

Git Tutorial

Dr. Steffen Stärz



McGill
UNIVERSITY



GitLab

Part I: Git Basics

Part II: Git Rebase

Part III: Beyond Git: GitLab/GitHub/...

16 May 2023

(GitLab Enterprise Edition 15.0.4-ee)

About the presenter

- 2015: Dr. rer. nat. (PhD), Technische Universität Dresden, Germany
 - "Energy Reconstruction and high-speed Data Transmission with FPGAs for the Upgrade of the ATLAS Liquid Argon Calorimeter at LHC"
 - 2015-2018: Applied Physicist Fellow at CERN in the EP-ADE-CA group, LAr calorimeter
 - LAr Online Software Coordinator
 - Coordinator of the "Demonstrator" group (pre Phase-I Upgrade)
 - 2019-now: (Research → Academic) Associate in Department of Physics at McGill University
 - Former Firmware Coordinator and now Lead Firmware Technical Manager for the LASP board (ATLAS LAr Phase-II Upgrade)
- ⇒ ATLAS Liquid Argon Calorimeter, Software (C++), firmware (VHDL), **git**, **GitLab**, **CI/CD**, ...

Plan of the Day

- 1 Git Basics
 - Concept
 - Novice
 - Beginner
 - Competent
 - Expert
- 2 Git Rebase
 - The rebasing problem
 - Git rebase features
- 3 Beyond Git: GitLab/GitHub/...

Remark for this tutorial: Links are lowlighted in black.

Part I: Git Basics

Disclaimer: Git is awesome!

This tutorial cannot cover all aspects and is hence completely incomplete!

- Only basic commands with few options are presented here.

Git comes with its help - read it!

- From shell: `git help [subcommand]` (or `man git [subcommand]`)
- Official Git Tutorial
- Git Cheat Sheet (exhaustive but by definition incomplete list)

This (part of the) tutorial: Git commands

- 1 Concept: The general idea of git
- 2 Novice: Git setup
- 3 Beginner: Basic Git commands
- 4 Competent: Advanced Git commands
- 5 Expert: Fancy Git commands



Follow this tutorial with the sandbox Git repository to gain experience

Git: Concept

What is Git!?

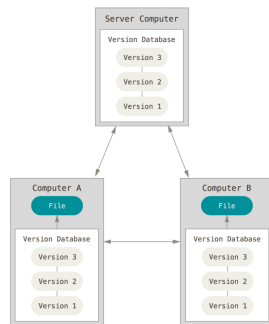
Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. It allows to

- revert selected files back to a previous state,
- revert the entire project back to a previous state,
- compare changes over time,
- see who last modified something that might be causing a problem,
- see who introduced an issue and when, ...

Git is a **Distributed** Version Control System:

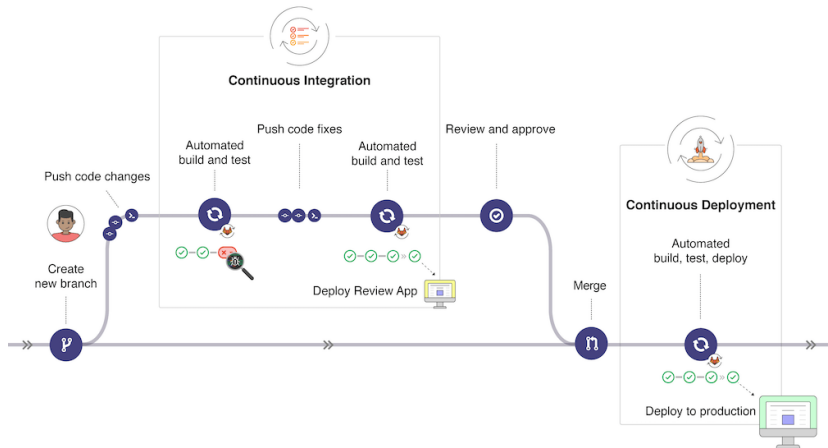
- Clients don't just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history.
- ⇒ Every clone is a full backup of all the data: client can work even if server is down.

This tutorial will explicitly not go into **forks**.



Git workflow¹: The suggested work flow for any project

This tutorial will **not** cover CI/CD, but to give you a big picture:



Stable master → development on temporary branch → new stable master

¹From the official GitLab documentation

Git: Novice




<http://www.xkcd.com/1597/>

Git Setup: Remote profile

Make yourself known to the remote repository (e.g. GitHub)

Identification via SSH keys: Add your public key

- ❶ Login to the repo (e.g. GitHub) with your account
- ❷ Add your SSH keys in your profile^a
 - ❶ Check your key: `$ ls ~/.ssh/id_rsa.pub`
⇒ If not existing, create key: `$ ssh-keygenb`
`$ cat ~/.ssh/id_rsa.pub`
⇒ long text, contains in the end sth. like `username@host`
 - ❷ Avatar → Settings → SSH Keys: copy & paste into "Key" → Add key
⇒ Redo for all machines you'll be working from
- ❸ Add yourself an avatar 
⇒ It's not only nice, but also allows you to spot profile misconfiguration

^aSSH is default for some projects, so make sure you set this up properly.

^bDo **not** set a passphrase, otherwise each `git push` and `git pull` will be a pain!

Git Setup: Local profile

Once set user name and email address

git config

```
$ git config --global user.name yourusername
```

```
$ git config --global user.email you@cern.ch
```

⇒ Do for all machines you'll be working from

- Make sure it's **absolutely identical** on all your machines!
- Settings get written into your `~/.gitconfig` file

Git config and style up

Customize Git text highlighting!

```
$ git config --global color.ui true
```

```
$ git config --global color.status.header yellow
```

- ... or edit `~/.gitconfig` directly, e.g.:

```
[color]
__ui__=true
__status__=true
[color "status"]
__header__=yellow
__added__=green
__updated__=cyan
__changed__=red_blink
__untracked__=magenta_bold
__branch__=cyan
__nobranch__=black
[color "branch"]
__meta__=white_bold
[user]
__name__=Steffen_Staerz
__email__=steffen.staerz@cern.ch
[core]
__editor__=vim
__pager__=less-x4
```

- And many more settings, see [official documentation](#)

Git: Beginner

Get a Git repository

A) Cloning an existing (remote) Git repository

git clone

Cloning^a the sandbox^b:

```
$ git clone ssh://git@github.com/staerz/sandbox.gitc
```

```
$ git clone <repo>: clone existing Git repository
```

```
$ git clone <repo> <dir>: ... into directory
```

```
$ git clone --recursive <repo>: ... with all sub-modules
```

^aGiven you have rights to do so

^bOnce you have that, go to slide 46 and play with the sandbox!

^c For some reason GitHub suggests ':' in the url for SSH which is wrong!

B) Creating (local) Git repository

git init

To create a new Git repository in the (existing local) directory `project`:

```
$ cd <project>
```

```
$ git init
```

Basic Commands


See what has changed

git status

\$ git status: shows everything

\$ git status -uno: only status of files tracked by Git

\$ git ls-files: list files tracked by Git in the current directory

 Make gitst an alias for git status -uno

Adding files to index (staging area) (not yet committing!)

git add


Before any commit, first tell Git which file to **stage for commit** via add:

\$ git add .: add **all** files in the current directory^a

\$ git add file1 [file2 ...]: add file[s] explicitly

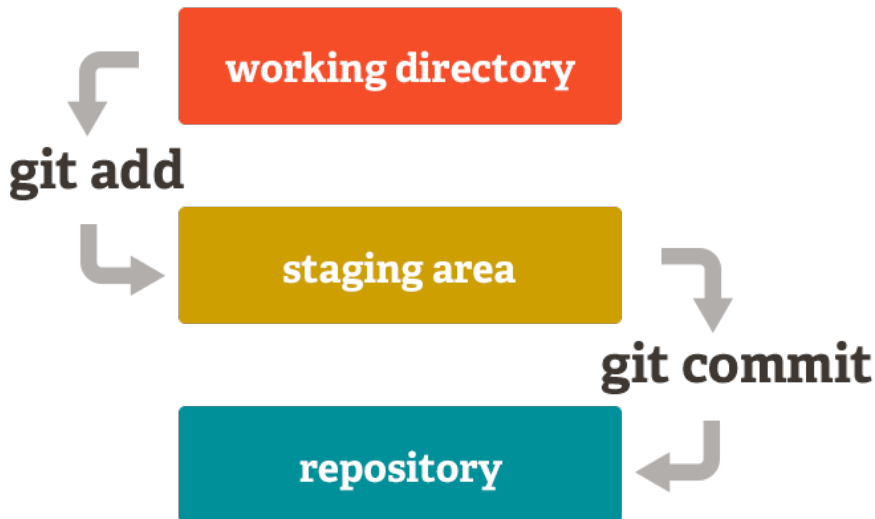
\$ git add -u: add all updated (= modified) files

 Use git status before and after git add to see effect

^a  Usually not desired unless the .gitignore file is set up properly!

Basic Commands 2

What just happened: we're half way!



Basic Commands 3

Tracking file modifications

git diff

At any time, browse the differences using

```
$ git diff: show changes of all tracked files
$ git diff filename: show changes of file filename
$ git diff --cached: show changes of all staged files
$ git diff --cached filename: imagine what
$ git diff --word-diff: show more compact diff
```



Make `git dif` an alias for `git diff --ignore-submodules --word-diff`

Reverting the current modifications

git checkout

Undo the last changes to a file (or more) **since the last** `git add`

```
$ git checkout -- filename: Restore only filename
$ git checkout -- .: Restore all files in the current directory
```

Basic Commands 4

Unstaging files (= "un-add")

git reset

Remove files from the index via

```
$ git reset HEAD filename: remove only filename
```

Since Git 2.23.0, there is a new alternative Git command for that:

Unstaging files (= "un-add")

git restore

Remove files from the index via

```
$ git restore --staged filename: remove only filename
```

Reverting the current modifications

git restore

restore is also an alternative to checkout -- (previous slide)

```
$ git restore filename: Revert only filename
```

GIT-RESTORE(1)

Git

GIT-RESTORE(1)

...

