Git Class 2

Houtan Bastani[*][†]

30 August 2012

1 Creating a Git Repository For Yourself

1.1 From Nothing

```
> mkdir myProject
> cd myProject
> git init
```

And that's it! Now, create, add and commit files as usual.

1.2 From an Existing Directory

```
> cd myProject
> git init
```

And that's it! Now, you'll need to add all the files that you want to have in the repository (the commit message can be something like "initial commit", because there's nothing to revert) and commit as usual. Of course, there's no pushing because this repository is just for you.

1.3 File Structure of Private Repo

Under the root project folder, change into the `.git` directory and take a look at what's there:

```
drwxr-xr-x   2 houtan  staff    68B Aug 30 09:30 branches
-rw-r--r--   1 houtan  staff   111B Aug 30 09:30 config
-rw-r--r--   1 houtan  staff    73B Aug 30 09:30 description
-rw-r--r--   1 houtan  staff    23B Aug 30 09:30 HEAD
drwxr-xr-x  12 houtan  staff   408B Aug 30 09:30 hooks
drwxr-xr-x   3 houtan  staff   102B Aug 30 09:30 info
drwxr-xr-x   4 houtan  staff   136B Aug 30 09:30 objects
drwxr-xr-x   4 houtan  staff   136B Aug 30 09:30 refs
```

- `branches` is depricated and only here for historical reasons (only used by older versions of git)
- `config` lists the configuration for the git repository. Can be edited by hand or by using the `git config` command. See Section 4.
- `description` only used with GitWeb (a git web interface used for browsing repositories)
- `HEAD` The branch you're currently on

[*]Dynare & GPM Teams, CEPREMAP
[†]`houtan.bastani@ens.fr`

- `hooks` Actions to take after specified git commands are executed (*e.g.*, email after `git commit`). See Section 5.
- `info` contains an `exclude` file, used the same way as `.gitignore`
- `objects` contains the contents of the database (all commits)
- `refs` contains pointers to data in the objects directory

Basically, you don't need to know much about git "plumbing". Everything more or less takes care of itself. However, when setting up a git repo, you'll probably want to change the configure file as well as add some hooks. I'll talk about these in the sections specified.

2 Creating a Git Repository To Share

2.1 From Nothing On The Server

```
> mkdir myProject.git
> cd myProject.git
> git init --bare --shared=group
```

So, what do those flags mean? Well, the `--bare` flag indicates that we don't want to store the actual files in this repository. Remember, in the non-bare repo, all of the necessary git files were contained in the `.git` folder. Now, in the bare repository, those files will be in the root folder, as no one will actually be working directly from this repository. People will only clone, push and pull with this repo. And the `--shared=group`? That just tells git to set the permissions on the repository such that group members associated with it can read and write to it. If you wanted a more restrictive setting, you could use the typical linux octals, *e.g.*, `--shared=0640` to allow group members to read but not to write, or *e.g.*, `--shared=0644` to allow everyone to read but not to write. The default is `--shared=false`, which uses the default file permissions settings for the process you're using (reported by `umask` on linux/os x).

2.2 From an Existing Directory and Putting it on the Server

First, create a git repository as detalied above:

```
> cd myProject
> git init
> git add .
> git commit -m "initial commit"
```

Next, on the server, create a bare git repo:

```
> mkdir myProject.git
> cd myProject.git
> git init --bare --shared=group
```

Now we have an empty, bare repository on the server and all the files we want to include in that repository on our computer. All we need to do now is connect the two and push the changes to the repository on the server:

```
> git remote add origin ssh://user@server.com/path/to/myProject.git
> git push -u origin master
```

This is the regular way you would push any new branch to the repository (all the `-u` switch tells git to do is to make your local `master` branch track the `origin/master` branch you just created)

2.3 Bare Git Repo File Structure

Is just the contents of the `.git` directory from above in the root level. Thus, the bare git repo does not contain the actual composed files but, rather, the minimum files necessary to clone a repository and recreate the files on your local machine.

3 Git Remote

In git, we use `remote`s to point to remote repositiories. Remember that git is a decentralized (distributed) version control system and, thus, every git repository is a full repository. A remote simply allows us to point to other repositories. Remote repositories should track the same **development history** as the project you're working.

We use remotes for `fetch`ing and `push`ing. Thus, if we have a repository on our computer, that we don't share with anyone and that is completely self-contained (`i.e.` we won't be pulling from elsewhere), then we don't need any remote repositories. In such a repository, if we were to run `git remote -v` to list all remotes, we would have a blank list.

On the other hand, in a shared repository, you will necessarily be fetching from a remote and, if you have write access, you'll be pushing too.

