

Git Introduction *

Jens Lechtenbörger

VM OER 2020/2021

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?

*This PDF document is an inferior version of an OER [HTML page](#); [free/libre Org](#) mode source repository.

```

Commits in master touching lisp/gnus/mml-sec.el
f02ce3b * ; Add fixe comments re passwor Glenn Morris 2 weeks
d3437ea * Replace some obsolete functions Glenn Morris 3 weeks
5c7dd8a * Update copyright year to 2018 Paul Eggert 3 months
bc511a6 * Prefer HTTPS to FTP and HTTP in Paul Eggert 6 months
bcf244e * Merge from origin/emacs-25 Paul Eggert 1 year

5badc81 * Update copyright year to 2017 Paul Eggert 1 year
37b9099 * - Paul Eggert 37 years

56df617 * Address compilation warnings Glenn Morris 2 years
f3cdf9c * Remove compat code from some Lars Ingebrigtsen 2 years
9efc29a * Remove several gnus-util comp Lars Ingebrigtsen 2 years
f466bf3 * Remove the gnus-union alias Lars Ingebrigtsen 2 years
46ef01f * Fix encoding problem introdu Lars Ingebrigtsen 2 years
93c3363 * Fix epg-related compilation v Lars Ingebrigtsen 2 years
37cf445 * Remove XEmacs compat function Lars Ingebrigtsen 2 years

2e239c * Compare recipient and keys case David Edmondson 2 years
e85e0d5 * Add some missing version tags Glenn Morris 2 years
5213ded * Refactor mml-smime.el, mml1991 Jens Lechtenboerg... 2 years
U:%- magit-log: emacs Top (19,0) (Magit Log -1)

Commit emacs-25.0.91-48 limited to file lisp/gnus/mml-sec.el
Compare recipient and keys case-insensitively

* lisp/gnus/mml2015.el: (mml-secure-check-user-id): When comparing a
recipient address with that from a key, do so in a case insensitive
manner (bug#22603).

1 file changed, 4 insertions(+), 4 deletions(-)
lisp/gnus/mml-sec.el | 8 ++++----

modified lisp/gnus/mml-sec.el
@@ -655,10 +655,10 @@ mml-secure-check-user-id
 (catch 'break
  (dolist (uid uids nil)
   (if (and (stringp (epg-user-id-string uid))
            (equal (car (mail-header-parse-address
                      (epg-user-id-string uid)))
                  (car (mail-header-parse-address
                      recipient))))
       (equal (downcase (car (mail-header-parse-address
                            (epg-user-id-string uid))))
              (downcase (car (mail-header-parse-address
                            recipient))))
       nil))))
U:%- magit-revision: emacs 31% (21,0) (Magit Rev -1)

```

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, CodiMD, 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

- 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

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

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

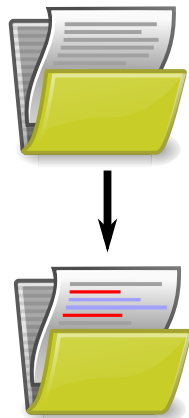


Figure 5: “Folder” under CC0 1.0; derived from Pixabay

- 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)

2.3 Key Terms: Branch, Merge

- **Branches**

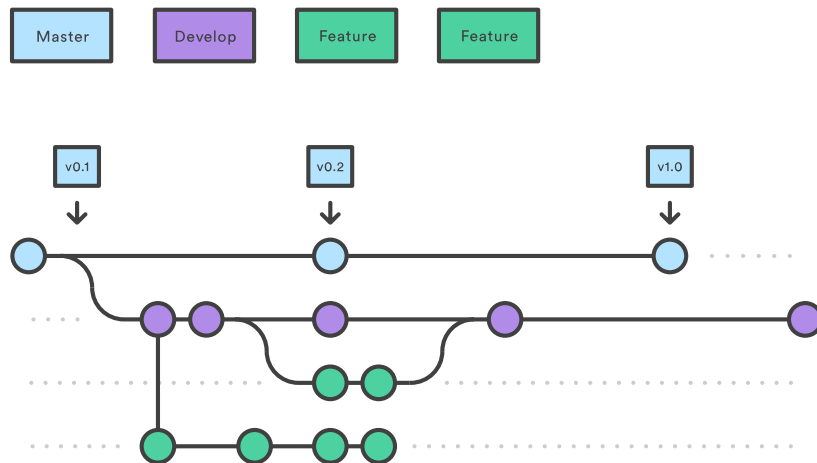


Figure 6: “Git Branches” by Atlassian under CC BY 2.5 Australia; dimension attributes added, from Atlassian

- 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.)

