Version Control using `git`

Maximilian Nöthe

Astroparticle Physics, TU Dortmund

2nd Asterics-Obelics School 2 – 4th of June 2018

Overview

What is version control and why do we need it?

What is Version Control?

→ Version Control tracks changes of a (collection of) document(s)
→ This can basically be anything: software, legal documents, documentation, scientific paper
→ We will call a snapshot of such a collection a “revision”.
→ Revisions are the complete history of our projects

Why Use Version Control?

→ Allows us to go back to arbitrary revisions
→ Shows differences between revisions
→ Enables collaborative working
→ Acts as backup if used together with a remote server

Why Use Version Control?

Most Version Control Systems (VCS) make answering the following questions easy:

What?	What changed from revision a to revision b?
Who?	Who made a change? Who contributed?
Why?	VCS usually encourage or even force adding explanations to changes.
When?	In which revision was a bug introduced or fixed?

Why Use Version Control?

Most Version Control Systems (VCS) make answering the following questions easy:

What?	What changed from revision a to revision b?
Who?	Who made a change? Who contributed?
Why?	VCS usually encourage or even force adding explanations to changes.
When?	In which revision was a bug introduced or fixed?

Version Control is a basic requirement for reproducible science

→ Created by Linus Torvalds in 2005 for the Linux Kernel

→ Most widely used VCS in FOSS

→ Distributed, allows offline usage

→ Much better branching model than precursors like SVN, more later

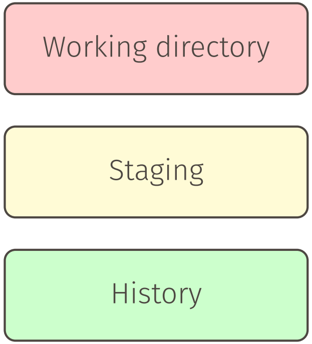
The Git Repository

Zentrales Konzept: Das Repository

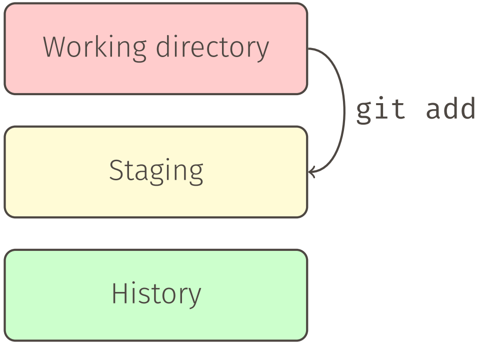
- → `git init` creates a git repository in the current working directory
- → All git data is stored in the `.git` directory.
- → Git has three different areas, changes can reside in:

Working directory	What actually is on disk in the current working directory.
Staging	Changes that are saved to go into the next commit.
History	The history of the project. All changes ever made. A Directed Acyclic Graph of commits.

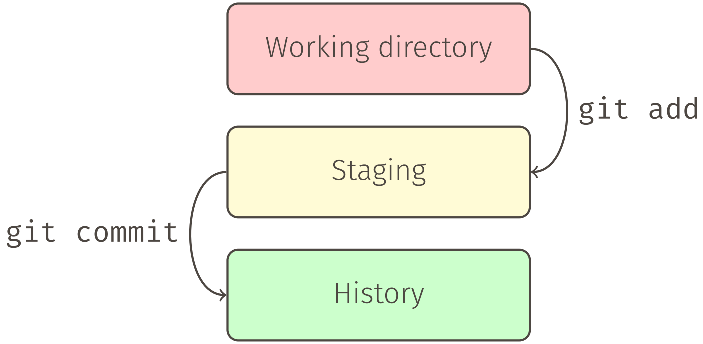
Central Concept: Repository



Central Concept: Repository

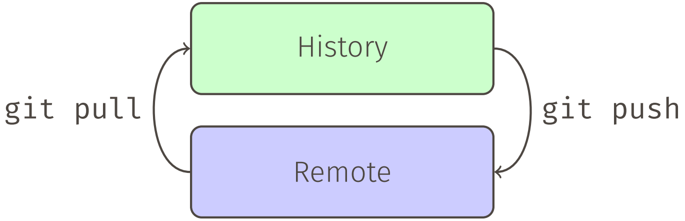


Central Concept: Repository



Remotes

Remotes are central places, e.g. servers, where repositories can be saved and which can be used to synchronize different clients.



The main remote is canonically named `origin`.

