Git Survival Guide

Sonny LION (sonny.lion@ijclab.in2p3.fr)

Michel JOUVIN (michel.jouvin@ijclab.in2p3.fr)

Version-control software

Manual versioning is quickly a nightmare

- Generally, you start with a file: `myscript.py` and you end up with a mess: `myscript_new.py`, `myscript_final.py`, `myscript_final_final.py`, ...
- No traceability, comparison between versions is difficult, collaboration is a nightmare (sharing files by email, USB keys, ...)

The rise of version-control software

- 1970+: Local projects (e.g. GNU Revision Control System)
- 1980+: client-server architecture (CVS, SVN, ...)
- 2000+: atomic & decentralised (BitKeeper, Git, Hg, ...)

What to version?

Source code, configuration files, documentation or any small text files.

Versionning objectives:

- track changes and incrementally update the files
- compare and merge changes between versions

Avoid if possible: images (except to illustrate), binaries, data, or any large files:

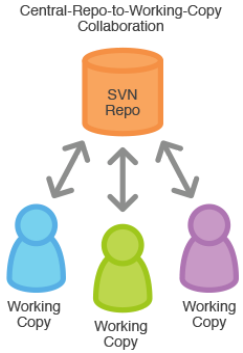
- Binary files are not easily readable nor comparable
- Large files can slow down the versioning system and make it difficult to use/manage
- Use dedicated tools for these files (e.g. LFS for Git)

NEVER VERSION passwords, API keys, sensitive information !

Strategies

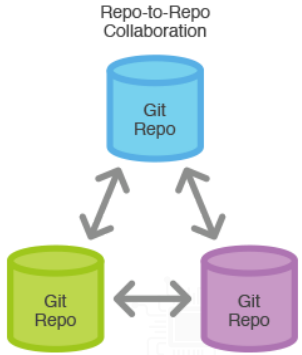
Centralized (e.g. SVN)

- One "main" server/repository with the "truth"
- A user has a copy of only one revision
- All changes are pushed to the "main" server
- Easy to manage but single point of failure and network dependency



Decentralized (e.g. Git)

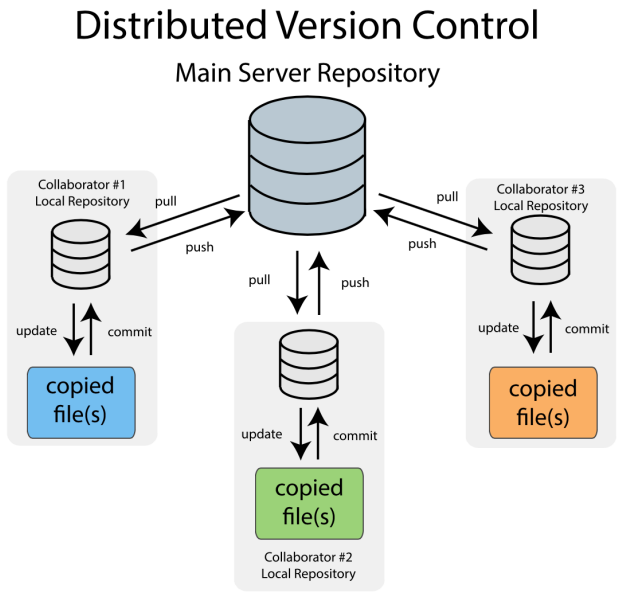
- Each user has a full copy of the repository
- Changes are synchronized between users
- No single point of failure, no network dependency
- More complex to understand / manage / synchronize



Git + Forge (e.g. Gitlab, Github, etc.) = ❤️

Mix between centralized and decentralized:

- One repository is considered as the "main" repository (e.g. Github, Gitlab, ...)
- Each user has a full copy of the repository and can synchronize with the "main" repository or other users' repositories
- No single point of failure, each user can work offline, simpler to manage than a fully decentralized system



