Git

Blaise Thompson

March 30, 2020

Contents

Part of the training materials prepared by the Chemistry Shops at UW–Madison.
This document was prepared using LaTeX.
Source code and all associated files can be found at git.chem/shop/training/git.
If you find any mistakes or feel that any information is missing, please open an issue.

1 introduction

1.1 what is Git?

Git is a project management system. Git helps people to manage the files within a project, store different versions of those files over time, and back-up / share that project online. Git can be used as a "personal cloud" like Google Drive, Microsoft OneDrive, Dropbox, and Box. However Git extends this basic "file backup" idea with additional powerful features.

Within Git, a project (typically called a "repository") is a folder ("directory") containing files and subfolders. As edits are made to the files within the repository, Git tracks those changes. At any time, the person editing a Git repository can save their progress by "commit"ing their edits. Git will store (locally, within a hidden folder) the changes that were made between each commit. Anyone with access to that repository can, then, go back to the state of the project when that commit was made. All Git repositories contain their entire history within the local hidden folder.

Git repositories may be backed up ("pushed") and copied ("pulled") from an online server (the "remote"). In this way, Git repositories can be shared between multiple computers and possibly multiple people. Pushing changes up and pulling changes down is explicitly initiated by the user, so there is never any worry about things changing in the "background" e.g. Google Drive Sync. There are many options for Git servers, and all of them do the same basic job of storing your repository on a remote machine and allowing you to push and pull changes. Many Git servers also present an online web-based interface for interacting with your files, which can be very convenient. Popular options include:

→ GitHub
→ GitLab
→ Gitea
→ Built-in Git server

GitLab deserves special mention here, because our Chemistry Department runs a departmental GitLab instance at `https://git.chem.wisc.edu/`. Repositories hosted here can be open to the world, private only to departmental members, private within a group, or private to just one person. Again, while Git is good at backing up and sharing projects, usage of an online server is always strictly optional. Everything is stored locally, so even if the server goes down or is deleted, each machine has a complete copy of the repository.

The most advanced features of Git are used to enable collaboration between multiple people working on the same repository. Git has a concept of "branches", which are different versions of the project that can be worked on simultaneously. Two people can branch off from a particular version, do their work separately committing several times along the way, and then "merge" their changes back together later. This collaboration model is very powerful and enables thousands of software developers to simultaneously work on huge projects like the Linux kernal. Branching and merging are strictly optional features.

1.2 a few short stories

Katie uses Gaussian computational software in her research. She is exploring a large range of initial conditions using a grid search strategy. Katie uses Git to manage a collection of three or four scripts that she uses to run her simulations and process resulting data. Katie uses the departmental GitLab instance to store her scripts in a private repository. Even though she is the only student working on the project right now, Katie benefits from the version control and backup features as she continues to tweak her script. Katie appreciates the assurance that she can always go back to an earlier version.

Louis uses several custom instruments in his daily research. Each of these is a typical analytical/physical chemistry instrument with many components and a large LabVIEW software stack originally written by a long-since-graduated student. Many people rely on these instruments, so it is crucial that their functionality is not interrupted even as Louis improves the software. Louis uses Git to store working versions of the existing LabVIEW code. He then feels confident that he can make edits and improvements without "losing" the old functionality. While he irons out bugs, Louis makes sure that he reverts to the original code so that other users are not interrupted. Louis backs-up the LabVIEW software on the departmental GitLab instance, using a "Group" to ensure that his labmates and advisor also have access.

The Wright Group's research requires them to process large, complex, & multidimensional datasets. In pursuit of this goal, several graduate students spend a significant amount of their time developing a custom data processing library in Python. Due to the scale of this development effort and the number of graduate students working simultaneously on the project, the Wright Group decides to use a branching and pull request workflow to help everyone collaborate. The Wright Group decides to host their code on GitHub, making it publicly available in the hope that other scientists might benefit from and contribute to the library. After more than 1000 commits and hundreds of thousands of changes, the Wright Group publishes their software efforts to advertise their accomplishment to the academic community.

1.3 what Git is not

Git is not a particularly good choice for general file backup, and Git servers should probably not be used for general file storage. Git is not a good choice for a collection of publications in PDF form. Git is not a good choice for a series of collected data files. Git is not a good choice for an excel spreadsheet containing student grades. Git is made to help people manage projects made up of files which change over time. What Git gives you is the ability to go back to a previous version of the project, and optionally to work collaboratively with others. If your project is made up of a bunch of static files, which you will never plan to edit, Git is not for that project.

