



UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386



Tools 2026 – GIT Intro

Shigehiko Schamoni
21.05.2026

With material from F. Beenen (SuS)

Overview

1 Git Basics

- Introduction
- Commits
- References
- Housekeeping

2 Git Commands

- Initializing a Repository
- Inspecting the Repository
- Creating New Commits
- Branches
- Emergency Help

3 Working with Remotes

- Setting up Remotes
- Synchronizing a Repository
- Automated GitLab Jobs

What is Git?

- A fast, scalable, distributed revision control system
- Unusually rich command set
- Provides both high-level operations and full access to internals

The name “git” was given by Linus Torvalds when he wrote the very first version. He described the tool as “the stupid content tracker” and the name as (depending on your mood):

- random three-letter combination that is pronounceable, and not actually used by any common UNIX command. [...]
- stupid. contemptible and despicable. simple. [...]
- “global information tracker”: you’re in a good mood, and it actually works for you. [...]
- “goddamn idiotic truckload of sh*t”: when it breaks

(Source: <https://github.com/git/git/blob/master/README.md>)

So, what is Git??

- Software to keep track of file versions
- Similar software: SVN (Subversion), Mercurial
- Git can be used for managing:
 - Simple programming exercises (e.g. for lectures)
 - Bachelor/Master projects and theses
 - Text files and $\text{\LaTeX}$ documents
 - Personal notes
 - ...
- In principle, Git can work with any kind of file.
- In practice, Git works best for text-based files where *partial changes* occur over time.

Why use Git?

Git is useful because:

- You can *track changes* to files:
 - Who did what, why and when?
 - What did I do today?
- You can try out new stuff without breaking an existing project.
- You cannot accidentally lose data when using delete files too aggressively.
- You can track errors back in time to find the root cause.
- You can collaborate with multiple people on the same project.
- You can use Git to do really awesome stuff (automated building, testing, deploying, etc.).

Git Terms

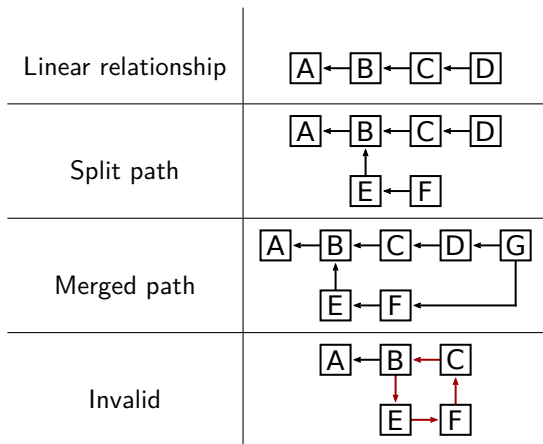
- A project that is managed with Git is called *repository*.
- A repository is saved on your computer.
- You can share a repository with other people/computers to collaborate.
- Data points where file versions are saved are called *commits*.
- You can access old commits.
- You can compare different commits and find out what has changed.

Git Commits

- Every Git commit has some meta data assigned:
- Name of the user who issued the commit.
- Time when the commit was issued.
- Description of the commit: Has to be created by the user in a meaningful way.
- A reference to the previous commit(s).
- Git commits are identified by a name which is formed by a SHA-256 hash
(e.g. 64efe35c4535b472d2fe38a6f696adc56dc38ff4).
- Meant to be unique for all commits on the world.
- Collision attacks are theoretically possible but unlikely.

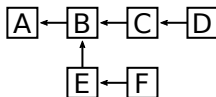
Commit Graph

- Every commit has at least one predecessor (except for the first commit).
- Commits cannot have circular dependency: the graph is always acyclic and directed.



References

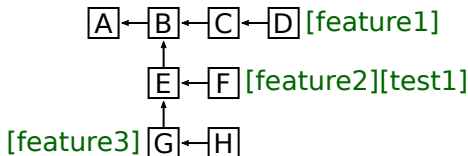
- If a commit has multiple parents, it is a *merged commit*.
- From one commit, only the parent commits are available:



- From commit E the history is $A \leftarrow B \leftarrow E$.
- All other commits are not reachable.
- The repository therefore needs to store where the “newest” commits are.

