GIT

What is it?

Git is a version management software for working on collaborative projects (in groups).

This tool will help to share the work done and make 'backups'.

Similar to Google Drive, which allows multiple contributors to write, edit, and add content to a single text file, Git allows developers (and IT teams in general) to collaborate on a project.

Git will install on your computer and then manage the versions of your project.

Version Management / Version Control

Git will keep track of the changes you make to all your files. As you go, you will add files, modify them, remove them. Git will take care of taking 'captures' (these are backups at a time T) of the current version of your project. This is called version management (or version control).

Git will save these 'captures' in a chronological order. As a result, you can use git to navigate between versions of your project.

This is also very useful if when you make a mistake you can go back in time to the last good version before you have corrupted your code.

Install GIT

Git will install on your computer.

If you use MacOs: https://git-scm.com/download/mac
If you use Windows: https://gitforwindows.org/
If you use Linux: https://git-scm.com/book/en/ v2

To verify that Git is installed on your machine, type this small command:

```
git --version
```

First steps

Now that Git is installed on your machine, you will be able to start!

First, you have to configure some things.
Type these lines of command:

```
git config --global user.name 'your_Name'
git config --global user.email 'youremail@mail.com'
git config --global color.ui 'auto'
```

Here we configure our name and email address, they will serve us later. We also tell GIT to put cool colors on the terminal outputs.
The fact that it is global means that it is set up on your machine once and for all. You will not have to do this again on other projects.

It is also possible to define aliases.
For example, instead of typing 'git commit', I could type 'git ci'. Here is an example with two commands:

```
git config --global alias.ci commit
git config --global alias.st status
```

Now that you have configured GIT, it is time to create your project!

Create a *repository*

You will create an empty folder. This folder will contain your project.
In my case I call it '*TestGit*'.

Go to this folder and type this command:

git init

This command will create a new *git repo* (or git repository).
This repo is like the file of your project, but with the super powers of Git. A repository contains all your project files, as well as additional, mostly invisible, items generated by git to track and store the history of each file.

Add a README file

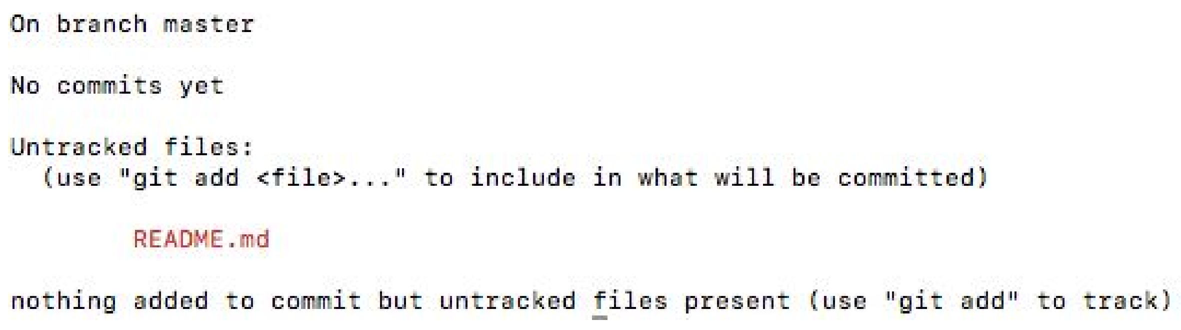
This is not required but a good practice.
You will start by creating a 'README' file to explain what your project is, what could be in it, and so on.
Add this file to your project.

Check the status of the repo

You can view the current state of the repository with the command:

git status

For now, the result of the command should look like this:



```
On branch master

No commits yet

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)
```

Adding files

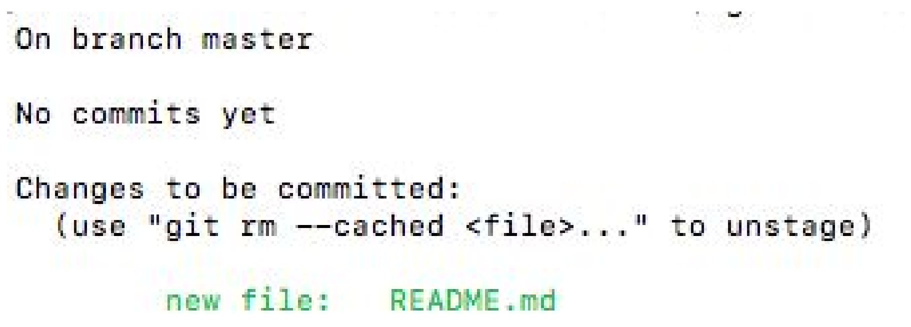
'git status' tells us that we have new files that have not been officially added to the git tracking process.
To tell git to add and follow these files, use the command:

git add README.md

You can add all files in this way (deprecated):

git.

