

A hands-on Git tutorial

1st Reproducible Research Workshop, KB, February 7

Ali R. Vahdati

Table of contents

- 1 Configuring git for the first time
 - 2 Starting a local repository
 - 3 Comparing differences
 - 4 Viewing previous commits
 - 5 Reverting mistakes
 - 6 Managing branches
 - 7 Resolving conflicts
 - 8 Working with remote repositories
 - 9 Other meterial
 - 10 Resources
-

1 Configuring git for the first time

```
$ git config --global user.name "John Doe"  
$ git config --global user.email johndoe@example.com
```

You need to do this only once if you pass the `-global` option.

Check your configurations:

```
$ git config --list
  user.name=John Doe
  user.email=johndoe@example.com
  color.status=auto
  color.branch=auto
  color.interactive=auto
  color.diff=auto
  ...
```

For more configuration options, see [this page](#).

2 Starting a local repository

```
$ mkdir learngit
$ cd learngit
$ git init
```

The `git status` command can be used to obtain a summary of which files have changes that are staged for the next commit:

```
$ git status
On branch master

Initial commit

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

Create a new file and add/stage one of them:

```
$ touch file1.txt file2.txt
$ git add file1.txt
$ git status
On branch master
```

```
Initial commit

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

        new file:   file1.txt

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

        file2.txt
```

A file is tracked if it is under version control, i.e. it was once added and committed. If you do not want to track a file type, you can add them to `.gitignore`:

```
$ echo "*.tmp" >> .gitignore
$ touch file3.tmp
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore

nothing added to commit but untracked files present (use "git add" to track)
```

A `.gitignore` file can include both files and directories to exclude:

```
*.tmp
**/*.tmp
tmpdir/
**/tmpdir/
```

The `**/*.tmp` syntax means git should exclude any file with *.tmp* name in any directory, whereas a **.tmp* will only exclude *.tmp* files in the main directory.

If you already have staged a file, and you want to ignore it, Git will not ignore the file if you add a rule later. In those cases, you must untrack the file first, by running the following command in your terminal:

```
$ git rm --cached <FILENAME>
```

3 Comparing differences

You can see what is about to be committed using `git diff` with the `-cached` option: `$ git diff --cached`

(Without `-cached`, `git diff` will show you any changes that you've made but not yet added to the index.)

```
$ echo "hello" >> file1.txt
$ git diff
diff --git a/file1.txt b/file1.txt
index e69de29..ce01362 100644
--- a/file1.txt
+++ b/file1.txt
@@ -0,0 +1 @@
+hello

$ git diff
$ git diff --cached
diff --git a/file1.txt b/file1.txt
index e69de29..ce01362 100644
--- a/file1.txt
+++ b/file1.txt
@@ -0,0 +1 @@
+hello
```

This header is called set of change, or *hunk*. Each hunk starts with a line that contains, enclosed in `@@`, the line or line range **from,no-of-lines** in the file before (with a `-`) and after (with a `+`) the changes. After that come the lines from the file. Lines starting with a `-` are deleted, lines starting with a `+` are added. Each line modified by the patch is surrounded with 3 lines of context before and after.

4 Viewing previous commits

The `git log` command displays committed snapshots. It lets you list the project history, filter it, and search for specific changes. While `git status` lets you inspect the working directory and the staging area, `git log` only operates on the committed history.

```
$ git log
commit 1d0a7ab8ec7873b0fccc55f0b1659a82ae46fe7a
Author: Ali R. Vahdati <arv.kavir@gmail.com>
Date:   Mon Feb 5 22:40:15 2018 +0100

    Second commit

commit 1b8b5818ef36542402922a873fad14c3700b2cc5
Author: Ali R. Vahdati <arv.kavir@gmail.com>
Date:   Mon Feb 5 22:02:08 2018 +0100

    First commit
```

To see only the commits of a certain author:

```
$ git log --author=Ali
```

To see a very compressed log where each commit is one line:

```
$ git log --pretty=oneline
1d0a7ab8ec7873b0fccc55f0b1659a82ae46fe7a Second commit
1b8b5818ef36542402922a873fad14c3700b2cc5 First commit
```

Or maybe you want to see an ASCII art tree of all the branches, decorated with the names of tags and branches:

```
$ git log --graph --oneline --decorate --all
* 1d0a7ab (HEAD -> master) Second commit
* 1b8b581 First commit
```

See only which files have changed:

```
$ git log --name-status
commit 1d0a7ab8ec7873b0fccc55f0b1659a82ae46fe7a
```

```

Author: Ali R. Vahdati <arv.kavir@gmail.com>
Date:   Mon Feb 5 22:40:15 2018 +0100

    Second commit

A       .gitignore
M       file1.txt

commit 1b8b5818ef36542402922a873fad14c3700b2cc5
Author: Ali R. Vahdati <arv.kavir@gmail.com>
Date:   Mon Feb 5 22:02:08 2018 +0100

    First commit

A       file1.txt
A       file2.txt

```

These are just a few of the possible parameters you can use. For more, see `git log --help`

5 Reverting a mistake

In case you did something wrong, you can replace local changes using the command `git checkout <filename>`. this replaces the changes in your working tree with the last content in HEAD. Changes already added to the index, as well as new files, will be kept.

```

# I first make a "wrong" change in file1.txt
$ echo "mistake" >> file1.txt
# and I add a good change to file2.txt
$ echo "correct change" >> file2.txt
# now I stage file2.txt
$ git add file2.txt

# Now I try to revert changes in file1.txt

```

```

$ git checkout file1.txt
$ 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:   file2.txt

# Reverting back file2.txt, however, does not work as it was staged
$ git checkout file2.txt
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   file2.txt

# to unstage the file:
$ git reset file2.txt

```

6 Managing branches

6.1 fast-forward merge

First, I commit any changes we have made so far, so that the directory is clean:

```

$ git add .
$ git commit -m "Second commit"
[master 1d0a7ab] Second commit
2 files changed, 2 insertions(+)
create mode 100644 .gitignore

```

Then, I create a new branch, work on some files, and when I am happy about the changes, I merge the new branch into the **master**.

```
# Start a new feature
$ git checkout -b new-feature
# see which branch you are working in
$ git branch
  master
* new-feature
# Edit some files
$ echo "more edits" >> file1.txt
$ git add file1.txt
$ git commit -m "Start a feature"
[new-feature ea464b4] Start a feature
 1 file changed, 1 insertion(+)
$ # Edit some files
$ echo "even more edits" >> file1.txt
$ git add file1.txt
$ git commit -m "Finish a feature"
[new-feature 8d118f4] Finish a feature
 1 file changed, 1 insertion(+)
# Merge in the new-feature branch
$ git checkout master
Switched to branch 'master'
$ git merge new-feature
Updating be9e536..8d118f4
Fast-forward
  file1.txt | 2 ++
  1 file changed, 2 insertions(+)
# remove the new branch
$ git branch -d new-feature
Deleted branch new-feature (was 8d118f4).
```

‘git branch -d new-feature’ makes sure that you have committed changes before deleting a branch. Use **git branch -D <branch>** to force delete the specified branch, even if it has unmerged changes. This is the command to use if you want to permanently throw away all of the commits associated with a particular line of development.

In the event that you require a merge commit during a fast forward merge for record keeping purposes you can execute `git merge` with the `--no-ff` option.

```
git merge --no-ff <branch>
```

This command merges the specified branch into the current branch, but always generates a merge commit (even if it was a fast-forward merge). This is useful for documenting all merges that occur in your repository.

6.2 3-way merge

```
# Start a new feature
$ git checkout -b new-feature
Switched to a new branch 'new-feature'
# Edit some files
$ echo "some changes" >> file2.txt
$ git add file2.txt
$ git commit -m "Start a feature 2"
[new-feature 1c990cb] Start a feature 2
 1 file changed, 1 insertion(+)
# Edit some files
$ echo "some more changes" >> file2.txt
$ git add file2.txt
$ git commit -m "Finish a feature 2"
[new-feature dd12f4a] Finish a feature 2
 1 file changed, 1 insertion(+)
# Develop the master branch
$ git checkout master
# Edit some files
$ echo "some changes on master branch" >> file2.txt
$ echo "some changes on file1" >> file1.txt
$ git add .
$ git commit -m "Make some super-stable changes to master"
# Merge in the new-feature branch
$ git merge new-feature
Auto-merging file2.txt
```