THIS COMMAND IS EXPERIMENTAL. THE BEHAVIOR MAY CHANGE.

Basic Commands 5

Committing files

git commit

Once staged for commit (added to the index), commit changes via

```
$ git commit: commit, editor (e.g. vima) opens to enter message
```

```
$ git commit --amend: "update" your last commit (message)
```

- Very use- and powerful: You forgot sth. in your commit or just spot a typo (in actual code or commit message)?

⇒ E.g. modify culprit file, `git add` it again, then "re-commit"

⇒ **Replaces** last commit^b

```
$ git commit -m "message": directly enter commit message
```

^aVim is the default, but can be changed via the `~/.gitconfig` file

^b`git push` in the meantime will result in a conflict requiring a forced push to remote (see later)

Basic Commands 5.5



	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

<http://www.xkcd.com/1296>

In general **avoid that** and rather:

- **Use meaningful commit messages:** avoid any information which is obvious from the commit itself like "Me: Updated file xyz"
- Follow commit message conventions by the project (if any)
 - E.g.: mention the associated ticket in the commit:
"#123: Document usage of ... in README.md"

Basic Commands 6

Browsing the history

git log

```
$ git log: detailed commit log: who, when, what (message)
$ git log --oneline: single line per commit
$ git log --graph: "GUI": graph with branching info
$ git log filename: Only commits where filename was modified
$ git log --author="Jon": imagine what
$ git log -3 (or -n 3): show last 3 commits
```



Make gitlog an alias for `git log --oneline --graph`

Showing actual modifications

git show

```
$ git show <commit>: show detailed modifications of that commit
$ git show <commit> --word-diff=color: inline highlighting
$ git show <commit> --name-status: affected files only
```

Git: Competent

Prevent certain files to be "git added" (Blacklisting)

Set file exclusions for tracking via `.gitignore` file (per repo), e.g. latex:

```
# Specifies intentionally untracked files for this repository
# Find documentation (syntax) here: https://git-scm.com/docs/gitignore
#
# Files already tracked by Git are not affected
#####

# don't check in any of the temporarily created latex files:
*-blx.bib
*.pdf
# and so on ...
# nor any hidden file (e.g. *.swp file created by editors)
.*
# also don't check in any backup files people might have created
*.backup
*.bak
*.orig
*~
*.synctex.gz

# don't check in any file in shared
shared/*
# except the figures directory
!shared/figures/
# except the style files
!shared/style/
# and explicitly allow pdf's (since actually excluded via list on top!)
!shared/figures/*.pdf
```


Prevent certain files to be "git added" (Whitelisting)

Set file exclusions for tracking via `.gitignore` file (per repo), e.g. latex:

```
# Specifies intentionally untracked files for this repository
# Find documentation (syntax) here: https://git-scm.com/docs/gitignore
#
# White-listing approach: forbid everything, only allow explicitly
#
# We aim not to cover all cases, things can always be added by "-f" manually;
# and files already tracked by Git are not affected anyway.
#####

# First, ignore everything
*
# Now, white-list anything that's a directory
!*/
# and all the file types we're interested in:

# Markdown files
!*.md

# source files
!*.tex

#temporary source files are forbidden again
!*.tex

# allow figures directory
!shared/figures/
```

Whitelisting is recommended as it is more restrictive than blacklisting.

Advanced Commands

Rename or move tracked files

git mv

```
$ git mv <file> <newfilename>: Rename file
```

```
$ git mv <file> <directory>: Move file to directory
```

Removing tracked files

git rm

```
$ git rm <file>: Remove file from repo and local file system
```

```
$ git rm --cached <file>: Remove file only from repo
```

```
$ git rm -r <dir>: Remove directory from repo and file system
```

Reset to some previous state

git reset

```
$ git reset <file>: opposite of git add
```

```
$ git reset <commit>: reset to that commit (unstage changes)
```

```
$ git reset --soft <commit>: "un-commit" (stage changes)
```

```
$ git reset --hard <commit>: reset and discard changes
```

Branches

The idea of branches

Changes (even adding new files or the deletion of existing ones) in branches are completely transparent to each other until branches are merged. This allows parallel development of different features on a first come first serve basis.

Git branches

- Default branch is `master` (⚠️ or `main`^{ab})
 - `Main` branch: should be `always operational`
- Developments to be done in dedicated `development branches`

^aFor repos created on GitLab 13.11++ (unless explicitly configured differently by the instance admin) or GitHub since 1st Oct 2020

^bThis tutorial will go with `master`

Branches 2

Changing branches

git checkout

- \$ `git checkout -b devbranch`: create branch devbranch from current branch (master or other: branching branches is allowed)
- \$ `git checkout targetbranch`: switch branches (only possible if uncommitted changes don't conflict targetbranch: Git complains)

Since Git 2.23.0, there is a new alternative Git command for that:

Changing branches

git switch

- \$ `git switch -c devbranch`: create branch devbranch from current branch (master or other: branching branches is allowed)
- \$ `git switch targetbranch`: switch branches (only possible if uncommitted changes don't conflict targetbranch: Git complains)

Branches 3

Working on devbranch and master was updated in the meantime:
How to include these updates into devbranch?

a) Merging branches

git merge

```
$ git checkout devbranch: switch to devbranch (possibly  
currently active)  
$ git fetch origin master:master: update local master  
$ git merge master: merge master to current branch
```

- General idea: Keep developments independent!
 - ⇒ While working on devbranch, any changes to other branches are "invisible": Nobody affects your (local) branch
 - ⇒ If important update pops in, include it at your convenience
 - ⇒ **Avoid merging with other parallel dev branches (keep topology simple!)**
 - ⇒ **Merge from master regularly** to spot issues early and minimise chances of divergent developments (Git nicely handles conflicts, but try to avoid)

Branches 3

Working on devbranch and master was updated in the meantime:
How to include these updates into devbranch?

b) Rebasing branches (Applied for LASP FW)

git rebase

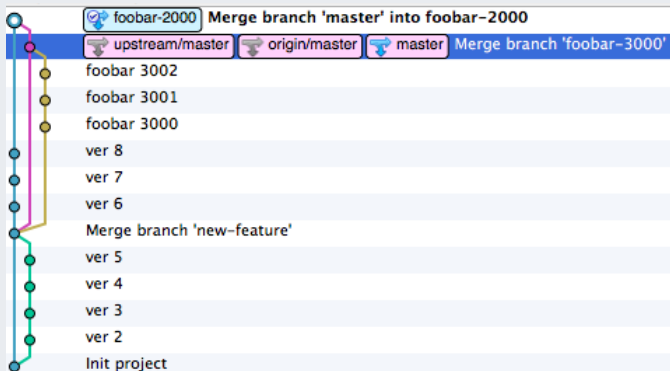
```
$ git checkout devbranch: switch to devbranch (possibly  
currently active)  
$ git fetch origin master:master: update local master  
$ git rebase master: replay devbranch onto updated master
```

- General idea: Keep developments independent!
 - ⇒ While working on devbranch, any changes to other branches are "invisible": Nobody affects your (local) branch
 - ⇒ If important update pops in, include it at your convenience
 - ⇒ **Avoid merging with other parallel dev branches (keep topology simple!)**
 - ⇒ **Rebase** onto master to sequentialise developments (Git will handle conflicts in case, just like when merging)

Branches 4

Least advisable branch workflow

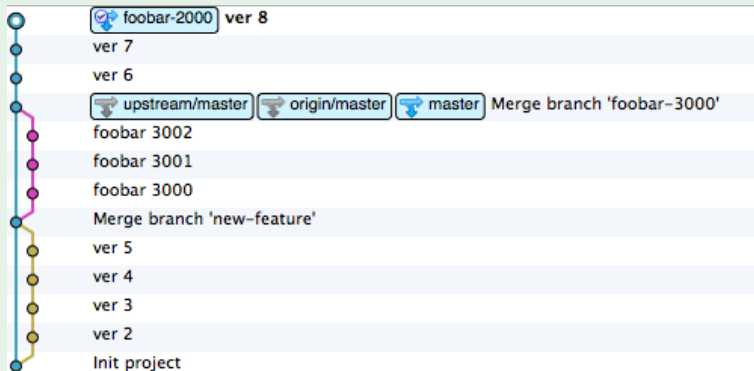
- Merge master back to development branch prior to its merge



Branches 4

Branch workflow with semi-linear history

- Rebase to master prior to its merge



More pictures and explanation here, see also the [Git Rebase Part \(Part II\)](#).

 General → Merge requests → Merge method

Branches 4.5

Branch names

Branches can be named almost anything, only some very not standard patterns are forbidden, e.g. ' .' (see **git check-ref-format**).

Branch naming: good practices

Good practice: Not problem but **solution oriented branch name!** E.g.:

- fix-update-script
- implement-feature-xyz

⇒ letters and dash as word separator

Branch name from issue title: conventions

See later: When working with GitLab/GitHub/... the branch name is generated from the issue title automatically

⇒ Apply good practice already for issue title

Working with a remote Git repository

Private (local) Git repository

Up to here, changes (in all branches) were **local** only.
Locally there are no restrictions:

- Commit to `master` branch is allowed
- There are no conventions to follow except your own

When working in a project with others: Conventions and Restrictions!