The most common remote repository is `origin` and for the basic git setup, it's all you'll need. To add a remote named origin, we do:

```
> git remote add origin ssh://user@server.com/path/to/myProject.git
```

using whatever protocol necessary to access the bare git repo, which will set `ssh://user@server.com/path/to/myProject.gi` as both the fetch and push repo for origin. You won't usually need to do this because when you clone from a repository, it will usually be declared as `origin` by default (unless your system admin did something unusual).
At Dynare, we have a layer of quality-assurance between reading and writing to a repository. So, my local git has two remotes:

```
> git remote -v
origin  ssh://houtanb@kirikou.dynare.org/srv/d_kirikou/git/dynare.git (fetch)
origin  ssh://houtanb@kirikou.dynare.org/srv/d_kirikou/git/dynare.git (push)
personal  ssh://houtanb@kirikou.dynare.org/srv/d_kirikou/git/dynare-houtanb.git (fetch)
personal  ssh://houtanb@kirikou.dynare.org/srv/d_kirikou/git/dynare-houtanb.git (push)
```

Here, the `origin` repository has the official version of Dynare while the `personal` repository contains a repository to which I push changes I make to be reviewed by someone else (a "gatekeeper") before being merged into the central repository. This was setup with the following command:

```
> git remote add personal ssh://houtanb@kirikou.dynare.org/srv/d_kirikou/git/dynare-houtanb.git
```

Of course, I could also have used the following commands:

```
> git remote set-url --push origin ssh://houtanb@kirikou.dynare.org/srv/d_kirikou/git/dynare-houtanb.git
> git remote -v
origin  ssh://houtanb@kirikou.dynare.org/srv/d_kirikou/git/dynare.git (fetch)
origin  ssh://houtanb@kirikou.dynare.org/srv/d_kirikou/git/dynare-houtanb.git (push)
```

which would have the effect of fetching from the main dynare repo and pushing to my personal dynare repo, without forcing me to specify a remote. You should do this only if you're comfortable with it. I don't do it because it adds a layer of unnecessary obfuscation.

4 Git Config

There are three configuration files: project-specific (located `.git/config`, modified using `git config`), personal (located ∼`/.gitconfig`, modified using `git config --global`) and system-wide (varies by system, modified using`git config --system`). These files can be edited by hand or by using the `git config` command.

Thus, the commands we used last time produce the following personal git config file:

```
[user]
        name = Houtan Bastani
        email = houtan.bastani@ens.fr
[color]
        ui = auto
[core]
        editor = emacs
```

The `ui` line tells git that I want it to use color for all commands that support it (*e.g.* `git diff`, `git status`, etc), while the `editor` line tells it what editor I prefer for editing (used by `git commit`, `git rebase -i`, etc)

The Dynare project config file:

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = true
        sharedrepository = 1
[receive]
        denyNonFastforwards = true
[hooks]
        mailinglist = commit@dynare.org
        showrev = "git show -C %s; echo"
        emailprefix = "[Dynare Git] "
```

The first three lines under core are inserted when you use the `--bare` flag when initializing a git repo. Lines 4-6 were inserted because the Dynare repo was created with `--shared=true`. The 6th line specifies that users cannot rewrite the git history (by forcing changes into the repository). As we spoke about the other day, one key characteristic of git is that it stores the entire project history. Thus, you can always ensure this is the case with `denyNonFastforwards = true`. Of course, if you want to increase the odds of shooting yourself in the foot, you should buy a gun and thus, you can set this value to false if you want to tempt fate ☺

Of course, there are many other key/value pairs (we'll talk about those `hooks` below), and you just have to look at the manual to see which ones will enable you to use git in the way you prefer.

5 Hooks

Hooks are just shell scripts that run before and/or after certain git commands are executed. They can be set both on the client side and on the server side. Actually, when you initialize a git repository, by default you are provided with sample hooks. Listing the contents of the hooks directory of a new git repository shows the following files:

```
applypatch-msg.sample    post-commit.sample       post-update.sample       pre-commit.sample        prepare-com
commit-msg.sample        post-receive.sample      pre-applypatch.sample    pre-rebase.sample        update.samp
```

Here, you have sample scripts that perform basic actions before/after a certain git command is issued. The names of the sample files are fairly self-descriptive as to when they act and based on what action they act. And the contents of these files provide a guide on which to build the actions of your dreams ☺

Several useful actions are:

- Send an email to everyone when the main repository is updated (indicating that they should now updated their local repositories)
- Enforce specific commit message format
- Denying forced pushes (can be set in config for git $\geq$ 1.6)

You can use any scripting language to create a hook (even Python) but many of the actions are canned and can be found online with simple google searches. Before writing your own, you should probably search for one of these first.

6 Git Interactive Rebase

Though the idea of git is to maintain the entire development history (and, hence, we don't want to overwrite it), as we are developing we may run into situations where we want to reorder/merge our commits. This may happen when you are working locally and, for example, you have made two bug fix commits and are about to push them to the shared repo when you notice that your first bugfix commit didn't actually fix the bug. You edit the code, commit it, but now have a tree that looks like this:

```
o [master] bug fix 1 update - timestamp today 12:01 (SHA sha3)
o bug fix 2                 - timestamp today 12:00 (SHA sha2)
o bug fix 1                 - timestamp today 11:59 (SHA sha1)
o [origin/master] msg       - timestamp whenever    (SHA sha0)
```

This, however, breaks the defining characteristic of a commit: that it reprepsents a **logical** change to the code because you have two commits that fix bug 1, whereas you should only have one commit that fixes bug 1. Before pushing to the shared repository, we can fix this issue by using an interactive rebase (command: `git rebase -i sha0` or `git rebase -i origin/master`), which will yield a screen akin to:

```
pick fb65f2d bug fix 1
pick f7f45c6 bug fix 2
pick 853c3f0 bug fix 1 update

# Rebase 81d5998..853c3f0 onto 81d5998
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

To modify the commit history/ordering/commit messages/etc, you can then reorder the lines, delete the lines and/or modify the word `pick` according to the `Commands:` section. To accomplish what we originally wanted in this example, we would change those lines to be:

```
pick fb65f2d bug fix 1
f 853c3f0 bug fix 1 update
pick f7f45c6 bug fix 2
```

7 Git Merge

Git `merge` is a command that combines two or more development histories (branches). You can encounter it when deciding to merge some experimental branch back into the main branch, as we spoke about in the first class. Or, you might need it if you're a gatekeeper and responsible for merging developers' commits into the main git repository.

To merge two branches, first checkout the branch you want to merge into, then merge in the other branch:

```
> git checkout master
> git merge development
```

Easy as that! Just, not really. If it is, breathe a sigh of releif. If it's not, well, you have some more work to do. If it doesn't go well, git will let you know with a message akin to:

```
Auto-merging trash.txt
CONFLICT (add/add): Merge conflict in trash.txt
Automatic merge failed; fix conflicts and then commit the result.
```

while the file(s) with the conflicts will be marked up like:

```
<<<<<<< HEAD
My first version of trash.txt
=======
My second version of trash.txt
>>>>>>> trash1
```

and here is where the thought comes in. You must resolve this conflict by hand because git simply cannot do it. So, you would edit this file, add it and commit it as:

```
> git add trash.txt
> git commit
```

git remembers that you are in the middle of a merge and will set the commit message for you, which, of course, you can edit as you please.

If you find yourself in the middle of a merge that you do not want to complete, simply run either `git merge --abort` or `git reset --hard [<<SHA>>]`.

8 Git Submodule

A `submodule` is used for including a separate project within your git repository (*e.g.*, a 3rd party library, Dynare, Iris, some project you're developing but using in multiple other projects, etc). It allows you to include the project within your git repository and point to a certain branch/SHA whithin that project. The idea is to keep the development histories of the host and submodule projects separated; it is thus different than a remote pointer to a repository. Further, you usually do not have write access to the repository included in the submodule. As a submodule is a complete git repository, you can view its development tree just by switching into the directory containing it.

So, for example, say you're working on GIMF or GPM and want to use the unstable versions of both Dynare and Iris. Well, you can simply include them as submodules to your project and update the submodule pointers as you see fit to use the bleeding edge of those two projects.

In Dynare, we use submodules to include code developed by Sims, Waggonner and Zha. To create a submodule, you would do the following:

```
> git submodule add http://www.dynare.org/git/frbatlanta/utilities_dw.git contrib/ms-sbvar/utilities_dw
```

This will create a directory called `dynare_root/contrib/ms-sbvar/utilities_dw` containing the utilities_dw.git repository. (The reason it points to a repository housed at dynare and not at the Atlanta Fed is that we have a cron job that runs every night, replicating their repositories, just for our own backup purposes. Usually, however, a submodule will point directly to the 3rd party's git repository.) This creates a new file, `.gitmodules`, located in the root directory, with the following information:

```
[submodule "contrib/ms-sbvar/utilities_dw"]
  path = contrib/ms-sbvar/utilities_dw
  url = http://www.dynare.org/git/frbatlanta/utilities_dw.git
```

Now, the `.gitmodules` file and the `contrib/ms-sbvar/utilities_dw` directory need to be added and committed as per usual. If you don't want to point to the `HEAD` of the submodule, you need to change into the directory, and change the current position to the SHA that corresponds to the commit you want to use (via a `git reset`).

Remember, a submodule is a **complete** and **independent** git repository, so you can treat it as such. You can change into the submodule directory, view its git commit tree, modify files, push/fetch them and do basically whatever else you would do in a regular git repository. However, in your project, the only thing that is ever stored is a **pointer** to the SHA that represents the commit you want to include in your project.

Finally, to clone from a repository containing submodules, all you need to do is the following:

```
> git clone ssh://to/repository.git
> cd repository
> git submodule init
> git submodule update
```

where the penultimate line initializes the submodule and the final line brings in the entire submodule repository, checking out the appropriate commit.

Whenever a submodule pointer is changed, all that a user needs to do is:

```
> git submodule update
```

which will update the pointer to the appropraite SHA.

9 Good References

- official: `http://git-scm.com`
- man pages: `http://git-scm.com/docs`
- another: `http://www.kernel.org/pub/software/scm/git/docs/v1.7.3/user-manual.html`
- list of guis: `http://git-scm.com/downloads/guis`