Git Introduction

Jens Lechtenbörger

Summer Term 2023

Contents

1 Introduction

1.1 Learning Objectives

- Discuss benefits and challenges of version control systems (e.g., in the context of university study) and contrast decentralized ones with centralized ones
- Explain states of files under Git and apply commands to manage them
- Explain Feature Branch Workflow and apply it in sample scenarios
- Edit simple Markdown documents

Learning objectives specify what you should be able to do after having worked through a presentation. Thus, they offer guidance for your learning.

Each learning objective consists of two major components, namely an *action* verb and a topic. Action verbs specify what actions you should be able to perform concerning the topic, and they indicate the target level of skill (in Bloom's Taxonomy or its revised version as sketched under the hyperlink above).

You may want to think of learning objectives as sample exam tasks.

1.2 Core Questions

- How to **collaborate** on shared documents as distributed team?

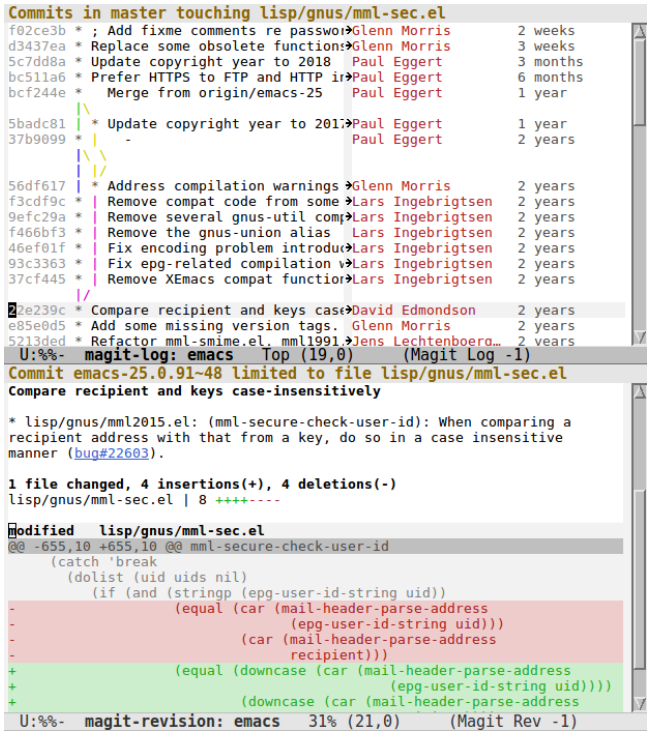

Figure 1: "Magit screenshot" under CC0 1.0; from GitLab

  - Consider **multiple** people working on **multiple** files
    * Potentially in **parallel** on the **same** file
    * Think of group exercise sheet, project documentation, source code

- How to keep track of who changed what why?
- How to support unified/integrated end result?

1.3 Your Experiences?

- Briefly write down your own experiences.
  - Did you collaborate on documents
    * by sending them via e-mail,
    * by using shared (cloud) storage (e.g., Sciebo with OnlyOffice, Google),
    * by using collaborative editors (e.g., Sciebo with OnlyOffice, Etherpad, HedgeDoc, Overleaf)
    * by using version control systems (e.g., Git, SVN),
    * by using other means?
  - Why did you choose what alternative? What challenges arose? Do you bother to read Terms of Service when you entrust "your" documents and thoughts (each individual keystroke, including "deleted" parts) to third parties (e.g., in the cloud)?

1.4 Version Control Systems (VCSs)

- Synonyms: Version/source code/revision control system, source code management (VCS, SCM)
- Collaboration on **repository** of documents
  - Each document going through various versions/revisions
    - Each document improved by various authors
      - April 2012, Linux kernel 3.2: 1,316 developers from 226 companies

1.4.1 Major VCS features

- VCS keeps track of **history**
  - Who changed what why when?