```
CONFLICT (content): Merge conflict in file2.txt
Automatic merge failed; fix conflicts and then commit the result.
# manually resolve the conflicts, then:
$ git add file2.txt
$ git commit
$ git branch -d new-feature
```

7 Resolving conflicts

When Git encounters a conflict during a merge, It will edit the content of the affected files with visual indicators that mark both sides of the conflicted content. These visual markers are: <<<<<<, =====, and >>>>>>. Its helpful to search a project for these indicators during a merge to find where conflicts need to be resolved. here is some content not affected by the conflict

```
<<<<<< HEAD
some changes on master branch
=====
some changes
some more changes
>>>>>> new-feature
```

Generally the content before the ===== marker is the receiving branch and the part after is the merging branch.

When you're ready to finish the merge, all you have to do is run `git add` on the conflicted file(s) to tell Git they're resolved. Then, you run a normal `git commit` to generate the merge commit. It's the exact same process as committing an ordinary snapshot.

8 Working with remote repositories

8.1 Clone an existing repository

I have created an online repository on Github of all we have done so far. You may delete the directory `learngit` and all its contents. We will grab them from the online repository:

```
$ cd ../
$ rm -r learngit
```

Now copy the address of [my online repository](#) (see Figure 1) and clone it on your computer:

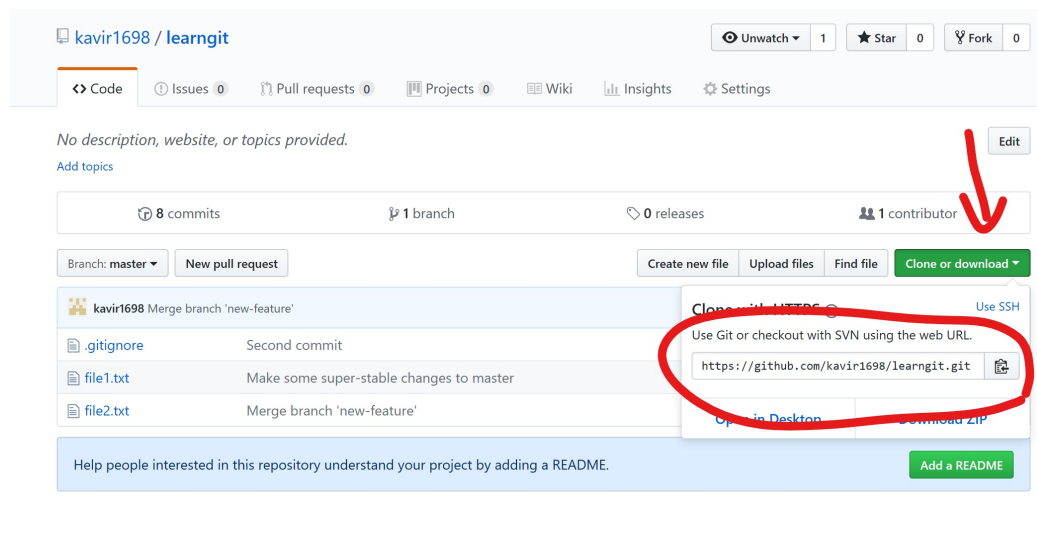


Figure 1: Copy the address that is shown in the figure. The git repository is also accessible [here](#)

```
$ git clone https://github.com/kavir1698/learngit
Cloning into 'learngit'...
remote: Counting objects: 27, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 27 (delta 4), reused 27 (delta 4), pack-reused 0
```

```
Unpacking objects: 100% (27/27), done.  
Checking connectivity... done.
```

Now you can start working with the same workspace and make commits. If you want to update the online repository (i.e. remote), you can use the following command:

```
git push <remote-name> master
```

Where is typically origin, the name which git gives to the remote you cloned from. The remote branch is automatically created when you push it to the remote server.

If you are on a new branch and want to push to a new branch on remote:

```
git push <remote-name> <branch-name>
```

8.1.1 Always pull before pushing

It is best to always pull before you push, because if someone have pushed changes to the server after your last pull, your local copy and the current server copy would not not in sync. Pulling will merge the remote copy with your local one, which brings them back into sync and allows you to push.

If you push straight away, it may need merging and might lead to conflicts which need manual resolving. To avoid merging errors and conflicts, merging must always be done on the client side, never on the server.

9 Other useful material

9.1 Tagging

You can tag ceertain commits to find them easier. Usually major releases of code are tagged, e.g. each version of a software. You can create a new tag named 1.0.0 by executing

