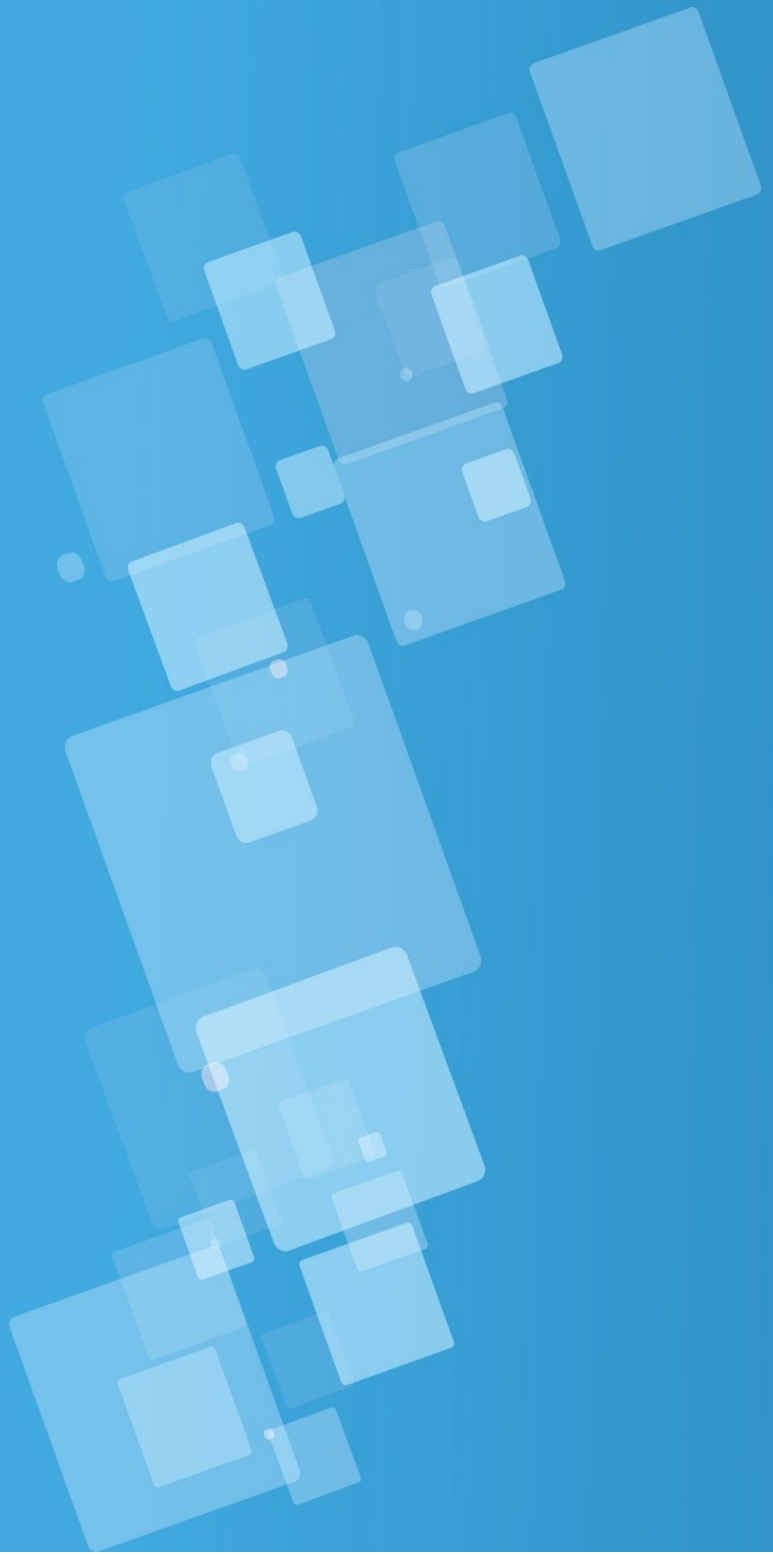


Version Control

leandro.frigeriodigitalcareerinstitute.org

Git

Git is a distributed version control and source code management system developed by Junio C. Hamano, Shawn O. Pearce, Linus Torvalds and others from 2005.

What is version control?

Version control is a system that **records and tracks changes** to a file(s) or directory over time.

How it works?

It takes a series of **snapshots** of your project, and it works with those snapshots to provide you with functionality to version and manage your source code.