Figure 2: "Meeting arrows" under CC0 1.0; rotated from Pixabay

  - Restore/inspect old versions if necessary
- VCS supports **merging** of versions into **unified/integrated** version
  - Integrate intermediate versions of single file with changes by multiple authors
- Copying of files is **obsolete** with VCSs
  - Do **not** create copies of files with names such as `Git-Intro-Final-1.1.txt` or `Git-Intro-Final-reviewed-Alice.txt`
    - Instead, use VCS mechanism, e.g., use tags with Git

2 Git Concepts

2.1 Git: A Decentralized VCS

- Various VCSs exist

  - E.g.: Git, BitKeeper, SVN, CVS
    * (Color code: decentralized, centralized)

- Git created by Linus Torvalds for the development of the kernel Linux
  - Reference: Pro Git book

Figure 3: "Git Logo" by Jason Long under CC BY 3.0; from git-scm.com

  - Git as example of **decentralized** VCS
    * Every author has **own copy** of all documents and their history
    * Supports **offline** work without server connectivity
      · Of course, collaboration requires network connectivity
    * Distributed trust/control/visibility/surveillance

2.2 Key Terms: Fork, Commit, Push, Pull

- **Fork/clone** repository: Create copy of repository

Figure 4: "Folder" under CC0 1.0; derived from Pixabay

  - Clone: Create copy of remote repository on your machine
  - Fork: Create copy within online Git platform; then clone that

- **Commit** (aka check-in)

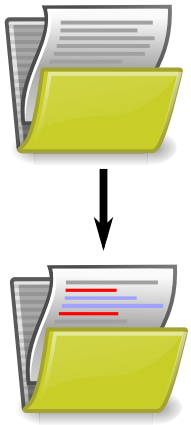

Figure 5: "Folder" under CC0 1.0; derived from Pixabay

- Make (some or all) changes permanent; announce them to version control system
- **Push**: Publish (some or all) commits to remote repository
  - Requires authorization
- **Fetch** (**pull**): Retrieve commits from remote repository (also merge them)

Some Git repositories are *public*, which means that everyone is allowed to read and copy their contents. For example, this is the case for free software projects and open educational resources. Obviously, owners of projects want to keep control over their repositories, so they do *not* allow random people to *change* files.

The Git command line operation to copy a repository to your local machine is called `git clone`. This operation downloads all files and the history of some project, e.g.: `git clone https://gitlab.com/oer/cs/programming.git`

As you own the cloned repository on your local machine, you can change everything to your liking. With `git commit`, you record changes as permanent, to be remembered by the version control system. E.g., this commit shows a simple improvement of wording while that one introduces larger changes of slides.

To publish local commits to a repository, use `git push`. However, if you just cloned someone else's source project, you will *not* be allowed to integrate your commits into the source project. Thus, `git push` would fail. In contrast, you can push to your own repositories (and those where you were granted appropriate permissions).

On Web based Git platforms such as GitLab, you can create your own fresh projects, to which you are allowed to push commits.

To contribute to someone else's project, such platforms provide a so-called *fork* operation, creating a full copy of the source, which is also called *upstream*. You can clone your fork to your own machine and apply commits. This time, however, you own the repository from which you cloned and you are allowed to push commits to that repository.

Also, as we will see later on, such platforms provide so-called merge requests (or pull requests), with which you can propose commits on your forked project to be integrated into the upstream project.

Also, if you clone some repository you receive all commits up to that point in time. Over time, more commits may be applied to the repository, and Git offers fetch and pull operations to retrieve those.

2.3 Key Terms: Branch, Merge

- **Branches**

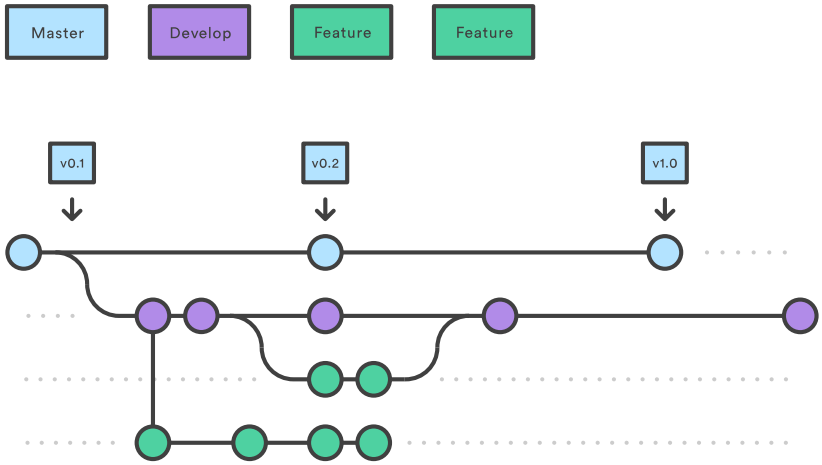

Figure 6: "Git Branches" by Atlassian under CC BY 2.5 Australia; dimension attributes added, from Atlassian