Branches

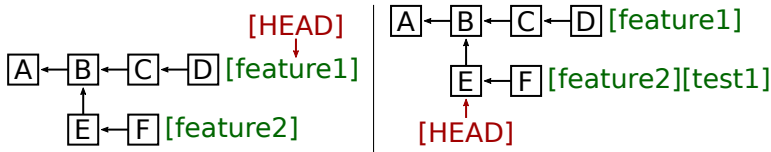
- A *branch* is the most common form for a reference to a Commit.
- A branch gives a name to a commit.
- Multiple branches may point to the same commit.
- Branches do not necessarily point to the newest commits.



- In this graph, commit *H* is not referenced by any branch and therefore inaccessible.
- If you know the name of the commit (the hash), you can still create a branch for it.
- Inaccessible commits will be purged eventually by the garbage collector.

Special Reference: HEAD

- There is a special reference called *HEAD*.
- It is used to point to the current “active” branch.
- The active branch is the one you are currently working on, i.e. where you create commits.
- Normally, HEAD is a reference to a branch, not a commit.
- You can set HEAD to a commit by specifying its hash (“Detached HEAD”).



Project State

- There are three places where file changes are recorded:
 - The latest commit – the branch that is referenced by HEAD.
 - The current working directory: you may have edited some files (this has nothing to do with Git).
 - The *index*: A staging area where files are collected that should be included in the next commit.
- The Repository is regarded “clean” if all three data sets are identical.
- What you have to do in order to create a new commit:
 - 1 Edit your files – do the actual work on the project.
 - 2 Mark files that you want to commit. Add them to the index.
 - 3 After all files are marked / the index is updated correctly, issue a commit and specify the corresponding message.

The Index

- Why do we need an *index*?
- This sounds like unnecessary bureaucracy – we could just directly put all changed files into one Commit automatically!
- Advantages of separate index:
 - Split file changes in different commits.
 - Splitting is useful to separate changes in a logical way, e.g.:
 - 1 Fixed a security bug in the server application. $\Rightarrow$ One commit.
 - 2 Re-wrote a function in module xyz. $\Rightarrow$ Another commit.
 - Commit messages are more meaningful, it is easier to understand what has happened when inspecting the commit graph.

A Summary of Git Usage



Figure: <https://xkcd.com/1597/>

Fundamentals

- To use Git you must install Git on your system.
- Install with `sudo apt install git` on Debian-like Linux distributions.
- You manage everything Git-related with the `git` command.
- Get help: <https://git-scm.com/docs> and `man git <subcommand>`.
- When using Git for the first time:
 - Set your (real) name:
`git config --global user.name <first name> <last name>`.
 - Set your mail address:
`git config --global user.email <e-mail address>`.
 - This data is included in every Commit! Please set it to something meaningful.
 - Enable colorful output:
`git config --global color.ui auto`.
 - Set the preferred text editor:
`git config --global core.editor <editor name>`
(e.g. `nano`, `vim` or `gedit`).

Initializing a Git Repository

- Create a fresh Repository with `git init`.
- If a Repository already existed, nothing is overwritten $\Rightarrow$ It is safe to run `git init` on an existing Repository.
- The command will create the hidden directory `.git` inside your current working directory.
- The `.git` directory contains everything that Git needs to operate (all the Commits, meta data, Branches, ...).
- You should not delete / move / edit the `.git` directory.

Cloning a Git Repository

- If you already have a Git repository you can download / clone it.
- Use `git clone <url>`. Example:

```
git clone https://sus.ziti.uni-heidelberg.de/Lehre/WS1718_Tools/GIT/uebung.git .
```
- Creates a new directory with the name of the cloned repository.
- Give it a custom name with

```
git clone <url> <local-repo-name> .
```
- You can either clone with HTTP(s) or with SSH (preferred).
- Cloning and syncing with remote repositories is discussed later!

Check Repository State

- Inspect the working set with `git status`.
 - Lists files on the Index (planned for next Commit).
 - Lists changed files that are not on the Index.
 - Lists deleted files that are still present in the Repository.
- Check the current Branch: `git branch`.
 - There is always the master Branch.
 - You can see which Branch is currently active.
- Check Commit history with `git log`.
 - See all Commits of the current Branch.
 - Check out where the HEAD reference is pointing.

