ZapDos7 / 00_git_intro.md

Last active 2 weeks ago • Report abuse

Code Revisions 47 Stars 9 Forks 7

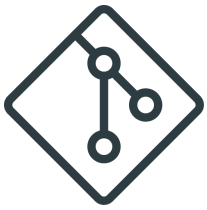
Git Notes

00_git_intro.md

Git

01_GIT_Intro.md

Git Notes



Introduction - Brief

Git = Control Version System.

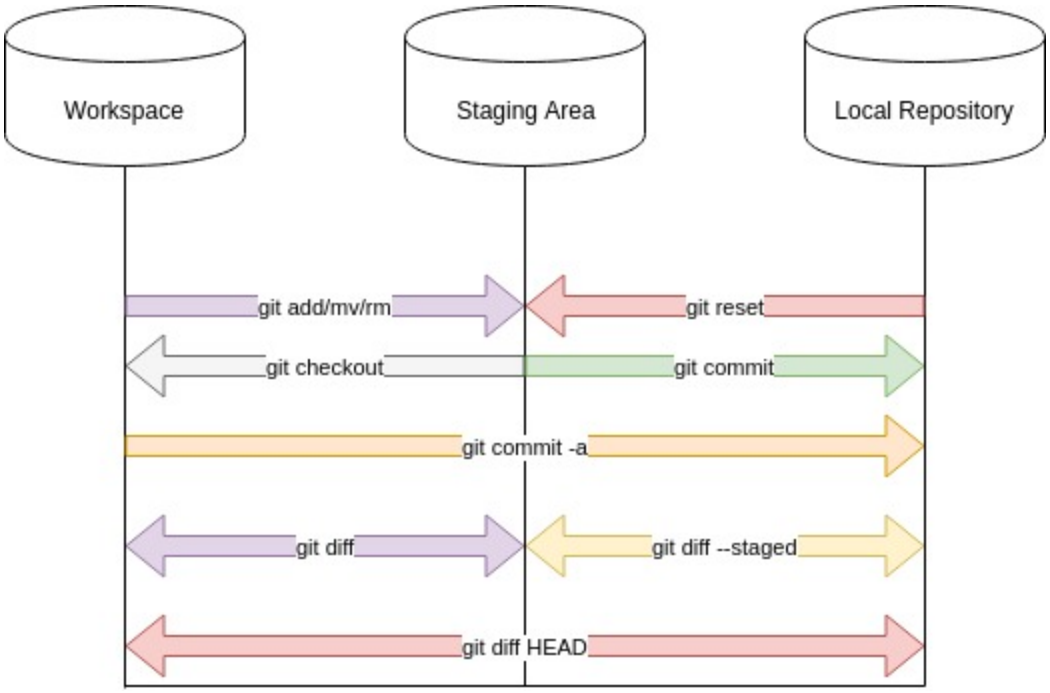
Files have 3 states:

- commited (aka stored in database)
- modified (aka changed, not commited)
- staged (aka a modified version of the file, marked for the next commit)

Each commit has its own unique ID, it is a hash of its containers and its metadata. It must have a perfix of at least 4 chars (unique in repo).

Staging Area

is the virtual space where the changes "go" when we `git add` . It allows the preparation for any `commit`



Special git files

- .gitignore = file in the main folder which contains a list of files that are ignored by git (so they're not affected by `git add` s and `git status` es). It can contain patterns (`*.pyc`) and paths to files.

02_Commands.md

Commands

Install:

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

Set up:

- `git config --global user.name "User Name"`
- `git config --global user.email "email@email.com"`

Create a New Project:

- `mkdir project`
- `cd project`
- `git init` *(initialise empty git repo in my folder (based on path) aka .git folder)*
- `ls -la` *(check my folder)*

Check "world status":

```
git status
```

Help for each command

```
git help <command>
```

Add files

- `git add .` = add all on current branch
- `git add -p <param=file>` = add part of file to staging area, ask for each change (if no param => all files) so we have more control and cleaner commits. *After any `git add` , we need a `git commit` , either a file or a pattern (e.g. `*.txt`)*

Delete a file

- `git rm <filename>` = deletes a file, updates git and then commit!
- `git rm --cached <filename>"` = delete a previously tracked file

Move a file

`git mv <old path> <new path>` *should be followed by:*

```
git rm <old path>
git add <new path>
```