  - Alternative versions of documents, on which to commit
    * Without being disturbed by changes of others
    * Without disturbing others
      · You can share your branches if you like, though

- **Merge**
  - Combine changes of one branch into another branch
    * May or may not need to resolve conflicts

- (Don't worry if this seems abstract, we'll try this out.)

With VCSs, different independent developments can happen at the same time in a single project. For example, while one team member adds some feature to the UI, a second one might fix a bug, and a third one might improve the backend. To isolate those changes from each other, they can take place in so-called *branches*, where the default branch is called `master` (or `main`) and represents a stable version. Later on, the commits of other, "finished" branches can be merged into the `master` branch.

Note that here we see four branches, the `master` branch in blue, one for ongoing development in purple and two feature branches in green. The feature branches were created from different states of the development branch: the first one, let's call it `f1`, contains the changes of one purple commit; the second one, say `f2`, contains two purple commits. Note that `f2` introduces two commits while the purple branch independently advances with another commit, before `f2` is *merged* into the purple branch with a so-called *merge commit*. At this point in time, the changes of all commits of `f2` are integrated into the purple branch, and `f2` itself is no longer particularly interesting.

For some point in the future, we should expect `f1` to be merged into the purple branch and the purple branch to be merged into `master`.

2.4 Git explained by Linus Torvalds

- Video at archive.org (Tech Talk, 2007, by Google Talks under CC BY-NC-SA 3.0)
  - Total length of 84 minutes, suggested viewing: 7:40 to 29:00

2.4.1 Review Questions

Prepare answers to the following questions

- What is the role of a VCS (or SCM, in Torvalds' terminology)?
- What differences exist between decentralized and centralized VCSs?
  - By the way, Torvalds distinguishes centralized from distributed SCMs. I prefer "decentralized" over "distributed". You?

3 Git Basics

3.1 In-Browser Tutorial

- Some students recommended this tutorial to try out Git commands in browser: `https://learngitbranching.js.org/`
  - Several levels of the tutorial cover Git commands that appear on later slides
    - Tab "Main", Level "1: Introduction to Git Commits" introduces commit, branch, merge, rebase
    - Tab "Remote", Level "1: Clone Intro" introduces clone, fetch, pull, push

3.2 Getting Started

- Quickstart for Git installation and setup for GitLab
  - (Actually, you can only follow sections 4 and 5 after Git accounts have been set up)
  - At least, install Git and perform first-time setup
- You may use Git without a server
  - Run `git init` in any directory
    - Keep track of your own files
  - By default, you work on a branch called `main` or `master`
    - That branch is not more special than any other branch you may create
    - (The term "master" is offensive; migration to "main" is under way in lots of places)

3.3 Accessing Remote Repositories

- **Download** files from public repository: `clone`
  - `git clone https://gitlab.com/oer/cs/programming.git`
    - Change into that directory: `cd programming`
      - Try out Git commands (but not `git push`, which you are not allowed here)

        - Later on, `git pull` merges changes to bring your copy up to date

- **Contribute** to remote repository
  - Push commits as explained earlier and revisited later on

3.3.1 A quick check

3.4 First Steps with Git

- Prerequisites
  - You installed Git
  - You performed the First-time Git setup
- Part 0
  - Create repository or clone one
    - `git clone https://gitlab.com/oer/cs/programming.git`
    - Creates directory `programming`
      - Change into that directory
      - Note presence of "real" contents and of sub-directory `.git` (with Git meta-data)

3.4.1 Part 1: Inspecting Status

- Execute `git status`
  - Output includes current branch and potential changes
- Open some file in text editor and improve it
  - E.g., add something to `Git-Introduction.org`
- Create a new file, say, `test.txt`
- Execute `git status` again
  - Output indicates
    - `Git-Introduction.org` as **not staged** and **modified**
    - `test.txt` as **untracked**
    - Also, follow-up commands are suggested
      - `git add` to stage for commit
      - `git checkout` to discard changes