Now, the command 'git status' shows us this:



```
On branch master

No commits yet

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

        new file:   README.md
```

Validate changes

Now that your files have been added and are 'tracked' by Git, you will be able to 'commit'.
A commit is like a 'snapshot' of a moment where you can come back, if necessary, to access your repo in its previous state.

To identify each of these captures, you must provide a message (through the -m option).

git commit -m 'first commit'

You have now started versioning your project locally!

Choosing remote hosting

Now it's time to register your project remotely (online)!
We call it a *remote repo*.
There are many hosting that can store code. We will use GitHub here.
So we go on GitHub to create a new repo (for now empty).
We will then 'link' our local repo to our remote repo.

git remote add origin https://github.com/InsteanAzeros/TestGit.git

You can now 'send' your code to GitHub:

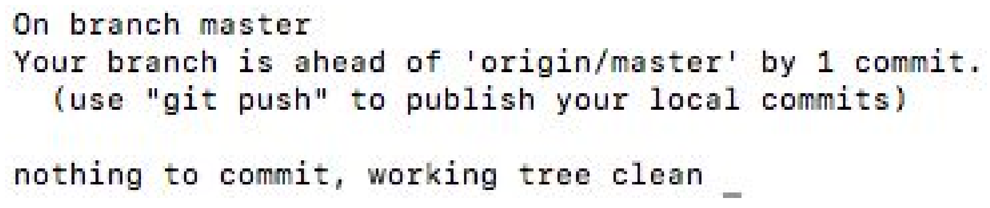
git push -u origin master

The -u option allows to permanently associate all the next 'git push 'on thebranch *master* of the deposit *origin*.
It is possible to have several distant rest, this one will be referenced by *origin*.

Compare local and remote

For the example, I will add a file '*test.java*' to the project.
Then I add it with the command '*git add * .java*'.
And I make a '*commit*' to validate my changes.

Now thanks to the status command, I can see that I'm ahead of a commit compared to my remote repo:



```
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean _
```

The branches

The main branch of your repo (*Master*) must always contain functional and stable code.
However, it is common to work on code that is not stable, such as adding features or debugging etc ... However, you need to save these changes. This is where the branch concept comes in.

It's possible to create branches that allows you to work on a separate copy of your project without affecting the main branch. When you create a branch, a complete clone of your main branch is created under a new name. You will then be able to modify the code of this new branch without impacting your main branch!

You can, when your code is stable, merge the two branches!

First, create a test branch:

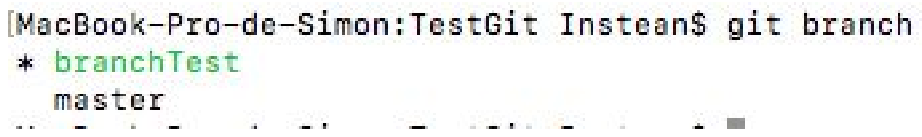
git checkout -b branchTest

A new branch is created and you are automatically 'transferred' to it.

Try this small command:

git branch

This command will list the branches available on your repo and color the one you are on:

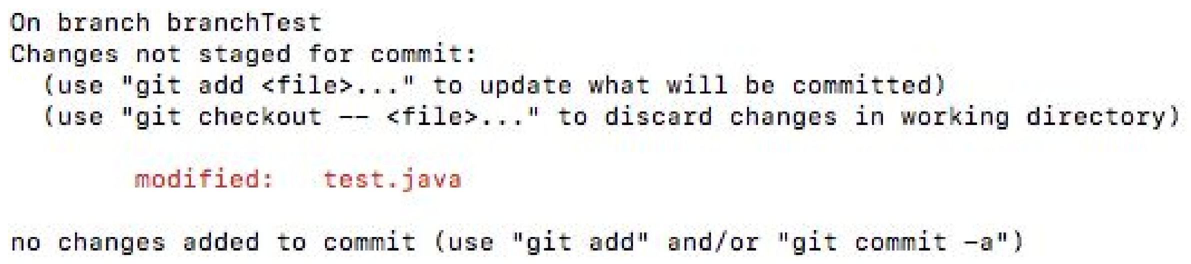


```
[MacBook-Pro-de-Simon:TestGit Instean$ git branch
* branchTest
  master
```

