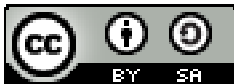
Source Code Management wih git

Matthieu Herrb

November 2023

https://homepages.laas.fr/matthieu/cours/pi2-git.pdf

License



This work is licensed under a *Creative Commons Attribution-ShareAlike 3.0 Unported* License.

To get a copy of the license, use the following address:

http://creativecommons.org/licenses/by-sa/3.0/

Agenda

1 Introduction – VCS and Git concepts
2 Individual developer
3 Using branches
4 Advanced branching
5 Good Practices
6 Other goodies
7 Working in teams
8 Git work flows
9 Webography

Agenda

1 Introduction – VCS and Git concepts
2 Individual developer
3 Using branches
4 Advanced branching
5 Good Practices
6 Other goodies
7 Working in teams
8 Git work flows
9 Webography

What is a version control system ?

Software that manages the history of changes in a set of documents.

Typically: source code

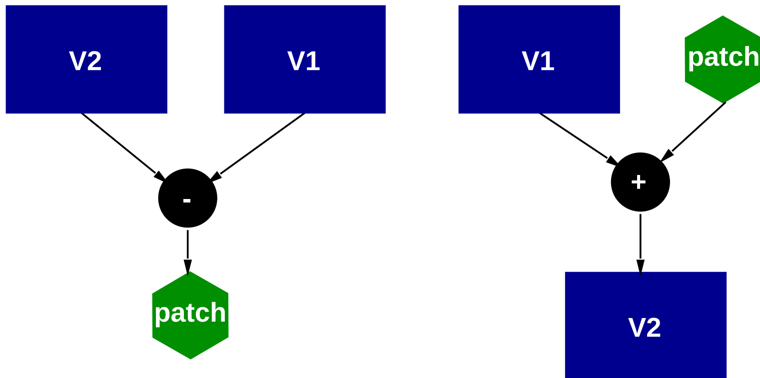
But also:

- documentation
- web sites
- configuration files
- etc.

Basic functionalities

- keep an history of changes
- make it possible to work in teams
- allow parallel work
- provide security (integrity, availability, confidentiality)

Diff & patch



Text diff

A **patch** represents the changes between 2 versions of a text file.

`diff -u fileA fileB` produces a patch in *unified diff* format ($b - a$):

```
--- a/src/server.c
+++ b/src/server.c
@@ -222,7 +222,9 @@ reset_log(void)
 #ifdef HAVE_SS_LEN
 #define sockaddr_len(s) s.ss_len
 #else
-#define sockaddr_len(s) sizeof(s)
+#define sockaddr_len(s) (s.ss_family == AF_INET6 ? \
+                     sizeof(struct sockaddr_in6) \
+                     : sizeof(struct sockaddr_in))
 #endif

 void
```

The `patch` command

`patch` applies a patch to a file to produce the new version ($b = a + \texttt{diff}$)

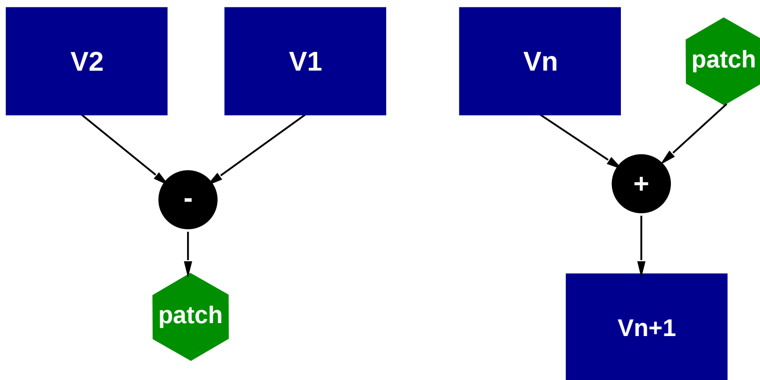
Example:

```
# patch -p1 -E < diff
```

`patch` can handle small inconsistencies, thanks to the context.

Merge

Applying a patch to a slightly different version



May fail → **conflict**

About file formats

text source code or equivalent. File content is a sequence of ASCII or UTF-8 characters, structured in lines with `newline` characters.
Examples: `.c`, `.h`, `.rs`, `.txt`, `.md`, `.conf`, `.ini`, `.xml`, `.json`, `.svg`...

binary the file format has a structure controlled by the application that creates / opens it.
Examples: multimedia files (`.jpg`, `.png`, `.mp3`, `.webm`, `.mp4`...), office documents (`.docx`, `.xlsx`, `.odt`, `.pdf`,...) executable macine code (`.o`, `.so`, `.exe`,...)

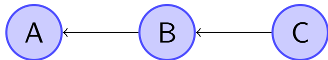
The text-based **diff** and **patch** commands only make sense on **text** files.
Git can handle binary files, but less efficiently, and no contents history will be available.

GIT

- *Distributed* version control system
- by opposition to CVS or SVN which are *Centralized*
- Developed by Linus Torvalds for the Linux kernel
- Similar to Monotone, Darcs, Mercurial, Bazaar, etc.

Git concepts (1)

- **Repository** Storage area that keeps the history of modifications.
- **Revision** Unique identifier of each state of the source files
  Also called **commit** as language shortcut.