```
git tag 1.0.0 1b2e1d63ff
```

the `1b2e1d63ff` stands for the first 10 characters of the commit id you want to reference with your tag. You can get the commit id by looking at the `git log`

9.2 When to commit

[Here](#) is a good advice on when to commit:

I make a commit when:

- I complete a unit of work.
- I have changes I may want to undo.


Anytime I satisfy one of these rules, I commit that set of changes. To be clear, I commit only that set of changes. For example, if I had changes I may want to undo and changes that completed a unit of work, I'd make two commits - one containing the changes I may want to undo and one containing the changes that complete the work.

Let's start with how **not** to define a unit of work. A unit of work is absolutely not based on time. Making commits every X number of minutes, hours, or days is ridiculous and would never result in a version history that provides any value outside of a chronicling system.

A unit of work is not based on the type of change. Making commits for new files separate from modified files rarely makes sense. Neither does any other type abstraction: code (e.g. JavaScript vs HTML), layer (e.g. Client vs API), or location (e.g. file system).

So if a unit of work is not based on time or type, then what? I think it's based by feature. A feature provides more context. Therein making it a far better measurement for a unit of work. Often, implicit in this context are things like time and type, as well as the nature of the change. Said another way, by basing a unit of work by feature will guide you to make commits that tell a story.

9.2.1 The seven rules of a great Git commit message



	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT
MESSAGES GET LESS AND LESS INFORMATIVE.

Figure 2: An example of commits you can easily find on Github

- Separate subject from body with a blank line
- Limit the subject line to 50 characters
- Capitalize the subject line
- Do not end the subject line with a period
- Use the imperative mood in the subject line
- Wrap the body at 72 characters
- Use the body to explain what and why vs. how

Read [this blog post](#) for more details.

9.3 Stashing and cleaning

Often, when you've been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don't want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the git stash command.

Stashing takes the dirty state of your working directory — that is, your modified tracked files and staged changes — and saves it on a stack of unfinished changes that you can reapply at any time (even on a different branch).

To see which stashes you’ve stored, you can use `git stash list`:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

In this case, we have three different stashes. You can reapply the one you just stashed by using the command shown in the help output of the original stash command: `git stash apply`. If you want to apply one of the older stashes, you can specify it by naming it, like this: `git stash apply stash@{2}`. If you don’t specify a stash, Git assumes the most recent stash and tries to apply it:

The apply option only tries to apply the stashed work — you continue to have it on your stack. To remove it, you can run `git stash drop` with the name of the stash to remove: `$ git stash drop stash@{0}`

You can also run `git stash pop` to apply the stash and then immediately drop it from your stack.

Another common thing you may want to do with stash is to stash the untracked files as well as the tracked ones. By default, `git stash` will stash only modified and staged *tracked* files. If you specify `--include-untracked` or `-u`, Git will include untracked files in the stash being created.

9.3.1 Creating a Branch from a Stash

If you stash some work, leave it there for a while, and continue on the branch from which you stashed the work, you may have a problem reapplying the work. If the apply tries to modify a file that you’ve since modified, you’ll get a merge conflict and will have to try to resolve it. If you want an easier way to test the stashed changes again, you can run `git stash branch <branch>`, which creates a new branch for you with your selected branch name, checks

out the commit you were on when you stashed your work, reapplies your work there, and then drops the stash if it applies successfully:

```
$ git stash branch testchanges
M index.html
M lib/simplegit.rb
Switched to a new branch 'testchanges'
On branch testchanges
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

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:   lib/simplegit.rb

Dropped refs/stash@{0} (29d385a81d163dfd45a452a2ce816487a6b8b014)
```

This is a nice shortcut to recover stashed work easily and work on it in a new branch.

10 Resources

We covered the basic, but probably most common, use cases of Git. For more details, you may use the resources below:

- [Visual Git cheat-sheet](#)
- [Git's official tutorial](#)
- [Recommended git workflows](#)
- [Book: pro Git by Scott Chacon](#)
- [A comprehensive online git tutorial](#)
- [Git Immersion](#)