Git Introduction

September 5, 2017

1 Introduction

This is a small introduction to Git based on *Jump Start Git*[1]. It is mainly developed for use on Grendel which uses an older version of Git. Hence some of the information might be outdated for newer versions.

1.1 What is Git?

Git is a distributed version control system. Here distributed means that every developer has a copy of the entire project allowing everyone to make changes without affecting the other developers. Upon finishing changes to the code it can be synchronized with the other developers.
Version control (or revision control) record changes to files and allow users to revert back to an earlier version.

1.2 GitHub

GitHub is a service that complements Git. Specifically it manages Git repositories by uploading remote versions to `https://www.github.com` for easy synchronization between developers.

1.3 Setting up Git

After installing Git one should set up some initial configurations, especially user name and email. This will be useful as this information is attached to updates (commits) of the project.

```
git config --global user.name "UserNameHere"
git config --global user.email "UserEmailHere"
git config --global color.ui "auto"
git config --global core.editor emacs
```

The global tag specifies that this information will be used for all repositories. The color configuration allows Git to use coloring schemes when displaying outputs in the terminal. The final line changes the default editor Emacs. Otherwise Git uses the default terminal editor which is vi unless changed by the user. Git will sometimes open this editor if for no commit message is supplied. Configuration settings can also be altered in the **~/.gitconfig** file.

2 Creating a Git project

To create a Git repository enter a directory and use the command

```
git init
```

Initially there will be no files added to this project even if files are present in the directory. To see this, run the following command which displays useful information about the current repository

```
git status
```

To tell git to track files do

```
git add fileExample
```

G will then start tracking fileExample. Alternatively do

```
git add .
```

Where . is an alias for current directory. Git then tracks the current directory and all subdirectories. An example for the above commands can be seen below.

```
mkdir Test
cd Test/
echo "Test" > Test.txt
git init
    Initialized empty Git repository in Test/.git/
git status
    On branch master

    Initial commit

    Untracked files:
      (use 'git add <file>...' to include in what will be committed)

            Test.txt

    nothing added to commit but untracked files present (use 'git
    add' to track)
git add Test.txt
git status
    On branch master

    Initial commit
```

```
    Changes to be committed:
      (use 'git rm --cached <file>...' to unstage)

            new file:   Test.txt
```

2.1 Ignoring files

There might be files unsuited for sharing between developers, such as *.DS_Store* files on OS X. To disable tracking for some files create a *.gitignore* file. Simply writing **.DS_Store* in this file tells Git to ignore all files ending with *.DS_Store*. While one can create a personal *.gitignore* file it us usually easier to visit `https://www.gitignore.io` and type in editor, programming language, operating system etc. Then a *.gitignore* template will be created automatically. Note that already tracked files will not be untracked by adding a *.gitignore*.

3 Committing

3.1 First commit

Assuming that Git is now tracking a file we can save a version of this file. This is done by committing. The following command does the trick

```
git commit -m 'First commit'
    [master (root-commit) 25fbe35] First commit
     1 file changed, 1 insertion(+)
     create mode 100644 Test.txt
```

Here -m specifies that we are adding a message to the commit.

3.2 Additional commits

Now assume that we append some text to the first line in our Text.txt file. The following happens

```
git status
    On branch master
    Changes not staged for commit:
      (use 'git add <file>...' to update what will be committed)
      (use 'git checkout -- <file>...' to discard changes in working
      directory)

            modified:   Test.txt

    no changes added to commit (use 'git add' and/or 'git commit -a')
```

We can also see what is modified by running

```
git diff
    diff --git a/Test.txt b/Test.txt
    index 9f4b6d8..68a52ea 100644
    --- a/Test.txt
    +++ b/Test.txt
    @@ -1 +1 @@
    -This is a test file
    +This is a test file, more info
```

Here the line with the minus is the changed line, and the one with plus is the new line. Lets say we are satisfied with the changes and want to commit. Before doing so we have to stage the changes. Perhaps we have changed multiple files but want this commit to only record changes in some of the files. We can stage the modified file by running

```
git add Test.txt
```

Or alternatively

```
git add -u
```

to add all tracked files. Now the modified file can be committed as usual. To skip the staging step do

```
git commit -a -m 'Second commit'
```

Where -a automatically stages all tracked files.