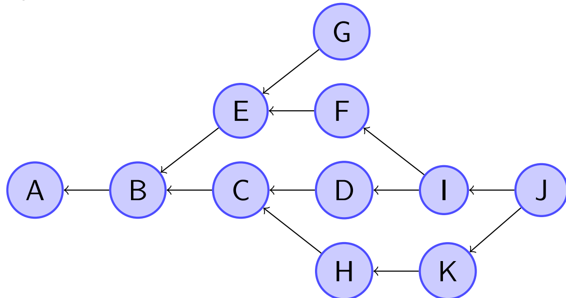
  Ordered sequence : arrows point to the ancestor. Represented by a 128 hexadecimal SHA-1 hash.

  $\rightarrow$ **Marketing project version** $\neq$ VCS revision !

- **commit (verb)** action to register a version of a set of files
  to the repository.
- **tag** a symbolic identifier for a commit or a branch

Git concepts (2)

- **Branch** one line of development.
  by default all development is done in **main**.



Uses of branches

Branches can be used to :

- fix a bug in an released version
- develop new ideas in parallel
- manage a customized version of the software
- merge back a version that diverged for some reason
- track local modifications to externally maintained sources
- ...

About the default branch

Because `#BlackLiveMatters`, “**master**” was a bad choice for the name of the main branch.
This can be changed for all new repositories :

```
git config --global init.defaultBranch main
```

Other possible names for the default branch :

- trunk
- development

References :

- git 2.28
- gitlab
- github

Git concepts (3)

Working tree the set of files being worked on currently.

Index an object tracking modified, added, removed files.

Blob binary data used to store files, objects, and other data

Git User Interfaces

- Command line
- Git GUIs
  - gitk (part of git distribution)
  - gitg (Gnome project)
  - git-cola `https://git-cola.github.io/`
  - TortoiseGit (Windows) `https://tortoisegit.org/`
- Editor plugins
  - Atom (built-in)
  - Visual Studio Code (built-in)
  - Eclipse (`https://eclipse.org/egit/`)
  - Emacs (VC, magit,...)
- Web browsers: `cgit`, `gitweb`.

Git forges

Web sites dedicated to git projects hosting.

- github `https://github.com/`
- gitlab `https://gitlab.com/`
- gogs `https://gogs.io/` / gitea `https://gitea.io/`
- redmine with the git plugin

Include interesting features for collaboration.

Better suited for distributed development than traditional centralized forges

Agenda

1. Introduction – VCS and Git concepts
2. Individual developer
3. Using branches
4. Advanced branching
5. Good Practices
6. Other goodies
7. Working in teams
8. Git work flows
9. Webography

Initial setup

Sets defaults for commit messages:

- user name & email
- preferred text editor

```
$ git config --global --add user.name "Matthieu Herrb"
$ git config --global --add user.email "<matthieu.herrb@laas.fr>"
$ git config --global --add core.editor emacs -nw
$ git config --global --add init.defaultBranch main
```

```
$ cat ~/.gitconfig
[user]
        name = Matthieu Herrb
        email = <matthieu.herrb@laas.fr>
[core]
        editor = emacs -nw
[init]
        defaultBranch = main
```

Creating a repository

`git init` creates an empty repository in the current directory.

```
$ mkdir git-tutorial
$ cd git-tutorial
$ git init
Initialized empty Git repository in /home/mh/git-tuturial/.git/
$ ls -l .git
total 24
-rw-r--r-- 1 mh mh  23 Oct 26 09:14 HEAD
-rw-r--r-- 1 mh mh 111 Oct 26 09:14 config
-rw-r--r-- 1 mh mh  58 Oct 26 09:14 description
drwxr-xr-x 12 mh mh 408 Oct 26 09:14 hooks
drwxr-xr-x 3 mh mh 102 Oct 26 09:14 info
drwxr-xr-x 4 mh mh 136 Oct 26 09:14 objects
drwxr-xr-x 4 mh mh 136 Oct 26 09:14 refs
```

Adding files

`git add` adds new or modified files to the index.

```
$ echo "Hello World" > file.txt
$ git add file.txt
```

Querying status

Shows the status of the repository and the index.

```
$ git status
# On branch main
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file: file.txt
#
```

Committing changes

```
$ git commit
Created initial commit 0ba7bd8: Initial version
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 file.txt
```

Opens a text editor to enter a commit message
and commits the change to the repository.

```
$ git status
On branch main
nothing to commit, working tree clean
```

More changes

```
$ echo "Hello Matthieu" > file.txt
```

Add changes and commit in one command (not recommended) :

```
$ git commit -a
Created commit 7fbf4cb: Modif
 1 files changed, 1 insertions(+), 1 deletions(-)
```

The git index

Represents modifications pending commit.
2 stages:

1. add modified files to the index (`add`,`rm`)
2. "flush" the index to the repository (`commit`)

Short-cuts chaining both operations:

- `git commit` *`file`*
- `git commit` *`dir`* (or `git commit .`)
- `git commit -a`

Interactive add

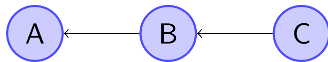
```
$ git add -p [files]
```

Enters an interactive session to pick up changes to be added to the *index*.

Allows to have several unrelated un-committed modifications, and still do clean, separate commits.

Commits

Adds a node at the end of the current branch.

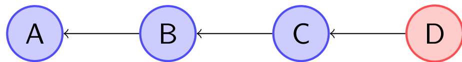


Includes:

- the patch from old to new revision for text files
- the full new revision for binary files
- attributes of the committed file (access modes)
- name and e-mail address of the committer
- a log message
- optionally, a digital signature

Commits

Adds a node at the end of the current branch.



