Git Cheat Sheet

- Legend: `<>` required, `[]` optional

Common Terminology

- commit ≙ object with annotated changes in relation other commits
- branch ≙ collection of commits
- stage ≙ files earmarked for the next commit
- HEAD ≙ reference to the top commit in the current branch
- remote = bookmark for a repository origin (a repository may have several remotes)

Create & Clone

- Clone an existing repository

```
git clone <URL> [folder]
```

Default protocoll is SSH, eg. »git@example.com:repo.git«. HTTPS would be »https://example.com/repo.git«, another local repo »/home/user/repo.git«

Creates a subfolder with the repo, if the folder name is not given, then the repo name is used (»foo.git« = »./foo« subfolder)

- Create a new local repository

```
git init <folder>
```

If a folder name is given, a subfolder is created, otherwise the current folder is used

- Send existing local repository to remote

```
git remote add origin <URL> && git push <remote> <branch>
```

- Special: Create an empty repository on a remote server

Connect with the remote server first

```
mkdir <repo>.git && cd <repo>.git && git init --bare
```

The remote repository has to be »bare« (does not contain a working filetree, but a special .git subdirectory only) in order to accept a push

Show changes

- Show working status - show current branch name and changed or new files

```
git status
```

Hint: Set a short alias for often used commands, like `git st` for `git status` → see »Configuration«

- Difference between HEAD and files not yet staged

```
git diff
```

Note: This ignores new files = files which were not added to the repository yet and therefore arent »tracked«

- Difference between HEAD and staged files

```
git diff --cached
```

- Difference between HEAD and all files (staged and not staged)

```
git diff HEAD
```

- Difference between branches, two commits, etc

```
git diff <foo> <bar>
```

»+« line does exist in »bar« but not in »foo«, »-« reverse

- Difference to another branch and show names of changed files only

```
git diff <branch> --name-status
```

- Show all commits of current branch which are not merged into another branch

```
git log <reference>.. --oneline
```

The reference may be a branch or a tag, note the two dots at the end

- Show branches in which one commit exists

```
git branch --contains <commit ID>
```

Show history

- Show all commits of current branch

```
git log
```

- Show all commits of current branch and names of each changed file

```
git whatchanged
```

- Show commits and each difference for a specific file

```
git log -p <file>
```

- Examination: Show who changed what and when in a file

```
git blame <file>
```

Left side shows the last commit ID for the content on the right side

- Show a single commit and its differences

```
git show <commit ID>
```

- Show all commits with a certain word in the commit message

```
git log --grep=<searchword>
```

Commit

- Stage all (even untracked) files

```
git add -A
```

- Stage a tracked and modified file

```
git add <file>
```

- Add hand-picked changes in a file to the next commit (≙ partial commit)

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

`y` *Yes, add this part to the next commit*
`n` *No, skip this part*
`d` *Don't add this and all remaining parts of the file*
`s` *Try to split the current part into smaller ones*
`e` *Manually edit the part*

- Stage all changes in tracked files and start a commit

```
git commit -a
```

- Commit all previously staged changes

```
git commit
  git commit -m "<message>"
```

Branches

- List local branches

```
git branch
```

`*` *marks the current branch*

- List remote branches

```
git branch -r
```

use `-a` to show local and remote branches at once

- Switch to a different branch

```
git checkout <branch>
  git checkout -t <remote>/<remote-branch>
```

`-t` *checkout a new branch based on remote branch and save their connection*

- Create a new branch based on HEAD

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

use `git checkout -b <branch>` to create a branch and switch right into it

- Create a new branch based on a remote branch

```
git branch --track <new-branch> <remote>/<remote-branch>
```

use `--no-track` to create a new branch based on a remote branch, but don't save a connection between both

- Connect a remote branch with a local branch

```
git branch --track <local-branch> <remote>/<remote-branch>
```

- Show merged branches

```
git branch -a --merged
```

`--no-merged` *will show branches not merged yet*

- Delete a local branch

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

`-d` *will only delete the branch if it is merged with its remote branch (if set),* `-D` *will force the deletion*

- Delete a remote branch

```
git push <remote> :<remote-branch>
```

Tags

Use tags to save a specific version (the commit relations up to this point) of a project. Merging older commits into the branch afterwards hence wont affect the tag.