- Best practice
 - ✓ (Remote) `master` protected, only project maintainers can merge to it
 - ⇒ NEVER directly commit to `master`!
- Good practice
 - Branches follow workflow: Development → Verification → Master
 - ⇒ Handle large projects with many developers
 - ⇒ Many more complicated workflows possible, of course

Working with a remote Git repository

Private (local) Git repository

Up to here, changes (in all branches) were **local** only.
Locally there are no restrictions:

- Commit to `master` branch is allowed
- There are no conventions to follow except your own

When working in a project with others: **Conventions** and **Restrictions!**

- **Best practice**
 - ✓ (Remote) `master` protected, only project maintainers can merge to it
⇒ **NEVER directly commit to `master`!**
- **Good practice**
 - Branches follow workflow: Development → Verification → Master
⇒ Handle large projects with many developers
⇒ Many more complicated workflows possible, of course

Communicating with remote

Get remote changes into local Git repository

git pull

```
$ git pull: fetch remote changes and merge with local brancha  
$ git pull --all: pull for all local branches  
$ git pull origin master:master:  
pull master without checkout of master while on another branch
```

^aNote that `git pull` is a `git fetch` followed by a `git merge`:
fetching updates the local information about the remote repository and
mergeing finally applies these remote changes to the local branch

Get local changes into remote Git repository

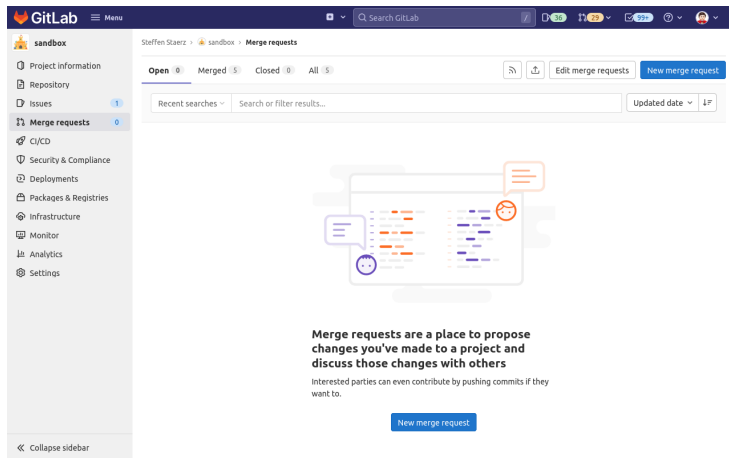
git push

```
$ git push -u origin <branch>: initial push of branch  
$ git push: push (new commits) to remotea (branch already exists)
```

^a configure `git push` to only push current branch: `git config --global push.default current`

Merging to **protected** remote master

Best practice: Merging to the (protected) master branch is done via the web interface via a **Merge Request** (GitLab) [or "Pull request" on GitHub]:



But see the second part of this tutorial for the full work flow...

Tags

The idea of tags

Tags are human-readable labels (pointers) attributed to selected commits. These commits (and hence tags) identify a given project^a status, usually an **operational version** with a well defined list of features.

^aIt is up to each project to define a dedicate tagging convention. Using **semantic versioning** is recommended.

Working with tags

git tag

```
$ git tag <tagname>: create new simple tag
$ git tag -a <tagname>: create annotated tag
$ git tag --list: list all local tags
$ git push origin <tagname>: push tag to remote
$ git ls-remote --tags: imagine what
$ git checkout <tagname> -b <branchname>: imagine
```

Git: Expert

Some random fancy stuff

I'm stuck in my development: Can you check it on <mybranch>?

Checkout any remote branch to a new local branch

```
$ git checkout -b <branch name> origin/<mybranch>
```

There is a stupid typo in my branch name!?

Rename branch

git branch

```
$ git branch -m <newbranchname>
```

I've done stupid things on my branch, time to delete it!

Delete branch locally and on remote

```
$ git branch -d <branchname>
```

```
$ git push origin --delete <branchname>
```


Some more random fancy stuff

I changed local commits (e.g. via `--amend` or `rebase`):

Force remote to overwrite its branch

```
$ git push -fa
```

^a ⚠ Remote changes (esp. remote-only commits) can be **lost**! Do a `git fetch` first to not loose anything!

My browser is dead - what are the current remote branches?

List (locally known) remote branches

```
$ git branch -r
```

I've done stupid things to local master, everything is messed up! Help!

Drop local changes and reset to remote master

git fetch

```
$ git fetch --all
```

```
$ git reset --hard origin/master
```

Some particularly fancy stuff

I need to undo a commit that was done long time ago!

Revert commit

git revert

```
$ git revert <commit>a
```


^aA new (inverse) commit is created! Only works if current status is compatible with these changes.

All these branches don't exist anymore remotely, get rid of them!

Delete all stale remote-tracking branches

git remote

```
$ git remote prune origina
```

^a Make `grpo` an alias for `git remote prune origin` (it's very unlikely that you'd have another remote repo)

I really need to pick this one commit, I need it, now!

Apply a single commit to the current branch

git cherry-pick

```
$ git cherry-pick <commit>
```

One more word on `remote`

Recall `origin` from `git pull` and `git push`?

`origin` is the **default** name of the remote repository.

→ A repository can have **multiple** remotes, all distinguished by different names!

Work with multiple remotes

`git remote`

```
$ git remote add <name> <URL>: Add a(nother) remote repoa
```

```
$ git remote -v: List remotes (names and URLs)
```

```
$ git remote remove <name>: Guess what
```

^aNote that `<URL>` can also be a local path, i.e. a some directory on your local machine → helpful when trying to check if all files needed are committed

Only when you have multiple remotes, `git push` and `git pull` need to explicitly know the name of it

Intermediate work

Stash ongoing work

git stash

```
$ git stash: Stash all current changes  
$ git stash pop: Retrieve latest stash entry  
$ git stash list: List all stash entries
```

Note that the stash is distinct from the staging area.

It's meant for the case when you're currently working on something, are not yet ready to commit, but suddenly (→ "hotfix") need to switch to a different topic for a while to come back **soon** after.



Let `git stash` be the exception to your work flow!



Possibly better make a "quick and dirty" commit and **fix (amend)** **that commit** once coming back to it.

Submodules


A repository can contain other repositories!

- Useful to separate big project into smaller ones
⇒ Distributes developers over (semi-independent) repositories
- `.gitmodules` stores information of submodules (URLs, paths)

Work with submodules

git submodule

```
$ git submodule: List all submodules  
$ git submodule add <repository>: Add a new submodule  
$ git submodule update: Update to committed state  
$ git submodule foreach: Do shell command in each submodule
```

- `git pull` on a repo with submodules also fetches all submodules²
- Option `--recursive` becomes important for some git commands
- Once `git submodule add`, submodules are like files (`git add`)
-  Be vigilant to know which repo you actually commit to!

²Possibly subject to configuration

Git Pro Tip

Configure bash to show the current branch

Modify your `~/ .bashrc` to contain the following:

```
parse_git_branch() {
    which git &>/dev/null && git rev-parse --abbrev-ref HEAD 2>
    /dev/null | sed -e 's/./ / (&)/'
}
export PS1="[u@h \[\033[32m\]\W\[\033[33m\]\$(
    parse_git_branch)\[\033[0m\]]$ "
```

to print the branch name (if in a repo) directly:

```
[staerz@staerz-dell5820 sandbox (master)] $ _
```

Colours are your choice, of course:

- 33 is yellow, 32 is green
- See more here...

Recap: Git Commands

Summary Part I: Git Basics

Commands that will be used regularly (and will be known by heart)

Everyday life commands

```
$ git checkout (or maybe git switch and git restore)
```

```
$ git status (💡 or rather git st)
```

```
$ git add
```

```
$ git diff
```

```
$ git commit
```

```
$ git log (💡 or rather git log)
```

```
$ git pull
```

```
$ git push
```


Checking out the sandbox - step by step

- 1 Make sure to have (developer) access to the sandbox Git repository
 - I.e. create an account for GitHub

- 2 Once you have set up Git, get a local copy of the sandbox

```
$ cd <some local directory>
$ git clone ssh://git@github.com/staerz/sandbox.git
$ cd sandbox
```

- 3 Apply Git commands as they are introduced in this tutorial, modifying files, adding new files, fixing typos in the BrothersGrimm.txt, ...

```
$ git status -uno
$ git checkout -b yourdevbranch
$ git add newfile
$ git commit -m "Fancy message"
$ git push -u origin yourdevbranch
```



Remember the Git Cheat Sheet and print a personal copy of it!

Part II: Git Rebase

The rebasing problem

git merge vs. git rebase

Disclaimer: It's done "wrong" in so many places!

Let's settle a long argument and misconception!^a

^aFor git there is no right or wrong.

A seamless work flow **without any merge conflicts ever, guaranteed**

When do you merge?

- When a development on a branch is finished, you merge (that branch) to master.
 - By design, there **cannot** be any merge conflicts if your branch started from master **and there is no intermediate modification to master**

What's next?

- Pick up the next development and work on it, i.e. create a branch (from the "new" master) and work until it's ready to merge.

git merge vs. git rebase 2

Source of merge conflicts when merging

Merge conflicts appear when a file modification is incompatible with **another** modification in a **parallel** branch and manifest at the time a merge of these parallel branches is attempted.

⇒ Git cannot solve that automatically, manual user intervention is needed.

Source of merge conflicts when rebasing

Merge conflicts appear when a file modification is incompatible with **another** modification in the branch rebased to and manifest at the time a rebase of the branch is attempted.

Same reason!

⇒ There is no difference between merge and rebase here: If there is a conflict, it must be solved manually.

git merge vs. git rebase 3

So why to prefer rebasing?

- Parallel developments are supposed to be independent, hence interchangeable (which branch is merged first shouldn't matter).
 - Dependant developments anyway need to be done sequentially.
- ⇒ Keep your history clean and easy to parse for a human.
- Isolate different steps of development!
- ⇒ Keep it easy to track down any potential issue brought into the code.

Repo history with git merge



Repo history with git rebase



git log can parse anything - but can you!?

That's what the tutorial said - the `git rebase` pitfall

Working on `devbranch` and `master` was updated in the meantime:
How to include these updates into `devbranch`?

b) Rebasing branches

`git rebase`

```
$ git checkout devbranch: switch to devbranch (possibly  
currently active)
```

```
$ git rebase master: replay devbranch onto updated master
```

Attention: that's the shortlist of `git` commands, see later ...