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](#)
- Really, please install Git now
- You may use Git without a server
 - Run `git init` in any directory
 - * Keep track of your own files
 - By default, you work on the `master` branch
 - * `master` is not more special than any other branch you may create

3.3 Git with Remote Repositories

- **Download** files from public repository: `clone`
 - `git clone https://gitlab.com/oer/cs/programming.git`
 - * Later on, `git pull` merges changes to bring your copy up to date
- **Contribute** to remote repository
 - Two variants described later on

3.3.1 A quick check

3.4 First Steps with Git

- Prerequisites (see [quickstart](#))
 - 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 (`master`) 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 exists
 - * 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
 - Execute `git checkout -b testbranch`
 - * Option `-b`: Create new branch and switch to it
 - Execute `git status` and find yourself on new branch
 - * With uncommitted modifications from `master`
 - * Change more, commit on branch
 - * Later on, merge or rebase with `master`
 - Execute `git checkout master` and `git checkout testbranch` to switch branches

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 contribution instructions first
 - E.g., Feature Branch Workflow
 - * Fork upstream project (in GUI)
 - Which creates **your own** project
 - * Clone it, full permissions on `origin`
 - * Create branch for your improvements, commit, push branch to `origin`
 - * In GUI, open merge request (GitLab) or pull request (GitHub) for branch
 - If accepted, its changes usually go into upstream's `master`

3.4.10 Remotes (3)

- If your merge request was merged upstream, maybe update your fork to mirror upstream' 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 and integrate state of `upstream`:
 - `git fetch upstream`
 - Integrate its `master` into your `master`, maybe with rebase:
 - `git checkout master`
 - `git rebase upstream/master`
 - * Push updated `master` to your fork: `git push`

3.5.2 Merge vs Rebase (2)

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

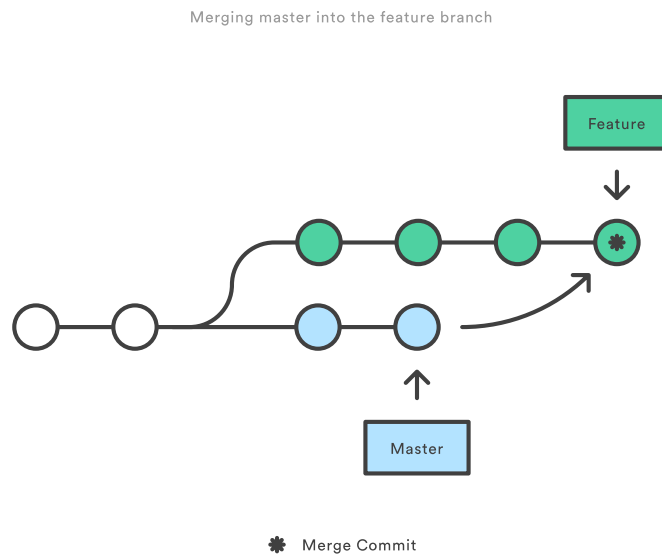


Figure 8: “Merging” by Atlassian under CC BY 2.5 Australia; from Atlassian

3.5.3 Merge vs Rebase (3)

- Rebase rewrites **feature** branch on **master**
 - Applies commits of **feature** on **master**
 - Cleaner end result, but branch’s history lost/changed

Rebasing the feature branch onto master

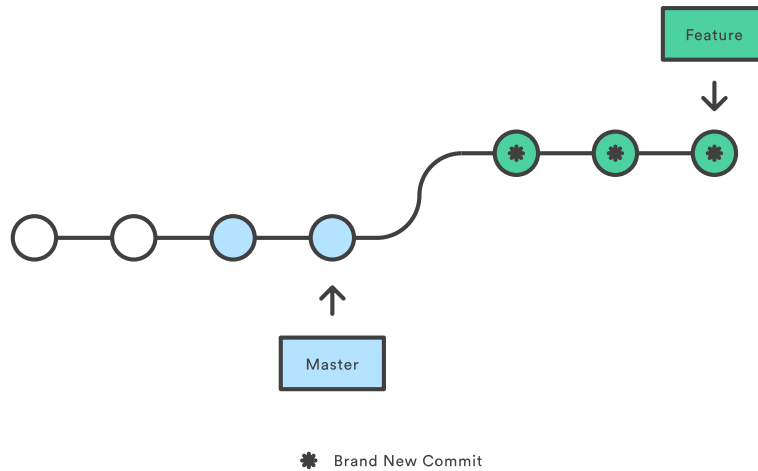


Figure 9: “Rebasing” by Atlassian under CC BY 2.5 Australia; from Atlassian

3.6 Git Workflows

- For collaboration, team needs to agree on **git workflow**
 - Several alternatives exist
- Feature Branch Workflow may be your starting point
 - Clone repository (maybe your forked version)
 - * You need a repository to which you are allowed to push commits
 - Create separate branch for **each** independent contribution
 - * E.g., bug fix, new feature, improved documentation
 - * Enables independent work
 - Once done, push that branch, create pull/merge request, receive feedback
 - * **Pull/Merge request**: special action asking maintainer to include your changes
 - * Maintainer may merge branch into **master**

3.6.1 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

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 [JbA+16; WFW+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)

Bibliography

License Information

This document is part of a larger course. Source code and source files are available on [GitLab](#) under free licenses.

Except where otherwise noted, the work “Git Introduction”, © 2018-2020 Jens Lechtenbörger, is published under the Creative Commons license CC BY-SA 4.0.