Check difference

- `git diff` = displays what will be added if i `git add` , so what changed in the folder and hasn't been updated yet
- `git diff <filename>` = displays the alterations of a file (the modified and the commited versions of it)
- `git diff --staged` = displays what has already been added and thus what changed will be recorded

- `git diff HEAD` = displays changes since last commit

Display history

`git-log` = displays the history, the chronologival order of commits (based on their IDs), who did them, what was their description
`git show <id>` = displays what the commit did = `git log + git diff`

Make an alias

- `git config --global alias.<aliasname> "command(s)"`
  e.g.:
  `git config --global alias.lg "log --color --graph --pretty=format: '%(red%h%(green(%cr)%((bold blue)<%an>%(reset' --abbrev -commit"`
- `git config --list` - displays our aliases

Make archive

```
git archive --format=zip -o latest.zip HEAD
```

Revert to old commit

```
git log
git checkout <commit hex id>
```

Cancel not staged changes

`git checkout` = it copies staging area (usually last commit) to out working copy

Reset

`git reset` - remove all that exists in my staging area by copying them from the most recent commit (basically undoes `git add`)

Copy a commit to another branch

`git cherry-pick <commit>` = we copy a commit from a point of the graph, we put it on active branch (therefore creating a copy of the selected commit) - new ID, same changes and description!

Copy changes to new commit

`git revert <commit>` = inverted add/deletes etc. It cancels the commit that has already happened.

Tags

- `git tag -a <tag>` = adds tag to last commit of current branch
- `git tag -a <tag> <commit>` = add tag to selected commit
- `git tag` = shows all tags in repo
- `git tag -d <tag>` = deletes a tag

Publish tags

`git push <remote> <tag>` = publishes tag in remote `git fetch --tags <remote>` = brings all tags from remote

Serial list of changes

- `git reflog` = all the changes
- `git reflog <branch>` = changes on our branch
- `git reflog --date=relative` = displays changes relative to time

Prune stale references

- `git fetch -p`

Questions

How many `git add <filename>` do I need?

Times	What it does
1	Tracks `<filename>`
2	Makes `<filename>` staged
3	If modified again after staged, we need a thrid `git add` to stage it again

What does `git commit` do?

- commit a file = create a snapshot of the current world state (files, folders & their contents)
- contains an explanatory message
- automatically stores metadata (creator, date etc)
- has a unique (hex) id number
  e.g.: `git commit -m "Added README file`

Combinations

- `git commit -a = git add + git commit` (not desirable due to lack of control)
- `git pull = git fetch + git merge` (very useful)

03_Branches.md

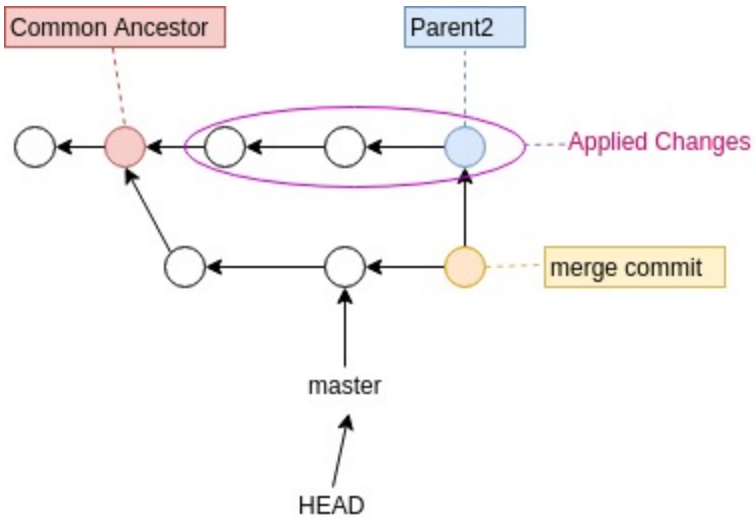
What is a branch?

It is a version of our code. Branches have a name and are pointing to a commit (there's a different history+past commits depending on our branch, but some commits may be common).
One branche per feature (the smaller the better) so changes happen to the branch, not the master workflow until the final merge. Afterwards, we merge and delete the branch.