- General idea: Keep developments independent!
 - ⇒ While working on `devbranch`, any changes to other branches are "invisible": Nobody affects your (local) branch
 - ⇒ If important update pops in, include it at convenience
 - ⇒ Avoid merging with other parallel dev branches (keep topology simple!)
 - ⇒ **Rebase** onto `master` to sequentialise developments (Git will handle conflicts in any case)

To use it properly, it's important to understand ...

- 1 Rebase intention (what rebase does and intends to do)

To use it properly, it's important to understand ...

- ➊ Rebase intention (what rebase does and intends to do)
- ➋ What the remote repository (remote branches) and the local repository (local branches) really are

To use it properly, it's important to understand ...

- ❶ **Rebase intention** (what rebase does and intends to do)
 - Keep the history clean and linear
 - Avoid (or spot) dependencies early
- ❷ **What the remote repository (remote branches) and the local repository (local branches) really are**

To use it properly, it's important to understand ...

- ❶ **Rebase intention (what rebase does and intends to do)**
 - Keep the history clean and linear
 - Avoid (or spot) dependencies early
- ❷ **What the remote repository (remote branches) and the local repository (local branches) really are**
 - ❶ In pictures: repo → plant (tree), branch → branch, commit → leaf

To use it properly, it's important to understand ...

- ❶ **Rebase intention (what rebase does and intends to do)**
 - Keep the history clean and linear
 - Avoid (or spot) dependencies early
- ❷ **What the remote repository (remote branches) and the local repository (local branches) really are**
 - ❶ In pictures: repo → plant (tree), branch → branch, commit → leaf
 - ❷ The remote repository
 - It's the plant in the store, it's not for sale and a gardener takes care of it.
 - The plant has a root (initial commit), a stem (master) and other branches.
 - Taking photos of the plant and taking them home is allowed at any time.

To use it properly, it's important to understand ...

- ❶ **Rebase intention (what rebase does and intends to do)**
 - Keep the history clean and linear
 - Avoid (or spot) dependencies early
- ❷ **What the remote repository (remote branches) and the local repository (local branches) really are**
 - ❶ In pictures: repo → plant (tree), branch → branch, commit → leaf
 - ❷ The remote repository
 - It's the plant in the store, it's not for sale and a gardener takes care of it.
 - The plant has a root (initial commit), a stem (master) and other branches.
 - Taking photos of the plant and taking them home is allowed at any time.
 - ❸ The local repository
 - It's the home recreation (clone) of that plant based on that photo
 - At home, **any** modification (commits) to the plant is allowed: growing new leafs on a branch, growing new branches, transplanting branches, cutting (= deleting) branches and leafs, really anything, even killing the plant

To use it properly, it's important to understand ...

- ❶ **Rebase intention** (what rebase does and intends to do)
 - Keep the history clean and linear
 - Avoid (or spot) dependencies early
- ❷ **What the remote repository (remote branches) and the local repository (local branches) really are**
 - ❶ In pictures: repo → plant (tree), branch → branch, commit → leaf
 - ❷ The remote repository
 - It's the plant in the store, it's not for sale and a gardener takes care of it.
 - The plant has a root (initial commit), a stem (master) and other branches.
 - Taking photos of the plant and taking them home is allowed at any time.
 - ❸ The local repository
 - It's the home recreation (clone) of that plant based on that photo
 - At home, **any** modification (commits) to the plant is allowed: growing new leafs on a branch, growing new branches, transplanting branches, cutting (= deleting) branches and leafs, really anything, even killing the plant
 - ❹ Linking them together
 - The plant will grow in the store. Go there to take a new photo: `git fetch`
 - The plant at home grows, tell the gardener about it: `git push`
 - Details apply (see next slides)

An example git history: remote vs. local

The remote repository

GitLab Projects Groups Activity Milestones Snippets Search or jump to...

atlas-lar-be-firmware > LASP > LASP-doc > Graph

97-add-rule-line-fo... Note the selection of the branch to reorder the view

You can move around the graph by using the arrow keys. master is somewhere here

Git revision Search Begin with the selected commit We look for 97-xxx now

97-add-rule-line-fo... bib: adding SX and MX package description refs

Make the temporary chapter take number 0 to have the right numbers once it's

Adding details on threads (discussions), pdf page 86 to 91 and 96, 97

functional diagram: moving from v2.1 to v2.2

generating v2.1 functional diagram

performing the geographical loaction of the firefly devices int the function

indicating link numbers

applying some modifications to the LASP documents

adding additional documents for LASP

applying some modifications to the LASP documents

adding additional documents for LASP

Note this reference: It's the initial origin of 97-xxx

bib: Fixing 3 entries (missed comma)

Adding separation lines between template remarks and content, updating FW bl

33-fw-spec-defa... Merge branch '96-populate-chapter-physical-description' into 'master'

Filling chapter Physical Description: basically use what was in Manufacturei

An example git history: remote vs. local

The local repository

```
[staerz@staerz-dellt5820 doc (97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from
-actual-content)]$ gitlog
* f9f911a (HEAD -> 97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-con
tent, origin/97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content)
Make the temporary chapter take number 0 to have the right numbers once it's removed
* cd9f814 bib: Fixing 3 entries (missed comma)
* 74cd623 Adding separation lines between template remarks and content, updating FW block diagram
to v2.5
* 797885b (origin/master, origin/HEAD, origin/33-fw-spec-define-ttc-receiver-2, master) Merge br
anch '96-populate-chapter-physical-description' into 'master'
| \
| * ffb75b3 (origin/96-populate-chapter-physical-description) Filling chapter Physical Description
: basically use what was in Manufacturer
| * 5185ca0 bib: adding SX and MX package description refs
| /
* 9464e81 Merge branch '80-populate-chapter-interfaces' into 'master'
```

An example git history: remote vs. local

The local repository

```
[staerz@staerz-dellt5820 doc (97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content)]$ gitlog
* f9f911a (HEAD -> 97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content, origin/97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content)
  Make the temporary chapter take number 0 to have the right numbers once it's removed
* cd9f814 bib: Fixing 3 entries (missed comma)
* 74cd623 Adding separation lines between template remarks and content, updating FW block diagram to v2.5
* 797885b origin/master origin/HEAD origin/33-fw-spec-define-ttc-receiver-2, master Merge branch '96-populate-chapter-physical-description' into 'master'
  |
  | * ffb75b3 (origin/96-populate-chapter-physical-description) Filling chapter Physical Description: basically use what was in Manufacturer
  | * 5185ca0 bib: adding SX and MX package description refs
  | /
  * 9464e81 Merge branch '80-populate-chapter-interfaces' into 'master'
```

Note the git colouring scheme for references:

`git log`

- The local repository contains **local** (green) and **remote** (red) references
- Exception: **local HEAD** is always cyan - no link with **origin/HEAD**!

Be reminded: "Remote references" **are not** necessarily the actual remote references but only those as of the last photo you took

An example git history: Updating remote references

The local repository

```
[staerz@staerz-dell:~]$ git fetch
remote: Enumerating objects: 59, done.
remote: Counting objects: 100% (59/59), done.
remote: Compressing objects: 100% (49/49), done.
remote: Total 59 (delta 26), reused 23 (delta 10)
Unpacking objects: 100% (59/59), done.
From ssh://gitlab.cern.ch:7999/atlas-lar-be-firmware/LASP/LASP-doc
  3bafdee..58bce6f  31-fw-spec-define-buffer-requirements -> origin/31-fw-spec-define-buffer-requirements
  797885b..af8387f  master -> origin/master
+ b4fa588...81cc36c  preview -> origin/preview (forced update)
```

① Take a new photo: `git fetch`

An example git history: Updating remote references

The local repository

```
[staerz@staerz-dell:~]$ git log
* f9f911a (HEAD -> 97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content)
  origin/97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content
  Make the temporary chapter take number 0 to have the right numbers once it's removed
* cd9f814 bib: Fixing 3 entries (missed comma)
* 74cd623 Adding separation lines between template remarks and content, updating FW block diagram to v2.5
* 797885b (origin/33-fw-spec-define-ttc-receiver-2, master) Merge branch '96-populate-chapter-physical-description' into 'master'
| \
|  * ffb75b3 (origin/96-populate-chapter-physical-description) Filling chapter Physical Description: basically use what was in Manufacturer
|  * 5185ca0 bib: adding SX and MX package description refs
| /
* 9464e81 Merge branch '80-populate-chapter-interfaces' into 'master'
```

- ① Take a new photo: `git fetch`
- ② See history again: `git log`
Note the updates:
 - `origin/master` and `origin/HEAD` have "disappeared"

We come back to this later ...

Working with git - in pictures

When working on the local repository, work is done on the plant at home.

- ❶ Creating a branch (from master) and doing some commits on it
 - From the upper tip of the plant (that was seen on the photo to be master), a new branch with new leafs (commits) is started.
 - ⇒ All the `git add`, `git commit`, `git log` are done on the plant at home

Working with git - in pictures

When working on the local repository, work is done on the plant at home.

- ❶ Creating a branch (from master) and doing some commits on it
 - From the upper tip of the plant (that was seen on the photo to be master), a new branch with new leafs (commits) is started.
 - ⇒ All the `git add`, `git commit`, `git log` are done on the plant at home
- ❷ Pushing back to remote
 - Memorise the new grown branch (with its root) of the plant at home and go back to the store, asking the gardener to also grow that branch there.

Working with git - in pictures

When working on the local repository, work is done on the plant at home.

- ❶ Creating a branch (from master) and doing some commits on it
 - From the upper tip of the plant (that was seen on the photo to be master), a new branch with new leafs (commits) is started.
 - ⇒ All the `git add`, `git commit`, `git log` are done on the plant at home
- ❷ Pushing back to remote
 - Memorise the new grown branch (with its root) of the plant at home and go back to the store, asking the gardener to also grow that branch there.
 - The gardener doesn't blindly do what is **asked** (`git push`)
 - ✓ He does accept entirely new branches (new names)
 - ✓ He does accept new leafs on top of branches that he has already
 - ✗ He doesn't accept new leafs on **protected branches**³ (e.g. master)
 - ✗ He doesn't accept modified leafs (which are not extensions), instead he proposes to grow a hybrid branch at home and come back later (`git pull`)

³ depending the repo setting, a push may be privileged to (a few) maintainers or entirely forbidden

Working with git - in pictures

When working on the local repository, work is done on the plant at home.