You can move from one branch to another with the command:

git checkout branchname

Now you will modify thefile *test.java* and see what happens with thecommand '*git status*':



```
On branch branchTest
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.java

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

You will save the changes:

git commit -am 'java test'

modifyThe -a option will automatically add changes to known files .

Merge branches

Now that the branch has been tested and validated, you will be able to integrate (merge) the main branch.
For that, we will first position ourselves on the main branch and tell it that it is the branch with which to merge.

git checkout master
git merge branchTest

Now that our development branch has been merged, it has become 'useless'. We will destroy it. The change history will not be lost even by destroying the branch.

git branch -d branchTest

Now that our repo contains a stable version, we will be able to transfer it to the remote repo:

git push

Clone a remote repo

If you want to retrieve the contents of a remote repo, for example if you join a project in progress , use this command:

git clone https://github.com/InsteanAzeros/TestGit

Release

With git, it is possible to publish specific versions of your code. Once the release is created, the branch loses the ability to change the commits history.

To publish a release we use tags.
We will tag branches and it is this mechanism that allows to simply manage the releases.

git tag -a v1.0 -m "Release 1.0"

the -a option creates an 'annotated tag'.
The -m option specifies the tag message, which is stored with the tag.

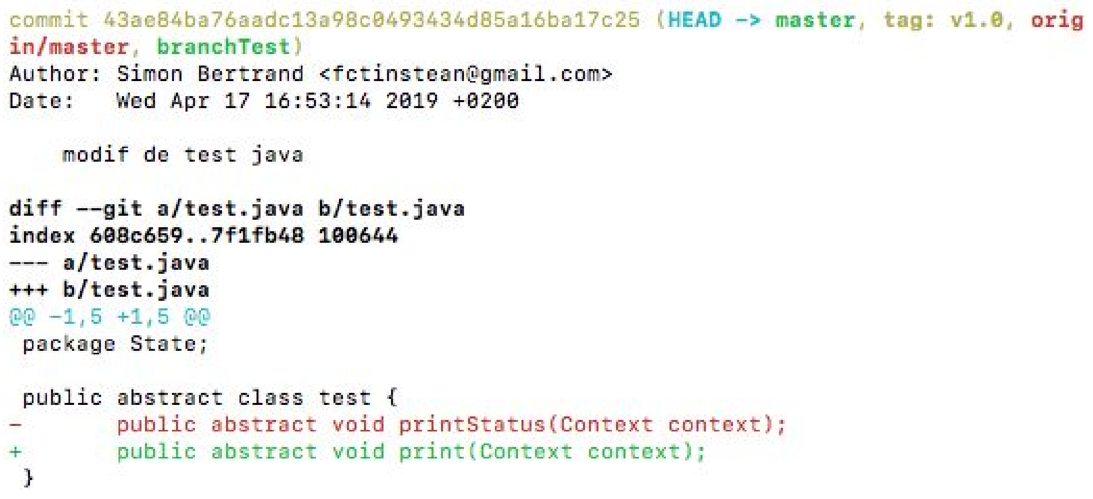
View Tags:

git tag

You can see the details of a tag:

git show v1.0

You should see something similar:



```
commit 43ae84ba76aadc13a98c0493434d85a16ba17c25 (HEAD -> master, tag: v1.0, orig
in/master, branchTest)
Author: Simon Bertrand <fctinstean@gmail.com>
Date:   Wed Apr 17 16:53:14 2019 +0200

    modif de test java

diff --git a/test.java b/test.java
index 608c659..7f1fb48 100644
--- a/test.java
+++ b/test.java
@@ -1,5 +1,5 @@
 package State;