Includes:

- the patch from old to new revision for text files
- the full new revision for binary files
- attributes of the committed file (access modes)
- name and e-mail address of the committer
- a log message
- optionally, a digital signature

Looking back: git log

Various ways to display the history of modifications.

```
$ git log
commit 7fbf4cb7c8977061fbfb609016f5414e833a3a1c
Author: Matthieu Herrb <matthieu.herrb@laas.fr>
Date:   Tue Oct 28 12:29:33 2014 +0100

    Modif

commit 0ba7bd8b93ef9ddd8917814bde8cbdaaf9732559
Author: Matthieu Herrb <matthieu.herrb@laas.fr>
Date:   Tue Oct 28 12:28:38 2014 +0100

    Initial version
$ git log --stat
$ git log -p
```

Examining changes: git diff

Display the changes between the working files and the index,
or between the index and the repository.

```
echo "Good bye" > file.txt
$ git diff
diff --git a/file.txt b/file.txt
index 6bd8f3c..c0ee9ab 100644
--- a/file.txt
+++ b/file.txt
@@ -1 +1 @@
-Hello Matthieu
+Good bye
$ git add file.txt
$ git diff --cached
```

Marking a version: git tag

Create a tag object, containing a name and a comment.
Opens the text editor to enter the comment.

```
$ git tag -a git-tuto-1.0
$ git tag -l
git-tuto-1.0
```

Fixing mistakes, reverting to a good version

Revert a given commit

```
$ git revert 03bace
Finished one revert.
Created commit c333ab5: Revert "3rd version"
 1 files changed, 1 insertions(+), 1 deletions(-)
```

creates a new commit that stores the revert action.

Restore the working dir to a given committed version, **loosing all local changes**:

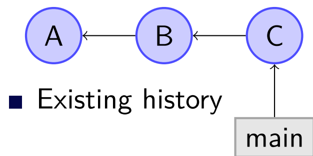
```
$ git reset --hard [commit-id]
```

If commit-id is missing, defaults to `HEAD`.

Agenda

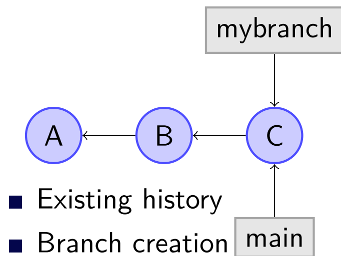
1 Introduction – VCS and Git concepts
2 Individual developer
3 Using branches
4 Advanced branching
5 Good Practices
6 Other goodies
7 Working in teams
8 Git work flows
9 Webography

Branches



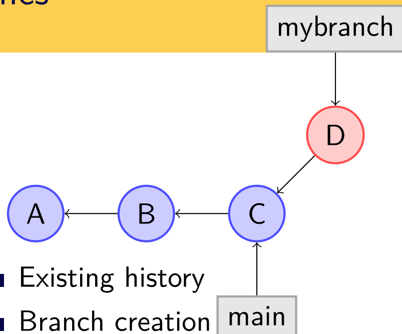
- Existing history

Branches



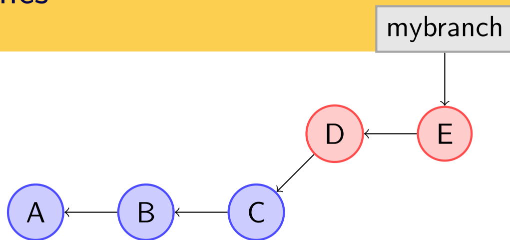
- Existing history
- Branch creation

Branches



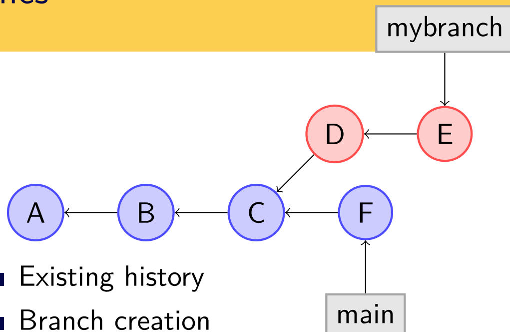
- Existing history
- Branch creation
- commits in the new branch

Branches



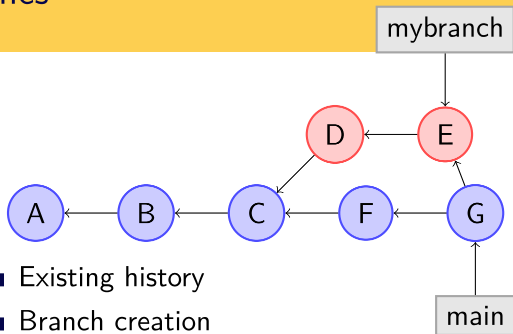
- Existing history
- Branch creation
- commits in the new branch

Branches



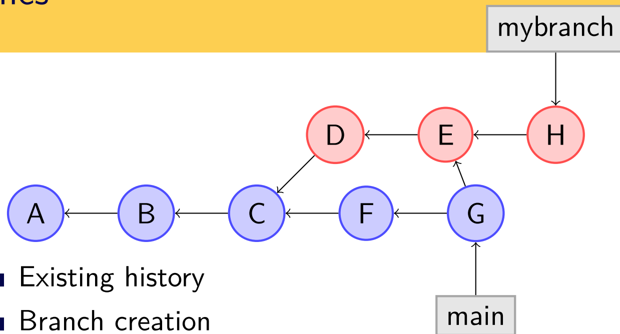
- Existing history
- Branch creation
- commits in the new branch
- commits in *main*