Commands

- `git branch <name>` = creates the branch, it's an exact duplicate of our current/previous branch (they point to the same commit)
- `git branch` = returns my current branch
- `git checkout <name>` = changes current (`HEAD`), `<name>` points to `HEAD` now
- `git branch -d <name>` = deletes this branch (**NOT** the commits also)
- `git checkout -b <name>` = creates a new branch and makes this new branch as our current working one = `git branch <name> + git checkout <name>`
- `git merge <branch>` = merges 's history with my current branch + try to merge changes in files from both the branches => 2 parents in new commit. *(Afterwards we find the most recent parent of those two parents => commits of the new branch =*

commits of parent1 + commits of parent2 => updates master, master in new commit - see schema (1))



schema (1)

Note: If you make a branch on terminal and want it to show on GitHub, you need to `git push origin branchname` first! Note2: After being done with a branch, `git checkout <productionbranch>` , and then `git merge <tesbranch>` and then `git branch –d <testbranch>` (you can delete the testbranch from GitHub's UX)

Master Branch

- Our default branch after a `git init` command.
- (For most projects) it has a 'current' code
- Usually we create a new branch as a copy of master

References to parental nodes

Symbol	Meaning
~	1 commit behind
^	the first commited parent
~2	commit's grandpa (2 commits back based on `^` (if merged))
^2	second parent from merge

e.g. `192a812~2` = 2 commits before commit #192a812, or `HEAD^2`

Rewriting History

We can change our commits' sequence, description and changes, but: **you should not rewrite a history in commits that others may pull**

`git commit --amend` = changes most recent commit, add to it the staged stuff.

`git commit --amend --no-edit` = check here

Back Merging

When I work on a branch, it is possible that some changes might have happened on master => we need `git merge master` and resolve the conflicts. Or...

- `git rebase` = like `merge` but better, it happens between two branches and changes the base where a branch has been made, rewrites its history (clean). Followed by a clean `pull request`. Generally we merge only for final pull request on each branch.
- `git rebase -i` = dynamic: changes the sequence of commit applies changes, fixes multiple commits or can break a commit to many.
- `git reset <commit>` (usually `git reset HEAD`) = returns current branchto , cancels in between changes.

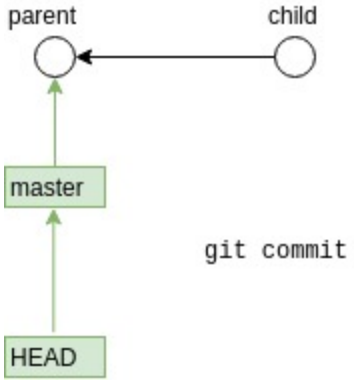
Not for published commits!!

- `git push --force-with-lease` or `git push --force` = if I change history, git denies pushing w/o it.

`04_Graphs.md`

Git = graph editing system

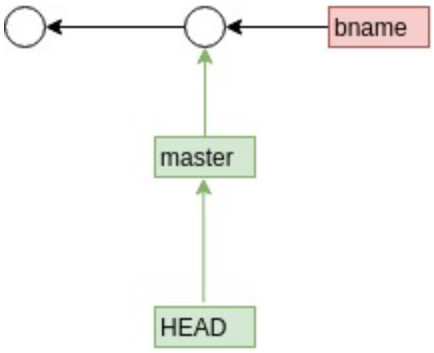
- Each commit is a node
- Each previous commit is a parent node
- Each branch points to a commit (this allows the creation of branches on the graph)
- HEAD is our current commit



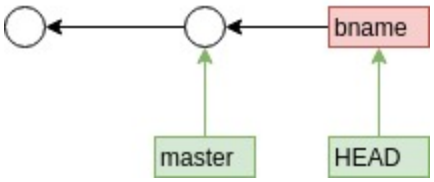
`git commit`

- creates a new commit object
- parent = commit where current branch (aka HEAD) points to
- transfers HEAD to child commit