Focus on Git

Before starting: configuration

Before using Git, you need to configure your identity (name and email) so that your commits can be identified. You can do this by running the following commands in your terminal:

```
git config --global user.name "John Doe"
git config --global user.email "john.doe@example.com"
```

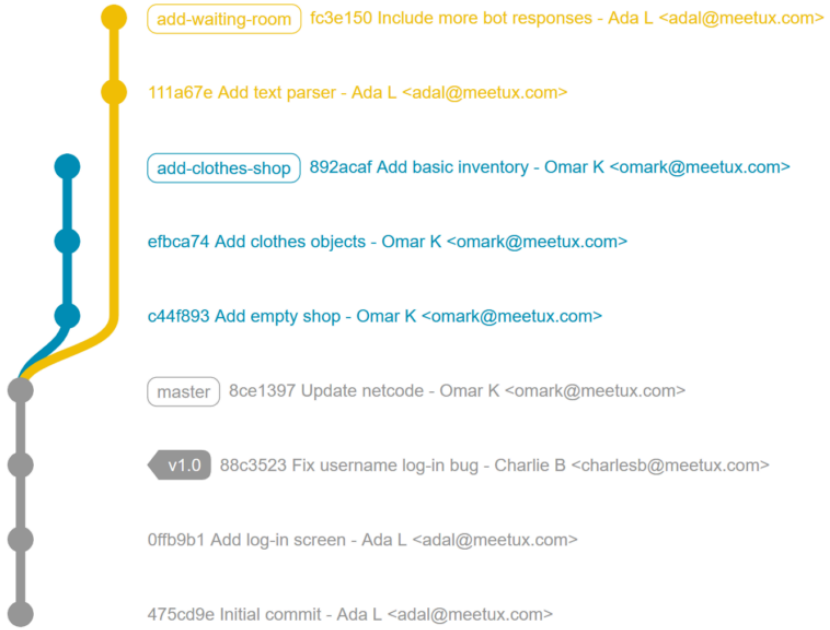
To check your configuration settings, you can use the git config command:

```
git config --list --show-origin --show-scope
```

What is a commit ?

Commits are the core building block units of a Git project timeline :

- Each commit represents a snapshot of the repository at a specific point in time
- Each commit has a unique identifier (hash), a commit message, and a pointer to its parent commit(s)
- Commits are organized in a directed acyclic graph (DAG) structure, forming a history of changes in the repository



How to create a commit ?

To know how to create a commit with Git, you need to understand how Git organizes the files and directories in your project. Git uses three main areas to manage changes and create commits:

1. Working directory
2. Staging area (index)
3. Local Repository

The working directory

(Basically) synonyms :

Working directory = Workspace = Project folder

The working directory (workspace) is essentially your project folder:

```
amazing_project/
    README.md
    src/
```

Git repository

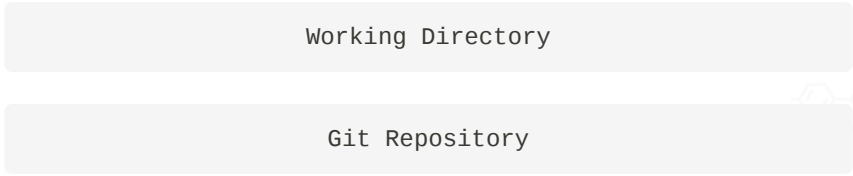
The repository is essentially the .git hidden folder inside the working directory (workspace): **never delete it.**

To initialize a new repository, you can simply run `git init` in your project folder.