Branches



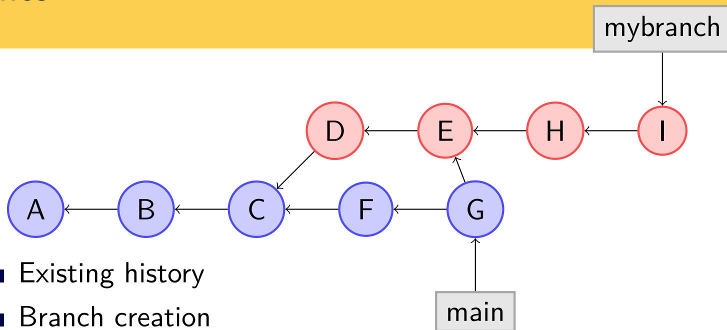
- Existing history
- Branch creation
- commits in the new branch
- commits in *main*
- merge the branch into *main*

Branches



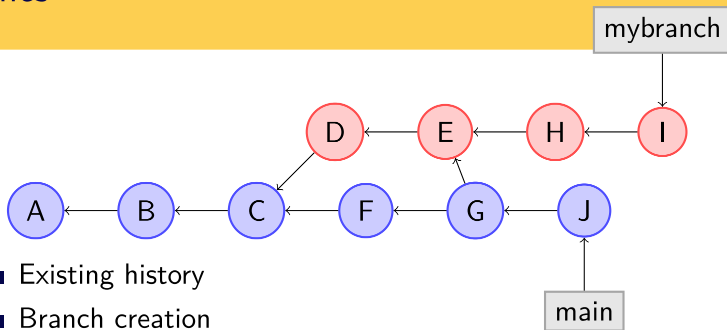
- Existing history
- Branch creation
- commits in the new branch
- commits in *main*
- merge the branch into *main*
- further commits in the branch

Branches



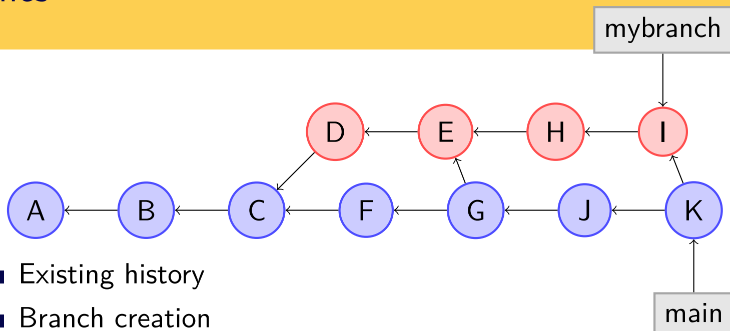
- Existing history
- Branch creation
- commits in the new branch
- commits in *main*
- merge the branch into *main*
- further commits in the branch
- etc...

Branches



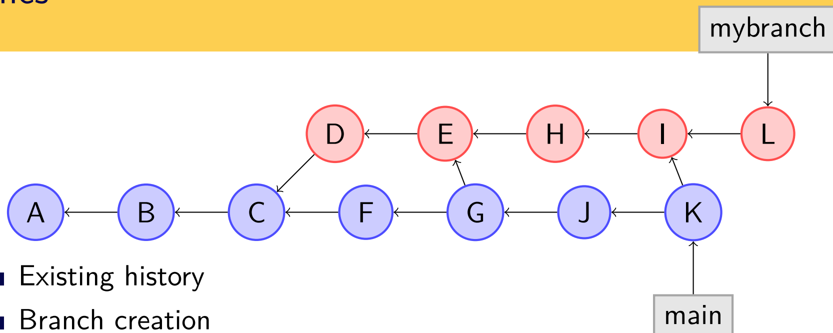
- Existing history
- Branch creation
- commits in the new branch
- commits in *main*
- merge the branch into *main*
- further commits in the branch
- etc...

Branches



- Existing history
- Branch creation
- commits in the new branch
- commits in *main*
- merge the branch into *main*
- further commits in the branch
- etc...

Branches



- Existing history
- Branch creation
- commits in the new branch
- commits in *main*
- merge the branch into *main*
- further commits in the branch
- etc...

Switching branches

Create a new branch:

```
$ git checkout -b newbranch
```

Switch back to main:

```
$ git checkout main
```

Listing available branches

```
$ git branch
* main
  newbranch
```

Merging changes from another branch

```
$ git merge branch
```

Merge commits from “branch” and commits the result.

2 kinds of merges:

- fast forward: no conflicts, only new commits to add to your version
- normal merge: there are local changes - use a 3 way merge algorithm.

Handling conflicts

Conflicts happen when changes in a merged branch are incompatible with changes in the target branch :

- Files with conflicts contain conflict markers
- They are not automatically added to the index.

To proceed :

- Resolve the conflict (ie choose the *correct* version)
- Add the manually merged files to the index
- Commit the result

Tools to help with merge

To solve conflicts git can use existing tools to help merging:
meld, xxdiff, opendiff, DiffMerge...

```
$ git config --global merge.tool meld
```

```
$ git mergetool
```

Agenda

1 Introduction – VCS and Git concepts

2 Individual developer

3 Using branches

4 **Advanced branching**

5 Good Practices

6 Other goodies

7 Working in teams

8 Git work flows

9 Webography

Stashing local changes

Git refuses to merge a branch if there are un-commited changes.