 public abstract class test {
-       public abstract void printStatus(Context context);
+       public abstract void print(Context context);
 }
```

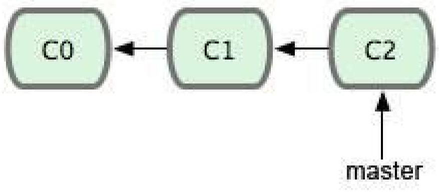
You can also annotate without tag, like this:

git tag -a v1.2

the option -a creates an 'annotated tag'.
The -m option specifies the message of thetag

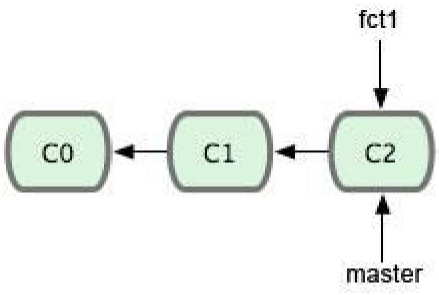
Understanding branches a little better

In this section, we will illustrate an example of branch usage.
Imagine a project where there is only the branch Master and already 3 commits.



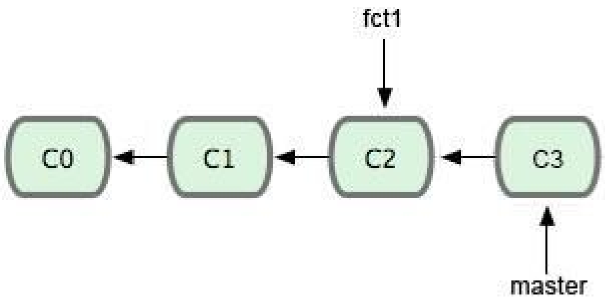
C0, C1 and C2 represent the commits.

Now, we need to develop a new feature. For this, we create a new branch 'fct1' (for feature number 1).

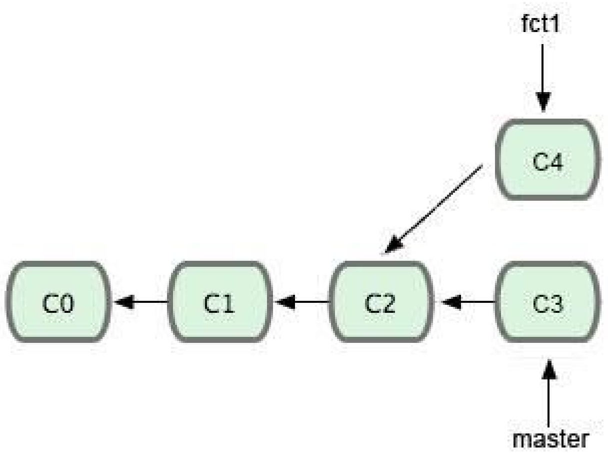


Git creates a pointer to the C2 commit.

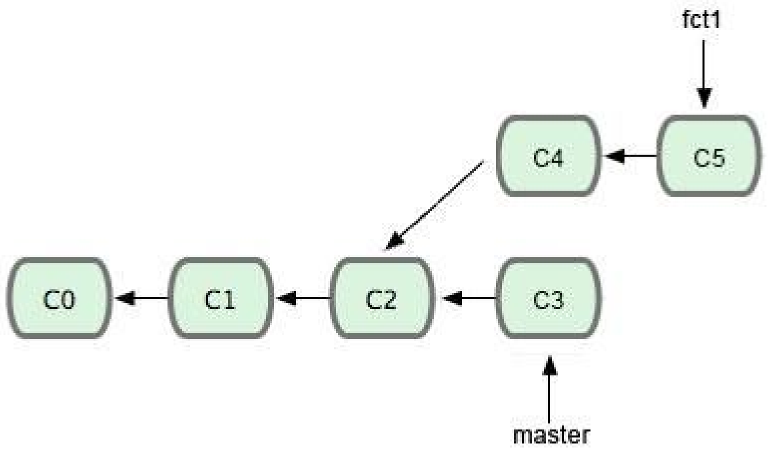
Then we will position ourselves on the master, make a change and commit.



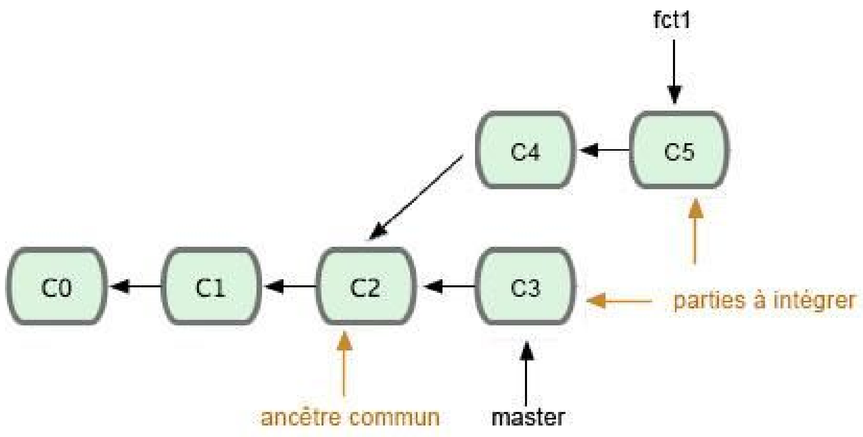
We will do the same on the branch 'fct1', change and commit.



We will make a second change and commit on the branch (fct1).



Our functionality is now operational, we will want to merge the branch fct1 in the master.



Git searches for the common root (here C2) to integrate the commits one by one and check conflicts by iteration from this root.

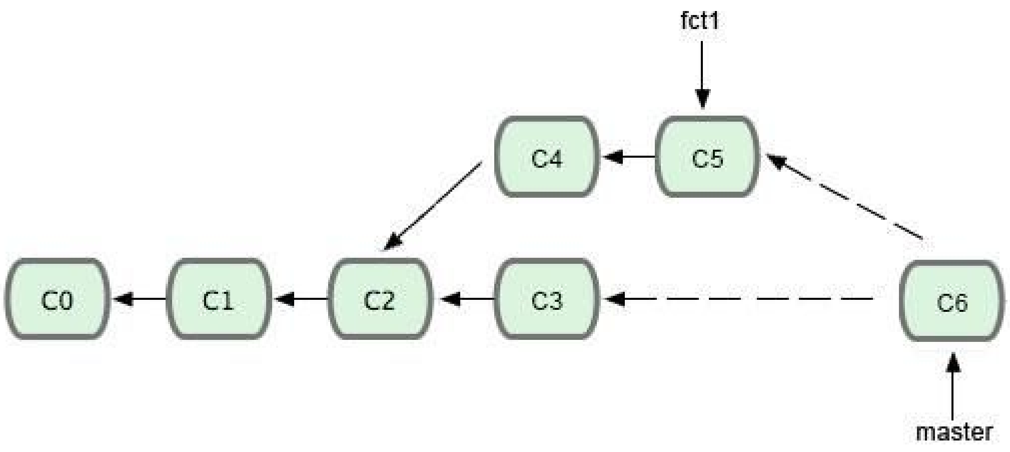


Illustration inspired by: https://git-scm.com/book/en/v2

Good practice (s)

<u>git ignore</u>

Attention to not version anything!
You choose the file (s) to be versioned in your project.
You can create a '.gitignore' file to prevent the addition of unwanted files.

Generally, one does not version executables, images, binaries etc ...

<u>Standard</u>

There are several standards as for the management of your projects under git.
Here is one.

Main branches:

Two main branches with an infinite lifespan: Master and Develop.

Master is an always stable branch and is used for deployment.