Inspecting a Commit

- Use `git show <Commit>` to inspect a specific Commit.
 - Can be given a Commit hash, a Branch name or another reference.
 - Shows the files that have changed.
 - Shows individual lines that have been added or deleted.
 - Show changes relative to previous Commit.
 - Changes are displayed properly only for text-based files (e. g. program sources, $\text{\LaTeX}$ files, etc.)
- Differences between Commits:
`git diff <Commit1> <Commit2> .`
 - Useful to manually compare two Commits.
 - Also get information on every changed line.

Working with the Index

- In order to create a new Commit, you need to populate the Index.
- Add new files to the Index with `git add <file>`.
 - You can use Glob Expressions like `git add *.tex` to add all `.tex` files in the current directory to the Index.
 - You can use `git add -u` to add all changed files to the Index.
 - If you edited a file, it will now be on the Index.
 - If you deleted a file manually, the file deletion will be on the Index.
 - If you created a new file that is not in the Repository, it will not be on the Index. Perform `git add <file>` separately.
- To check which files are on the Index use `git status`.
- To remove a file from the Index again use `git reset <file>`.
- To delete a file from disk and put the deletion on the Index use `git rm <file>`.
 - If you only want to delete the file from the Repository but not from disk use `git rm --cached <file>`.

Ignoring Files

- You can configure your environment that you can perform `git add *` and not add any “garbage”.
- You may want to exclude certain directories or file types from ever getting in the Repository.
 - Binary files that you can generate automatically (e.g. compiled programs, FPGA bitfiles, compiled $\text{\LaTeX}$ files, ...).
 - Log files that are useless.
 - Local configuration files that are bound to your computer but are useless for other people (e.g. Eclipse workspace config, ...)
- You can write a `.gitignore` file (the dot is important).
 - List of files and directories that are ignored by Git.
 - Can contain Glob Expressions.
 - Ignore every `.txt` file: `*.txt`.
 - Ignore log files only in repository root: `/*.log`.
 - Ignore all directories named “build”: `build/`.
- Check with `git status` that it does not report useless “untracked files”. Populate your `.gitignore` until `git status` does not report any false positives any more.

Creating a Commit

- Once you are satisfied with the status of the index, you can create the *commit*.
- Commit the index with `git commit`.
- The configured text editor will open and you need to enter a *commit message*.
- Make it a *useful* message that describes what exactly you did (e.g. *not* “Changes of today”).
- You can also directly specify the message without a text editor:
`git commit -m "my message"`.

Branches

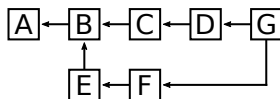
- Inspect current Branch with `git branch`.
- Create new Branch with `git branch <name> [<commit>]`.
 - Creates the Branch with the given name.
 - Optionally creates the Branch based on the given commit hash.
- Change the Branch with `git checkout <branch>`.
 - Potentially changes the content of files managed by Git.
 - Will not touch untracked files.

Deleting Branches

- Delete a Branch with `git branch -d <branch>`.
- If Git complains, there are Commits on that Branch that are not otherwise referenced.
- If you delete the Branch, you will lose data!
- Either merge the Branch somewhere else before deleting.
- Or forcefully delete the Branch with `git branch -D <branch>`.

Merging Branches (I)

- You normally keep your “golden” working version on the master branch.
- It is good practice to never publish stuff on master that is obviously broken/work in process.
- Develop your stuff on an extra Branch. Test it extensively. Then merge it to master and delete your development Branch.
- To merge a Branch:
 - 1 Switch to the target Branch.
 - 2 Run `git merge <other-branch>`.
 - 3 Potentially resolve conflicts.
 - `git status` will tell you what is going on.
 - Manually go through the files and decide what shall be used.
 - Commit your changes.



Merging Branches (II)

- If you can merge directly, no conflicts exist.
- This is the case if the set of files modified on the different Branches is disjoint.
- Special case: The target Branch did not diverge: fast-forward (no separate merge Commit exists).
- Otherwise, Git does not know what to do. You need to inspect all offending files.
- When you are done, `git add` the changed files.
- Commit your changes.

