git

Help

Find help on a command

```
$ git <command> --help
```

Basics

- **master** default development branch
- **origin** default upstream repository
- **.config** stored in homedir

Create Repositories

Clone an existing repository

```
$ git clone ssh://user@domain.com/repo.git
```

Create a new local repo

```
$ git init
```

Local Changes

Status of files in your working directory

```
$ git status
```

View all changes to tracked files

```
$ git diff
```

View changes to certain file

```
$ git diff <filename>
```

Add all changes to staging area

```
$ git add .
```

Add certain changes to staging area (hit ? for help)

```
$ git add -p <file>
```

Commit staged changes

```
$ git commit
```

Commit all changes (tracked files only)

```
$ git commit -a
```

Commit staged changes with message

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

Change last commit (never do this when already pushed)

```
$ git commit --amend
```

Branches

List existing branches

```
$ git branch -av
```

Create new branch based on current HEAD

```
$ git branch <new_branch>
```

Switch HEAD to branch

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

Create new branch based on HEAD and checkout

```
$ git checkout -b <new_branch>
```

Create new tracking branch based on remote branch

```
$ git checkout --track <remote/branch>
```

Delete local branch

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

Create new branch from commit and checkout

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

Tags

List existing tags

```
$ git tag -l
```

Mark current commit with tag

```
$ git tag <tagname>
```

Remotes

List configured remotes

```
$ git remote -v
```

Show info about remote

```
$ git remote show <remote>
```

Change remote url

```
$ git remote set-url <remote> <url>
```

Add new remote

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

Download changes from remote, but don't integrate into HEAD

```
$ git fetch <remote>
```

Download changes from remote and integrate into HEAD

```
$ git pull <remote> <branch>
```

Publish local changes to remote

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

Publish tags

```
$ git push --tags
```

Patches

Create path for other developers (extract commits which are in current branch but not in origin)

```
$ git format-patch origin
```

Apply a given patch

```
$ git am -3 <patchfile>
```

Hacks

Nicer log format

```
git config --global alias.lg "log --graph
--pretty=format:'%Cred%h%Creset -
%C(yellow)%d%Creset %s %Cgreen(%cr)
%C(bold blue)<%an>%Creset' --abbrev-commit
--date=relative"
```

Config ssh host

```
Host gitlab
  User git
  Hostname git.labict.be
  Port 22
  IdentityFile ~/.ssh/id_rsa
```

Commit Related Changes

A commit should be a wrapper for related changes. For example, fixing two different bugs should produce two separate commits. Small commits make it easier for other developers to understand the changes and roll them back if something went wrong. With tools like the staging area and the ability to stage only parts of a file, Git makes it easy to create very granular commits.

Commit Often

Committing often keeps your commits small and, again, helps you commit only related changes. Moreover, it allows you to share your code more frequently with others. That way it's easier for everyone to integrate changes regularly and avoid having merge conflicts. Having few large commits and sharing them rarely, in contrast, makes it hard to solve conflicts.

Dont Commit Half-done Work

You should only commit code when it‘s completed. This doesn‘t mean you have to complete a whole, large feature before committing. Quite the contrary: split the feature‘s implementation into logical chunks and remember to commit early and often. But don‘t commit just to have something in the repository before leaving the office at the end of the day. If you‘re tempted to commit just because you need a clean working copy (to check out a branch, pull in changes, etc.) consider using Git‘s «Stash» feature instead.

Test Code Before You Commit

Resist the temptation to commit something that you "think" is completed. Test it thoroughly to make sure it really is completed and has no side effects (as far as one can tell). While committing half-baked things in your local repository only requires you to forgive yourself, having your code tested is even more important when it comes to pushing/sharing your code with others.

Write Good Commit Messages

Begin your message with a short summary of your changes (up to 50 characters as a guideline). Separate it from the following body by including a blank line. The body of your message should provide detailed answers to the following questions: › What was the motivation for the change? › How does it differ from the previous implementation? Use the imperative, present tense («change», not «changed» or «changes») to be consistent with generated messages from commands like git merge.

Version Control is Not a Backup System

Having your files backed up on a remote server is a nice side effect of having a version control system. But you should not use your VCS like it was a backup system. When doing version control, you should pay attention to committing semantically (see related changes) - you shouldn't just cram in files.

Use Branches

Branching is one of Git‘s most powerful features - and this is not by accident: quick and easy branching was a central requirement from day one. Branches are the perfect tool to help you avoid mixing up different lines of development. You should use branches extensively in your development workflows: for new features, bug fixes, ideas…

Agree on a Workflow

Git lets you pick from a lot of different work-flows: long-running branches, topic branches, merge or rebase, git-flow… Which one you choose depends on a couple of factors: your project, your overall development and deployment workflows and (maybe most importantly) on your and your teammates‘ personal preferences. However you choose to work, just make sure to agree on a common workflow that everyone follows.