Git

a quickstart guide

INDEX

1. WHAT IS GIT

Git is a *Distributed Version Control System*, which allows the user to track changes through different version of a project, displaying the authors of such changes, and allowing the user to access older versions of the project. Being a DVCS, Git keeps a copy of the whole repository locally, meaning that every user has the whole repository, instead of a standard VCS, where the user checks out the working repository but the whole repository is still on the server, meaning that should the server go down, the repository won't be accessible, which might cause data loss.

2. INSTALLATION

2.1. LINUX

Installing Git on Linux is fairly simple, on most operative systems, via the package management tool that comes with most OS:

```
sudo yum install git-all
```

or, with a Debian based operative system, use

```
sudo apt-get install git-all
```

For other Linux distributions, check git-scm.com/download/linux.

2.2. WINDOWS

Installing Git on Windows is as simple as downloading and running the executable file, available here. Alternatively it is possible to install Git by installing Github for Windows, which is available for download desktop.github.com/.

2.3. MAC

To install Git on Mac, download and run the binaries available for download git-scm.com/download/mac. It is also possible to install Git by installing Github for Mac, downloadable desktop.github.com/.

3. FIRST STEPS

3.1. CREATING A LOCAL REPOSITORY

Once you have installed Git, you need to move to the directory where the pippo repository will be located, and, run the following command via Bash or, on Windows, either the Git Bash or the Windows Command Prompt:

```
git init
```

This initializes a git repository in the directory you're at, creating a .git hidden folder which you don't need to access in order to use git, but contains the pointers to the current file, the configuration for the repository and so on. Once initialized a repository, you need to configure it, via the `git config` command, which accepts several fields for its arguement, and their relative setting. For a basic configuration however, you just need to set up your identity, composed of name and email address.

```
git config user.name "name here"

git config user.email email@address.com
```

These 2 commands set your username and email address, which are used to identify authors of changes in a project; Git and Github record changes by email address. These commands though set your credentials only for the current repository; to set them globally on all repositories you might work on, you need to pass the `--global` option:

```
git config --global user.name "name here"

git config --global user.email email@address.com
```

It is also possible to set up a text editor for Git to use when it needs the user to enter text, as in commit comments and the like:

```
git config --global core.editor emacs
```

On Windows, you need to specify the directory where the text editor is located. If not specified, Git will use your system's default text editor.

Once the basic configuration is done, you can check your settings with:

```
git config user.name
```

This command displays the value to that key. Alternatively it is possible to output the various settings, showing a list of all the available options.

```
git config --list
```

When done with the configuration it is time to start working with the repository.

3.2. ADDING FILES TO THE REPOSITORY

The way Git works is fairly simple: all files present in the repo directory are considered *untracked*, which means that the files present are not part of the repo, although they are in the repo directory. Git assigns different states for files, indicating whether a file is part of the repo or not, has been modified or will be added. To check the state of the file use the command:

```
git status

On branch master

Initial commit

Untracked files:
(use "git add <file>..." to include in what will be committed)
README.md
nothing added to commit but untracked files present (use "git add" to
track)
```

In this example, `git status` tells us that the `README` file is not tracked, so it will not be added to the repo in case of a commit. In order to tell Git to track the file and the various changes, we will need to add the `git add` command:

```
git add README.md
```

Now if we run the status command again, we'll see that the file has been added to the staging area, which means that the file will be added on the next commit.

```
git status

On branch master

Initial commit

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

new file:   README.md
```

Suppose we have another file though, which is already part of the repository, let's call it `stuff.txt`. If we modify that file and run the status command, we'll see something like this:

```
git status

On branch master

Initial commit

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

new file:   README.md

Changes not staged for commit:
(use "git add <file>..." to update what will be committed)

Recording Changes to the Repository

(use "git checkout -- <file>..." to discard changes in working directory)

modified:   stuff.txt
```

This means that should we commit the changes to the repo, the modified `stuff.txt` will not be added, so we need to run an add command on that file too ion order to stage it. But before we stage it, we might want to check what we've actually changed about the file. To do so we'll use the `git diff` command.

```
$ git diff
diff --git a/stuff.txt b/stuff.txt
index dc1b047..a422a68 100644
--- a/stuff.txt
+++ b/stuff.txt
@@ -1,2 +1,3 @@
 this is a file
 there are many files like this
+but this is my file
-i like my file
warning: LF will be replaced by CRLF in stuff.txt.
The file will have its original line endings in your working directory.
```

This tells us what has changed in the file: the + tells us that something has been added to the file, and the - tells us that something has been removed. Note that the `README.md` file is not listed in the `diff` output because the file has been staged and is ready to be committed.

```
git status

On branch master

Initial commit

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

new file:   README.md
modified:   stuff.txt
```

3.3. COMMITTING CHANGES

A commit means that everything that was staged will be added to the repository, and a new snapshot will be taken. To commit changes simply type:

```
git commit
```

Doing so however isn't quite enough. If you run that command git will open the text editor and ask that you input a message. All commits must have a comment, generally describing what it does. Common practice on the Git community is to add comments in the simple present, for example: "Update changelog". Once the message has been entered, exit the editor and git will work its magic and commit the changes, outputting something like this:

```
[master cb8f0cf] update changelog
 1 file changed, 1979 insertions(+)
 create mode 100644 README.md
 create mode 100644 stuff.txt
```

Git will open the default text editor, usually vi or emacs. You can change that using the core.editor option with the config command.