3.3 Information about different commits

Information about the Git repository is displayed by

```
git log
    commit f76a8004256d5852422822d081bb12dc35bb2c9b
    Author: Sm <sm@phys.au.dk>
    Date:   Fri Sep 1 14:34:16 2017 +0200

        Second commit

    commit 74e37f02fb20e6cf1c9963e4e17b5924e22f850e
    Author: Sm <sm@phys.au.dk>
    Date:   Fri Sep 1 14:27:43 2017 +0200
```

```
        First commit
```

or perhaps more conveniently

```
git log --oneline
    f76a800 Second commit
    74e37f0 First commit
```

where the first number identifies the commit. If one needs more information about the commit do

```
git show f76a
    commit f76a8004256d5852422822d081bb12dc35bb2c9b
    Author: Sm <sm@phys.au.dk>
    Date:   Fri Sep 1 14:34:16 2017 +0200

        Second commit

    diff --git a/Test.txt b/Test.txt
    index 9f4b6d8..68a52ea 100644
    --- a/Test.txt
    +++ b/Test.txt
    @@ -1 +1 @@
    -This is a test file
    +This is a test file, more info
```

where the commit ID just needs to be long enough to uniquely identify the commit, but seems to require at least 4 characters.

4 Branches

Branches are essentially a copy of the project which can be developed without affecting the original. If the changes are successful they can be merged back to the original project. When a repository is initialized only the master branch is created (after the first commit).

```
git branch
    * master
```

4.1 Creating branches

To create a branch simply do

```
git branch test_branch
git branch
    * master
      test_branch
```

We now see two branches, where the new branch is a copy of the branch from which we created it. The asterisk indicate that we are still on the master branch. To switch branch do

```
git checkout test_branch
    Switched to branch 'test_branch'
```

4.2 HEAD

Branches are essentially just links between different commits. The test_branch has now created a link to the commit 'Second commit'. The 'tip' of the current branch is called HEAD and is thus the newest commit in this branch. Thus currently both branches have the same HEAD.

4.3 Visualizing branches

Lets add a few commits to the test_branch to see how the branches can be visualized. Afterwards I have added a single commit in the master branch which I am now in (as indicated by HEAD).

```
git log --oneline --graph --decorate --all
    * ded37a0 (HEAD -> master) Adding commit in master
    | * 9d74907 (test_branch) Commit 2 in test_branch
    | * fab5d45 Commit 1 in test_branch
    |/
    * f76a800 Second commit
    * 74e37f0 First commit
```

Here graph is the visualization, decorate shows information such as HEAD, branch names, etc. and all specifies that we want the log of all branches. The graph is read from the bottom and shows the diverging of test_branch from the master branch. Note that the naming of the commits are bad as they are not tied to a specific branch. For example the entire test_branch could be rebased on top of the master branch. Note also that one should most likely not make a commit in the master branch. Usually new features are developed in branches and then merged back into the master branch when finished. Then the master branch always contains the newest fully working code.

4.4 Merging

Assume that the feature developed in the test_branch is finished and is the be implemented in the working code. We thus need to reintroduce it in the master branch. This process is called merging.

```
git checkout master
    Switched to branch 'master'
git merge test_branch -m 'Merging'
    Merge made by the 'recursive' strategy.
     Test.txt | 2 ++
     1 file changed, 2 insertions(+)
git log --oneline --graph --decorate --all
    *   d2dc03d (HEAD -> master) Merging
    |\
    | * 9d74907 (test_branch) Commit 2 in test_branch
    | * fab5d45 Commit 1 in test_branch
    * | 3251f18 Adding commit in master
    |/
    * f76a800 Second commit
    * 74e37f0 First commit
```

The merge command merges test_branch into the current branch, which is why we changed branch to master. Now the changes added in test_branch have been added to the master branch. The changes in text_branch were adding two lines to Test.txt and the change in the master branch was adding a new file. Since these operations do not overlap we are able to merge without conflicts. If we wish to we can continue working in the test_branch. Since we merged into the master branch the commit in the master branch has not been implemented in the test_branch. If we do another commit in the test_branch the structure is then as follows. Note that the branches have switched places.

```
git checkout test_branch
''' change some files and commit '''
git log --oneline --graph --decorate --all
    * 3201485 (HEAD -> test_branch) Commit 3 in test_branch
    | *   d2dc03d (master) Merging
    | |\
    | |/
    |/|
    * | 9d74907 Commit 2 in test_branch
    * | fab5d45 Commit 1 in test_branch
    | * 3251f18 Adding commit in master
    |/
    * f76a800 Second commit
```

```
    * 74e37f0 First commit
```

5 Remote/GitHub

So far we have only been working on the local repository. If we wish to synchronize with other developers a remote repository is needed. GitHub supplies such an option. Simply create an account and a repository and copy the URL of the GitHub repository. Then attach it to your local repository

```
git remote add origin https://github.com/SoerenMeldgaard/Test.git
```

This adds a remote repository and names it origin. For Grendel SSH might be better. See section 5.4.