Centralized Versioning vs. Distributed Versioning

Version Control System (VCS) can be
Centralized: SVN, CVS
Distributed: GIT

- Centralized version control focuses on synchronizing, tracking, and backing up files.
- Distributed version control focuses on sharing changes. Every change has a unique id.

https://www.atlassian.com/blog/software-teams/version-control-centralized-dvcs

Advantages Over Centralized Version Control

- Performing actions other than pushing and pulling changesets is extremely fast because the tool only needs to access the hard drive, not a remote server.
- Committing new changesets can be done locally without anyone else seeing them. Once you have a group of changesets ready, you can push all of them at once.
- Everything but pushing and pulling can be done without an internet connection. So you can work on a plane, and you won't be forced to commit several bugfixes as one big changeset.
- Since each programmer has a full copy of the project repository, they can share changes with one or two other people at a time if they want to get some feedback before showing the changes to everyone.

Why Use Git?

- Can work offline.
- Collaborating with others is easy!
- Branching is easy!
- Branching is fast!
- Merging is easy!
- Git is fast.
- Git is flexible

Git Architecture

- **Repository**
  A set of files, directories, historical records, commits, and heads. Imagine it as a source code data structure, with the attribute that each source code “element” gives you access to its revision history, among other things. A git repository is comprised of the .git directory & working tree.
- **.git Directory**
  The .git directory contains all the configurations, logs, branches, HEAD, and more. In most cases you will never need to touch this directory and you shouldn't do it if you don't know what you are doing.

  http://gitready.com/advanced/2009/03/23/whats-inside-your-git-directory.html

Git Architecture

- **Working Tree**
  Basically the directories and files in your repository.
  It is often referred to as your working directory.
- **Commit**
  A git commit is a snapshot of a set of changes, or manipulations to your Working Tree.
- **Index**
  is the staging area in git. It's basically a layer that separates your working tree from the Git repository. This gives developers more power over what gets sent to the Git repository.
- **Clone**
  A clone is a copy of an existing Git repository

Git Architecture

- **Branch**
  ”a lightweight movable pointer to one commit” (cit.)
- **Master**
  The default branch
- **Tag**
  a mark on specific point in history; typically used to mark release points (es: v1.0)
- **HEAD**
  a pointer to the tip of the current branch. It is simply a reference to a commit. A repository only has 1 active HEAD.

Init, add and commit

- **git init**
  Create an empty Git repository. The Git repository's settings, stored information, and more is stored in a directory (a folder) named ".git".
- **git add**
  add files to the staging area/index.
- **git commit**
  stores the current contents of the index in a new "commit."
  It contains the changes made and a message created by the user.
- **git status**
  show differences between the index file (and/or working tree) and the current HEAD commit

Share and Sync with a remote repository

Remote repository

A remote repository is simply a git project that is hosted on a remote system (e.g. GitHub, BitBucket, GitLab, ...). It is usually accessible through SSH or HTTPS.

Until now we have only worked locally on our repository. Now we want to share our code with the world by synching it to our newly created GitHub repository.