3.4.2 Part 2: Staging Changes

- Changes need to be `staged` before commit
  - `git add` is used for that purpose
  - Execute `git add Git-Introduction.org`
  - Execute `git status`
    - Output indicates `Git-Introduction.org` as **to be committed** and **modified**
- Modify `Git-Introduction.org` more
- Execute `git status`
  - Output indicates `Git-Introduction.org` as
    - **To be committed** and **modified**
      - Those are your changes added in Part 1
    - As well as **not staged** and **modified**
      - Those are your changes of Part 2

3.4.3 Part 3: Viewing Differences

- Execute `git diff`
  - Output shows changes that are not yet staged
    - Your changes of Part 2
- Execute `git diff --cached`
  - Output shows difference between staged changes and last committed version
- Execute `git add Git-Introduction.org`
- Execute both `diff` variants again
  - Lots of other variants exits
    - Execute `git help diff`
    - Similarly, **help** for other `git` commands is available

3.4.4 Part 4: Committing Changes

- Commit (to be committed) changes
  - Execute `git commit -m "<what was improved>"`
    - Where `<what was improved>` should be meaningful text
    - 50 characters or less
- Execute `git status`
  - Output no longer mentions `Git-Introduction.org`
    - Up to date from Git's perspective

  - Output indicates that your branch advanced; `git push` suggested for follow-up

- Execute `git log` (press `h` for help, `q` to quit)
  - Output indicates commit history
  - Note your commit at top

3.4.5 Part 5: Undoing Changes

- Undo premature commit that only exists **locally**
  - Execute `git reset HEAD~`
    * (**Don't** do this for commits that exist in remote places)
  - Execute `git status` and `git log`
    * Note that state before commit is restored
    * May apply more changes, commit later
- Undo `git add` with `git reset`
  - Execute `git add Git-Introduction.org`
  - Execute `git reset Git-Introduction.org`
- Restore committed version
  - Execute `git checkout -- <file>`
  - **Warning**: Local changes are **lost**

3.4.6 Part 6: Stashing Changes

- Save intermediate changes without commit
  - Execute `git stash`
    * If you performed `git checkout ...` on previous slide, change some file first
  - Execute `git status` and find yourself on previous commit
- Apply saved changes
  - Possibly on different branch or after `git pull`
  - Execute `git stash apply`
    * May lead to conflicts, to be resolved manually

3.4.7 Part 7: Branching

- Work on different branch
  - E.g., introduce new feature, fix bug, solve task
  - Execute `git checkout -b testbranch`
    - Option `-b`: **Create** new branch and switch to it
      - (Leave out for switch to existing branch)
  - Execute `git status` and find yourself on new branch
    - With uncommitted modifications from `main` (or `master`)
    - Change more, commit on branch
    - Later on, merge or rebase with `main`
  - Execute `git checkout main` and `git checkout testbranch` to switch branches
    - (Newer versions of `git` know `git switch` for the same purpose)

3.4.8 Remotes (1)

- Show remote repositories, whose changes you track:
  - `git remote -v`
    - By default, remote after `git clone` is called `origin`
    - No remote exists after `git init`
    - For a forked project, one usually adds an `upstream` remote (see next two slides)
- Contribute to project, two variants
  1. Operation `push` (requires permission)
     - You can `push` to your own projects
     - E.g., push new branch to remote `origin`:
       - `git push -u origin testbranch`
  2. Use merge/pull requests for other projects (next slide)

3.4.9 Remotes (2)

- Contribute to some project, the `upstream` (section in Pro Git)
  - Projects follow different **workflows**; read project's contribution instructions first
  - E.g., (Forking) **Feature Branch Workflow**
    - Fork upstream project (in GUI)
      - Which creates **your own** project with full permissions
    - Clone it
    - Create separate branch for **each** independent contribution
      - E.g., bug fix, new feature, improved documentation
      - Commit, push branch (to fork)

    * In GUI, open **merge request** (GitLab) or **pull request** (GitHub) for branch
        · If accepted, its changes are merged into upstream project

Different projects follow different workflows regarding their use of branches. Unfortunately, terminology concerning such workflows is not unified, and several alternatives exist.