Suppose we didn't commit though, because we wanted to do everything in one go, so the files are still staged and ready to be committed. In order to commit and enter a comment at the same time, you need to run the following command:

```
git commit -m 'update changelog'

$ git commit -m 'update changelog'
[master cb8f0cf] update changelog
 1 file changed, 1979 insertions(+)
 create mode 100644 README.md
 create mode 100644 stuff.txt
```

3.4. UNDOING CHANGES TO THE REPOSITORY

Now, we've added our files to the repo, but later on, we realize that we've made a mistake, and need to undo something. In this case Git offers some ways to undo things, although it is possible to lose data undoing something by mistake.

First off we'll see the `--amend`, which allows us to add a file we might have forgotten to stage before our last commit:

```
git commit -m 'another commit'
git add forgotten-file.xml
git commit --amend
```

This will add the `forgotten-file.xml` to the last commit we made, though it will open up the text editor and allow us to change the commit message. What if we staged some files by mistake though? If we run the `git status` command, git will tell us what to do:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.md
```

In order to unstage a staged file, we need to run the `git reset HEAD <file>` command, we'll see something like this:

```
$ git reset HEAD README.md
Unstaged changes after reset:
M       README.md
```

This removes the file from the staging area, and a run of the `git status` command will tell us that the file is unstaged. Suppose we modified a file and committed the changes and later on we want to revert to a previous version of the file: Git gives us a way to do so with the `git checkout` command:

```
git checkout -- file
```

This replaces the current file with the last version of the file itself. Be careful though, since the command overwrites the file, the content of the file is lost.

3.5. THE LOG

The log, displayed with `git log` is a powerful tool that shows us the commit history of our repository.

```
git log

commit aedb3b0a02afb5105ba9b2a962fdc93923429412
Author: Alex Zuan <studioquattrodue@gmail.com>
Date:   Tue May 17 12:05:01 2016 +0200

    file
```

By default, the log passed without options shows the commit's SHA-1 checksum, the commit author, the commit date and the commit message. The log sorts every commit in reverse chronological order, meaning that most recent snapshots of the repo are at the top of the list.

One of the most useful options is `-p` which shows the differences introduced with each

 Copyright © Jmatica srl - Tutti i diritti riservati.

commit. We can limit the output to a set number of commits, by passing a second option, `-x` where x is the number of commits to display.

```
git log -p -2

commit f057005cccd83ad4dd363fb9674fc225a272bf22
Author: Alex Zuan <studioquattrodue@gmail.com>
Date:   Tue May 17 12:05:48 2016 +0200

    file.txt-master

diff --git a/file.txt b/file.txt
index 93d88e9..c063fce 100644
--- a/file.txt
+++ b/file.txt
@@ -1 +1,2 @@
 some other file is there
+because i forgot to add it

commit aedb3b0a02afb5105ba9b2a962fdc93923429412
Author: Alex Zuan <studioquattrodue@gmail.com>
Date:   Tue May 17 12:05:01 2016 +0200

    file

diff --git a/file.txt b/file.txt
index 93d88e9..da4a5ef 100644
--- a/file.txt
+++ b/file.txt
@@ -1 +1,3 @@
 some other file is there
+
+meaning that another file is present there.
```

This shows that some lines were added to `file.txt`, shown by prepending them with a `+`. The `--pretty` option changes the output fromat of the log, with 3 possible settings:

- `--pretty=oneline`
- `--pretty=short`
- `--pretty=full`
- `--pretty=fuller`

`oneline` displays the information in a single line, while `short`, `full` and `fuller` display an increasing amount of information about the commit, such as author, committer and dates. Another option for the `--pretty` arguement is `format` which

lets the user choose what to display:

|Option |Displays |%H |Commit hash |%h |Abbreviated commit hash |%T |Tree hash |%t |Abbreviated tree hash |%P |Parent hashes |%p |Abbreviated parent hashes |%an |Author name |%ae |Author email |%ad |Author date (format respects the --date=option) |%ar |Author date, relative |%cn |Committer name |%ce |Committer email |%cd |Committer date |%cr |Committer date, relative |%s |Subject

The log also can output, with the `--graph` option, an ascii graph that shows the branching of thhe current repository along with the commit history:

```
git log --pretty=short --graph

*   commit 28cafabd833494f26413042cbb75272e40373f10
|\  Merge: f057005 aedb3b0
| | Author: Alex Zuan <studioquattrodue@gmail.com>
| |
| |     merge test into master
| |
| * commit aedb3b0a02afb5105ba9b2a962fdc93923429412
| | Author: Alex Zuan <studioquattrodue@gmail.com>
| |
| |     file
| |
* | commit f057005cccd83ad4dd363fb9674fc225a272bf22
|/  Author: Alex Zuan <studioquattrodue@gmail.com>
|
|       file.txt-master
```

One of the most useful things about the log though is the possibility to limit the output in several ways, for example to show the commit history of a single author, or the commits done in a certain period. To do that we have the following options:

|Option |Example |Function |-(n) |git log -n |Show only the last n commits |--since, --after |git log--since 2016-01-31 |Limit the commits to those made after the specified date. |--until, --before |git log--until 2015-12-31 |Limit the commits to those made before the specified date. |--author |git log--author=name |Only show commits in which the author entry matches the specified string. |--committer |git log--committer=name |Only show commits in which the committer entry matches the specified string. |--grep |git log--grep=abc |Only show commits with a commit message containing the string. |-S |git log-Smethod |Only show commits adding or removing code matching the string. (useful to find commits that worked with specific code parts)

The date specified in `since`, `after`, `before` and `until` can be full dates, or relative dates, as in `--since=12.hours` or `5.days` and so on.

4. WORKING WITH REMOTE REPOSITORIES

A remote repository is, as the name suggests, a repo that is hosted one a different machine or directory, which we can use to store data.

4.1. BARE REPOSITORIES

At the creation of our remote, we have 2 choices: we can add a remote with a working directory, which means that the files will be stored in the remote and we can edit them from there, or we can create a `bare` repo, which won't have a working directory, meaning that we won't be able to access the files from there. Generally bare repositories are used to host the data on a server, while regular repos are used locally for edits. Creating a bare repository means initializing a repo with the `bare` option, as shown in the example:

```
git init --bare
```

Another way to create a bare repo is to clone the content of a repository into a directory, again, with the `bare` option.

```
git clone --bare <url>
```

Another important difference between a regular repository and a bare one is that we cannot push to a regular repository's branch if that branch is currently checked out. In this section, we'll work with a bare remote.

4.2. ADDING A REMOTE REPOSITORY

To add a remote repository we simply need to type `git remote add` and some arguements, as shown in the example:

```
git remote add <name> <URL>
```

This will allow us to get from and send data to the remote given that we have access to

the specified URL, and simplify the command by using the remote name insead of the full url when we need to work with it.

4.3. COMMITTING CHANGES TO A REMOTE REPOSITORY

Now that we've added our remote we might want to check what's in there, as both need to be up to date, and we won't be able to push data to the remote if the remote has been modified. In order to do so we'll use the `remote show` command:

```
git remote show <name>
* remote <name>
Fetch URL: <url>
Push  URL: <url>
HEAD branch: (unknown)
```

This tells us that the remote we're using is empty, as it has no `HEAD` branch. We can now add data to our remote, via the `push` command.

```
git push <remote> <branch>
```

The command needs the remote name, and the branch we're trying to push to, so if we wanted to push to a remote named `test`, on its `master` branch, we'd type this:

```
$ git push test master
Counting objects: 101, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (76/76), done.
Writing objects: 100% (101/101), 272.04 KiB | 0 bytes/s, done.
Total 101 (delta 39), reused 21 (delta 8)
To C:/prova.git
 * [new branch]      master -> master
```

This tells us that a new brach named `master` was created on our remote, and that a snapshot of our local repo's current branch (master) was taken. Pushing is the way to commit changes to a remote, meaning that if we have a working directory on our remote, every file we push will overwrite the same file present on our remote.

4.4. GETTING DATA FROM A REMOTE REPOSITORY

Unless we initialize a repository and then add a remote, we might add a remote that already contains data we need to work on, so we need to get the files from that remote to our own local repo. There are three ways to do that in git: clone, fetch and pull. First we'll go over the `clone` command:

```
git clone <url> [folder]
Cloning into 'folder'...
done.
```

With the `clone` command Git takes everything present in the repository at the specified url and copies it into a folder of which we might specify the name. If we do not specify the name of the folder that will contain the cloned repository, Git will create a folder with the same name of the folder containing the repository we are cloning. Using `clone` basically tells Git to perform a `git init` on the new folder and then it will perform a `fetch` to get the data.

the `clone` command does not work with remote shortnames, you need to specify the url of the repository.

The `git fetch` command downloads all data that was pushed or added to the remote since the last fetch or clone, without merging it with existing files or modifying your work.