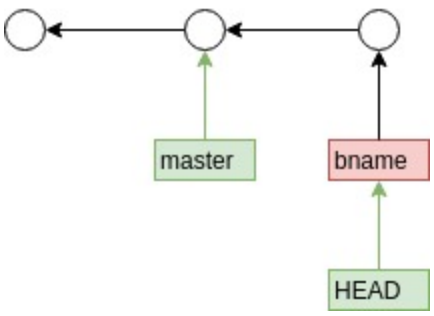
`git branch bname`

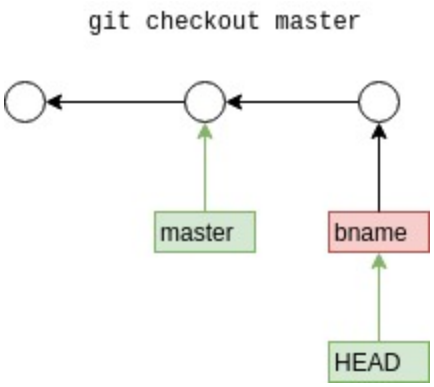


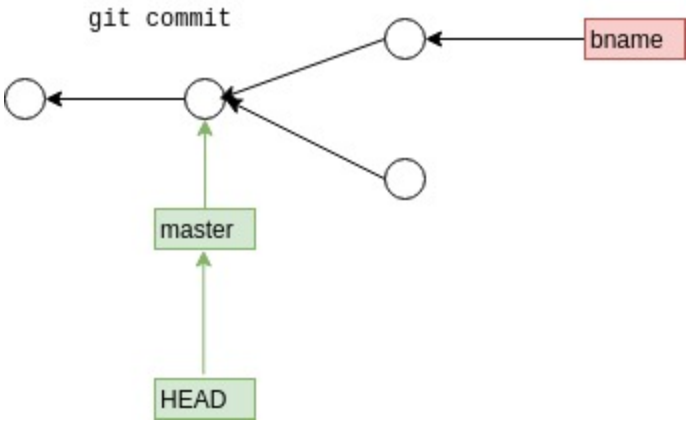
`git checkout bname`

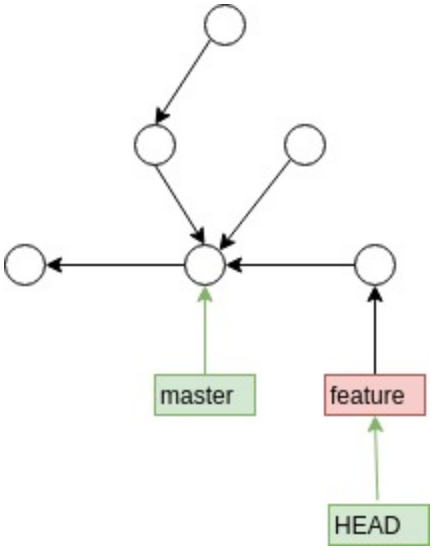


`git commit`




git checkout master
master
bname
HEAD


git commit
bname
master
HEAD


master
feature
HEAD

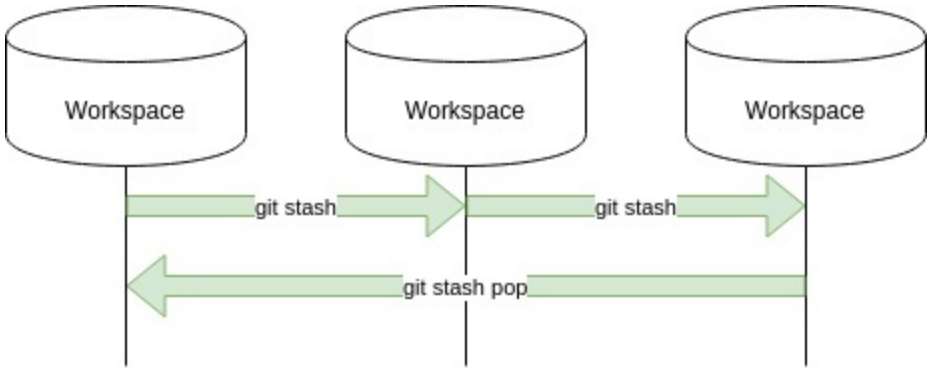
05_stash.md

Stash

`git stash` It is used for hotfixes in branch (meanwhile there are changes in working copy) without commits and checkouts.
Therefore it keeps changes in the working copy and the staging area and thus cleans them up.
We can checkout another branch without committing or losing changes!

Commands

- `git stash pop` = reapplies the saved working copy



Remotes

Remotes are copies used to update our code.

06_github.md

GitHub isn't git! GitHub is a site!