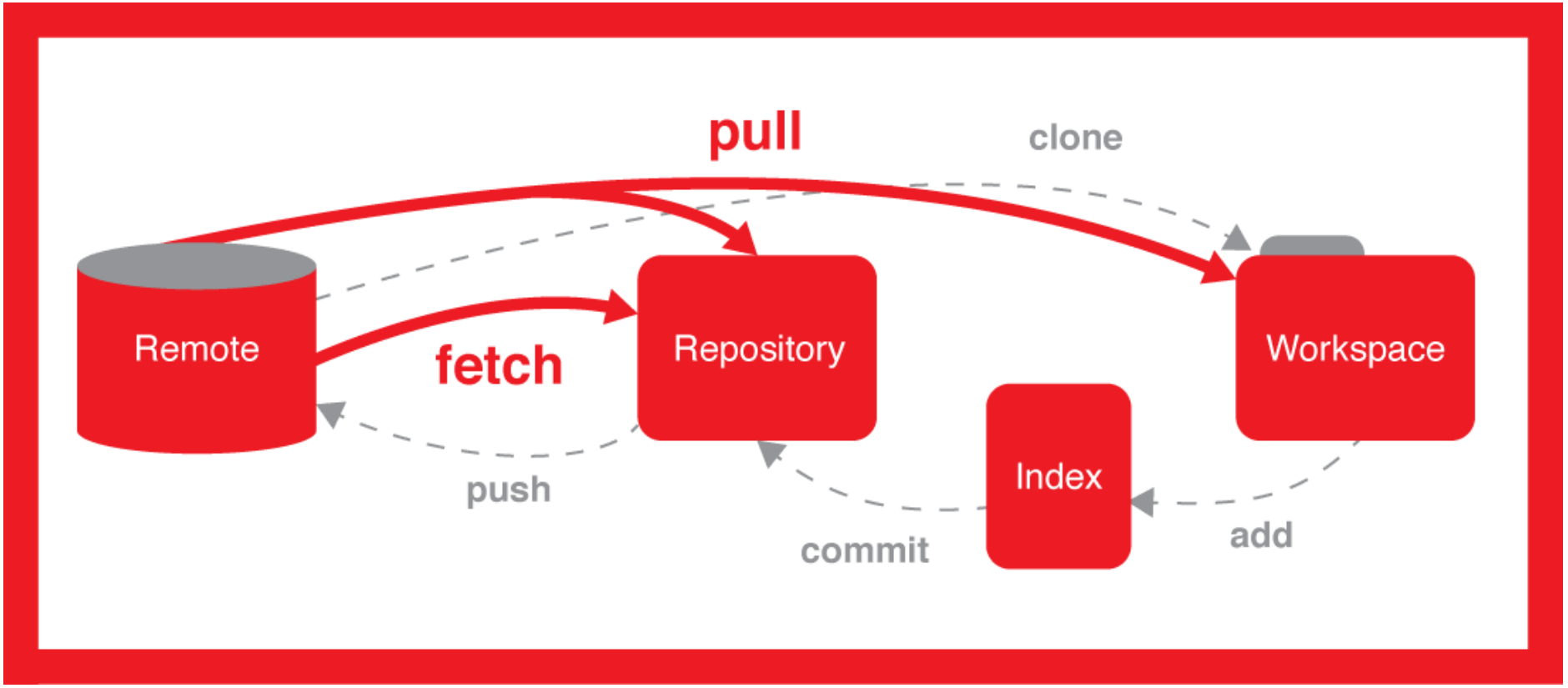
- We create a repo on Github
- We connect the local and the remote repositories

We can do this with the **git remote** command.

Established the connection between local/remote

- **git remote add origin https://github.com/FBW-12/collection.git**
  adding a remote to sync repos
- **git remote -v**
  lists all the remotes
- **git remote rm origin**
  remove a remote origin
- **git push**
  Push (upload) the new commits from a local branch to a remote branch.
- **git clone**
  download an exact copy of a git project
- **git pull**
  pull (download) the new commits from the remote branch to local branch

Remote vs Local vs Working Tree



Branching and merging

Branching and merging is what makes Git so powerful and for what it has been optimized, being a distributed version control system (VCS).

I**ndeed, feature branches are quite popular to be used with Git.**

Feature branches are created for every new kind of functionality you're going to add to your system and they are normally deleted afterwards once the feature is merged back into the main integration branch (normally the master branch).

The advantage is that you can experiment with new functionality in a separated, isolated "playground" and quickly switch back and forth to the original "master" branch when needed. Moreover, it can be easily discarded again (in case it is not needed) by simply dropping the feature branch.

Working with branches

- **git branch <branch_name>**
  create a branch
- **git branch -a**
  get a list of branches (-a is all, also remote!)
- **git branch -d <branch_name>**
  delete a branch
- **git branch -m <oldname> <newname>**
  rename a branch
- **git branch myBranchName –edit-description**
  edit a branch description

Checkout

Checkout is used to change branch

- **git checkout <branch_name>**
  switch to branch ‘branch_name’
- **git checkout master**
  switch to branch 'master'

- **git checkout -b my-feature-branch**
  same as *git branch my-feature-branch*
- ***git checkout -- my-file***
  checkout a file (discarding changes)

Checkout to a certain commit

Lets imagine we want to go back to a given commit.

We can use the **git log** command to get all the SHA identifiers that uniquely identify each node in the tree.

commit 4fef7b324d8d58a5678cdc57eaf822cd4c2307cb
Author: Leandro <leandro.frigerio@devugees.org>
Date: Wed Oct 17 13:49:06 2018 +0200

Okay, we need to take one of the identifiers (also if it isn't the whole one, it doesn't matter) and jump to that node by using the checkout command.

git checkout 4fef7b324d8d58a5678cdc57eaf822cd4c2307cb

Detached HEAD!