- ❶ Creating a branch (from master) and doing some commits on it
 - From the upper tip of the plant (that was seen on the photo to be master), a new branch with new leafs (commits) is started.
 - ⇒ All the `git add`, `git commit`, `git log` are done on the plant at home
- ❷ Pushing back to remote
 - Memorise the new grown branch (with its root) of the plant at home and go back to the store, asking the gardener to also grow that branch there.
 - The gardener doesn't blindly do what is **asked** (`git push`)
 - ✓ He does accept entirely new branches (new names)
 - ✓ He does accept new leafs on top of branches that he has already
 - ✗ He doesn't accept new leafs on **protected branches**³ (e.g. master)
 - ✗ He doesn't accept modified leafs (which are not extensions), instead he proposes to grow a hybrid branch at home and come back later (`git pull`)
 - For the last, it can be made **an order** (`git push -f`)
 - He **replaces his branch** in the store with it (with all its leafs and root)

³ depending the repo setting, a push may be privileged to (a few) maintainers or entirely forbidden

An example git history: The `git rebase` in action

So when it comes to a `git rebase`, what you do is

- 1 `git fetch origin master:master`: take a new photo of the plant in the shop to know what the current tip is (might be the same as on the previous photo) and apply this new master to the plant at home

```
-actual-content)]$ git fetch origin master:master
From ssh://gitlab.cern.ch:7999/atlas-lar-be-firmware/LASP/LASP-doc
 797885b..af8387f  master      -> master
```

An example git history: The `git rebase` in action

So when it comes to a `git rebase`, what you do is

- 1 `git fetch origin master:master:` take a new photo of the plant in the shop to know what the current tip is (might be the same as on the previous photo) and `apply` this new master to the plant at home

Note the updated history again: now also the `local master` has disappeared

```
[staerz@staerz-dell:5820 doc (97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content)]$ gitlog
* f9f911a (HEAD -> 97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content, origin/97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content)
Make the temporary chapter take number 0 to have the right numbers once it's removed
* cd9f814 bib: Fixing 3 entries (missed comma)
* 74cd623 Adding separation lines between template remarks and content, updating FW block diagram to v2.5
* 797885b (origin/33-fw-spec-define-ttc-receiver-2) Merge branch '96-populate-chapter-physical-description' into 'master'
| \
| * ffb75b3 (origin/96-populate-chapter-physical-description) Filling chapter Physical Description : basically use what was in Manufacturer
| * 5185ca0 bib: adding SX and MX package description refs
| /
* 9464e81 Merge branch '80-populate-chapter-interfaces' into 'master'
```


An example git history: The `git rebase` in action

So when it comes to a `git rebase`, what you do is

- ① `git fetch origin master:master`: take a new photo of the plant in the shop to know what the current tip is (might be the same as on the previous photo) and `apply` this new master to the plant at home
- ② `git rebase master`: Now cut off the local branch from its initial root with all its leafs and transplant it on top of the new master

```
[staerz@staerz-dellt5820 doc (97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content)]$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Adding separation lines between template remarks and content, updating FW block diagram to v2.5
Applying: bib: Fixing 3 entries (missed comma)
Applying: Make the temporary chapter take number 0 to have the right numbers once it's removed
```

An example git history: The `git rebase` in action

So when it comes to a `git rebase`, what you do is

- ① `git fetch origin master:master:` take a new photo of the plant in the shop to know what the current tip is (might be the same as on the previous photo) and apply this new master to the plant at home
- ② `git rebase master:` Now cut off the local branch from its initial root with all its leafs and transplant it on top of the new master

Again, note the updated history: `remote` and `local` master have re-appeared

```
[staerz@staerz-dellt5820 doc (97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content)]$ git log
* 677c7f9 (HEAD -> 97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content) Make the temporary chapter take number 0 to have the right numbers once it's removed
* 2aba0c9 bib: Fixing 3 entries (missed comma)
* 12483d5 Adding separation lines between template remarks and content, updating FW block diagram to v2.5
* af8387f origin/master origin/HEAD master Merge branch '65-adding-lasp-documents' into 'master'
| \
| * 4ae182b (origin/65-adding-lasp-documents) functional diagram: moving from v2.1 to v2.2
```

An example git history: The `git rebase` in action

So when it comes to a `git rebase`, what you do is

- ① `git fetch origin master:master`: take a new photo of the plant in the shop to know what the current tip is (might be the same as on the previous photo) and apply this new master to the plant at home
- ② `git rebase master`: Now cut off the local branch from its initial root with all its leafs and transplant it on top of the new master

Now the critical part is to properly talk to the gardener

- ③ A simple `git push` is rejected: although the branch and leafs look the same, it has a **different root** (and different commit SHAs)

```
[staerz@staerz-dellt5820 doc (97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content)]$ git push
To ssh://gitlab.cern.ch:7999/atlas-lar-be-firmware/LASP/LASP-doc.git
! [rejected]        97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-c
ontent -> 97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content (non
-fast-forward)
error: failed to push some refs to 'ssh://git@gitlab.cern.ch:7999/atlas-lar-be-firmware/LASP/LASP-
doc.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

An example git history: The `git rebase` in action

So when it comes to a `git rebase`, what you do is

- ① `git fetch origin master:master`: take a new photo of the plant in the shop to know what the current tip is (might be the same as on the previous photo) and apply this new master to the plant at home
- ② `git rebase master`: Now cut off the local branch from its initial root with all its leafs and transplant it on top of the new master

Now the critical part is to properly talk to the gardener

- ③ A simple `git push` is rejected: although the branch and leafs look the same, it has a **different root** (and different commit SHAs)
- ④ A `git push -f` will force the gardener to discard his branch and re-grow your branch