GitHub's features

- `fork` = copy a repo to your GitHub account
- `clone` = copy at your PC:

```
git clone
git @ github.com/UserName/git-repo.git
cd git-repo
```

- Your PC repo has the site repo as its remote
- Each repo copy has it's own commits/branches/history (some may be common)
- Each remote has its own URL & name

Commands

- `git remote` = shows repo's remotes
- `git remote add <name> <url>`
- `git remote rm <name>`

Origin

Origin = repo's copy to our GitHub (automatically made when cloning)
If you want to publish your code: `git push origin master` = sends master's local commits to remote's master named origin If you want to get your code: `git pull origin master` = brings commits at our local master from remote's master named origin

Outdated

If someone else pushed in the meantime when we're working locally, we can't overwrite their work.

```
push conflict: A-B-C-D
                    |
                    E
```

Can't merge E,D because split => `git pull` or `git pull -rebase` (the second keeps E only locally and D is our last state)

Hub

A set of extra commands that expand git specifically for GitHub

07_pullrequests.md

Intro to PRs

If you want to push to a repo which isn't yours (after forking, cloning, branching, changing and committing our work), we request a pull from the repo's owner (usually at master) so we publish our changes:

- `git push origin feature`
- create pull request (PR) at repo's owner's GitHub
- Finally, the repo's owner merges, after reviewing them (push anew etc).

Group Workflow

We have a central repo, the developers fork and do their work.
Latest code = master of central repo.
NO-ONE PUSHES TO ANY MASTER: for each change, each person pushes to their fork, PRs to master of control repo, the group reviews and then they merge.
Additionally, each member adds to the central repo an extra remote named 'upstream' (so now each member has 2 remotes: `origin` - their own and `upstream` - central).
From upstream master: everyone pulls to local master & pushes to origin master => Syncing master PRs (All: local & remote).
Each member is a contributor to the central repo therefore can merge.

08_bug_tracking_system.md

GitHub Issues

Each issue:

- has a unique ID
- is a bug or a task (open/resolved)
- has a description
- has commits

Issues can be opened even by users who are not the repo's owners. *Note: GitHub allows references to links*

When we locate a bug on a commit, we backtrack to a previous commit where there was no bug and try to find what caused it in the between commits (current=HEAD), or (see (*)).

Commands

- `git commit -m "Refactor code for issue #765`
  `git commit -m "Closes #765`
  or from PR description (similarly)

- `git checkout <commit>` = state detached head can't commit, we need to checkout again into a branch (and check code for some time)
- `git checkout <commit> <file>` = changes file's contains and brings them into commited state (no new commit/no stage)
- (*) `git bisect` :
  `git bisect start`
  `git bisect bad` (HEAD is buggy)
  `git bisect good <commit>` (is a known bug free commit).
  Basically it asks for each commit if the bug exists, we reply with `bisect good/bad`
  We stop it by using `git bisect reset` (last one). Check here too.
- `git blame <file>` = marks a file as the culprit

09_conflicts.md

Conflicts can't be automatically merged! Git has changed our file, merges what it can, but the rest is left on us:
Open an editor and look for lines like so: `<<<<<` , `=====` , `>>>>>`
These mark our conflict areas, where we keep all we need and discard what we don't.
Afterwards, we `git add` each file, `git commit` (therefore merge) and all is OK!

10_Workflows.md

Basic Git Workflow

- edit file (e.g. vim)
- `git add <filename>`
- `git status`
- `git commit -m "Explain what happened"`

`master` Workflow

- `git remote add origin <giturl>`
- `git checkout master`
- `git pull upstream master`
- `git push origin master`

- `git checkout -b feature`
- `vim && git add && git commit // git add .  , git commit -m "msg"`
- `git push origin feature`
- `hub pull-request -b upstream:master`

Workflow Overview

- `git init`
- `git commit`
- `git commit`
- `git checkout -b bugfix`
- `git commit`
- `git checkout master`
- `git checkout -b bname`
- `git commit`
- `git commit`
- `git checkout master`
- `git branch -D bname`
- `git chckout -b feature ...`

Branch Workflow

- `git checkout master`
- `git checkout -b parent2`
- `vim && git add && git commit` (as needed)
- `git checkout master`
- `git merge parent2`
- `git branch -d parent2`

Merge Workflow

- `git checkout master`
- `git pull upstream master`
- `git push origin master`
- `git checkout feature`
- `git merge master`

- `git push origin feature`
- `hub pull-request`

Rebase Workflow

- `git checkout master`
- `git pull upstream master`
- `git push origin master`
- `git checkout feature`
- `git rebase master`
- `git push origin feature`
- `hub pull-request`

Master - Develop Workflow

- `git checkout develop`
- `git pull upstream develop`
- `git push origin develop`
- `git checkout -b feature`
- changes...
- `git push origin feature`
- `hub pull-request -b upstream:develop`