Emergency Help

- Sometimes you screw up your Repository.
 - Accidentally have garbage in a Commit (e.g. add large binary data to the repository).
 - Deleted important files.
 - A screwed-up merge.
- Direct help (Warning: These commands can lose you data):
 - `git checkout -- <file>` : Restore/Reset a file to the version from the Repository.
 - `git reset --soft <Commit>` : Sets HEAD to the given Commit but does not touch anything.
 - `git reset [--mixed] <Commit>` : Resets the Index but does not touch the working directory.
 - `git reset --hard <Commit>` : Resets Index and working directory to given Commit (this may cost you unsaved data).
- Tremendously helpful website on how to “un-screw” your Git:
<https://sethrobertson.github.io/GitFixUm/fixup.html>

Remote Repositories



- Until now we only worked on our local machine.
- Git becomes very useful when sharing the Repository.
 - Other people can work on the Repository.
 - You have a graphical overview with GitHub/GitLab.
 - You can use external services for automation.
- I recommend to use GitLab:
 - It is free.
 - Unlike commercial solutions, it has almost no limitations.
 - You can run your own server without feeding someone's AI mouth.
- We have a GitLab instance for the Tools lecture here:
`https://ziti-peac.ziti.uni-heidelberg.de`. Just log in with your UnilD.

Register SSH Key

- In order to synchronize with the remote Repository, it is best to register your SSH public key with GitLab.
- Within Gitlab, click on your profile picture → Settings → SSH Keys.
- Obtain your SSH key (`cat .ssh/id_ed25519.pub`).
- Paste it in the text box.

Clone a Repository

- You can create a new Repository via the web UI.
- You can clone it via the Clone button, use SSH.
- Get your terminal and enter `git clone <ssh-url>`.
- Example:

```
git clone git@ziti-peac.ziti.uni-heidelberg.de:myuser/testproject.git
```

Inspecting the Remote

- Your local Git repository that you cloned from GitLab is now linked to GitLab.
- This “link” is called a *remote*.
- Check out your remotes by executing `git remote -v`.
- By default the remote is called *origin*.

Synchronizing with the Remote

- When you issue commits locally, you need to send them to the Remote as well.
 - `git push` will send your local commits.
 - If you get errors, the Remote may have commits that you don't know about (e. g. by someone else). You need to load these commits first.
- When the remote is updated you need to download the new commits.
 - Load the commits with `git fetch`.
 - Merge them with your local branch.
 - Use `git pull` to perform `git fetch` and `git merge` simultaneously.
 - You will get conflicts if fast-forward is not possible.

Automated GitLab Jobs

- You can trigger automated actions in GitLab:
 - When a commit to `master` is made.
 - When a commit is made that updates `.tex` files.
 - On a regular basis, e. g. every Wednesday on 9:00.
 - ...
- Automation is based on Docker:
 - You create or supply a base image: Basically a throw-away Linux instance with specific configuration.
 - Docker container ensures that the automated job has all the software it needs in the correct version to create reliable output.
 - Get Docker containers at: <https://hub.docker.com/>
- Configuration is done in `.gitlab-ci.yml` which must be located in the repository's root directory.

Example: Automated $\text{\LaTeX}$ Compilation

- If you have $\text{\LaTeX}$ in your Repository, you can automatically compile a PDF.
- You may use the Docker image `fbeenen/texlive`. It has the following packages installed: `git unzip curl openssh-client texlive-full`
- YAML files are sensitive with regard to whitespace. Do not mix tabs and spaces. Use the correct amount of indentation!
- Minimalistic `.gitlab-ci.yml`:

```
image: fbeenen/texlive

build:
  script:
    - latexmk -pdf raytrace.tex
  artifacts:
    paths:
      - "*.pdf"
```

Example: More Sophisticated L^AT_EX Compilation

```
image: fbeenen/texlive
variables:
  PDF_PREFIX: "MyProj"
build:
  before_script:
    - apt-get update && apt-get -y install wget
  script:
    - cd latex
    - source autobuild.sh
    - buildPDF mainfile.tex
    - mv *.pdf ../
    - cd ../praesentation/
    - buildPresentation myPresentation.tex
    - mv *.pdf ../
  only:
    changes:
      - "**/*.tex"
  artifacts:
    paths:
      - "${PDF_PREFIX}*.pdf"
```

Artifacts

- Your `.gitlab-ci.yml` should specify some output products: Artifacts.
- These serve two purposes:
- They are used to deliver files to the next stage of the pipeline.
- They can be downloaded by the user.
- Multiple pipeline stages can be chained: For software e.g. “Build”, “Test”, “Deploy”.