Solutions:

- commit local changes before merging...
- or `stash` local changes before merging.

```
$ git stash
$ git merge mybranch # or git pull --rebase
$ git stash pop
```

It's also possible to create a new branch with the stashed changes

```
$ git stash branch newbranch
```

Picking individual changes

Take one commit from another branch (bug fix)
and apply it to the working branch.

```
$ git cherry-pick SHA1_HASH
```

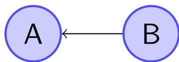
Replaying changes from a branch

Merges create lots of unwanted links in the git data graph.
When a branch has only few local commits, **rebase** is more efficient.

```
$ git rebase main
```

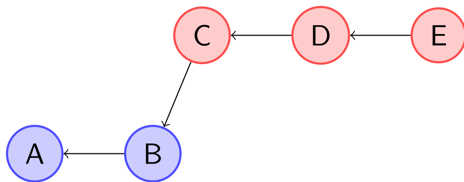
Only use rebase before pushing to a remote repository !

Rebase



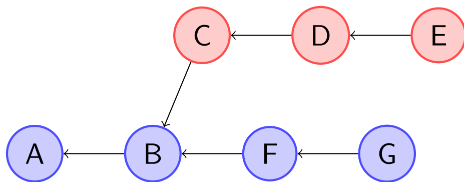
- Existing commits

Rebase



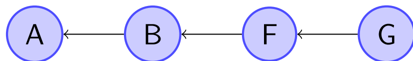
- Existing commits
- Development branch

Rebase



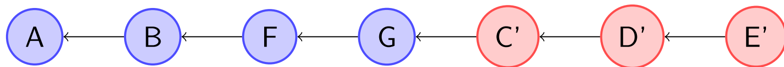
- Existing commits
- Development branch
- Commits in *main*

Rebase



- Existing commits
- Development branch
- Commits in *main*
- Start of rebase: remove commits from the branch

Rebase



- Existing commits
- Development branch
- Commits in *main*
- Start of rebase: remove commits from the branch
- End of rebase: recreate commits starting from *HEAD* of *main*

Interactive rebase

Useful to re-arrange commits locally, in order to clean up the history.

```
$ git rebase -i COMMITS
```

$\rightarrow$ opens a text editor with the list of commits specified.
Rearrange the list according to instructions and save it.
$\rightarrow$ history will be re-written, following the new list.

Only use rebase before pushing to a remote repository !

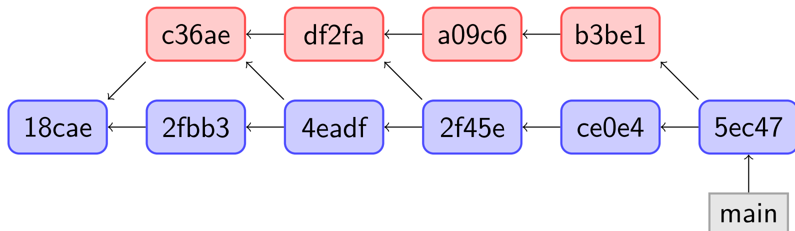
References "Tree-ish"

Alternative ways to refer to objects or ranges of objects

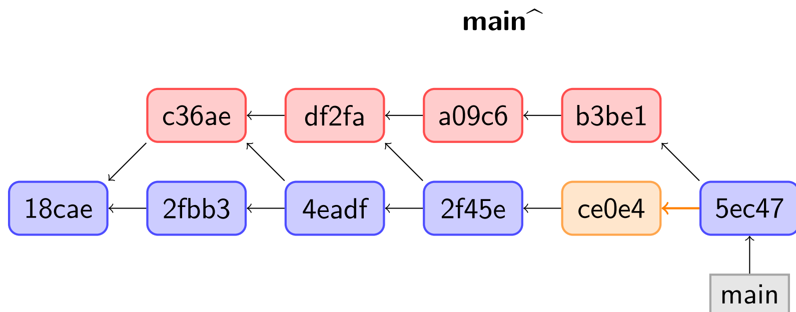
- full sha-1: `8f8aca4bd6c29048636966247aa582718559d871`
- partial sha-1: `8f8aca4b`, `8f8aca`, `8f8ac`,...
- branch or tag name: `v1.0`, `main`, `origin/testing`
- date spec `main@{yesterday}` `main@{1 month ago}`
- ordinal spec `main@{5}`
- carrot parent `main^2`
- tilde spec `main~2`
- tree pointer `main^{tree}`
- blob spec `main:/path/to/file`
- ranges `4c032a..8faca4`