Inside your computer

```
amazing_project/
    .git/   <--- Hidden folder containing the Git repositc
    README.md
    src/
```

Git areas



Adding a file

When you run `git add <file>`, you're essentially telling Git that you want to include the changes made to `<file>` in the next commit:

- Tracks changes
- Updates the staging area (index)
- Prepares for the next commit

Before git add README.md:

```
README.md
.git/
├── branches/
├── ...
├── objects/
└── refs/
```

After git add README.md:

```
README.md <--- The README file is still here
.git/
├── branches/
├── ...
├── objects/
├── refs/
└── index  <--- Staging area containing info about README.
```

Commiting a file

When you run git commit, you're essentially telling Git that you want to create a new commit with the changes currently in the staging area (index):

```
git commit -m "Add README.md"
```

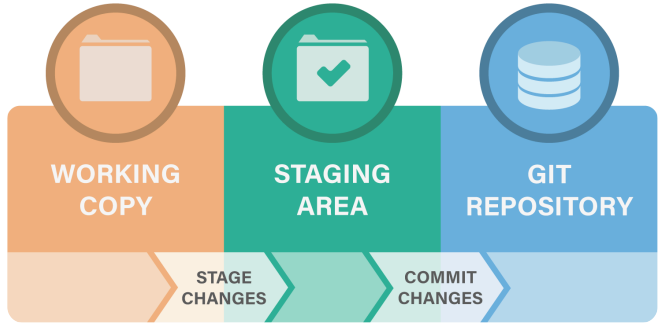
if `-m` is not specifed, your preferred editor will be open so that you can edit the commit message and the commit will happen once the editor is closed. If you want to abort the commit, you need to **save an empty message**.

It is possible to amend (modify) the last commit instead of creating a new one by adding the option `--amend`. If doing it, **be sure that your branch has not been already pushed to a remote directory** or you will turn everybody else life into a nightmare...

```
git commit --amend -m "Add enhanced README.md"
```

Summary: how to create a commit

1. Making changes
2. Staging changes
3. Commiting Changes



Add and commit modified files in a single command (only if the file is already tracked by Git)

```
git commit -am "Add README.md"
```

Commit Message Guidelines

```
Short (72 chars or less) summary

More detailed explanatory text. Wrap it to 72 characters. The blank
line separating the summary from the body is critical (unless you omit
the body entirely).

Write your commit message in the imperative: "Fix bug" and not "Fixed
bug" or "Fixes bug." This convention matches up with commit messages
generated by commands like git merge and git revert.

Further paragraphs come after blank lines.

- Bullet points are okay, too.
- Typically a hyphen or asterisk is used for the bullet, followed by a
  single space. Use a hanging indent.
```

Use multiple `-m` flags to add multiple paragraphs to your commit message

```
git commit -m "Short summary" -m "Detailed explanation"
```

Display the working directory status

It is important to regularly check the status of the working directory:

```
git status
```

- Identify the current branch
- Get a list of the new/modified/deleted files
- Know which modifications will be part of the next commit
- Get the main commands that make sense in the current working directory state

Display the commit history

It is often useful to display the commit history of the current branch:

```
git log
```

It is also possible to display the history from any commit in the repository

```
git log commit_id
```

`commit_id` can be a branch name. Many options available to customise the visualisation.

This command displays the history from the initial commit to the commit specified, in reverse order by default. Enter `q` to quit `git log` when paging mode is activated.

`git log` accepts many options. `--name-status` allows to display the files modified by each commit. It is also possible to add a parameter representing a file or directory name to list only the commits modifying this file or directory. It is recommended to add `--` before the file names to avoid any ambiguity with other parameters.

```
git log -- README.md
```

Display content of changes

It is sometimes useful to check the changes introduced by commits, staged changes or local modifications. The command `git diff` accept various options depending on what you want to check. The most common variants are:

- Differences introduced by modifications in the working copy not yet staged for commit:

```
git diff [file_or_dir_name]
```

- Differences introduced by modifications staged for commit (in the index):

```
git diff --cached [file_or_dir_name]
```

- Differences between 2 commits (if `last_commit_id` is omitted, `HEAD` is used):

```
git diff first_commit_id..last_commit_id [-- file_or_dir_name]
```