This slide focuses on a specific *feature branch workflow* to contribute to someone else's project, say, one which we found on GitLab.com. That project is the "upstream project", or "upstream" for short.

Suppose we aim to contribute an upstream feature, which may add some functionality or fix a bug. As we do not have any permissions on upstream, we first *fork* the upstream project, i.e., we create our own copy of the project, also called "a fork", with which we are allowed to do whatever we want (from Git's perspective at least; license information specifies what we are allowed to do from a legal perspective—clearly, free/libre and open source licenses are necessary for collaboration and sustainability).

Then, we *clone* this fork to our local machine.

Locally, we *create a new branch* for our feature, which is why the workflow is called "feature branch workflow". Then, we develop the feature, involving at least one *commit* on the feature branch, before we finally *push* the completed feature branch to our own fork. (To avoid merge conflicts, it is good practice to *rebase* the feature branch on the most recent state of upstream—see also the next slide for a rebase operation in a different context.)

Finally, we create a *merge request* in the upstream project, asking that some maintainer merges our feature into their code.

Besides, concerning terminology, note that an initial fork is largely orthogonal to the use of feature branches: If you are permitted to push new branches to a project, forking is not necessary, and the resulting workflow may be called feature branch workflow, while the workflow including the fork operation as outlined on this slide is also called forking workflow or GitHub flow elsewhere.

3.4.10 Remotes (3)

- When merge request was accepted upstream, maybe update your fork to mirror upstream's state
  - Goal: Update your `master` branch based on upstream's `master` branch
  - Approach
    * Set up source project as remote `upstream`:
      · `git remote add upstream <HTTPS-URL of source project>`
    * Fetch `upstream`: `git fetch upstream`
    * Integrate `upstream/master` into your `master`, maybe with rebase:
      · `git checkout master`
      · `git rebase upstream/master`
    * Push updated master to your fork: `git push`

A point that might be confusing at first is the following one: We talk about *different* `master` (and other) branches. When you fork a project, the source project and your fork both contain a `master` branch, and while those branches contain the same commits initially, they may *diverge* over time: You may apply commits to your `master`, while the source project also evolves. Thus, those two branches are really different things. As you see on the previous and this slide, we can use names such as `origin` and `upstream` to name different remote repositories, which allows us to distinguish different versions of branches such as `origin/master` from `upstream/master`.

In addition, when you clone a repository, you create a `master` branch on your local machine, which again is different from the one in the remote repository. You may change that local `master` branch, while others might change `master` in the remote repository. Thus, those two branches are different things, and we use `push` and `fetch` or `pull` to synchronize them.

It is important to note that commits on a single branch are ordered *linearly*. Thus, suppose that you clone a remote repository, where the latest commit on `master` is C0. You then add commit C1 to your local `master` branch (after C0), and someone else concurrently adds C2 to the remote `master` (after C0). In this situation, attempts to `push` the local `master` branch would *fail* (because the two commits C1 and C2 both have the same parent commit C0 and are not ordered with respect to each other). In this situation, one needs to `merge` or to `rebase` to integrate both master branches (which you will explore in the exercises).

Workflows based on branches (as mentioned on the previous slide) help to gain control over such situations.

3.4.11 Review Questions

- As part of First Steps with Git, `git status` inspects repository, in particular file **states**
  - Recall that files may be `untracked`, if they are located inside a Git repository but not managed by Git
  - Other files may be called `tracked`
- Prepare answers to the following questions
  - Among the `tracked` files, which states can you identify from the demo? Which commands are presented to perform what state transitions?
  - Optional: Draw a diagram to visualize your findings

3.5 Merge vs Rebase

- Commands `merge` and `rebase` both unify two branches
- Illustrated subsequently
  - Same unified file contents in the end, but different views of history

3.5.1 Merge vs Rebase (1)

- Suppose you created branch for new `feature` and committed on that branch; in the meantime, somebody else committed to `master`