Git is not a good tool for large file storage. If your files are approaching or exceeding 100 MB, consider a different solution.

2 installation

This document assumes you are running Git version 2.23.0 or newer. Use `git --version`.

Windows

On Windows this guide recommends installing directly from the maintained build at https://git-scm.com/download/win.

This download comes with several additional components which are useful for software development on Windows. Click through the installer, the default choices are fine for everything.

As a Windows user you will interact with git via the "git bash" application. It's recommended that you open this application and set notepad as your default text editor. Then, you want to tell Git your email and name.

```
$ git config --global core.editor notepad
$ git config --global user.email "bthompson@chem.wisc.edu"
$ git config --global user.name "Blaise Thompson"
```

MacOS

On MacOS, simply type `git` into your terminal and you will be prompted to install. If Git is already installed on your machine make sure it is up to date. Once Git is installed, give it your email and name.

```
$ git config --global user.email "bthompson@chem.wisc.edu"
$ git config --global user.name "Blaise Thompson"
```

Linux

On Linux, use your distribution's package manager. Make sure Git is up to date. Once git is installed, give it your email and name.

```
$ git config --global user.email "bthompson@chem.wisc.edu"
$ git config --global user.name "Blaise Thompson"
```

3 exercise

This section is intended to be used within the context of a teaching workshop for first-time git learners. In this exercise you will learn the basics of Git. You will create your own Git repository, edit it, make commits, and push it to a remote server. You will use the terminal in this exercise, so we start by reviewing a few basic terminal commands.

```
$ pwd # print current working directory
$ ls # list files in current working directory
$ ls -a # list files, including hidden files
$ cd <NAME> # change directory to <NAME>
$ cat <FILE> # print contents of <FILE> to terminal
$ mkdir <NAME> # create a directory called <NAME>
```

3.1 initializing your first repository

On the Desktop of your computer, create a directory called `myrepo`. This is just a simple folder, and you can create it through your graphical file browser or using the terminal. Now open Git *within* this new folder. On Windows, you can right click in the file browser and choose "Git Bash here". On MacOS and Linux, you want to navigate to that directory in your terminal. Initialize the directory `myrepo` as a Git repository:

```
$ git init
Initialized empty Git repository in C:/Users/User/Desktop/myrepo/.git/
```

This command tells Git that this directory is now a repository that Git should track changes within. This isn't magic! `myrepo` is still just a normal directory. Now let's see what Git thinks about the new repository:

```
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

The `git status` command will be our best friend whenever using Git. As we can see here, Git is very talkative and helpful!

3.2 changing, adding, commiting, and reverting

Let's make some changes to our repository and see how Git behaves. Using any text editor of your choice, create a file `hello.txt` in your `myrepo` directory. After you're done, type `git status` again. This time you will see that there is an untracked file. Git even gives you a clue: use git add ... to include in what will be committed. Let's do that, then commit!

```
$ git add hello.txt
$ git commit -m "initial commit"
```

Next, let's learn about `diff`. Make some more edits to your `hello.txt` file. Save your file and type `git diff` into the terminal. Git will attempt to summarize the changes that were made. Red is lines deleted, and green is lines added. `diff` will always give the differences relative to the previous commit (in this case, the initial commit we made in the last paragraph).

Let's imagine that we regret changing our file. If we wish to simply throw away our changes to that file and return to the last commit we may use `git restore`. After restoring, open the file again in your favorite text editor and you will see that the changes you just made have been reverted. `git diff` should now indicate that nothing has changed. Please note: restoring permanently deletes your changes, so it should only be used when you are sure you don't want to keep your work.

Edit your `hello.txt` file again. Use your inspection tools (`git status`, `git diff`) to make sure that everything is happening the way you expect This time, rather than "throwing away" our edits, let's keep them: `git commit -am "edit hello.txt"`. The "a" flag says "commit all tracked files", and the "m" flag says "the following is the commit message".

We've made two commits now: "initial commit" and "edit hello.txt". Type `git log` to view a log of git commits. Now let's switch between commits. Looking at the output of `log`, copy down the first four characters from the long string that appears above your initial commit. For me, those characters are `4d9f`. Let's return to that state: `git checkout 4d9f`. Look at your file, has it returned to what it was? To return to the most recent commit, type `git checkout master`. Try going back and forth a few times to get a feel for it.

Let's review what we've learned in this section:

- → use `git add <FILE>` to begin tracking a file
- → use `git commit -am "<MESSAGE>"` to commit all tracked files
- → `git status`, `git diff`, & `git log` help you inspect your repository
- → use `git restore <FILE>` to permanently return a file to the last commit state
- → use `git checkout <SHA>` to temporarily return a repository to an earlier commit, `git checkout master` to return to the most recent commit

3.3 remotes, pushing, and pulling

In this section we will learn how to "backup" our repository on a remote server. In this case we will be using the departmental GitLab instance at https://git.chem.wisc.edu/. Begin by logging into the GitLab web interface using your chemistry username and password (LDAP).

From the GitLab homepage click the big green "New Project" button. Give the project the name "myrepo", and keep all the other options the same. Returning to the command line, type the following (replace "bthompson" with your username):