Release Workflow

- `git checkout master`
- `git pull upstream develop`
- `git push origin develop`
- `git checkout master`
- `git pull upstream master`
- `git merge develop`
- `git tag -a v2.0`
- `git push origin master`
- `git push origin v2.0`
- `git push upstream master, v2.0`

11_git_notes_course.md

Git Notes from LinkedInLearning Course by Ronnie Sheer

Stashing

```
git diff                        # shows edits
git stash                       # stashes code
git stash pop                   # removed code from stash
git stash save "edit"           # save a stash named "edit"
git stash list                  # shows stashes w/ index
git stash apply 0               # applies stashed state with index 0
```

Staging changes

```
git status                      # shows current state
git commit -am                  # add files & commit at the same time
                                # less granular control
git add -p                      # accept selectively which
                                # changes you wish
                                # to commit interactively
                                # (type y/n & enter to accept
                                # or decline, respectively)
```

Mistakes & Fixes

Be careful where you push

```
git push                        # quicker
git push <remote> <branch>      # safer
```

Undo a commit - NOT pushed

```
git commit -am "something"
git reset --soft HEAD~1         # one commit back
git status                      # now back to staged
```

Challenge: Rename a commit & move it to another branch

```
git add .
git commit -m "something"

git log                          # shows changes
git reset --soft HEAD~1          # undo commit
git checkout -b new-branch       # create new branch
git commit -m "something again" # redo commit with new name
git checkout -d                  # go back to previous banch
```

Pre-commit and Pythong

Check it here

Amend, Revert

1. Amend: quickly modify a commit before pushing

```
git commit --amend -m "foo bar"
```

2. Revert: creates a new commit where we can undo our action

```
git revert <commit hash>    # optionally edit commit message
git log                     # shows the new state without
                            # losing the erroneous commit
```

11_git_pull_after_force_push.md

`git pull` after remote forced update

Source

```
# Erase local changes in feat1 and match the remote Git repo.
git switch feat1
git pull --rebase

# To preserve work in feat1
git switch feat1
```

```
git fetch
git reset origin/feat1 --soft
```

12_init.md

New GitHub repo from existing project from local directory

1. In the directory containing the project:

```
$ git init
$ git add .
$ git commit
```

2. On GitHub, create new repo
3.

```
$ git remote add origin git@github.com:username/new_repo
$ git push -u origin main
```

13_bits.md

Bits & Tips

1. *Your branch and origin have diverged and X and Y different commits each respectively*

```
git fetch origin branchName
git reset --hard origin/branchName
```

2. When merge develop to master: NOT SQUASH (keep history of what was merged for PROD)
3. Tags:

```
git fetch --tags (brings tags)
git tag (displays all)
```

```
git checkout tags/v1.0 -b v1.0 (creates local branch for this tag)
```

14_rerere.md

`$ git rerere` = reuse recorded resolution: git remembers how you've resolved a conflict so it resolves it on its own in the future.

`$ git config --global rerere.enabled` or `.gt/rr-cache`

`$ git rerere status` to verify

`$ git rerere diff`

20_gitignore_gitkeep.md

How to Use .gitignore and .gitkeep?

`.gitignore`

- In Git, the `.gitignore` file is used to specify which files or directories the change tracking process should ignore.
- Here you can find ready-made `.gitignore` templates for various technologies and languages.

`.gitkeep`

- Not a standard Git element
- Used to keep track of directories that we wish to keep track of, even if they are empty.

How to keep empty directories using `.gitignore`

```
# Ignore everything in this directory
*
# But do not ignore this .gitignore file
!.gitignore
```

Source

98_git_notes_general.md

Git

Global Settings

- Related Setup: https://gist.github.com/hofmannsven/6814278
- Related Pro Tips: https://ochronus.com/git-tips-from-the-trenches/
- Interactive Beginners Tutorial: http://try.github.io/
- Git Cheatsheet by GitHub: https://services.github.com/on-demand/downloads/github-git-cheat-sheet/

Reminder

Press `minus + shift + s` and `return` to chop/fold long lines!

Show folder content: `ls -la`

Notes

Do not put (external) dependencies in version control!