`git remote add <name> <url>`, e.g.
`git remote add origin https://github.com/maxnoe/myrepo`

History

$$\begin{tikzcd} a & b \arrow[l] & c \arrow[l] & d \arrow[l] & \texttt{master} \arrow[l] \end{tikzcd}$$

→ Commit: State/Content at a given time

  → Contains a commit message to describe the changes
  → Commits always point to their parent(s)
  → Commits are identified by a hash of the content, message, author(s), parent(s)

History

$$\begin{tikzcd}
\mathtt{a} & \mathtt{b} \arrow[l] & \mathtt{c} \arrow[l] & \mathtt{d} \arrow[l] & \mathtt{master} \arrow[l] \\
 & & & \mathtt{e} \arrow[ul] & \mathtt{foo} \arrow[l]
\end{tikzcd}$$

- → Commit: State/Content at a given time
  - → Contains a commit message to describe the changes
  - → Commits always point to their parent(s)
  - → Commits are identified by a hash of the content, message, author(s), parent(s)
- → Branch: A named pointer to a commit
  - → Development branches
  - → Main branch: `master`
  - → Moves to the next child if a commit is added

History

$$\begin{tikzcd}
a & b \arrow[l] & c \arrow[l] & d \arrow[l] & e \arrow[l] & f \arrow[l] & h \arrow[l] \arrow[lld] & i \arrow[l] & \text{master} \arrow[l] \\
 & & & & g \arrow[lu] & & & j \arrow[lu] & \text{foo} \arrow[l]
\end{tikzcd}$$

- → Commit: State/Content at a given time
  - → Contains a commit message to describe the changes
  - → Commits always point to their parent(s)
  - → Commits are identified by a hash of the content, message, author(s), parent(s)
- → Branch: A named pointer to a commit
  - → Development branches
  - → Main branch: `master`
  - → Moves to the next child if a commit is added

History

$$\begin{tikzcd}
 & & & & \texttt{ICRC 2017} \arrow[d] & & & & \\
\texttt{a} & \texttt{b} \arrow[l] & \texttt{c} \arrow[l] & \texttt{d} \arrow[l] & \texttt{e} \arrow[l] & \texttt{f} \arrow[l] & \texttt{h} \arrow[l] \arrow[dll] & \texttt{i} \arrow[l] & \texttt{master} \arrow[l] \\
 & & & & \texttt{g} \arrow[ul] & & & \texttt{j} \arrow[ul] & \texttt{foo} \arrow[l]
\end{tikzcd}$$

- → Commit: State/Content at a given time
  - → Contains a commit message to describe the changes
  - → Commits always point to their parent(s)
  - → Commits are identified by a hash of the content, message, author(s), parent(s)
- → Branch: A named pointer to a commit
  - → Development branches
  - → Main branch: `master`
  - → Moves to the next child if a commit is added
- → Tag: Fixed, named pointer to a commit
  - → For important revisions, e.g. release versions or version used for a certain paper

Typical single-branch workflow

If new Create or clone repository : `git init`, `git clone <url>`

If exists `git pull`

1. Work

1.1 Edit files and build/test

1.2 Add changes to the next commit: `git add`

1.3 Save added changes in the history as *commit*: `git commit`

2. Download commits that happend in the meantime: `git pull`

3. Upload your own: `git push`

`git init`, `git clone`

`git init`	Creates a new git repo in the CWD
`git clone <url>`	Clones (downloads) the repo from url
`rm -rf .git`	Deletes all traces of git from the repository

`git status`, `git log`

`git status`	Shows current state of the repository (New, changed, deleted, renamed, untracked files)
`git log`	List the commits of the current branch

`git add`, `git mv`, `git rm`, `git reset`

`git add <file> …`	Add files to the staging
`git add -p …`	Powerfull tool to only add parts of a file
`git mv`	like `mv`, stages automatically
`git rm`	like `rm`, stages automatically
`git reset <file>`	Removes changes/files from the staging area

`git diff`

`git diff`	Show difference between CWD and staging
`git diff --staged`	Show difference between staging and last commit
`git diff <commit1> <commit2>`	Show difference between two commits

`git commit`

`git commit`	Create a new commit from the changes in the stagin area, opens your favourite editor to compose the commmit message
`git commit -m "message"`	Create a new commit giving the message on the commandline
`git commit --amend`	Change the last commit (Adds staging to last commit, message editable)

Never change commits that are already pushed

- → Style guide for commits
  - → First line is title/summary for the commit and should be < 60 characters
  - → Followed by one empty line
  - → Longer description of the changes, e.g. using bullet points.
- → Commits should be small, logical units
  - → `git add -p` very handy
- → “Commit early, commit often”

`git pull`, `git push`

`git fetch`	Download commits from the remote
`git pull`	Download commits and merge current branch with the remote
`git push`	Upload commits

`git checkout`

`git checkout <commit>`	Load a certain commit from the history into the CWD (check with `git log`)
`git checkout <file>`	Reset `<file>` to the last commit, throwing any changes away

Working using multiple branches – GitHub Workflow

There are multiple models of working together with git using branches

Simplest and most popular: “GitHub-Workflow”

- → Nobody directly commits into the `master` branch
- → For each new feature / change / bugfix a new branch is created
- → Branches should be rather shortlived
- → Merge into master as soon as possible, then delete branch
- → Master should always contain a working version

Branches

`git branch <name>`	Create a new branch pointing to the current commit
`git checkout <name>`	Switch to branch `<name>`
`git checkout -b <name>`	Create a new branch and change to it
`git merge <other>`	Merge the changes of branch <other> into the current branch

Beware: Merge conflicts

Happens when git can't merge automatically, e.g. two people edited the same line.

