push and hence share your clearly separated, unfinished work with other users or other computers.

Remark. A recent extension module for Mercurial allows stashing for that version control system as well: `hg shelve --all` is similar for `git stash`, and `hg unshelve` is similar to `git stash apply`.

6.4 More documentation on Git

- Web course on Git
- Everyday GIT With 20 Commands Or So

- Git top 10 tutorials
- Lars Vogel's Git Tutorial
- Git Community Book (*explains* Git very well)
- Git for Designers (aimed a people with no previous knowledge of version control systems)
- Git Magic: Basic Tricks
- The official Git Tutorial
- Git Tutorial Video on YouTube
- Git Questions
- Git Reference (can also be used as a tutorial on Git)
- Git User Manual
- Git home page
- Git and Mercurial command equivalence table
- Git/GitHub GUIs on Windows and Mac

6.5 Diff of the two most recent versions of a file

Both Mercurial and Git have "diff" commands for viewing differences in files. However, it sometimes takes quite some manual steps to answer the following question: I have just pulled new changes and want to view what these changes are for some selected files, using my favorite diff tools (maybe different tools for different types of files)? The answer consists of doing the following steps:

- Run a log command to see the identity of the previous version
- Run `hg cat` or `git checkout` (or `git show`) to get the previous version of the file, and save that version as a separate file
- Run some diff tool on the present and previous version of the file