Stashing changes

If you have changes in your working directory that you don't want to commit yet, you can use the `git stash` command to temporarily store those changes. This is useful when you need to switch branches or work on a different task without committing your changes.

```
git stash
```

To apply the stashed changes back to your working directory:

```
git stash apply
```

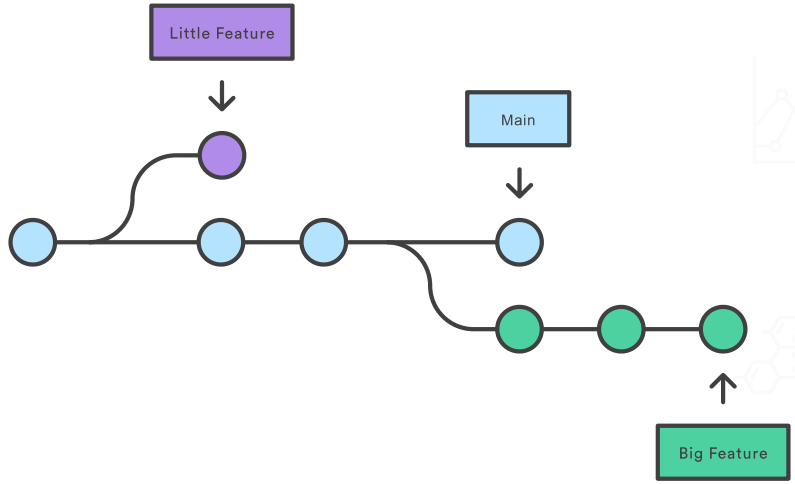
Popping the stash element (apply it and remove it from the stash list) with `git stash pop` works the same way as `apply`. Other useful stash-related commands:

```
# name your stash element for easier retrieval
git stash push -m "my_stash_name"
# list all stash elements
git stash list
# apply the stash element "my_stash_name" to the working directory
git stash apply stash^{/my_stash_name}
```

How to structure your workflow: Branches

Concept of branches

Each branch represents an **independent line of development**, allowing different **features** or **bug fixes** to be developed concurrently **without interfering** with each other.



Implementation in Git

A branch is essentially a lightweight movable pointer that points to a specific commit within the repository's commit history.

How to create a branch

List all branches

```
# local branches
git branch
# remote branches
git branch -r
```

Create a new branch

```
git branch <branch_name>
```

Switch to the branch

```
git checkout <branch_name>
```

Create and switch to the branch in one command

```
git switch -c <branch_name>
```

or

```
git checkout -b <branch_name>
```

Renaming a branch

To rename a branch, you can use the `-m` flag with the `git branch` command. This flag will rename the specified branch to the new name provided.

```
git branch -m <old_branch_name> <new_branch_name>
```

If you are currently on the branch you want to rename, you can omit the old branch name and just provide the new name.

```
git branch -m <new_branch_name>
```

Deleting a branch

To delete a branch, you can use the `-d` flag with the `git branch` command: this flag will delete the specified branch if it has already been merged.

```
git branch -d <merged_branch_name>
```

If the branch has not been merged, you can use the `-D` flag to force the deletion of the branch:

```
git branch -D <unmerged_branch_name>
```

Note: You cannot delete the branch you are currently on. You must switch to a different branch before deleting the branch you want to remove.

Incorporating changes from another branch

A typical development workflow implies several parallel developments made by one or more person that have to be combined to produce the final "product" (application, article, documentation...). Depending on what you want to achieve, there are two way of combining these parallel developments, typically done in different branches (and sometimes different repositories, see later):

- Incorporate changes made in another branch on top of your current branch: it is called a *merge* operation and is done with the `git merge` command. This is typically done inside your repository to incorporate into your main branch some features developped in another branch.
- Rebuild your development branch on another branch: it is called a *rebase* operation and is done with the `git rebase` command. This is typically done when working in a collaborative environment like GitHub or GitLab to incorporate changes made by others during your development to ensure that your changes can be applied on top of them and rebuild your commits to take into account the changes from the other branch.