1. Open the files with conflicts
2. Find the lines with conflicts and resolve by manually editing them

```
<<<<<<< HEAD
foo
||||||| merged common ancestors
bar
=======
baz
>>>>>>> Commit-Message
```

3. Commit merged changes:
   3.1 `git add …`
   3.2 `git commit`

Usefull: `git config --global merge.conflictstyle diff3`

`git stash`

`git stash`	Reset CWD to last commit but save the changes in the “stash”
`git stash pop`	Get the saved changes back

`.gitignore`

Many files or filetypes should not be put under version control

- → Compilation results
- → Files reproducibly created by scripts
- → Config files containing credentials
- → …

- → `.gitignore` in the base of a repository
- → One file or glob pattern per line for files that git should ignore

Example:

```
build/
*.so
__pycache__/
```

Github provides a default `.gitignore` for most programming languages:
github.com/github/gitignore

Global `.gitignore`

For some files it makes sense to ignore them globally, for every repository

```
git config --global core.excludesfile $HOME/.gitignore
```

E.g. strange macOS files or editor backup files

```
__MACOSX   # weird mac directory
.DS_STORE  # weird mac metadata file
*.swp      # vim backup files
*~         # nano / gedit backup files
```

Git Hosting Services

Git Hosting Providers

- → Several Providers and self-hosted server solutions available
- → Usually provide much more than just hosting the repositories
  - → Issue tracking
  - → Code review using pull requests
  - → Wiki
  - → Project Management, e.g. Canban boards
  - → Continuous integration
  - → Releases

Git Hosting Providers

GitHub

- → Largest Hoster
- → Many Open Source Projects, e.g. Python
- → Unlimited private repositories for students and reasearch organisations education.github.com

GitLab

- → open-source community edition
- → paid enterprise edition with more features
- → unlimited private repositories
- → Self hosted or as service at gitlab.com

Bitbucket

- → Unlimited private repos with up to 5 contributors
- → Lacks far behind GitHub and GitLab

Git Hosting Providers

GitHub

- → Largest Hoster
- → Many Open Source Projects, e.g. Python
- → Unlimited private repositories for students and reasearch organisations education.github.com

GitLab

- → open-source community edition
- → paid enterprise edition with more features
- → unlimited private repositories
- → Self hosted or as service at gitlab.com

Bitbucket

- → Unlimited private repos with up to 5 contributors
- → Lacks far behind GitHub and GitLab

"Now, everybody sort of gets born with a GitHub account" – Guido van Rossum commenting on Python's move to GitHub

SSH Keys

Git can communicate using two ways with a remote:

HTTP Works out of the box, requires entering credentials at every push/pull

SSH Using keys, you only need to enter the key password once per session

SSH-Keys:

1. `ssh-keygen -t rsa -b 4096 -C "GitHub Key for <username> at <machine>" -f  /.ssh/id_rsa.github`
2. Passwort wählen
3. `cat ~/.ssh/id_rsa.github.pub`
4. Add key to profile

Forking

→ Using git and hosting providers, it's easy to contribute to projects you do not have write access to.

→ This is arguably the most important reason for git's success.

→ Forking means to create a copy of the main repository in your namespace, e.g. `http://github.com/matplotlib/matplotlib` to `http://github.com/maxnoe/matplotlib`

→ You can then make changes and create a pull request in the main repository!

→ To keep you fork up to date, you should add both your fork and the main repo as remotes.

Forks

`git clone <your fork>`	Clone your fork
`git add remote upstream <main repo>`	Add the main repo
`git fetch upstream`	Download changes from the main repo
`git reset --hard upstream/master`	Reset the current branch to the master of the main repo to synchronize with the changes

Integration with Issue Tracking

Start working on fixing a bug, that was documented in issue 42.

```
$ git checkout -b fix_42

... do stuff to fix bug ...

$ git add src/foo.cxx
$ git commit -m "Fix segmentation fault when doing stuff, fixes #42"
$ git push -u origin fix_42
```

If this commit get's merged into master, issue 42 will automatically be closed.

Continuous Integration

Continuous Integration

- → Automatically run your test suite on new pushes and pull requests
- → Let’s you see if a PR will break or fix stuff
- → Automatically create releases on tagged versions
- → Build and upload documention
- → …

Travis-CI

→ Travis provides free CI linux servers for public github repositories

→ Configured by a `.travis.yml` file in the repo

Git compared to other VCS

Git compared to other VCS

→ Only widely used alternative is `SVN`.

→ Outdated and not maintained anymore e. g. `CVS` (last release 13 years ago)

→ Mercurial (`hg`) rarely used alternative (`Python` just moved from `hg` to Git(Hub))

Git vs. SVN

Git

- + Faster
- + Full history available offline
- + Cheap branching
- + Much better tooling (GitHub/GitLab)
- + Branch/Fork → Pull Request Workflow (Outside Contribution, Code Review)

- – Harder to learn

SVN

- + Simpler

- – Slower
- – Accessing the history needs server connections
- – Branching/Merging is expensive and not well supported