```
Writing objects: 100% (29/29), 25.36 KiB | 3.62 MiB/s, done.
Total 29 (delta 22), reused 0 (delta 0)
remote:
remote: View merge request for 97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content:
remote:   https://gitlab.cern.ch/atlas-lar-be-firmware/LASP/LASP-doc/merge_requests/92
remote:
To ssh://gitlab.cern.ch:7999/atlas-lar-be-firmware/LASP/LASP-doc.git
 + f9f911a...677c7f9 97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content -> 97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content (forced update)
```

An example git history: The `git rebase` in action

So when it comes to a `git rebase`, what you do is

- 1 `git fetch origin master:master`: **take a new photo** of the plant in the shop to know what the current tip is (might be the same as on the previous photo) and **apply** this new master to the plant at home
- 2 `git rebase master`: Now cut off the local branch from its initial root with all its leafs and transplant it on top of the new master

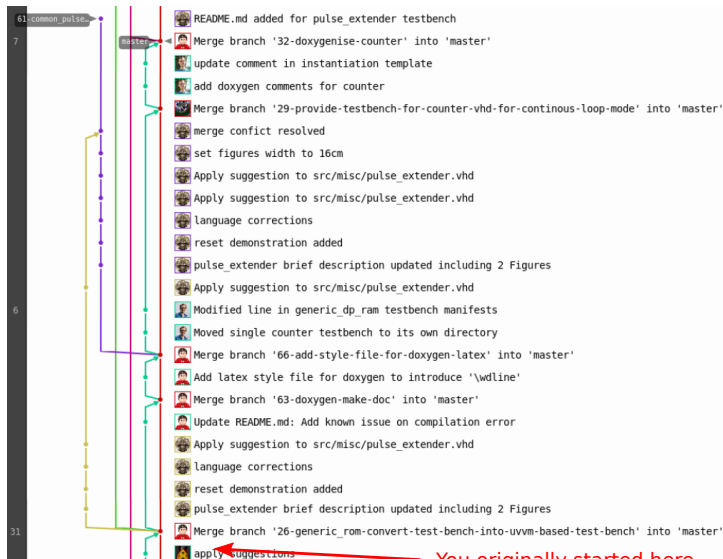
Now the critical part is to properly talk to the gardener

- 3 A simple `git push` is rejected: although the branch and leafs look the same, it has a **different root** (and different commit SHAs)
- 4 A `git push -f` will force the gardener to discard his branch and re-grow your branch

The same applies for any actions that involve `git rebase` in any kind:

As soon as "history is changed" forcing the gardener to accept it (`git push -f`) is required!

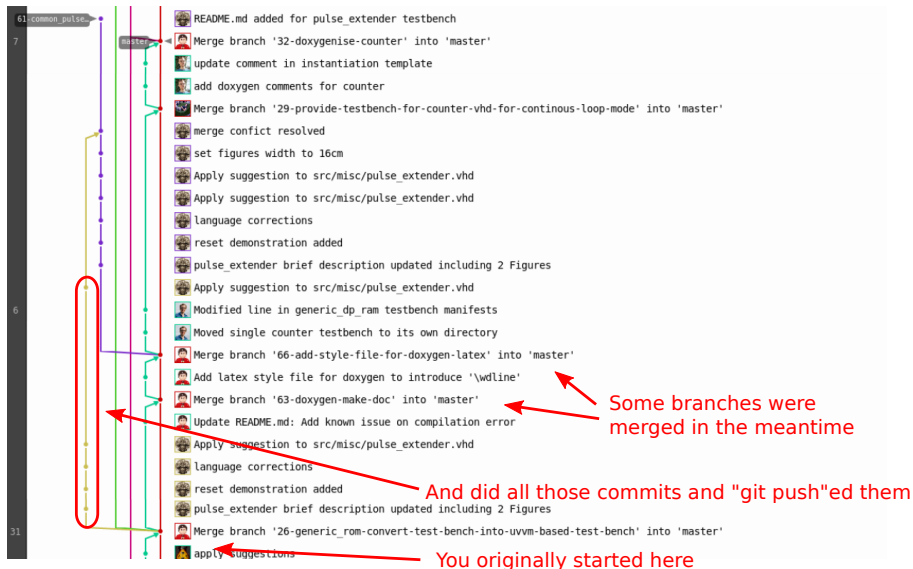
The common git rebase pitfall: some git history



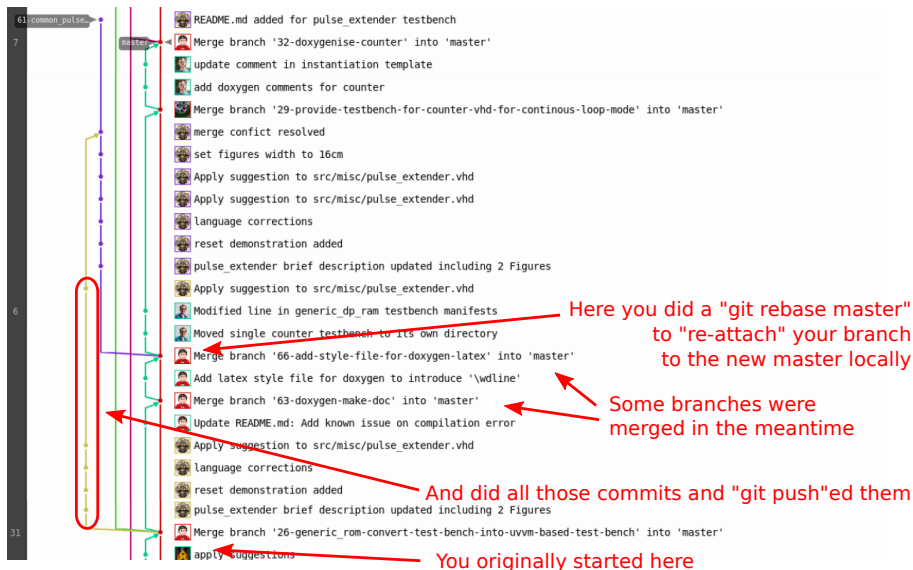
The common git rebase pitfall: some git history



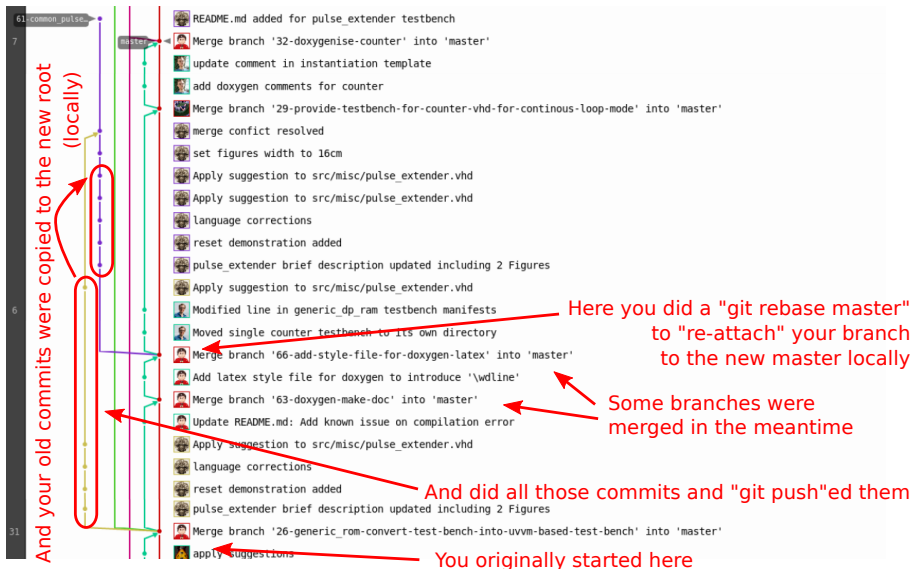
The common git rebase pitfall: some git history



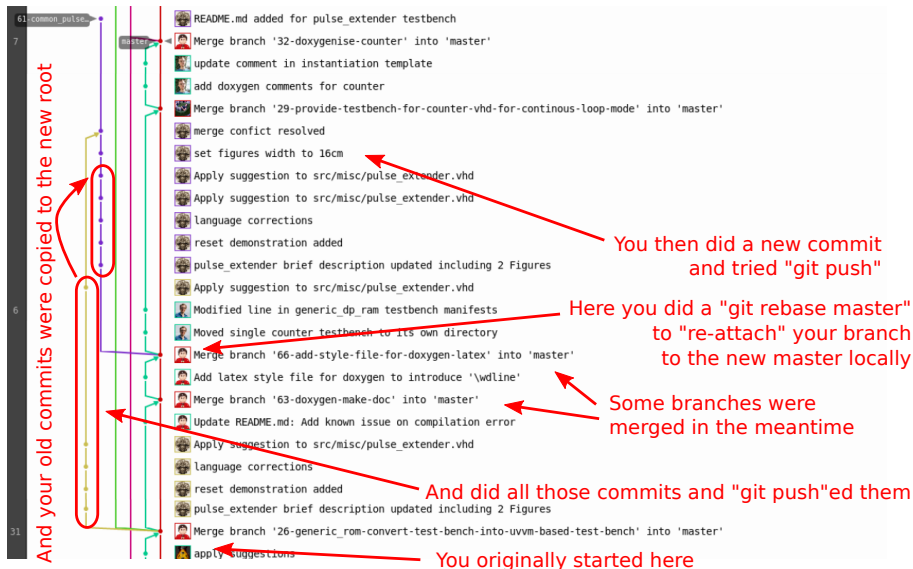
The common git rebase pitfall: some git history



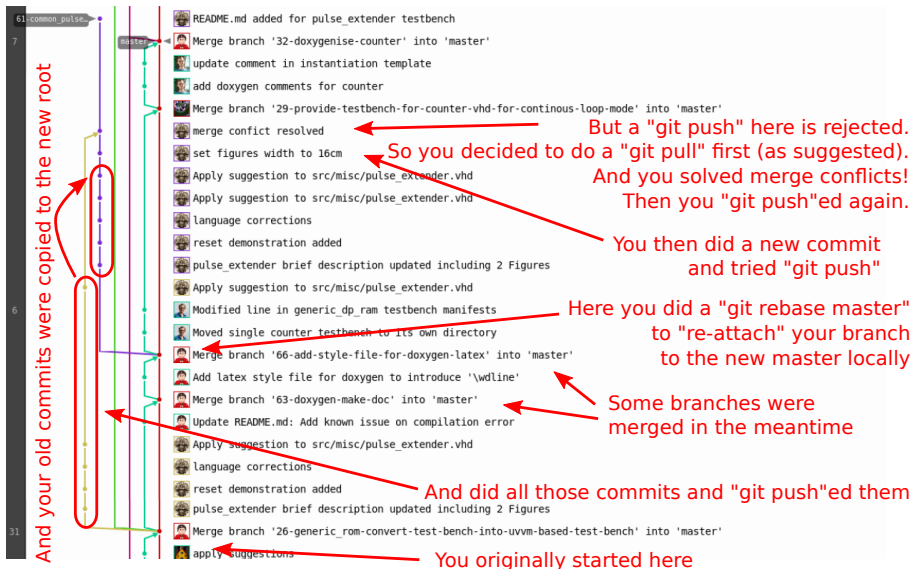
The common git rebase pitfall: some git history



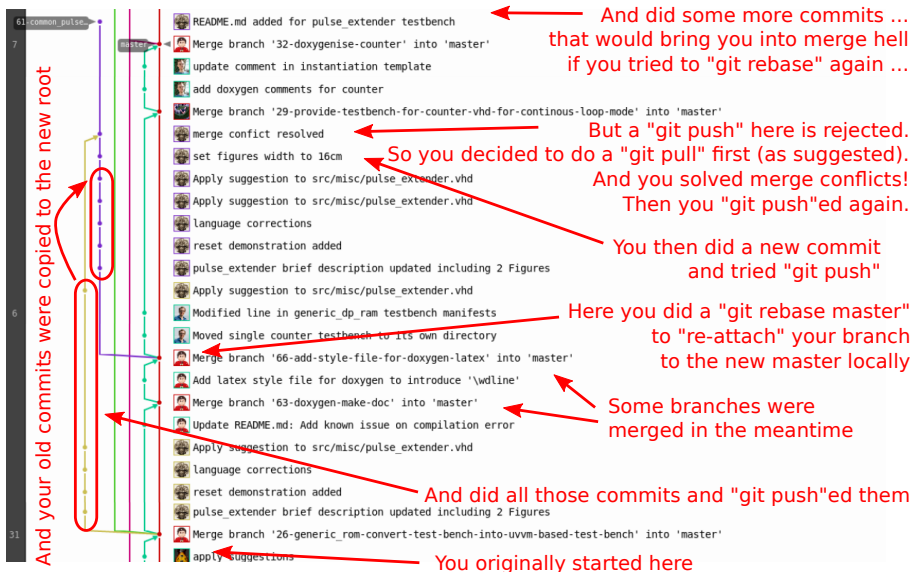
The common git rebase pitfall: some git history



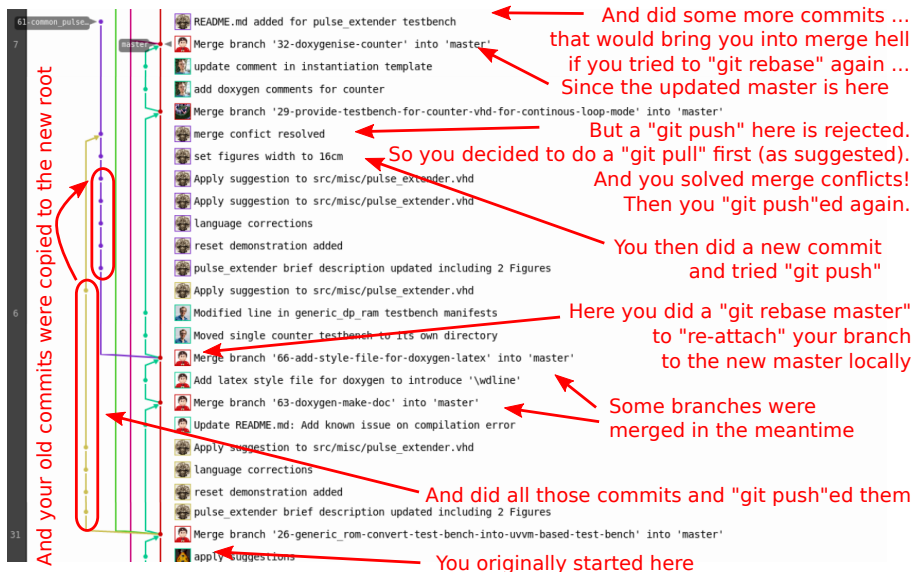
The common git rebase pitfall: some git history



The common git rebase pitfall: some git history



The common git rebase pitfall: some git history



The common git rebase pitfall: some git history



The common git rebase pitfall: some git history



The common git rebase pitfall: some git history



How to solve the pitfall once trapped

There's not only "the one option", but the general idea (being on your local "affected branch") is (to create a backup branch and then):

- ❶ Reset your branch to the one commit right before merging the remote and local branch ("set figures width to 16cm" in the example)
 - `git reset --hard <commit>`

How to solve the pitfall once trapped

There's not only "the one option", but the general idea (being on your local "affected branch") is (to create a backup branch and then):

- ❶ Reset your branch to the one commit right before merging the remote and local branch ("set figures width to 16cm" in the example)
 - `git reset --hard <commit>`
- ❷ Pick up (/transplant) the actual later commits
 - `git cherry-pick <commit>` (for single commits)
 - or `git rebase --onto ...` (for longer branches)

How to solve the pitfall once trapped

There's not only "the one option", but the general idea (being on your local "affected branch") is (to create a backup branch and then):

- ❶ Reset your branch to the one commit right before merging the remote and local branch ("set figures width to 16cm" in the example)
 - `git reset --hard <commit>`
- ❷ Pick up (/transplant) the actual later commits
 - `git cherry-pick <commit>` (for single commits)
 - or `git rebase --onto ...` (for longer branches)
- ❸ Force the remote to accept your new version of the branch
 - `git push -f`

How to solve the pitfall once trapped

There's not only "the one option", but the general idea (being on your local "affected branch") is (to create a backup branch and then):

- ❶ Reset your branch to the one commit right before merging the remote and local branch ("set figures width to 16cm" in the example)
 - `git reset --hard <commit>`
- ❷ Pick up (/transplant) the actual later commits
 - `git cherry-pick <commit>` (for single commits)
 - or `git rebase --onto ...` (for longer branches)
- ❸ Force the remote to accept your new version of the branch
 - `git push -f`
- ❹ Double-check that the result looks like expected in the graph view using the GitLab web interface

How to solve the pitfall once trapped

There's not only "the one option", but the general idea (being on your local "affected branch") is (to create a backup branch and then):

- ❶ Reset your branch to the one commit right before merging the remote and local branch ("set figures width to 16cm" in the example)
 - `git reset --hard <commit>`
- ❷ Pick up (/transplant) the actual later commits
 - `git cherry-pick <commit>` (for single commits)
 - or `git rebase --onto ...` (for longer branches)
- ❸ Force the remote to accept your new version of the branch
 - `git push -f`
- ❹ Double-check that the result looks like expected in the graph view using the GitLab web interface

So that got rid of the older branch (the yellow one in the example), but is not yet fully rebased onto master. Now you can (see as before):

- ❶ `git fetch origin master:master`
- ❷ `git rebase master`
- ❸ `git push -f`

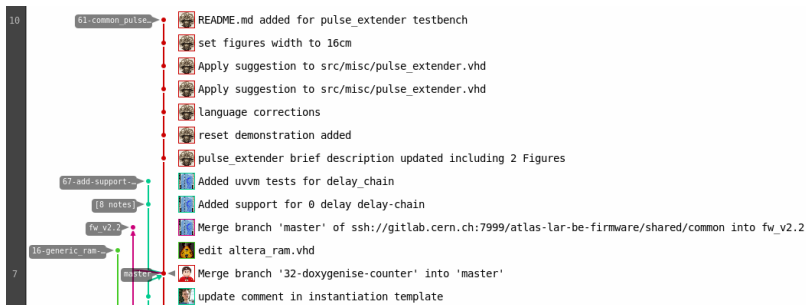
So once fixed

It should now look like that:



So once fixed

It should now look like that:



Note that:

- If you (or someone) rebased via the web interface, your local branch will be behind and you need to update it (rebasing **your local history**):
 - 1 `git pull -r` (**and not just `git pull`!**)
 - This is a pull with rebase: In case you have new local commits, they will be rebased on top of the new remote branch
- Without that, you run into the same `git pull` (merge) issue again

More git rebase features

Interactive rebase

Interactive rebasing is a powerful tool to alter the commit history and can:

- Reword the commit message
- Edit the commit (split, add files)
- Squash (meld into/combine) with other commit
- Drop (remove) commits
- Reorder the commit history

Note the difference to rebase: A "simple" rebase re-applies all the commits to a different root while an interactive rebase can change the entire history.

Interactive rebase

Interactive rebasing is a powerful tool to alter the commit history and can:

- Reword the commit message
- Edit the commit (split, add files)
- Squash (meld into/combine) with other commit
- Drop (remove) commits
- Reorder the commit history

Note the difference to rebase: A "simple" rebase re-applies all the commits to a different root while an interactive rebase can change the entire history.

Syntax: `git rebase -i <reference>`

- `<reference>` is any valid git reference, e.g. a tag, a branch, a commit or a relative reference like `HEAD~3`

Interactive rebase

Interactive rebasing is a powerful tool to alter the commit history and can:

- Reword the commit message
- Edit the commit (split, add files)
- Squash (meld into/combine) with other commit
- Drop (remove) commits
- Reorder the commit history

Note the difference to rebase: A "simple" rebase re-applies all the commits to a different root while an interactive rebase can change the entire history.

Syntax: `git rebase -i <reference>`

- `<reference>` is any valid git reference, e.g. a tag, a branch, a commit or a relative reference like `HEAD~3`

Attention: Never attempt to rebase commits that have already been merged into master (never go prior to/beyond master)

Interactive rebase: an example

Split a commit (which is not the most recent one*) into two