Merging the content of another branch

Merging a branch is the process of combining the changes from this branch into another one, typically the current one. This incorporates new features or bug fixes developed on a separate branch into the another one, generally the main branch of the project.

`git merge` command requires the name of the source branch that you want to merge into the current one:

```
git merge <source_branch_name>
```

If there are no conflicts between the branches, Git will automatically merge the changes and create a new commit with the combined history of both branches.

Resolving conflicts: for more complex merges, you may encounter conflicts that need to be resolved manually. Git will indicate where the conflicts occur in the affected files: you will need to edit the files to resolve the conflicts and `git add` the changes before completing the merge with `git commit`, or abort the merge with:

```
git merge --abort
```

Rebasing the branch content

Instead of merging another branch into the current branch, it is possible to rebase its content on another branch, i.e. to rebuild the commits since the common parent of the 2 branches so that they apply the same modifications to the new source branch. The rebase process involves:

1. Rewinding the current branch up to the common parent, i.e. temporarily removing all the other commits
2. Adding all the commits from the source branch on top of the common parent
3. Rebuild all the commits temporarily removed on the new tip of the branch

`git rebase` requires the name of the source branch to use to rebuild the current branch:

```
git rebase <source_branch_name>
```

As with a merge operation, conflicts can occur and must be fixed the same way. One they are resolved, you must finish the rebase operation or abort it with :

```
git rebase --continue/--abort
```

What is the HEAD ? (Or where am I ?)

`HEAD` is a reference to the last commit (tip) of the current branch you have checked out. It can be used as a synonym of the current branch name in Git commands.

You can use the HEAD to move easily to the parent commit (`HEAD^`), the grandparent commit (`HEAD^^`), or even further back in the commit history (`HEAD~n` where `n` is the number of commits you want to move back).

```
git checkout HEAD^
```

Note that when moving to a commit other than the last commit of a branch, your working directory is in a state called `detached HEAD` . This state is not an error but it means that every modification from this commit will not be part of a branch, making more difficult (but not impossible) to move to it.

Reverting changes

If you want to revert a commit, you can use the `git revert` command. This command creates a new commit that undoes the changes made in the specified commits.

```
# A single commit
git revert <commit_hash>
# or a commit range
git revert <start_commit_hash>..<end_commit_hash>
```

If you want to remove the last commit from the commit history, you can use the `git reset` command and the `HEAD` pointer. There are several variants depending whether you want to remove the commit and forget about the corresponding changes in your working copy (`--hard`) or keep them in the index and in your working copy (`--soft`) or only in the working directory (`--mixed` , default)

Keep the changes in the staging area and working directory

```
git reset --soft HEAD^
```

Keep the changes only in the working directory (default)

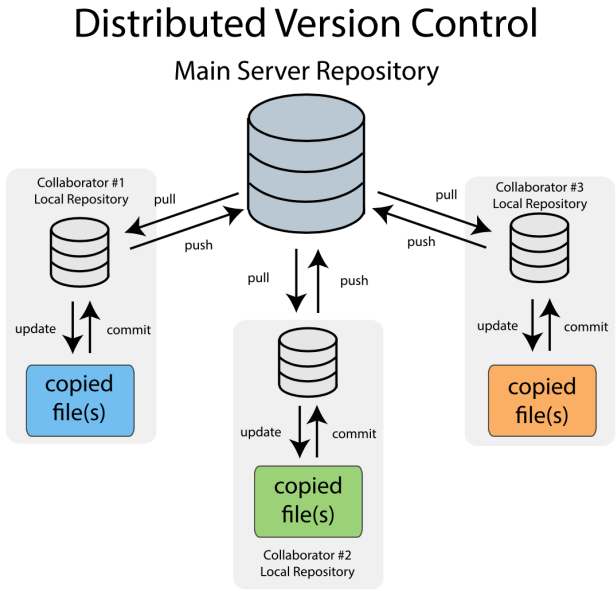
```
git reset [--mixed] HEAD^
```

Discard the changes

```
git reset --hard HEAD^
```

Remote repositories

Git is generally paired with a forge (e.g. GitHub, GitLab, Bitbucket) to share your code with others. Git offers specific commands to push and fetch changes to/from the remote repository hosted at the forge.