```
git fetch <remote name>
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From <remote URL>
   a7d7f94..4914003  <local branch>     -> <remote name>/><remote brach>
   * [new branch]    <local branch>       -> <remote name>/<remote
branch>
```

Finally, another way to get data from a remote is with the `git pull` command, which performs a `fetch` from the remote, but also merges the fetched files into the ones present in the working directory.

```
git pull <remote name>
From <remote URL>
 * branch            <local branch>       -> FETCH_HEAD
Updating a7d7f94..4914003
Fast-forward
 test.md | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 test.md
```

The `git pull` command requires that you also specify a branch from which to download data, and it will merge the downloaded files with the ones on the branch that is currently checked out.

5. BRANCHING

We've mentioned branches several times over in the previous sections, and here we'll cover what a branch is and its functionalities.

5.1. WHAT IS A BRANCH

A branch is a section of a repo independent from the others. Each branch of a repository will have different working directories, and files which can be edited independently. Branches are commonly used to develop and test new features, separately, before integrating them with the main branch.

5.2. MAKING A BRANCH

The `git branch` command allows us to see the branches that are present in our repository as well as creating a new branch. By default every Git repository has a main branch called `master`.

```
git branch test
```

This command creates a new branch for us to work on. The new branch will start with the same content as the branch it was split from, but now the `test` branch and the `master` branch are independent of each other, meaning that a commit on the test branch will not affect content of the master branch. If we now want to work on the new branch though, we need to use the `checkout` command, as we still are working on the master branch.

```
git checkout test
Switched to branch 'test'
```

If a branch is present on a remote, Git will also tell you the current state of the branch relative to the remote branch; your local branch can be:

- up to date:
    - both local and remote branches have the same versions of files
- out of date:
    - your local branch is not up to date with the remote
- ahead by x commits
    - x commits done locally have not been pushed to the remote branch

Now that we've checked out the test branch, we can perform some changes there, and commit them. We can also push the new branch on the remote, by typing

```
git push <remote> test
```

This will create a new branch on the remote, with the same commit history of our branch.

Another way to create a new branch uses the `-b` option of the checkout command:

```
git checkout -b test2
Switched to branch 'test2'
```

The `-b` option of the checkout command creates and checks out a new branch, named `test2`.

5.3. MERGING BRANCHES

Once we're done working on our separate branch, we may want to include those changes onto the main branch, and this is done by merging branches together. Merging performs a three way merge between your current branch, the branch you want to merge with and their most recent common commit. In order to merge test back into master then we need to switch to master fist, and then merge the two branches.

```
git checkout master
Switched to branch 'master'

git merge test
```

At this point Git will open up the text editor and as you to insert a commit message for the merging. Once done, the merge will be done, and the content of the `test` branch will be copied over to the `master` branch.

5.4. REBAISNG

Rebasing is another way of merging, but works in a different way. Insead of performing a three way merge, rebasing commits all the changes on a branch ontop of another one. The branch order here is inverted: when we run the `git rebase` command, it will rebase our current branch ontop of the target, while with the `git merge`, the target is merged into our current branch.

```
git checkout test2
Switched to branch 'test2'

git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

What we get from that is that our test branch has now been moved to our master branch, and our master branch is one commit behind. In order to fix that, we can perform a fast-forward merge:

```
git checkout master
Switched to branch 'master'

git merge test
```

Rebasing is a way of merging branches that leaves with a cleaner commit history. It also works in a way more similar to commits rather than merge, which makes it a powerful but also dangerous tool, especially when working in a distributed enviornment.

5.5. CONFLICT RESOLUTION

Most of the times Git will solve conflicts between files by cpying the newest file over the oldest, this however happens only if one of the files hasn't been modified. For example if we branch out of our master branch, edit a file on a test branch and then merge back

in.

If when we've branched out from master we did some work on a file, and then made a commit, and did the same on our secondary branch, Git will put the merging on hold until we've resolved all the conflicts. The changes of both branches will be inserted on the conflicting file in our current branch, and we'll have to manually fix that. Git tells us where the conflict is in the file by placing markers showing the conflicting content of the current branch frist, then the content to be merged.

```
<<<<<<<< HEAD:file.txt
Is this the real life?

Is this just fantasy?
=======
Is this real life? Is this just fantasy?
Caught in a landslide
>>>>>>> test:file.txt
```

Once we've fixed our file and resolved the conflict, we need to stage and commit the conflicting file to finalize the merging.

5.6. DELETING BRANCHES

Now that we're done working on our branches we can delete them. Merging and rebasing a branch doesn't remove it: a merged branch can still be worked on. To delete it we need to use the `-d` option of the `branch` command:

```
git branch -d test
Deleted branch test (was aedb3b0).
```

Once we're done with a remote branch, we can remove it from the remote with the following command:

```
git push <remote> --delete <branch>
To <remote URL>
 - [deleted]         <branch>
```