- Show all tags

```
git tag -n
```

`-l` *will show tag names only,* `-n<num>` *will add a number of lines from the annotation (default is one)*

- Mark the current commit with a tag

```
git tag <tag-name> -m "<annotation>"
```

Hint: Use semantic version numbers as tags

Update

- Download all changes from , but don't merge to HEAD yet

```
git fetch <remote>
```

A manual merge is required now

- Download changes and directly merge to HEAD

```
git pull [<remote> <branch>]
```

If the connection between remote & local branch is saved, then `git pull` is sufficient

- List all currently configured remote repositories

```
git remote -v
```

- Show information about a remote, eg. which branches exist in this remote

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

- Remove stale remote branch trackings (outdated connections)

```
git remote prune <remote>
```

Remove connections to branches deleted on the remote by now - does not delete the local branch

- Add a new remote repository

```
git remote add <remote> <url>
```

Publish

- Push local branch or tag to remote

```
git push [<remote> <branch|tag>]
```

use »-u« to push the branch and automatically save the connection between local & remote

- Push all local branches to remote

```
git push --all <remote>
```

- Push all tags to remote

```
git push --tags <remote>
```

Merge

- Merge a branch into your current HEAD

```
git merge <branch>
```

- Manually solve conflicts and mark file as resolved

```
git add <resolved-file> && git commit -m 'Manual Merge'
```

- Use a tool to solve merge conflicts

```
git mergetool
```

will use tool set in »merge.tool«, use »-t « to start a custom tool

- Use a merge strategy

```
git merge -s recursive -X <ours|theirs|patience>
```

»recursive« is the default merge strategy when pulling or merging one branch, so this param may be redundant
»ours« merge commits but try to ignore all conflicting changes from the other branch
»theirs« merge commits but try to ignore conflicts introduced by the own branch
»patience« will cause GIT run rather time-consuming intelligent merge routines to avoid merge conflicts and errors in the first place

- Cancel merge

```
git merge --abort
```

Rebase

Use rebase with care! It will rewrite the history and therefore requires additional efforts when working with a team! Dont rebase unless every project member knows about the required workflow!

- Rewrite commits from HEAD until given commit

```
git rebase -i <commit ID>
```

Opens an editable rebase command list - reorder the commands to change commit order, remove a line to delete the commit, change the preceded keyword to change the command
`p|pick` *keep commit*
`r|reword` *use commit, but edit the commit message*
`e|edit` *use commit, but halt the rebase sequence to change the commit (use* `git commit --amend -a` *)*
`s|squash` *use commit, but meld into previous commit*

- Rebase your current HEAD onto

```
git rebase <branch>
```

Merges all commits of given branch and applies new commits of the local branch on top (creates new commit IDs for these)

- Abort a rebase

```
git rebase --abort
```

- Continue a rebase after resolving conflicts

```
git rebase --continue
```

Stash

Use stash to save all current changes to a clipboard and retrieve them later.

- Stash all changes away

```
git stash save [comment]
```

- Show all available stashes

```
git stash list
```

»stash@{0}« is the rather unreadable name of the stash state, where 0 is the latest

- Retrieve a state form the stash list

```
git stash apply <stash-name>
```

default is »stash@{0}«, use `git stash pop <stash-name>` *to apply changes and remove the state from stash list*

- Remove a state from the stash list

```
git stash drop <stash-name>
```

- Remove all the stashed states

```
git stash clear
```

Revert

Git is merciful and lets you undo allmost all changes with ease.