See *gitrevisions(1)*

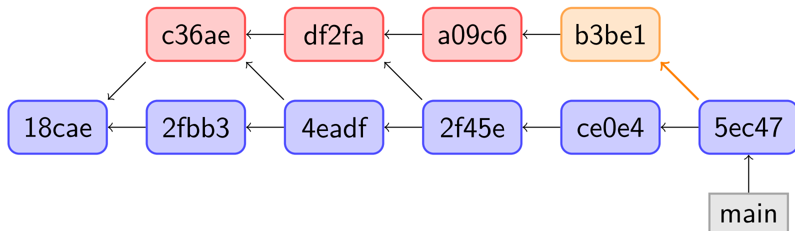
Examples



Examples

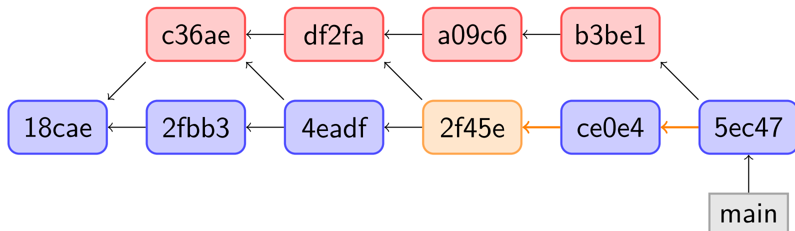


Examples



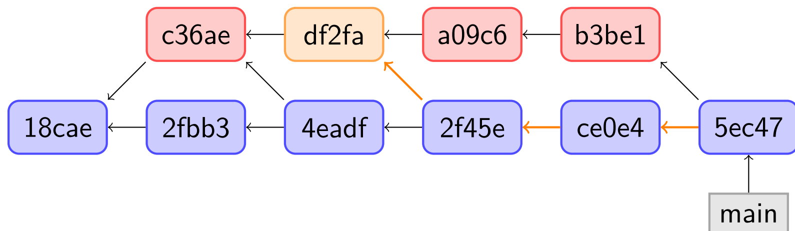
Examples

main~2



Examples

main~2^2 or **main^^^2**



Agenda

1. Introduction – VCS and Git concepts
2. Individual developer
3. Using branches
4. Advanced branching
5. Good Practices
6. Other goodies
7. Working in teams
8. Git work flows
9. Webography

Good practices - repositories

- Do not abuse git as a generic cloud storage
  - Git is good with text format files (source code, Markdown/LaTeX doccuments, ...)
  - Git is not useful on images, PDF or Office documents
  - Large binary files are wasting resources in a git repository
- Keep repositories on topic
  - One repository per project
  - Do not store un-related files
- Keep repository clean and minimal
  - do not add editor backup files
  - do not add files that can be generated automatically
  - do not add alternative versions as separate files. Use branches.

Good practices - commits

Good commits are hard, but add a lot of value to the project

- Always provide informative commit messages
- Follow the git commit convention :
  - one line summary
  - one empty line
  - detailled information below (multiple lines)
- Keep commits small and focused
  - Don't blindly add changes. Use `git status` and `git diff` to review them
  - Use `git add -p` to break-up large uncommited changes
- Commit early and commit often
- Use `amend` or `rebase` to fix broken commits before pushing

Good practices - branches

- Keep the number of active branches small
- Use self-documenting branches names
- Prefer `rebase` to `merge` whenever possible
- Try to keep a linear history on the main branch

Agenda

1. Introduction – VCS and Git concepts
2. Individual developer
3. Using branches
4. Advanced branching
5. Good Practices
6. **Other goodies**
7. Working in teams
8. Git work flows
9. Webography

Identifying authors

```
$ git blame -- file.txt
```

for each line of the file, shows the id and author of the last modification.

```
$ git shortlog --summary --numbered --email
```

list all authors, ordered by number of commits.

Making a release with git

(Alternative to automake's `make dist` or CMake's `CPack`)

- Commit all changes, including the new marketing version number in documentation.
- Tag the result with **`git tag -a`**
- Use **archive** to produce a release.

```
$ git tag -a foo-1.3
$ git archive --prefix=foo-1.3/ foo-1.3 \
  | gzip -c - > foo-1.3.tar.gz
```

Binary search of bugs

```
$ git bisect start
$ git bisect bad           # Current version is bad
$ git bisect good v2.6-rc2 # v2.6-rc2 was the last version
                           # tested that was good
...
$ git bisect reset         # back to initial state
```

With a script that can tell if the current code is good or bad:

```
$ git bisect run my_script arguments
```

- `my_script` returns 0 → good
- `my_script` returns 1..124 → bad

Reflog

reflog is the safety net of git.

- records all changes done in the repository
- keeps track of commits not otherwise accessible anymore
- allows to recover from some mistakes
- local only and expires after 90 days

Example:

```
$ git add foo.txt
$ git commit
$ git reset --hard <older version> # OOPS !
$ git reflog
$ git checkout HEAD@{n}
```

Sub-modules

Sub-modules provide a way to glue several existing repositories into a bigger project.

- **`git submodule add`** ***`url path`***
  adds a submodule, at *path*
- **`git submodule init`**
  init the sub-modules
- **`git submodule update`**
  clone or pull the submodules
- **`git submodule status`**
  display information about submodule status

git-lfs Large Files

Extension to support large binary files in git repositories

- uses external storage (cloud)
- replaces actual files with a link
- on checkout, fetch the real file from the external storage
- saves space in the repository
- but adds a dependency to an external service

https://git-lfs.github.com/

git-crypt - encrypt contents

Secrets (passwords, application keys,...) should not be stored in (public) git repositories

git-crypt provides a way to store encrypted contents with GPG or with simple shared keys

Create `.gitattributes`:

```
secretfile filter=git-crypt diff=git-crypt
*.key filter=git-crypt diff=git-crypt
```

Use:

```
    $ git-crypt init
    $ git-crypt add-gpg-user USER_ID
    $ git-crypt unlock
```

BFG Repo-Cleaner

Tool to clean up mistakes commited in a repository :

- remove Huge binary files (executables, images, video)
    - either commited by mistake
    - or removed voluntarily by `git rm` but still occupying space for nothing in the repository
- remove passwords or other kind of confidential data commited by mistake