```
$ git remote add origin https://git.chem.wisc.edu/bthompson/myrepo.git
$ git push -u origin --all
```

If you get an "SSL certificate problem", use `git config --global http.sslVerify false`. You will be asked for your chemistry username and password.

Returning to your repository on GitLab, you will see your `hello.txt` file. Click around the GitLab interface and get familiar with where everything is. One feature that deserves special mention is the ability to download a zip of your repository—this is an excellent way to share your files with those who would prefer not to learn Git. We can even edit our `hello.txt` file through the online interface! Do that now, and commit your edit.

Because we edited our repository online through GitLab, the remote version is now changed relative to our local code. Returning to the terminal, type `git pull`. This will "pull" the changes that remote knows about down to your machine. Use your inspection tools: `status`, `log`, `diff` to see what's going on.

This tutorial has thus-far dealt with a single repository that you created locally. What if we want to use or work on a repository that already exists on a remote server? Using your terminal, navigate until your current directory is the Desktop (hint: `cd ..`). Now type:

```
$ git clone https://git.chem.wisc.edu/shop/training/git
```

You will notice that a new "git" directory appears on the Desktop. Navigating into this directory, you will find that all of your favorite git tools work.

Let's review what we've learned in this section:

- → use `git remote ...` to manage which servers your local repository communicates with
- → use `git push` to send local commits to the server
- → use `git pull` to take remote commits from the server
- → use `git clone` to clone a new repository

3.4 branches

In one of your repositories, type `git switch -c mybranch` to switch to a new branch named "mypbranch". `c` for create. Make some edits and commit them. Now, `git switch master`—changes go away. `git switch mybranch` to get them back. Use `git merge mybranch` when in the master branch to merge the changes back in. `git branch -vv` prints useful information about repository branches.

3.5 learning more

This author recommends the following for those looking to collaborate using Git:

→ git flow
→ git branching

Remember how we had to type in random numbers and letters to checkout a previous commit? Consider using tags if you find yourself going back to a certain point in the repositories history often. Read more: Git-Basics-Tagging.

By the way, how *does* Git store all this information locally? Using your operating system's file browser, turn on "view hidden items". You will see a hidden directory ".git"—this is where git stores all the extra information about previous versions of your repository. Go ahead—look inside this hidden directory and poke around! You'll get to see the "internal guts" of Git, which are not very human readable but still can be interesting to look at. Read more: Git-Internals.

3.6 alternative interfaces

Finally, a note about alternative interfaces to Git. Git is popular to the point of being virtually ubiquitous, and many different tools have been made which offer alternative interfaces to Git. In this author's opinion, using standard Git from the command line is still the easiest option. Still, you may wish to be aware of some of the popular alternatives:

→ GitHub Desktop
→ GitKraken
→ git-gui

Many text editors and other tools also have extensions for Git. The Git wiki has collected an extensive list of Interfaces, Frontends, & Tools.

4 cheatsheet

Inspect a repository.

- → `git status`
- → `git diff`
- → `git log` (Blaise prefers `git log --oneline`)

Commit changes.

- → `git add <FILE>` add a new file to be tracked
- → `git commit -am "<MESSAGE>"` commit all tracked changes

Revert changes.

- → `git restore <FILE>` return file to previous commit, throwing away changes
- → `git checkout <SHA>` return repository to state at previous commit (consider tagging)
- → `git checkout master` fast-forward repository to most recent commit

Interact with remote.

- → `git remote -v` list remote servers
- → `git push`
- → `git pull`

Branching

- → `git branch -vv` inspect repository branches
- → `git switch <NAME>` switch to branch "NAME"
- → `git switch -c <NAME>` switch to NEW branch "NAME"
- → `git merge <NAME>` merge branch "NAME" into current branch

5 resources

Other resources to learn git.

http://swcarpentry.github.io/git-novice/

https://tom.preston-werner.com/2009/05/19/the-git-parable.html