5.1 Pushing

To upload committed changes to a remote repository a push is needed. This is done by the following command

```
git push -u origin master
    Counting objects: 17, done.
    Delta compression using up to 8 threads.
    Compressing objects: 100% (10/10), done.
    Writing objects: 100% (17/17), 1.41 KiB | 0 bytes/s, done.
    Total 17 (delta 1), reused 0 (delta 0)
    remote: Resolving deltas: 100% (1/1), done.
    To https://github.com/SoerenMeldgaard/Test.git
     * [new branch]      master -> master
    Branch master set up to track remote branch master from origin.
```

Here we tell Git to push the local branch master into the remote repository called origin. If this branch does not exist on origin it will create it. This is the case for us since our remote repository is empty. The -u flag tells Git that this will be our upstream remote. Now this branch is being tracked allowing for easier pulling and pushing. A similar procedure can be applied when a new branch is pushed for the first time. Simply typing *git push/pull* will now update from the tracked remote. Note however, that by default (at least on older Git versions, such as on Grendel) *git push* will push ALL tracked branches and not just the one you are in. To change the default behavior of pushing one can do

```
git config --global push.default tracking
```

which will change the default behavior to only pushing the current branch to its upstream branch. For newer versions of Git (2.x) tracking is called upstream. However, in this version the default behavior only pushes the current branch. Alternatively one can always use the full (safer) syntax for pushing

```
git push remote_name local_branch:remote_branch
```

5.2 Cloning

Sometimes you might want to download an existing repository. This process is called cloning.

```
git clone https://github.com/SoerenMeldgaard/Test.git
    Initialized empty Git repository in /home/sm/GitHub/Test/.git/
    remote: Counting objects: 17, done.
    remote: Compressing objects: 100% (9/9), done.
    remote: Total 17 (delta 1), reused 17 (delta 1), pack-reused 0
    Unpacking objects: 100% (17/17), done.
```

This will automatically track the origin/master branch and create a local/master branch. However, there might be multiple remote branches that you want local copies of. Simply do

```
git checkout remote_branch
```

Which will create a local copy of the remote branch and start tracking the remote branch. If cloning on Grendel one can also set up SSH instead. See section 5.4

5.3 Pulling

In case another developer has made changes to the remote repository you will need to pull these changes to your local repository in order to work with them. Assuming that the changed remote branch is already tracked do

```
git pull
```

This will fetch data for all branches but only update the branch you are on. Alternatively there is the full (safer) command

```
git pull remote_name local_branch:remote_branch
```

5.4 Authentication on Grendel

There seems to be some trouble using HTTPS with Grendel (at least for me), so instead one can use SSH. Presumably you already have an SSH key on Grendel stored in ~/.ssh/id_rsa.pub. Copy this key and link it with your GitHub account. Now for adding do

```
git remote add origin git@github.com:SoerenMeldgaard/Test.git
```

and for cloning do

```
git clone git@github.com:SoerenMeldgaard/Test.git
```

6 Solving conflicts

When merging, pulling and pushing, conflicts between files might appear. Here a simple example for a merge between two local branches is shown. The idea is essentially the same for all conflicts. Assume that the following file (test.py) is created in the master branch.

```
CONSTANT = 5
    def add_constant(number):
        return CONSTANT + number
```

Now I create a new branch called feature and changes the value of CONSTANT to 7. Then I switch back to my master branch and change CONSTANT to 9. Now the two branches have diverged. If I had not changed the master branch no conflicts will rise since feature branch is just ahead of the master branch. Merging feature branch into master branch would then override CONSTANT = 5 to CONSTANT = 7. However, now a merge conflict will arise.

```
git checkout master
git merge feature -m 'Merging feature into master'
    Auto-merging test.py
    CONFLICT (content): Merge conflict in test.py
    Automatic merge failed; fix conflicts and then commit the result.
```

Now opening test.py shows

```
<<<<<<< HEAD
CONSTANT = 9
=======
CONSTANT = 7
>>>>>>> feature

def add_constant(number):
    return CONSTANT + number
```

The lines between <<<<<<< HEAD and ======= shows the file content in the master branch. Lines between ======= and >>>>>>> feature, shows the content in the feature branch. To resolve the conflict edit the file(s) so they are as you want them to be in the master branch.