Setup

See where Git is located: `which git`

Get the version of Git: `git --version`

Create an alias (shortcut) for `git status` : `git config --global alias.st status`

Help: `git help`

General

Initialize Git: `git init`

Get everything ready to commit: `git add .`

Get custom file ready to commit: `git add index.html`

Commit changes: `git commit -m "Message"`

Commit changes with title and description: `git commit -m "Title" -m "Description..."`

Add and commit in one step: `git commit -am "Message"`

Remove files from Git: `git rm index.html`

Update all changes: `git add -u`

Remove file but do not track anymore: `git rm --cached index.html`

Move or rename files: `git mv index.html dir/index_new.html`

Undo modifications (restore files from latest commited version): `git checkout -- index.html`

Restore file from a custom commit (in current branch): `git checkout 6eb715d -- index.html`

Reset

Go back to commit: `git revert 073791e7dd71b90daa853b2c5acc2c925f02dbc6`

Soft reset (move HEAD only; neither staging nor working dir is changed): `git reset --soft 073791e7dd71b90daa853b2c5acc2c925f02dbc6`

Undo latest commit: `git reset --soft HEAD~`

Mixed reset (move HEAD and change staging to match repo; does not affect working dir): `git reset --mixed 073791e7dd71b90daa853b2c5acc2c925f02dbc6`

Hard reset (move HEAD and change staging dir and working dir to match repo): `git reset --hard 073791e7dd71b90daa853b2c5acc2c925f02dbc6`

Hard reset of a single file (`@` is short for `HEAD`): `git checkout @ -- index.html`

Update & Delete

Test-Delete untracked files: `git clean -n`

Delete untracked files (not staging): `git clean -f`

Unstage (undo adds): `git reset HEAD index.html`

Update most recent commit (also update the commit message): `git commit --amend -m "New Message"`

Branch

Show branches: `git branch`

Create branch: `git branch branchname`

Change to branch: `git checkout branchname`

Create and change to new branch: `git checkout -b branchname`

Rename branch: `git branch -m branchname new_branchname` or: `git branch --move branchname new_branchname`

Show all completely merged branches with current branch: `git branch --merged`

Delete merged branch (only possible if not HEAD): `git branch -d branchname` or: `git branch --delete branchname`

Delete not merged branch: `git branch -D branch_to_delete`

Merge

True merge (fast forward): `git merge branchname`

Merge to master (only if fast forward): `git merge --ff-only branchname`

Merge to master (force a new commit): `git merge --no-ff branchname`

Stop merge (in case of conflicts): `git merge --abort`

Stop merge (in case of conflicts): `git reset --merge` // prior to v1.7.4

Undo local merge that hasn't been pushed yet: `git reset --hard origin/master`

Merge only one specific commit: `git cherry-pick 073791e7`

Rebase: `git checkout branchname » git rebase master` or: `git merge master branchname` (The rebase moves all of the commits in `master` onto the tip of `branchname`.)

Cancel rebase: `git rebase --abort`

Squash multiple commits into one: `git rebase -i HEAD~3` (source)

Squash-merge a feature branch (as one commit): `git merge --squash branchname` (commit afterwards)

Stash

Put in stash: `git stash save "Message"`

Show stash: `git stash list`

Show stash stats: `git stash show stash@{0}`

Show stash changes: `git stash show -p stash@{0}`

Use custom stash item and drop it: `git stash pop stash@{0}`

Use custom stash item and do not drop it: `git stash apply stash@{0}`

Use custom stash item and index: `git stash apply --index`

Create branch from stash: `git stash branch new_branch`

Delete custom stash item: `git stash drop stash@{0}`

Delete complete stash: `git stash clear`

Gitignore & Gitkeep

About: https://help.github.com/articles/ignoring-files

Useful templates: https://github.com/github/gitignore

Add or edit gitignore: `nano .gitignore`

Track empty dir: `touch dir/.gitkeep`

Log

Show commits: `git log`

Show oneline-summary of commits: `git log --oneline`

Show oneline-summary of commits with full SHA-1: `git log --format=oneline`

Show oneline-summary of the last three commits: `git log --oneline -3`

Show only custom commits: `git log --author="Sven"  git log --grep="Message"  git log --until=2013-01-01  git log --since=2013-01-01`

Show only custom data of commit: `git log --format=short  git log --format=full  git log --format=fuller  git log --format=email  git log --format=raw`

Show changes: `git log -p`

Show every commit since special commit for custom file only: `git log 6eb715d.. index.html`

Show changes of every commit since special commit for custom file only: `git log -p 6eb715d.. index.html`