Cloning a repository

To clone a repository, you can use the `git clone` command followed by the URL of the repository you want to clone.

```
git clone <repository_url>
# or with a specific name
git clone <repository_url> <directory_name>
```

To clone a repository using SSH, you have to set up an SSH key pair and add the public key to your Git hosting service (e.g., GitHub, GitLab).

Adding a remote repository to an existing repository

To add a remote repository to an existing repository, you can use the `git remote add` command followed by the name of the remote and the URL of the repository.

```
git remote add <remote_name> <repository_url>
```

Usually, the remote name is set to origin by default when you clone a repository. You can use the `git remote -v` command to view the remote repositories associated with your local repository.

```
git remote -v
```

Modifying a remote address

To modify the URL of a remote repository, you can use the `git remote set-url` command followed by the name of the remote and the new URL.

```
git remote set-url <remote_name> <new_repository_url>
```

Fetching changes from a remote repository

Fetching changes from a remote repository is a 2-step process:

1. Update the copy of the remote repository in the local one using `git fetch` command:

```
# Fetch one specific remote
git fetch <remote_name>
# or to fetch all remotes
git fetch --all
```

2. Optionally, incorporate the changes in a remote branch into the current branch: it is typically done with `git rebase` as you generally want to redo your changes on the last version of a remote branch (.e.g. main branch of the upstream repository):

```
git rebase <remote_name>/<branch_name>
```

Dans certains cas particuliers, on peut être amené à faire un `git merge` plutôt qu'un `git rebase` mais dans ce cas il n'est plus possible de pousser la branche locale vers la branche remote.

Pushing changes to a remote repository

To push changes made in a local branch to a remote repository, you must use the `git push` command. It requires 2 parameters: the name of the remote repository and the local branch you want to push. By default, it will create or update a branch with the same name on the remote. `HEAD` can be used to refer to the current local branch:

```
git push <remote_name> HEAD
```

The push operation will fail if the history of the 2 branches has diverged.

If you are pushing changes to a remote branch for the first time, you can set the upstream branch (the remote branch linked with your local branch) using the `--set-upstream` flag. Once the upstream branch is defined, you can use `git push` without any parameter.

```
git push --set-upstream <remote_name> HEAD
```

To push changes to a remote branch with a name different from the local one:

```
git push <remote_name> <local_branch_name>:<remote_branch_name>
```

Pulling changes: git pull

In Git, pulling changes means fetching the content of a remote branch and incorporating the remote changes into the current local branch. The command implementing this is `git pull`:

```
# equivalent to git fetch <remote_name> followed by git merge <remote_name>/<branch_name>
git pull <remote_name> <remote_branch_name>
```

Despite this command is handy, its use is discouraged because by default it does a merge operation to incorporate the changes into the local branch but, as explained earlier, incorporating the changes of a remote repository is generally done with a rebase operation instead of a merge. To do this with `git pull`, you need to add the option `--rebase` but it tends to add complexity at the end...

Before pulling changes, it is a good practice to check the remote repository/branch associated with a local branch with the `git branch` command:

```
git branch -vvv
```

Ignoring files

Sometimes you may have files in your working directory that you don't want to include in your Git repository. This could be temporary files, build artifacts, or sensitive information that should not be shared (e.g., passwords, API keys).

A `.gitignore` can be placed in the root directory or any subdirectory of your Git repository to specify which files and directories should be ignored by Git. The `.gitignore` file uses glob patterns to match files and directories that should be ignored.