Develop is the branch of work that contains the latest version of the current code.
It therefore contains the latest development changes for the next release.

When the code of the Develop branch reaches a stable point and is ready to be deployed, all changes must be merged into the master and then tagged with a release number.

Secondary branches:

In addition to the main branches *master* and *develop*, you can use a variety of secondary branches to facilitate parallel development between team members, facilitate feature tracking, prepare production versions and help resolve quickly the problems of real production.

Unlike the main branches, these branches always have a limited life because they will eventually be removed.

The different types of frequently used branches:
> Feature branches
> Release branches

> Hotfix branches

> Feature branches
The feature branches are used to develop new features for the future or future version.
The branch exists as long as the feature is in development but will eventually be merged with Develop (to permanently add the new feature to the upcoming version) or removed (if a disappointing test).

> Release branches
Release branches support the preparation of a new production version.
In addition, they can fix minor bugs and prepare metadata for a version (version number, build dates, etc.).

The key moment to 'get out' of develop and create a release branch is when Develop is (almost) the desired state of the new version.
At least all features that are targeted for the build version must be merged to grow at this point.

> Hotfix branches Hotfix branches
are very similar to release branches as they are also intended to prepare a new production version, although not planned. They arise from the need to act immediately on an unwanted state of a production version.
When a critical bug in a production release needs to be resolved immediately, a hotfix branch is created.

SOURCE: https://nvie.com/posts/a-successful-git-branching-model/