Warning: this modifies the repository; need to inform all users before pushing

https://rtyley.github.io/bfg-repo-cleaner/

Agenda

1 Introduction – VCS and Git concepts

2 Individual developer

3 Using branches

4 Advanced branching

5 Good Practices

6 Other goodies

7 **Working in teams**

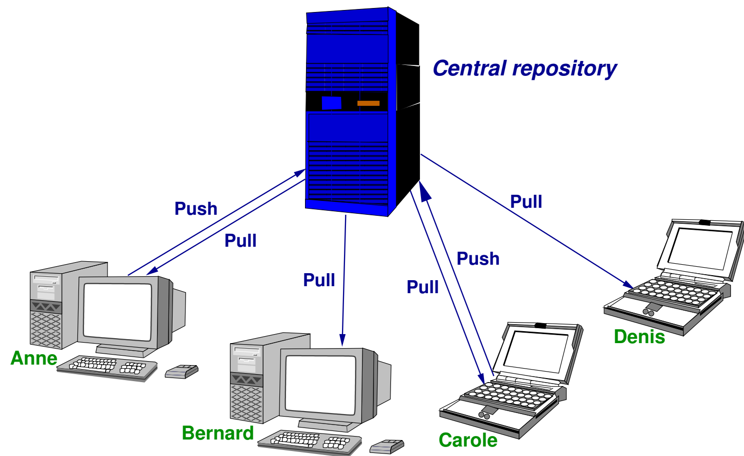
8 Git work flows

9 Webography

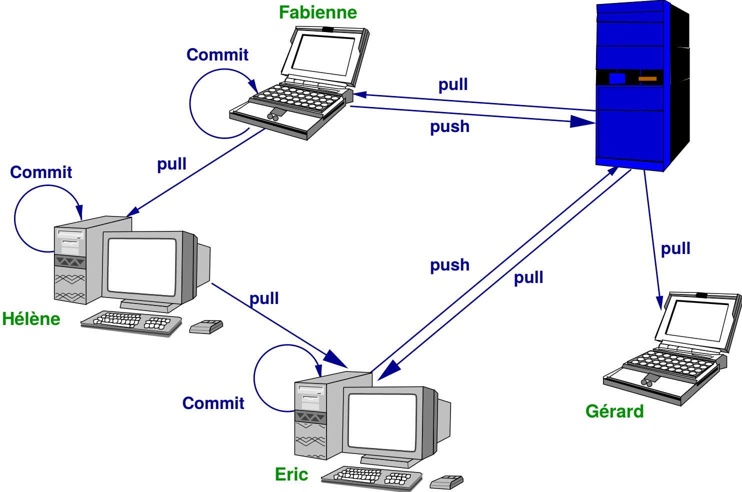
Working in teams

- No locks on source code.
  Each developer has its own copy of the source and repository.
- Conflicts handling:
  - First merge other people's contribution
  - Automated merges as much as possible
  - Conflict detection $\rightarrow$ manual resolution
  - No new `commit` before solving the conflict.

Centralized model



Semi-Distributed model



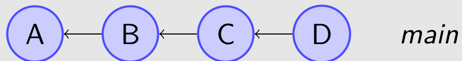
Copying a repository

```
git clone repo
```

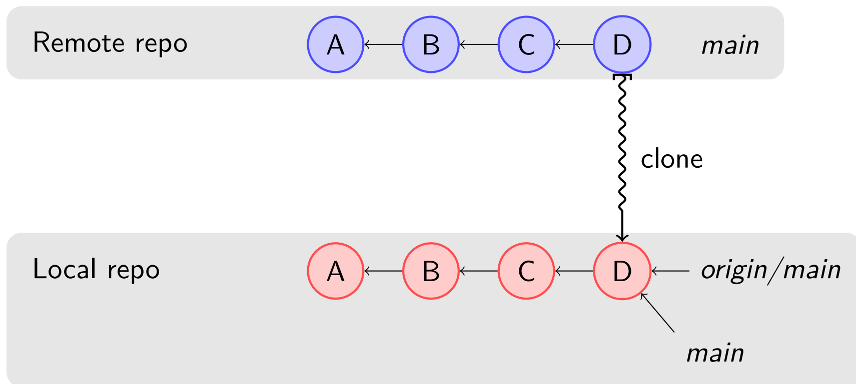
repo: an url to the remote repository. Can be:

- a pathname to a local repository on the same file-system
- `ssh://[user@]host/path` - use SSH with given user
- `ssh://git@host/path` - SSH with public key authentification (github, redmine)
- `https://host/path` - access with HTTPS protocol
- `git://host/path` - anonymous access with the GIT protocol

Remote repository



Remote repository



Updating from a remote repository

```
$ git pull
```

- Fetches the remote branches to the local repository,
- Merges the default remote branch into the current one.

- Can produce a conflict:
  - Solve the conflict
  - Commit the result

Using rebase with remote repositories

`git fetch` fetches remote commits without merging them.

Fetch and rebase at once

```
$ git pull --rebase
```

equivalent to:

```
$ git fetch
$ git rebase origin/main
```

Remote branches

```
$ git branch -r
```

lists remote branches (*origin/branch*).

Remote branches can be tracked (automatically merged/pushed) using:

```
$ git checkout -t -b newbranch origin/newbranch
```

Sending changes to a repository

```
$ git push
```

Sends local commits to remote tracked branches.
Produces an error if not up-to-date (need to pull or rebase first).