Example of a .gitignore file:

```
# Ignore build artifacts
build/
# Ignore temporary files
*.log
```

Create aliases

To save time and avoid typing long commands, you can create aliases for Git commands using the `git config` command.

```
git config --global alias.<alias_name> <command>
```

For example, to create an alias for the log command that displays a custom log format:

```
git config --global alias.ls "log --pretty=format:'%C(yellow)%h%Cred%d\\ %Creset%s%Cblue\\ [%cn]' --decorate"
```

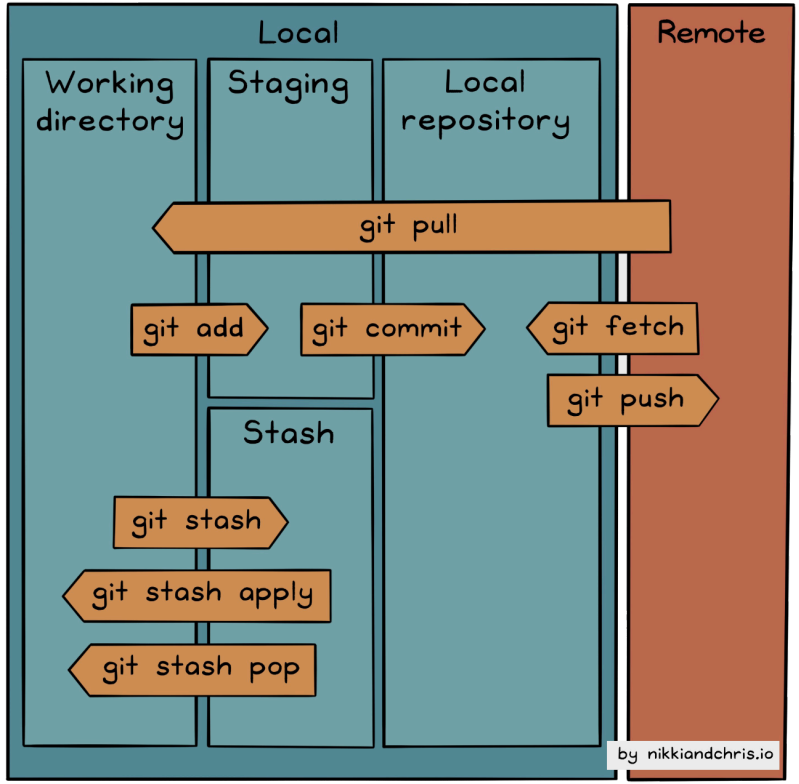
Or undo the last commit:

```
git config --global alias.undo "reset HEAD~1 --mixed"
```

Draw a graph of the commit history:

```
git config --global alias.graph "log --graph --abbrev-commit --decorate --format=format:'%C(bold blue)%h%C(reset) -
%C(bold green)(%ar)%C(reset) %C(white)%s%C(reset) %C(dim white)- %an%C(reset)%C(auto)%d%C(reset)' --all"
```

One image to rule them all



Git Cheat Sheet

Setup

Set the name and email that will be attached to your commits and tags

```
$ git config --global user.name "Danny Adams"
$ git config --global user.email "my-email@gmail.com"
```

Start a Project

Create a local repo (omit <directory> to initialise the current directory as a git repo

```
$ git init <directory>
```

Download a remote repo

```
$ git clone <url>
```

Make a Change

Add a file to staging

```
$ git add <file>
```

Stage all files

```
$ git add .
```

Commit all staged files to git

```
$ git commit -m "commit message"
```

Add all changes made to tracked files & commit

```
$ git commit -am "commit message"
```

Basic Concepts

main: default development branch
origin: default upstream repo
HEAD: current branch
HEAD^: parent of HEAD
HEAD~4: great-great grandparent of HEAD