If multiple conflicts occur there will be several instances of <<<<<<< and >>>>>>> which all must be solved. After solving the conflicts add the file(s) and do a commit.

Another conflict worth mentioning is pushing updates to a remote that has changed since the last synchronization. Git is not able to do this, so instead you must first pull the changes from the remote and solve any conflicts that arise. Then commit the changes and you are able to push.

7 GUI tools

Multiple GUI tools are available that remove the need for any terminal commands. Popular choices are GitHub Desktop (GitHub's own GUI tool) and SourceTree, both available for Windows and Mac but not Linux. Especially SourceTree seems to be a popular choice.

The GitHub website can also be used to commit and merge branches in the remote repository.

8 Useful commands

Some useful commands when working with Git are displayed below. Words in capital letters are to be supplied by the user. Note that some of the commands might have different outcomes based on the Git version.

8.1 Commits

```
git add FILE              # Track a file
git add .                 # Track current directory and
                            sub-directories
git add -u                # Stage all tracked files

git rm --cached FILE      # Untrack file, but let it remain in system

git commit -m 'MESSAGE'   # Commit with message

git reset HEAD~1          # Undo 1 commit and unstage changes
git reset --soft HEAD~1   # Undo 1 commit but let changes remain
                            staged.
git reset --hard HEAD~1   # Undo 1 commit and delete changes

git revert HEAD~1         # Undo 1 commit by doing new commit with
                            changes removed

git checkout COMMITID     # Look at files at this commit
git checkout BRANCHNAME   # Then revert back to newest version
                            of branch
```

```
git show COMMITID          # Show information about a commit

git diff                   # Show changes in all files
git diff FILE              # Show changes in file
```

8.2 Branches

```
git branch                     # List local branches
git branch -a                  # List local + remote branches
git branch BRANCHNAME          # Create a branch
git branch -D BRANCHNAME       # Delete a branch

git checkout BRANCHNAME        # Switch to branch
git checkout -b BRANCHNAME     # Create and switch to branch
```

8.3 Merging

```
git merge BRANCHNAME                # Merge BRANCHNAME into current
                                      branch
git merge --no-ff BRANCHNAME        # Merge BRANCHNAME into current
                                      branch and create a commit
git merge --abort                   # Abort a merge
git merge --rebase BRANCHNAME       # Rebase BRANCHNAME into current
                                      branch
```

8.4 Pulling

```
git clone ADDRESS                           # Clone a remote repository
git fetch                                   # Update local branches
                                              but do not merge
git pull                                    # Download and merge remote
                                              branch with same name as
                                              current branch
git pull REMNAME LOCBRANCH:REMBRANCH        # Pull REMBRANCH from
                                              REMNAME into LOCBRANCH
```

8.5 Pushing

```
git remote add origin ADDRESS          # Add a remote repository
                                         called origin
```

```
git push                                    # Push all matching branches
                                              (in old Git version)
git push -u origin master                   # Push master to origin and
                                              set it as upstream
git push REMNAME LOCBRANCH:REMBRANCH        # Push LOCALBRANCH to
                                              REMOTEBRANCH on
                                              REMOTENAME repository
```

8.6 Other

```
git config --global user.name "NAME"        # Set user name
git config --global user.email "EMAIL"      # Set email
git config --global color.ui "auto"         # Color terminal outputs
git conifg --global core.editor emacs       # Default editor emacs

git status                                  # Git status

git log                                     # Show information about
                                              commits in current
                                              branch
git log --oneline                           # Condensed information
                                              about commits
git log --all                               # Show information about
                                              commits in all branches
git log --decorate                          # Show which branch
                                              commits belong to
git log --graph                             # Graphical representation
git log --all --decorate --oneline --graph # Usefull graphical
                                              representation
```

References

[1] S. Daityari. Jump Start Git, 2015.