Tags need to be pushed separately:

```
$ git push --tags
```

Agenda

1 Introduction – VCS and Git concepts
2 Individual developer
3 Using branches
4 Advanced branching
5 Good Practices
6 Other goodies
7 Working in teams
8 **Git work flows**
9 Webography

Git work flows

Work flows that help maintaining a consistent central main branch.
Developers use *private* repositories.
Several models of workflow exist, using several tools :

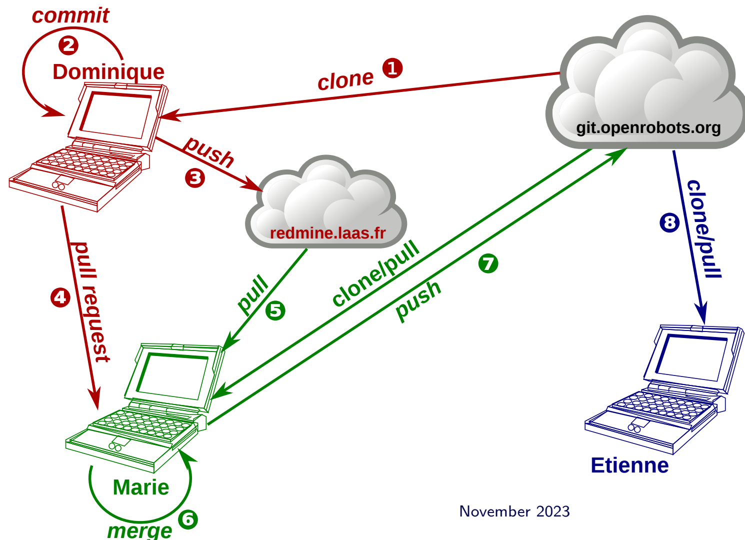
- *e-mail* : `git send-email`, `git am`
- Forge-provided tools (example: Github pull request)
- Direct access to the private repositories

In all cases, branches in the private repository allow to work on several changes until they are accepted.

Maintainer / pull request work flow

- Only the maintainer can push to main.
- Developers submit *pull requests* to the maintainer.
- The maintainer reviews, merges and then pushes the result.

Pull requests



Pull requests

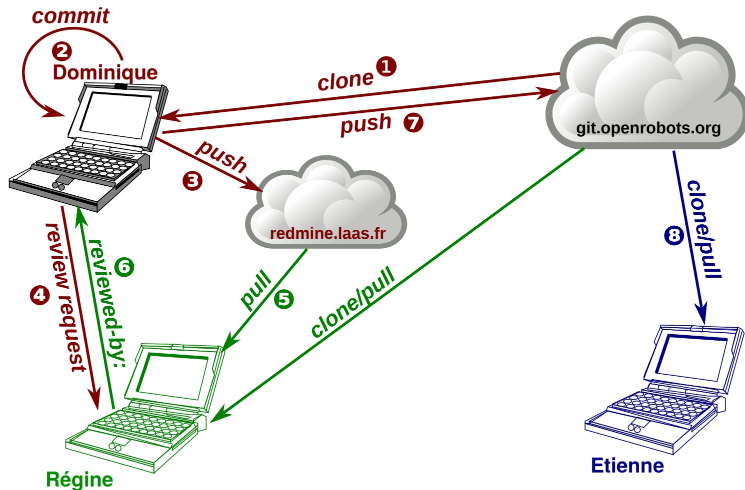
Marie is the maintainer, Dominique a developer and Etienne an end-user.

1. Dominique clones the repository,
2. Dominique works on the code, does some commit(s),
3. Dominique pushes his commits to his private repository,
4. Dominique sends a pull request to Marie,
5. Marie merges the pull request in her local repository,
6. Marie checks Dominique's work,
7. If Marie is happy with the result, pushes it to the main repository,
8. Etienne can grab the result.

Reviews work flow

- Push to main is open
  but no one can push to main without a review.
- Developers ask others for reviews.
- Reviewers reply with an '*OK*'
- Developer amends the commit message to add 'Reviewed-by:' headers and pushes the result.

Reviews



Reviews

Regine is a reviewer, Dominique a developer and Etienne the end-user

1. Dominique clones the repository,
2. Dominique works on the code, does some commit(s),
3. Dominique pushes his commits to his private repository,
4. Doninique sends a review request to the community,
5. Regine picks up the request,
6. Regine accepts the change and sends a reviewed-by message to Dominique,
7. Dominque amends his commit and pushes it to the main repository,
8. Etienne can grab the result.

Using email

For small changes (patches), using email to interact with reviewers/maintainers is easier/faster.

- configure `sendemail.smtpserver` and `sendemail.smtpuser`
- use `git format-patch` to generate the patches for the commits to submit for review / pull
- use `git send-email` to send an email containing the patches generated above
- the maintainer or reviewer can use `git am` to apply patches from his mail client

See The advantages of an email-driven git workflow for more information

Managing remote repositories

`git remote` ***`command`***

- **`add`** ***`name url`*** add a remote
- **`set-url`** ***`name url`*** changes the url
- **`rename`** ***`old new`*** renames
- **`rm`** ***`name`*** removes a remote

Pushing to multiple remote repositories

`git push` ***`remote branch`*** push a given branch to a given remote

Example:

```
$ git push origin main # same as git push
$ git push github mybranch # push branch to github
```

Good practices - distributed development

- Ask for reviews
- Review the patches that are sent to you
- Always use `pull --rebase` when possible before pushing
- Do not push experimental/test branches if not needed
- Be careful to push to the correct branch
- In case of a mistake, **communicate** with other developers

Agenda

1. Introduction – VCS and Git concepts
2. Individual developer
3. Using branches
4. Advanced branching
5. Good Practices
6. Other goodies
7. Working in teams
8. Git work flows
9. **Webography**

Webography

- https://imgs.xkcd.com/comics/git.png →
- https://git-scm.com/book/en/v2
  The online *Pro Git* book
- https://www.atlassian.com/git/tutorials
- http://learngitbranching.js.org/
  graphical tutorial on branches.
- Git for computer scientists, Tommi Virtanen, june 2007.
- Demystifying Git internals, Pawan Rawal, august 2016.
- Confusing Git terminology, Julia Evans, november 2023.