```
[staerz@staerz-dell:5820 doc (97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from
-actual-content)]$ git show 12483d5 --name-status
commit 12483d551fd0418f9c29c6338dc3be84cb2f5b
Author: Steffen Stärz <steffen.staerz@cern.ch>
Date: Mon Sep 9 13:09:04 2019 -0400

    Adding separation lines between template remarks and content, updating FW block diagram to v2.
5
M   fw_lasp/specs/text/CodeManagement.tex
M   fw_lasp/specs/text/DescriptionOfComponent.tex
M   fw_lasp/specs/text/FunctionalDescription.tex
M   fw_lasp/specs/text/InputOutput.tex
M   fw_lasp/specs/text/Interfaces.tex
M   fw_lasp/specs/text/Manufacturer.tex
M   fw_lasp/specs/text/PhysicalDescription.tex
M   fw_lasp/specs/text/Power.tex
M   fw_lasp/specs/text/RadiationTolerance.tex
M   fw_lasp/specs/text/RelatedDocuments.tex
M   fw_lasp/specs/text/Reliability.tex
M   fw_lasp/specs/text/Testing.tex
A   shared/figures/LASP_FW_diagram_FPGA_v2-5.pdf
```

Interactive rebase: an example

Split a commit (which is not the most recent one*) into two

```
[staerz@staerz-dell:~]$ git show 12483d5 --name-status
commit 12483d551fd0418f9c29c6338dc3be84cb2f5b
Author: Steffen Stärz <steffen.staerz@cern.ch>
Date:   Mon Sep 9 13:09:04 2019 -0400

    Adding separation lines between template remarks and content, updating FW block diagram to v2.
5
M   fw_lasp/specs/text/CodeManagement.tex
M   fw_lasp/specs/text/DescriptionOfComponent.tex
M   fw_lasp/specs/text/FunctionalDescription.tex
M   fw_lasp/specs/text/InputOutput.tex
M   fw_lasp/specs/text/Interfaces.tex
M   fw_lasp/specs/text/Manufacturer.tex
M   fw_lasp/specs/text/PhysicalDescription.tex
M   fw_lasp/specs/text/Power.tex
M   fw_lasp/specs/text/RadiationTolerance.tex
M   fw_lasp/specs/text/RelatedDocuments.tex
M   fw_lasp/specs/text/Reliability.tex
M   fw_lasp/specs/text/Testing.tex
A   shared/figures/LASP_FW_diagram_FPGA_v2-5.pdf
```

Let's separate out the pdf file into a dedicated commit **after** 'adding separation ...' (reference: 12483d5)

① `git rebase -i 12483d5~1`

* there are far simpler alternative ways in that case: `git commit --amend`

Interactive rebase: an example

Split a commit (which is not the most recent one*) into two

- Alter the line of the commit from `pick` to `edit`, save and exit

```
edit 12483d5 Adding separation lines between template remarks and content, updating FW block diagram
am to v2.5
pick 2aba0c9 bib: Fixing 3 entries (missed comma)
pick 677c7f9 Make the temporary chapter take number 0 to have the right numbers once it's removed

# Rebase af8387f..677c7f9 onto af8387f (3 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
~

1,1 All
```

Note that git will open the editor that you configured - here it's simply vim

Interactive rebase: an example

Split a commit (which is not the most recent one*) into two

④ Notice the git output