By @DoableDanny

Branches

List all local branches. Add -r flag to show all remote branches. -a flag for all branches.

```
$ git branch
```

Create a new branch

```
$ git branch <new-branch>
```

Switch to a branch & update the working directory

```
$ git checkout <branch>
```

Create a new branch and switch to it

```
$ git checkout -b <new-branch>
```

Delete a merged branch

```
$ git branch -d <branch>
```

Delete a branch, whether merged or not

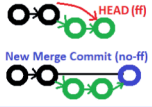
```
$ git branch -D <branch>
```

Add a tag to current commit (often used for new version releases)

```
$ git tag <tag-name>
```

Merging

Merge branch a into branch b. Add --no-ff option for no-fast-forward merge



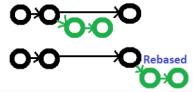

```
$ git checkout b
$ git merge a
```

Merge & squash all commits into one new commit

```
$ git merge --squash a
```

Rebasing

Rebase feature branch onto main (to incorporate new changes made to main). Prevents unnecessary merge commits into feature, keeping history clean




```
$ git checkout feature
$ git rebase main
```

Interatively clean up a branches commits before rebasing onto main

```
$ git rebase -i main
```

Interatively rebase the last 3 commits on current branch

```
$ git rebase -i Head~3
```

Undoing Things

Move (&/or rename) a file & stage move

```
$ git mv <existing_path> <new_path>
```

Remove a file from working directory & staging area, then stage the removal

```
$ git rm <file>
```

Remove from staging area only

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

View a previous commit (READ only)

```
$ git checkout <commit_ID>
```

Create a new commit, reverting the changes from a specified commit

```
$ git revert <commit_ID>
```

Go back to a previous commit & delete all commits ahead of it (revert is safer). Add --hard flag to also delete workspace changes (BE VERY CAREFUL)

```
$ git reset <commit_ID>
```

Review your Repo

List new or modified files not yet committed

```
$ git status
```

List commit history, with respective IDs

```
$ git log --oneline
```

Show changes to unstaged files. For changes to staged files, add --cached option

```
$ git diff
```

Show changes between two commits

```
$ git diff commit1_ID commit2_ID
```

Stashing

Store modified & staged changes. To include untracked files, add -u flag. For untracked & ignored files, add -a flag.

```
$ git stash
```

As above, but add a comment.

```
$ git stash save "comment"
```

Partial stash. Stash just a single file, a collection of files, or individual changes from within files

```
$ git stash -p
```

List all stashes

```
$ git stash list
```

Re-apply the stash without deleting it

```
$ git stash apply
```

Re-apply the stash at index 2, then delete it from the stash list. Omit stash@{n} to pop the most recent stash.

```
$ git stash pop stash@{2}
```

Show the diff summary of stash 1. Pass the -p flag to see the full diff.

```
$ git stash show stash@{1}
```

Delete stash at index 1. Omit stash@{n} to delete last stash made

```
$ git stash drop stash@{1}
```

Delete all stashes

```
$ git stash clear
```

Synchronizing

Add a remote repo

```
$ git remote add <alias> <url>
```

View all remote connections. Add -v flag to view urls.

```
$ git remote
```

Remove a connection

```
$ git remote remove <alias>
```

Rename a connection

```
$ git remote rename <old> <new>
```

Fetch all branches from remote repo (no merge)

```
$ git fetch <alias>
```

Fetch a specific branch

```
$ git fetch <alias> <branch>
```

Fetch the remote repo's copy of the current branch, then merge

```
$ git pull
```

Move (rebase) your local changes onto the top of new changes made to the remote repo (for clean, linear history)

```
$ git pull --rebase <alias>
```

Upload local content to remote repo

```
$ git push <alias>
```

Upload to a branch (can then pull request)

```
$ git push <alias> <branch>
```

Resources

- Git documentation
- Pro Git book
- Another Git Graphical Cheat Sheet
- Git Graph
- Git Exercises