We checked out a previous commit and HEAD is now at 4fef7b3

This state is called 'detached HEAD'. We can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If we want to create a new branch to retain commits we create, we may do so (now or later) by using -b with the checkout command again.

git checkout -b <new-branch-name>

When we're done, we move back to master branch

git checkout master

Merge branches

Let's imagine we are happy with our **my-feature-branch.**

We want to add it back into our **master** branch so everyone else on our team and the public can see our glorious new feature.

The next step would be to merge our feature branch back into master:

git checkout master

git merge my-feature-branch

https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

Merge branches and resolve conflicts

We merged a branch but we encountered a problem:

Auto-merging hello.txt
CONFLICT (content): Merge conflict in Readme.md
Automatic merge failed; fix conflicts and then commit the result.

This is the content of the file after the merge conflict:

Hello, world!
<<<<<<< HEAD
Hi I was changed in master
=======
Hi
>>>>>>>
my-feature-branch

The top part, called HEAD represents the branch we are in, the master branch. The bottom part below the === line represents the my-feature-branch. In this case I want to keep both changes and I will simply remove the Git indicators. And commit the changes.

Git Log

Display the commits of the repository.

- **git log**
  show logs
- **git log –-oneline**
  show only commit message & ref
- **git log –-stat**
  shows the files involved in the commit
- **git log –-patch**
  shows the +/- lines
- **git log –graph**
  show all commits represented by an ASCII graph

Reset

Reset the current HEAD to the specified state. This allows you to undo merges, pulls, commits, adds, and more. It’s a great command but also dangerous if you don’t know what you are doing.

- **git reset**
  Does not touch the index file or the working tree at all (but resets the head to <commit>, just like all modes do). This leaves all your changed files "Changes to be committed", as git status would put it.
- **git reset –hard**
  Resets the index and working tree. Any changes to tracked files in the working tree since <commit> are discarded.
- **git reset --hard 31f2bb1**
  Same but to a specific commit

Revert & Ammend

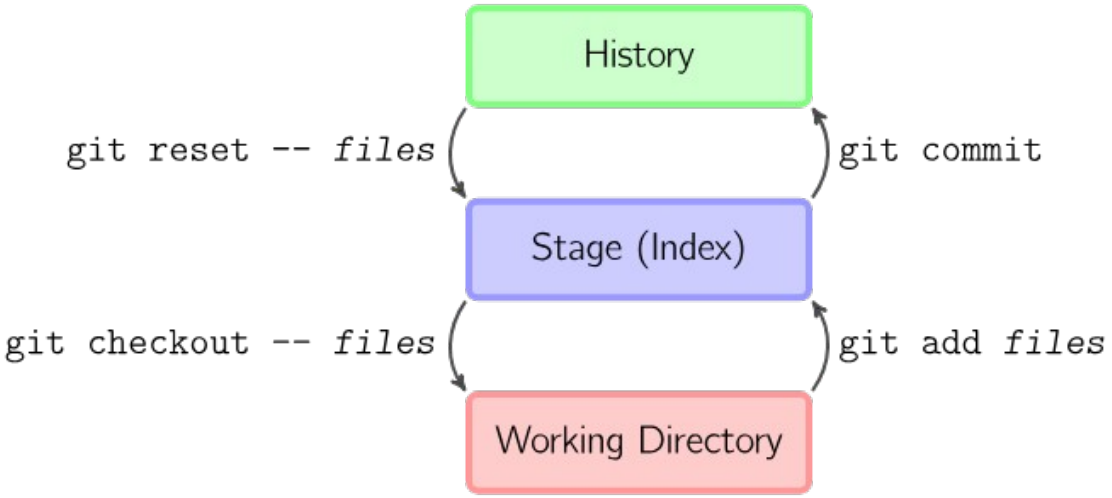
Revert can be used to undo a commit. It should not be confused with reset which restores the state of a project to a previous point.

Revert will add a new commit which is the inverse of the specified commit, thus reverting it.

- **git revert <commit>**
- **git commit --amend**
  modify the most recent commit
- **git commit --amend –no-edit**
  ammend the last commit without edit the message

  example:
  git add the_left_out_file
  git commit --amend --no-edit

Visual Git Reference



Resources

- Git Book
- Video Github Training
- Learn Git
- Git-guide by rogerdudler.github.io
- Visualizing-git
- Interactive Game