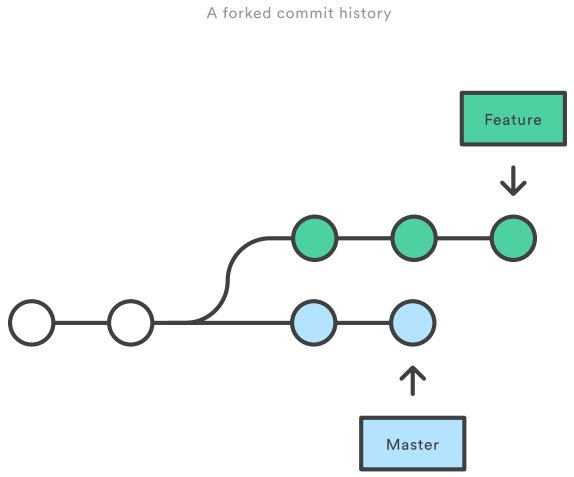

Figure 7: "A forked commit history" by Atlassian under CC BY 2.5 Australia; from Atlassian

3.5.2 Merge vs Rebase (2)

- Merge creates **new** commit to combine both branches
  - Including all commits
  - Keeping parallel history

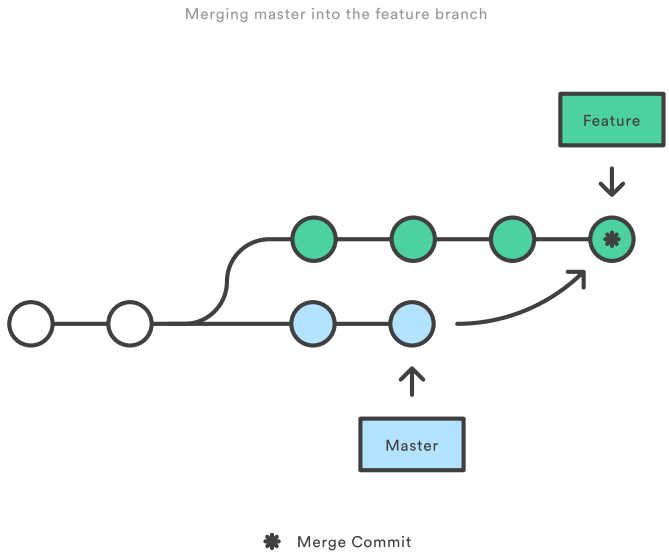

Figure 8: "Merging" by Atlassian under CC BY 2.5 Australia; from Atlassian

3.5.3 Merge vs Rebase (3)

- Rebase rewrites `feature` branch on `master`

- Applies **local** commits of `feature` on `master`
- Cleaner end result, but branch's history lost/changed
  - Only do this for local commits (i.e., before you pushed `feature`)
    - Rebase changes history, so use `merge` for remote branches

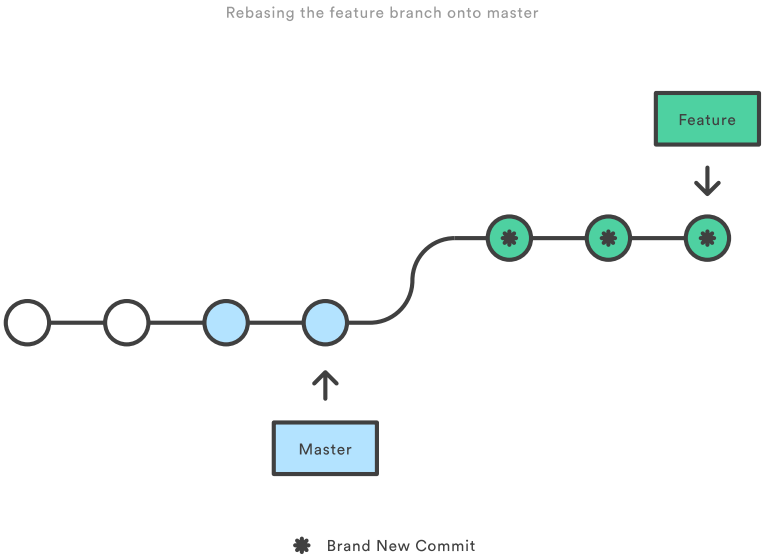

Figure 9: "Rebasing" by Atlassian under CC BY 2.5 Australia; from Atlassian

3.6 Sample Commands

```
git clone <project-URI>
# Then, later on retrieve latest changes:
git fetch origin
# See what to do, maybe pull when suggested in status output:
git status
git pull
# Create new branch for your work and switch to it:
git checkout -b nameForBranch
# Modify/add files, commit (potentially often):
git add newFile
git commit -m "Describe change"
# Push branch:
git push -u origin nameForBranch
# Ultimately, merge or rebase branch nameForBranch into branch master
git checkout master
git merge nameForBranch
# If conflict, resolve as instructed by git, commit.  Finally push:
git push
```