Show stats and summary of commits: `git log --stat --summary`

Show history of commits as graph: `git log --graph`

Show history of commits as graph-summary: `git log --oneline --graph --all --decorate`

Compare

Compare modified files: `git diff`

Compare modified files and highlight changes only: `git diff --color-words index.html`

Compare modified files within the staging area: `git diff --staged`

Compare branches: `git diff master..branchname`

Compare branches like above: `git diff --color-words master..branchname^`

Compare commits: `git diff 6eb715d  git diff 6eb715d..HEAD  git diff 6eb715d..537a09f`

Compare commits of file: `git diff 6eb715d index.html  git diff 6eb715d..537a09f index.html`

Compare without caring about spaces: `git diff -b 6eb715d..HEAD` or: `git diff --ignore-space-change 6eb715d..HEAD`

Compare without caring about all spaces: `git diff -w 6eb715d..HEAD` or: `git diff --ignore-all-space 6eb715d..HEAD`

Useful comparings: `git diff --stat --summary 6eb715d..HEAD`

Blame: `git blame -L10,+1 index.html`

Releases & Version Tags

Show all released versions: `git tag`

Show all released versions with comments: `git tag -l -n1`

Create release version: `git tag v1.0.0`

Create release version with comment: `git tag -a v1.0.0 -m 'Message'`

Checkout a specific release version: `git checkout v1.0.0`

Collaborate

Show remote: `git remote`

Show remote details: `git remote -v`

Add remote upstream from GitHub project: `git remote add upstream https://github.com/user/project.git`

Add remote upstream from existing empty project on server: `git remote add upstream ssh://root@123.123.123.123/path/to/repository/.git`

Fetch: `git fetch upstream`

Fetch a custom branch: `git fetch upstream branchname:local_branchname`

Merge fetched commits: `git merge upstream/master`

Remove origin: `git remote rm origin`

Show remote branches: `git branch -r`

Show all branches (remote and local): `git branch -a`

Create and checkout branch from a remote branch: `git checkout -b local_branchname upstream/remote_branchname`

Compare: `git diff origin/master..master`

Push (set default with `-u`): `git push -u origin master`

Push: `git push origin master`

Force-Push: `git push origin master --force

Pull: `git pull`

Pull specific branch: `git pull origin branchname`

Fetch a pull request on GitHub by its ID and create a new branch: `git fetch upstream pull/ID/head:new-pr-branch`

Clone to localhost: `git clone https://github.com/user/project.git` or: `git clone ssh://user@domain.com/~/dir/.git`

Clone to localhost folder: `git clone https://github.com/user/project.git ~/dir/folder`

Clone specific branch to localhost: `git clone -b branchname https://github.com/user/project.git`

Clone with token authentication (in CI environment): `git clone https://oauth2:<token>@gitlab.com/username/repo.git`

Delete remote branch (push nothing): `git push origin :branchname` or: `git push origin --delete branchname`

Archive

Create a zip-archive: `git archive --format zip --output filename.zip master`

Export/write custom log to a file: `git log --author=sven --all > log.txt`

Troubleshooting

Ignore files that have already been committed to a Git repository: http://stackoverflow.com/a/1139797/1815847

Security

Hide Git on the web via `.htaccess` : `RedirectMatch 404 /\.git` (more info here: http://stackoverflow.com/a/17916515/1815847)

Large File Storage

Website: https://git-lfs.github.com/

Install: `brew install git-lfs`

Track `*.psd` files: `git lfs track "*.psd"` (init, add, commit and push as written above)

99_sources.md

Sources

1. This Git Class
2. Lectures on Git from NKUA (Course name: Software Development for Algorithmic Problems)
3. Interactive LearnGitBranching
4. hofmannsven's gist
5. LinkedInLearning Course by Ronnie Sheer, exercise code
6. Optionally install ohmyzsh for ease of use of git via terminal
7. LinkedInLearning Course by Ray Villalobos, exercise code (*start with branch* `01_02b`)
8. Notes on rebasing
9. What can go wrong when rebasing?
10. Notes on stashing
11. Notes on cherry-picking

12. The difference between merging & rebasing
13. This LinkedInLearning course by Kevin Skoglund
14. GitLab HandBook
15. jgitver
16. This amazingly brief Git tutorial
17. Git For Professionals
18. Advanced Git Tools
19. The Definitive Guide to `git merge`
20. Notes for the GitHub Foundations Certification Course
21. GitLab academy - DevSecOps oriented

And just Googling issues :)