- Clear stage (≙ unadd files)

```
git reset HEAD --
```

- Discard all changes

```
git checkout -- [file]
```

- Change the last commit

```
git commit --amend -a
```

Replaces the last commit (new ID), so it should only be used if the modified branch was not pushed yet

- Special: Change author of the last commit

```
git commit --amend --author "John Doe <doe@example.com>"
```

- Remove the last commit but keep all files and changes

```
git reset HEAD~1
```

Removes the last commit from the local history

- Revert a commit (≙ apply inversion)

```
git revert <commit-id>
```

Inverts changes of the given commit, applies them to the working directory and starts a new commit

- Undo a local merge

```
git reset --hard <merge commit ID>
```

Use only if the branch wasn't pushed yet, otherwise rebase or revert

- Remove a file

```
git rm --cached
```

Removes the file from the git repository index but keeps it on the file system

Configuration

- Get configuration option

```
git config <section>.<key>
```

- Set configuration option

```
git config --local <section>.<key> <value>
```

»local« will write to ».git/config« in current repository, »global« to »~/.gitconfig» and »system« to your systems »/etc/gitconfig«

- Set username and e-mail

```
git config --local user.name "<username>" && git config --local user.email <e-mail>
```

- Ignore mode changes (chmod)

```
git config --local core.filemode false
```

- Set alias »st« for »status«

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

Commit Message Format

```
[BUGFIX] Short summary

Optional explanatory text. Separated by new line. Wrapped to 74 chars. Written in imperative present tense ("Fix bug", not "Fixed bug").

Help others to understand what you did (Motivation for the change? Difference to previous version?), but keep it simple.

Mandatory title prefix: [BUGFIX], [FEATURE] (also small additions) or [TASK] (none of the above, e.g. code cleanup). Additionall flags: [!!] (breaking change), [DB] (alter database definition), [CONF] (configuration change), [SECURITY] (fix a security issue).

Bug tracker refs added at bottom (see http://is.gd/commit_refs).

Resolve #42
Ref #4 #8 #15 #16
```

shortened, detailed example at http://is.gd/commitformat

Best practices

- Commit related changes
  - Each commit should adress one logical unit. Two different bugs should result into two commits.
- Commit early & often
  - Keep your commits small and comprehensible, split large features into logical chunks.
- Test code before committing
  - Make sure the code works, don't guess. Or let tools test your commit automatically. Revert faulty commits if necessary.
- Don't commit half-done work
  - Commit only complete, logical changes, not half-done chunks. »Stash« changes if applicable.
- Don't commit hot files
  - Don't commit configuration files (commit a config template instead), personal data, temporary files (GIT is no backup system) or things that can be regenerated form other commited things.
- Write good commit messages
  - Help others to understand what you did (Motivation for the change? Whats the difference to the previous version?)
  - Useless commit messages may be forwarded to whatthecommit.com
- Write in imperative present tense («change», not «changed» or «changes»)
  - A commit is a set of instructions for how to go from a previous state to a new state, so you should describe was the commit does and not what it did to your repository.
- Don't panic
  - GIT lets you undo, fix or remove a bunch of actions
- Don't change published history
  - GIT allows you to rewrite public history, but it is problematic for everyone and thus it is just not best practice to do so.
- Use branches
  - Branching is cheap. Use separate branches for each bugfix, feature & idea. Make branching a part of your local workflow.
- Merge regularly
  - Don't merge a huge feature into the master, instead merge the master regularly with your branch.
- Use conventions
  - As with every development process: use conventions. For naming of branches and tags, how to write commit messages, when to commit into what branch, etc.

Sources

- Git Cheat Sheet by Ying Guo
- Git Cheat Sheet by Git Tower
- http://gitready.com/
- http://git-scm.com/documentation
- http://wiki.typo3.org/CommitMessage_Form

About

- Supervisor: Dan Untenzu @pixelbrackets
- License: CC-BY-SA 3.0
- Download & Contribution: pixelbrackets.de/git-cheat-sheet