4 GitLab

4.1 GitLab Overview

- Web platform for Git repositories
  - https://about.gitlab.com/
  - Free software, which you could run on your own server
- Manage Git repositories
  - Web GUI for forks, commits, pull requests, issues, and much more
  - Notifications for lots of events
    - Not enabled by default
  - So-called Continuous Integration (CI) runners to be executed upon commit
    - Based on Docker images
    - Build and test your project (build executables, test them, deploy them, generate documentation, presentations, etc.)

4.2 GitLab in Action

- Exercise (during session)

5 Aside: Lightweight Markup Languages

5.1 Lightweight Markup

- Markup: "Tags" for annotation in text, e.g., indicate sections and headings, emphasis, quotations, ...
- Lightweight markup
  - ASCII-only punctuation marks for "tags"
  - Human readable, simple syntax, standard text editor sufficient to read/write
  - Tool support
    - Comparison and merge, e.g., three-way merge
    - Conversion to target language (e.g. (X)HTML, PDF, EPUB, ODF)
      - Wikis, blogs
      - pandoc can convert between lots of languages

5.2 Markdown

- Markdown: A lightweight markup language
- Every Git repository should include a README file
  - What is the project about?
  - Typically, `README.md` in Markdown syntax
- Learning Markdown
  - In-browser tutorial (source code under MIT License)
  - Cheatsheet (under CC BY 3.0)

5.3 Org Mode

- Org mode: Another lightweight markup language
  - My favorite one
- For details see source file for this presentation as example

6 Conclusions

6.1 Summary

- VCSs enable collaboration on files
  - Source code, documentation, theses, presentations
- Decentralized VCSs such as Git enable distributed, in particular offline, work
  - Keeping track of files' states
    - With support for subsequent merge of divergent versions
  - Workflows may prescribe use of branches for pull requests
- Documents with lightweight markup are particularly well-suited for Git management

6.2 Where to go from here?

- Version control is essential for DevOps
  - Combination of Development and Operations, see [Jab+16; Wie+19]
  - Aiming for rapid software release cycles with high degree of automation and stability
- Variant based on Git is called GitOps, see [Lim18]
  - Self-service IT with proposals in pull requests (PRs)
  - Infrastructure as Code (IaC)

6.3 Concluding Questions

- What did you find difficult or confusing about the **contents** of the presentation? Please be as specific as possible. For example, you could describe your current understanding (which might allow us to identify misunderstandings), ask questions in a Learnweb forum that allow us to help you, or suggest improvements (maybe on GitLab). Most questions turn out to be of general interest; please do not hesitate to ask and answer in the forum. If you created additional original content that might help others (e.g., a new exercise, an experiment, explanations concerning relationships with different courses, ...), please share.

Bibliography

[Jab+16] Ramtin Jabbari et al. "What is DevOps? A Systematic Mapping Study on Definitions and Practices". In: *Proceedings of the Scientific Workshop Proceedings of XP2016*. XP '16 Workshops. 2016. DOI: 10.1145/2962695.2962707. URL: https://doi.org/10.1145/2962695.2962707.

[Lim18] Thomas A. Limoncelli. "GitOps: A Path to More Self-Service IT". In: *Commun. ACM* 61.9 (2018), pp. 38–42. DOI: 10.1145/3233241. URL: https://doi.org/10.1145/3233241.

[Wie+19] Anna Wiedemann et al. "Research for Practice: The DevOps Phenomenon". In: *Commun. ACM* 62.8 (2019), pp. 44–49. DOI: 10.1145/3331138. URL: https://doi.org/10.1145/3331138.

License Information