```
[staerz@staerz-dellt5820 doc (97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content)]$ git rebase -i 12483d5~1
Stopped at 12483d5... Adding separation lines between template remarks and content, updating FW b
lock diagram to v2.5
You can amend the commit now, with

    git commit --amend

Once you are satisfied with your changes, run

    git rebase --continue
[staerz@staerz-dellt5820 doc ((no branch, rebasing 97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content))]$
```


Interactive rebase: an example

Split a commit (which is not the most recent one*) into two

- 5 Reset the **now** top commit by `git reset HEAD~`

```
[staerz@staerz-dellt5820 doc ((no branch, rebasing 97-add-rule-line-for-all-chapters-to-separate-  
ff-remark-header-from-actual-content))] $ git reset HEAD~
```

```
Unstaged changes after reset:
```

```
M    fw_lasp/specs/text/CodeManagement.tex  
M    fw_lasp/specs/text/DescriptionOfComponent.tex  
M    fw_lasp/specs/text/FunctionalDescription.tex  
M    fw_lasp/specs/text/InputOutput.tex  
M    fw_lasp/specs/text/Interfaces.tex  
M    fw_lasp/specs/text/Manufacturer.tex  
M    fw_lasp/specs/text/PhysicalDescription.tex  
M    fw_lasp/specs/text/Power.tex  
M    fw_lasp/specs/text/RadiationTolerance.tex  
M    fw_lasp/specs/text/RelatedDocuments.tex  
M    fw_lasp/specs/text/Reliability.tex  
M    fw_lasp/specs/text/Testing.tex
```

Interactive rebase: an example

Split a commit (which is not the most recent one*) into two

⑥ Add files individually and commit step by step

```
[staerz@staerz-dellt5820 doc ((no branch, rebasing 97-add-rule-line-for-all-chapters-to-separate-
ff-remark-header-from-actual-content))] $ git add -u
[staerz@staerz-dellt5820 doc ((no branch, rebasing 97-add-rule-line-for-all-chapters-to-separate-
ff-remark-header-from-actual-content))] $ git commit -m "Adding separation lines between template r
emarks and content"
[detached HEAD b9f7fdb] Adding separation lines between template remarks and content
12 files changed, 30 insertions(+), 11 deletions(-)
[staerz@staerz-dellt5820 doc ((no branch, rebasing 97-add-rule-line-for-all-chapters-to-separate-
ff-remark-header-from-actual-content))] $ git add shared/figures/LASP_FW_diagram_FPGA_v2-5.pdf
The following paths are ignored by one of your .gitignore files:
shared/figures/LASP_FW_diagram_FPGA_v2-5.pdf
Use -f if you really want to add them.
[staerz@staerz-dellt5820 doc ((no branch, rebasing 97-add-rule-line-for-all-chapters-to-separate-
ff-remark-header-from-actual-content))] $ git add shared/figures/LASP_FW_diagram_FPGA_v2-5.pdf -f
[staerz@staerz-dellt5820 doc ((no branch, rebasing 97-add-rule-line-for-all-chapters-to-separate-
ff-remark-header-from-actual-content))] $ git add -u
[staerz@staerz-dellt5820 doc ((no branch, rebasing 97-add-rule-line-for-all-chapters-to-separate-
ff-remark-header-from-actual-content))] $ git commit -m "updating FW block diagram to v2.5"
[detached HEAD 486cb45] updating FW block diagram to v2.5
2 files changed, 1 insertion(+), 1 deletion(-)
create mode 100644 shared/figures/LASP_FW_diagram_FPGA_v2-5.pdf
```

Interactive rebase: an example

Split a commit (which is not the most recent one*) into two

7 Continue rebasing by `git rebase --continue`

```
[staerz@staerz-dellt5820 doc ((no branch, rebasing 97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content))$ gitlog -n 4
* 486cb45 (HEAD) updating FW block diagram to v2.5
* b9f7fdb Adding separation lines between template remarks and content
* af8387f Merge branch '65-adding-lasp-documents' into 'master'
|\
| * 4ae182b (origin/65-adding-lasp-documents) functional diagram: moving from v2.1 to v2.2
[staerz@staerz-dellt5820 doc ((no branch, rebasing 97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content))$ git rebase --continue
Successfully rebased and updated refs/heads/97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content.
[staerz@staerz-dellt5820 doc (97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content)]$ gitlog -n 4
* 4a0d8ab (HEAD -> 97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content) Make the temporary chapter take number 0 to have the right numbers once it's removed
* 3a8fd79 bib: Fixing 3 entries (missed comma)
* 486cb45 updating FW block diagram to v2.5
* b9f7fdb Adding separation lines between template remarks and content
```

Interactive rebase: an example

Split a commit (which is not the most recent one*) into two

8 Conclude by pushing your changes (`git push -f`)

```
[staerz@staerz-dellt5820 doc (97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content)]$ git push -f
Counting objects: 35, done.
Delta compression using up to 16 threads.
Compressing objects: 100% (35/35), done.
Writing objects: 100% (35/35), 25.77 KiB | 3.68 MiB/s, done.
Total 35 (delta 26), reused 0 (delta 0)
remote:
remote: View merge request for 97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content:
remote:  https://gitlab.cern.ch/atlas-lar-be-firmware/LASP/LASP-doc/merge_requests/92
remote:
To ssh://gitlab.cern.ch:7999/atlas-lar-be-firmware/LASP/LASP-doc.git
 + 677c7f9...4a0d8ab 97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content -> 97-add-rule-line-for-all-chapters-to-separate-off-remark-header-from-actual-content (forced update)
```

Interactive rebase: another example

Squash multiple sections of the history

Similar procedure as for splitting:

- ① `git rebase -i <reference>`
 - e.g. `git rebase -i HEAD~10*` (most recent 10 commits)
or `git rebase -i master` (all commits since master)
- ② Select `squash` (allows to alter commit message) or `fixup` (meld with previous commit and use its commit message) for commits to be combined
- ③ Follow instructions on terminal:
 - Rebase without problem as long as order is kept
 - Conflicts may emerge when reordering dependent commits. Solve as usual and `git rebase --continue`, or abort by `git rebase --abort`
- ④ `git push -f` to conclude

* Attention, pit fall: Make sure not to go beyond `master`. If it happens though, run `git rebase master` right after the interactive rebase to remove duplicate commits.

Rebase: general remarks

A matter of habit

- It's a matter of practice!
- Clean commits ease rebasing
- Use squashing for fairly long history and cleaning up commit messages

A matter of work flow (and repo settings)

- Before merging, branches could require rebase to master by default
- Maintainers can request developers to do the rebasing of their branches

There is always an alternative

- There is almost always another way to achieve the same goal with git.
- Once you understand what's going on, chose your preferred way.

Fixing conflicts via a simple text editor can be tedious and error-prone.



Summary Part II: Git Rebase

Git rebase changes the history

- `git rebase ...` "transplants" branches to a new root commit (e.g. on a newer commit on master)
- A regular `git push` will be rejected (if the branch was pushed before with its old root)
- Use a `git push -f` to force the update
- Be vigilant in comparing remote and local histories
- Use `git fetch` to notice remote updates
- Be ready to also update your local branch if the remote was rebased

Change your habits

- A `git pull` (that implies `git merge`) is the native enemy of `git rebase`: use `git fetch` instead to see first, then `git pull -r` to sync with remote

Part III: Beyond Git: GitLab/GitHub/...

Issue-based work flow

Disclaimer: GitLab/GitHub constantly being developed!

Here only basic features are mentioned as a starting point for your own further investigation:
there's too much to cover here and things may easily change

Issues and merge requests (pull requests)

Make use of the issue-based work flow!

- ① Create an issue: Be brief, but as verbose as necessary in the issue description to explain what needs to be done and what the context is (possibly fill in a provided template)
- ② Title the issue solution oriented ("This is broken" is a bad example)
- ③ Work on related branch and push regularly
- ④ Possibly incorporate any feedback from a review
- ⑤ Merge via merge (pull) request (which closes issue and deletes branch)

Documentation: Markdown files (*.md)

Documentation is always underrated and underestimated

Guideline of good documentation

- A `README.md` is always rendered directly when found in a directory
- Document purpose of the repository and its context
- Introduce content of repo and its functions (overview...)
- Point to details where appropriate
- Do not duplicate information
- Ideally your code documentation is already in the code (use doxygen, sphinx, ...)

If you don't like Markdown, use something else to document your code/repo, but document it!

CODEOWNERS

When working in a project with others:

CODEOWNERS for collaborative work

CODEOWNERS are used to identify users that are responsible for certain files in a Git repository.

- Owner(s) per file, directory, or wildcards
- ⇒ Code owners should be defined for a project with clear responsibilities
- ⇒ Can affect required approval of merge request / pull request ("Don't touch my code without my knowledge")
- ⇒ Improves code quality

CI/CD⁴: Pipelines and jobs

Pipelines are meant to automate as much as possible the work flow and assure a constantly high quality of the code base (repository).

They can be used to do **anything** as long as it can be scripted, e.g.:

- Run code syntax checks (spelling, style, ...)
- Compile and run some code (→ unit tests)
- Deploy software somewhere
- Produce documentation (latex, doxygen, ...)
- Automate repetitive tasks on the repository itself (merging, tagging, cleanup, etc.)

Setting up a CI/CD for a project is a task of its own, just a few notes:

- Pipelines are usually triggered for each single commit pushed
- Pipeline jobs can be tied to conditions → quickly getting complex

⁴CI: Continuous Integration, CD: Continuous Deployment

Summary Part III: Beyond Git: GitLab/GitHub/...

Use provided features to improve the project and simplify your work flow

- Use the issue-based work flow to keep track of your development!
- Document your code and repository!
- Code Owners allow to identify responsible experts
- Pipelines (CI/CD) assure quality and automate otherwise tedious tasks

Finally: The last slide

You made it!



- ✓ Git Basics
- ✓ Git Rebase
- ✓ Beyond Git: GitLab/GitHub/...



Thanks for your attention!
Your feedback is welcome at any time!
Questions?

Good Luck!